

# A High-Level Specification for Mediators (Virtual Providers)\*

Michael Altenhofen<sup>1</sup>, Egon Börger<sup>2</sup>, and Jens Lemcke<sup>1</sup>

<sup>1</sup> SAP Research, Karlsruhe, Germany

{michael.altenhofen, jens.lemcke}@sap.com

<sup>2</sup> Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy

boerger@di.unipi.it

On sabbatical leave at SAP Research, Karlsruhe, Germany

egon.boerger@sap.com

**Abstract.** We define a high-level model to mathematically capture the semantical meaning of abstract Virtual Providers (VP), their instantiation and their composition into rich mediator structures.

## 1 Introduction

Current work performed in the area of Web service composition identifies the need for communicating via a central, commonly shared vocabulary (like [8,9,6]). To establish such an agreed level of communication, approaches like MIBIA [4], WSMF [5] and WebTransact [7] envision the use of different types of mediators.

The goal of this paper is to abstractly model process mediation (Sects. 2, 3). In addition, we want to prepare the ground for using the strategy pattern for the run-time selection of mediators, where any suitable provider interface is viewed as one of the implementations (“mediator orchestration”) of a strategy pattern assigned to a requestor interface (see Fig. 1). Thus, one has to investigate the following questions:

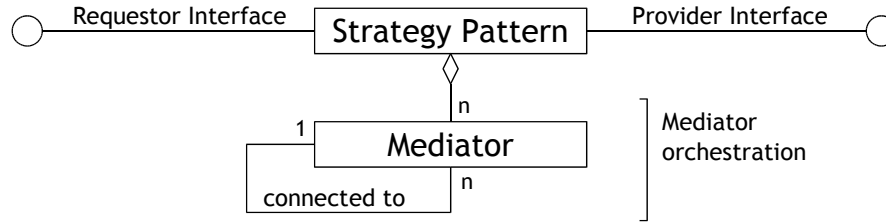
- How to assure that a provider interface matches the strategy pattern of the requester?
- How and starting from which information can one build automatically the strategy pattern implementations?

We use as modeling framework Abstract State Machines (ASM). An introduction into the ASM method for high-level system design and analysis is available in textbook form in [3], but most of what we use below to model VP constructs is self-explanatory.

For protocol mediation, we start with a simplified model of interaction patterns where single requests from the requestor side are being responded by a single answer from the Virtual Provider. Technically spoken, there is yet no stateful session concept situated inside the Virtual Provider. Thus, it could not relate one request by a requester to another one.

---

\* Work and research on this paper were partly funded by the EU-project DIP.



**Fig. 1.** Strategy Pattern

This model is then extended in Sect. 2.3 by a notion of state facilitating stateful process descriptions. Therefore, in a set of subsequent messages whose processing interrelates, the initial request a message belongs to must be extractable from each but the first message sent in a sequence. In practical Web applications, this additional information is commonly attached to each message by a wrapping session handling module.

Inside the Virtual Provider, an initial request can be split up into sequential and parallel subrequests to be distributed to further providers (Sect. 3.1). For performing more sophisticated operations, one could either extend the presented model itself. Or, without touching the current model, one could wrap and chain the proposed model by machines providing further functionality of process adaption respectively.

## 2 The Interface Description of Virtual Providers

We see a Virtual Provider (VP) or mediator as an interface (technically speaking as an ASM module), which is defined by the following five methods (technically speaking ASMs):

- RECEIVEREQ for receiving request messages (elements of a set  $InReqMssg$ ) from clients<sup>1</sup>,
- SENDANSW for sending answer messages (elements of a set  $OutAnswMssg$ ) back to clients,
- PROCESS to handle *ReceivedRequests* (elements of a set  $ReqObj$  of internal representations of received requests, say as request objects), typically by sending to providers request messages for a series of subrequests, which are needed to service the currently handled request  $currReqObj$ <sup>2</sup>,

<sup>1</sup> Since instances of the abstract machines VIRTUALPROVIDER we are going to define here can be composed (see Sect. 3.1), such a client can also be another virtual provider *VP* asking for servicing a subrequest of a request received by *VP*.

<sup>2</sup> We deliberately keep the underlying message passing system abstract, so that the scheme we are going to develop for Virtual Providers can be instantiated in such

- SENDREQ for sending outgoing request messages (elements of a set  $OutReqMssg$ ) to providers, which may again be virtual providers (see the composition patterns discussed in Sect. 3.1,
- RECEIVEANSW for receiving incoming answer messages (elements of a set  $InAnswMssg$ ) from providers.

This view of a VIRTUALPROVIDER as a module (i.e. a collection of defined and callable machines, without a main machine that controls the execution flow) results from the decision to separate the scheduling of these components from their execution. The underlying architecture is illustrated in Fig. 2

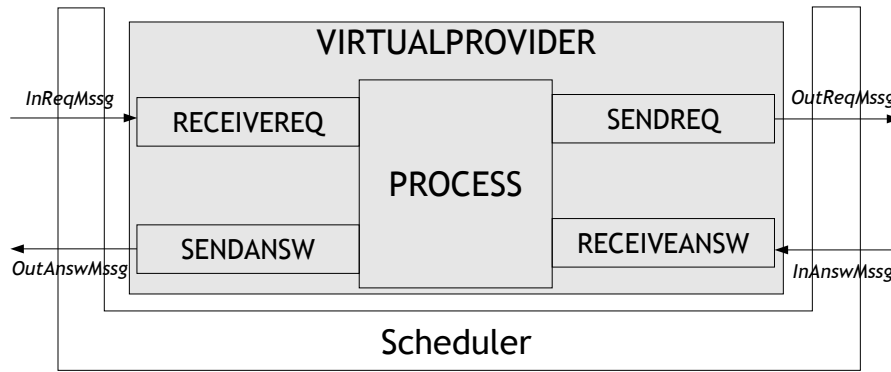


Fig. 2. Architecture

```
MODULE VIRTUALPROVIDER =
  RECEIVEREQ SENDANSW PROCESS SENDREQ RECEIVEANSW
```

If one wants to express a pure interleaving scheduler, it suffices to turn the module into an ASM, which at each moment non-deterministically chooses one of its submachines for execution (where we abstain from representing here the selection of the parameters involved in such submachine calls):

```
VIRTUALPROVIDER =
  choose M ∈ {RECEIVEREQ, SENDANSW} ∪
    {PROCESS, SENDREQ, RECEIVEANSW}
  M
```

---

a way that also PROCESS itself can be a provider and thus service a subrequest ‘internally’. This reflects that the mediation role for a request is different from the role of servicing a (sub)request.

We formulate first the “stateless” model for the communication between clients and (virtual) providers, which assumes that the relevant state information for every received or sent message is contained in the message. The stateless model is conceptually simpler than a model with state, which can be defined as a refined version (read: more detailed purely incremental extension) of the stateless model, as we will discuss below.

## 2.1 Abstract Message Passing

For receiving and sending request and answer messages we abstract from the particularities of a concrete message passing system. This means that we use the following behavioral communication interfaces (to be imported predicates and machines, specified and implemented elsewhere) for mail boxes of incoming and outgoing messages.

- a predicate *ReceivedReq*, used by RECEIVEREQ and expressing that an incoming request message (which is passed as argument to the predicate) has been received from some client (which we suppose to be encoded into the message),
- a predicate *ReceivedAnsw* used by RECEIVEANSW and expressing that an answer message (to a previously sent request message, which we suppose to be retrievable from the message) has been received,
- a machine SEND, used a) by SENDANSW for sending out answer messages to requests back to the clients where the requests originated, b) by SENDREQ for sending out requests to providers. In both cases we assume the addressees to be known or encoded into the message.

For reasons of modularity we separate the internal preparation of outgoing answer or request messages in PROCESS from the machine SEND which does the actual sending and relates to the communication medium we want to keep abstract. We therefore use in addition to *ReceivedReq* and *ReceivedAnsw* the following two abstract predicates for mail boxes of outgoing mail:

- *SentAnswToMailer* expressing that an outgoing answer message (elaborated from a PROCESS internal representation of an answer) has been sent to SEND, to be actually passed to the external message passing system,
- *SentReqToMailer* expressing that an outgoing request message (corresponding to an internal representation of a request) has been sent to SEND to be actually passed to the external message passing system.

## 2.2 The SEND/RECEIVE submachines

The interaction between a client and a virtual provider, which is triggered by the arrival of a client’s request message via the message passing system, is characterized by creating at the VP a request object (a request ID, say element  $r$  of a set *ReqObj* of currently alive request objects), which is appropriately initialized by recording in an internal representation the relevant data, which are encoded in the received request message. This includes decorating that object

by an appropriate *status*, say  $status(r) := started$ , to signal to (the scheduler for) PROCESS its readiness for being processed, and by other useful information.

This requirement for the machine RECEIVEREQ is captured by the following definition, which is parameterized by the incoming request message  $inReqMsg$  (supposed to belong to the set  $InReqMssg$  of legal incoming request messages) and by the set  $ReqObj$  of current request objects of the VIRTUALPROVIDER. For simplicity of exposition we assume a preemptive *ReceivedReq* predicate<sup>3</sup>.

```

RECEIVEREQ( $inReqMsg, ReqObj$ ) =
  if ReceivedReq( $inReqMsg$ ) then
    CREATENEWREQOBJ( $inReqMsg, ReqObj$ )
where
CREATENEWREQOBJ( $m, R$ ) =
  let  $r = New(R)$ 4 in
    INITIALIZE( $r, m$ )

```

The interaction between a virtual provider and a client, which consists in sending back a message providing an answer to a previous request of the client, is characterized by the underlying request object having reached through further PROCESSING a *status* where a call to SENDANSW with corresponding parameter  $outAnswerMsg$  has been internally prepared by PROCESS — say by setting an answer-mailbox predicate *SentAnswToMailer* for this argument to *true*. Thus one can specify SENDANSW, and symmetrically SENDREQ with a request-mailbox predicate *SentReqToMailer*, as follows:<sup>5</sup>

```

SENDANSW( $outAnswerMsg, SentAnswToMailer$ ) =
  if SentAnswToMailer( $outAnswerMsg$ ) then SEND( $outAnswerMsg$ )

SENDREQ( $outReqMsg, SentReqToMailer$ ) =
  if SentReqToMailer( $outReqMsg$ ) then SEND( $outReqMsg$ )

```

For the definition of RECEIVEANSW we use as parameter the *AnswerSet* function which provides for every *requestor*  $r$ , 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.<sup>6</sup>

<sup>3</sup> Otherwise a DELETE( $inReqMsg$ ) has to be added with the effect that the execution of RECEIVEREQ( $inReqMsg, ReqObj$ ) switches *ReceivedReq*( $inReqMsg$ ) from *true* to *false*.

<sup>4</sup> *New* is assumed to provide at each call a sufficiently fresh element in the indicated domain.

<sup>5</sup> For the sake of generality, we assume the destinators of messages to be encoded into the message.

<sup>6</sup> The function  $requestor(inAnswMsg)$  is defined below to denote the value of  $seqSubReq$  in the state when the request message  $outReq2Mssg(s)$  for the parallel subrequest  $s$  was sent out to which the  $inAnswMsg$  is received now.

RECEIVEANSW(*inAnswMsg*, *AnswerSet*)<sup>7</sup> =  
**if** *ReceivedAnsw*(*inAnswMsg*) **then**  
    insert *answer*(*inAnswMsg*) into *AnswerSet*(*requestor*(*inAnswMsg*))

**Behavioral interface types.** Through the definitions below, we will link RECEIVEREQ and SENDANSW by the *status* function value for a *currReqObj*. This realizes that the considered communication interface is of the “provided behavioral interface” type, discussed in [1], where the RECEIVEREQ action corresponds to receive an incoming request, through which a new *reqObj* is created (*initStatus*(PROCESS)), and occurs before the corresponding SENDANSW action, which happens after the outgoing answer message in question has been *SentAnswToMailer*, namely when *reqObj* was reaching the *status deliver*. The pair of machines SENDREQ and RECEIVEANSW in PROCESS (more precisely in the submachine ITERATESUBREQPROCESSG defined below) realizes the symmetric “required behavioral interface” communication interface type, where the SEND actions in SENDREQ correspond to outgoing requests and thus occur before the corresponding RECEIVEANSW actions of the incoming answers to those requests.

### 2.3 Refinement by introducing a “state” component

We shortly discuss here how to extend the above model for RECEIVEREQ to equip virtual providers with some state for recording information on previously received requests, to be recognized when for such a request at a later stage some additional service is requested. The changes on the side of PROCESS defined below concern the inner structure of that machine and its refined notion of state and state actions. We concentrate our attention here therefore on the refinement of the RECEIVEREQ machine. This refinement is a simple case of the more general ASM refinement concept in [2].

The first addition needed for RECEIVEREQ is a predicate *NewRequest* to check, when an *inReqMsg* is received, whether that message contains a new request, or whether it is about an already previously received request. In the first case, CREATENEWREQOBJ as defined above is called. In the second case, instead of creating a new request object, the already previously created request object corresponding to the incoming request message has to be retrieved, using some function *prevReqObj*(*inReqMsg*), to REFRESHREQOBJ by the additional information on the newly arriving further service request. In particular, a decision has to be taken upon how to update the *status*(*prevReqObj*(*inReqMsg*)), which depends on how one wants the processing *status* of the original request to be influenced by the additional request or information presented through *inReqMsg*. Since we want to keep the scheme general, we assume that an external scheduling function *refreshStatus* is used in an update

<sup>7</sup> Without loss of generality we assume this machine to be preemptive (i.e. *ReceivedAnsw*(*inAnswMsg*) gets false by firing RECEIVEANSW for *inAnswMsg*).

$status(r) := refreshStatus(r, inReqMsg)$ <sup>8</sup>. This leads to the following refinement of RECEIVEREQ, where we skip notationally the set and function parameters  $ReqObj$ ,  $prevReqObj$ :

```

RECEIVEREQ( $inReqMsg$ ) = if  $ReceivedReq(inReqMsg)$  then
  if  $NewRequest(inReqMsg)$  then
    CREATENEWREQOBJ( $inReqMsg, ReqObj$ )
  else let  $r = prevReqObj(inReqMsg)$  in
    REFRESHREQOBJ( $r, inReqMsg$ )

```

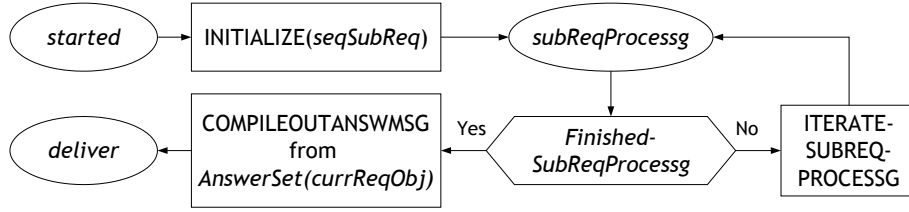
### 3 The PROCESSING Submachine of VIRTUALPROVIDERS

In this section we define the signature and the transition rules of the Abstract State Machine PROCESS for the processing kernel of a virtual provider. The signature definition really provides a schema, which is to be instantiated for each particular PROCESSING kernel of a concrete Virtual Provider, namely by giving concrete definitions for the abstract functions we are going to introduce (see their listing in the documentation section below). We give an example for this in Sect. 4.

In our definition we want to abstract from the scheduling mechanism which calls PROCESS for a particular current request object  $currReqObj$ . We therefore describe the machine as parameterized by such a  $currReqObj \in ReqObj$ , which plays the role of a global instance variable. The definition is given in terms of control state ASMs in Fig. 3, using the standard graphical representation of finite automata or flowcharts as graphs with circles (for the internal states, here to be interpreted as current value of  $status(currReqObj)$ ), rhombs (for test predicates) and rectangles (for actions).

The definition in Fig. 3 expresses that each PROCESSING call for a *started* request object  $currReqObj$  triggers to INITIALIZE an iterative sequential subrequest processing, namely of the immediate subrequests of this  $currReqObj$ , in the order defined by an iterator over a set  $SeqSubReq(currReqObj)$ . This reflects the first part of the hierarchical view underlying our specification of VP request processing, namely that each incoming (top level) request object  $currReqObj$  triggers the sequential elaboration of a finite number of immediate subrequests, members of a set  $SeqSubReq(currReqObj)$ , called for short sequential subrequests. As will be explained below, we view each such sequential subrequest to trigger a finite number of further subsubrequests, which are sent to external providers where they are elaborated independently of each other, so that we call them parallel subrequests of the sequential subrequest.

<sup>8</sup> What if  $status(prevReqObj(inReqMsg))$  is simultaneously updated by the refined RECEIVEREQ and by PROCESS as defined below? In case of a conflicting update attempt the ASM framework stops the computation; at runtime such an inconsistency is notified by ASM execution engines. Implementations will have to solve this problem in the scheduler of VIRTUALPROVIDER.



**Fig. 3.** PROCESSING( $currReqObj$ )

PROCESS uses for the elaboration of the sequential subrequests of  $currReqObj$  a submachine ITERATESUBREQPROCESSG specified below. Once PROCESS has  $FinishedSubReqProcessg$ , it compiles from  $currReqObj$  (which allows to access  $AnswerSet(currReqObj)$ ) an answer, say  $outAnswer(currReqObj)$ , and transforms the internal answer information  $a$  into an element of  $OutAnswMssg$  using an abstract function  $outAnsw2Mssg(a)$ . We guard this answer compilation by a previous check whether  $AnswToBeSent$  for the  $currReqObj$  (including the  $AnswerSet(currReqObj)$ ) evaluates to true.

For the sake of illustration we also provide here the textual definition of the machine defined in Fig. 3. For this purpose we use a function  $initStatus$  to yield for a control state ASM its initial control status, which is hidden in the graphical representation. The function  $seqSubReq(currReqObj)$  denotes the current item of the iterator submachine ITERATESUBREQPROCESSG defined below.

```

PROCESS( $currReqObj$ ) =
  if  $status(currReqObj) = started$  then
    INITIALIZE( $seqSubReq(currReqObj)$ )
     $status(currReqObj) := subReqProcessg$ 
  if  $status(currReqObj) = subReqProcessg$  then
    if  $FinishedSubReqProcessg$  then
      COMPILEOUTANSWMSG from  $currReqObj$ 
       $status(currReqObj) := deliver$ 
    else
      StartNextRound(ITERATESUBREQPROCESSG)
  where
    COMPILEOUTANSWMSG from  $currReqObj$  =
      if  $AnswToBeSent(currReqObj)$  then
        SentAnswToMailer( $outAnsw2Mssg(outAnswer(currReqObj))$ ) :=
          true
    StartNextRound(M) =
       $status(currReqObj) := initStatus(M)$ 
  
```

The submachine to ITERATESUBREQPROCESSG is an iterator machine defined in Fig. 4. For every current item  $seqSubReq$ , it starts to FEEDSENDREQ with a request message to be sent out for every immediate subsubrequest  $s$  of the current  $seqSubReq$ , namely by setting  $SentReqToMailer(outReq2Mssg(s))$  to  $true$ . Here  $outReq2Mssg(s)$  transforms the outgoing request into the format for an outgoing request message, which has to be an element of  $OutReqMssg$ . Since those immediate subsubrequests, elements of a set  $ParSubReq(seqSubReq)$ , are assumed to be processable by other providers independently of each other, FEEDSENDREQ elaborates simultaneously for each  $s$  an  $outReqMssg(s)$ .

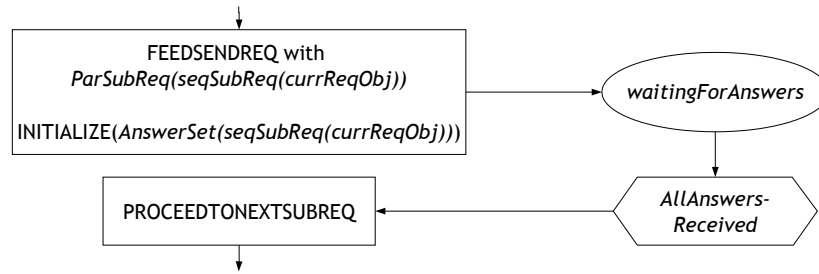


Fig. 4. ITERATESUBREQPROCESSG

Simultaneously ITERATESUBREQPROCESSG also INITIALIZES the to be computed  $AnswerSet(seqSubReq)$  before assuming  $status$  value  $waitingForAnswers$ , where it remains until  $AllAnswersReceived$ . When  $AllAnswersReceived$ , the machine ITERATESUBREQPROCESSG will PROCEEDTONEXTSUBREQ.

As long as during  $waitingForAnswers$ ,  $AllAnswersReceived$  is not yet true, RECEIVEANSW inserts for every  $ReceivedAnsw(inAnswMsg)$  the retrieved internal  $answer(inAnswMsg)$  representation into  $AnswerSet(seqSubReq)$  of the currently processed sequential subrequest  $seqSubReq$ , which is supposed to be retrievable as  $requestor$  of the incoming answer message.

For the sake of illustration we provide here also the textual version for the definition given in Fig. 4.

```

ITERATESUBREQPROCESSG =
  if status(currReqObj) = initState(ITERATESUBREQPROCESSG) then
    FEEDSENDREQ with ParSubReq(seqSubReq(currReqObj))
    INITIALIZE(AnswerSet(seqSubReq(currReqObj)))
    status(currReqObj) := waitingForAnswers
  if status(currReqObj) = waitingForAnswers then
    if AllAnswersReceived then
      PROCEEDTONEXTSUBREQ
      status(currReqObj) := subReqProcessg
  
```

**where**

FEEDSENDREQ with  $ParSubReq(seqSubReq) =$   
**forall**  $s \in ParSubReq(seqSubReq)$   
 $SentReqToMailer(outReq2Mssg(s)) := true$

For the sake of completeness we now define the remaining macros used in Fig. 4, though their intended meaning should be clear from the chosen names.

**Iterator Pattern on  $SeqSubReq$ .** The following three definitions concern an iterator pattern defined by

- $seqSubReq$ , denoting the current item in the underlying set  $SeqSubReq \cup \{ Done(SeqSubReq(currReqObj)) \}$ ,
- two functions  $FstSubReq$  and  $NxtSubReq$ , the latter defined on the sets  $SeqSubReq$  of sequential subrequests and on  $AnswerSets$ ,
- the stop element  $Done(SeqSubReq(currReqObj))$ , which is constrained by the condition that it is not an element of any set  $SeqSubReq$ .

INITIALIZE( $seqSubReq$ ) =  
**let**  $r = FstSubReq(SeqSubReq(currReqObj))$   
 $seqSubReq := r$   
 $ParSubReq(r) := FstParReq(r, currReqObj)$

$FinishedSubReqProcessg =$   
 $seqSubReq(currReqObj) = Done(SeqSubReq(currReqObj))$

PROCEEDTONEXTSUBREQ =  
**let**  $s = NxtSubReq(SeqSubReq(currReqObj), seqSubReq(currReqObj),$   
 $AnswerSet(currReqObj))$   
 $seqSubReq(currReqObj) := s$   
 $ParSubReq(s) := NxtParReq(s, currReqObj, AnswerSet(currReqObj))$

This iterator pattern foresees the possibility that  $NxtSubReq$  and  $NxtParReq$  are determined in terms of the answers accumulated so far for the overall request object, i.e. taking into account the answers obtained for preceding sequential subrequests ( $currReqObj$  can be used to access  $AnswerSet(currReqObj)$ ).

**Answer Sets.**

INITIALIZE( $AnswerSet(seqSubReq)$ ) =  
 $AnswerSet(seqSubReq) := \emptyset$

$AllAnswersReceived =$   
**let**  $seqSubReq = seqSubReq(currReqObj)$   
**for each**  $req \in ToBeAnswered(ParSubReq(seqSubReq))$   
**there is some**  $answ \in AnswerSet(seqSubReq)$

The definition foresees the possibility that some of the parallel subrequest messages, which are sent out to providers, may not necessitate an answer for the virtual provider: a function  $ToBeAnswered$  filters them out from the condition  $waitingForAnswers$  to leave the current iteration round.

The answer set of any main request object can be defined as a derived function of the answer sets of its sequential subrequests:

$$\begin{aligned} \text{AnswerSet}(\text{reqObj}) = \\ \text{Combine}(\{\text{AnswerSet}(s) \mid s \in \text{SeqSubReq}(\text{reqObj})\}) \end{aligned}$$

### 3.1 Composing Virtual Providers

We have formulated VIRTUALPROVIDER in such a way that different instances  $VP_1, \dots, VP_n$  of it can be configured into say a sequence, leading to a virtual provider  $VP_1$ , which involves a subprovider  $VP_2$ , which involves its own subprovider  $VP_3$  etc. For such a composition it suffices to connect the communication interfaces in the appropriate way, in the case of a sequence as follows:

- SENDREQ of  $VP_i$  with the RECEIVERREQ 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).

Such a sequential composition allows one to configure VP (mediator) schemes where each element  $seq_1$  of a sequential subrequest set  $SeqSubReq_1$  of an initial request can trigger a set  $ParSubReq(seq_1)$  of parallel subrequests  $par_1$ , each of which can trigger a set  $SeqSubReq_2$  of further sequential subrequests  $seq_2$  of  $par_1$ , each of which again can trigger a set  $ParSubReq(seq_2)$  of further parallel subrequests, etc. This provides the possibility of unfolding arbitrary alternating seq/par trees, where at each level one has sequential subrequests each of which branches into a subtree of parallel subsubrequests, each of which may have a subtree of other sequential subrequests, etc.

Obviously for VIRTUALPROVIDER even more complex composition schemes can be defined, if one wants to do this.

### 3.2 Defining Equivalence Notions for Virtual Providers

To be able to speak about the relation between incoming requests and outgoing answers, one has to relate the elements of the corresponding sets  $InReqMssg$  and  $OutAnswMssg$ . Formally, this comes up to unfold the function *originator*, which for an  $outAnswerMsg$  yields the  $inReqMsg$  to which  $outAnswerMsg$  represents the answer. In fact this information is retrievable by COMPILEOUTANSWMSG from the  $currReqObj$ , if it was recorded there by CREATENEWREQOBJ( $inReqMsg, ReqObj$ ) as part of INITIALIZE.

One can then define the *ServiceBehavior*( $VP$ ) of a virtual provider  $VP = \text{VIRTUALPROVIDER}$  as the correspondence between any  $inReqMsg$  and the  $outAnswerMsg$  related to it by the *originator* function:

$$\text{originator}(\text{outAnswerMsg}) = \text{inReqMsg}$$

Two virtual providers  $VP$ ,  $VP'$  can be considered equivalent if an equivalence relation  $ServiceBehavior(VP) \equiv ServiceBehavior(VP')$  holds between their service behavior. To concretely define such an equivalence involves detailing of the meaning of service ‘requests’ and provided ‘answers’, which comes up to further detail the abstract  $VP$  model in such a way that the intended ‘service’ features and how they are ‘provided’ by  $VP$  become visible. On the basis of such definitions one can then formally define different  $VP$ s to be alternatives for a strategy pattern for providing requested services.

Another interesting class of relations becomes rigorously analysable if we consider the pair  $OutReqMssg$  and  $InAnswMssg$  on the requester side of a  $VP$ .

## 4 Virtual ISP

One of the use cases in the DIP project<sup>9</sup> deals with implementing a so-called *Virtual ISP (VISP)*. A Virtual ISP resells customers products that are bundled from offerings of different Internet Service Providers. A typical example for such a product bundle could be an *Internet presence* including a personal web server and a personal e-mail address, both bound to a dedicated, user-specific domain name, like, e.g., `michael-altenhofen.de`. Such an Internet presence would require this domain name to be registered (at a central registry, e.g. DENIC). Ideally, the VISP wants to handle domain name registrations in a unified manner using a fixed interface. For this example we assume that this interface contains only one request message:<sup>10</sup> *RegisterDomain*. It requires four input parameters, namely

- **DomainName**, the name of the new domain that should be registered
- **DomainHolderName**, the name of the person that will (in a legal sense) own the domain
- **AdministrativeContactName** the name of the person that will administer the domain, and
- **TechnicalContactName**, the name of the person that will be responsible for technical issues.

On successful registration, the answer will contain four so-called RIPE-Handles,<sup>11</sup> uniquely identifying the four names provided in the request message in the RIPE database for future reference.

## 5 A possible VP refinement for *RegisterDomain*

Let’s assume that the VISP is extending its business into a new country whose domain name registry authority implements a different interface for registering new domain names. Its interface consists of four request messages:

<sup>9</sup> See <http://dip.semanticweb.org>

<sup>10</sup> We stick to a procedural description of the request message to keep the example short. A full Web Service example would require a WSDL description.

<sup>11</sup> RIPE stands for Réseaux IP Européens, see <http://ripe.net> .

- *RegisterDH* (DomainHolderName),
- *RegisterAC* (AdministrativeContactName),
- *RegisterTC* (TechnicalContactName), and
- *RegisterDN* (DomainName, DO-RIPE-Handle, AC-RIPE-Handle, TC-RIPE-Handle).

A possible Virtual Provider instance for that scenario is depicted in Fig. 5.<sup>12</sup>

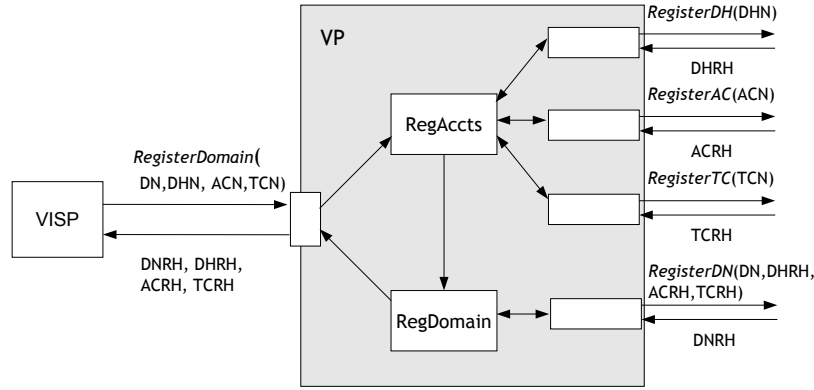


Fig. 5. VP instance

Within that VP, the incoming request *RegisterDomain* is split up into a sequence of two subrequests. The first subrequest is further divided into three parallel subrequests, each registering one of the resp. contacts. Once all answers for these parallel subrequests have been received, the second sequential subrequest can be performed, whose outgoing request message is constructed from the answers of the previous subrequest and the *DomainName* parameter from the incoming request.

Using the notation convention of appending *Obj* when referring to the internal representations of the different requests we can now refine the VP ASM to the instance described above as follows: We start with refining the INITIALIZE ASM:

$$\begin{aligned}
 \text{INITIALIZE}(\text{RegisterDomainObj}, \text{RegisterDomain}(\text{DN}, \text{DHN}, \text{ACN}, \text{TCN})) = & \\
 \text{params}(\text{RegisterDomainObj}) := \{\text{DN}, \text{DHN}, \text{ACN}, \text{TCN}\} & \\
 \text{SeqSubReq}(\text{RegisterDomainObj}) := \{\text{RegAccts}, \text{RegDomain}\} & \\
 \text{FstSubReq}(\{\text{RegAccts}, \text{RegDomain}\}) := \text{RegAccts} & \\
 \text{NxtSubReq}(\{\text{RegAccts}, \text{RegDomain}\}, \text{RegAccts}, -) := \text{RegDomain} &
 \end{aligned}$$

<sup>12</sup> We have abbreviated the request message and parameter names to keep the size of the figure manageable.

$$\begin{aligned}
& \text{NxtSubReq}(\{ \text{RegAccnts}, \text{RegDomain} \}, \text{RegDomain}, -) := \text{nil} \\
& \text{FstParReq}(\text{RegAccnts}, \text{RegisterDomainObj}) := \\
& \quad \{ \text{RegisterDH}(\text{DHN}), \text{RegisterAC}(\text{ACN}), \\
& \quad \text{RegisterTC}(\text{TCN}) \} \\
& \text{NxtParReq}(\text{RegDomain}, \text{RegisterDomainObj}, \text{AS}) := \\
& \quad \{ \text{RegisterDN}(\text{DN}, \text{handle}(\text{DHRHObj}), \\
& \quad \text{handle}(\text{ACRHObj}), \text{handle}(\text{TCRHObj}) \} \\
& \text{AnswToBeSent}(\text{RegisterDomainObj}) := \text{true} \\
& \text{ToBeAnswered}(\{ \text{RegisterDH}, \text{RegisterAC}, \text{RegisterTC} \}) := \\
& \quad \{ \text{RegisterDH}, \text{RegisterAC}, \text{RegisterTC} \} \\
& \text{ToBeAnswered}(\{ \text{RegisterDN} \}) := \{ \text{RegisterDN} \} \\
& \text{status}(\text{RegisterDomainObj}) := \text{started}
\end{aligned}$$

**where**

$$\begin{aligned}
& \text{AS} = \{ \text{DHRHObj}, \text{ACRHObj}, \text{TCRHObj} \} \\
& \text{handle}(X) = \begin{cases} \text{DHRH} & \text{if } X = \text{DHRHObj} \\ \text{DNRH} & \text{if } X = \text{DNRHObj} \\ \text{ACRH} & \text{if } X = \text{ACRHObj} \\ \text{TCRH} & \text{if } X = \text{TCRHObj} \end{cases}
\end{aligned}$$

The derived function *Combine* simply computes the union of the two answer sets:

$$\begin{aligned}
& \text{Combine}(\text{RegisterDomainObj}) = \\
& \quad \text{AnswerSet}(\text{RegAccnts}) \cup \text{AnswerSet}(\text{RegDomain})
\end{aligned}$$

Next, we define the *answer* function that maps an incoming message to their internal representation:

$$\text{answer}(\text{inAnswMsg}) = \begin{cases} \text{DHRHObj} & \text{if } \text{inAnswMsg} = \text{DHRH} \\ \text{DNRHObj} & \text{if } \text{inAnswMsg} = \text{DNRH} \\ \text{ACRHObj} & \text{if } \text{inAnswMsg} = \text{ACRH} \\ \text{TCRHObj} & \text{if } \text{inAnswMsg} = \text{TCRH} \end{cases}$$

Finally, we give a definition of *outAnsw2Msg*.

$$\begin{aligned}
& \text{outAnsw2Msg}(\{ \text{DHRHObj}, \text{DNRHObj}, \text{ACRHObj}, \text{TCRHObj} \}) = \\
& \quad \text{Formatted}(\{ \text{DNRH}, \text{DHRH}, \text{ACRH}, \text{TCRH} \})
\end{aligned}$$

Note that this definition introduces a new function *Formatted* that is used to transform the parameters into the format expected by the requester, in our case the VISP. We leave it abstract here, since the actual implementation is irrelevant in the context of this example scenario<sup>13</sup>.

<sup>13</sup> All it would probably do is sorting the parameters.

## 6 Conclusions and Future Work

This paper provides a formal, high-level model of process mediation. By the use of ASMs as a modeling paradigm, the presented abstract state machine builds a base for “*communicating and documenting design ideas*” and supports “*an accurate and checkable overall understanding*” of the controversially discussed topic of process mediation. Furthermore, the ASM method allows to “*isolate the hard part of a system*” [3, p. 14-15] and thus to concentrate on the essential parts for refinement.

Process mediation can be seen as one part of the whole Semantic Web services (SWS) usage process [5]. This area of current research also is under strong motion and encounters heterogeneous usage of terms. Different approaches are published as frameworks each presenting a consistent view on “their” SWS usage process (cmp. [4,5,7]). However, different frameworks are not necessarily using terms in the same way as competing frameworks do. ASMs could help providing a means of *explicit, exact and formal specification* and delimitation of terms used in different frameworks, and combine them towards a consistent view of the general SWS usage process.

Specifying a fixed, abstract SWS usage framework, moreover yields the possibility of bridging controversial approaches like dynamic composition vs. static composition through explicitly showing their differences by their individual refinements of the abstract framework. The same is true for the presented VP. As shown in Sect. 4, different specializations constrain the VP’s behaviour to specific patterns. So far, only very narrow refinements have been presented. We could also imagine more generous specializations framing the later refinement possibilities, e.g. especially for specific types of interaction patterns.

## References

1. A. Barros, M. Dumas, and P. Oaks. A critical overview of the web services choreography description language (WS-CDL). White paper, 24th of January 2005.
2. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
3. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
4. C. Bornhövd and A. Buchmann. Semantically meaningful data exchange in loosely coupled environments. 2000.
5. D. Fensel and C. Bussler. The web service modeling framework wsmf. 2002.
6. Y. Lee, C. Patel, S. A. Chun, and J. Geller. Compositional knowledge management for medical services on semantic web. In *WWW*. ACM, May 2004.
7. P. F. Pires, M. R. F. Benevides, and M. Mattoso. Building reliable web services compositions. 2002.
8. M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition, 2004.
9. M. Stumptner. Configuring web services, 2004.