A High-Level Specification for Virtual Providers

M. Altenhofen, A. Friesen, J. Lemcke
SAP Research, CEC Karlsruhe, D-76131 Karlsruhe
E-Mail: {michael.altenhofen, andreas.friesen, jens.lemcke}@sap.com

E. Börger
Università di Pisa,Dipartimento di Informatica, I-56125 Pisa, Italy
E-Mail: boerger@di.unipi.it

Abstract: In this paper 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. We will show how this model can be sucessfully applied to two application scenarios, Web service protocol mediation and Semantic Web Service discovery.

Keywords: Virtual Provider, Mediator, Protocol Mediation, Semantic Web Service Discovery

Reference to this paper should be made as follows: Altenhofen, M., Börger, E., Friesen, A. and Lemcke, J. (2006) 'A High-Level Specification for Mediators', *International Journal on Business Process Integration and Management (IJBPIM)*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Michael Altenhofen has graduated in Computer Science at the University of Karlsruhe and is now a Development Architect with SAP Research. His research topics include Software Engineering, Service-Oriented Architectures, Modelling, and Semantics.

Egon Börger is professor of CS in Pisa (Italy) and has been visiting with IBM, Siemens, Microsoft, SAP. He is (co)author of four books and of over 100 research papers in Logic and in CS. His current research interest is in rigorous methods and their industrial applications for the design and the analysis of computer-based systems.

Andreas Friesen has graduated in Computer Sciences and got his PhD from the University of Siegen and is now a Senior Researcher with SAP Research. His research topics include Service-Oriented Architectures, Security, and Semantic Web Services.

Jens Lemcke has graduated in Computer Science at the University of Rostock and is now a Research Associate with SAP Research. His research topics include Semantic Web Services, Web service composition and their behavioural description.

1 INTRODUCTION

In Service-Oriented Architectures (SOA), message-based interactions promise a new level of flexibility, where the same service offering can be provided by totally different implementations, as long as they adhere to the same interaction protocol. Focusing on interfaces rather than implementations reduces the dependencies between the interacting parties, leading to so-called *loosely-coupled* systems [1], and new, more complex services can be easily composed out of using simpler service-oriented building blocks. This compositional approach can also be used to implement *mediators* [2] that provide solutions in heterogeneous scenarios, both at the data and the protocol/process level [3].

In the past, much attention has been spent on providing specification *languages*, for such type of compositions, like BPEL4WS [4] and WS-CDL [5], but their expressive power is rather high. Also, they tend to focus on implementation issues with procedural descriptions, but lack the possibility to specify useful and interesting system properties.

We specify here an abstract execution model, a *Virtual Provider* (VP), that can be used to describe compositional, message-based interaction scenarios. The proposed model in simpler than those of the languages mentioned above, but still powerful enough to cover a sufficiently large area of interesting application scenarios.

We will start with an informal description of the basic execution model in Sect. 2, and then formalize it using the Abstract State Machines (ASM) [6] modeling framework. In the following two sections, we prove the applicability of the generic model (applying the refinement concept from [7]) to support two concrete application scenarios. Section 4 exemplifies a real-world protocol mediation scenario, where we use the formal model to investigate system properties of interest. In Sect. 5, we extend the abstract VP model to define a high-level model for Semantic Web Service discovery.

2 INFORMAL VIRTUAL PROVIDER MODEL

The general architecture of the Virtual Provider (VP) component is based on the assumption that we mostly deal with two-way conversations using the Request-Reply pattern [1]: One application, normally called the *requestor*, sends a message (*request*) to the second application. This second application, in service-oriented scenarios often called the *provider* returns a message (an *answer* or *reply*) to the requestor. These two-way messages are exchanged via appropriate message *channels*. The Virtual Provider can be seen as an intermediary component in the communication model, intercepting requests from the requestor and forwarding them to the provider and, in turn, collecting replies from the provider and returning them to the requestor. From the requestor's point of view it acts like a provider, hence the name *Virtual Provider*.

Since we're only interested in the core functionality of the VP, we devise the architecture in Fig. 1, where we abstract from, i.e. externalize, scheduling and message-passing functionality.

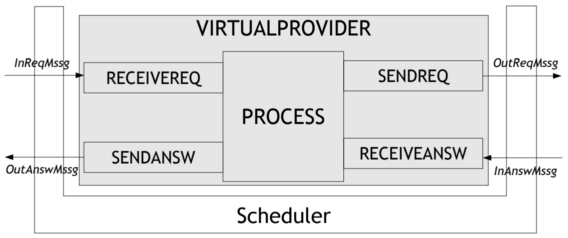

Figure 1: Architecture

For the execution model we build on the assumption underlying the SOA paradigm, that the messages exchanged between reqestor and provider carry all information to establish and maintain context and state, i.e. there is no out-of-band communication between the parties involved in the interaction. Given that, we limit the processing of an incoming request to *one* control structure, namely a *sequence* of subrequests, but with a significant modification: Each subrequest in such a processing sequence can be decomposed into an arbitrary large set of *parallel* outgoing requests which can be forwarded independently of each other. This yields a simple hierarchical processing structure of so-called *seq/par trees*, that can be used to describe rich interaction schemes in a modular fashion. More sophisticated control flows can be achieved via VP composition as we will see in Sect. 3.4. This control structure may lead to a different decomposition of the message exchange with requestors and providers than using the various control structures available in orchestration languages, such as BPEL4WS: The overall processing sequence can be derived from the dependencies between subsequent messages ($request_{i+1}$ may require parts from the reply to $request_i$), but within a sequencing step i we group the outgoing requests that independently may contribute to the result of i. With the expressive power of BPEL4WS, one can ignore this and combine these interactions in various ways making it harder to identify these dependencies, or, in the worst case, ignoring them. Giving the designer the freedom to choose among a set of alternative approaches makes it harder to detect inconsistencies or errors.

3 THE COMMUNICATION INTERFACE OF VIRTUAL PROVIDERS

We now specify the VP model formally using the Abstract State Machine modeling framework, a form of pseudo-code coming with a rigorously defined semantics. An introduction into the ASM method for high-level system design and analysis is available in textbook form in [6] and in form of a tutorial in [8], but most of what we use below to model VP constructs is self-explanatory.

For example, Figure 2 contains traditional flowchart notations to sequentially compose control state ASMs, an extension of Finite State Machines, where $initStatus(M)$ and $finalStatus(M)$ denote respectively the initial and final control state of M, which usually remain hidden in the graphical notation. For M **seq** N we stipulate

$$initStatus(M \textbf{ seq } N) = initStatus(M)$$
$$finalStatus(M \textbf{ seq } N) = finalStatus(N)$$

For M **until** $Cond$ we define

$$yes(Cond, M) = finalStatus(M \textbf{ until } Cond)$$
$$no(Cond, M) = initStatus(M)$$

Following the SOA paradigm, we treat a Virtual Provider as an interface VirtualProvider providing the following five methods:

- ReceiveReq for receiving request messages (elements of a set *InReqMssg* of legal incoming request messages) from requestors,
- SendAnsw for sending answer messages (elements of a set *OutAnswMssg*) back to requestors,
- Process to handle request objects, elements of a set *ReqObj* of internal representations of *ReceivedRequests*, typically by sending to providers a series of subrequests to service the currently handled request *currReqObj*,[1]

[1]Since the underlying message passing system is abstract,

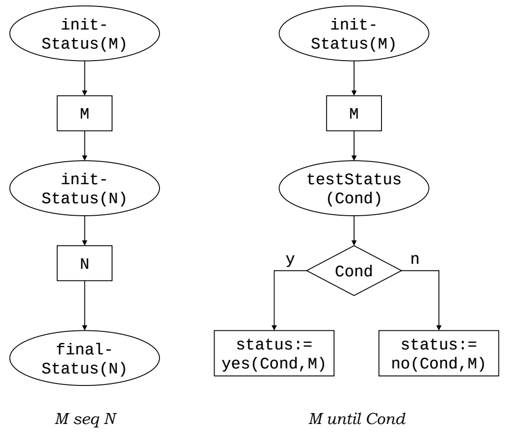

Figure 2: Sequential Composition of Control State ASMs

- SENDREQ for sending request messages (elements of a set *OutReqMssg*) to providers,
- 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 — follows the devised architecture in Fig. 1 and separates the specification of the functionality of VP components from that of their schedulers.

```
MODULE VirtualProvider =
  **choose** $M \in \{\textsc{ReceiveReq}, \textsc{SendAnsw}\} \cup$
    $\{\textsc{Process}, \textsc{SendReq}, \textsc{ReceiveAnsw}\}$
      $M$
```

We start with formulating a “stateless” communication model between requestors and (virtual) providers. In this model, we assume that one incoming request contains all information that is needed to generate a reply. Opposed to that, a “stateful” model requires the virtual provider to store “state” in order to return an answer since that answer is computed from data that is spread across different requests. This extended model is covered in Sect. 3.3.

3.1 Abstract Message Passing

For sending and receiving request and 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 requestor (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 requestors 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.

3.2 The SEND and RECEIVE Submachines

The interaction between a requestor and a VIRTUALPROVIDER is triggered by the arrival of a requestor's request message: *ReceivedReq*(*inReqMsg*) becomes true and a request object is created which is appropriately initialised with an internal representation of the relevant data, which can be extected from the 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 parameterised by the incoming request message *inReqMsg* and by the set *ReqObj* of current request objects of the VP. For simplicity of exposition we assume a preemptive *ReceivedReq* predicate.[2]

```
ReceiveReq(inReqMsg, ReqObj) =
  **if** ReceivedReq(inReqMsg) **then**
    CreateNewReqObj(inReqMsg, ReqObj)
  **where** CreateNewReqObj(m, R) =
    **let** $r = new(R)$[3] **in** Initialize(r, m)
```

The inverse interaction between a VP and a requestor, which is sending back an answer message to a previous request of the requestor, is characterised 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:

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.

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

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

```
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.[4]

```
ReceiveAnsw($inAnswMsg, AnswerSet$)[5] =
  if $ReceivedAnsw(inAnswMsg)$ then
     insert $answer(inAnswMsg)$
        into $AnswerSet(requestor(inAnswMsg))$
```

Behavioural 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 [9]: 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 realises 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.

3.3 Refinement by a "State" Component

Given the definitions above, it is easy to extend ReceiveReq to support a "stateful" communication model where VirtualProviders need to store some state on previously received requests for further processing at a later stage. 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.

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 $status(r) := refreshStatus(r, inReqMsg)$.[6] 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($inReqMsg, ReqObj$)
    else
      let $r = prevReqObj(inReqMsg)$ in
        RefreshReqObj($r, inReqMsg$)
```

3.4 Virtual Provider Composition

Combining the inner seq/par structure of our VPs with their composition, along the communication interfaces for sending/receiving requests/answers, provides the possibility to describe in a modular way rich mediator structures with sophisticated control flows.

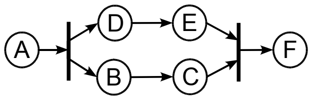

Figure 3: A Control Flow Example

For example, the control flow of Fig. 3 can be realized by a composition of three VPs as indicated in Fig. 4: Let VP1 upon the InitialRequest send a first subrequest for A, followed by sending two parallel requests to subproviders VP2 and VP3 respectively, whose answers in turn trigger VP1 to send the final subrequest for F before eventually providing its FinalAnswer. VP2 and VP3 themselves can be sequentially structured, requesting first B and then C respectively first D and then E. Note that these two sequences (and in particular their termination behavior) are independent of each other. Without the possibility to compose our VPs, the only way to realize such a structured

[4]The function $requestor(inAnswMsg)$ is defined below to denote the value of *seqSubReq* in the state when the request message $outReq2Msg(s)$ for the parallel subrequest s was sent out to which the *inAnswMsg* is received now.

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

[6]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 VP.

control flow would be via programmming the corresponding internal VP behavior.

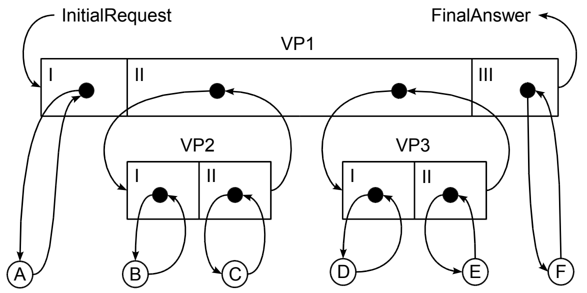

Figure 4: Virtual Provider Composition

The composition of VPs allows the designer to express such control flows in a modular fashion, connecting the communication interfaces in the appropriate way (see also 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 composition allows one to configure 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.

3.5 The Processing Submachine

In this section we define the signature and the transition rules of the ASM Process for the processing kernel of a VirtualProvider. The definition provides a schema, which is to be instantiated for each particular Processing kernel of a concrete VP by giving concrete definitions for the abstract functions and machines we are going to introduce. For an example see Sect. 4.

Since we want to abstract from the scheduler, which calls Process for particular current request objects, we describe the machine as parametrised by a global instance variable $currReqObj \in ReqObj$.

In Fig. 5, each Processing call for a *started* request object *currReqObj* triggers some form of iterative sequential subrequest processing, which will be detailed below. Once this subrequest processing has finished (expressed by the fact that the processing status of the request object *currReqObj* has become *compileAnswer*), the Process machine compiles an answer for *currReqObj*. This internal answer information, say $outAnswer(currReqObj)$ is transformed into an element of *OutAnswMssg* using an abstract function $outAnsw2Msg(a)$, ready for *delivery*. We guard this answer compilation by a check whether *AnswToBeSent* for the *currReqObj* evaluates to true.

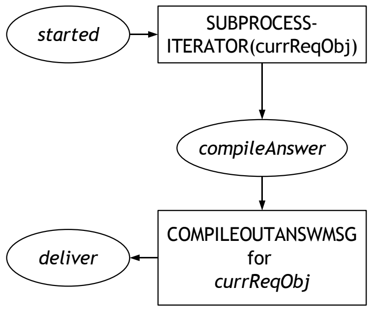
started
SUBPROCESS-ITERATOR(currReqObj)
compileAnswer
COMPILEOUTANSWMSG for currReqObj
deliver

Figure 5: Processing($currReqObj$)

This leads to the following textual definition of the Process machine:

```
Process($currReqObj$) =
  if $status(currReqObj) = started$ then
    SubProcessIterator($currReqObj$)
  if $status(currReqObj) = compileAnswer$ then
    CompileOutAnswMsg for $currReqObj$
    $status(currReqObj) := deliver$
  where
    CompileOutAnswMsg for $o$ =
      if $AnswToBeSent(o)$ then
        $SentAnswToMailer($
          $outAnsw2Msg(outAnswer(o))) := true$
    $compileAnswer$ =
      $yes(FinishedSubReqProcessg)$
```

In a SubProcessIterator step, the immediate subrequests of *currReqObj* will be processed *in order*, as 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*.

```
SubProcessIterator($currReqObj$) =
  InitializeIterator($currReqObj$) seq
  IterateSubReqProcessg($currReqObj$) until
    $FinishedSubReqProcessg$
  where
    $yes(FinishedSubReqProcessg) = compileAnswer$
    $no(FinishedSubReqProcessg) =$
      $initStatus($IterateSubReqProcessg$)$
```

The machine that performs the processing of a single (sequential) subrequest is defined in Fig. 6.

For every current item *seqSubReq*, it starts to FeedSendReq with a request message to be sent out for

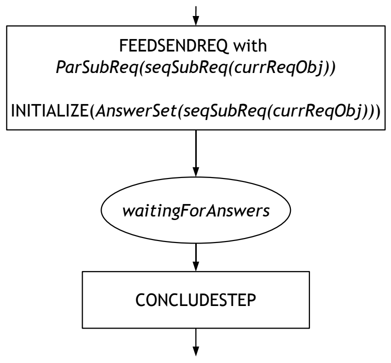
FEEDSENDREQ with
ParSubReq(seqSubReq(currReqObj))
INITIALIZE(AnswerSet(seqSubReq(currReqObj)))
waitingForAnswers
CONCLUDESTEP

Figure 6: IterateSubReqProcessg

every immediate subsubrequest s of the current *seqSubReq*, namely by setting $SentReqToMailer(outReq2Msg(s))$ to *true*. Here, $outReq2Msg(s)$ 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*, a submachine ConcludeStep lets the iterative subprocess ProceedToNextSubReq.

As long as during *waitingForAnswers*, the predicate *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 =
  if $status(currReqObj)$ =
    $initStatus$(IterateSubReqProcessg) then
      FeedSendReq with
        $ParSubReq(seqSubReq(currReqObj))$
      Initialize($AnswerSet(seqSubReq(currReqObj))$)
      $status(currReqObj) := waitingForAnswers$
  if $status(currReqObj)$ =
    $waitingForAnswers$ then
      ConcludeStep
  where
    FeedSendReq with
      $ParSubReq(seqSubReq)$ =
        forall $s \in ParSubReq(seqSubReq)$
          $SentReqToMailer(outReq2Msg(s))$ :=
            $true$
    ConcludeStep =
      if $AllAnswersReceived$ then
        ProceedToNextSubReq
          $status(currReqObj)$ :=
            $Nxt(status(currReqObj))$
        $Nxt(waitingForAnswers)$ =
          $testStatus(FinishedSubReqProcessg)$
```

For the sake of completeness we now define the remaining macros used in the definitions above, 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 set $SeqSubReq \cup \{ Done(SeqSubReq(currReqObj)) \}$,
- The 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*.

```
InitializeIterator($currReqObj$) =
  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.

```
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 VP: 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 APPLICATION SCENARIO: PROTOCOL MEDIATION

In this section we exemplify the protocol mediation scenario [3] by the use case scenario of 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 provider interface for its clients. For the following discussions, we assume that the VISP has chosen an interface that 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,[7] uniquely identifying the four names provided in the request message in the RIPE database. We skip the obvious instantiation of VirtualProvider to formalise this VISP. [8]

Let's 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`).

Does the VISP now have to revise its provider interface or can it extend its business using these new registration authorities without changes?

As depicted in Fig. 7, the VISP can indeed support these new authorities without changes by using a proper VP instantiation.[9] Within this VP, the incoming request *RegisterDomain* is split into a sequence of two subrequests. The first subrequest, which we call *RegAccts*, 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, called *RegDomain*, can be performed, whose outgoing request message is constructed from the answers of *RegAccts* and the `DomainName` parameter from the incoming request.

[7]RIPE stands for "Réseaux IP Européens", see http://ripe.net.
[8]A detailed refinement for this scenario can be found in [10].
[9]We use mnemonic abbreviations for the request message and parameter names.

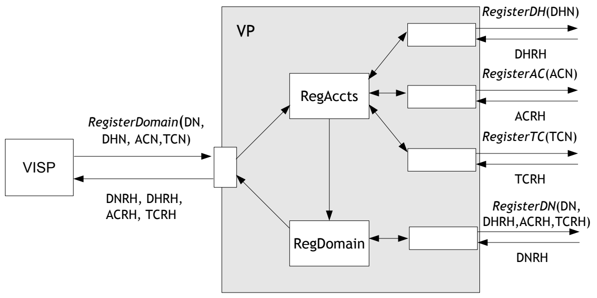

Figure 7: VirtualProvider Instance

4.1 Proving System Properties

Having 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 VP instance defined above correctly supports the refined interface for the new registration authorities.

The claim follows if we can show the correctness of the 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 parameters of *RegisterDomain*. This is trivial for the first VP and can be stated for the second VP more precisely precisely by saying that the following holds for every *successful* pair of *inReqMsg* and its *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 instance we only consider the case of successful registrations, without further interaction between requestor and VP):

Correctness Lemma. 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*(`DNRH`, `DHRH`, `ACRH`, `TCRH`).

Proof. A simple analysis of VISP runs shows that an incoming request message *RegisterDomain*(DN, DHN, ACN, TCN) triggers the VP instance to first SEND three subrequests *RegisterDH*(DHN), *RegisterAC*(ACN), *RegisterTC*(TCN), which are (assumed to be) answered by RIPE handles DHRH, ACRH, and TCRH. Then the subrequest *RegisterDN*(DN, DHRH, ACRH, TCRH) is sent, 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 this proof only works under the assumption that the real service providers work correctly, since the VP only acts as a mediator.

What if the VISP had decided to use the interface with four messages as its provider interface? Could it then use an authority that supports the interface with the single message *RegisterDomain*(DN, DHN, ACN, TCN)? To relate the four incoming messages to the single provider message one has to construct a stateful VP instance that stores the three names provided in *RegisterDH*(DHN), *RegisterAC*(ACN), and *RegisterTC*(TCN). Then, after receiving the *RegisterDN*(DN, DHRH, ACRH, TCRH) request, it could forward the *RegisterDomain* request to the provider with the information collected for that client. A closer analysis reveals, though, that this is not possible: The RIPE handles, that the VP would return to the client for the first three requests, would have to be created by the VP itself (there is no interaction yet with the registration authority at this stage) and, thus, would not denote real-life RIPE handles. In other words: The claims for the correctness lemma we would have to formulate for that scenario would not hold! It is easy to see that this dilemma cannot be solved by equipping the VP with more powerful control structures, but only by changing the interaction protocol. For example, we could change the "semantics" of the RIPE handles returned by the first three requests and declare them as being "temporary". Then, a modified *RegisterDN* request could return all four handles, where the temporary handles are replaced by their official counterparts.

5 APPLICATION SCENARIO: SEMANTIC WEB SERVICE DISCOVERY

As a second application scenario for the VP model, we investigate Semantic Web Service (SWS) discovery. SWS discovery relies on capability-based semantic matchmaking and is an activity within a more general Web service framework comprising all activities required to find, select and invoke a Web service [11]. OVerall, such a framework performs the following three activities:

1. Goal Discovery
2. Semantic Web Service Discovery
3. Service Selection

In this paper, we skip the discussion on goal discovery and service selection and provide only an overview of the SWS discovery framework and show how its formal specification can be realized through refinement of the VIRTUALPROVIDER model from Sect. 3.

5.1 SWS Discovery Framework

SWS discovery is based on matching semantically described goal descriptions (goal queries) with semantic annotations of Web services (capability descriptions). Several capability-based Semantic Web Service discovery solutions have been proposed in the literature [11, 12, 13, 14]. A capability description annotates thereby either the inputs and outputs of a Web service [11, 14] or describes an abstract service capability [11, 12, 13] and can be applied in the frame of both OWL-S [15] and WSMF/WSMO [3, 16]. The discovery process in the capability-based approaches can only happen on an ontological level, i. e. it can only rely on conceptual and reusable things. For this, two processes are required:

- the concrete user input has to be generalised to a more abstract goal description, and
- concrete services and their descriptions have to be abstracted to the classes of services a Web service can provide.

We believe that this twofold abstraction is essential for lifting Web service discovery on an ontological level that is the prerequisite for a scalable and interoperable solution. However, the above two steps are beyond the scope of SWS discovery. SWS discovery requires only data/ontology mediation. Service selection, for instance, requires in general also data, protocol, and process mediation. SWS discovery takes as input a goal formulated in the goal discovery and returns a set of Web services matching this goal. One could assume that Web services and goals are described by the same terminology. Then no data or ontology mediation problem exists during the discovery process. However, it is unlikely that a potentially huge number of distributed and autonomous parties will agree before-hand on a common terminology.

Alternatively, one could assume that goals and Web services are described by completely independent vocabularies. Although this case might happen in a real setting, discovery would be impossible to achieve. In consequence, only an intermediate approach can lead to a scenario where neither unrealistic assumptions nor complete failure of discovery has to occur. Such a scenario relies on three main assumptions:

- Goals and Web services most likely use different vocabularies, or in other words, we do not restrict our approach to the case where both need to use the same vocabulary.
- Goals and Web services use controlled vocabularies or ontologies to describe requested and provided services.
- There is some data/ontology mediation service in place. Given the previous assumption, we can optimistically assume that a mapping has already been

established between the used terminologies, not to facilitate our specific discovery problem but rather to support the general information exchange process between these terminologies.

Under these assumptions, we do not simply neglect the mapping problem by assuming that it does not exist and, at the same time, we do not simply declare discovery as a failure. We rather look for the minimal assumed data/ontology mediation support that is a prerequisite for successful discovery. A framework for Semantic Web Service discovery taking into account ontology mediation and search distribution over different discovery locations is illustrated in Fig. 8.

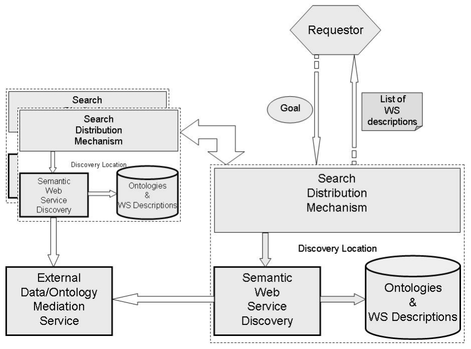
Requestor
Goal
List of WS descriptions
Search
Search Distribution Mechanism
Discovery Location
Semantic Web Service Discovery
Ontologies & WS Descriptions
Search Distribution Mechanism
Discovery Location
External Data/Ontology Mediation Service
Semantic Web Service Discovery
Ontologies & WS Descriptions

Figure 8: SWS Discovery Framework

A user sends a goal (initial goal) to the SWS discovery service and expects as a result a set of Web service descriptions matching this goal. A SWS discovery service may be distributed over several search locations. In that case, the SWS discovery service has a search distribution mechanism identifying external discovery locations and querying them as well as the local search location with the goal as a search parameter. The distribution mechanism must be able to detect search loops and terminate search for duplicate requests coming from external discovery locations. The distribution mechanism waits until search results from all search locations have arrived or a timeout occurs (Fig. 9).

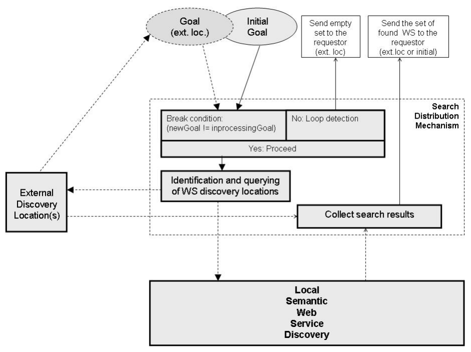
Goal (ext. loc.)
Initial Goal
Send empty set to the requestor (ext. loc)
Send the set of found WS to the requestor (ext.loc or initial)
Search Distribution Mechanism
Break condition: (newGoal != inprocessingGoal)
No: Loop detection
Yes: Proceed
Identification and querying of WS discovery locations
External Discovery Location(s)
Collect search results
Local Semantic Web Service Discovery

Figure 9: Search Distribution Mechanism

A local discovery location performs two steps in order to match the goal with the locally available Web service descriptions (Fig. 10).

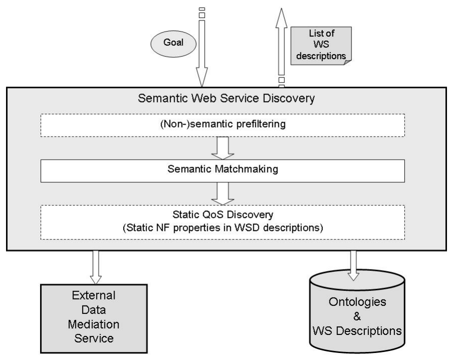
Goal
List of WS descriptions
Semantic Web Service Discovery
(Non-)semantic prefiltering
Semantic Matchmaking
Static QoS Discovery
(Static NF properties in WSD descriptions)
External Data Mediation Service
Ontologies & WS Descriptions

Figure 10: Local Service Discovery Process

In the first optional step, a "(non-)semantic prefiltering" reduces the set of Web services to be semantically matched in the second step "Semantic Matchmaking". *The filtering procedures applied in the first step must guarantee that no potentially matching Web services are filtered out.* In the second step, the set of Web services is matched against the goal and the resulting set of matching Web services is delivered to the distribution mechanism collecting the search results. Both steps may require an external data/ontology mediation service for the reasons already described above. Optionally, the set of services matching the goal can be further reduced or ranked by matching non-functional QoS properties of the goal and Web services, e. g. security or reliability requirements.

5.2 A Possible VirtualProvider Refinement for SWS Discovery

To formalise the Semantic Web Service discovery framework as a refinement of our VP model we adjust our terminology in the definitions to clarify the intended interpretation of the abstractions by applying the following syntactical mappings:

- $Req \rightarrow Goal$
- $Answ,\ Answer,\ AnswerSet \rightarrow \{SetofWS, WS\}$
- $\textsc{Process} \rightarrow \textsc{ProcessGoal}$
- $ParSubReq(seqSubReq(currReqObj))$
  $\rightarrow ParGoalQuery(currGoalObj)$

Applying this renaming scheme,[10] the communication interface described in Sect. 3 is turned into the following description for SWS discovery:

$$\begin{array}{l}
\textsc{DiscoveryServiceProvider} = \\
\quad \textbf{choose } M \in \{\textsc{ReceiveGoal}, \textsc{SendSetOfWS}\} \cup \\
\qquad \{\textsc{ProcessGoal}, \textsc{SendGoal}, \textsc{ReceiveSetOfWS}\} \\
\qquad M
\end{array}$$

[10]Below we will include into these purely notational changes renamings like *mailer* into *requestor* or *provider*, etc.

5.3 Refinement of the SEND and RECEIVE Submachines

In the SWS discovery scenario, we assume the "stateless" model described in Sect. 3.2. The notion of state within the VP is restricted to the detection of loops, which can be caused by other discovery service providers sending a goal that is already in the processing by the receiving provider. Therefore, a goal must be uniquely identifiable in the global context.

This "loop detection" feature is captured in the refinement of the RECEIVEGOAL submachine which is derived from the RECEIVEMSG submachine defined in Sect. 3.2. To illustrate this refinement we repeat the definition of the RECEIVEGOAL submachine for comparison:

```
RECEIVEREQ($inReqMsg, ReqObj$) =
  if $ReceivedReq(inReqMsg)$ then
    CREATENEWREQOBJ($inReqMsg, ReqObj$)
  where CREATENEWREQOBJ($m, R$) =
    let $r = new(R)$[11] in
      INITIALIZE($r, m$)
```

Next, we present the refined version with the changes highlighted:

```
RECEIVEGOAL($inGoalMsg, GoalObj$) =
  if $ReceivedGoal(inGoalMsg)$ then
    CREATENEWGOALOBJ($inGoalMsg, GoalObj$)
  where
    CREATENEWGOALOBJ($m, R$) =
      let $g = new(R)$ in
        INITIALIZE($g, m$)
        INITIALIZE($SetOfWS(g)$)
        if $NewGoal(g, m)$ then
          $status(g) := started$
        else
          $status(g) := loopDetected$
    INITIALIZE($SetOfWS(g)$) =
      $SetOfWS(g) := \emptyset$
```

When the predicate *NewGoal* returns *false*, the status of the goal g is set to *loopDetected* which will influence the processing of the goal as described in the section below.

The remaining three submachines directly correspond to the VP machines in Sect. 3 after applying the following additional renamings:

- In SENDGOAL : $SentReqToMailer \rightarrow SentGoalToProvider$.
- In SENDSETOFWS : $SentAnswToMailer \rightarrow SentSetOfWSTorequestor$.
- In RECEIVESETOFWS : $answer \rightarrow setOfWS$, $requestor \rightarrow goal$, $AnswerSet \rightarrow SetOfWS$.

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

5.4 The PROCESSGOAL Submachine

As the name implies, the PROCESGOAL submachine is a refinement of the VP PROCESS submachine defined in Sect. 3.5.

As one can see in Fig. 11, the overall processing model from Fig. 5 is just extended by an additional rule involving the new controlstate *loopDetected*. As outlined above, this status indicates that a request for an already processed goal has been received. If such a "loop" is detected, the processing status for that goal is changed to *compileAnswer*, indicating that the goal queries don't have to be sent again.

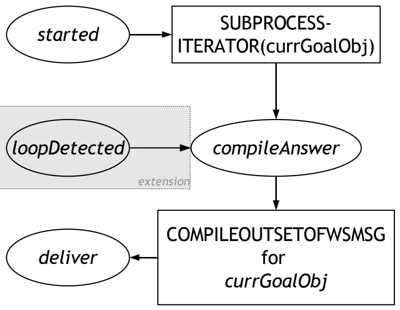
started
SUBPROCESS-ITERATOR(currGoalObj)
loopDetected
compileAnswer
extension
COMPILEOUTSETOFWSMSG for currGoalObj
deliver

Figure 11: PROCESSGOAL

In the discovery scenario, the sequence of subrequests suggested by the generic VP process model reduces to just one element that is determined by the incoming goal request and thus by *currReqGoal*. Therefore there is no iteration and we immediately FEEDSENDGOAL with a goal query to all relevant discovery locations l known to *currReqGoal* and set *FinishedSubReqProcessing* to *true*. Formally this comes up to refine the machine SUBPROCESSITERATOR in the following way:

```
INITIALIZEITERATOR = skip
$FinishedSubReqProcessing = true$
```

Correspondingly the ITERATESUBREQPROCESSG submachine is refined as follows:

```
FEEDSENDREQ with
  $ParSubReq(seqSubReq(currReqObj))$ =
    FEEDSENDGOAL with
      $ParGoalQuery(currGoalObj)$
INITIALIZE($AnswerSet(\ldots)$) = skip
```

Finally, we put the second submachine CONCLUDESTEP under an additional new guard, namely that a certain *BreakCondition* (e.g. a timeout) does not hold:

```
if $status = waitingForAnswers$ then
  if not $BreakCondition$ then CONCLUDESTEP
```

In case *BreakCondition* is true, a new submachine GENERATEEXCEPTION is called that will generate the appropriate exception message to be sent back to the requestor. Note that in both cases the sub-processing will stop since the status condition *FinishedSubReqProcessing* is *true* and the status will process to *compileAnswer* as depicted in Fig. 12.

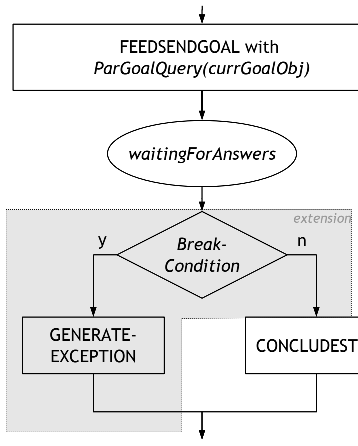

Figure 12: Refinement of ITERATESUBREQPROCESSG

5.5 DISCOVERYENGINE **ASM**

Similar to DISCOVERYSERVICEPROVIDER, we see the discovery engine performing the local Web service discovery (Fig. 10) as an interface, which is defined by the following methods:

- RECEIVEGOAL for receiving goal queries from a requestor,[12]
- 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*

We define a discovery engine as 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):

```
DISCOVERYENGINE =
  choose $M \in \{$RECEIVEGOAL, SENDSETOFWS$\} \cup$
    $\{$MATCHGOAL$\}$
      $M$
```

The machine SENDSETOFWS has been simply reused from DISCOVERYSERVICEPROVIDER. The machine RECEIVEGOAL from DISCOVERYSERVICEPROVIDER has been slightly modified.

[12]Each instance of the abstract machines DISCOVERYENGINE we are going to define here is associated with a DISCOVERYSERVICEPROVIDER, i.e. only the associated Discovery Service Provider (*DSP*) asking for servicing a goal query of a goal received by *DSP* will be served.

```
RECEIVEGOAL($inGoalMsg, GoalObj$) =
  if $ReceivedGoal(inGoalMsg)$ then
    let $g = New(GoalObj)$ in
      INITIALIZE($g, inGoalMsg, inSetOfWS$[13])
      INITIALIZE($SetOfWS(g)$)
    $status(g) := started$
```

The machine MATCHGOAL executes sequentially the machines PREFILTERING, SEMANTICMATCHMAKING and QOSMATCHMAKING reducing stepwise the initial set of Web services *inSetOfWS* to the final set of Web services matching the goal. The final set is sent to the *DSP* as soon as *currGoalObj* is set to $status(currGoalObj) := compileAnswer$.

```
MATCHGOAL($currGoalObj$) =
  if $status(currGoalObj) = started$ then
    PREFILTERING(CURRGOALOBJ)
    $status(currGoalObj) := filtered$
  if $status(currGoalObj) = filtered$ then
    SEMANTICMATCHMAKING(CURRGOALOBJ)
    $status(currGoalObj) := matched$
  if $status(currReqObj) = matched$ then
    QOSMATCHMAKING(CURRGOALOBJ)
    $status(currGoalObj) := compileAnswer$
  if $status(currGoalObj) = compileAnswer$ then
    COMPILEOUTSETOFWSMSG from $currReqObj$
    $status(currGoalObj) := deliver$
```

The machines PREFILTERING, SEMANTICMATCH- MAKING and QOSMATCHMAKING can now be further refined in order to implement different filtering and matchmaking methods or strategies.

6 CONCLUSION

This paper provides a formal, high-level model of an *Virtual Provider* execution engine. We have strived for a simple execution model with limited expressive power, yet powerful enough to support interesting application scenarios. Successive refinements towards executable descriptions could easily mapped to existing orchestration languages, like BPEL4WS.

This model has originally been designed to support process mediation scenarios in the context of Service-Oriented Architectures and we have shown in Sect. 4 how this model can be refined to support real-world scenarios providing means to study important system properties.

We could also show in Sect. 5 that the VP has proven to be useful as a basis for formal specifications of distributed semantic discovery frameworks. Here, only minor changes on the VP structure were 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

[13] *inSetOfWS* is assumed to contain an initial set of Web services to be matched with the goal g.

for an implementation of a discovery service, can be derived as different refinements of the same abstractions.

We can also imagine more involved practical instantiations of VIRTUALPROVIDER than the simple one illustrated in Sect. 4. Another direction of research concerns replacing the simple communication patterns used by VP by more complex ones. RECEIVEREQ and SENDANSW are identified in [17] as basic bilateral service interaction patterns, namely as mono-agent ASM modules RECEIVE and SEND; The FEEDSENDREQ submachine together with SENDREQ in PROCESS realise an instance of the basic multilateral mono-agent service interaction pattern called ONETOMANYSEND in [17], whereas the execution of RECEIVEANSW in ITERATESUBREQPROCESSG until *AllAnswersReceived* is an instance of the basic multilateral mono-agent ONEFROMMANYRECEIVE pattern from [17]. One can refine VP to concrete business process applications by enriching the communication flow structure built from basic service interaction patterns as analysed in [17].

REFERENCES

[1] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[2] G. Wiederhold. Mediators in the architecture of future information systems. In Michael N. Huhns and Munindar P. Singh, editors, *Readings in Agents*, pages 185–196. Morgan Kaufmann, San Francisco, CA, USA, 1997.

[3] D. Fensel and C. Bussler. The web service modeling framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.

[4] Business process execution language for web services, version 1.1. http://www.ibm.com/deverloperworks/library/ws-bpel/.

[5] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web services choreography description language version 1.0 — w3c working draft, 17 December 2004. http://www.w3.org/TR/ws-cdl-10/.

[6] E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003.

[7] E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.

[8] E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *Frontiers of Combining Systems*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 264–283. Springer, 2005.

[9] A. Barros, M. Dumas, and P. Oaks. A critical overview of the Web Serives Choreography Description Language (WS-CDL). White paper, 24 January 2005.

[10] M. Altenhofen, E. Börger, and J. Lemcke. A high-level specification for mediators. *1st International Workshop on Web Service Choreography and Orchestration for Business Process Management, BPM 2005*, 2005.

[11] U. Keller, R. Lara, and A. Polleres. D5.1,v0.1 WSMO web service discovery. White paper, 12 November 2004. http://www.wsmo.org/TR/d5/d5.1/v0.1/.

[12] B. Motik, S. Grimm, and C. Preist. Variance in e-business service discovery. *In Proc. 1st Intl. Workshop SWS2004 at ISWC 2004*, 2004.

[13] L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *Proc. of the Twelfth World Wide Web Conference*, 2003.

[14] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web service capabilities. In *Proc. of the 1st Intern. Semantic Web Conf. (ISWC)*, pages 333–347, 2002.

[15] OWL-S: Semantic markup for web services. Technical Overview, 2004. http://www.daml.org/services/owl-s/1.1/overview/.

[16] D. Roman, H. Lausen, and U. Keller. D2 v1.2. Web Service Modeling Ontology (WSMO). WSMO final draft. Digital Enterprise Research Institute (DERI), 10 February 2005. http://www.wsmo.org/TR/d2/v1.1/.

[17] A. Barros and E. Börger. A compositional framework for service interaction patterns and communication flows. In K.-K. Lau and R. Banach, editors, *Formal Methods and Software Engineering. Proc. 7th International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *LNCS*, pages 5–35. Springer, 2005.