An Architecture for Web Service Mediation and Discovery

Università di Pisa, Dipartimento di Informatica, I-56127 Pisa, Italy
boerger@di.unipi.it

In collaboration with
Michael Altenhofen, Andreas Friesen and Jens Lemcke
SAP Research, Karlsruhe, Germany
Goal

Provide a Programming Language Independent Precise Mediation Model for mediation between message-based interactions of heterogeneous systems. We want the model to be ‘designed for change’:

- refinable (instantiatable) to current mediation concepts
- offering accurate practical composition concepts
- providing a basis for defining rigorous equivalence notions supporting
  - discovery algorithms and service selection procedures in real-life applications
  - proofs of properties of interest in complex mediation schemes
- offering abstractions for both data and data transformations (abstract state and abstract behavior) that go beyond pure message sequencing or control flow analysis
- adaptable to different underlying communication mechanisms
The Method: using Machines operating on Abstract States

- within a single *precise yet simple conceptual framework*

the ASM method naturally supports and uniformly links the major activities occurring during the software life cycle:

- **requirements capture** by constructing rigorous *ASM ground models*, i.e. accurate concise high-level system blueprints (contracts)

- **architectural and component design** bridging the gap between specification and code by *piecemeal, systematically documented detailing* of abstract models via intermediate models to code (*general ASM refinement notion*)

- **validation** of models by their tool-supported *simulation*

- **verification** of model properties by tool-supported *proof techniques*

- **documentation** for *inspection, reuse and maintenance* by providing, through the intermediate models and their analysis, explicit descriptions of the *software structure* and of the major *design decisions*
Variety of applications of ASMs (1)

- **industrial standards**: *ground models* for the standards of
  - OASIS for Business Process Execution Language for Web Services
  - ECMA for C#
  - ITU-T for SDL-2000
  - IEEE for VHDL93
  - ISO for Prolog

- **design, reengineering, testing of industrial systems**:
  - railway and mobile telephony network component software at Siemens
  - fire detection system in German coal mines
  - implementation of behavioral interface specifications on the .NET platform and conformance test of COM components at Microsoft
  - compiler testing and test case generation tools
Variety of applications of ASMs (2)

- **Programming languages**: definition and analysis of the semantics and the implementation for the major real-life programming languages, among many others for example
  - SystemC
  - Java/JVM (including bytecode verifier)
  - Domain-specific languages used at the Union Bank of Switzerland including the verification of numerous compilation schemes and compiler back-ends

- **Architectural design**: verification (e.g. of pipelining schemes or of VHDL-based hardware design at Siemens), architecture/compiler co-exploration

- **Protocols**: for authentication, cryptography, cache-coherence, routing-layers for distributed mobile ad hoc networks, group-membership etc.

- **Modeling e-commerce and web services** (at SAP)
if $ctl\_state = i$ then
  if $cond$ then
    $rule$
    $ctl\_state := j$
  where $cond \equiv input = a$  $rule \equiv output := b$  for FSM

ASMs use parameterized locations and first-order conditions:
- $rule$ = set of updates $f(t_1, \ldots, t_n) := t$
- $cond$ = arbitrary first-order formula
Basic Request Structure: Seq/Par Trees

- each arriving request viewed as root of a *seq/par tree* of subrequests, forwarded to and answered by subproviders
- subrequests (seq-subtree nodes) can be elaborated in sequence — forwarded to and to be answered by subproviders before proceeding to the next subrequest, until the final answer can be compiled
- subrequests may consist of multiple independent subsubrequests (par-subtree nodes)
- next sequential subrequest may depend on received answers to the subsubrequests of the current sequential subrequest

Nestings of such alternating seq/par trees and other more sophisticated hierarchical subrequest structures can be obtained by appropriate compositions of VPs.
Separating Tree Processing and Communication

VP defined as interface with five methods:

- **ReceiveReq** for receiving request messages from clients
- **SendAnsw** for sending answer messages back to clients
- **Process** to handle ReceivedRequests via the seq/par tree of auxiliary subrequests and answers received for them
- **SendReq** for sending request messages to (sub-) providers
- **ReceiveAnsw** for receiving answer messages from (sub-) providers

**MODULE VirtualProvider =**

- **ReceiveReq**
- **SendAnsw**
- **Process**
- **SendReq**
- **ReceiveAnsw**
**Send/Receive Machines (Abstract Msgg Passing)**

\[
\text{ReceiveReq}(in\text{ReqMssg}, \text{ReqObj}) = \\
\text{if } \text{ReceivedReq}(in\text{ReqMssg}) \text{ then} \\
\text{CreateNewReqObj}(in\text{ReqMssg}, \text{ReqObj}) \\
\text{where } \text{CreateNewReqObj}(m, R) = \\
\text{let } r = \text{New}(R) \text{ in } \text{Initialize}(r, m) \\
\text{SendAnsw}(out\text{AnswMssg}, \text{SentAnswToMailer}) = \\
\text{if } \text{SentAnswToMailer}(out\text{AnswMssg}) \text{ then } \text{SEND}(out\text{AnswMssg}) \\
\text{SendReq}(out\text{ReqMssg}, \text{SentReqToMailer}) = \\
\text{if } \text{SentReqToMailer}(out\text{ReqMssg}) \text{ then } \text{SEND}(out\text{ReqMssg}) \\
\text{ReceiveAnsw}(in\text{AnswMssg}, \text{AnswerSet}) = \\
\text{if } \text{ReceivedAnsw}(in\text{AnswMssg}) \text{ then} \\
\text{insert } \text{answer}(in\text{AnswMssg}) \text{ into} \\
\text{AnswerSet(requestor}(in\text{AnswMssg}))
\]
Compositional VP Architecture

Sequential 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 $VP_i$-$InAnswMssg$
Composing VP Mediator Structures: Example

Fig. 0.1.

Fig. 0.2.
The core `PROCESS(currReqObj)` machine

- `currReqObj` yields a sequence of `SubRequests`, to be elaborated by an `Iterator` on `SeqSubReq(currReqObj)`
- `AnswMsg` to the `currReqObject` is compiled from the `AnswerSet(seqReq)` of all answers collected from the subrequests

\[
\text{CompileOutAnswMsg for } o = \\
\text{if } \text{AnswToBeSent}(o) \text{ then } \\
\text{SentAnswToMailer}(\text{outAnsw2Msg}(\text{outAnswer}(o))) := \text{true}
\]
Elaboration of Sequential Subrequests: SubProcessIterator

\text{SubProcessIterator}(\text{currReqObj}) = \\
\text{InitializeIterator}(\text{currReqObj}) \text{ seq} \\
\text{IterateSubReqProcessg}(\text{currReqObj}) \text{ until} \\
\text{FinishedSubReqProcessg}

\text{where}

\begin{align*}
\text{yes}(\text{FinishedSubReqProcessg}) &= \text{compileAnswer} \\
\text{no}(\text{FinishedSubReqProcessg}) &= \text{initStatus}(\text{IterateSubReqProcessg})
\end{align*}

Realizes the sequential part of the hierarchical VP request processing view: each incoming (top level) request object \texttt{currReqObj} triggers the sequential elaboration of a finite number of immediate subrequests, members of a set \texttt{SeqSubReq}(\texttt{currReqObj})
Elaboration of Parallel Subrequests: IterateSubReqProcessg

- each sequential SubRequest triggers forwarding finitely many independent parallel SubRequests and waitingForAnswers
- ReceivedAnswers are collected in the \( \text{AnswerSet(seqSubReq)} \)
- until AllAnswersReceived triggers PROCEEDing to NextSubRequest

\[
\text{FEEDSENDREQ with } ParSubReq(seqSubReq(currReqObj)) = \\
\text{forall } s \in ParSubReq(seqSubReq) \\
\text{SentReqToMailer(outReq2Msg(s))} := true
\]
ConcludeStep =
    if AllAnswersReceived then
        ProceedToNextSubReq
        status(currReqObj) :=
            Nxt(status(currReqObj))
    where Nxt(waitingForAnswers) =
        testStatus(FinishedSubReqProcess)

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

Initialize(AnswerSet(seqSubReq)) =
    AnswerSet(seqSubReq) := ∅
**Adapting Standard Iterator Pattern to** \textit{SeqSubReq} 

\textbf{InitializeIterator}(\textit{currReqObj}) = \\
let \( r = \text{FstSubReq}(\text{SeqSubReq}(\textit{currReqObj})) \) in \\
\textit{seqSubReq} := r \\
\textit{ParSubReq}(r) := \text{FstParReq}(r, \textit{currReqObj}) \\

\textit{FinishedSubReqProcessg} = \\
\textit{seqSubReq}(\textit{currReqObj}) = \text{Done}(\text{SeqSubReq}(\textit{currReqObj})) \\

\textbf{ProceedToNextSubReq} = \text{let} \\
\textit{o} = \textit{currReqObj} \\
\textit{s} = \text{NxtSubReq}(\text{SeqSubReq}(\textit{o}), \text{seqSubReq}(\textit{o}), \text{AnswerSet}(\textit{o})) \text{ in} \\
\textit{seqSubReq}(\textit{o}) := \textit{s} \\
\textit{ParSubReq}(\textit{s}) := \text{NxtParReq}(\textit{s}, \textit{o}, \text{AnswerSet}(\textit{o})) \\

\textit{NxtSubReq} \textit{ and } \textit{NxtParReq} \textit{ may depend on answers accumulated so far}
Definition of *ServiceBehavior*

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

− *originator* is retrievable by `CompileOutAnswMssg` from `currReqObj` if recorded as part of `Initialize` by `CreateNewReqObj(inReqMssg, ReqObj)`

Definition of *Service Equivalence*

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

where the equivalence of ServiceBehavior can be defined in terms of message contents extracted from `InReqMssg` and `OutAnswMssg`

− opens space for practical, not syntax-based but content-driven semantical \(\equiv\)-notions
Refine $VP$ by internal state component
– for recording request and answer data:

\[
\text{RECEIVEREQ} \left( \text{inReqMssg} \right) =
\begin{aligned}
\text{if } \text{ReceivedReq} \left( \text{inReqMssg}, \text{ReqObj} \right) \text{ then } \\
\text{if } \text{NewRequest} \left( \text{inReqMssg} \right) \text{ then } \\
\text{CREATENEWREQOBJ} \left( \text{inReqMssg}, \text{ReqObj} \right) \\
\text{else } \\
\text{let } r = \text{prevReqObj} \left( \text{inReqMssg} \right) \text{ in } \\
\text{REFRESHREQOBJ} \left( r, \text{inReqMssg} \right)
\end{aligned}
\]

NB. This is a simple (but frequently occurring) case of the general ASM refinement concept.
Refinement of VP for Semantic Web Service Discovery

- concept instantiations (data refinement)
- rule extensions

*Concept instantiation*: changing “view” of the abstractions from requests/answers to goals/webservices, formally resulting in the following substitutions:

- \( \text{Req} \rightarrow \text{Goal} \)
- \( \text{Answ}, \text{Answer}, \text{AnswerSet} \rightarrow \{\text{SetofWS, WS}\} \)
- \( \text{Process} \rightarrow \text{ProcessGoal} \)
- \( \text{ParSubReq( seqSubReq( currReqObj) )} \rightarrow \text{ParGoalQuery( currGoalObj) } \)
- \( \text{SentReqToMailer} \rightarrow \text{SentGoalToProvider} \) (in \( \text{SENDGOAL} \))
- \( \text{SentAnswToMailer} \rightarrow \text{SentSetOfWSToRequestor} \) (in \( \text{SENDSETOFWS} \))
- Reducing SubReqSeq to *Singleton* determined by currReqGoal
\textbf{ReceiveGoal}(\textit{inGoalMsg}, \textit{GoalObj}) =
\begin{align*}
\textbf{if } & \textbf{ReceivedGoal}(\textit{inGoalMsg}) \textbf{ then} \\
\textbf{CreateNewGoalObj}(\textit{inGoalMsg}, \textit{GoalObj}) \\
\textbf{where} \\
\textbf{CreateNewGoalObj}(m, R) = \\
\textbf{let } g = \textbf{new}(R) \textbf{ in} \\
\textbf{Initialize}(g, m) \\
\textbf{Initialize}(\textit{SetOfWS}(g)) \\
\textbf{if } \textbf{NewGoal}(g, m) \textbf{ then} \\
\textbf{status}(g) := \textbf{started} \\
\textbf{else} \\
\textbf{status}(g) := \textbf{loopDetected} \\
\textbf{Initialize}(\textit{SetOfWS}(g)) = (\textit{SetOfWS}(g) := \emptyset)
\end{align*}
Detection of loops (receiving a request for an already processed goal) to guarantee that no goal query is serviced twice

Fig. 0.3.
Refined IterateSubReqProcessg for DSP

Typical BreakCondition: timeout. SubReqSeq reduces to singleton, reducing **SubProcessIterator**
Discovery Engine

\[ \text{DiscoveryEngine} = \]
\[ \text{choose } M \in \{ \text{ReceiveGoal}, \text{SendSetOfWS} \} \cup \{ \text{MatchGoal} \} \]

Interface with three main methods:

- **ReceiveGoal** for receiving goal queries from a requestor \(DSP\)
- **SendSetOfWS** for sending sets of found Web services back to the associated \(DSP\)
- **MatchGoal** to handle \(ReceivedGoals\) (elements of a set \(GoalObj\) of internal representations of received goals, say as goal objects), typically by filtering and matching the locally available set of Web services to service the currently handled goal request \(currGoalObj\)
Goal: stepwise reduction of the initial set $inSetOfWS$ of Web services to the final set of goal matching Web services, which is sent to $DSP$

\[
\text{MatchGoal}(currGoalObj) = \\
\text{if } \text{status}(currGoalObj) = \text{started} \text{ then} \\
\text{Prefiltering}(currGoalObj) \\
\text{seq SemanticMatchmaking}(currGoalObj) \\
\text{seq QoSMatchmaking}(currGoalObj) \\
\text{seq} \\
\text{CompileOutSetOfWSMsg from currReqObj} \\
\text{status}(currGoalObj) := \text{deliver}
\]

Prefiltering, SemanticMatchmaking and QoSMatchmaking can be further and independently refined to implement different filtering and matchmaking methods or strategies.
Applications and Future Work

- Evaluate competing approaches in terms of the VP model abstractions
- Implement a VP platform as mediation pattern
- Analyse impact on VP of more general communication patterns
  - \texttt{ReceiveReq} and \texttt{SendAnsw}: basic bilateral service interaction patterns
  - \texttt{FeedSendReq} with \texttt{SendReq}: instance of basic multilateral mono-agent service interaction pattern \texttt{OneToManySend}
  - \texttt{ReceiveAnsw} until \texttt{AllAnswersReceived}: instance of basic multilateral mono-agent \texttt{OneFromManyReceive} pattern
- Formulate and prove properties for practical VP instances
References

- M. Altenhofen and E. Börger and J. Lemcke: An Abstract Model for Process Mediation
  – Proc. ICFEM 2005, Springer LNCS 3785, pp. 81-95
- M. Altenhofen and E. Börger and A. Friesen and J. Lemcke: An High-Level Specification for Virtual Providers
  – International J. for Business Process Integration Management 2006
- A. Barros and E. Börger: A compositional framework for service interaction patterns and communication flows
  – Proc. ICFEM 2005, Springer LNCS 3785, pp. 5-35
- E. Börger: The ASM Refinement Method
- E. Börger and R. F. Stärk: Abstract State Machines