An integrated framework for semantic service composition using answer set programming

Yilong Yang¹, Jing Yang², Xiaoshan Li¹, Weiru Wang¹
¹Department of Computer and Information Science
Faculty of Science and Technology
University of Macau
yylonly@gmail.com, xsl@umac.mo, belindawwr@gmail.com
²College of Computer Science and Technology
Guizhou University
yangjing6646@yahoo.com.hk

ABSTRACT

Notwithstanding the advancement of service computing in recent years, service composition is still main issue in this field. In this paper, we present an integrated framework for semantic service composition using answer set programming. Unlike the AI planning approaches of top-down workflow with nested composition and combining composition procedure into service discovery, this proposed framework integrates designed workflow with nested composition. In addition, the planning is based on interface variables with validation through pre and post conditions. Moreover, a unified implementation of service discovery, selection, composition and validation is achieved by answer set programming. Finally, the framework performance is demonstrated by a travel booking example on QWSDataset.

Keywords: service; service composition; answer set programming

Introduction

Service computing has been extensively studied, and major issues include formalizing the specification of service, service discovery, selection and composition (Rao & Su, 2005; Dustdar & Schreiner, 2005). There are two primary paradigms for service composition: top-down and bottom-up paradigms (Bartalos & Bielikova, 2011). For top-down paradigm, the complex workflow is designed manually. Hence, the workflow could be refined in terms of requirements. Nevertheless, bottom-up paradigm can composite services automatically by AI method. Furthermore, mixture paradigm architecture is proposed in (Paik, Chen, & Huhns, 2014), in which, unlike top-down paradigm, HTN to plan workflow is utilized instead of designed workflow, and like top-down paradigm to discover services, the best service is selected. If no existed service is matched, same as bottom-up paradigm, the nested composition procedure will be triggered to fulfil the requirement. Nevertheless, full automatic approaches usually are not the best solution for world-wild problems. Especially for human dominant
activities, AI planned workflow is not correct all the time. Even though nested composition compensates this deficiency when planning service is coarse-grained, it would be failure when dependent services are decomposed incorrectly. Hence, medium solution and efficiency AI approach should be exploited. Regularly, service discovery and composition are regarded as two separate processes (Syu, Ma, Kuo, & FanJiang, 2012). In (Kušter, Końig-Ries, Stern, & Klein, 2007), they proposed an approach to combining composition procedure into discovery process in compliance with multiple effects, and the effects coverage must be computed foremost. Therefore, the approaches to computing coverage are required implicitly.

In order to address these related issues of service-computing, we propose an integrated framework for semantic service composition using answer set programming (FSSC), which includes the following features:

1. **Modified mixture composition paradigm.** Our approach combines top-down and bottom-up paradigms by designing a workflow for discovering and selecting service foremost, when no service is matched or discovered, a bottom-up nested composition procedure will be triggered.

2. **Unified procedure for service discovery and composition (using simplified service description).** In order to boost planning and validation, we apply simplified service description to divide goal service (i.e, to discover and composite service) into source and target services. Hence, service discovery and composition can be cooked in the identical procedure. Only one difference is the length of composition chain. An outstanding feature of this approach is not required to compute the multiple effects.

3. **On-the-fly planning and validation by answer set program.** Using interface variables for unified planning, whereas, Hoare logic (pre/post condition) for service validation. Our paper aims at employing answer set programming (ASP) (Brewka, Eiter, & Truszczynski, 2011) for planning, declarative programming oriented towards difficult search problems. ASP allows for a unified representation of the problem including rules and constraints except the solution algorithms, in terms of features such as solid logic foundations, high-expressiveness, nondeterminism and high-declaratively.

The remainder of this paper is organised as follows: Section 2 overviews FSSC. Section 3 presents the formal specification of service as well as ASP implementation. Section 4 defines the service compositionability formally, and then Section 5 discusses unified service discovery, selection and composition and validation base on the formal specification. Section 6 shows the performance of FSSC with a travel booking example. Section 7 discusses related work, and finally, Section 8 concludes this paper and outlines future work.

**Overview of FSSC**

In this section, the informal definition of service is introduced first, and then the low level composition methodology is discussed. Accordingly, the high level mixture design paradigm and details of generation engine are presented.

To illustrate our framework, after checking all the examples in file *OMG BPMN 2.0 example1*, we choose the travel booking example shown in figure 1, which includes 4 sub-processes, 6 gateways, 18 activities and 26 events with error handling and compensation mechanisms. Thus, the complexity of this example is adequate for evaluating the performance of proposed framework.
This example provides travel booking services to clients. A client requires to interact with the system by following this process for his or her travel booking. The process includes four portions: acquiring alternatives reservation, making decision, providing credit card information and booking, and charging the credit card. The detail information can be found in above OMG document. In the remain of this paper, we will show how to use modified mixture paradigm to composite this process and on-the-fly validate.

**Modified Mixture Paradigm**

In FSSC, we introduce the mixture design paradigm which combines complex top-down paradigm and automatic bottom-up paradigm. For top-down paradigm, the complex workflow is designed manually. Then all the un-matched services are matched (i.e. service discovery) in the workflow with services in the database. Once more than one services are founded, the best one will be selected. This paradigm has higher efficiency and can produce more complex process, however it would be failed if no one service is matched. For bottom-up paradigm, once providing the specification of interface including pre/post condition, the workflow would be automatically generated by AI planning. Whereas this approach is time consuming and has less accuracy. The better choice is to customise two paradigms first, and then combine both together, so that we can not only make use of the advantage of automatic AI planning, but also can obtain more complex and accuracy service than bottom-up methodology.

For customised bottom-up paradigm, pre and post conditions are used for on-the-fly validation of service composition, rather than for planing, only main semantic properties of service like domain, class, input and output variables. On-the-fly validation is hot-plug, and whether it is utilised or not is based on the requirements of time and accuracy. For top-down paradigm, the hot-plug validation layer is added. When the procedure is triggered for the services discovery, the correctness can be checked automatically.

Combining two customised paradigms can be devided into two levels:

**Top Level** – Like top-down paradigm: designing workflow, discovering candidate services (Service A,B,H,D) and selecting best one (Service A,B) in Fig 3.

**Bottom Level** - If no service was discovered, customised bottom-up paradigm are used to composite services to fulfil the requirements (see Service G,H in Fig 3).
FSSC framework with Travel Booking Example

FSSC is a cloud-based framework shown in Fig. 2. When you are a travel agency or even a traveler, *In client side*, you would like to find or develop a travel booking system, however, purchasing or developing a new system requires high expenditure. Once FSSC is used, you can design your requirements as BPMN, because BPMN can describe complex process service with user friendly interface and standardised specification. As a cloud client user in Fig 2, you can submit your requirement to FSSC cloud. *In the cloud side*, the service crawler finds and downloads open services and partner services, moreover FSSC transforms all service indexes to unified ASP service facts, which can be saved into service database. Here we use ASP to implements our generation engine, and all the services are denotes as ASP facts. The service can be any kind of WebWCF/OSGI/COBRA Services. Accordingly, the requirements of BPMN activities will be extracted and transformed to unmatched services (ASP goal facts).

Now we discuss *generation engine* shown in Fig. 3. Based on the goal service (goal fact), service facts in database, all the pre-defined ASP rules and constraints, the generation engine will take the actions for each goal service: *discover service* matched the functional requirements (for example, search flights: domain as travel, class as fights, input days and location, output available choices); *select best service* with less cost of the non-functional requirements (such as price under $1000, time under 2 seconds), if more than one services were discovered; *composite services*, if no service was discovered, for example, booking service contains booking fight and hotel services; *check correctness*. After then, the result of composition (Answer Sets) with all matched services was produced by generation engine. Finally, FSSC will transform it into an executable workflow (BPEL), and deploy it into workflow engine by returning a web service endpoint for agency or traveller for invoking. After using this service, agency or traveller can give the feedback to FSSC for improving the accuracy of service generation engine.
ASP and SMT solver for FSSC

ASP (Answer Set Programming) can be used for realizing service discovery, service selection and service composition. In this paper, we use Gringo as ASP grounder, Clasp as ASP solver (Gebser et al., 2010), and write the asp code based on (Lifschitz, 2002). But there are some disadvantages for Gringo and Clasp without supporting set/list and its operations. Thus, more powerful ASP solver DLV-complex\textsuperscript{2} can be taken into account which supports set/list and its operations. In addition, it supports self-defined predicate which can invoke SMT solver for validation in ASP solver.

For SMT solver, we choose Z3\textsuperscript{3}, because Z3 is a high-performance theorem prover developed by Microsoft Research. Z3 supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted functions, and quantifiers. Z3 is integrated with a number of program analysis, testing, and verification tools from Microsoft Research. But Z3 solver can only answer satisfiable problem. A formula condition is valid if and only if its negation is un-satisfiable. If the running result of Z3 solver is unsat, then the formula condition is valid. Hence, Z3 can be used to check the composibility for service composition.

Service Description

In this section, we define FSSC formally first, and then discuss how to generate the corresponding declarative ASP from the formal specification. Based on the formal defined service in logic tuples as well as composition tasks, we can transform it into ASP, the requirements and service description as ASP facts, and composition tasks as ASP rules. However, there is a constraint assumption on pre and post conditions in this work, which is that all constraints should be isolated. For example, precondition : $x > 0$ and postcondition : $y < 3$ are isolated, but postcondition : $x + y > 2$ is not isolated.
Generally semantic services are defined on ontology and tuples. Approaches to define semantic service are ontology and tuples. Considering the efficiency and portability, we only focus on the essential of OWL-S, WSMO and QoS by defining **Atomic Service** as follows:

\[\text{AS} : \langle \text{SDec, Spec, Sign, FuncR, NonFuncR} \rangle\]

where

1. **SDec** : interface of service, is denoted as: \(\text{ServiceName}(\text{in} : U, \text{out} : V)\), where the server with name of ServiceName with input variables \(\text{in}\) and output variables \(\text{out}\), as well as their corresponding types of \(U\) and \(V\).
2. **Spec** : pre and post conditions of service, is signified as Hoare Logic: \(P \vdash R\).
3. **Sign** : \(\langle \text{PType, URL} \rangle\), where
   - (a) **PType** indicates category of service vendor, could be provided by local, partner or open cloud as \(\text{PType} \in \{\text{LocalService, PartnerService, OpenService}\}\)
   - (b) **URL** denotes the service location address for invocation
4. **FuncR** : \(\langle \text{Desc, Class, Domain, In, Out} \rangle\), where
   - (a) **Desc** : the human readable functional description of service.
   - (b) **Class** : business class of service (e.g. hotel).
   - (c) **Domain** : business domain of service (e.g. travel).
   - (d) **In** : \(\{\text{inX} : U\}\), **Out** : \(\{\text{outY} : V\}\) : input and output variables of service
5. **NonFuncR** : \(\langle \text{ResponseTime, Availability, Throughput, Price, Rank, Privacy} \rangle\), where
   - (a) **ResponseTime** ∈ \(\text{Float}^+\) : is the duration time from sending request to receiving a response with unit ms.
   - (b) **Availability** ∈ \([0,1]\) : the number of successful invocations out of the total.
   - (c) **Throughput** ∈ \(\text{Integer}^+\) : the maximum number of service invocations for a given period of time such as 1 minute.
   - (d) **Price** ∈ \(\text{Float}^+\) : Price for using this service.
   - (e) **Rank** ∈ \(\text{Integer}^+\) : customer satisfaction.
   - (f) **Privacy** ∈ \(\{\text{OpenService, PartnerService, PrivateService}\}\) : Privacy level. Three levels are ordered with \(\text{OpenService} \sqsubseteq \text{PartnerService} \sqsubseteq \text{PrivateService}\), which can be denoted as 0, 1, and 2, respectively. If Privacy equals 0, as PartnerService, only Partner Service and Private Service can be taken into account for matching procedure.

Since evaluating service should estimate all inside operations, the non-functional requirement is group property. Hence, all operations inside one component have same non-functional properties.

### Listing 1: Atomic Service - Search Flights

```plaintext
atomService(  
sdec(searchFlight),  
  spec("days ∈ datatype ∧ location ∈ cities ⊨ flights ≠ ∅"),  
sign(openservice, "http://hostname/path"),  
funcR({searchFlight}, {flight}, {travel},  
in({var(days, data), var(location, string)}),  
out({var(flights, list)}),  
nonFuncR("20","0.98","120","0.99","0.00","12","0")).
```

Let’s use the service **Search Flights** in ASP shown in Listing 1. as an example. Its privacy attribute is open service, functional requirements are class as flight, domain as travel, input variables are days and
location, output variables are flights list, non-functional requirements are response time as 20 ms, availability as 98%, throughput as 1200, no price, rank 12. Therefore, ASP is really tight and expressive technique. Here noted, Spec is string type in ASP, when invoking the SMT solver, it should be transformed to the corresponding language.

**Service Composition**

Service composition is the pivotal of generation engine. This section focuses on service composability. In order to unify service discovery, selection, nested composition into the same procedure, target and source services are introduced.

**Equivalence of Variables**

Let \( \text{var}_1 \) and \( \text{var}_2 \) are two variables defined by \( \langle \text{Name}_1, \text{Type}_1, \text{Class}_1, \text{Domain}_1 \rangle \) and \( \langle \text{Name}_2, \text{Type}_2, \text{Class}_2, \text{Domain}_2 \rangle \). They are equivalence, denoted as \( \text{var}_1 =_{\text{eq}} \text{var}_2 \), if and only if the following conditions hold:

1. \( \text{Type}_1 = \text{Type}_2 \)
2. \( \text{Class}_1 = \text{Class}_2 \)
3. \( \text{Domain}_1 = \text{Domain}_2 \)

For instance, there are two variables: variable location, \text{type:string}, \text{class:hotel}, \text{domain:travel}, and variable place, \text{type:string}, \text{class:hotel}, \text{domain:travel}, they are equivalence because of the same type, class and domain.

**Composability**

Let \( A_1 \) and \( A_2 \) are services, they can be composited if the following conditions hold:

1. \( \exists \text{out} \in A_1. \text{FuncR}.\text{Out}, \text{in} \in A_2. \text{FuncR}.\text{In} \cdot \text{out} =_{\text{eq}} \text{in} \)
2. \( A_1.\text{Spec}.R \uparrow \text{out} \Rightarrow A_2.\text{Spec}.P \uparrow \text{in} \)
3. \( A_1.\text{FuncR}.\text{Domain} = A_2.\text{FuncR}.\text{Domain} \)

where \( \text{Formula} \uparrow \text{VariableSet} \) denotes the derived formula by removing all sub-formulas in \( \text{Formula} \) which variables of \( \text{VariableSet} \) do not appear. E.g., \( (x > 1 \land y < 3) \uparrow \{x\} = (x > 1) \).

The conditions of \( \text{Composability} \) exhibit that the composable services must have the same identical domain, and output variables of service \( A_1 \) must have one variable which should be inside input variables of service \( A_2 \). In addition, the postcondition of \( A_1 \) should imply the precondition of \( A_2 \).

**Atomic Composition**

Let \( A_1 \) and \( A_2 \) are atomic services. If they are composable, composable variable set \( \text{Internal} \) can be defined as:

\[
\text{Internal} = \{ \text{in} \mid \text{out} \in A_1.\text{FuncR}.\text{Out} \land \text{in} \in A_2.\text{FuncR}.\text{In} \land \text{in} =_{\text{eq}} \text{out} \}
\]

Atomic composition of \( A_1 \) and \( A_2 \) can be denoted as \( AC : A_1 \gg A_2 \), whose elements can be calculated as follows:
\[ AC.SDec = A_1.SDec \land A_2.SDec \]
\[ AC.Spec.p = A_1.Spec.p \uparrow \{ A_1.FuncR.In \} \land A_2.Spec.p \uparrow \{ A_2.FuncR.In - \text{Internet} \} \]
\[ AC.Spec.R = A_1.Spec.R \uparrow \{ A_1(FuncR.Out - \text{Internet} \} \land A_2.Spec.R \uparrow \{ A_2.FuncR.Out \} \]
\[ AC.FuncR.Desc = A_1.FuncR.Desc \cup A_2.FuncR.Desc \]
\[ AC.FuncR.Class = A_1.FuncR.Class \cup A_2.FuncR.Class \]
\[ AC.FuncR.Domain = A_1.FuncR.Domain \]
\[ AC.FuncR.In = \{ A_1.FuncR.In \} \cup \{ A_2.FuncR.In - \text{Internet} \} \]
\[ AC.FuncR.Out = \{ A_1.FuncR.Out - \text{Internet} \} \cup \{ A_2.FuncR.Out \} \]
\[ AC.Sign.Type = \bot \{ A_1.Sign.Type, A_2.Sign.Type \} \]
\[ AC.NoFuncR.ResonseTime = \text{Sum} \]
\[ AC.NoFuncR.Price = \text{Sum} \]
\[ AC.NoFuncR.Others = \text{LowValue} \]
\[ AC.NoFuncR.Privacy = \bot \{ A_1.NoFuncR.Privacy, A_2.NoFuncR.Privacy \} \]

The result of service composition can be obtained by the conjunction of services without the composable variables, where function requirements is disjunction, the non-functional requirements are the sum of properties or lowest value except the attribute of Privacy.

Composition Chaining

Let \( S_1, S_2, \ldots, S_n \) are services. If they are composable, the outcome of composition \( S_1 >> S_2 >> \ldots >> S_n \) is: \( \langle \text{Services}, \text{CompositeResources}, \text{SourceService}, \text{TargetService}, \text{CompositeRelations} \rangle \)

1. \text{Services} is the collection of all the services participated in composition.
2. \text{CompositeResources} is the set of the composable variables.
3. \text{SourceService} is the head service in chaining.
4. \text{TargetService} is the tail service in chaining.
5. \text{CompositeRelations} is the composite function: \( \text{Services} \times \text{CompositeResources} \rightarrow \text{Services} \)

Let us port the formal description to ASP in the following list. Because of the advantage of declarative ASP and powerful ASP solver, we only need to define the problem and constraints. After then, ASP solver can recursively composite service as an chaining, compute the cost, and finally select best chaining after it terminates.

In Listing 2, lines 12-24 represent two services, lines 26-32 are for checking composibility. If they are composable, the composited service is expressed in lines 1-10. This rule will be invoked recursively by ASP solver. The ASP solver will terminate successfully if the corresponding constraints are satisfied.

**Listing 2: ASP Code for Composition Rule**

```plaintext
1  % Composited service
2  atomService(sdec(cp(ServiceNameA, ServiceNameB)),
3  sign(#newType(TypeA, TypeB), #newUrl(new)),
4  funcR(#union(DescA,DescB),#union(ClassA,ClassB),
5  DomainA,in(CIN),out(COUT)),
6  nonFuncR(#sum(RTA,RTB),
7  #min(AvailabilityA, AvailabilityB),
8  #min(ThroughputA, ThroughputB),
```
9  \#min(ReliabilityA, ReliabilityB),
10 \#sum(PriceA, PriceB), \#min(RankA, RankB))
11
12 % First service
13 :: atomService(rdec(RDEC1),
14 sdec(ServiceNameA), sign(TypeA, URLA),
15 funcR(DescA, ClassA, DomainA,in(IN1), out(OUT1)),
16 nonFuncR(RTA, AvailabilityA, ThroughputA,
17 ReliabilityA, PriceA, RankA)),
18
19 % Second service
20 atomService(rdec(RDEC2),
21 sdec(ServiceNameB), sign(TypeB, URLB),
22 funcR(DescB, ClassB, DomainB,in(IN2), out(OUT2)),
23 nonFuncR(RTB, AvailabilityB, ThroughputB,
24 ReliabilityB, PriceB, RankB)),
25
26 % Check Compositibility:
27 \#intersection(OUT1, IN2, INTERNEL),
28 \#difference(IN2, INTERNEL), CIN),
29 \#union(#difference(OUT1, INTERNEL),
30 \#typeOrder(goaltype, TypeA),
31 \#typeOrder(goaltype, TypeB),
32 \#card(INTERNEL) != 0,
33 DomainA == DomainB, ServiceNameA != ServiceNameB.

Unify Service Discovery, Selection, Composition and Validation

In this section, we present generation engine, and the algorithms of mixture diagram for service discovery, selection, composition and validation. And then these algorithms are integrated into a unified procedure.

Algorithm 1 Generation Engine

**Require:** Input workflow is valid

\* Precondition

**Ensure:** Executable Workflow

\* Postcondition

1: procedure GENERATIONENGINE(designwf)
2: services ← EXTRACT SERVICES(unworkflows)
3: for all s ∈ services do
4: SERVICECOMPOSITION(s)
5: end for
6: workflow ← GENERATIONWORKFLOW(services, designwf)
7: return workflow
8: end procedure

Algorithm 2 FSSC Composition

**Require:** Unmatched service

\* Precondition

**Ensure:** Matched service

\* Postcondition

1: function SERVICECOMPOSITION(unmservice)
2: \( rs \leftarrow \text{SERVICE DISCOVERY}(\text{unmservice}) \)
3: \( \text{if} \ \text{lens}(rs) > 1 \ \text{then} \)
4: \( \text{bests} \leftarrow \text{SERVICE SELECTION}(rs) \)
5: \( \text{return} \ \text{bests} \)
6: \( \text{else if} \ \text{lens}(rs) == 0 \ \text{then} \)
7: \( \text{rnc} \leftarrow \text{NESTED COMPOSITION}(\text{service}) \)
8: \( \text{if} \ \text{lens}(\text{rnc}) > 1 \ \text{then} \)
9: \( \text{bestnc} \leftarrow \text{SERVICE SELECTION}(\text{rnc}) \)
10: \( \text{return} \ \text{bestnc} \)
11: \( \text{end if} \)
12: \( \text{else} \)
13: \( \text{return} \ \text{rs} \)
14: \( \text{end if} \)
15: \( \text{end function} \)

The pseudo codes in Algorithm 1 presents the process of FSSC. Code in line 2 extracts all atomic services from the submitted workflow. Afterward, for every service, Algorithm 2 of service composition will be invoked. In line 2 of algorithm 2, service discovery function is called for matching all the services to their functional requirements. If the result more than one, from line 3 to 5, service selection can be invoked to choose best service according to non-function requirements, but if no service is matched, then in line 7, low level nested composition function will be triggered to composite services. In line 8 - 10, if the composition result contains more than one chains, the best one would be chosen. We find that both service discovery and composition need to select best one from candidate services or service chainings. Since selected one service can be regarded a special chaining with the length 1. Thus, service discovery and composition can be unified into one procedure.

**Unify service discovery and composition**

In order to unify, discovery or composition **goal service** is divided into two services. The first one is **source service** which only has output variables no input variables, output is the same as input variables of goal service, and the other one is **target service** which only has input variables without output variables. The input variables of target service is the same as output variables of goal service, where

\[
\begin{align*}
\text{SourceService.FuncR.Out} &= \text{GoalService.FuncR.In} \\
\text{SourceService.FuncR.In} &= \emptyset \\
\text{SourceService.Spec.P} &= \text{true} \\
\text{SourceService.Spec.R} &= \text{GoalService.Spec.P} \\
\text{TargetService.FuncR.Out} &= \emptyset \\
\text{TargetService.FuncR.In} &= \text{GoalService.FuncR.Out} \\
\text{TargetService.Spec.P} &= \text{GoalService.Spec.R} \\
\text{TargetService.Spec.R} &= \text{true}
\end{align*}
\]

Therefore, **SourceService**, **TargetService** and **Services** in service database are composited according to the composition rules. The positions are fixed: **Source Service** is the first one (head), **TargetService** is the last service (tail). The discovery or composition result is a chaining revealed as follows:

\[
\text{Chaining : SourceService} \gg S_1 \gg S_2 \gg \ldots \gg S_n \gg \text{TargetService},
\]

where \( \text{Chaining.FuncR.In} = \emptyset \wedge \text{Chaining.FuncR.Out} \cap \text{TargetService.FuncR.In} = \emptyset \)

And ASP snippet codes are shown in Listing 3.
Listing 3: A Snippet Code for Stop Condition

33 % Contrainst
34 :- atomService(sdec(ServiceName), sign(Type, URL),
35 funcR(Desc, Class, Domain, in(IN), out(OUT)),
36 nonFuncR(RT, Availability, Throughput, Price,
37 Rank, Privacy)), #card(IN) == 0, #card(OUT) == 0.

Hence, the composition is satisfied if the composition service has no input variables and output variables without containing any input variable of TargetService. Composited chaining may provide more output variables. The required composition service GoalService is

\[
S_1 \gg S_2 \gg \ldots \gg S_n = \begin{cases} 
\text{service discovery} & \text{if } n = 1 \\
\text{service composition} & \text{if } n > 1 \\
\text{failures} & \text{if } n < 1 
\end{cases}
\]

Now, service discovery and composition are unified into one procedure.

Service Selection
Following unified service discovery and composition, service selection will make global optimal decision. \(N_{\text{target}}\) and \(N_{\text{chain}}\) are non-functional requirements of GoalService and composition result, respectively. Service selection can be denoted as:

\[
Service = \min_{\text{chain}} \text{Cost}(W, N_{\text{target}}, N_{\text{chain}})
\]

ASP implementation is shown in Listing 4.

Listing 4: A Snippet Code for Composition Rules

38 % Compute Cost
39 composition(ServiceName, sign(Type, URL),
40 funcR(Desc, Class, Domain, in(IN), out(OUT)),
41 nonFuncR(RT, Availability, Throughput, Reliability,
42 Price, Rank), #costZZZ(RT, Availability,
43 Throughput, Reliability, Price, Rank,
44 Type, rt, av, tr, re, pr, rank, goaltype))
45
46 % Service Selection
47 resultcost(X) :- #min\{Cost, ServiceName :
48 composition(ServiceName, sign(Type, URL),
49 funcR(Desc, Class, Domain, in(IN), out(OUT)),
50 nonFuncR(RT, Availability, Throughput, Price,
51 Rank, Privacy), Cost\} = X.

Hence, the best chaining or service is chosen since it has the minimum cost to the target service. In this paper, the least square cost is applied, and other standards could be chosen as well.

Evaluation of FSSC

Preprocess for QWSDataset
The standard web service database with QOS attributes is QWSDataset[^4], which can be used for travel booking example.
Table 1: QWSDataset Download

<table>
<thead>
<tr>
<th>Codes</th>
<th>Descriptions</th>
<th>Num</th>
<th>Percents</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>All Entries</td>
<td>2507</td>
<td>100%</td>
</tr>
<tr>
<td>400</td>
<td>Bad Request</td>
<td>253</td>
<td>10.09%</td>
</tr>
<tr>
<td>403</td>
<td>Forbidden</td>
<td>145</td>
<td>5.78%</td>
</tr>
<tr>
<td>404</td>
<td>File Not Found</td>
<td>196</td>
<td>7.82%</td>
</tr>
<tr>
<td>502</td>
<td>Bad Gateway</td>
<td>45</td>
<td>1.79%</td>
</tr>
<tr>
<td>N/A</td>
<td>Timeout</td>
<td>128</td>
<td>5.11%</td>
</tr>
<tr>
<td>N/A</td>
<td>Unknown Host</td>
<td>143</td>
<td>13.65%</td>
</tr>
<tr>
<td>N/A</td>
<td>downloadable</td>
<td>1642</td>
<td>65.50%</td>
</tr>
</tbody>
</table>

Table 2: QWSDataset Transform

<table>
<thead>
<tr>
<th>Descriptions</th>
<th>Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downloadable</td>
<td>1642</td>
</tr>
<tr>
<td>Empty File</td>
<td>246</td>
</tr>
<tr>
<td>Invalid File Format</td>
<td>967</td>
</tr>
<tr>
<td>Error Parsing</td>
<td>197</td>
</tr>
<tr>
<td>Invocation Target Exception</td>
<td>67</td>
</tr>
<tr>
<td>Databinding Exception</td>
<td>147</td>
</tr>
<tr>
<td>Pass Preprocess</td>
<td>31</td>
</tr>
<tr>
<td>Transformed</td>
<td>207</td>
</tr>
</tbody>
</table>

Nevertheless, QWSdataset was collected from the Internet in 2008, but it has not been updated from then on. Hence, all services must be pre-processed. The whole pre-process procedures are displayed in Table 1 and 2. QWSdataset includes 2507 entries (services), the Bash script downloads all services from web, if the URL could be opened and connected, which are called downloadable with 1642 entries. Afterward, the Bash script with Perl can check and transform downloadable original pages to standard form with QOS attributes, which only have 31 entries. Finally, Using Go language and Bash script transform standard web services to ASP. The number of services is 207.

Evaluation ASP Performance on FSSC

In this section, the evaluation result of compositing travel booking example on ASP-based FSSC and QWSdataset is displayed in Table 3.

Table 3: Travel Booking on QWSdataset

<table>
<thead>
<tr>
<th>Service Discovery/Composition</th>
<th>Service Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>clength</td>
<td>nchain</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>Search Fights</td>
<td>1</td>
</tr>
<tr>
<td>Search Hotels</td>
<td>1</td>
</tr>
<tr>
<td>Package</td>
<td>2</td>
</tr>
<tr>
<td>Notification</td>
<td>1</td>
</tr>
<tr>
<td>Book Fights</td>
<td>1</td>
</tr>
<tr>
<td>Book Hotel</td>
<td>1</td>
</tr>
<tr>
<td>Charge Card</td>
<td>1</td>
</tr>
<tr>
<td>Booking</td>
<td>4</td>
</tr>
</tbody>
</table>

We pick up 8 services in travel booking BPMN: SearchFights, SearchHotels, Package, Notifications, Book Fights, BookHotel, ChargeCard and Booking. For example, Service Package is the package service for fights and hotels to users, therefore, its length is 2. The length of chaining is denoted by clength. Because service discovery and composition is unified as one procedure, if clength = 1, it means that the service is discovered; if clength is more than 1, it means that serveral services should be
composited. Similarly, $nchain$ is the number of candidate composition chainings; $rtime$ is the planning time; $vtime$ is the validation time, and $mcost$ and $lcost$ are the min and max cost of candidate chainings. From the result, we can find that nested composition is much time consuming than service discovery, and selection time depends on the number of chaining.

**Related Work**

**Semantic Web Service**

The major issue of state-of-art semantic web service is too complex to effectively composite service. SAWSDL (semantic annotations for WSDL and XML schema) is the first attempt to define semantic web service, which is extended from WSDL by adding some simple elements that denote the semantics of IO(input and output variables of service). The comparative research (Bouchiha, Malki, Dja, Alghamdi, & Alnafjan, 2013) shows the effectiveness comparing this approach to other works. OWL-S (OWL for services) and WSMO (web service modeling ontology) are alternative techniques for semantic web service. OWL-S can delineate business process, functional requirement of service and QOS (quality of service) properties, nonetheless, it does not define how to map business process to standard specification such as WSDL. WSMO is much flexible than OWL-S by allowing using expression of normal logic, however, it has the same issue to OWL-S that can not map process to BPEL. Although the following works (Bordbar, Howells, Evans, & Staikopoulos, 2007; Le, Nguyen, & Goh, 2009) dissolve those issues, they only model the semantics of service and process, but not touch service validation. All the semantic web services techniques purposed for specific service composition, our tuples based and simplified formal description are much suitable.

**Service Composition Framework**

The comprehensive service composition are mentioned in composition survey. There are various frameworks of service composition. In (Rao & Su, 2005), a general framework is proposed based on workflow technique and AI planning. We refine and extend the abstract framework. For the top-down framework, the requester should build an abstract process model before the composition planning. Then discovery and selection atomic web services can be processed automatically by program. EFLOW (Casati, Ilnicki, Jin, Krish- namoorthy, & Shan, 2000) uses a static workflow generation workflow method belonged to top-down paradigm. All the services and execution dependency are discribed on graph. Service discovery and selection can be done in a highly dynamic environment. Polymorphic process model (Schuster, Georgakopoulos, Cichocki, & Baker, 2000) combines the static and dynamic service composition based on state machine. The process can consist of abstract subprocess (nested service composition). However, state machine and graph description are not easy to use for normal user. That is the mainly reason that we use BPMN. For AI planning methods such situation calculus, PDDL, SWORD, SHOP2, they take pre and post conditions for planning, but time consuming. Hence, we take mixture paradigm to deal with service composition.

**Answer Set Program for Service Composition**

There are rare related works about ASP-based service composition. (Rainer, 2005; Dorn, Hrastnik, & Rainer, 2007) are the first two attempts using answer set programming to service composition. Furthermore, they won the first and second service composition competition. However, they only use the syntax information such as input and output variables for composition. Just like other bottom-up composition, the composition target is one service. Our approach treats this issue with the nested service composition. (Qian, Huang, & Zhao, 2011) presents the algorithm for translating semantic web service described by OWL-S to action language C, and then to answer set programming. Similar like the other AI planing approaches, they plan the service composition based on pre and post conditions.
However, there are two defects with them: one is the reasoning speed, and the other is most current practical services without condition description. In our approach, we take interface and domain information for reasoning with one option of pre and post conditions.

**Conclusions and Future Work**

In this paper, we present an integrated framework for semantic service composition. To tackle the relevant issues in a same framework, all the portions of FSSC are formally specified, in which each atomic service is defined as a pair of pre and post conditions. We first customise the bottom-up paradigm, where the pre and post conditions are used for the validation of service composition rather than planning. Moreover, the bottom-up is first modified, and then nested into top-down paradigm. Service discovery, composition, selection and validation are unified into one procedure subsequently. Almost all page code of answer set programming can be obtained by achieve generation engine. Finally, an example of travel booking is used to evaluate the performance of FSSC on QWSDataset.

In the future, we will improve the framework on its limitations, such as the constraints of pre and post conditions, and not enough experiments.

**Acknowledgement**

The work presented in this paper was funded by project SAFEHR of Macao Science and Technology Development Fund ref. 018/2011/AI.

**References**


Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., & Thiele, S. (2010). A user’s
guide to gringo, clasp, clingo, and iclingo. *University of Potsdam, Tech. Rep.*


**ENDNOTE**