An Abstract Model for Process Mediation *

Michael Altenhofen[1], Egon Börger[2], and Jens Lemcke[1]

[1] SAP Research, Karlsruhe, Germany
{michael.altenhofen, jens.lemcke}@sap.com
[2] 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 behavioural interface of abstract Virtual Providers (VP), their refinements and their composition into rich mediator structures. We show for a Virtual Internet Service Provider example how to use such a model for rigorously formulating and proving properties of interest.

1 Introduction

For the configuration [1] and composition [2,3] of web services in interaction protocols, a central role is played by process mediation (see MIBIA [4], WSMF [5], WebTransact [6]). We propose here an abstract model for mediators (Sects. 2, 3), viewed as Virtual Providers (VP). The model supports provably correct mediator composition and the definition of appropriate equivalence concepts (Sect. 4), which underlay algorithms for the discovery and run-time selection of services satisfying given requests. In Sect. 5 we illustrate our definitions by a Virtual Internet Service Provider case study.

We start with a simple interaction model where each single request receives a single answer from the VP, with no need to relate multiple requests. However, to process single requests the VP has a hierarchical structure at its disposition: each request arriving at VP is viewed as root of a so-called *seq/par tree* of further requests, which are forwarded to other providers. The children of a request node represent subrequests which are elaborated in sequence. Each subrequest node may have in turn children representing multiple subsubrequests, which are elaborated independently of each other. Nestings of such alternating seq/par trees and other more sophisticated hierarchical subrequest structures can be obtained by appropriate compositions of VPs as defined in Sect. 4.1.

The compositionality of our mediator model stems from an explicit separation of its tree processing component from its communication interfaces for sending/receiving requests/answers. This separation, defined in Sect. 2 on the basis of an abstract message passing system, supports a flexible definition of the service behaviour of VPs and of their behavioural equivalence (Sect. 4), which

* Work on this paper was partly funded by the EU-project DIP.

also allows one to clearly identify the place of data mediation during the discovery and runtime selection of providers able to satisfy given requests. Furthermore, the separation of communication from proper request processing supports a smooth integration of a variety of workflow and interaction patterns [7,8].

In Sect. 2.3 the single-request oriented model is refined by a notion of internal state, so that the relevant information about previous requests, which may be related to an incoming request, can be extracted from the internal state — in practical Web applications typically by a wrapping session handling module. This refinement step is only a tiny illustration of much more one can do to turn our abstract VP model in a faithful way into fully developed mediator code.

As modelling framework we use Abstract State Machines (ASM),[3] a form of pseudo-code working on arbitrary structures. An introduction into the ASM method for high-level system design and analysis is available in textbook form in [9], but most of what we use here is self-explanatory. The various refinements used are instances of the general ASM refinement concept defined in [10].

2 The Communication Interface of VIRTUAL PROVIDERs

We see a VP as an interface (technically speaking as an ASM module VIRTUALPROVIDER) providing the following five methods (read: ASMs):

- RECEIVEREQ for receiving request messages (elements of a set *InReqMssg* of legal incoming request messages) from clients,[4]
- SENDANSW for sending answer messages (elements of a set *OutAnswMssg*) back to clients,
- PROCESS to handle request objects, elements of a set *ReqObj* of internal representations of *ReceivedReq*uests, typically by sending to providers a series of subrequests to service the currently handled request *currReqObj*,[5]
- SENDREQ for sending request messages (elements of a set *OutReqMssg*) to providers (possibly other VPs, see the VP composition in Sect. 4.1),
- RECEIVEANSW for receiving incoming answer messages (elements of a set *InAnswMssg*) from providers.

This module view of VIRTUALPROVIDER — as a collection of defined and callable machines, without a main ASM defining the execution flow — separates the specification of the functionality of VP components from that of their schedulers. The underlying architecture is illustrated in Fig. 1.

[3] This is not the place for a systematic comparison of different methods. The model developed in this paper starts from scratch, which explains that, besides what is cited in Sect. 6, there is no other related work we used.

[4] Since instances of VIRTUALPROVIDER can be composed (see Sect. 4.1), such a client can be another VP' asking for servicing a subrequest of a received request.

[5] Since the underlying message passing system is abstract, VIRTUALPROVIDER can be instantiated in such 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 actually servicing it.

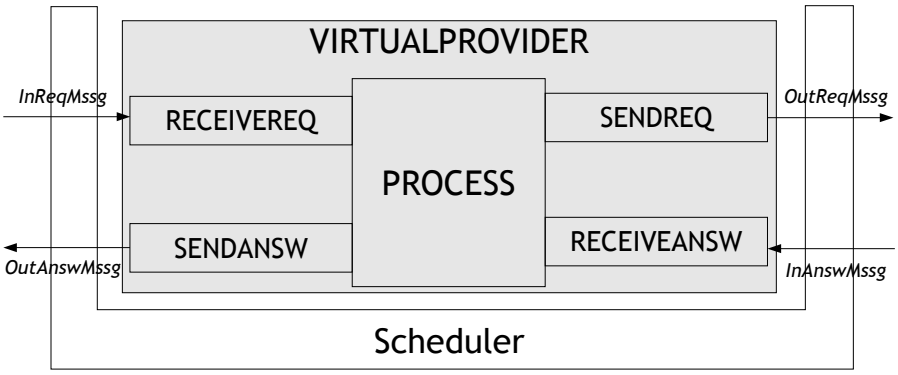

Fig. 1. Architecture

MODULE VirtualProvider =
ReceiveReq SendAnsw Process SendReq ReceiveAnsw

2.1 Abstract Message Passing

For sending/receiving request/answer messages we abstract from a concrete message passing system by using abstract communication interfaces (predicates) for mail boxes of incoming and outgoing messages.

- *ReceivedReq* in ReceiveReq expresses that an incoming request message has been received from some client (supposed to be encoded into the message).
- *ReceivedAnsw* in ReceiveAnsw expresses that an answer message (to a previously sent supposed to be retrievable request message) has been received.
- An abstract machine Send is 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. We assume the addressees to be encoded into messages.

We separate the internal preparation of outgoing messages in Process from their actual sending in Send by using the following abstract predicates for mail boxes of outgoing mail:

- *SentAnswToMailer* expresses that an outgoing answer message (elaborated from a Process internal representation of an answer) was passed to Send.
- *SentReqToMailer* expresses that an outgoing request message (corresponding to an internal representation of a request) has been passed to Send.

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 so that *ReceivedReq*(*inReqMsg*) becomes

true, is characterized by creating 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.

This requirement for the machine RECEIVEREQ is captured by the following definition, which is parameterized by the incoming request message *inReqMsg* and by the set *ReqObj* of current request objects of the VIRTUALPROVIDER. For simplicity of exposition we assume a preemptive *ReceivedReq* predicate.[6]

RECEIVEREQ(*inReqMsg*, *ReqObj*) = **if** *ReceivedReq*(*inReqMsg*) **then**
  CREATENEWREQOBJ(*inReqMsg*, *ReqObj*)
where CREATENEWREQOBJ(m, R) =
  let $r = new(R)$[7] **in** INITIALIZE(r, m)

The inverse 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 *outAnswMsg* has been internally prepared by PROCESS — namely by setting the answer-mailbox predicate *SentAnswToMailer* for this argument to *true*. Thus one can specify SENDANSW, and symmetrically SENDREQ with the request-mailbox predicate *SentReqToMailer*, as follows:

SENDANSW(*outAnswMsg*, *SentAnswToMailer*) =
  if *SentAnswToMailer*(*outAnswMsg*) **then** SEND(*outAnswMsg*)

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.[8]

RECEIVEANSW(*inAnswMsg*, *AnswerSet*)[9] =
  if *ReceivedAnsw*(*inAnswMsg*) **then**
    insert *answer*(*inAnswMsg*) into *AnswerSet*(*requestor*(*inAnswMsg*))

[6] Otherwise a DELETE(*inReqMsg*) has to be added, so that the execution of RECEIVEREQ(*inReqMsg*, *ReqObj*) switches *ReceivedReq*(*inReqMsg*) to *false*.

[7] *new* is assumed to provide at each application a sufficiently fresh element.

[8] 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.

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

Behavioral interface types. Through the definitions below, we link calls of ReceiveReq and SendAnsw by the *status* function value for a *currReqObj*. Thus the considered communication interface is of the "provided behavioural interface" type, discussed in [11]: the ReceiveReq action corresponds to receive an incoming request, through which a new *reqObj* is created, and occurs before the corresponding SendAnsw action, which happens after the outgoing answer message in question has been *SentAnswToMailer* when *reqObj* was reaching the *status deliver*. The pair of machines SendReq and ReceiveAnsw in Process realizes the symmetric "required behavioural interface" communication interface type, where the Send actions correspond to outgoing requests and thus occur before the corresponding ReceiveAnsw actions of the incoming answers to those requests.

2.3 Refinement by a "state" component

It is easy to extend 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 on the refinement of the ReceiveReq machine. This refinement is a simple case of the general ASM refinement concept in [10].

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 *status*(*r*) := *refreshStatus*(*r*, *inReqMsg*).[10] This leads to the following refinement of ReceiveReq (we skip the parameters *ReqObj*, *prevReqObj*):

ReceiveReq(*inReqMsg*) = **if** *ReceivedReq*(*inReqMsg*) **then**
  if *NewRequest*(*inReqMsg*) **then**
    CreateNewReqObj(*inReqMsg*, *ReqObj*)

[10] 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.

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 ASM Process for the processing kernel of a virtual provider. The definition provides a schema, which is to be instantiated for each particular Processing kernel of a concrete Virtual Provider by giving concrete definitions for the abstract functions and machines we are going to introduce. For an example see Sect. 5.

Since we want to abstract from the scheduler, which calls Process for particular current request objects *currReqObj*, we describe the machine as parametrized by a global instance variable $currReqObj \in ReqObj$. The definition is given in Fig. 2 in terms of control state ASMs, 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 $status(currReqObj)$), rhombuses (for test predicates) and rectangles (for actions).

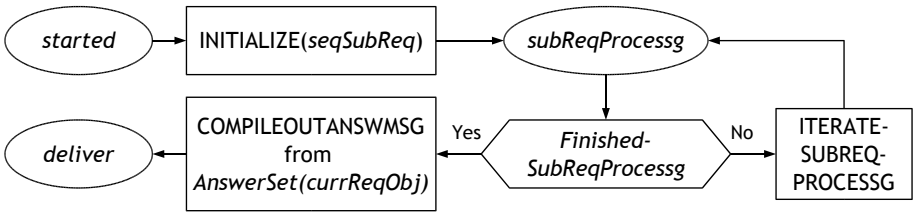

Fig. 2. Processing$(currReqObj)$

Figure 2 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 VP request processing view, 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 sequential subrequests. As explained below, each sequential subrequest may 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.

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 check whether *AnswToBeSent* for the *currReqObj* evaluates to true.

For the sake of illustration we also provide here the textual definition of the machine defined in Fig. 2. 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 o = if AnswToBeSent(o) then
    SentAnswToMailer(outAnsw2Mssg(outAnswer(o))) := true
  StartNextRound(M) = (status(currReqObj) := initStatus(M))
```

The submachine to IterateSubReqProcessg is an iterator machine defined in Fig. 3. 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 *outReqMsg*(*s*).

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.

```
IterateSubReqProcessg =
```

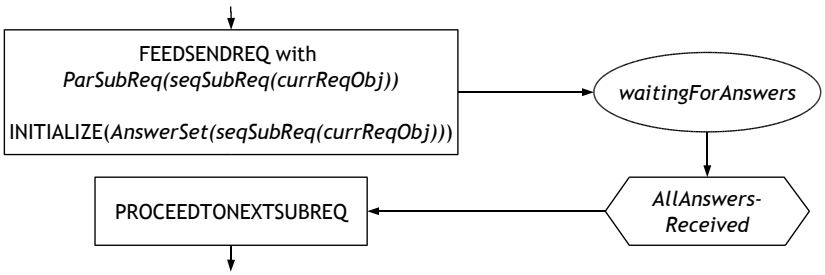

Fig. 3. IterateSubReqProcessg

if $status(currReqObj) = initStatus(\text{IterateSubReqProcessg})$ **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. 3, though their intended meaning should be clear from the chosen names. The **Iterator Pattern on** $SeqSubReq$ is defined by the following items:

- $seqSubReq$, denoting the current item in the underlying set $SeqSubReq \cup \{ Done(SeqSubReq(currReqObj)) \}$,
- functions $FstSubReq$ and $NxtSubReq$ operating on the set $SeqSubReq$ and $NxtSubReq$ also on $AnswerSet(currReqObj)$,
- the stop element $Done(SeqSubReq(currReqObj))$, constrained by not being an element of any set $SeqSubReq$.

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

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

ProceedToNextSubReq $=$
  let $o = currReqObj$
    $s = NxtSubReq(SeqSubReq(o), seqSubReq(o), AnswerSet(o))$ **in**

$seqSubReq(o) := s$
$ParSubReq(s) := NxtParReq(s, o, AnswerSet(o))$

This iterator pattern foresees that *NxtSubReq* and *NxtParReq* may be determined in terms of the answers accumulated so far for the overall request object, i. e. taking into account the answers obtained for preceding subrequests.

$\textsc{Initialize}(AnswerSet(seqSubReq)) = (AnswerSet(seqSubReq) := \emptyset)$

$AllAnswersReceived =$ **let** $seqSubReq = seqSubReq(currReqObj)$ **in**
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:

$AnswerSet(reqObj) =$
$Combine(\{AnswerSet(s) \mid s \in SeqSubReq(reqObj)\})$

4 Mediator Composition and Equivalence Notions

We show how to combine VPs and how to define their service behaviour, which allows one to define rigorous equivalence notions for VPs one can use a) to formulate algorithms for the discovery and runtime selection of providers suitable to satisfy given requests, and b) to prove VP runtime properties of interest.

4.1 Composing Virtual Providers

Instances $VP_1, \ldots, VP_n$ of VirtualProvider can be configured into a sequence with a first virtual provider VP_1 involving a subprovider VP_2, which involves a subprovider VP_3, etc. For such a composition it suffices to connect the communication interfaces in the appropriate way (see Fig. 1):

- 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).

Such a sequential composition allows one to configure mediator schemes (see Fig. 4) 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. More complex composition schemes can be defined similarly.

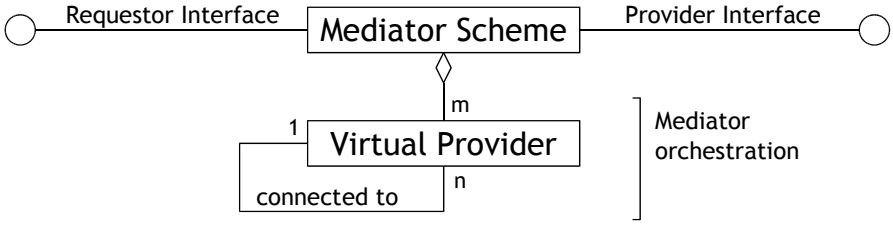

Fig. 4. Mediator Scheme

4.2 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* on the provider side (the left hand side in Fig. 1) or *OutReqMssg* and *InAnswMssg* on the requester side of a *VP* (the right hand side in Fig. 1). In the first case this comes up to unfold the function *originator*, which for an *outAnswMsg* yields the *inReqMsg* to which *outAnswMsg* 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* = VirtualProvider as (based upon) the correspondence between any *inReqMsg* and the *outAnswMsg* related to it by the *originator* function:

$$originator(outAnswMsg) = inReqMsg$$

Two virtual providers VP, VP' can be considered equivalent if an equivalence relation $ServiceBehavior(VP) \equiv ServiceBehavior(VP')$ holds between their service behaviours. To concretely define such an equivalence involves detailing of the meaning of service 'requests' and provided 'answers', which comes up to providing further detail of the abstract *VP* model in such a way that the intended 'service' features and how they are 'provided' by *VP* become visible in concrete locations.

On the basis of such definitions one can then formally define different *VP*s to be alternatives for a Strategy pattern [12, p. 315] for providing requested services. For the run-time selection of mediators, any suitable provider interface can be viewed as one of the implementations ("mediator orchestration") of a Strategy pattern assigned to a requester interface. This provides the basis for investigating questions like: How can one 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?

5 Illustration: Virtual Internet Service Provider

One of the use cases in the DIP project (see http://dip.semanticweb.org) deals with a *Virtual Internet Service Provider* (VISP). A VISP resells products that are bundled from offerings of different providers. A typical example for such a product bundle is an *Internet presence* including a personal web server and a personal e-mail address, both bound to a dedicated, user-specific domain name, 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. We assume now that this interface contains only one request message *RegisterDomain*, requiring four input parameters:

- `DomainName`, the name of the new domain that should be registered
- `DomainHolderName`, the name of the domain owner
- `AdministrativeContactName` the name of the domain administrator
- `TechnicalContactName`, the name of the technical contact

On successful registration, the answer will contain four so-called RIPE-Handles,[11] uniquely identifying in the RIPE database the four names provided in the request message. We skip the obvious instantiation of VirtualProvider to formalize this VISP.

5.1 A possible VP refinement for *RegisterDomain*

We now consider the case that the VISP is extending it's business into a new country whose domain name registry authority implements a different interface for registering new domain names, say consisting of four request messages:

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

A VP instance for that scenario is depicted in Fig. 5.[12] Within this VP, the incoming request *RegisterDomain* is split into a sequence of two subrequests. The first subrequest is further divided into three parallel subrequests, each registering one of the 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 notational convention of appending *Obj* when referring to the internal representations of the different requests, we formalize this VP instance by the following stipulations. We start with refining the Initialize ASM:

[11] RIPE stands for "Réseaux IP Européens", see http://ripe.net.
[12] We use mnemonic abbreviations for the request message and parameter names.

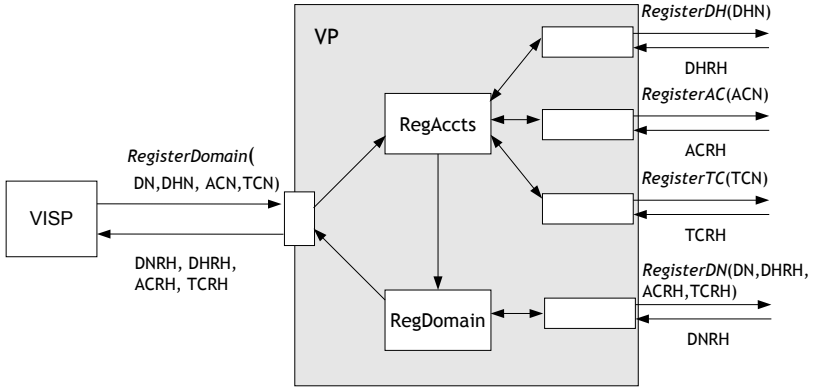

Fig. 5. VP instance

$\textsc{Initialize}(\mathit{RegisterDomainObj}, \mathit{RegisterDomain}(\texttt{DN}, \texttt{DHN}, \texttt{ACN}, \texttt{TCN}) =$
$\mathit{params}(\mathit{RegisterDomainObj}) := \{\texttt{DN}, \texttt{DHN}, \texttt{ACN}, \texttt{TCN}\}$
$\mathit{SeqSubReq}(\mathit{RegisterDomainObj}) := \{\mathit{RegAccnts}, \mathit{RegDomain}\}$
$\mathit{FstSubReq}(\{\mathit{RegAccnts}, \mathit{RegDomain}\}) := \mathit{RegAccnts}$
$\mathit{NxtSubReq}(\{\mathit{RegAccnts}, \mathit{RegDomain}\}, \mathit{RegAccnts}, -) := \mathit{RegDomain}$
$\mathit{NxtSubReq}(\{\mathit{RegAccnts}, \mathit{RegDomain}\}, \mathit{RegDomain}, -) := \mathit{nil}$
$\mathit{FstParReq}(\mathit{RegAccnts}, \mathit{RegisterDomainObj}) :=$
$\{\mathit{RegisterDH}(\texttt{DHN}), \mathit{RegisterAC}(\texttt{ACN}),$
$\mathit{RegisterTC}(\texttt{TCN})\}$
$\mathit{NxtParReq}(\mathit{RegDomain}, \mathit{RegisterDomainObj}, \mathit{AS}) :=$
$\{\mathit{RegisterDN}(\texttt{DN}, \mathit{handle}(\mathit{DHRHObj}),$
$\mathit{handle}(\mathit{ACRHObj}), \mathit{handle}(\mathit{TCRHObj})\}$
$\mathit{AnswToBeSent}(\mathit{RegisterDomainObj}\ ) := \mathit{true}$
$\mathit{ToBeAnswered}(\{\mathit{RegisterDH}, \mathit{RegisterAC}, \mathit{RegisterTC}\}) :=$
$\{\mathit{RegisterDH}, \mathit{RegisterAC}, \mathit{RegisterTC}\}$
$\mathit{ToBeAnswered}(\{\mathit{RegisterDN}\}) := \{\mathit{RegisterDN}\}$
$\mathit{status}(\mathit{RegisterDomainObj}) := \mathit{started}$

where

$\mathit{AS} = \{\mathit{DHRHObj}, \mathit{ACRHObj}, \mathit{TCRHObj}\}$

$$\mathit{handle}(X) = \begin{cases} \texttt{DHRH}\ \mathit{if}\ X = \mathit{DHRHObj} \\ \texttt{DNRH}\ \mathit{if}\ X = \mathit{DNRHObj} \\ \texttt{ACRH}\ \mathit{if}\ X = \mathit{ACRHObj} \\ \texttt{TCRH}\ \mathit{if}\ X = \mathit{TCRHObj} \end{cases}$$

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

$\mathit{Combine}(\mathit{RegisterDomainObj}) =$
$\mathit{AnswerSet}(\mathit{RegAccnts}) \cup \mathit{AnswerSet}(\mathit{RegDomain})$

Function *answer* maps an incoming message to its internal representation:

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

The abstract function *Formatted* is used to transform the parameters into the format expected by the requester, in our case the VISP:

$$outAnsw2Msg(\{DHRHObj, DNRHObj, ACRHObj, TCRHObj\}) = \\ Formatted(\{\texttt{DNRH}, \texttt{DHRH}, \texttt{ACRH}, \texttt{TCRH}\})$$

In [13] we give five other simple examples for refinements of VP to capture the execution semantics of some workflow patterns discussed in [14].

5.2 Proving Properties for VPs

Once one has a mathematical model of VPs, this can be used to prove properties of interest for the model and its refinements to executable code. We illustrate this by a proof sketch that the two VISPs defined above are equivalent.

The claim follows if we can show the correctness of both VPs with respect to the requested service, namely that any successful initial *inReqMsg* to $RegisterDomain(DN, DHN, ACN, TCN)$ will receive an *outAnswMsg* containing four RIPE-Handles, one for each of the $RegisterDomain(DN, DHN, ACN, TCN)$ parameters. For the first VP this is trivial under the assumption that the (sub)provider provides real RIPE handles as answers to $RegisterDomain(DN, DHN, ACN, TCN)$ requests. For the refined VP, the claim can be stated more precisely by saying that the following holds for every successful pair of *inReqMsg* and corresponding *outAnswMsg* (the correspondence is formally established by their belonging to one *reqObj* in VP, successful refers to the fact that in the example VP we consider only the case of successful registrations, without further interaction between requester and mediator):

CorrectnessLemma.
For corresponding successful *inReqMsg*, *outAnswMsg* holds :
$RIPE\text{-}Handle(DomainName(inReqMsg)) = \\ DomainNameRipeHan(outAnswMsg)$
$RIPE\text{-}Handle(DomainHolderName(inReqMsg)) = \\ DomHolderNameRipeHan(outAnswMsg)$
$RIPE\text{-}Handle(AdminContactName(inReqMsg)) = \\ AdmContactNameRipeHan(outAnswMsg)$
$RIPE\text{-}Handle(TecContactName(inReqMsg)) = \\ TecContactNameRipeHan(outAnswMsg)$

Here the function *RIPE-Handle* denotes a real-life RIPE handle, which uniquely identifies its argument name in the RIPE database. *DomainNameRipeHan*, etc.

denote projection functions, which extract the corresponding information from the $outAnswMsg = Formatted(\{\texttt{DNRH}, \texttt{DHRH}, \texttt{ACRH}, \texttt{TCRH}\})$.

Proof. A simple analysis of VISP runs shows that an incoming request message $RegisterDomain(DN, DHN, ACN, TCN)$ triggers VP to SEND first three subrequests $RegisterDH(DHN)$, $RegisterAC(ACN)$, $RegisterTC(TCN)$, which are (assumed to be) answered by RIPE handles $DHRH$, $ACRH$, $TCRH$. Then VP SENDs the subrequest $RegisterDN(DN, DHRH, ACRH, TCRH)$, which is (assumed to be) answered by a domain name ripe handle $DNRH$. By definition of the *answer* function, the *outAnswMsg* contains a *Formatted* version of the four RIPE handles obtained for the parameters in the *inReqMsg*, from where the projection functions extract these RIPE handles.

We want to stress that the proof works only under the assumption that the subproviders work correctly, i. e. that they provide upon request ripe handles for domain holder names, administrative contact names, technical contact names and domain names. This is the best one can prove for VP, which is only a mediator and relies for the correctness of the provided service upon the correctness of its subproviders.

6 Conclusions and Future Work

Our formal, high-level ASM model of process mediation provides a basis for *"communicating and documenting design ideas"* and supports *"an accurate and checkable overall understanding"* of the controversially discussed topic of process mediation, a part of the Semantic Web Services (SWS) usage process [4,5,6]. ASM models can help to provide *explicit, exact and formal specifications* with an accurate meaning of all underlying terms, needed to produce a consistent view of the general SWS usage process. Furthermore, the ASM method allows to *"isolate the hard part of a system"* [9, p. 14-15] and thus to concentrate on the essential parts for refinement, targeted at bridging controversial approaches, like dynamic composition vs. static composition, through explicitly showing their differences by deriving them as different refinements of the same abstractions. We look for more involved practical instantiations of VP than the simple one illustrated in Sect. 5. Another direction of research concerns replacing the simple communication patterns used by VP by more complex ones. RECEIVEREQ and SENDANSW are identified in [15] as basic bilateral service interaction patterns, namely as mono-agent ASM modules RECEIVE and SEND; the FEEDSENDREQ submachine together with SENDREQ in PROCESS realize an instance of the basic multilateral mono-agent service interaction pattern called ONETOMANYSEND in [15], whereas the execution of RECEIVEANSW in ITERATESUBREQPROCESSG until *AllAnswersReceived* is an instance of the basic multilateral mono-agent ONEFROMMANYRECEIVE pattern from [15]. One can refine VP to concrete business process applications by enriching the communication flow structure built from basic service interaction patterns as analysed in [15].

Besides the mediation and composition topics, VP has proven to be useful as a basis for formal specifications of distributed semantic discovery frameworks.

As shown in [16], only minor changes on the VP structure are required in order to specify a formal, high-level ASM model of distributed semantic discovery services. The different distribution and semantic matchmaking strategies, depending on the technology used for an implementation of a discovery service, can be derived as different refinements of the same abstractions.

References

1. Stumptner, M.: Configuring web services. In: Proceedings of the Configuration Workshop at the 16th European Conference on Artificial Intelligence (ECAI). (2004) 10–1/10–6
2. Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P.: Planning and monitoring web service composition. In: AIMSA. (2004) 106–115
3. Lee, Y., Patel, C., Chun, S.A., Geller, J.: Compositional knowledge management for medical services on semantic web. In: WWW (Alternate Track Papers & Posters). (2004) 498–499
4. Bornhövd, C., Buchmann, A.: Semantically meaningful data exchange in loosely coupled environments. In: Proceedings of the International Conference on Information Systems Analysis and Synthesis (ISAS). (2000)
5. Fensel, D., Bussler, C.: The web service modeling framework wsmf. Electronic Commerce Research and Applications **1** (2002) 113–137
6. Pires, P.F., Benevides, M.R.F., Mattoso, M.: Building reliable web services compositions. In: Web, Web-Services, and Database Systems. (2002) 59–72
7. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases **14** (2003) 5–51
8. Barros, A., Dumas, M., ter Hofstede, A.: Service interaction patterns: Towards a reference framework for service-based business process interconnection. Technical report, Faculty of IT, Queensland University of Technology (2005)
9. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer (2003)
10. Börger, E.: The ASM refinement method. Formal Aspects of Computing **15** (2003) 237–257
11. Barros, A., Dumas, M., Oaks, P.: A critical overview of the web services choreography description language (WS-CDL). White paper (2005)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
13. 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 (2005) ISSN 1613-0073, online CEUR-WS.org/Vol-134/lemcke-wiw05.pdf.
14. Cimpian, E., Mocan, A.: D13.7 v0.1 Process mediation in WSMX – WSMX working draft (2005) http://www.wsmo.org/TR/d13/d13.7/v0.1/.
15. Barros, A., Börger., E.: A compositional framework for service interaction patterns and communication flows. In: Proc. 7th International Conference on Formal Engineering Methods (ICFEM). LNCS, Springer (2005)
16. Friesen, A.: A high-level specification for semantic web service discovery framework. In preparation (2005)