Egon Börger (Pisa) & Alexander Raschke (Ulm)

A Virtual Provider Model

for Web Service Mediation and Discovery

Università di Pisa, Dipartimento di Informatica boerger@di.unipi.it
Universität Ulm, Abteilung Informatik alexander.raschke@uni-ulm.de

See Ch. 5.1 of Modeling Companion
http://modelingbook.informatik.uni-ulm.de

Problem Context and Goal

- Web applications are truly distributed systems of heterogeneous components
  - which interact via the internet (communicating ASMs)
- Resulting challenge for service-oriented system development: how to provide accurate formulation and documentation of
  - system design structure
  - system properties and their verification

  as support for checkably correct understanding by the variety of stakeholders (domain experts, designers, programmers, users) of quickly changing systems for service-oriented computations

Goal: define a precise, abstract, compositional mediator model

- for message-based interactions of heterogeneous systems
- suitable for composition of concurrent web services

A High-Level Virtual Provider Model

From: Altenhofen,Boerger,Lemcke: J.BPIM 2006 & European/US Patent

- defines a programming lg independent concept of mediation for configuring/composing message-based interactions of web services
  - to establish agreed level of component communication
- instantiates to current mediation concepts via ASM refinements, supporting ‘design for change’
- offers accurate practical composition methods
- provides a basis for rigorous (e.g. equivalence) definitions supporting
  - refinements to service discovery algorithms and selection procedures
  - proofs of concurrent run properties of interest
- offers abstractions for data (*state*) and their transformations (*behavior*) beyond pure message sequencing or control flow analysis
- uses one bilateral and one multilateral interaction pattern, both compatible with widely used communication mechanisms

Role of the Virtual Provider

Request-Reply Pattern: mediator stays bw participants of an interaction where, in a concurrent run, a *requestor* sends a *request* to a *provider* which is supposed to provide and return an *answer*.

VP (Virtual Provider)

- receives requests
- forwards requests to potential actual providers
- collects answers
- constructs out of (possibly a subset of) answers a final answer
- sends the final answer to the requestor

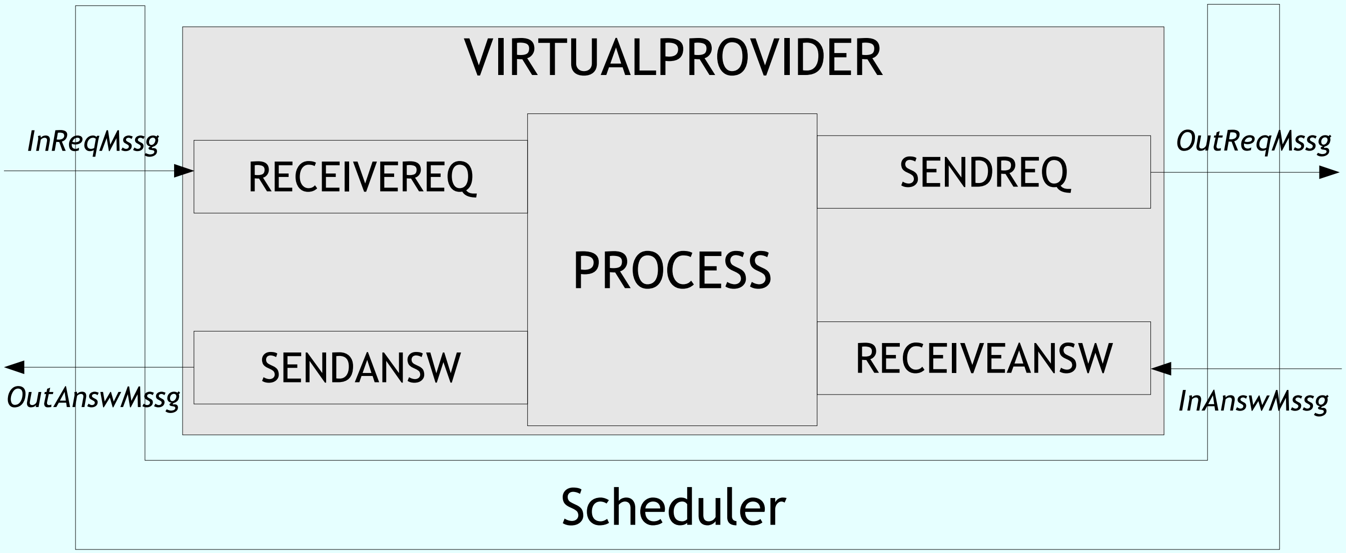
Idea: separate communication from VP internal PROCESSing

This leads to the following VP architecture:[1]

---

[1] Figure © 2006 Springer-Verlag Berlin Heidelberg, reused with permission

VirtualProvider **architecture**



VP *components* : -- 5 agents, managed by a scheduler

- ReceiveReq -- receiving request messages from clients
- SendAnsw -- sending answer messages back to clients
- Process -- handle ReceivedRequests
- SendReq -- sending request messages to (sub-) providers
- ReceiveAnsw -- receiving answer messages from (sub-) providers

Defining VP and its Send components

Leaving scheduling to orthogonal design decisions:

$\textsc{VirtualProvider} =$ **one of**

$\{\textsc{Process},$

$\textsc{ReceiveReq}, \textsc{SendReq}, \textsc{ReceiveAnsw}, \textsc{SendAnsw}\}$

Communication components are communication pattern instances:

$\textsc{SendReq} = \ \textsc{SendAnsw} = \ \textsc{SendPattern}$

leaving the variation of Send parameters (about acknowledgements, discarding or buffering, etc.) to later design decisions

- specification of the VP and its understanding are independent of the details of the $\textsc{SendPattern}$ definition
  - see Sect.4.4.1.1 of the Modeling Companion Book for a precise definition

ReceiveReq **component: instance of** ReceivePattern

- ReceiveReq is defined as ReceivePattern instance.
  - see Sect.4.4.1.2 of the Modeling Companion Book for a precise definition of the parameters
- The parameter variations are left to later design decisions.

For VP we use only that ReceiveReq, as instance of the ReceivePattern, contains the following rule

if *ReadyToReceive* **then** Receive

where the predicate *ReadyToReceive* and the machine Receive are tailored for receiving VP request messages.

Tailoring $\textsc{ReceiveReq}$ for VP

The following concretizations are stipulated:

- a run constraint to *filter out not-genuine 'req msgs'*:

  if $\mathit{ReadyToReceive}(m)$ **then** $m \in \mathit{InReqMssg}$

- *requests are recorded internally* for further elaboration:

```
$\textsc{Receive}(m) =$
    $\textsc{CreateReqObj}(m)$          -- internal request representation
    $\textsc{Consume}(m)$
where
    $\textsc{CreateReqObj}(m) =$
        let $r =$ new $(\mathit{ReqObj})$ in $\textsc{Initialize}(r, m)$
    $\textsc{Initialize}(r, m) =$
        $\mathit{status}(r) := \mathit{start}$
        $\mathit{reqMsg}(r) := m$
```

ReceiveAnsw reflects specific Process Requirements

The ReceivePattern instance ReceiveAnsw must capture the following VP requirements (requested by SAP):

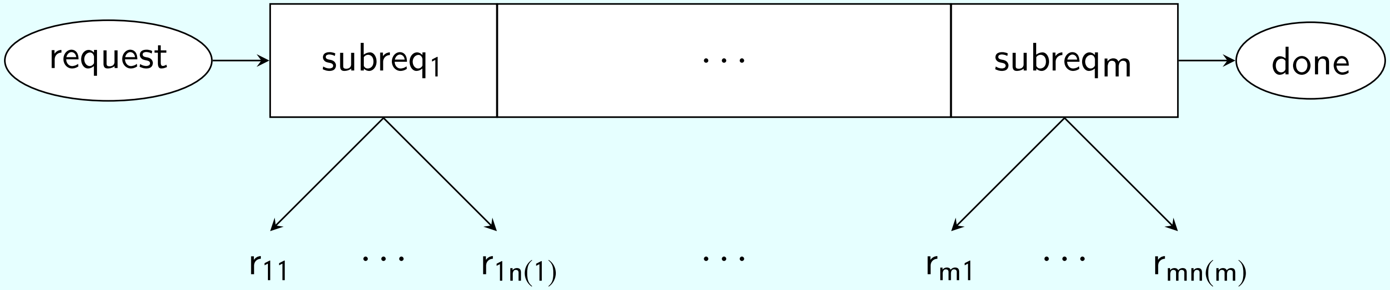
- each arriving request can trigger a *sequence* of (sub)requests
  - forwarded to and to be answered by subproviders before proceeding to the next subrequest, until the final answer can be compiled
- each subrequest may consist of *multiple independent* (subsub)requests
- next sequential subrequest may depend on received answers to the subsubrequests of the current sequential subrequest

Thus requests viewed as root of an alternating *seq/par tree*:

- each subrequest (seq-subtree node) may be root of a tree of subsubrequests (par-subtree nodes)

NB. Sophisticated hierarchical subrequest structures can be obtained by appropriate compositions (nesting) of VPs (see below)

Seq-Par-Tree request structure



- For each request object $req \in ReqObj$ a sequence $seqSubReq(req)$ of one-after-the-other to be processed subrequests $subreq_i \in SubReq$ $(1 \leq i \leq m)$.
- For each $subreq_i \in SubReq$ a set $parSubReq(subreq_i) \subseteq ParReq$ of subsubrequests r_{ij} $(1 \leq j \leq n(i))$ which are sent out in parallel to other providers.

NB. $seqSubReq$ and $parSubReq$ may be dynamic.[2]

---

[2] © 2006 Springer-Verlag Germany, reused with permission

Seq-Par-Tree structure determines $\textsc{ReceiveAnsw}$

As for $\textsc{ReceiveReq}$, also for $\textsc{ReceiveAnsw}$ we use only that, as instance of the $\textsc{ReceivePattern}$, it contains the following rule

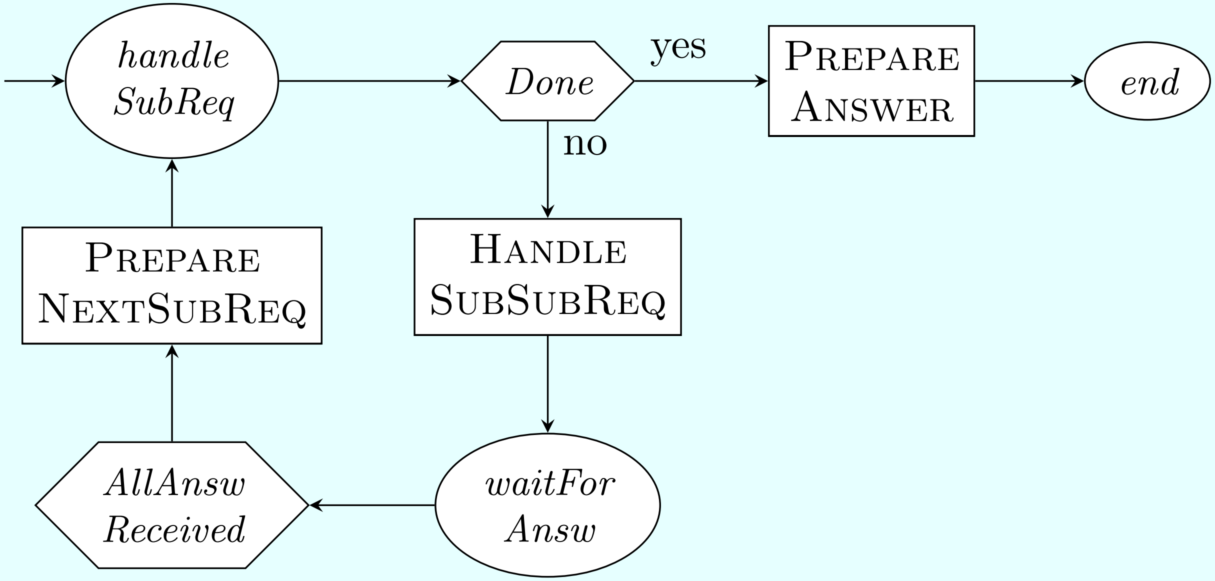
```
if $ReadyToReceive$ then $\textsc{Receive}$
  where
    if $ReadyToReceive(m)$ then $m \in InAnswMssg$
                                 -- filters out not-genuine ‘answ msgs’
    $\textsc{Receive}(m) =$
      $\textsc{Insert}(m, AnswerSet(subRequestor(m)))$
      $\textsc{Consume}(m)$
```

- answer msgs to each subrequest s are collected in $AnswerSet(s)$
- $subRequestor(m)$ identifies subrequest to which m provides an answer

PROCESS delegates to HANDLESUBREQ

```
$\textsc{Process} =$ choose $r \in ReqObj$ with $status(r) = start$
    $\textsc{CreateSubReqHandler}(r)$
    $\textsc{Initialize}(AnswerSet(r))$                          -- to $\emptyset$
where $\textsc{CreateSubReqHandler}(r) =$
    let $a =$ new $(Agent)$                -- delegate processing incoming req
        $\textsc{Initialize}(a, r)$
        $pgm(a) := \textsc{HandleSubReq}$
$\textsc{Initialize}(a, r) =$                              -- record relevant data
    $handler(r) := a$
    $req(a) := r$
    $subReq(a) := head(seqSubReq(r))$                  -- current subrequest
    $status(r) := handleSubReq$         -- start mode of $\textsc{HandleSubReq}$
```

HANDLESUBREQ iterates through *seqSubReq(req(a))*



- HANDLESUBSUBREQ makes subsubrequests *readyToSend*
- then $handler(r) = a$ must *waitForAnswers*
  - inserted by RECEIVEANSW into $AnswerSet(subReq(a))$
- must PREPARENEXTSUBREQuest when *AllAnswReceived*[3]

---

[3] Figure © 2018 Springer-Verlag Germany, reprinted with permission

Components of HandleSubReq

$\textsc{HandleSubSubReq} =$

$\quad \textsc{PrepareBroadcast}(parSubReq(subReq))$

$\quad \textsc{Initialize}(AnswerSet(subReq))$ -- to $\emptyset$

$\textsc{PrepareBroadcast}(S) =$

$\quad$ **forall** $s \in S$ $readyToSend(outReq2Msg(s)) := true$

-- $readyToSend$ for component $\textsc{SendReq}$

$AllAnswReceived$ **iff** -- maybe only some answers needed

$\quad$ **forall** $q \in toBeAnswered(parSubReq(subReq))$

$\qquad$ **forsome** $m \in AnswerSet(subReq)$ $IsAnswer(m, q)$

$Done$ **iff** $subReq = done$ -- NB $done \notin SubReq$

$\textsc{PrepareAnswer}$

$handler(r)$ accumulates in $AnswerSet(r)$ the $AnswerSet(s)$ of answers to sequential subrequests $s \in seqSubReq(r)$

$\textsc{PrepareNextSubReq} =$

$$subReq := next(subReq, seqSubReq(req), AnswerSet(subReq))$$

$$\textsc{Add}(AnswerSet(subReq), AnswerSet(req(\textbf{self})))$$

When $Done$ the handler a must $\textsc{PrepareAnswer}$

- using accumulated $AnswerSet(req_a)$ to compute the $answer$
- to transform it to a msg in the format required for $OutAnswMssg$

$\textsc{PrepareAnswer} =$

$$readyToSend(outAnsw2Msg(answer(req, AnswerSet(req))))$$

$$:= true$$

Defining Mediator Equivalence

- Definition of ServiceBehavior for $\textsc{VirtualProvider}$ instances

$$
\begin{aligned}
&\mathit{ServiceBehavior}(\mathit{VP}) = \\
&\quad \{(\mathit{inReqMssg}, \mathit{outAnswerMssg}) \mid \\
&\qquad \mathit{IsAnAnswer}(\mathit{outAnswerMssg}, \mathit{inReqMssg})\} \\
&\quad \textbf{where } \mathit{IsAnAnwer}(\mathit{answer}, \mathit{request}) \textbf{ iff} \\
&\qquad \textbf{forsome } \mathit{handler} \ \ \mathit{reqMsg}(\mathit{req}(\mathit{handler})) = \mathit{request} \textbf{ and} \\
&\qquad\quad \mathit{handler} \text{ in its last step did } \textsc{PrepareAnswer} \\
&\qquad\qquad \text{with argument } \mathit{answer}
\end{aligned}
$$

- Definition of *Service Equivalence*

$$
\begin{aligned}
&\mathit{VP} \equiv \mathit{VP}' \text{ iff} \\
&\quad \mathit{ServiceBehavior}(\mathit{VP}) \equiv \mathit{ServiceBehavior}(\mathit{VP}')
\end{aligned}
$$

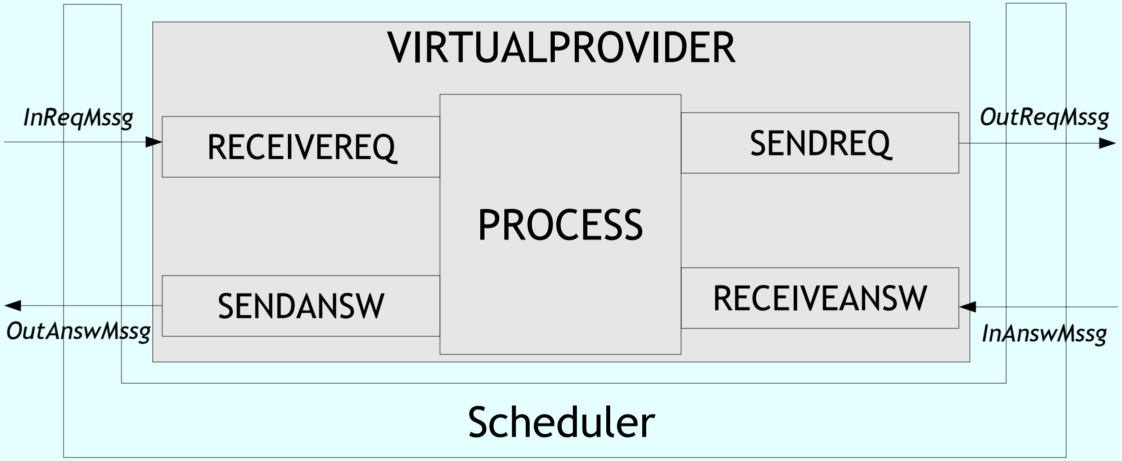
where the equivalence of ServiceBehavior can be defined in terms of message contents extracted from $\mathit{InReqMssg}$ and $\mathit{OutAnswMssg}$

– opens space for practical, not syntax-based but content-driven precise semantical equivalence concepts and their mathematical analysis

Functional VP Composition $VP_1 \dots VP_n$

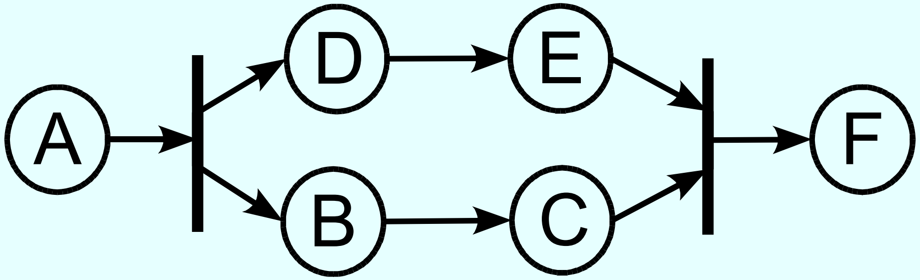
by connecting the communication interfaces:

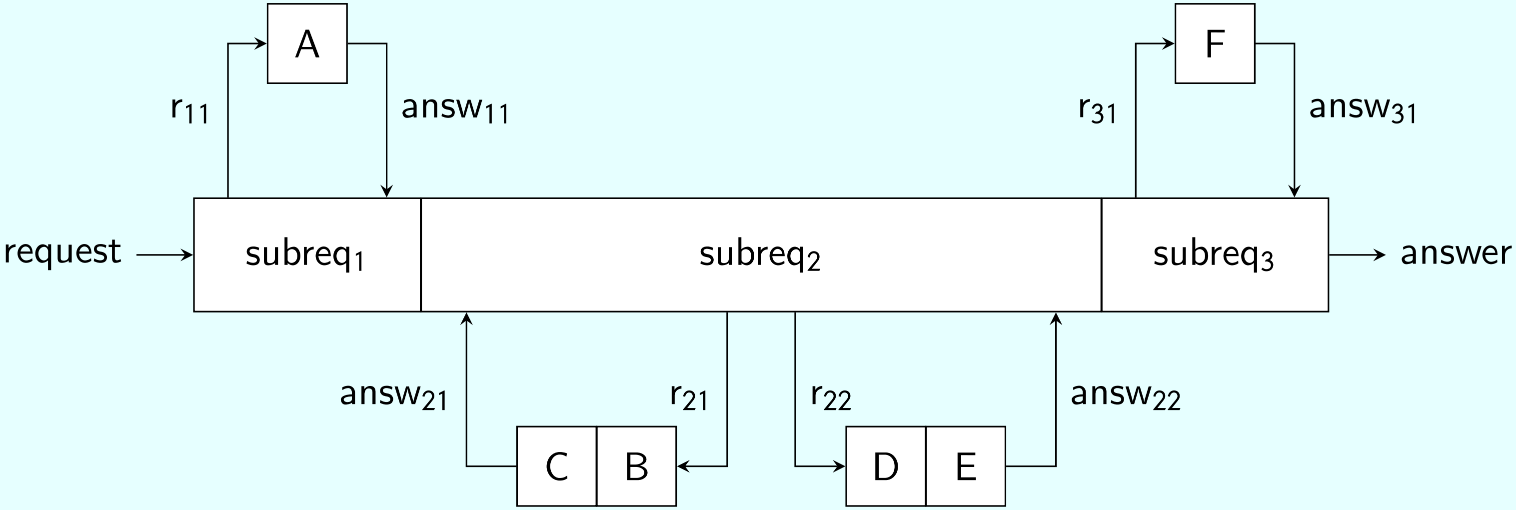
- SendReq of VP_i to ReceiveReq of VP_{i+1}
  - data mediation bw VP_i-OutReqMssg and VP_{i+1}-InReqMssg
- SendAnsw of VP_{i+1} to ReceiveAnsw of VP_i
  - data mediation bw VP_{i+1}-OutAnswMssg and VP_i-InAnswMssg



Together with seq/par tree structure this VP composition provides simple descriptions of sophisticated web service interaction patterns.

Modular VP composition for control flow structures





[4]

---

[4] © 2018 Springer-Verlag Berlin Heidelberg, reprinted with permission

Stateful refinement of VirtualProvider

- Refine VP by an internal state component
  - for recording request data to relate additions to previous requests

```
$\textsc{ReceiveReqStateful} = \textsc{ReceiveReq}$
  where
$\textsc{Receive}(m) =$
  if $\mathit{NewRequest}(m)$ then $\textsc{Receive}_{\textsc{ReceiveReq}}(m)$
  else
    let $r = \mathit{prevReqObj}(m)$ in
      $\textsc{RefreshReqObj}(r, m)$
      $\textsc{Consume}(\textsc{m})$
```

For a refinement to capture distributed web service discovery see Friesen/Börger 2006 (reference below).

Exl: VIRTUALPROVIDER as interface adapter

Let VISP be a Virtual Internet Service Provider which serves $InternetDomain$ registration requests.

Assume the following request parameters:

- $DomainName$ for the new to-be-registered domain,
- $DomainHolderName$ of the legal domain owner,
- $AdministrativeContactName$ of the domain administrator,
- $TechnicalContactName$ of the person to be contacted for technical issues.

Assume any request $InternetDomain(DN, DHN, ACN, TCN)$ gets an $answer \in OutAnswMssg$ containing four RIPE-Handles

- Réseaux IP Européens

uniquely identifying the four request message parameters in the RIPE database.

Adaptation of VISP to new interface

Consider a domain name registry authority which implements a different interface for registering new domain names, say consisting of four request messages (instead of one):

- $RegisterDH(DomainHolderName)$
- $RegisterAC(AdministrativeContactName)$
- $RegisterTC(TechnicalContactName)$
- $RegisterDN$ with parameters $DoName, DHRipeHandle, ACRipeHandle, TCRipeHandle$

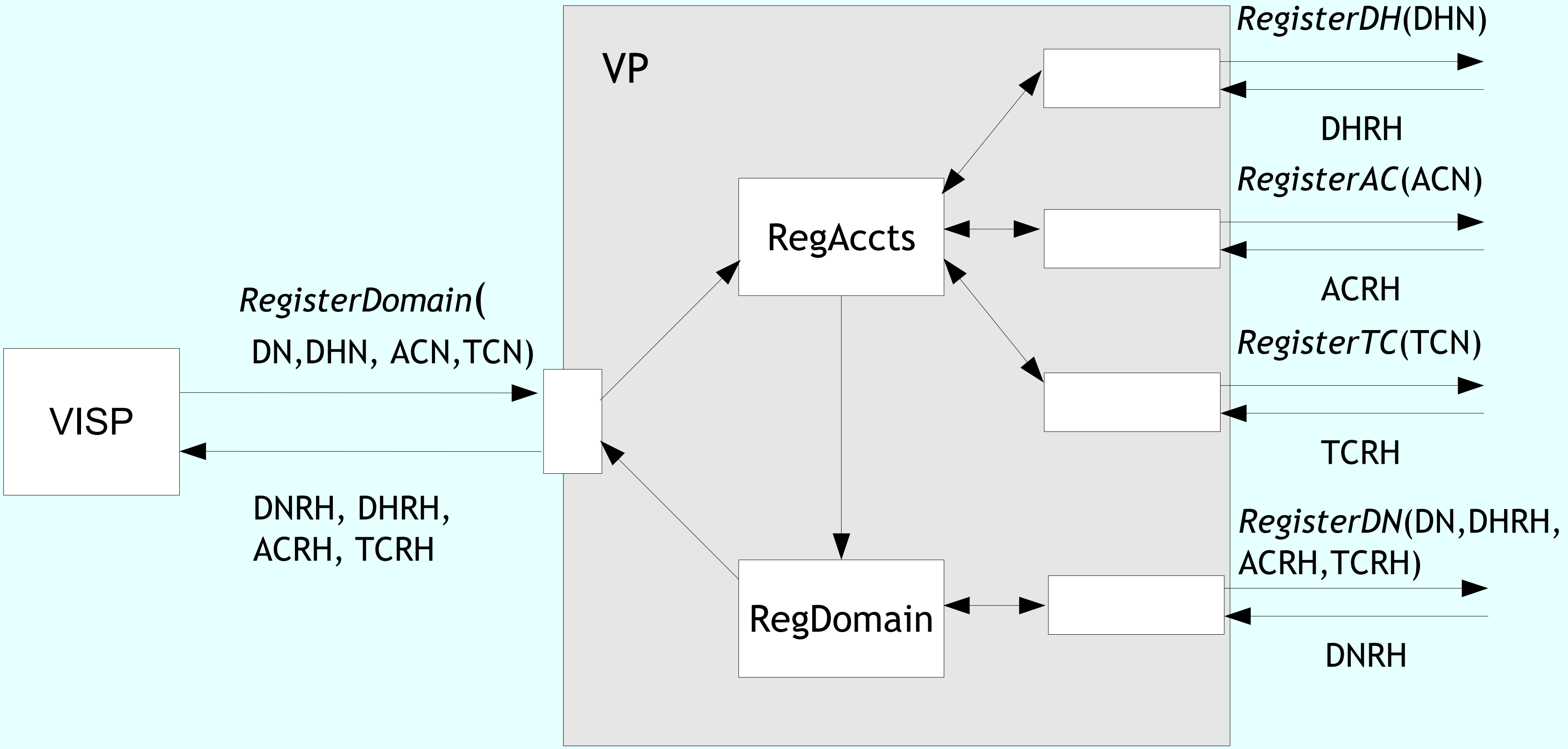
We configure a VIRTUALPROVIDER instance linking it to VISP without changing its internal structure

Subprovider structure

- incoming $RegisterDomain$ request is split up into a sequence $seqSubReq(RegisterDomain)$ of two subrequests
  - $RegAccts$ has a set $parSubReq(RegAccts)$ of three parallel subsubrequests, each registering one of the indicated contacts
  - when $AllAnswReceived$ for these parallel subsubrequests, the second sequential subrequest $RegDomain$ is sent out
- request message for $RegDomain$ is constructed from:
  - $AnswerSet(RegAccts)$ of the first subrequest $RegAccts$ and
  - $DomainName$ parameter DN of the original $RegisterDomain$ request
- finally $\textsc{PrepareAnswer}$ triggers outgoing answer message to be sent by the subprovider back to $\textsc{Visp}$

By assumption $\textsc{Visp}$, from the received data, can build its answer msg to the user who sent the initial $InternetDomain$ registration request

VIRTUALPROVIDER **instance to adapt** VISP



[5]

---

[5] © 2006 Springer-Verlag Berlin Heidelberg, reprinted with permission

References

- E. Börger and A. Raschke: Modeling Companion for Sw Practitioners. Springer 2018. `http://modelingbook.informatik.uni-ulm.de`
- M. Altenhofen, E. Börger, A. Friesen, J. Lemcke: A High-Level Specification for Virtual Providers. Intern. J.BPIM (2006) pp. 267-278
- M. Altenhofen and E. Boerger and J. Lemcke: System and a method for mediating within a network. US Patent US7984188 B2 (July 19, 2011. `https://www.google.de/patents/US7984188`
- A. Barros, E. Börger: A compositional framework for service interaction patterns and communication flows. LNCS 3785 (2005) pp. 5-35
- A. Friesen and E. Börger: A High-Level Specification for Semantic Web Service Discovery Services. Proc. 6th Int. Conf. on Web Engeneering (ICWE 2006). ISBN 1-59593-435-9, ACM Digital Library `http://doi.acm.org/10.1145/1149993.1150012`
- E. Börger and A. Cisternino and V. Gervasi: Modeling Web Applications Infrastructure with ASMs. SCP 94(2), 2014, 69-92

Copyright Notice