Composition of Semantic Web Services using Linear Logic Theorem Proving

Jinghai Rao [a] Peep Küngas [a] Mihhail Matskin [b]

[a] *Department of Computer and Information Science, Norwegian University of Science and Technology, NO-7491, Trondheim, Norway, Email: {jinghai,peep}@idi.ntnu.no*

[b] *Department of Microelectronics and Information Technology, Royal Institute of Technology, SE-164 40 Kista, Sweden, Email: misha@imit.kth.se*

Abstract

This paper introduces a method for automatic composition of Semantic Web services using Linear Logic (LL) theorem proving. The method uses a 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. LL, as a resource conscious logic, enables us to define attributes of Web services formally (including parameters, states and non-functional attributes). We use a process calculus to present the process model of 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 complete 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, LL Theorem Prover and Semantic Reasoner can operate together. This architecture has been implemented in Java.

Key words: Service composition, Web service, Semantic Web

1 Introduction

Recent progress in the field of Web services makes 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 to 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 the 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 service initiatives provide platforms and languages that should allow easy integration of heterogeneous systems. In particular, Universal De-

scription, Discovery, and Integration (UDDI) [7], Web Services Description Language (WSDL) [11], Simple Object Access Protocol (SOAP) [8] and parts of DAML-S [25] 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 Services (BPEL4WS) [4] 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. Firstly, Web services can be created and updated on the fly, and it may be beyond human capabilities to analyze a huge amount of available services and to compose them manually. Secondly, the Web services are usually developed by different organizations that use different semantic model presenting services' characteristics (for example, non-functional attributes) 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 the Web service composition task emerging from the above-mentioned sources.

Firstly, we present a method for automated Web service composition which is based on the proof search in a fragment of propositional Linear Logic (LL) [14]. The main idea of the method is as follows. Given a set of existing Web services and the requirements in term of DAML-S ServiceProfile, the method finds a composition of atomic services that satisfies the user requirements. The reason why we use LL theorem proving as Web service composition method is that LL is resource-conscious logic. This means that, first, we can distinguish the information transformation and the state change produced by the Web service and, second, we can perform planning using both qualitative and quantitative non-functional attributes. Because of soundness of the logic fragment used the correctness of composite services is guaranteed with respect to the initial specifications. Completeness of the logic fragment ensures that all composable solutions would be found.

Secondly, we present a composition approach that allows reasoning with types from a service specification. We introduce a set of subtyping rules that defines a valid dataflow for composite services. The subtyping relationships between the DAML concepts are detected by the Semantic Reasoner separately from the LL Theorem Prover. Since the rules are defined as logical implications, the interoperability is ensured between the Linear Logic Theorem Prover and the Semantic Reasoner.

Web service composition using theorem proving is a relatively new topic, and it has been 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 [23].

There is still a long way towards complete automation of service composition in a general case. In this paper we restrict the problem to composition of value-added services only assuming that a core service (atomic or composed) already exists.

The rest of this paper is organized as follows: Section 2 describes a system architecture for composition of Semantic Web services, Section 3 gives a motivating example. Section 4 presents background knowledge for service description language and LL. Section 5 presents a method for transformation of DAML-S profile to extralogical axioms in LL language. Section 6 discusses the usage of type system to enable semantic composition. Section 7 presents the implementation of a prototype system. Finally, related works and conclusions are presented.

2 The service composition architecture

A general architecture of the proposed Web service composition system is depicted on Figure 1. The system takes available atomic services and user's requirements (both of them are described in DAML-S ServiceProfile and grounded by WSDL documents) as inputs and produces a set of possible composite services presented by the ServiceModel in DAML-S service model or BPEL4WS.

The basic components of the system are the follows:

Translator: performs automated transformation between an external presentation and extralogical axioms in LL. In our system, DAML-S ServiceProfile is used externally for presentation of atomic Web service specifications, while LL axioms are used internally for composition. The process model is internally presented by a process calculus. The calculus can be translated either into DAML-S ServiceModel 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 yes, 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 descriptions. The formal logics used in Semantic Reasoner could be logics developed for expressing knowledge and reasoning about concepts and concept hierarchies, for example, Description Logics [15]. A transformation between Description Logics and LL is done by the Adapter.

Adapter: performs translation between LL and the internal presentation used by the Semantic Reasoner.

The composition process in our case is as follows. Firstly, 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. Then, the LL Theorem Prover checks whether the request can be satis-

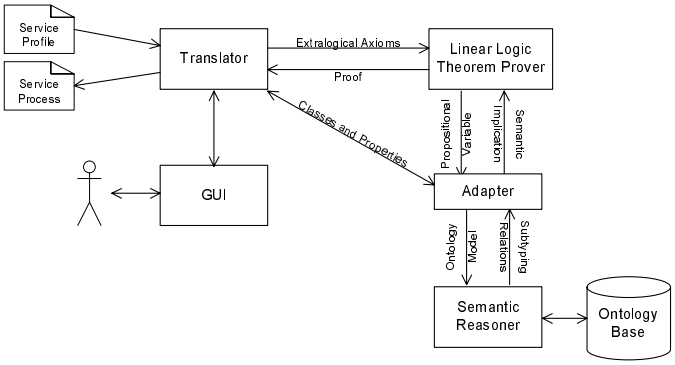

Fig. 1. The architecture of the service composition system.

fied by a composition of existing atomic services (this is done by applying theorem proving in LL). During the theorem proving the Theorem Prover may interact with the Semantic Reasoner. As a result of this interaction the Semantic Reasoner may provide extensions of the set of extralogical axioms via performing subtyping inference using Ontology Base. New axioms are asserted into the Theorem Prover as logical implications. 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 the construction of flow models - the process calculus is translated either to DAML-S ServiceModel or BPEL4WS upon the request. During the composition and execution processes, the user is able to monitor and control the system via GUI.

3 Motivating example

We consider how our service composition method can be applied to the composition of value-added services. Value-added services differ from core services—they are not a part of core services but act as complements to the core services. In particular, they may stand alone in terms of operation and profitability as well as provide adds-on to core services. It is important to mention that value-added services may allow different their combination and they may provide incremental extension of core services. For example, in online shopping, the core services range from product search, ordering, payment and shipment. However, some other services, such as currency exchange, measurement converter and language translation can also be required in a case when the core services cannot meet the users' requests exactly. Usually these value-added services are not designed for a particular core service but they rather extend abilities of core services or, in other words, add value to the core services.

As a working example we consider a ski selling Web service. In this case the core service receives characteristics of a pair of skis (such as, length, brand, model etc) as input and provides prices, availability and other requested characteristics as output. We assume that a user would like to use this service but there are gaps between the user's requirements and the functionalities the service provides.

The differences could exist, for example, in the following details:

- the user would like to receive prices in a local currency (for example, in Norwegian Crowns), however, the service provides price in US Dollars only;
- the user would like to use centimeters as length measurement units but the service uses inches;
- the user does not know what ski length or model are needed and s/he would like that this can be calculated from his/her height and weight;
- the user does not know which brand is most suitable and s/he would like to get a recommendation from a Web service.

Here we illustrate a case where only functionality attributes are considered (basically, input and output of a service). However, our method works with non-functional attributes as well and this will be considered in the later Sections.

We assume that the user provides the body height measured in centimeters (cm), the body weight measured in kilograms (kg), his/her skill level and the price limit. The user would like to get a price of recommended pair of skis in Norwegian Crowns (NOK).

The core service *selectSkis* accepts the ski length measured in inches, ski brand, ski model and gives the ski price in US Dollars (USD).

Some available value-added services are as follows :

- *selectBrand*—given a price limit and a skill level, provides a brand;
- *selectModel*—given body height in cm and body weight in kg, provides ski length in cm and a model;
- *cm2inch*—given length in cm provides length in inches;
- *USD2NOK*—given price in USD provides price in NOK;
- *inch2cm*—given length in inches provides length in cm;
- *NOK2USD*—given price in NOK provides price in USD;
- *kg2lb*—given weight in kg provides weight in pounds;
- *lb2kg*—given weight in pounds provides weight in kg.

The core service and some available value-added services are depicted respectively in Fig. 2 and Fig. 3. A required service is presented in Fig. 4. The structure of a solution for the required service is presented in Fig. 5.

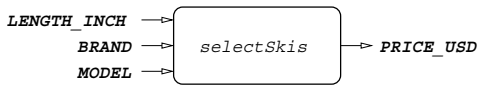

Fig. 2. The core service for buying skis.

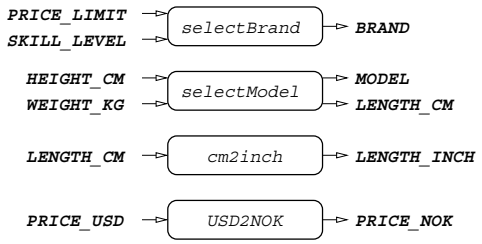

Fig. 3. Some available value-added services.

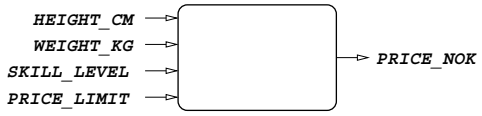

Fig. 4. The required service for buying skis.

We would like to mention that our working example is intentionally made simpler than it is required for practical cases. This has been done in order to keep simplicity of presentation. In practice there can be more value-added services available and more parameters for the core service (in particular, there may exist many other converters for currency, measurements and other units), and the user may not always be able to find a solution intuitively.

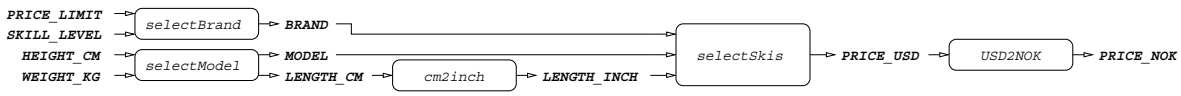

Fig. 5. The final service structure for buying skis.

In addition, it may also be beyond the user's ability searching a huge amount of available value-added services to find all possible solutions. In particular, if the set of possible solutions consists of all existing converters to all inputs and outputs of all Web services (both core and value added), this may cause big overhead in service provision. Taking this into account, we think that automatic composition would be an efficient and practical solution in this case.

4 Background

4.1 Semantic Web services' languages

DAML-S (also called OWL-S in the most recent versions) is a Web service description language built on top of DAML+OIL [1] and OWL [12] which are W3C recommendation for Semantic Web markup languages. Enabled by the Semantic Web technology, DAML-S allows to present semantically enriched datatypes than WSDL which employs XSD as the canonical type system. Therefore, DAML-S can be considered as a step towards automatic matching and composition of services by semantic description, if compared with WSDL.

A description of Web services in DAML-S includes the following three parts.

ServiceProfile: The *ServiceProfile* is used for advertising and discovering the service. The information provided by *ServiceProfile* consists of three parts: information about the providers, information about functionalities and information about non-functional attributes. In [5], the authors give definitions of functionality and non-functional attributes that are used to specify the service profile in DAML-S. The service functionalities are the declarative specification of computational ability of a Web service. They include the service's input parameters, output parameters, preconditions, effects 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).

ServiceGrounding: The *ServiceGrounding* provides information on service execution. It can be regarded as the DAML presentation of WSDL.

ServiceModel: The *ServiceModel* conceives each service as either an atomic or composite process. The process of a composite service specifies control constructs of included atomic or composite services together with control and data flow between the included services. In particular, the *ServiceModel* includes a dataflow model which indicates how messages are transferred from one service to other services.

A significant drawback of DAML-S is that there is no platform to support execution of the process model specified by ServiceModel. An alternative way of specification of the process model is BPEL4WS.

BPEL4WS is essentially a process modeling language proposed by industry. It is designed to enable aggregation of one or more Web services into an execution of a single composite Web service.

Composite services in BPEL4WS are modeled as directed graphs where the nodes are services and the edges represent a dependency link from one service to another. Canonic programming constructs (such as, SWITCH, WHILE and PICK) allow properties of inter-service messages to direct an execution path through the graph. BPEL4WS also contains references to port types of WSDL documents. An expressiveness of service behavior and its inputs/outputs are constrained by XML and XML schema.

Because of the DAML-S service is grounded by WSDL, we can also regard BPEL4WS as the grounding of DAML-S ServiceModel. Although DAML-S ServiceModel and BPEL4WS are very similar, they are not completely intertranslatable [27]. In the section 5, we propose an upper-ontology for the process model. The upper-ontology is designed formally as a process calculus and is compatible with both BPEL4WS and DAML-S ServiceModel. Thus the result of a proof can be translated either into BPEL4WS or DAML-S ServiceModel upon the user's request.

4.2 Linear logic

LL is a refinement of classical logic introduced by J.-Y. Girard to provide a means for keeping track of "resources"—in LL two copies of a formula A are distinguished from a single copy of A. Although LL is not the first attempt to develop resource-oriented logics (well-known examples are relevance logic [13] and Lambek calculus [19]), it is by now one of the most investigated ones. Since its introduction LL has enjoyed increasing attention from researchers in both proof theory and computer science. Therefore, because of its maturity and well-developed semantics, LL is useful as a declarative language and inference system.

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

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

Here, P stands for a propositional variable and A ranges over formulae. The logic fragment consists of multiplicative conjunction $\otimes$, additive disjunction $\oplus$, linear implication $\multimap$ and "of course" (!) operator. 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. Thus the connective $\otimes$ defines deterministic relations between resources. After literals A and B are consumed, literals C and D are generated. In that way we can encode different behaviors of computations. The disjunction $A \oplus B$ defines that either A or B are consumed or generated. It has the same function as the disjunction in classical logic. 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. The unit 1 presents a trivial goal which

Table 1
Inference rules for our LL fragment.

Logical axiom and Cut rule:

$$A \vdash A \ \ (Id) \qquad \frac{\Gamma \vdash A \quad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta} \ (Cut)$$

Rules for propositional constants:

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

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C} \ (L\otimes) \qquad \frac{\Gamma \vdash A \quad \Gamma' \vdash B}{\Gamma, \Gamma' \vdash A \otimes B} \ (R\otimes)$$

$$\frac{\Gamma \vdash A \multimap B}{\Gamma, A \vdash B} \ (Shift) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \ (R\multimap)$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \oplus B \vdash \Delta} \ (L\oplus) \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \ (R\oplus)(a) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B} \ (R\oplus)(b)$$

Rules for exponential !:

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

requires no resources. The inference rules of this fragment are presented in Table 1.

An intuitionistic LL sequent is divided into two parts by $\vdash$ symbol. We call the formula at the left side as "resource", and the formula at the right side is called "goal". Ignoring unrestricted hypothesis for the moment, a LL sequent can be explained as follows: the goal can be achieved by consuming the resource.

Comparing to classical logic, LL provides more expressive power and therefore, allows us to represent more features. In general, LL provides us with the following features for presenting useful properties of Web services that are hardly presented in other formalisms:

(1) Propositional LL enables us to present quantities of consumable resources in Web services, such as price, time and the size of cache.

(2) The "of course" modality enables us to distinguish two aspects of the service functionalities: the information transformation and the state change triggered by the execution of the service. The information transformation is presented by the input/output parameters. We assume that information is reusable, thus the input values are not consumed after the execution of a service. Reusability is determined by using the "of course" modality. Contrarily, the states of the world are changed by service execution. We interpret the change as follows: the old states are consumed and the new states are performed. Therefore, the state variables are presented by propositions without "of course" modality. It means the state values can be consumed only once. In addition, the inference rules

for the "of course" modality enable us to duplicate information in a explicit way. The rules are similar to the weakening and contraction in classical logic.

(3) LL has close connections with concurrent processes that are foundations for modeling composite Web services. In particular, a translation from proofs in LL into Milner's π-calculus [30] has been extensively studied in [3, 29, 6]. For example, the multiplicative conjunction($\otimes$) can present the "composition" in π-calculus; the disjunction($\oplus$) presents "choice", and the "of course" modality(!) presents "replication".

After the available services are specified in the form of LL formulae and a requested service is specified as a theorem to be proven, we apply 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. The 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. To make representation of formulae more compact, we use the following abbreviation: $a^n = \underbrace{a \otimes \ldots \otimes a}_{n}$, for $n > 0$.

5 Transformation from DAML-S profiles to LL axioms

5.1 The upper model of Web services and LL

Fig. 6 shows the upper ontology for the declarative specification of a Web service. The upper ontology can be used as a specification framework to present either the request to the service or the advertisement of an existing service. For the existing service, we do not distinguish the atomic service or composite service, because we model an existing service as a black-box, therefore, the service process that denotes the internal control- and data-flow inside the composite service is not considered when selecting the service. The specification describes a service by functionalities and non-functional attributes. The functionalities include inputs, outputs, preconditions, effects and exceptions. The non-functional attributes are classified into three categories: consumable quantitative attributes, qualitative constraints and qualitative results.

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

$$(\Gamma_c, \Gamma_v); \Delta_c \vdash ((I \otimes P) \multimap (O \otimes F) \oplus E) \otimes \Delta_r$$

where both Γ_c and Γ_v are sets of extralogical axioms representing available value-added Web services and core services respectively. Δ_c is a conjunction of non-functional constraints. Δ_r is a conjunction of non-functional results. We

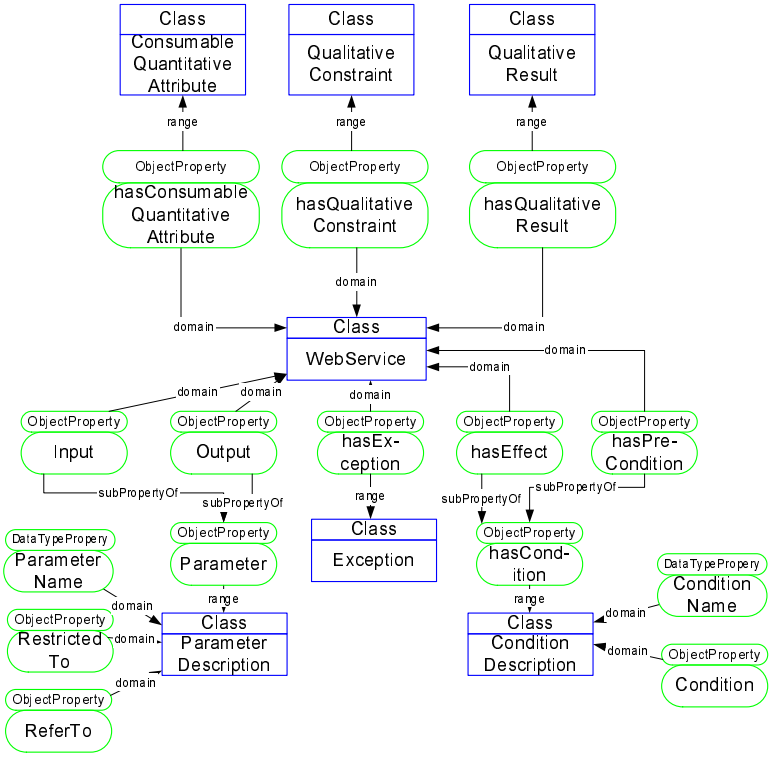

Fig. 6. The upper ontology for Web service declarative specification.

will distinguish these two concepts later. $I \otimes P \multimap (O \otimes F) \oplus E$ is a functionality description of the required service. Both I and O are multiplicative conjunctions of literals with "of course" modality. I represents a set of input parameters for the service and O represents output parameters produced by the service. P and F are multiplicative conjunction of preconditions and effects respectively. E presents 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 as well as changes the world state from P to F. If the execution of the service fails, an exception is thrown. Every element in Γ_c and Γ_v is in form $\Delta_c \vdash ((I \otimes P) \multimap (O \otimes F) \oplus E) \otimes \Delta_r$, where meanings of Δ_c, Δ_r, I, P, O, F and E are the same as described above.

5.2 Transformation of functionalities

The functionalities of a service include two parts: the information transformation and the state change produced by the execution of the service. The information transformation is represented as a transformation from the input parameters required by the service to the output parameters produced by the service. It provides information of data flow during the execution of a Web service. The state change provides information about what the Web service actually does and how execution of the Web service changes the states of the environment. It is modeled as the transformation from the preconditions to the effects. A typical example is a service for logging into a Web site. The input information is the username and password, and the output is a confirmation message. After the execution, the world state changes

from "not_logged_in" to "logged_in". The "logged_in" state will be kept until the "log_out" service is invoked.

We use the "log_in" service as an example to elaborate the process about the transformation of service functionalities. The example DAML-S code is shown in Fig. 7.

The code is consistent with the upper ontology shown in Fig. 6. The inputs and outputs are subproperties of "Parameter" property that ranges over the instance of class "ParameterDescription". This class has three fields. "parameterName" is the name of the actual parameter, which is presented by a string. "restrictedTo" points to a DAML class as the type of the parameter. "refersTo" refers to the parameter's port defined in DAML-S ServiceModel. When translating a specification parameter, either input or output, we take the "restrictedTo" and "referTo" field. For example, the input "UserName" is presented as follows (here we use entity types as a shorthand for URIs).

$$!onto\#Username(mode\#User)$$

The value of "restrictedTo" property is translated into a LL proposition and the value of "referTo" is translated into a proof term which identifies the proposition. The proof term in the parentheses is used to guarantee the proposition (which is introduced during the reasoning process) is not used outside the proposition's scope. In terms of program, it means that the proof term is a program of type that is restricted by the proposition. The "of course" operator ! in front of the proposition indicates an information parameter that can be duplicated.

The states are described by a "ConditionDescription" class that has two parts: "conditionName" and "condition". "condition" refers to a DAML class defining the meaning of a state value. For the state variables, we translate the URI in "condition" to a LL proposition directly.

From the computation point of view, this service requires two input parameters that are of types "`onto#Username`" and "`onto#Password`" respectively and produces an output that has type "`onto#LoginOk`". Before the service is executed, the user can not have been logged into the Web site. After the successful execution of the service, the state changes from "NotLoggedIn" to "LoggedIn". The result LL axiom is shown as follows:

$$\begin{aligned}\vdash\ & !onto\#Username(model\#User) \otimes \\ & !onto\#Password(model\#Passwd) \otimes \\ & onto\#NotLoggedIn \multimap_{service\#Login} \\ & !onto\#LoginOk(model\#LoginOk) \otimes \\ & onto\#LoggedIn\end{aligned}$$

5.3 Non-functional attributes

Non-Functional attributes are useful in evaluating and selecting services when there are several services having the same functionalities. In service presentation, the non-functional attributes are specified as results and constraints. We classify the non-functional attributes into the following three categories:

- **Consumable Quantitative Attributes:** These attributes limit the amount of resources that can be consumed by the composite service. The

```
<profileHierarchy:Service rdf:ID="Login">
 <!-- reference to the service specification -->
 <service:presentedBy rdf:resource="&service;#Login"/>
 <profile:serviceName>Log_into_A_Web_site</profile:serviceName>

 <profile:textDescription>
     This service enables the user to log into a Web site by username and password
 </profile:textDescription>

 <profile:input>
  <profile:ParameterDescription rdf:ID="Username">
   <profile:parameterName>UserName</profile:parameterName>
   <profile:restrictedTo rdf:resource="&onto;#Username"/>
   <profile:refersTo rdf:resource="&model;#User"/>
  </profile:ParameterDescription>
 </profile:input>
 <profile:input>
  <profile:ParameterDescription rdf:ID="Password">
   <profile:parameterName>PassWord</profile:parameterName>
   <profile:restrictedTo rdf:resource="&onto;#Password"/>
   <profile:refersTo rdf:resource="&model;#Passwd"/>
  </profile:ParameterDescription>
 </profile:input>
 <profile:output>
  <profile:ParameterDescription rdf:ID="Confirmation">
   <profile:parameterName>Login_Confirmation</profile:parameterName>
   <profile:restrictedTo rdf:resource="&onto;#LoginOk"/>
   <profile:refersTo rdf:resource="&model;#LoginOk"/>
  </profile:ParameterDescription>
 </profile:output>

 <profile:precondition>
  <profile:ConditionDescription rdf:ID="NotLoggedIn">
   <profile:conditionName>Not_Logged_In_State</profile:conditionName>
   <profile:condition rdf:resource="&onto;#NotLoggedIn"/>
  </profile:ConditionDescription>
 </profile:precondition>
 <profile:effect>
  <profile:ConditionDescription rdf:ID="LoggedIn">
   <profile:conditionName>Logged_In_State</profile:conditionName>
   <profile:condition rdf:resource="&onto;#LoggedIn"/>
  </profile:ConditionDescription>
 </profile:effect>
</profileHierarchy:Service>
```

Fig. 7. An example DAML-S ServiceProfile for "Login" service.

total amount of resources is the sum of all resources of atomic services that form the composite service. For example, the price of composite service is the sum of prices for all included atomic services. This kind of attribute includes total cost, total execution time, etc.

- **Qualitative Constraints:** Attributes which can not be expressed by quantities are called qualitative attributes. Qualitative Constraints are qualitative attributes which specify requirements to execution of a Web service. For example, some services may response only to some authorized calls. The constraints are regarded as prerequisite resource in LL.
- **Qualitative Results:** Another kind of qualitative attributes (such as service type, service provider or geographical location) specify the results regarding the services' context. These attributes can be regarded as goals in LL.

The different categories of non-functional attributes have different presentation in extralogical axioms. As it was mentioned before, the non-functional attributes can be described either as constraints or results and they can be presented as follows:

- **The constraints for the service:**

$$\Delta_1 = Consumable^x \otimes !Constraint$$

- **The results produced by the service:**

$$\Delta_2 = !Fact$$

According to the definition of LL, the variable x ranges over positive integer. However, the type is supposed to be extended by revising the LL interpreter.

5.4 Example with functionality and non-functional attributes

Here, we illustrate the LL presentation of our example taking both functionalities and non-functional attributes into consideration. The services have been introduced in Section 3. For the sake of readability, we use abbreviations to represent the propositions. Here, PL, SL, BR, MO, HC, WK, LC, LI, PU and PN stand for PRICE_LIMIT, SKILL_LEVEL, BRAND, MODEL, HEIGHT_CM, WEIGHT_KG, LENGTH_CM, LENGTH_IN, PRICE_USD and PRICE_NOK respectively.

The available services, both value-added and core services, are specified as follows:

$$\begin{aligned}
\Gamma = \; & NOK^{10} \vdash PL \otimes SL \multimap_{selectBrand} BR \\
& \vdash HC \otimes WK \multimap_{selectModel} LC \otimes MO \\
& NOK^{20} \vdash LC \multimap_{cm2inch} LI \\
& CA_MICORSOFT \vdash PU \multimap_{USD2NOK} PN \\
& \vdash LI \otimes BR \otimes MO \multimap_{selectSki} PU \otimes LOC_NORWAY
\end{aligned}$$

NOK^{10} means that 10 NOK are consumed by executing the service. The service *cm2inch* costs 20 NOK. It is also specified that the currency exchange service *USD2NOK* only responses to the execution requests that have been certificated by Microsoft. The selectSki service is located in Norway. For other attributes which are not specified in service description, the values are not considered. It is possible that more advanced treatment of numeric values may require some other solutions, for example, extra-logical oracles. However, at the moment we try to use LL expressive power as much as possible in order to keep conceptual purity of

the method.

The required composite service is specified by the following formula:

$$(\Gamma); \Delta_1 \vdash (PL \otimes SL \otimes HC \otimes WK \multimap PN) \otimes \Delta_2$$

The constraints for the composite service are as follows:

$$\Delta_1 = NOK^{35} \otimes !CA\_MICROSOFT$$
$$\Delta_2 = !LOC\_NORWAY$$

They mean that we would like to spend at most 35 NOK for the composite service. The composite service consumer has certification from Microsoft (!CA_MICROSOFT) and it requires that all location-aware services are located within Norway ($!LOC\_NORWAY$). ! symbol describes that we allow unbounded number of atomic services in the composite service. We consider quantitative constraints (for example, price) as regular resources in LL. If the total number of resources required by services (which is determined by functionality attributes) is less than or equal to the number of available resources, the services can be included into composite service. Otherwise, if, for example, the *selectBrand* service would require 20 NOK for execution then the total required amount would be 40 NOK and the composition is not valid.

For the qualitative constraints (for example, location), the service uses a literal (for example, LOC_NORWAY) to present its value, and we can determine in the set of requirements Δ_1 whether a service meets the requirement. However, if there is no such literal in the service description, the constraint is not applied to this service at all.

We have discussed how DAML-S ServiceProfile can be translated into extralogical axioms. The next steps are theorem proving in LL and derivation of the process model from proof of the requested composite service. If it has been proven that the requested composite service can be generated using available services and inference rules then the process model can be extracted from the proof. A resulting dataflow of the selected atomic services can also be presented to the user for inspection via GUI component.

5.5 Extraction of a process model from proof

We use process calculus for presentation of process model. The process model is extracted from the proof generated by the LL Theorem Prover and it presents a composed service corresponding to the user request.

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 letter range refer to a set of processes):

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

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

The process calculus is built based on the idea of π-calculus [30]. The π-calculus is a powerful model of concurrency, which can be used to design communication-based programming languages. It is widely used for modeling the composite Web services. One recent example is the XLANG language proposed by Microsoft [36]. 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 easily adapted to π-calculus, although they do not announce π-calculus as their formal foundation.

At conceptual level the process calculus is specified by the upper-ontology shown in Fig. 8. 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 the input, output and internal variables. The "Invoke" activity invokes a process.

The connection between LL proof and π-calculus has been taken up formally by Abramsky [3]. Abramsky's view is essentially a modification of the formulae-as-types (Curry-Howard) isomorphism. Instead of proofs being functions presented by λ-calculus, Abramsky views proofs as processes (e.g. π-calculus or CCS terms). Abramsky's view is to transfer the propositions as types paradigm to concurrency, so that concurrent processes, rather than functional programs, become the computational counterparts of proofs. The key observation of this view is that the cut-elimination in deductive inference is modeled by the construction of a channel for the communication along two ports in process-calculi world.

The Abramsky translation between rules of a fragment of LL and π-calculus translation is further discussed by Bellin and Scott [6]. In their paper, the authors give a detailed treatment of information flow in proof-nets and show how to mirror various evaluation strategies for proof normalization. The authors also give soundness and completeness results for the process-calculus translations of the LL fragment. Bellin and Scott's translation is based on one-sided sequents in Classical Linear Logic(CLL), while the service presentation in this paper is based on Intuitionistic Linear Logic. Considering this, we make some modification for Abramsky's translation to fit our presentation. We summarize the inference rules with the attached proof terms in Table 2.

The Web services composition problem in the context of process calculus is presented by three parts including an output channel process, an input channel process and the local variables. We consider an example where a service P outputs the result through channel a, and then the

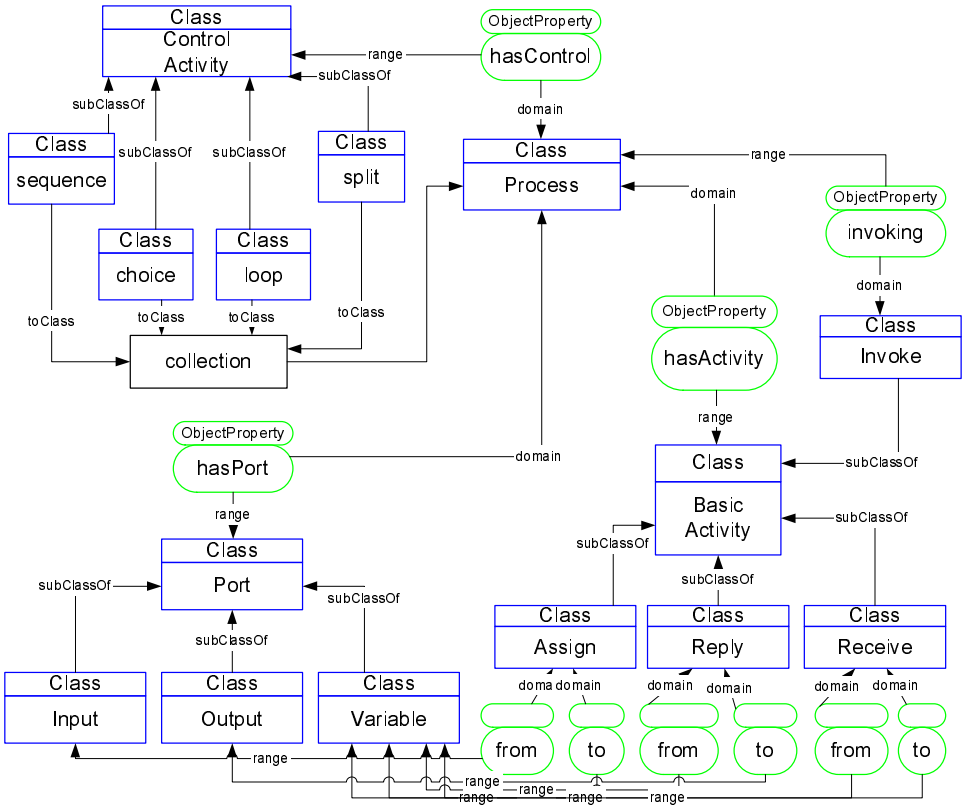

Fig. 8. The upper ontology for the flow model.

result is passed to channel b that is an input channel of service Q. The process is presented as follows:

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

This process is equal to a sequence of "receive", "copy" and "reply" operations in BPEL4WS. Thus the process calculus can be translated into BPEL4WS directly.

When thinking about the abstract process model, the local variables can be assigned to arbitrary name. Thus the above process can be also simplified as:

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

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

5.6 An example proof

In this section we show an example about how the process is generated from the service presentation. For the proof is clear and more readable, we consider only the dataflow (input/output parameters) and we remove the "of course" operators, but the proving process is still after introducing other attributes.

Applied to our motivating example, the predefined axioms and the theorem to be proven can be presented as follows. The meaning of each service has been in-

Table 2
Inference rules for process extraction.

Logical axiom and Cut rule:

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

Rules for propositional constants:

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

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

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

$$\frac{\Gamma, A(a) \vdash P : C(c_1) \quad \Gamma, B(b) \vdash Q : C(c_2)}{\Gamma, (A \oplus B)(a+b) \vdash (P+Q) : C(c_1 + c_2)} \ (L\oplus)$$

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

Rules for exponential !:

$$\frac{\Gamma \vdash \Delta}{\Gamma, !A(0) \vdash \Delta} \ (W!) \qquad \frac{\Gamma, A(a) \vdash \Delta}{\Gamma, !A(a) \vdash \Delta} \ (L!) \qquad \frac{\Gamma, !A(a), !A(a) \vdash \Delta}{\Gamma, !A(a) \vdash \Delta} \ (C!)$$

Structural congruence for a process calculus:

$$P|Q \equiv Q|P \qquad P + Q \equiv Q + P \qquad (P|Q)|R \equiv P|(Q|R) \qquad (P+Q)+R \equiv P+(Q+R)$$

$$P \equiv P|\mathbf{0} \equiv P + \mathbf{0} \equiv P.\mathbf{0} \equiv \mathbf{0}.P$$

$$A(a) \otimes B(b) \equiv A \otimes B(a,b)$$

$$\overline{(a_1, a_2, \ldots, a_n)}(b_1, b_2, \ldots, b_n) \equiv (\overline{a_1}b_1, \overline{a_2}b_2, \ldots, \overline{a_n}b_n)$$

troduced in Section 3 and illustrated in Fig. 2, 3 and 4

Axioms:

$\vdash PL(sbp) \otimes SL(sbs) \multimap_{SelectBrand} BR(sbb)$

$\vdash BR(ssb) \otimes MO(ssm) \otimes LI(ssl) \multimap_{SelectSki} PU(ssp)$

$\vdash HC(smh) \otimes WK(smw) \multimap_{SelectModel} MO(smm) \otimes LI(sml))$

Theorem

$\vdash PL \otimes SL \otimes HC \otimes WK \multimap PU$

Using the inference rules from the Table 2 the proof is shown as Fig. 9.

The extracted process model is presented as follows:

$(sbp, sbs, smh, smw).(SelectModel|SelectBrand)$
$.(\overline{smm}ssm, \overline{sml}ssl, \overline{sbb}ssb).SelectSki.\overline{ssp}$

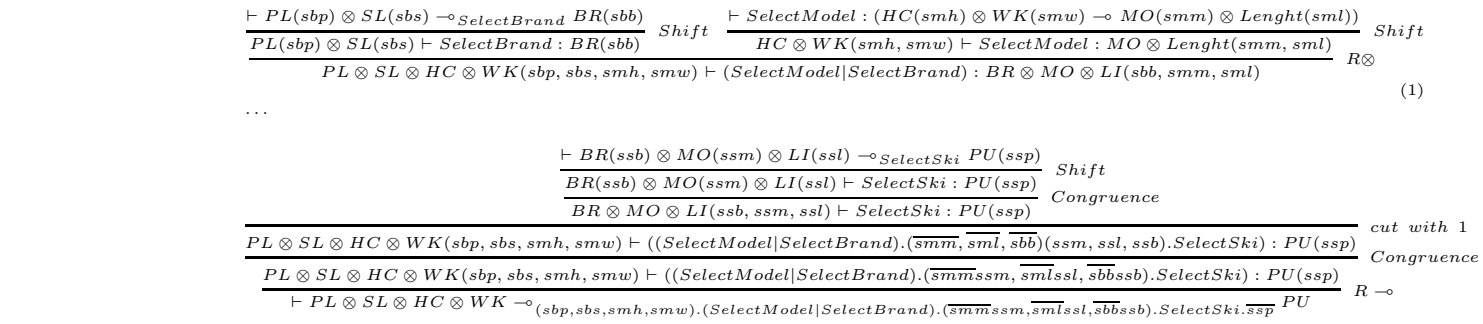

Fig. 9. An example proof.

This process model can be directly translated into most models for describing composite Web services. For example, a sketchy BPEL4WS sequence of the model is as follows:

```
<sequence>
 <receive input="sbp"/>
 <receive input="sbs"/>
 <receive input="smh"/>
 <receive input="smw"/>
 <split>
  <invoke process="SelectModel"/>
  <invoke process="SelectBrand"/>
 </split>
 <copy from="smm" to="ssm"/>
 <copy from="sml" to="ssl"/>
 <copy from="sbb" to="ssb"/>
 <invoke process="SelectSki"/>
 <reply output="ssp"/>
</sequence>
```

6 Composition using semantic description

So far, we considered only exact match of parameters of atomic services in composition process. However, in practice services could be connected together even if the output parameters of one service do not match exactly the input parameters of another service. Generally, if a type assigned to the output parameter of service A is a subtype of the type assigned to an input parameter of service B, it is safe to transfer data from the output to the input. In addition, if we consider non-functional attributes as well, the subtyping is used in the following two cases in the context of resources and goals in LL: 1) given a goal x of type T, it is safe to replace x by another goal y of type S, as long as it holds that T is a subtype of S; 2) given a resource x of type S, it is safe to replace x by another resource y of type T, as long as it holds that T is a subtype of S.

Now we define rules for describing the subtyping behavior of the propositional variables. Basically, the rules are close to a fragment of the subtype behavior presented by Markus Lumpe [22]. We extend the subtyping rules both for resources and goals. It should be also mentioned that the rules are not an extension to LL. Indeed, the subtyping rules are defined as certain inference figures in LL (see Appendix A for exact definitions). In order to emphasize that new inference rules are used for typing purposes (but not for sequencing components) we shall write $\multimap_<$ to denote the subtype relations.

The subtype relation is transitive:

Subtyping transitivity:

$$\frac{\vdash T \multimap_{<} S \quad \vdash S \multimap_{<} U}{\vdash T \multimap_{<} U}$$

In addition, subtyping rules allow type substitution.

Resource subsumption:

$$\frac{U \vdash S \quad \vdash T \multimap_{<} U}{T \vdash S}$$

Goal subsumption:

$$\frac{T \vdash U \quad \vdash U \multimap_{<} S}{T \vdash S}$$

The subtyping rules can be applied either to functionality (parameters) or non-functional attributes.

We use the following two examples to illustrate the basic idea of subtype relations.

First, we show a simple case of how two services with different types of parameters are composed. Let us assume that the output of the *selectSki* service is the price of skis presented in US Dollars, while the input of *USD2NOK* is any amount of money in US Dollars. Because the latter is more general than the former, it is safe to transfer the more specific output to the more general input.

Another example considers the qualitative facts. If an atomic service is located in Norway and since Norway is a European country then it is safe to replace Norway with Europe when matching inputs and outputs of services in a composition process. Intuitively, if the user requires a service that is located within Europe, the service located within Norway meets such requirement.

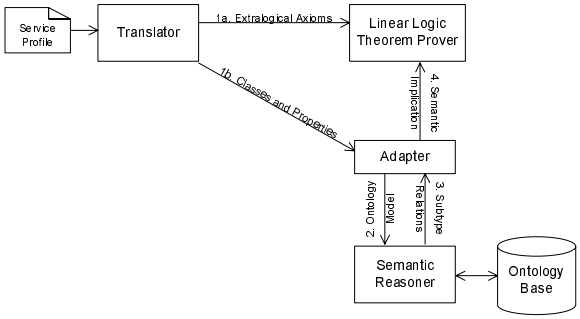

Fig. 10. A Candidate Interaction Mechanism

Here we 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.

Subtype relationships between propositional variables are discovered by the Semantic Reasoner. Since the propositional variables in the extralogical axioms are presented in the form of the URI to DAML classes and properties that are built based on the resource definition of RDFS [9], the reasoner is required to implement the transitive and symmetric properties of *rdfs:subPropertyOf* and *rdfs:subClassOf*. Many available semantic reasoners, such as FaCT [17] or Jena [2] fulfill such requirements.

A general mechanism of interaction between the Semantic Reasoner and the Linear Logic Theorem Prover has been presented in Fig. 1. Several specific interaction mechanisms can be designed according to the general schema. Fig. 10 reveals the a possible specific interaction process:

1. When the translator reads the DAML-S documents for all available atomic services, it creates a set of all classes and

properties that are stored the DAML-S documents. The classes and properties include both those defined inside the DAML-S document and those defined by third-party but imported by the the DAML-S document.

2. The Adapter translates the classes and properties to an ontology model in the format that is recognized by the Semantic Reasoner.
3. The Semantic Reasoner analyses the ontology model and derives all subtype relations of the classes and properties in the ontology model.
4. The Adapter translates the subtype relations to Linear Logic axioms. The axioms are sent to the Linear Logic Theorem Prover.

The Adapter is designed to enable flexible interoperation between different Theorem Provers and Semantic Reasoners. We are working on developing other specific interaction mechanisms for the Adapter. In particular, when the amount of the available classes and properties is huge, the Adapter may send axioms to the LL Theorem Prover in small portions or by request.

7 Prototype

For the service composition system presented in Fig. 1, we have implemented a prototype in Java. In the prototype, Jena (a Java framework for building Semantic Web applications) [2] is used to facilitate both the Translator and Semantic Reasoner. In particular, the TransitiveReasoner in Jena provides support for storing and traversing class and property lattices as graph structures.

The LL Theorem Prover supports reasoning over Web service composition problems both in propositional and first-order LL. Although in this paper we described only how propositional LL could be applied for Web service synthesis, we are prepared to take advantage of first-order representation of services as well. The prover can be downloaded from `http://www.idi.ntnu.no/~peep/RAPS`. According to our experiment with the system scale including thousands of literals, the performance is comparable to other AI planning languages. Some other LL theorem provers based on Prolog, such as lolli [16] and lygon [38]. can be also used with small modifications.

Basic GUI features of the prototype are depicted in Fig. 11. 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). The screenshot in Fig. 11 shows that the composed service is combined from five atomic services. 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 semantic relationships between parameters.

8 Related work

Several methods for dynamic composition of Web services have been proposed in recent years. Most of them fall into one of the following two categories: methods based on pre-defined workflow model and methods based on AI planning.

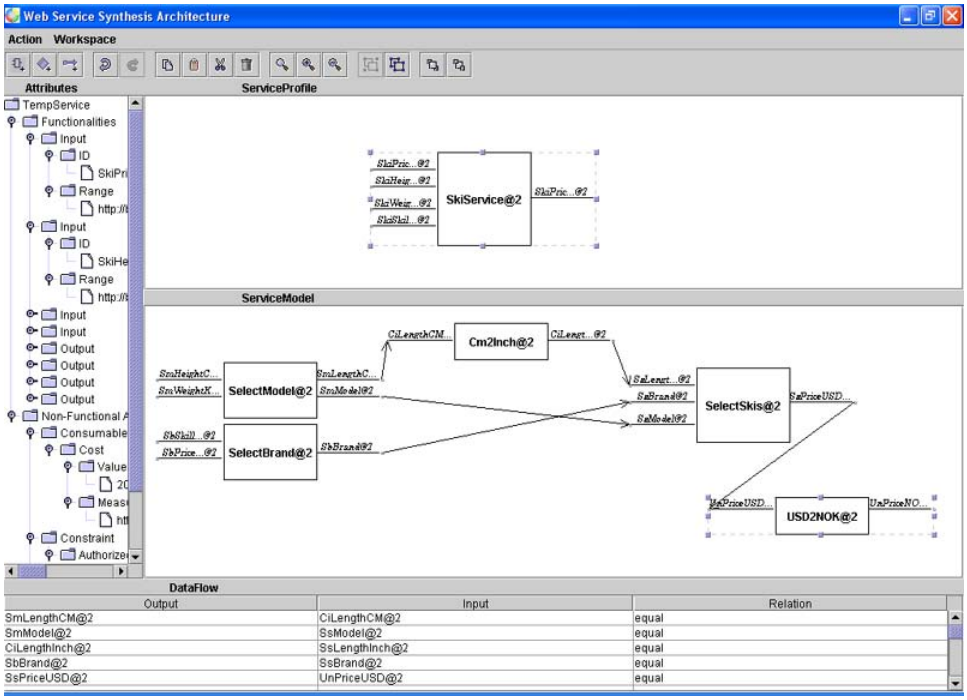

Fig. 11. A screen shot.

For the methods in the first category, the user should specify the workflow of the required composite service, including both nodes and the control flow and the data flow between the nodes. The nodes are regarded as abstract services that contain search recipes. The concrete services are selected and bound at runtime according to the search recipes. This approach is widely adopted by members of the Information Systems community (in particular, see [10] and [32]).

The second category includes methods related to AI planning. They are based on the assumption that each Web service is an action which alters the state of the world as a result of its execution. Since the services (actions) are software components, the input and the output parameters of a service act as preconditions and effects in the planning context. If the user can specify inputs and outputs required by the composite service, a process (plan) is generated automatically by AI planners without the knowledge of predefined workflow.

Much work has been done on planning based on semantic description of Web services. In [28], the authors adapt and extend the Golog language for automatic construction of Web services. Golog is a logic programming language built on top of the situation calculus. The authors address the Web service composition problem through the provision of high-level generic procedures and customizing constraints. Golog is adopted as a natural formalism for representing and reasoning about this problem.

Waldinger [37] elaborates an idea for service synthesis by theorem proving. The approach is based on automated deduction and program synthesis and has its roots to his earlier work [24]. Initially available services and user requirements

are described in a first-order language, related to classical logic, and then constructive proofs are generated with SNARK theorem prover. Finally, service composition descriptions are extracted from particular proofs.

Lämmermann [20] takes advantage of disjunctions in classical logic to describe exceptions, which could be thrown during service invocation. We take advantage of disjunctions for more general purpose—to represent mutually exclusive outputs of services, which could be both exceptions as well as other results provided by the services.

Usually in AI planning closed world assumption is made, meaning that if a literal does not exist in the current world, its truth value is considered to be *false*. In logic programming this approach is called *negation as failure*. The main trouble with the closed world assumption, from Web service construction perspectives, is that merely with truth literals we cannot express that new information has been acquired. For instance, one might want to describe that after sending a message to another agent, an identity number to the message will be generated. Thus during later communication the ID could be used.

McDermott [26] considers this problem in AI planning in composing Web services. He introduces a new type of knowledge, called *value of an action*, which persists and which is not treated as a truth literal. However, while using resource-conscious logics, like LL or transition logic, this problem is treated implicitly and there is no need to distinguish informative and truth values. Since LL is not based on truth values, we can view generated literals as references to informative objects. Thus, if a new literal is inserted into the world model, new piece of information will be available. Therefore, LL provides an elegant framework for modeling incomplete knowledge—although before plan execution only partial knowledge is available, during execution more details would be revealed.

A strong interest to Web service synthesis from AI planning community could be explained roughly by similarity between DAML-S and PDDL [26] representations. PDDL is widely recognized as a standardized input for state-of-the-art planners. Moreover, since DAML-S has been strongly influenced by PDDL language, mapping from one representation to another is straightforward (as long as only declarative information is considered). When planning for service composition is needed, DAML-S descriptions could be translated to PDDL format. Then different planners could be exploited for further service synthesis.

In [39] the SHOP2 planner is applied for automatic composition of Web services, which are provided with DAML-S descriptions. Another planner for automatic Web service construction is presented in [33].

Thakkar et al. [35] consider dynamic composition of Web services using a mediator architecture. The mediator takes care of user queries, generates wrappers around information services and constructs a service integration plan.

SWORD [31] is a developer toolkit for building composite Web services. SWORD does not deploy the emerging service-description standards such as WSDL and DAML-S, instead, it uses Entity-Relation (ER) model to specify

the inputs and the outputs of Web services. As a result, the reasoning is based on the entity and attribute information provided by an ER model.

Sirin et al [34] present a semi-automatic method for Web service composition. Each time when a user has to select a Web service, all possible services, that match with the selected service, are presented to the user. The choice of the possible services is based both on functionalities and non-functional attributes. The functionalities (parameters) are presented by OWL classes and OWL reasoner is applied to match the services. Afterward, the system filters the services based on the non-functional attributes that are specified by the user as constraints.

The main difference between our method and the above-mentioned methods is that we consider non-functional attributes during the planning. Usage of LL as a reasoning language allows us formally define the non-functional characteristics of Web services (in particular, quantitative attributes). We also distinguish the constraints and facts in qualitative attributes. They are treated differently while reasoning.

9 Conclusion

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 composition. First, DAML-S service descriptions are automatically translated into LL axioms. Next 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). Further, 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 value-added Web services. We also propose an architecture for automatic Semantic Web service composition. The non-functional settings of the systems are discussed and techniques for DAML-S presentation, LL presentation, and semantic reasoning are presented. A prototype implementation of the approach is done.

One of the advantages of LL over classical logic is the richer set of connectives. In particular, the additive conjunction and disjunction of LL distinguish the internal choice and the external choice. The internal choice is enabled by additive conjunction &. An applicable example is a dialog box that asks users to choose "yes" or "no". From the computational viewpoint, the dialog box service produce two outputs and user chooses one of them would lead the process to follow different branches. A typical external choice situation is that a service may produce one of several alternative outputs every time it is executed. In particular, this is the case with exception handling. In that case, the output produced by the service depends on the execution environment only. None of the existing Web service specification

languages supports internal choice, and, therefore this issue is not usable in current stage. Therefore, we regard this an important branch of future development of Web service languages.

We use a propositional fragment of LL as the specification of Web Services. Although first-order LL should allow better presentation of Web service properties, it suffers from poor efficiency (also completeness is not ensured). Before moving to the first-order LL we would like to gain experience in practical usage of propositional LL and better understand its niche in the whole Web service composition process.

Our other strand of future work is directed towards improving the efficiency of both the LL Theorem Prover and the Semantic Reasoner. Usually, the amount of the available services and the size of ontology models are huge. Therefore it is necessary to reduce the search space during problem solving.

There are also some extensions of the current composition method that have been under consideration. First, our experience with the Web service composition shows that users are not always able to completely specify the requirement of the composite service. We consider to apply the principle of partial deduction [21] to provide more flexibility to the user. Some initiate results about using partial deduction and symbolic negotiation for Web service composition based on a multi-agent architecture have been reported in [18]. Second, the composition of core services usually needs some business models to specify the relationship between multiple core services. The method presented in the paper is to be extended for core service composition if the business model is given.

References

[1] The DARPA Agent Markup Language homepage. Online: http://www.daml.org.

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

[3] S. Abramsky. Proofs as processes. *Theoretical Computer Science*, 135(1):5–9, 1994.

[4] Tony Andrews et al. Business Process Execution Language for Web Services (BPEL4WS) 1.1, May 2003.

[5] Anupriya Ankolekar and DAML-S Coalition. DAML-S: Semantic markup for web serivces. In *Proceedings of the International Semantic Web Workshop*, 2001.

[6] G. Bellin and P. J. Scott. On the pi-calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, 1994.

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

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

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

[10] Fabio Casati, Ski Ilnicki, and Li-Jie Jin. Adaptive and dynamic service composition in EFlow. In *Pro-*

ceedings of 12th Int. Conference on Advanced Information Systems Engineering(CAiSE), Stockholm, Sweden, June 2000. Springer Verlag.

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

[12] M. Dean et al. OWL Web Ontology Language 1.0 reference. Online: http://www.w3.org/TR/owl-ref/, July 2002.

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

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

[15] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan 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.

[16] J.S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic Linear Logic. *Information and Computation*, 110(2):327–365, 1994.

[17] 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.

[18] Peep Küngas, Jinghai Rao, and Mihhail Matskin. Symbolic agent negotiation for semantic web service exploitation. In *Proceedings of the Fifth International Conference on Web-Age Information Management, WAIM'2004*, Dalian, China, July 2004. Springer-Verlag.

[19] Joachim Lambek. The mathematics of semtence struture. *American Mathematical Monthly*, 65(3):154–170, 1958.

[20] Sven Lämmermann. *Runtime Service Composition via Logic-Based Program Synthesis*. PhD thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, 2002.

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

[22] Markus Lumpe. *A Pi-Calculus Based Approach for Software Composition*. PhD thesis, Institute of Computer and Applied Mathematics, University of Bern, 1999.

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

[24] Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

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

[26] Drew McDermott. Estimated-regression planning for interaction with web services. In *Proceedings of the 6th International Conference on AI Planning and Scheduling, Toulouse, France, April 23–27, 2002*. AAAI Press, 2002.

[27] Sheila McIlraith and Dan Mandell. Comparison of DAML-S and BPEL4WS. Knowledge Systems Lab, Stanford University, September 2002.

[28] Sheila McIlraith and Tran Cao 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.

[29] Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the 1992 Workshop on Extensions to Logic Programming*, number 660 in LNCS, pages 242–265. Springer-Verlag, 1993.

[30] Robin Milner. The ployadic picalculus: a tutorial. Technical report, Laboratory for Foundations of Computer Science, University of Edinburgh, October 1991.

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

[32] H. Schuster, D. Georgakopoulos, A. Cichocki, and D. Baker. Modeling and composing service-based and reference processbased multi-enterprise processes. In *Proceeding of 12th Int. Conference on Advanced Information Systems Engineering (CAiSE)*, Stockholm, Sweden, June 2000. Springer Verlag.

[33] M. Sheshagiri, M. desJardins, and T. Finin. A planner for composing services described in DAML-S. In *Proceedings of the AAMAS Workshop on Web Services and Agent-based Engineering*, 2003.

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

[35] Snehal Thakkar, Craig A. Knoblock, Jose Luis Ambite, and Cyrus Shahabi. Dynamically composing web services from on-line sources. In *Proceeding of 2002 AAAI Workshop on Intelligent Service Integration, Edmonton, Alberta, Canada*, 2002.

[36] Satish Thatte. XLANG: Web services for business process design.

[37] Richard Waldinger. Web agents cooperating deductively. In *Proceedings of FAABS 2000, Greenbelt, MD, USA, April 5–7, 2000*, volume 1871 of *Lecture Notes in Computer Science*, pages 250–262. Springer-Verlag, 2001.

[38] M. Winikoff and J. Harland. Implementing the Linear Logic programming language Lygon. In *Proceedings of the International Logic Programming Symposium*, pages 66–80, December 1995.

[39] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S web services composition using SHOP2. In *Proceedings of the 2nd International Semantic Web Conference, ISWC 2003, Sanibel Island, Florida, USA, October 20–23, 2003*, 2003.

A Proofs for subtyping rules

Subtyping transitivity:

$$\dfrac{\dfrac{}{\Gamma \vdash S \multimap U} \quad \dfrac{\dfrac{}{S \vdash S}\ Id \quad \dfrac{}{U \vdash U}\ Id}{S, S \multimap U \vdash U}\ L\multimap}{\Gamma, S \vdash U}\ Cut \tag{A.1}$$

...

$$\dfrac{\dfrac{\dfrac{}{\Sigma \vdash T \multimap S} \quad \dfrac{\dfrac{}{T \vdash T}\ Id \quad \dfrac{}{S \vdash S}\ Id}{T, T \multimap S \vdash S}\ L\multimap}{\Sigma, T \vdash S}\ Cut}{\dfrac{\Sigma, \Gamma, T \vdash U}{\Sigma, \Gamma \vdash T \multimap U}\ R\multimap}\ Cut\ with\ A.1$$

Goal subsumption:

$$\dfrac{\dfrac{}{\Sigma \vdash T} \quad \dfrac{\dfrac{}{\Gamma \vdash T \multimap S} \quad \dfrac{\dfrac{}{T \vdash T}\ Id \quad \dfrac{}{S \vdash S}\ Id}{T, T \multimap S \vdash S}\ L\multimap}{\Gamma, T \vdash S}\ Cut}{\Sigma, \Gamma \vdash S}\ Cut$$

Resource subsumption:

$$\dfrac{\dfrac{}{\Gamma \vdash T \multimap S} \quad \dfrac{\dfrac{}{T \vdash T}\ Id \quad \dfrac{}{S \vdash S}\ Id}{T, T \multimap S \vdash S}\ L\multimap}{\Gamma, T \vdash S}\ Cut \qquad \dfrac{}{\Sigma, S \vdash G}}{\Sigma, \Gamma, T \vdash G}\ Cut$$

B Additional inference figures

Inference figure *Shift* is defined in the following way:

$$\dfrac{\dfrac{}{C \vdash A \multimap B}\ Axiom \quad \dfrac{\dfrac{}{A \vdash A}\ Id \quad \dfrac{}{B \vdash B}\ Id}{A, A \multimap B \vdash B}\ L\multimap}{C \otimes A \vdash B}\ Cut$$