Semantic Service Markup with SESMA

Joachim Peer
MCM Institute University of St. Gallen
Blumenbergplatz 9
9000 St. Gallen, Switzerland
joachim.peer@unisg.ch

ABSTRACT

This paper presents the XML based service markup format SESMA. The proposed language aims to increase the usability of semantic Web service descriptions by presenting a compact syntax, providing convenient support for nondeterministic services and by aligning the annotation format with industry standards like WSDL and BPEL. The present paper describes the design goals of the language, contrasts it with existing work, gives an overview of syntax and semantics of the language and illustrates it by a collection of examples.

Categories and Subject Descriptors

H.3.5 [**Information Systems**]: INFORMATION STORAGE AND RETRIEVAL On-line Information Services[Data sharing]; I.2.4 [**Computing Methodologies**]: ARTIFICIAL INTELLIGENCE Knowledge Representation Formalisms and Methods[Semantic networks]

Keywords

semantic Web services, annotation, XML, WSDL, BPEL

1. INTRODUCTION

Web services are distributed software components that can be exposed and invoked over the internet using standard protocols. This concept was put forward by major IT companies like Microsoft, IBM and Sun as a Web-compatible solution for distributed computing, with the particularly attractive property of being a vendor neutral approach, created around a collection of open standards, such as WSDL, SOAP, UDDI and BPEL.

Web services communicate with their clients and with other Web services by sending XML based messages over the internet. The signatures of the operations a Web service offers and the message formats it supports form its syntactical interface, which is commonly captured by a Web Service Description Language (WSDL [7]) document.

WSDL allows for decoupling abstract descriptions of service types (called *port types*) from concrete service instances. Therefore, a single port type description can be used to describe the interface of several services. This allows for the definition of *standardized service interfaces*: Participants with a common interest can jointly reach agreements on the semantics of those descriptions. Based on such agreements, client applications may be crafted to use the Web services, and complex processes involving several services may be composed, for instance using the BPEL4WS [3] process description language.

A limitation of this approach becomes immanent when services diverge from the initial agreements. For instance, when a service changes its implementation (e.g. to refine its service offerings), its semantics and probably its syntactic interface will change. Since there is no formal machine interpretable connection defined between the semantics and the syntactic interface, human intervention is needed to decide whether the service is still compatible with the agreed semantics or not.

The root of this problem is that the descriptions are *monolithic* in the sense that the semantics of the descriptions is implicitly hidden in the syntactic structure.

A way of addressing this problem is to decouple the semantics from the syntax. This can be done by annotating the service interface with a syntax-independent specification of the service's meaning and behavior. This descriptions should have a formal foundation, such that it is eligible for automated processing.

However, among the lessons learned from earlier attempts of formal software component description is that it is critically important to impose restrictions on the description framework to maintain practical usability: The formal description language must be simple and intuitive enough to be successfully applied by users in the software industry and its intrinsic computational complexity must be kept low.

Our approach is to view service descriptions as assemblies of simple coarse grained *building blocks* with well established semantics, glued together using a very *reduced* logical formalism. The modesty of the logical formalism forces annotators to find agreements over the basic semantic building blocks instead of creating arbitrarily fine grained world axiomatizations that are too complex to reason with. We think that this middle ground between fine grained axiomatic semantic markup and purely syntactic monolithic descriptions serves the needs of Web agent- and Web service-developers best.

The remainder of this paper is structured as follows: Section 2 describes related work and the motivation for SESMA. Sect. 3 provides an overview of the proposed format. Section 4 then describes the core of SESMA, the functional profile of service and processes and Sect. 5 describes the nonfunctional profile. In Section 6 a formal account of the precise meaning of the language constructs is given, followed

Submitted to the WSS workshop at WWW'05

by a couple of examples in Sect. 7 and a summary in Sect. 8.

2. EXISTING WORK AND MOTIVATION

In the following we will briefly describe the two major semantic Web service efforts already in place and then motivate our proposal.

2.1 OWL-S

A very influential work in the area of semantic Web service description is OWL-S [6]. OWL-S is an ontology for service description based on the Web Ontology Language (OWL [1]). The OWL-S ontology consists of the following three parts: (i) a service *profile* can be used to provide descriptions that contains a high level description of the annotated service, its operations and parameters. This information aims to support service advertising and discovery. (ii) OWL-S defines the *process model* ontology, which can be used to describe a service in more detail. Service operations are modeled as *atomic processes*, which can have any number of inputs, outputs, preconditions (which must all hold in order for the process to be successfully invoked) and effects (that will hold after carrying out a service). Outputs and effects can depend on conditions that must hold at the time the process is performed. Moreover, atomic processes can be embedded in more complex process structures, i.e. *composite processes*, which combine atomic processes and other composite processes using workflow control constructs like sequence, selection, iteration and concurrency. (iii) OWL-S defines a *grounding* ontology, which provides vocabulary to define bridges between semantic OWL-S annotations and concrete service instances.

2.2 WSMO

Another major Web service framework is the Web Service Modeling Ontology (WSMO [9]). WSMO relies on four major components inspired by the Web Service Modeling Framework (WSMF [2]):

(i) it defines a framework for *ontologies*, which can contain concepts, relations, functions, instances and axioms to describe various domains on abstract or concrete levels. (ii) it defines an ontology for *service description*. This ontology encompasses a collection of recommended *non-functional properties*, a capability description and an interface description. The *capability* description allows to specify services using preconditions, assumptions, postconditions and effects, similar to OWL-S' notion of inputs, preconditions, outputs and effects. An *interface* description defines *choreographies* and *orchestrations*, which are communication patterns that describe the client's interaction with the defined service and with other services in order to achieve the specified functionality. (iii) WSMO provides a standard representation of *goals*. These elements allow to specify the client's objectives when invoking a Web Service. (iv) WSMO introduces the concept of *mediators*, which allow to specify transformations between related goals, ontologies and services. Mediators can be defined to resolve heterogeneity problems and to enable interoperability between heterogeneous vocabularies and services.

2.3 The SESMA Design Goals

Why is there a need for the SESMA approach, with these comprehensive frameworks already in place? The motivation behind SESMA was the desire for a simple, easy to use annotation format that allows a tight integration with the existing Web service standards WSDL, SOAP, BPEL and XML. This high level goal encompasses several design aspects. In particular, SESMA should:

- provide an XML based syntax; XML is well suited to provide declarative markup, is widely spread in the developer community and has strong tool support.
- provide important modeling constructs like *logical variables and formulas* as first class citizens that fit well into the language framework, both syntactically and conceptually.
- provide *precisely defined semantics*: In this paper we describe a state transition system-based model that provides the precise semantics of SESMA descriptions.
- provide *support for nondeterministic services*: SESMA allows to define conditions that are evaluated directly after a service execution to determine whether or not the desired effects have occurred.
- be *truly complementary to existing standards*: SESMA does not aim to reinvent any of the existing Web service description and Web service process description standards (e.g. WSDL, BPEL4WS). SESMA only provides markup to *annotate* those description formats, in order to support automatic reasoning over service and process descriptions.
- be *extensible*, i.e. usable beyond the current standards WSDL, BPEL and SOAP: To this end, we exploit the XML namespace framework which allows us to keep SESMA open for future requirements.

While design goals such as precisely defined semantics, support for nondeterministic services and extensibility are also relevant to OWL-S and WSMO, SESMA differs from the existing approaches regarding several other criteria: OWL-S, for instance, gives a strong priority to flexibility and Semantic Web standard compliance; it builds on top of RDF and OWL, which is considerable more complex to use than XML, especially when add-ons like SWRL are considered.

Another difference between SESMA and the OWL-S and WMSO approaches is that SESMA does not provide any constructs for explicitly modeling composite processes. Instead, SESMA allows to annotate existing process descriptions, e.g. BPEL descriptions, to leverage the process information (control flows, data flows) that are already present in the (BPEL) document. SESMA annotations are only added to give the *complementary* information about the semantics (e.g. preconditions and effects) of Web services processes and their components. In contrast, OWL-S uses its own process modeling ontology, and WSMO provides an Abstract State Machine-based process framework for modeling choreographies and orchestrations.

3. OVERVIEW OF THE SESMA FORMAT

In principle, SESMA annotations can be used to markup all kinds of computational entities on the internet. Currently WSDL based Web services and BPEL based processes are the supported targets of SESMA annotations. The labeled arrows in Fig. 1 illustrate the relations between SESMA and

the Web service and process description standards WSDL and BPEL:

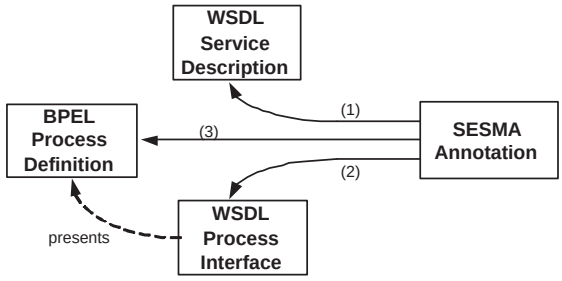

Figure 1: SESMA, WSDL and BPEL

1. SESMA can be used to annotate service descriptions based on WSDL.
2. SESMA can be used to annotate WSDL based process interfaces, i.e. documents that describe the public operations and message formats of a BPEL process.
3. SESMA can be used to directly annotate fragments of BPEL process definitions.

A SESMA annotation consists of two parts: a *functional* profile, which describes the operational aspects of the service (or process) and a *non-functional* profile which allows for semantic annotation of the service (or process) as a whole (e.g. information about the provider, quality of service guarantees)

The XML fragment below shows the skeleton of a SESMA Web service annotation:

```
<annotation target="WSDL-1.1"
  xmlns:wsdl="http://schema.org/sesma/wsdl"
  wsdl:url="http://some.com/GigashopService?wsdl"
  wsdl:serviceName="GigashopServiceService">

  <!-- the annotated operations -->
  <functional-profile>
     ...
  </functional-profile>

  <!-- other properties of the service -->
  <nonfunctional-profile>
    ...
  </nonfunctional-profile>
</annotation>
```

To distinguish the various different target formats, SESMA provides an attribute *target* for its top level element *annotation*. The value of that attribute is set according to the target format described, e.g. *target='WSDL-1.1'* in case of WSDL 1.1. and *target='BPEL4WS-1.1'* in case of the current BPEL version. Software agents can inspect this value to determine the exact type of service or process description format they are dealing with, to find out whether they can process this format or not.

Further, SESMA uses qualified namespaces to provide retrieval information related to the target format, e.g. *wsdl:url* refers to the URL where the annotated WSDL file can be found and *wsdl:serviceName* to refer to the name of the annotated service.

Analogously, the qualified attribute *bpel:url* refers to the URL of the annotated BPEL document and *bpel:name* refers to the name of the annotated process.

It is assumed that the semantic Web agent retrieves the referred documents, correctly interprets the syntactic descriptions they inclose[1], then parses the semantic annotations provided by the SESMA document and finally puts the complementary pieces together to get the complete picture of the annotated Web service or process.

In the following, we describe the core part of SESMA, i.e. the markup constructs for expressing the functional aspects of Web services and processes.

4. THE FUNCTIONAL PROFILE

The functional profile provides information about the semantics of annotated computational entities and their activities. Two types of basic activities are supported:

- A *service operation* is a basic atomic activity a service provides; depending on the transmission model of the operation, it can be invoked by the client (e.g. request-response or one-way messages) or it can be started by the service (e.g. notification messages).
- A *process activity* is a subset of a process definition. Process activities can be atomic (i.e. service invocations) or structured (e.g. sequence of activities, selections, iterations, concurrent executions).

For the rest of this document, we will use the term *activity* to refer to both service operations and process activities.

As shown in Fig. 2, every activity may be described by an optional *precondition* and a number of *results*, whereby a result may be described either by an *effect* or a *knowledge effect* formula. Each result may optionally have a *secondary precondition* and a *success condition*. Each activity has a number of *variable groundings*. The meaning of all these constructs will be informally described in the next paragraphs.

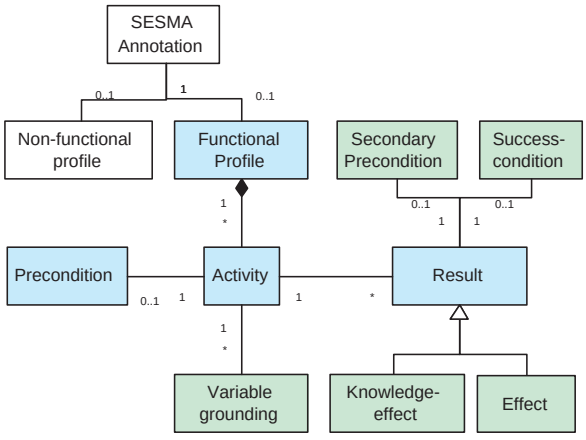

Figure 2: The functional profile of a service

Each activity that should be made available to semantic Web agents needs to be marked up, using an `<act-def>`

[1]Naturally the agent must be aware of the meaning of the technology specific information in order to make use of it.

element. Below we show an example of an annotation of a WSDL-based operation, which is identified by its name and WSDL port type:

```
<act-def wsdl:name="activateUser"
         wsdl:portType="RSAFuncsSoapPT">
   <!-- precondition
        and result formulas -->
</act-def>
```

If the activity to be described is part of a BPEL process, then SESMA provides two ways of pointing to the annotated process activity:

- If the process activity is explicitly named by a unique *name* attribute in the BPEL document, then the *bpel:name* attribute can be used to refer to the annotated process activity.
- If the process activity has no unique name, then the attribute *bpel:path* can be used, which contains an XPointer [8] expression identifying activity to be annotated.

For instance, if the process activity to be annotated is the second top-level *sequence*-element in the process, the following expression can be written:

```
<act-def bpel:path="process/sequence[1]">
   <!-- precondition
   and result formulas -->
</act-def>
```

4.1 Markup of Formulas

Before we go on to discuss the (precondition and result) constructs used to annotate activities, we discuss the markup of *logical formulas* used to build those constructs.

4.1.1 Atomic Formulas

```
<s:user-credentials
  username="?user"
  password="?pwd" />
```

Atomic formulas represent logical relations and are written as XML elements: The namespace-qualified name (QName) of the XML element represents the predicate of the relation; the attributes of the XML element represent the relation's parameters. In the context of the semantic Web, the predicate can be interpreted as the *concept* defining the relation and the parameters can be seen as the *properties* of the concept instance(s), as pointed out by [5].

For instance, in the example above, the concept `user-credentials` in a namespace `s` is used, and statements involving the properties `username` and `password` are made. The concepts and properties may be defined using arbitrary schema languages. The only imposed requirement is that the object instances classified by the schema language need to be uniquely identifiable and can be described by a collection of property values (i.e. relations with other objects); languages like RDF Schema and OWL fulfill this requirement.

Variables must start with a question mark `'?'` in order to distinguish them from constants. Complex formulas are built by combining atomic formulas with the logical constructors described below.

4.1.2 Negation

```
<not>
 <s:logged-in user="?user"/>
</not>
```

A negation of a formula (both atomic and complex ones) is constructed by wrapping the formula to be negated into a `<not>` element.

4.1.3 Conjunctions

```
<and>
  <s:logged-in user="?user"/>
  <s:in-catalog item="?item"/>
</and>
```

The conjunction of two or more formulas is constructed by placing the formulas inside an element `<and>`.

4.1.4 Disjunction

```
<or>
  <s:has-credit-card type="visa"/>
  <s:has-credit-card type="mastercard"/>
</or>
```

The disjunction of two or more formulas is constructed by placing the formulas inside an element `<or>`.

4.1.5 Implication

```
<imply>
  <!-- antecedent -->
  <s:in-catalog item="?item"/>
  <!-- consequence -->
  <s:sell item="?item"/>
</imply>
```

A logical implication *antecedent* $\Rightarrow$ *consequence* is constructed by placing the antecedent and the consequence into an element `<imply>`, whereby the antecedent is the first child, and the consequence formula is the second child of the `<imply>` element.

A similar construct is the `<when>` construct. The difference is that the antecedent of the `<imply>` construct refers to the *current* state of the world, while the `<when>` construct is only used in result formulas and its antecedent refers to the state *before* the activity was carried out. A more detailed formal account of these issues is given in Sect. 6.

4.1.6 Universal Quantification

```
<forall>
  <!-- list of quant. variables -->
  <var name="?item" />
  <when>
   <s:in-cart service="gservice.com" item="?item"
       count="?cnt" />
   <s:possess client="$client" item="?item"/>
  </when>
</forall>
```

The universal quantification of a formula can be expressed by placing the formula into an element `<forall>`. Each variable to be quantified has to be put into an element `<var>` with a required attribute `name` representing the name of the variable.

The reader may have noticed that certain parameters in the examples above have a leading dollar sign, e.g. `$client`. These identifiers represent constants which need to be *replaced* at runtime by the actual value. For instance, the identifier `$client` would need to be replaced by the URI of the agent that actually invokes the activity. Another identifier that may be useful in certain situations is `$timestamp`, which is to be substituted at runtime by the current time.

4.2 Preconditions of Activities

A precondition is a formula that describes the requirements that must hold upon service execution in order to achieve any of the activities's results. If a precondition is not fulfilled and an activity is still executed, the results are undefined.

The following example shows a precondition that states that the activity can only be carried out on a Monday and if the caller is a member of `MyCompany`. The namespace prefixes `n` and `m` are referring to some external vocabularies.

```
<precondition xmlns:n="http://biz.org/voc"
              xmlns:m="http://my.com/def">
  <and>
     <n:day-of-week day="Monday"/>
     <n:have-membership member="$client"
        organisation="m:MyCompany"/>
  </and>
</precondition>
```

4.3 Effects of Activities

An agent that invokes an activity with a valid precondition, can expect that the effect formula will evaluate to *true* in the world state after the activity is executed, as long as no success condition exists (cf. Sect. 4.6 below) and as long as no error occurs.

For instance, to describe the effect of an activity that purchases an item from an e-commerce site, the following annotation could be created, representing the transfer of ownership over the item:

```
<effect>
   <s:possess owner="$client" item="?item"/>
</effect>
```

4.4 Knowledge Effects of Activities

Very common in Web Service environments are effects that provide the caller some information but do not have world-altering effects. To model these kinds of effects, the markup element `<knowledge-effect>` is provided.

As an example, consider the following knowledge effect below; the meaning of this effect is that the price `?p` of the requested item `?item` gets known to the agent who invokes the activity without changing the world state. A formal account of the semantics of knowledge-effects is given in Section 6.

```
<knowledge-effect>
  <s:price-of item="?item" price="?p" />
</knowledge-effect>
```

An activity can have (several) combined effects and knowledge effects. This allows, for instance, to model an activity that provides information (⇒ knowledge effect) and charges for the information (⇒ effect with side effects, e.g. deposition of money).

4.5 Secondary Preconditions of Results

While preconditions as presented above in Sect. 4.2 describe conditions that are required for an activity as a whole (i.e. for all of its possible results), secondary preconditions describe conditions that are required just for a specific result (i.e. effect or knowledge effect)

To illustrate this, let us assume that an activity has a precondition P and two results A and B, whereby A has a secondary precondition SP_A and B has no such secondary precondition. Under these circumstances the result A can only be expected if $(P \wedge SP_A)$ holds, whereas B does only require P to hold.

Below we show how a secondary precondition is attached to a result definition:

```
<effect>
  <!-- 2nd-ary precondition -->
  <secondary-precondition>
     ...
  </secondary-precondition>
  <!-- the effect -->
  ...
</effect>
```

The separation of conditions into preconditions and secondary preconditions is a syntactic means to avoid redundant markup.

4.6 Success Conditions of Results

The success of an activity may be undetermined until the execution is actually over. For instance, an operation that sends e-mail messages may return an error report in case the message could not be delivered.

To provide agents with a tool to cope with this kind of nondeterministic behavior, we provide a `success-condition` construct that is attached to an activity's result formula. Unlike the other conditions described above, a success-condition can be represented by any type of expression language that is supported by the agent and capable of returning a Boolean value. The interpreter required to evaluate the expression needs to be specified by the attribute `lang`.

As an example, consider the success condition of the effect formula shown below; the expression language is Java(TM), and the statement tells the agent that the execution was successful if the value of output variable `?status` equals the string `c200`, representing some status code. A value different than `c200` would yield a return value of *false*, indicating that the specified effect of the activity was not achieved[2].

```
<effect>
  <success-condition lang="java">
    outputs.get("?status").equals("c200")
  </success-condition>

  <!-- effect formula follows here -->
  ...
</effect>
```

It should be clearly distinguished between the concept of success conditions described in this section and the concept of preconditions and secondary preconditions described in Sect. 4.5: A (secondary) precondition states that an effect occurs only if its condition is true *prior* to service execution.

[2]Note that the example assumes an API that allows to access the values of the outputs via a data structure *outputs*

In contrast, a *success condition* does not need to be true *before* service execution, but it must be true *after* service execution, to allow the effect to occur legally. This will be elaborated formally in the Sect. 6.

4.7 Grounding the Variables

In the previous sections we used variables in logical formulas to characterize the semantics of activities. We quietly assumed that the actual variables will be bound to actual values during runtime, i.e. when a user sends a message to a service or when a service responds to the user.

To provide semantic Web agents with the means to create the correct variable bindings at runtime, we need to specify the location of each of the variables in the input and output messages of activities.

4.7.1 Inputs

Variables whose values are defined by the input the agent sends to the service are defined within an element `<input>`:

```
<input>
   <var name="?to" wsdl:part="ToAddress"/>
   <var name="?from" wsdl:part="FromAddress"/>
   <var name="?subject" wsdl:part="Content"
        wsdl:path="Subject"/>
   <var name="?msg" wsdl:part="Content"
        wsdl:path="Message"/>
</input>
```

4.7.2 Outputs

In analogy to inputs, `output` definitions specify the connection between the data pieces of the service output and the logical variables used in the annotation.

In the example below, three data pieces are identified and connected to the variables `?x`, `?c` and `?p`, referring to the `id`, `color` and `price` of some `Item`.

```
<output>
   <var name="?x" wsdl:part="result"
        wsdl:path="Item/id"/>
   <var name="?c" wsdl:part="result"
        wsdl:path="Item/color"/>
   <var name="?p" wsdl:part="result"
        wsdl:path="Item/price"/>
</output>
```

As shown in the examples above, an (input or output) variable is introduced by an element `var` and its name is defined by an attribute `name`. In case the underlying description format is WSDL, we use an attribute `wsdl:part` to connect the variable to the correct WSDL message part and we use an optional attribute `wsdl:path` to specify the XPATH of the referenced piece of data within the given message part (if the message is a complex one).

This way, pieces of data sent from and to the service can be connected to variables used in the various formulas used to describe the activities semantics.

5. THE NON-FUNCTIONAL PROFILE

In addition to the functional profile, each service or process can be described by a non-functional profile that may be used to provide additional data about it (e.g. creator, provider, quality of service guarantees).

The nonfunctional (qualitative) profile consists of an arbitrary number of entries, whereby each entry is given by a unique key and a number of values for that key. This table-based schema is both easy to use and extensible. We do not define any mandatory entries nor do we specify syntax or semantics of possible entry types.

Instead of plain (unstructured) text, URIs of standardized concepts may be used to describe the qualitative profile entries, which is the added value of semantic annotation over text based description, since it simplifies the automatic evaluation. As an example, a non-functional profile of a travel planning service could look as follows:

```
<!ENTITY owl-s "http://www.daml.org/services/owl-s/1.1">
<!ENTITY travel "http://some.com/business-onto">
<nonfunctional-profile
  xmlns:owl-s="&owl-s;#Profile.owl">
  <entry key="owl-s:serviceName>
    <value>Joe's Travel-o-Rama</value>
  </entry>
  <entry key="owl-s:serviceCategory">
   <value>&travel;#TravelAgency</value>
   <value>&travel;#TravelInsurance</value>
  </entry>
</nonfunctional-profile>
```

6. FORMAL SEMANTICS OF THE LANGUAGE

This section provides a formal specification of the semantics of the SESMA language. We start by defining the semantics of SESMA formulas, then we describe the idealistic worlds assumed in our considerations, followed by a description of the intended semantics of SESMA preconditions, success conditions, effects and knowledge-effects.

6.1 Semantics of Formulas

The SESMA markup relies on logical formulas to express the characteristics of activities, for instance the precondition and effect formulas.

The syntax of formulas in SESMA was informally presented by example in the Sect. 4. A formal presentation of the syntax is provided by the BNF grammar obtainable from[3]. To define the *semantics* of the formulas used to describe preconditions and effects, we provide a model theoretic specification.

First, we introduce an interpretation $\mathcal{I}$ which is a triple $(\Delta, \mathcal{I}[c], \mathcal{I}[p])$ with:

- Δ being a non-empty set, the "universe".
- a function $\mathcal{I}[c] \subseteq \Delta^0$, which maps every constant symbol (i.e. 0-ary function symbol) to an element in Δ.
- a function $\mathcal{I}[p] \subseteq \Delta^n$, which maps every n-ary predicate symbol p to an n-ary relation Δ^n of the domain.

Since SESMA formulas can contain variables, we need to supplement interpretations with variable assignments, i.e. valuations. A *valuation* for a set of variables $\mathcal{V}$ regarding a universe Δ is a function $\alpha : \mathcal{V} \to \Delta$.

Given an interpretation $\mathcal{I}$ and a valuation α, we define a function $\mathcal{I}_\alpha$ for the interpretation of terms:

- $\mathcal{I}_\alpha[X] = \alpha(X)$, for all $X \in \mathcal{V}$,
- $\mathcal{I}_\alpha[c] = \mathcal{I}[c]$, for all constant symbols c.

[3]http://elektra.mcm.unisg.ch/sesma/ebnf.txt

Now that we have defined how terms (i.e. variables and constants) are evaluated in $\mathcal{I}_\alpha$, we proceed to inductively define the evaluation of SESMA *formulas*. These definitions are summarized in Tab. 1. We use the following conventions: (i) The symbols A and B refer to well formed SESMA formulas. (ii) If a formula F is true in an interpretation $\mathcal{I}_\alpha$, we say that $\mathcal{I}_\alpha$ is a *model* of F and we write $\mathcal{I}_\alpha \models F$.

6.2 Representing World State

We assume that the world is represented in terms of state descriptions. A state can be seen as a database of extensional knowledge, i.e. of ground atomic formulas. Formally, we define a world state s as a conjunction of atomic formulas $F_1 \wedge ... \wedge F_n$. Consequently, for each F_i in s holds $s \models F_i$, with $1 \leq i \leq n$.

As illustrated by Fig. 3, we distinguish between the global state s of a world and the state knowledge s_A of an agent A who reasons about this world. We assume that $s_A \subseteq s$. This means that an agent may have incomplete knowledge, but we do not assume that it has wrong knowledge.

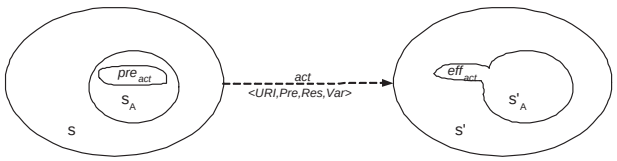

Figure 3: The world and its dynamics are represented by states and state transitions

We assume that the state remains the same over time unless explicitly changed by an activity. For instance, if an agent checks the availability of a product at time t_1, then it assumes that the retrieved information is still valid at some point $t2 \succcurlyeq t1$, as long as it did not invoke an activity that might have changed this fact in between; this assumption is called the *invocation and reasonable persistence (IRP)* assumption [4].

However, if an activity with effects is successfully executed, the system progresses from the state s to a successor state s'. This is illustrated by Fig. 3: The state s fulfills the precondition pre_{act}, then activity *act* is carried out, and the world progresses from s to s', which satisfies the effect eff_{act}.

6.3 Semantics of Activities

In the SESMA model, each service or process consists of a set of *activities*. Each activity can be written as a quadruple $act = \langle URI, Prec, Res, Var \rangle$, where *URI(act)* is a unique identifier of the activity, *Prec(act)* is the optional precondition formula of the activity, *Res(act)* is the set of pot. results of the activity and *Var(act)* are the variables used in the precondition and result formulas.

Each result $r \in Res(act)$ can be either an *effect* or a *knowledge effect*; each r may have a secondary precondition $SP(r)$ and a success condition $SC(r)$.

6.3.1 Conversation Data Sets

Conversation data sets capture the relevant data tokens that are exchanged between clients and services at runtime. Conversation data sets are an important prerequisite for evaluating SESMA formulas.

A conversation data set is a set of substitutions $\{\theta_1, ...\theta_n\}$. We denote a substitution as a finite set of the form $\theta = \{x_1/t_1, ..., x_n/t_n\}$, where each x_i is a distinct variable and each t_i is a constant, such that $x_i \neq t_i$. Note that the variables of an activity *act* are defined in the set $Var(act)$ (cf. Sect. 4.7). A substituted formula $F\theta$ is a variant of formula F where all variables x in F are replaced by a constant c if there is an element $x/c \in \theta$.

Let us consider the role of conversation data sets and substitutions in service execution:

- Substituting *input variables*: When invoking an activity *act*, the agent specifies the values certain input variables should have. This specification is represented by a conversation data set I. Each substitution θ in I contains at most one term t_i for every variable X_i of the input variables defined for the activity (cf. Sect. 4.7).

- Substituting *output variables*: Analogously, the values returned by the service are captured by a conversation data set O, which contains one or more substitutions ϑ, depending on the type of activity.

  Some activities (e.g. the `getPrice` operation in the example in Sect. 7) return just a single substitution, while other operations (e.g. the `getItemList` operation in the same example) return a set of substitutions. Each substitution contains at most one term t_o for every variable X_o of the output variables defined for the operation (cf. Sect. 4.7).

Let us illustrate this by some examples: For a particular invocation of an operation `getPrice`, the user might define the following input data set `I={{?item/MousePad }}`, and the service might return an output data set `O={{?price/4.00}}`. An operation `getItemList`, on the other hand, may return a data set O below, whereby the variables ?*item*, *price*?, ?*title*, ?*desc* are defined in the markup of the operations (cf. Sect. 4.7).

```
O = { {?item/2243, ?title/"MousePad", ?price/1.95
       ?desc/"Plastic mouse-pad, red"},
      {?item/2332, ?title/"SuperMouse", ?price/24.00
       ?desc/"wireless (IR) mouse for PC and MAC"} }
```

Next, we define how conversation data sets can be *combined*. We need combined conversation data sets to materialize effects (cf. Sect. 6.3.4). We start by defining the combination of substitutions: Given two substitutions $\theta = \{x_1/t_1, ..., x_k/t_k\}$ and $\vartheta = \{x_m/t_m, ..., x_n/t_n\}$, we define the joint substitution $\theta + \vartheta = \{x_1/t_1, ..., x_k/t_k, x_m/t_m, ..., x_n/t_n\}$. Given an input data set I and an output data set O, we define the *combined conversation data set* $C = I \otimes O$ as follows: For each $\theta \in I$ and each $\vartheta \in O$, C contains an element $\theta + \vartheta$.

6.3.2 Evaluating Preconditions

The precondition is evaluated against the state s, whereby the values in the conversation data set I provide the substitutions for the variables in the precondition. A precondition P is satisfied iff $s \models P\theta$ holds, for every substitution θ in the input data set I.

Since the agent is only aware of the subset s_A of s it must test the precondition against s_A. As long as the agent can

Name	Formula construct F	Semantics: $\mathcal{I}_\alpha \models F$...
Atomic Formula	`<P` $a_1 = t_1$, ..., $a_n = t_n$ `/>`	iff $(\mathcal{I}_\alpha[t_1], ..., \mathcal{I}_\alpha[t_n]) \in \mathcal{I}_\alpha[P]$, whereby P is the name of a predicate
Negation	`<not>A</not>`	iff $(\mathcal{I}_\alpha \not\models A)$
Conjunction	`<and>A B</and>`	iff $(\mathcal{I}_\alpha \models A)$ and $(\mathcal{I}_\alpha \models B)$
Disjunction	`<or>A B</or>`	iff $(\mathcal{I}_\alpha \models A)$ or $(\mathcal{I}_\alpha \models B)$
Implication	`<imply>A B</imply>`	iff $(\mathcal{I}_\alpha \not\models A)$ or $(\mathcal{I}_\alpha \models B)$
Universal Quantification	`<forall>` `<var name="X"/> A` `</forall>`	iff for every variable assignment α' differing from α in at most the value it assigns to x, $\mathcal{I}_{\alpha'}[A]$ is true

Table 1: The semantics of SESMA formulas

calculate the truth value of the precondition using the literals in s_A, it can conclude that the precondition is satisfied by s as well. However, if the agent has to resort to closed world assumption, i.e. if it assumes $s_A \not\models L^+$ for some positive literal $L^+ \notin s_A$, then there is the possibility that the evaluation against s_A differs from the evaluation against s.

6.3.3 Success Conditions

As already discussed, the success of a Web service operation may be undetermined until the execution is actually over; more generally, it is not clear a priori which of the results defined for the operation do occur. To provide agents with a tool to cope with this kind of uncertainty, the SESMA markup schema provides a construct `success-condition` which can be attached to each of the operation's results (cf. Sect. 4.6).

To decide which effects really occurred after an activity *act* was invoked, the success-conditions $SC(r)$ of the results $r \in Res(act)$ need to be evaluated by the evaluation function

$$eval(SC(r), s_A, I, O) \mapsto \{\top, \bot\}$$

The inputs of the evaluation function are the success condition SC of the result r whose success is evaluated, the pre-execution state s_A , the input data set I and the output data set O. The output of the evaluation function is a Boolean value, reflecting whether or not a defined result did indeed occur and if the result formula can be materialized. The calculus used to determine the result depends on the language attribute specified by the user (cf. Sect. 4.6).

If no success condition is defined for a result r, then *eval* returns `true` by default.

6.3.4 Materializing Effects

If an activity is executed (and if no error occurred), then the state s'_A can be calculated by adding the materialized effects whose optional success condition did not evaluate to *false*.

An effect formula E is *materialized* iff $s' \models E\vartheta$ holds for every ϑ in the combined data set $I \otimes O$. We write $A \models B$ iff every interpretation $\mathcal{I}$ that is a model of A is also a model of B.

These prerequisites allow us to specifiy the semantics of the *when* construct, which is only found in result formulas, not in precondition formulas: After a transition from state s to state s', the expression "`<when>A B</when>`" is true in state s' if and only if $(s \not\models A\theta)$ or $(s' \models B\vartheta)$ holds for all $\theta \in I$ and $\vartheta \in (I \otimes O)$, assuming that A and B are well formed SESMA formulas.

Having formally defined all relevant language constructs of SESMA, we can proceed to formally describe the truth value of an arbitrary atom A in a state using the function

$$val : P \times S \mapsto \{\top, \bot\}$$

which defines for each predicate in the sets of predicates P and for each state in the set of possible states S whether or not the predicate is true in that state. Given a state transition from state s to s' by invocation of activity *act* involving the conversation data sets I and O, the function is defined as follows:

$$val(s', A) = \begin{cases} \top, & \text{if } s \models Pre(act)\theta \text{ for every } \theta \in I \\ & \text{and if there is a result } r \in Res(act) \\ & \text{and } \exists\vartheta \in (I \otimes O) \text{ such that } r\vartheta \models A \\ & \text{and } s \models SP(r)\theta \text{ for every } \theta \in I \\ & \text{and } eval(SC(r), s_A, I, O) = \top \\ \bot, & \text{if } s \models Pre(act)\theta \text{ for every } \theta \in I \\ & \text{and if there is a result } r \in Res(act) \\ & \text{and } \exists\vartheta \in (I \otimes O) \text{ such that } r\vartheta \models \neg A \\ & \text{and } s \models SP(r)\theta \text{ for every } \theta \in I \\ & \text{and } eval(SC(r), s_A, I, O) = \top \\ val(s, A), & \text{elsewhere.} \end{cases}$$

The first case states that A is true in the new state s' if (i) the precondition formula (whose variables are substituted by the input values) is satisfied by the pre-execution state s and if (ii) A is a consequence of a result formula r (whose variables are substituted by the input and output values) and if (iii) the optional secondary precondition $SC(r)$ is true in s and if (iv) the evaluation function *eval* returns *true* after service execution. Analogously, the second case describes that A is not true in the new state, if $\neg A$ is a consequence of a result of the operation. The third case describes the situation where A is not affected by the execution of the operation, which addresses the frame problem of logical action descriptions. The definition does not deal with contradictory result definitions (e.g. having both P and $\neg P$ as an effect).

6.3.5 Semantics of Knowledge effects

From the agent's perspective, a knowledge effect is treated (and materialized) just like a normal effect. Nevertheless – as already pointed out in Sect. 4.4 – there is a difference, which is worth some additional elaboration: A knowledge effect KE of an activity describes an effect without side effects, i.e. it does not contribute to any changes occurring between s and s', but it may alter the world view s_A of the agent: the statement

$$[(s' \models KE\vartheta) \Rightarrow (s'_A \models KE\vartheta)] \wedge [(s' \not\models KE\vartheta) \Rightarrow (s'_A \not\models KE\vartheta)]$$

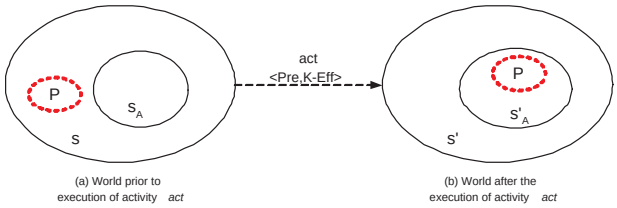

Figure 4: Meaning of a Knowledge Effect

holds, for every $\vartheta \in (I \otimes O)$.

Figure 4 visualizes the meaning of a knowledge effect. The activity that triggers the transition depicted in Fig. 4 has a knowledge effect P. The agent is not aware of it in the state s_A, but after execution of the activity, P is known to the agent in state s'_A.

7. ILLUSTRATING EXAMPLES

In the following we present examples of SESMA based service and process markup; the domain of the examples is an e-shopping domain, where several services offer various operations to retrieve information about goods and to purchase those goods.

We start by illustrating the semantic markup of a shopping mall service *megashop.com* and we describe how the precondition, result and success-condition formulas guide the interaction between agents and the described service. The WSDL description of the *megashop.com*-service can be found online at[4]. The service offers three operations, which are described below:

- `buyItem`: This operation expects a product ID, as defined by an EAN number, and a credit card number and expiration date as inputs. As a result, the ownership of the product will be transferred to the client.
- `getPrice`: This operation takes an EAN number of a commodity good as input and returns the price of it.
- `getItemList`: This operation does not have any inputs; when called, it returns the list of products in the catalog of the service. For each product, the EAN number, title and a description are presented.

7.1 The SESMA Annotation of the Service

The following SESMA description captures the semantics of the service (and its operations) relevant to our examples:

```
<annotation
  xmlns:wsdl="http://schemata.org/sws/wsdl"
  xmlns:s="http://foo.com/preds#"
  wsdl:url="http://sws.mcm.unisg.ch:8080/axis/services/
  MegashopService?wsdl"
  targetNamespace="http://sws.mcm.unisg.ch:8080/axis/
  services/MegashopService/"
  wsdl:serviceName="MegashopServiceService"
  xmlns="http://schemata.org/sws/sesma">
  <functional-profile>
    <!-- buy an item -->
    <act-def wsdl:name="buyItem"
             wsdl:portType="MegashopService">
      <input>
        <var name="?item" wsdl:part="ean"/>
        <var name="?cc" wsdl:part="ccNr" />
        <var name="?ccexp" wsdl:part="ccExpDate" />
      </input>
      <precondition>
        <and>
          <s:have-creditcard owner="$client" nr="?cc"
             expires="?ccexp"/>
          <s:in-catalog vendor="megashop.com"
             item="?item"/>
        </and>
      </precondition>
      <output>
        <var name="?result" wsdl:part="buyItemReturn"/>
      </output>
      <effect>
        <!-- to be checked after invocation -->
        <success-condition lang="beanshell">
          !("no".equals(output.get("?result")))
        </success-condition>
        <!-- the desired effect -->
        <s:possess owner="$client" item="?item"/>
      </effect>
    </act-def>
    <!-- get price quote -->
    <act-def wsdl:name="getPrice"
             wsdl:portType="MegashopService">
      <input>
        <var name="?item" wsdl:part="ean" />
      </input>
      <output>
        <var name="?p" wsdl:part="getPriceReturn"/>
      </output>
      <knowledge-effect>
        <s:price-at vendor="megashop.com"
           item="?item" price="?p" />
      </knowledge-effect>
    </act-def>
    <!-- retrieve content of product catalog -->
    <act-def wsdl:name="getItemList"
             wsdl:portType="MegashopService">
      <output>
        <var name="?item" wsdl:part="getItemListReturn"
             wsdl:path="Item/ean"/>
        <var name="?title" wsdl:part="getItemListReturn"
             wsdl:path="Item/title" />
        <var name="?desc" wsdl:part="getItemListReturn"
             wsdl:path="Item/description"/>
      </output>
      <knowledge-effect>
        <forall>
          <var name="?item" />
          <and>
            <s:in-catalog vendor="megashop.com"
               item="?item" />
            <s:has-title item="?item"
               title="?title" />
            <s:has-description item="?item"
               description="?desc" />
          </and>
        </forall>
      </knowledge-effect>
    </act-def>
  </functional-profile>
</annotation>
```

[4] `http://sws.mcm.unisg.ch:8080/axis/services/MegashopService?wsdl`

Note that at runtime, all occurrences of `$client` will be substituted by the agent's ID.

7.2 Ontology Concepts Used in the Samples

In the following, we describe the concepts used by the SESMA document shown above.

7.2.1 *Concept* `have-Credit-Card`

This concept represents the relation between agents (in this case, Web service users) and their credit cards.

Property-Name	Description
client	the Agent who is the credit card holder.
nr	the Credit Card.
expires	the expiration date of the credit card.

7.2.2 *Concept* `possess`

This concept represents the relation between agents and (commodity) goods. This construct is used to represent the (transfer of) ownership during Web service transactions.

Property-Name	Description
client	the agent who legally possesses the entity `item`.
item	the entity possessed by the agent.

7.2.3 *Concept* `price-at`

This concept represents at which price services offer goods for sale.

Property-Name	Description
service	the service who offers the item.
item	the item offered by the service.
price	the price at which the item is offered by the service.

7.2.4 *Concept* `has-title`

This concept represents the relation between wholesale goods and their title.

Property-Name	Description
item	the item.
title	the title of the item.

7.2.5 *Concept* `has-description`

This concept represents the relation between wholesale items and their descriptions.

Property-Name	Description
item	the item.
title	a description of the item.

7.2.6 *Concept* `in-catalog`

This relation represents whether an item is offered by a service (i.e. "listed in the catalog") or not.

Property-Name	Description
service	the service who offers the item.
item	the item offered by the service.

7.2.7 *Concept* `user-credentials`

This concept represents authentication data to be used by clients to log in at services.

Property-Name	Description
client	the client who is authenticated by the data captured by this relation.
service	the service where the client can use this data for authentication.
uid	the username.
pwd	the password.

7.3 Examples of Client-Service Interplay

In the following, we illustrate how the information captured in the SESMA description from Sect. 7.1 can be used to guide the interaction between an automated client and the annotated service. The service has the ID `megashop.com` and is described by the markup shown above in Sect. 7.1; the identity of the client is given by the ID `id.org/JohnDoe`.

7.3.1 *Example 1: Querying the price of a product*

Let us assume the following world state s, described by atomic formulas[5]

```
<n:in-catalog vendor="megashop.com" item="123" />
<n:in-catalog vendor="megashop.com" item="234" />
<n:has-title item="123" title="USB Mouse" />
<n:has-title item="234" title="USB Keyboard" />
<n:has-description item="123"
   description="transparent USB mouse" />
<n:has-description item="234"
   description="transparent USB keyboard"/>
<n:price-at vendor="megashop.com" item="123" price="99" />
<n:price-at vendor="megashop.com" item="234" price="49" />
<n:have-creditcard owner="id.org/JohnDoe"
   nr="32432434" expires="12/2005" />
```

Further, let us define a subset s_A of s, which represents the knowledge of the agent:

```
<n:have-creditcard owner="id.org/JohnDoe" nr="32432434"
expires="12/2005" />
```

Let us now assume that the agent wants to retrieve the price of the product "123" at vendor "megashop.com". To this end, the agent plans to call the operation `getPrice`. Since `getPrice` does not have a defined precondition, the request can be submitted without further considerations.

The input variable "?item" gets bound to "123", the ID of the product whose price is queried. Therefore, the conversation data set I for this operation call looks as follows: $I = \{\{?item/123\}\}$.

After invoking the operation, the agent receives the service's response. The agent uses the definition of the output variable `?p` to extract the conversation data set $O = \{\{?p/99\}\}$ from the response. The combined data set $I \otimes O$ is $\{?item/123, ?p/99\}$.

Since no error occurred and no success condition is defined for `getPrice`, the defined effect can materialize now, i.e. the effect formula, substituted by the contents of $I \otimes O$, is added to the agent's knowledge base:

```
<n:price-at vendor="megashop.com" item="123" price="99" />
```

7.3.2 *Example 2: Asking for a product list*

Suppose the agent wants to retrieve a list of the goods sold by megashop. To this end, the agent considers the operation

[5]the formulas are qualified by the namespace n of the concept definitions from Sect. 7.2.

`getItemList`, which can again be executed without further checks, because no precondition is defined for it. Hence, the agent invokes the operation, which does not have any input data, i.e. the conversation data set I is empty.

The service then returns a SOAP document that contains descriptions of the products currently in stock. Using the information captured in the `<output>` section of the operation's SESMA annotation, the agent can determine the following conversation data set O:

```
O = { {?item/123, ?title/"USB Mouse",
       ?desc/"transparent USB mouse"}
      {?item/234, ?title/"USB Keyboard",
       ?desc/"transparent USB keyboard" } }.
```

Since I is empty, the data set $I \otimes O$ is simply O. Again, no error occurred and no success condition exists; therefore, the effect of this operation can be materialized: For each element $\vartheta \in (I \otimes O)$, a substitution of the knowledge effect formula is created and added to the agent's fact base:

```
<n:in-catalog vendor="megashop.com" item="123" />
<n:in-catalog vendor="megashop.com" item="234" />
<n:has-title item="123" title="USB Mouse" />
<n:has-title item="234" title="USB Keyboard" />
<n:has-description item="123"
   description="transparent USB mouse" />
<n:has-description item="234"
   description="transparent USB keyboard"/>
```

Note that s has not been modified yet, because only operations with *knowledge-effects*, not *effects*, have been invoked so far.

7.3.3 Example 3: Purchasing a product

Suppose that the agent now wishes to purchase item "123", the transparent USB mouse. To this end, the agent plans to call the operation `buyItem`. Before it calls the operation, the agent first tests whether the precondition formula P of the operation is satisfied by the current state, i.e. it test whether $s \models P\theta$ holds, with $I = \{\{?item/123, ?cc = 32432434, ?expires = 12/2005\}\}$, the conversation data set for the request. The fact base s_A satisfies the precondition and the agent submits the request to the service.

Again, the response of the service is parsed using the information found in the `<output>` variable definition. The output conversion data set extracted from the service response is $O = \{\{?result/ok\}\}$. Before the effect of the operation can be taken for granted, the success condition needs to be evaluated:

```
<success-condition lang="Java">
  !("no".equals(output.get("?result")))
</success-condition>
```

Since `?result`, which is bound to "ok", does not equal "no", the Java(TM)-based evaluation of the fragment returns `true`, which means that the defined effect has indeed occured and can be added to the fact base.

The data sets $I \otimes O$, which is used to substitute the variables in the effect formula is: $\{\{?item/123, ?cc = 32432434, ?expires = 12/2005, ?result/ok\}\}$. Therefore the following atom is added to the new world state s':

```
<s:possess owner="id.org/JohnDoe" item="123" />
```

7.4 Example of a process annotation

In the following example we illustrate how to annotate BPEL processes in SESMA. Consider the following example of a business process: The process takes an order from a client which contains the IDs of a collection of products to be purchased, along with authentication credentials and credit card data.

The process defines the following behavior: The actor first logs in at the defined Web service *gigashop.com*. Then, it puts all desired items into the virtual shopping cart of the service, and then it finalizes the transaction by invoking the Web services' checkout method. After successful execution of the process, the user owns the desired collection of goods.

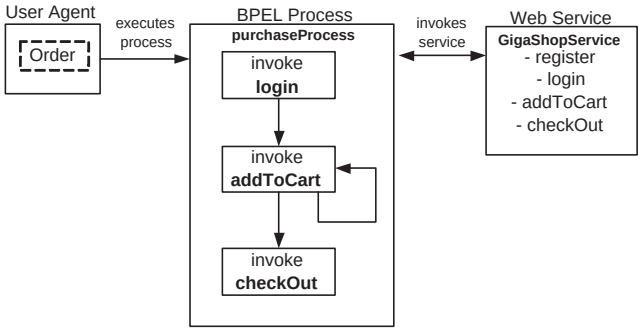

Figure 5: Example Web service process

An illustration of the process is depicted in Fig. 5. The WSDL and BPEL definitions of the process can be retrieved from[6]. The process can be annotated as follows:

```
<annotation target="BPEL4WS-1.1"
  xmlns="http://schemata.org/sws/sesma"
  xmlns:wsdl="http://schemata.org/sws/wsdl"
  xmlns:s="http://foo.com/preds#"
  wsdl:url="http://sws.mcm.unisg.ch:8080/axis/
            services/GigashopService?wsdl"
  wsdl:serviceName="GigashopServiceService"
  bpel:url="http://localhost:8080/bpel/purchase_process.bpel"
  bpel:name="ShoppingProcess">
  <import-voc url="http://schemata.org/preds_shopping.xml"/>
  <functional-profile>
    <act-def name="startPurchaseProcess"
      wsdl:portType="purchaseProcessPT">
      <input>
        <var name="?item" wsdl:part="items"
        <var name="?uid" wsdl:part="username" />
        <var name="?pwd" wsdl:part="password" />
        <var name="?cc" wsdl:part="ccNr" />
        <var name="?ccexp" wsdl:part="ccExpDate" />
      </input>
      <precondition>
        <and>
          <s:have-creditcard nr="?cc" expires="?ccexp"/>
          <s:user-credentials client="$client"
             server="gigashop.com"
             username="?uid" password="?pwd" />
        </and>
      </precondition>
      <effect>
        <forall>
          <var name="?item" />
          <when>
```

[6] `http://elektra.mcm.unisg.ch/sesma/process-example.html`

```
          <s:in-catalog vendor="gigashop.com"
             item="?item" />
          <s:possess owner="$client" item="?item" />
        </when>
      </forall>
    </effect>
  </act-def>
 </functional-profile>
</annotation>
```

As described in the SESMA annotation, the precondition of the process are (i) the credentials to access the retailer service and (ii) the availability of credit card data. The effect of the process is that all ordered goods are in possession of the agent after the execution, given that the goods are in the catalog of the retailer service.

In this example, the WSDL-defined operation `startPurchaseProcess` serves as the interface to the BPEL based process definition; if such a description is not available, then it would be possible to refer directly to the main sequence of the process:

```
<act-def bpel:path="/process/sequence[0]">
  <!-- markup of the sequence -->
</act-def>
```

Similarly, other process fragments (e.g. the while loop) could be selectively targeted via the respective XPointer expression in *bpel:path*.

8. SUMMARY

In this paper we presented the SESMA markup format proposed for semantic Web service and process annotation.

The aim of the SESMA effort is to deliver a markup format that provides the language constructs needed to define the semantics of services and processes in a developer friendly manner: This encompasses not only a concise XML based syntax and convenient support for nondeterministic operations but also a well defined alignment with mainstream standards like BPEL to avoid unnecessary overlaps with those standards.

The present paper gave an overview of SESMA's syntax and a detailed account of its semantics. It also discussed how SESMA annotations can be used to characterize Web service operations and process activities in a machine interpretable way and it presented a collection of illustrating examples.

9. REFERENCES

[1] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language 1.0 reference.

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

[3] IBM, Microsoft, and BEA. Business process execution language for web services, version 1.0, 2002.

[4] S. McIlraith and T. Son. Adapting golog for programming in the semantic web, 2001.

[5] N. Noy and A. Rector. Defining N-ary Relations on the Semantic Web: Use With Individuals. W3C Technical Report (Working Draft), 2004.

[6] OWL-S Coalition. OWL Web Services 1.1, http://www.daml.org/services/, 2004.

[7] W3C. Web Services Description Language (WSDL) Version 1.2, 2002.

[8] W3C. XML Pointer Language (XPointer), 2002.

[9] WSMO Working Group. Web Service Modeling Ontology (WSMO), http://www.wsmo.org, 2004.