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


- defines a programming language 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**

[Diagram showing the architecture of VirtualProvider]

**VP components:**
- **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

---

-- 5 agents, managed by a scheduler
Defining VP and its Send components

Leaving scheduling to orthogonal design decisions:

VirtualProvider = one of

{Process, ReceiveReq, SendReq, ReceiveAnsw, SendAnsw}

Communication components are communication pattern instances:

SendReq = SendAnsw = 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 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.
The following concretizations are stipulated:

- **a run constraint to filter out not-genuine ‘req msgs’:**
  
  ```
  if ReadyToReceive(m) then m ∈ InReqMssg
  ```

- **requests are recorded internally** for further elaboration:

  ```
  RECEIVE(m) =
  CREATEREQOBJ(m)
  CONSUME(m)
  -- internal request representation
  ```

  **where**

  ```
  CREATEREQOBJ(m) =
  let r = new (ReqObj) in INITIALIZE(r, m)
  ```

  ```
  INITIALIZE(r, m) =
  status(r) := start
  reqMsg(r) := m
  ```
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)
For each request object \( \text{req} \in \text{ReqObj} \) a sequence \( \text{seqSubReq}(\text{req}) \) of one-after-the-other to be processed subrequests \( \text{subreq}_i \in \text{SubReq} \) \((1 \leq i \leq m)\).

For each \( \text{subreq}_i \in \text{SubReq} \) a set \( \text{parSubReq}(\text{subreq}_i) \subseteq \text{ParReq} \) of subsubrequests \( r_{ij} \) \((1 \leq j \leq n(i))\) which are sent out in parallel to other providers.

NB. \( \text{seqSubReq} \) and \( \text{parSubReq} \) may be dynamic.\(^2\)

\(^2\) © 2006 Springer-Verlag Germany, reused with permission
As for \texttt{ReceiveReq}, also for \texttt{ReceiveAnsw} we use only that, as instance of the \texttt{ReceivePattern}, it contains the following rule

\begin{verbatim}
if ReadyToReceive then Receive
where
  if ReadyToReceive\((m)\) then \(m \in \text{InAnswMssg}\)
    -- filters out not-genuine ‘answ msgs’

Receive\((m)\) =
  Insert\((m, \text{AnswerSet}(\text{subRequestor}(m)))\)
  Consume\((m)\)
\end{verbatim}

- answer msgs to each subrequest \(s\) are collected in \texttt{AnswerSet}(s)
- \texttt{subRequestor}(m) identifies subrequest to which \(m\) provides an answer
**Process delegates to HandleSubReq**

\[
\text{Process} = \text{choose } r \in \text{ReqObj} \text{ with } \text{status}(r) = \text{start}
\]

\[
\text{CreateSubReqHandler}(r)
\]

\[
\text{Initialize}(\text{AnswerSet}(r))
\]

--- to \( \emptyset \)

\[
\text{where } \text{CreateSubReqHandler}(r) =
\]

\[
\text{let } a = \text{new} (\text{Agent})
\]

\[
\text{Initialize}(a, r)
\]

\[
pgm(a) := \text{HandleSubReq}
\]

\[
\text{Initialize}(a, r) =
\]

\[
\text{handler}(r) := a
\]

\[
\text{req}(a) := r
\]

\[
\text{subReq}(a) := \text{head}(\text{seqSubReq}(r))
\]

\[
\text{status}(r) := \text{handleSubReq}
\]

--- start mode of \text{HandleSubReq}

\]

Copyright CC BY–NC-SA 4.0
**HandleSubReq** iterates through $\text{seqSubReq}(\text{req}(a))$

- **HandleSubSubReq** makes subsubrequests readyToSend
- then $\text{handler}(r) = a$ must waitForAnswers
  - inserted by RECEIVEANsw into $\text{AnswerSet}(\text{subReq}(a))$
- must **PrepareNextSubRequest** when $\text{AllAnswReceived}^3$

---

Figure © 2018 Springer-Verlag Germany, reprinted with permission

Copyright CC BY–NC-SA 4.0
Components of \texttt{HandleSubReq}

\texttt{HandleSubSubReq} =
\begin{align*}
\text{\texttt{PrepareBroadcast}(parSubReq(subReq))} \\
\text{\texttt{Initialize}(AnswerSet(subReq))} \quad \text{-- to } \emptyset
\end{align*}

\texttt{PrepareBroadcast}(S) =
\begin{align*}
\text{\forall s \in S \text{ readyToSend}(outReq2Msg(s)) := true} \\
\text{-- readyToSend for component \texttt{SendReq}}
\end{align*}

\texttt{AllAnswReceived} \text{ iff} 
\begin{align*}
\text{\forall q \in toBeAnswered(parSubReq(subReq))} \\
\text{\exists m \in AnswerSet(subReq) \ IsAnswer(m, q)}
\end{align*}

\texttt{Done} \text{ iff} \texttt{subReq = done} \quad \text{-- NB done } \notin \texttt{SubReq}
handler(r) accumulates in AnswerSet(r) the AnswerSet(s) of answers to sequential subrequests s ∈ seqSubReq(r)

\textbf{PrepareNextSubReq} =

\begin{align*}
\text{subReq} & := \text{next}(\text{subReq}, \text{seqSubReq}(\text{req}), \text{AnswerSet}(\text{subReq})) \\
\text{ADD}(\text{AnswerSet}(\text{subReq}), \text{AnswerSet}(\text{req}(&\text{self})))
\end{align*}

When \textit{Done} the handler \textit{a} must \textbf{PrepareAnswer}

- using accumulated \text{AnswerSet}(\text{req}_a) to compute the answer
- to transform it to a msg in the format required for \textit{OutAnswMssg}

\textbf{PrepareAnswer} =

\begin{align*}
\text{readyToSend}(\text{outAnsw2Msg}(\text{answer}(\text{req}, \text{AnswerSet}(\text{req})))) \\
:= \text{true}
\end{align*}
Defining Mediator Equivalence

Definition of ServiceBehavior for VirtualProvider instances

\[ ServiceBehavior(\text{VP}) = \{ (\text{inReqMssg}, \text{outAnswerMssg}) \mid \text{IsAnAnswer}(\text{outAnswerMssg}, \text{inReqMssg}) \} \]

where \text{IsAnAnswer}(\text{answer}, \text{request}) \iff

\text{forsome handler reqMsg}(\text{req(handler)}) = \text{request} \text{ and handler in its last step did PrepareAnswer with argument answer}

Definition of Service Equivalence

\( \text{VP} \equiv \text{VP}' \iff \text{ServiceBehavior(\text{VP}) \equiv ServiceBehavior(\text{VP}')} \)

where the equivalence of ServiceBehavior can be defined in terms of message contents extracted from \text{InReqMssg} and \text{OutAnswMssg} – opens space for practical, not syntax-based but content-driven precise semantical equivalence concepts and their mathematical analysis.
Functional VP Composition $VP_1 \ldots VP_n$

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

A F
B C
D E

subreq

1
subreq

2
subreq

3

request

answer

r11
answ11
r31
answ31

r21
answ21

r22
answ22

subreq1
subreq2
subreq3

A

F

C

B

D

E

4

© 2018 Springer-Verlag Berlin Heidelberg, reprinted with permission

Copyright CC BY–NC-SA 4.0

18
Stateful refinement of VirtualProvider

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

\[
\text{ReceiveReqStateful} = \text{ReceiveReq}
\]

where

\[
\text{Receive}(m) =
\]

\[
\text{if } \text{NewRequest}(m) \text{ then } \text{ReceiveReceiveReq}(m)
\]

\[
\text{else }
\]

\[
\text{let } r = \text{prevReqObj}(m) \text{ in }
\]

\[
\text{RefreshReqObj}(r, m)
\]

\[
\text{Consume}(m)
\]

For a refinement to capture distributed web service discovery see Friesen/Börger 2006 (reference below).
Let $\text{Visp}$ be a Virtual Internet Service Provider which serves $\text{InternetDomain}$ registration requests.

Assume the following request parameters:

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

Assume any request $\text{InternetDomain}(\text{DN}, \text{DHN}, \text{ACN}, \text{TCN})$ gets an answer $\in \text{OutAnswMssg}$ 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):

- \texttt{RegisterDH(DomainHolderName)}
- \texttt{RegisterAC(AdministrativeContactName)}
- \texttt{RegisterTC(TechnicalContactName)}
- \texttt{RegisterDN \text{ with parameters} \ DoName, DHRipeHandle, ACRipeHandle, TCRipeHandle}

We configure a \texttt{VIRTUALPROVIDER} instance linking it to Visp without changing its internal structure
Subprovider structure

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

By assumption \textit{Visp}, from the received data, can build its answer msg to the user who sent the initial \textit{InternetDomain} registration request
Virtual Provider instance to adapt VISP

```
VirtualProvider instance to adapt VISP

RegisterDomain(DN,DHN, ACN,TCN)
DNRH, DHRH, ACRH, TCRH
RegAccts
RegDomain
RegisterDH(DHN)
DHRH
RegisterAC(ACN)
ACRH
RegisterTC(TCN)
TCRH
RegisterDN(DN,DHRH, ACRH,TCRH)
DNRH
```

Copyright CC BY–NC-SA 4.0
References

- A. Barros, E. Börger: A compositional framework for service interaction patterns and communication flows. LNCS 3785 (2005) pp. 5-35
- E. Börger and A. Cisternino and V. Gervasi: Modeling *Web Applications Infrastructure* with ASMs. SCP 94(2), 2014, 69-92
Copyright Notice

It is permitted to (re-) use these slides under the CC-BY-NC-SA licence

https://creativecommons.org/licenses/by-nc-sa/4.0/

i.e. in particular under the condition that

- the original authors are mentioned
- modified slides are made available under the same licence
- the (re-) use is not commercial