Logic-based Web Services Composition: from Service Description to Process Model

Jinghai Rao, Peep Küngas
Department of Computer and Information Science
Norwegian University Science and Technology
N-7491, Trondheim, Norway
Email: {jinghai,peep}@idi.ntnu.no
Tel: +47 7359 4480, Fax: +47 7359 4466

Mihhail Matskin
Department of Microelectronics
and Information Technology
Royal Institute of Technology,
SE-164 40 Kista, Sweden
Email: misha@imit.kth.se
Tel: +46 8 790 4128, Fax: +46 8 751 1793

Abstract

This paper introduces a method for automatic composition of Semantic Web services using Linear Logic (LL) theorem proving. The method uses Semantic Web service language (DAML-S) for external presentation of Web services, while, internally, the services are presented by extralogical axioms and proofs in LL. We use a process calculus to present the composite service formally. The process calculus is attached to the LL inference rules in the style of type theory. Thus the process model for a composite service can be generated directly from the proof. The subtyping rules that are used for semantic reasoning are presented with LL inference figures. We propose a system architecture where the DAML-S translator, the LL theorem prover and the semantic reasoner can operate together to fulfill the task. This architecture has been implemented in Java.

1. Introduction

Recent progress in the field of Web services has made it practically possible to publish, locate, and invoke applications across the Web. This is a reason why more and more companies and organizations now implement their core business and outsource other application services over Internet. The ability for efficient selection and integration of inter-organizational and heterogeneous services on the Web at runtime becomes an important requirement to the Web service provision. In particular, if no single Web service can satisfy the functionality required by a user, there should be a possibility to combine existing services together in order to fulfill the request. This trend has triggered a considerable number of research efforts on Web services composition both in academia and in industry.

Several Web services initiatives have provided platforms and languages that should allow easy use of Web services. In particular, Universal Description, Discovery, and Integration (UDDI) [3], Web Services Description Language (WSDL) [6], Simple Object Access Protocol (SOAP) [4] and parts of DAML-S [14] ontology (including ServiceProfile and ServiceGrounding) define standard ways for service discovery, description and invocation (message passing). Some other initiatives including Business Process Execution Language for Web Service (BPEL4WS) [2] and DAML-S ServiceModel, are focused on representing service compositions where a process flow and bindings between services are known a priori.

The problem of Web service composition is a highly complex task. Here we underline only the following two sources of its complexity. First, Web services can be created and updated on the fly, and it is a problem to analyze a huge amount of dynamic services and to compose them. Second, the Web services are usually developed by different organizations that use different semantic model for presenting services' characteristics and this requires utilization of relevant semantic information for matching and composition of Web service.

In this paper, we propose a solution that allows decreasing the complexity of Web services composition task emerging from the above-mentioned sources.

First, we present a method for automated Web service composition which is based on the proof search in a fragment of propositional Linear Logic (LL) [8]. The main idea of the method is as follows. Given a set of existing Web services, the method finds a composition of atomic services that satisfies the user requirements. The reason why we use propositional LL here is that it provides the expressive power that allows us to describe both functionalities and non-functional attributes of Web services. Because

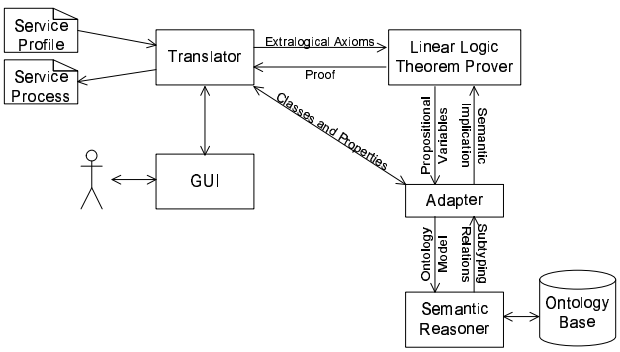

Figure 1. The architecture of the service composition system.

of soundness of the logic fragment used the correctness of composite services is guaranteed with respect to the initial specification. Completeness of the logic fragment ensures that if a composable solution exists then it will be found .

Second, the composition approach we present allows reasoning with types from a service specification using Semantic Web language. We introduce a set of subtyping rules that define a valid dataflow for composite services. The subtyping relationship between the classes or properties are defined in the domain ontology. Because we define the subtyping rules as logical implications, the interoperability is ensured between the LL theorem prover and the semantic reasoner.

Web service composition using theorem proving is a relatively new topic, and it has been mostly mentioned in quite recent publications. However, the idea of software construction from proofs is not new. In particular, deductive program synthesis is based on an observation that constructive proofs are equivalent to programs where each step of a proof can be interpreted as a step of a computation. The key ideas of the software composition approach, as well as correspondence between theorems and specifications and between constructive proofs and programs, are presented in [13].

The rest of this paper is organized as follows: Section 2 describes a system architecture for composition of Semantic Web services. Section 3 presents a method for transformation of DAML-S ServiceProfile to extralogical axioms in LL. Section 4 discusses how to extract a process from a proof. Section 5 introduces the prototype and its implementation. Finally, related work and conclusion are presented.

2. The service composition architecture

A general architecture of the proposed Web service composition system is depicted on Figure 1. The basic components of the system are as follows:

Translator performs a transformation between an external presentation of Web services and extralogical axioms in LL. In our system, DAML-S ServiceProfile is used externally for presentation of Semantic Web service specification, while LL axioms are used internally for planning and composition. The process model is internally presented by a process calculus. The calculus can be translated into either DAML-S or BPEL4WS.

GUI visualizes services (both composed and atomic). The graphical presentation includes visualization of functionalities and non-functional attributes.

LL theorem prover proves whether the user's request for service can be achieved by a composition of the available atomic services. If the answer is positive, the process model for the composite service is automatically extracted from the proof.

Semantic reasoner detects the subtyping and some other relationships between concepts in service description. The formal logics used in semantic reasoner could be logics developed for expressing knowledge and reasoning about concepts and concept hierarchies, for example, Description Logic [9]. A transformation between Description Logic and LL is done by the adapter.

Adapter performs translation between LL and the internal presentation used by the semantic reasoner.

A composition process in our case is as follows. First, a semantic description of existing Web services (in the form of DAML-S ServiceProfile) is translated into extralogical axioms of LL and the user's request for a composite service is specified in the form of a LL sequent to be proven. Second, the adapter asks the semantic reasoner to analyse the subtype relations of the classes and properties in the domain ontology and sends the relations as LL axioms to the LL theorem prover. Third, the LL theorem prover checks whether the request can be satisfied by composition of existing atomic services (this is done by performing theorem proving in LL). If the sequent corresponding to the requested composite service has been proven and the proof is generated then a process calculus presentation is extracted from the proof directly. The last step is construction of flow models - the process calculus is translated to either DAML-S ServiceModel or BPEL4WS upon the request. During the composition and execution processes, the user is able to monitor the system via GUI.

3. Transforming from DAML-S Profile to Linear Logic Axioms

3.1. Linear Logic

LL is a refinement of classical logic introduced by J.-Y. Girard [8] to provide a means for keeping track of "resources"—in LL two assumptions of a propositional constant A are distinguished from a single assumption of A. Although LL is not the first attempt to develop resource-oriented logics (well-known examples are relevance logic [7] and Lambek calculus [11]), it is by now one of the most investigated one. Therefore, because of its maturity and well-developed semantic, LL is useful as a declarative language and an inference system.

The resource-oriented feature of LL provides more expressive power for Web service specification. First, because the conjunction of two propositional constant A is not equal to a single occurrence of A in LL, this allows specification of both counted resources and multiple service parameters having the same type. For example, two service parameters of the integer type can be specified as $Int \otimes Int$. On the contrary, in classical logic, this specification is equal to a single Int and we lose the computational meaning of the specification. Second, by using "of course" modality, we can distinguish the consumable type (e.g. time and money) and non-consumable type (e.g. information) in the descriptions of Web services.

The syntax of the LL fragment that we use in this paper is presented by the following grammar:

$$A ::= \quad P | A \multimap A | A \otimes A | A \oplus A | !A | \mathbf{1}.$$

The logic fragment A consists of proposition P, multiplicative conjunction $\otimes$, additive disjunction $\oplus$, linear implication $\multimap$ and "of course" (!) modality. In terms of resource acquisition the logical expression $A \otimes B \multimap C \otimes D$ means that resources C and D are obtainable only if both A and B are available to be consumed. Thus the connective $\otimes$ defines deterministic relations between resources. In that way we can encode different behaviors of computations. The disjunction $A \oplus B$ defines that either A or B is consumed or generated. The formula $!A$ means that we can use or generate a literal A as much as we want—the amount of the resource is unbounded. While in classical logic literals may be copied by default, in LL this has to be stated explicitly.

After the available services are specified in the form of LL extralogical axioms and a requested service is specified as a theorem to be proven, we use LL theorem proving to determine whether the requested service can be composed. If a proof of the theorem exists then a composed service is extracted from the proof. A structure of the composed service reflects the structure of the proof. Since the composed service is guaranteed to meet the specification, no further verification is needed.

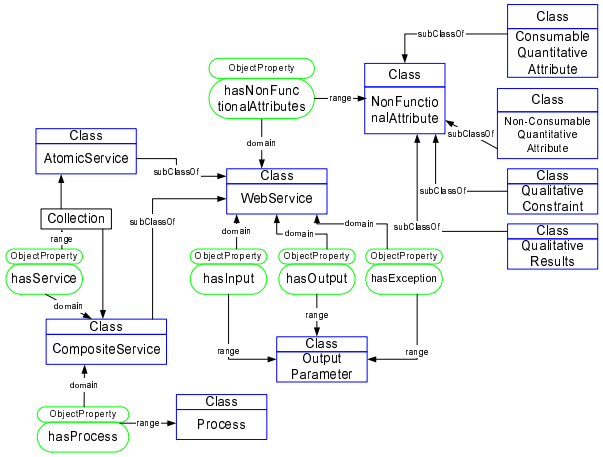

Figure 2. The upper ontology for Web services.

3.2. The upper model of Web services and LL

Figure 2 is the upper ontology of Web services. A Web service is either an atomic service or a composite service. The composite service contains a collection of atomic services in addition to the process model describing the control- and data-flow among the atomic services. Both atomic services and composite services are specified by functionalities and non-functional attributes. The functionalities are represented as a transformation from the inputs required by the service to the output produced by the service. They include inputs, outputs and exceptions. The non-functional attributes are other properties than functionalities that can be used to describe a service (for example, price, location, quality of the service). They are classified into four categories: consumable quantitative, non-consumable quantitative, qualitative constraints and qualitative results.

Generally, a requirement to composite Web service (including functionalities and non-functional attributes) can be expressed by the following LL formula:

$$\Gamma; \Delta_c \vdash (I \multimap (O \oplus E)) \otimes \Delta_r$$

where Γ is a set of extralogical axioms representing available atomic Web services. Δ_c is a conjunction of non-functional constraints. Δ_r is a conjunction of non-functional results. $I \multimap (O \oplus E)$ is a functionality description of the required composite service. Both I and O are conjunctions of literals, I represents a set of input parameters of the service and O represents output parameters pro-

duced by the service. E is a presentation of an exception. Intuitively, the formula can be explained as follows: given a set of available atomic services and non-functional attributes, try to find a combination of services that computes O from I. If the computation fails, an exception is thrown. Every element in Γ is in form $\Delta_c \vdash (I \multimap (O \oplus E)) \otimes \Delta_r$, where meanings of Δ_c, Δ_r, I, O and E are the same as described above.

In the following we will discuss the detail of transformation from the Web service documents to the LL axioms. This paper focuses on the transformation of the functionalities. The non-functional part has been discussed in a separate publication [18].

3.3. Transformation of functionalities

In our system a Web service is presented by DAML-S ServiceProfile. The functionality attributes of the "ServiceProfile" specify the computational aspect of the service, denoting inputs, outputs and exception of the service. The functionality attributes are used in composition for connecting atomic services by means of inputs and outputs. The composition is possible only if output of one service can be transferred to another service as an input.

The following DAML-S example presents a service that recommends the ski model according to the user's skill level:

```
<profileHierarchy:InformationService rdf:ID="SelectModel">
  <profile:serviceName>SelectModel</profile:serviceName>
  <profile:textDescription>
    The service outputs ski model given user's skill level
  </profile:textDescription>
  <profile:input>
    <profile:ParameterDescription rdf:ID="sms">
      <profile:parameterName> Ski_Skill_Level
                              </profile:parameterName>
      <profile:restrictedTo rdf:resource="&onto;#Skill"/>
      <profile:refersTo rdf:resource="&model;#sms"/>
    </profile:ParameterDescription>
  </profile:input>
  <profile:output>
    <profile:ParameterDescription rdf:ID="smm">
      <profile:parameterName> Ski_Model
                               </profile:parameterName>
      <profile:restrictedTo rdf:resource="&onto;#Model"/>
      <profile:refersTo rdf:resource="&model;#smm"/>
    </profile:ParameterDescription>
  </profile:output>
</profileHierarchy:InformationService>
```

From the computation point of view, this service requires an input that has type "`&onto;#Skill`"(the value of user's ski skill level) and produces an output that has type "`&onto;#Model`"(the recommended ski model). Here, we use entity types as a shorthand for URIs. For example, "`&onto;#Skill`" refers to the URI of the definitions for the DAML class to measure the ski Skill level: `http://bromstad.idi.ntnu.no/Ontology/ski.daml#Skill`. When translating the ServieProfile to a LL formula, we translate the field "restrictedTo" (variable type) instead of the parameter name, because we regard the parameters' types as their specification. An example of LL formula that describes the above presented DAML-S document is as follows:

$$\vdash \texttt{\&onto;\#Skill(\&profile;\#sms)} \multimap_{\texttt{\&profile;\#SelectModel}} \texttt{\&onto;\#Model(\&profile;\#smm)}$$

The above formula is different from the regular propositional LL formula and provides some supplementary information for generation of a process. First, the value of "parameterID" field is presented in parentheses after the parameter's type. Second, the service ID is attached to the implication symbol. We would like to underline that this supplementary information is not used by the LL theorem prover and it is utilized only for extraction of process model from the proof.

3.4. Presentation of the domain ontology

For Web service composition, the subtype relation is the most important semantic relation. In general, if an output of one service is subtype of an input of another service, it should be safe to transfer data from the output to the input. For the ontology languages based on RDFS [5] (for example, DAML and OWL) the subtype relationships are specified by the transitive and symmetric properties of *rdfs:subPropertyOf* and *rdfs:subClassOf*. We express these properties as subtyping rules. In order to emphasize that our new inference rules are used for subtyping purposes (but not for sequencing components) we further write $\multimap_<$ to denote the subtype relations. It should be also mentioned that the subtyping rules are not an extension to LL. Indeed, these rules are defined as certain inference figures in LL.

Some useful subtype rules are as follows. The small letters inside the parentheses are the variables that have the type indicated by the capital letters in front.

$$\frac{\Sigma \vdash T \multimap_< S \qquad \Gamma \vdash S \multimap_< U}{\Sigma, \Gamma \vdash T \multimap_< U} \quad \textit{Subtyping transitivity}$$

$$\frac{\Sigma \vdash T(t) \qquad \Gamma \vdash T \multimap_< S}{\Sigma, \Gamma \vdash S(t)} \textit{Goal subtyping}$$

$$\frac{\Sigma, S(s) \vdash G \qquad \Gamma \vdash T \multimap_< S}{\Sigma, \Gamma, T(s) \vdash G} \textit{Resource subtyping}$$

In our system subtype relations between propositional variables are discovered by the semantic reasoner. Several available semantic reasoners (for example, FaCT [10] or Jena [1]) can be utilized here.

We also assume that the ontology used by the service requesters and by the service providers are interoperable. Otherwise, the ontology integration should be done and this is beyond the scope of this paper.

4. Extract Process Model from the Proof

4.1. Proving in Linear Logic

After service specifications in DAML-S are translated into LL extralogical axioms and a service request from the customer is translated into the form of theorem to be proven, the LL prover is invoked. It tries to prove the specified theorem using LL inference rules. In this paper we are mostly focused on extraction of process model from the LL proof rather than on the proving process. However, some hints about LL theorem proving can be obtained from the example description in the Section 4.3.

A composite service that satisfies the customer request will be found only if the theorem specifying the customer service request can be proven. Otherwise the request should be re-formulated or existent services specification should be modified.

Finally, a process model of the composite service is extracted from the completed proof generated by the LL theorem prover. The process model is presented formally by a process calculus. In the next section we link the proof to the process calculus by attaching proof terms directly to the deduction rules in the style of type theory.

4.2. The process model

We use a process calculus for presentation of process model. The process calculus is built based on the idea of π-calculus [16] – a powerful model of concurrency, which can be used to design communication-based programming languages. The π-calculus is widely used for modeling the composite Web services. One recent example is the XLANG Language proposed by Microsoft [20]. This language is explicitly build on a model from the π-calculus. The most popular process languages, such as BPEL4WS and DAML-S ServiceModel, can be also adapted to π-calculus, although they do not announce π-calculus as their formality foundation.

The process calculus is built from the operators of inaction, input prefix, output prefix, parallel composition, global choice and restriction. The process calculus grammar is as follows (where the names starting with small letters range from messages to variables, and the names starting with capital letters refer to processes):

$$P ::= \mathbf{0} \Big| a(x).P \Big| \overline{a}\langle x\rangle.P \Big| P|P \Big| P+P \Big| (\nu a)P$$

The meaning of the operators can be explained in terms of BPEL4WS. $\mathbf{0}$ is the inactive process which is "empty" activity in BPEL4WS. An input prefixed process $a(x).P$ receives a variable or message x through channel a then executes process P. This operation equals to the "receive" action in BPEL4WS. An output $\overline{a}\langle x\rangle.P$ emits message x at

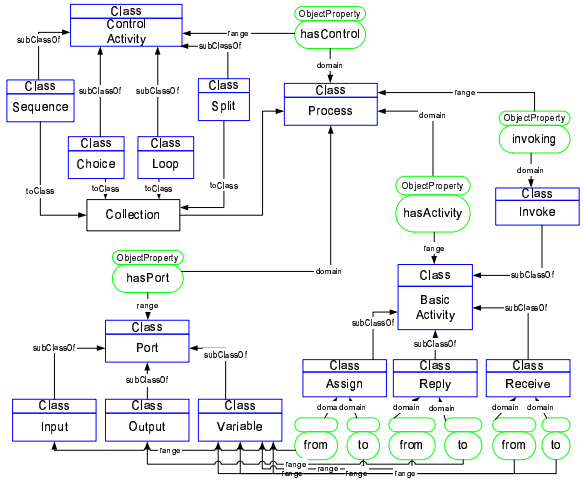

Figure 3. The upper ontology for the flow model.

channel a. This is the "reply" action in BPEL4WS. The restriction $(\nu a)P$ defines a name a local to P. The name a is similar to the local variable inside the BPEL4WS process. For the rest of operations, . is a sequence, | is a "flow" and $+$ is a "switch".

At conceptual level the process calculus is specified by the upper-ontology shown in Figure 3. The multiple processes are collected by programming constructs, such as, "sequence", "choice", "split" and "loop". Each process has a set of activities. Some activities, for instance "Assign", "Reply" and "Receive" can copy the data among inputs, outputs and internal variables. The "Invoke" activity invokes a process.

The data transferring activities for the service composition problem in the context of process calculus is presented by three parts including output channel, input channel and local variables. For example, if a service P outputs the result through channel a, and then the result is passed to channel b that is an input channel of service Q then the process is presented as follows:

$$(\nu x)(P.\bar{a}\langle x\rangle.b(x).Q)$$

Here x is a local variable that indicates the data transferring between the output channel a and input channel b, both of which are global accessible. When thinking about the abstract process model, the local variables can be assigned to arbitrary name and the above process can be simplified to:

$$P.\bar{a}b.Q$$

Such presentation is close to a Business Process Contract in XLANG, or a "process:sameValues" in DAML-S data flow. Both of them use a set to show the connection of

Logical axiom and Cut rule:

$$A \vdash 0 : A \ (Id) \qquad \frac{\Gamma \vdash P : A(a_1) \quad \Gamma', A(a_2) \vdash Q : C}{\Gamma, \Gamma' \vdash (P.\{\overline{a_1}a_2\}.Q) : C} \ (Cut)$$

Rules for propositional constants:

$$\vdash 1 \qquad \frac{\Gamma \vdash A}{\Gamma, 1 \vdash A}$$

$$\frac{\Gamma, A, B \vdash P : C}{\Gamma, A \otimes B \vdash P : C} \ (L\otimes) \qquad \frac{\Gamma \vdash P : A(a) \quad \Gamma' \vdash Q : B(b)}{\Gamma, \Gamma' \vdash (P|Q) : A \otimes B(a,b)} \ (R\otimes)$$

$$\frac{\Gamma \vdash A \multimap_P B}{\Gamma, A \vdash P : B} \ (Shift) \qquad \frac{\Gamma, A(a) \vdash P : B(b)}{\Gamma \vdash A \multimap_{a.P.\overline{b}} B} \ (R\multimap)$$

$$\frac{\Gamma, P : A \vdash \Delta \quad \Gamma, Q : B \vdash \Delta}{\Gamma, (P+Q) : (A \oplus B) \vdash \Delta} \ (L\oplus)$$

$$\frac{\Gamma \vdash P : A}{\Gamma \vdash P : A \oplus B} \ (R\oplus)(a) \qquad \frac{\Gamma \vdash Q : B}{\Gamma \vdash Q : A \oplus B} \ (R\oplus)(b)$$

Rules for exponential !:

$$\frac{\Gamma \vdash \Delta}{\Gamma, !A \vdash \Delta} \ (W!) \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta} \ (L!) \qquad \frac{\Gamma, !A, !A \vdash \Delta}{\Gamma, !A \vdash \Delta} \ (C!)$$

Structural congruence for a process calculus:

$$P|Q \equiv Q|P \qquad P+Q \equiv Q+P \qquad (P|Q)|R \equiv P|(Q|R)$$
$$(P+Q)+R \equiv P+(Q+R)$$
$$P \equiv P|\mathbf{0} \equiv P+\mathbf{0} \equiv P.\mathbf{0} \equiv \mathbf{0}.P \qquad A(a) \otimes B(b) \equiv A \otimes B(a,b)$$
$$\overline{(a_1, a_2, \ldots, a_n)}(b_1, b_2, \ldots, b_n) \equiv (\overline{a_1}b_1, \overline{a_2}b_2, \ldots, \overline{a_n}b_n)$$

Table 1. Inference rules.

one output parameter (a) of a service (P) with one input parameter (b) of another service (Q).

In order to make the relationship between the proof and the process model more concrete we attach process calculus description directly to the inference rules in the style of type theory. In this way the process calculus description for the composite service is formed as soon as the proof is finished. The extended LL inference rules that refer to the process calculus description are presented in Table 1.

4.3. Example

In order to illustrate our composition method let us consider a simplified service for recommending skies to the customer according to his/her size and skiing skill level. We consider only the functionalities in the example, but the proving process is same after introducing the non-functional attributes.

Available atomic services are specified in DAML-S in a way described in the Section 3.3. These services specifications are automatically translated to the following axioms (for the sake of readability, we omit the namespace of the parameters):

Axioms:

$$\vdash Skill(sms) \multimap_{SelectModel} Model(smm)$$
$$\vdash Model(ssm) \otimes Length(ssl) \multimap_{SelectSki} ProductNr(ssp) \otimes SportShop(sss)$$
$$\vdash Height(slh) \otimes Weight(slw) \multimap_{SelectLength} Length(sll))$$
$$\vdash ProductNr(gpn) \otimes Shop(gps) \multimap_{GetPrice} Price(gpp)$$
$$\vdash SportShop \multimap_{>} Shop$$

The axioms cam be explained as follows:

- *SelectModel* — given a skill level, provides a brand;
- *SelectLength* — given body height and body weight, provides the recommended ski length;
- *SelectSki* — given a model and a ski length, provides a sport shop name and the product number (ProductNr) of the specific skis;
- *GetPrice* — given the product number and the shop name provides price. Here the shop is a general shop that subsumes the sport shop;
- $>$ — specified by the domain ontology that the sport shop is a subtype of shop;

The customer request for price of skis according to her/his skill level and body height and weight is translated into the following theorem:

Goal/Theorem

$$\vdash Skill \otimes Height \otimes Weight \multimap Price$$

Using the inference rules from the Table 1 the LL theorem prover generates a proof of the theorem (this proof is shown in Figure 4).

And the following process calculi is extracted from the proof:

$$(sms, slh, slw).(SelectModel|SelectLength).(\overline{smm}ssm, \overline{sml}ssl)$$
$$.SelectSki.(\overline{ssp}gpn, \overline{sss}gps).GetPrice.\overline{gpp}$$

This process model can be illustrated by the following script that is similar to the presentation of BPEL4WS. The process model is able to be translated into other composite Web service models, such as DAML-S ServiceModel.

```
<sequence>
  <receive input="sms"/>
  <receive input="slh"/>
  <receive input="slw"/>
  <split>
    <invoke process="SelectModel"/>
    <invoke process="SelectLength"/>
  </split>
  <copy from="smm" to="ssm"/>
  <copy from="sml" to="ssl"/>
  <invoke process="SelectSki"/>
  <copy from="ssp" to="gpn"/>
  <copy from="sss" to="gps"/>
  <invoke process="GetPrice"/>
  <reply output="gpp"/>
</sequence>
```

5. Prototype

A prototype of the system presented in Figure 1 has been implemented in Java. In the prototype, Jena (a Java framework for building Semantic Web applications) [1] is used to facilitate both the translator and semantic reasoner. In

$$\dfrac{\dfrac{\vdash Skill(sms) \multimap_{SelectModel} Model(smm)}{Skill(sms) \vdash SelectModel : Model(smm)}\ Shift \quad \dfrac{\vdash Height(slh) \otimes Weight(slw) \multimap_{SelectLength} Lenght(sll))}{Height \otimes Weight(slh, slw) \vdash SelectLength : Lenght(sll)}\ Shift}{Skill \otimes Height \otimes Weight(sms, slh, slw) \vdash (SelectModel|SelectLength) : Model \otimes Length(smm, sll)}\ R\otimes$$

...

$$\dfrac{\dfrac{\vdash ProductNr(gpn) \otimes Shop(gps) \multimap_{GetPrice} Price(gpp)}{ProductNr(gpn) \otimes Shop(gps) \vdash GetPrice : Price(gpp)}\ Shift \quad \dfrac{subtyping}{\vdash SportShop \multimap_{>} Shop}}{\dfrac{PruductNr(gpn) \otimes SportShop(gps) \vdash GetPrice : Price(gpp)}{ProductNr \otimes SportShop(gpn, gps) \vdash GetPrice : Price(gpp)}\ Structual\ Congruence}\ Resource\ subtyping$$

...

$$\dfrac{\dfrac{\dfrac{\vdash Model(ssm) \otimes Length(ssl) \multimap_{SelectSki} ProductNr(ssp) \otimes SportShop(sss)}{Model(ssm) \otimes Length(ssl) \vdash SelectSki : (ProductNr(ssp) \otimes SportShop(sss))}\ Shift}{Model \otimes Length(ssm, ssl) \vdash (SelectSki.(\overline{ssp}, \overline{sss})(gpn, gps).GetPrice) : Price(gpp))}\ Structual\ Congruence}{Skill \otimes Height \otimes Weight(sms, slh, slw) \vdash ((SelectModel|SelectLength).(\overline{smm}, \overline{sll})(ssm, ssl).SelectSki.(\overline{ssp}, \overline{sss})(gpn, gps).GetPrice) : Price(gpp)}\ cut$$

$$\dfrac{Skill \otimes Height \otimes Weight(sms, slh, slw) \vdash ((SelectModel|SelectLength).(\overline{smm}ssm, \overline{sll}ssl).SelectSki.(\overline{ssp}gpn, \overline{sss}gps).GetPrice) : Price(gpp)}{\vdash Skill \otimes Height \otimes Weight \multimap_{(sms,slh,slw).(SelectModel|SelectLength).(\overline{smm}ssm,\overline{sml}ssl).SelectSki.(\overline{ssp}gpn,\overline{sss}gps).GetPrice.\overline{gpp}} Price}\ R\multimap$$

$$cut \quad Stru.\ Cong.$$

Figure 4. The example proof

particular, the "TransitiveReasoner" in Jena provides support for storing and traversing class and property lattices as graph structures.

Our earlier developed LL theorem prover that we use in the prototype supports reasoning both with propositional and first-order LL specification. The prover can be downloaded from `http://www.idi.ntnu.no/~peep/RAPS`.

Basic GUI features of the prototype are depicted in Figure 5 that illustrates the result of the example from the Section 4.3. The interface of the required service is presented in the **ServiceProfile** panel (upper right) and the dataflow of the component atomic services is presented in the **ServiceModel** panel (lower right). This screenshot shows the dataflow among the three atomic services to fulfill the functionality of the required service. For each Web service, the detailed information of functionalities and non-functional attributes is displayed in the left hand side panel. The bottom panel demonstrates the semantic relationship between the parameters.

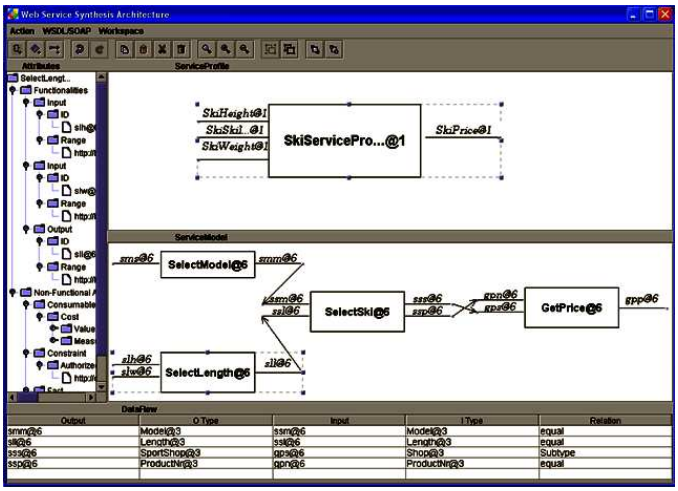

Figure 5. A screen shot.

6. Related work

AI planning has been applied for composition of Semantic Web services. In [15], the authors adapt and extend the Golog language for automatic construction of Web services. Golog is a high-level logic programming language built on top of situation calculus. The authors addressed the Web service composition problem through the provision of high-level generic procedures and customizing constraints. Golog is used as a natural formalism for this problem.

SWORD[17] is a developer toolkit for building composite Web services. SWORD doesn't deploy emerging service-description standards such as WSDL and DAML-S, instead, it uses ER model to specify the inputs and outputs of Web services. As a result, the reasoning is based on the entity and attribute information provided by ER model.

[19] presents a semi-automatic method for Web service composition. Whenever a user selects a web services, all services that are possible to connect to the selected service are presented to the user. The choice of the possible services is based on both functionalities and non-functional attributes. The functionalities (parameters) are presented by an OWL class and an OWL reasoner are used to match the services. Afterward, the system filter the services based on the non-functional attributes that are specified by the user as constraints.

7. Conclusion and future work

In this paper we describe an approach to automatic Semantic Web service composition. Our approach has been directed to meet the two main challenges in service composition, namely, automated composition and semantic reasoning. First, DAML-S service descriptions are automatically translated into LL axioms. Then the LL theorem prover proves the possibility to compose the required service from available atomic services. A process model of the composed service is constructed automatically from a proof (if the proof exists). The semantic reasoner is exploited as an auxiliary component to relax the service matching process while selecting and connecting atomic services. Finally, the result is presented to the user through a graphical user interface.

We argue that LL theorem proving, combined with semantic reasoning offers a flexible approach to the successful composition of Web services automatically. Comparing to the related work, an interesting feature of our method is that the service is composed by theorem proving. We also propose an architecture to support the whole composition process. A prototype of the approach has been implemented.

Our future work is directed towards improving efficiency of both the LL theorem prover and the semantic reasoner. Since the amount of the available services and the size of ontology models are huge it is necessary to reduce the search space during problem solving.

There are also some other extensions of the current composition method that are under our consideration. First, our experience with the Web service composition shows that users are not always able to completely specify the goal of required service. We consider to apply the principle of partial deduction [12] to provide more flexibility to the user. Second, the composition of services usually needs business model to specify the relationship between multiple services. The method presented in the paper is to be extended for service composition if the business model is given.

Acknowledgments

This work is partially supported by the Norwegian Research Foundation in the framework of the Information and Communication Technology (IKT-2010) program—the ADIS project. The authors also would like to thank anonymous reviewers for their constructive comments"

References

[1] Jena - semantic web framework for java. Online: http://jena.sourceforge.net.

[2] T. Andrews et al. Business Process Execution Language for Web Services (BPEL4WS) 1.1. Online: http://www-106.ibm.com/developerworks/webservices/library/ws-bpel, May 2003.

[3] T. Bellwood et al. Universal Description, Discovery and Integration specifi cation (UDDI) 3.0. Online: http://uddi.org/pubs/uddi-v3.00-published-20020719.htm.

[4] D. Box et al. Simple Object Access Protocol (SOAP) 1.1. Online: http://www.w3.org/TR/SOAP/, 2001.

[5] D. Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. Online: http://www.w3.org/TR/rdf-schema/.

[6] R. Chinnici et al. Web Services Description Language (WSDL) 1.2. Online: http://www.w3.org/TR/wsdl/.

[7] J. M. Dunn. *Handbook of Philosophical Logic*, volume III, chapter Relevance logic and entailment, pages 117–224. D. Reidel Publishing Company, 1986.

[8] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[9] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *The 12th International Conference on the World Wide Web (WWW-2003*, Budapest, Hungary, 2003.

[10] I. Horrocks, U. Sattler, S. Tessaris, and S. Tobies. How to decide query containment under constraints using a description logic. In *Proc. of LPAR'2000*, 2000.

[11] J. Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65(3):154–170, 1958.

[12] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Jornal of Logic Programming*, 11:217–242, 1991.

[13] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, 1992.

[14] D. Martin et al. DAML-S (and OWL-S) 0.9 draft release. Online: http://www.daml.org/services/daml-s/0.9/, May 2003.

[15] S. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning(KR2002)*, Toulouse, France, April 2002.

[16] R. Milner. The ployadic pi-calculus: a tutorial. Technical report, Laboratory for Foundations of Computer Science, University of Edinburgh, October 1991.

[17] S. R. Ponnekanti and A. Fox. SWORD: A developer toolkit for web service composition. In *The Eleventh World Wide Web Conference*, Honolulu, HI, USA, 2002.

[18] J. Rao, P. Küngas, and M. Matskin. Application of linear logic to web service composition. In *The First International Conference on Web Services*, Las Vegas, USA, June 2003. CSREA Press.

[19] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web services using semantic descriptions. In *Web Services: Modeling, Architecture and Infrastructure" workshop in conjunction with ICEIS2003*, 2002.

[20] S. Thatte. XLANG: Web services for business process design. Online: http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm.