Towards Constraint-Based Composition With Incomplete Service Descriptions

Matthew Moran, Tomas Vitvar, Maciej Zaremba
Digital Enterprise Research Institute
National University of Ireland, Galway
{firstname.lastname}@deri.org

Abstract

We apply our work on Web service discovery with initially incomplete information to the problem of service composition. Rich semantic descriptions of Goals and Web services allow the unambiguous specification of constraints on what services are required and offered respectively. However, the data provided in service descriptions may be incomplete. We describe how missing instance data may be fetched dynamically and used for richer service discovery. We show how the resultant knowledge base can be used for simple compositions of services having mutual constraints.

1 Introduction and Definitions

In this paper we describe how we extend our work on service discovery [11], with incomplete information in service descriptions, in the direction of constraint-based service composition. For example, we take a client wishing to make an order for a laptop and a docking station. Each type of laptop may only work with specific docking stations. The constraints on combinations of laptops and docking stations may either be declared explicitly or may only be available from a service provider at runtime. In our approach, we declare a generic relation between laptops and docking stations in a domain ontology and allow individual service ontologies to provide their own logical definition for the that top-level relation e.g. a particular ontology used for IBM products defines an axiom for the relation, specifying that IBM laptops only work with IBM docking station.

Our system architecture is goal-oriented. At run-time, we take a client goal and attempt to find a matching service. Ontologies used to describe the goal and the service, including axioms defining constraints, are merged into a knowledge base. Conditions specified in the capability for the goal provide the basis for a query on this knowledge base. If no match is found, we apply simple goal decomposition and then iteratively repeat this procedure for each new goal. Two varieties of constraint are taken into account. The first

are the constraints that are included in the query itself e.g. restrictions on price or delivery locations. The second are those defined between multiple products from (potentially) multiple services e.g. IBM laptops are only compatible with IBM docking stations. We use WSMO [8] as our conceptual model and adopt the definitions of information, functional and behaviour semantics provided in WSMO-lite [10].

Information semantics is defined as a structure:

$$O = (C, R, E, I) \tag{1}$$

with a set of classes, C, a set of relations R, a set of instances of C and R, E and a set of axioms, I.

Functional Semantics is defined as:

$$F = (\Sigma, \phi^{pre}, \phi^{eff}), \tag{2}$$

where $\Sigma \subseteq (\{x\} \cup C \cup R \cup E)$ is the signature of symbols, i.e. variable names $\{x\}$ or identifiers of elements from C, R, E of some information semantics O; ϕ^{pre} is a precondition which must hold in a state before the service can be invoked and ϕ^{eff} is the effect, a condition which must hold in a state after the successful invocation.

Behavioral Semantics is a description of the public and private behavior of a service. We only use the public behavior (called choreography in WSMO). We define the choreography X of the service using a state machine as

$$X = (\Sigma, L), \tag{3}$$

where $\Sigma \subseteq (\{x\} \cup C \cup R \cup E)$ is the signature of symbols, i.e. variable names $\{x\}$ or identifiers of elements from C, R, E of some information semantics O; and L is a set of rules. Further, we distinguish dynamic symbols denoted as Σ_I (input), and Σ_O (output) and static symbols denoted as Σ_S. Each rule $r \in L$ defines a state transition $r : r^{cond} \rightarrow r^{eff}$ where $cond$ is defined as an expression in logic which must hold in a state before the transition is executed; eff is defined as an expression in logic describing how the state changes when the transition is executed.

1

In addition, we denote the description of the Web service and the goal as W and G respectively. For each $D \in W \cup G$, we denote the information semantics as D_O, the capability as D_F, and choreography as D_X.

2 Composition with Data Fetching

Our approach to service composition has service discovery, including fetching information at run-time, at its core. There are two extensions (i) iterative goal decomposition (if necessary) and (ii) maintaining the knowledge base used of services matching any sub-goals until all sub-goals have been matched. Goal decomposition is iterative because, each time a goal can not be matched, an attempt is made to break it up into sub-goals. This continues until a match is found or no further decomposition is possible. Step (ii) means that constraints defined for services that are found to match one sub-goal are maintained when the discovery process is applied to any subsequent sub-goals.

The algorithm *wraps* around discovery. Its purpose is to maintain a stack of matching services and a knowledge base, called B_r, that is built up with the ontologies used by each of the respective matching services.

Input:

- The client's goal, G.

- A set of candidate Web services, W_c.

Output:

- A stack of services, W_m, matching the client's goal.

Uses:

- A stack, G_u, that holds any goals that have to be matched – initialized with the client's goal.

- A variable, m, that holds the result of each discovery call. It can have the values, $match$ or $nomatch$.

- A knowledge base, B_r, that holds the ontologies and corresponding instance data for each Web service that matches the goal, G, or any subgoal.

- The information semantics of a specific service, $w_{i(o)}$.

- A discovery function, $discover()$ (see algorithm 2).

- A decomposition function, $decompose()$.

- A set of decomposed goals, G_d.

The algorithm *pops* unmatched goals from G_u. For each goal, a discovery function is called iteratively for each candidate Web service. If a match is found, the service is added to W_m and its supporting ontologies and instances

Algorithm 1 High Level Algorithm for Composition

```
 1: Wm.init()
 2: Br ← ∅
 3: m ← nomatch
 4: while not Gu.isempty() do
 5:     gi ← Gu.pop()
 6:     while m ≠ match and wi ← Wc.getnext() do
 7:         m ← discover(gi, wi, Br)
 8:         if m = match then
 9:             Br ← Br ∪ wi(o)
10:             Wm.push(wi)
11:         else
12:             Gd ← decompose(gi)
13:             if Gd = ∅ then return error
14:             for all gj ∈ Gd do
15:                 Gu.push(gj)
16:             end for
17:         end if
18:     end while
19: end while
```

are added to the knowledge base, B_r. If the result of discovery is $nomatch$, then a decomposition function is called to break the goal into subgoals, which are added to G_u. The algorithm continues until the G_u stack is empty or an unmatched goal can not be decomposed. The knowledge base B_r is important because it maintains the ontologies, instance values and constraints for each of the services already discovered as a complete, or partial match, to the client's goal. This, along with a goal and Web service, provide the input for each call of the discovery function. We assume a goal decomposition function is available, that for example, breaks up a goal based on conjuncted clauses in its postcondition.

3 Discovery

In [11], we defined two phases for discovery: *Web Service Discovery* and *Service Discovery*. Web Service Discovery operates on the capability descriptions of the goal $\mathcal{G}_F$ and web service $\mathcal{W}_F$. For this paper, we only use the second phase where instance data, constraining the service that may only be available at runtime, is taken into account. For completeness, we include the data fetch algorithm of [11]. It is modified to take an additional input of a knowledge base containing knowledge previously retrieved for the composition. The matching function for service discovery is defined as:

$$s \leftarrow matching(\mathcal{G}, \mathcal{W}, \mathcal{B}_{gw}), \qquad (4)$$

where $\mathcal{G}$ and $\mathcal{W}$ is a goal and a service description respectively and $\mathcal{B}_{gw}$ is a common knowledge base for the goal

and the service. The knowledge base contains data which must *directly* (through Web service and goal ontologies, $\mathcal{G}_O$ and $\mathcal{W}_O$ respectively) or *indirectly* (through data fetching) be available so that the matching function can be evaluated. The result s of this function can be: (1) $match$ when the match was found (in this case all required data in $\mathcal{B}_{gw}$ is available), (2) $nomatch$ when the match was not found (in this case all required data in $\mathcal{B}_{gw}$ is available), or (3) $nodata$ when some required data in $\mathcal{B}_{gw}$ is not available and thus the matching function cannot be evaluated.

We further assume that all required data for the goal is directly available in the description $\mathcal{G}_O$. The data fetching step is then performed for the service when the matching function cannot be evaluated (the result of this function is $nodata$). We then define the knowledge base as:

$$\mathcal{B}_{gw} = \mathcal{G}_O \cup \mathcal{W}_O \cup \{y_1, y_2, ..., y_m\}, \qquad (5)$$

where $\{y_i\}$ is all additional data that needs to be fetched from the service in order to evaluate the matching function.

Further, we denote $\mathcal{W}_X$ as the data-fetch interface of the service $\mathcal{W}$ with output symbols Σ_O and input symbols Σ_I. The matching function can be then evaluated if data $\{y_i\}$ can be fetched from the service through the data fetch interface if input data Σ_I is either initially available in the knowledge base $\mathcal{B}_{gw}$ (data directly available from the goal or web service ontologies) or the input data becomes available during the processing of the interface.

In algorithm 2, the matching function is integrated with the data fetching which provides instance data for the concepts referred from the goal effect ϕ^{eff}. The algorithm operates on inputs, produces outputs and uses internal structures as follows:

Input:

- Web service W for which we denote W_O as the web service ontology with initial instance data and $\mathcal{W}_X$ as data-fetch interface of the Web service with rule base L. In addition, for each rule $r \in L$ we specify the data of the rule effect r^{eff} as $r.data$ and the action $r.action$ with values add, $update$, $delete$ meaning that if the rule is executed the action performs the effect of the rule, i.e. changing the state by adding, updating or deleting data in the memory (knowledge base).

- Goal description G for which we denote G_O as the goal ontology with initial instance data and G^{eff} as the goal capability effect. For W and G it must hold that they match at abstract level (Web service discovery).

- A knowledge base, B_r, which contains information on services already discovered as part of a composition.

Output:

- Boolean variable s indicating the result of the matching function between W and G, i.e. $match$ or $nomatch$.

Uses:

- Processing memory M containing data fetched during execution of rules of the data fetching interface.

- Knowledge base B_{gw} which contains data for processing of the matching function.

- Boolean variable $modified$ indicating whether the knowledge base has been modified or not during the processing.

Algorithm 2 Minimized Data Fetching for Discovery
1: $B_{gw} \leftarrow G_O \cup W_O \cup B_r$
2: $M \leftarrow B_{gw}$
3: **repeat**
4:     $modified \leftarrow false$
5:     $s \leftarrow matching(G, W, B_{gw})$
6:     **if** $s = nodata$ **then**
7:        **while** get r from L: $holds(r^{cond}, M)$ **and** $r.data \in G^{eff}$ **and not** $modified$ **do**
8:           **if** $r.action = add$ **then**
9:             $add(r.data, M)$
10:             $add(r.data, B_{gw})$
11:             $modified \leftarrow true$
12:           **end if**
13:           **if** $r.action = remove$ **then**
14:             $remove(r.data, M)$
15:           **end if**
16:           **if** $r.action = update$ **then**
17:             $update(r.data, M)$
18:             $update(r.data, B_{gw})$
19:             $modified \leftarrow true$
20:           **end if**
21:        **end while**
22:     **end if**
23: **until** $s \neq nodata$ **or not** $modified$

The algorithm tries to fetch data from the service by processing the service's data-fetch interface. For each rule present, which can be executed, it checks whether its result will provide any information referenced by G^{eff}. For example G^{eff} may refer to the concept *price* of a given product which is unavailable in the B_{gw}, however a rule exists which can result in an instance of the *price* concept being obtained. Once the data fetching operations are executed and new facts are added, updated or removed, a $modified$ flag is set to true and B_{gw} can be matched again. This cycle ends when no data can be fetched from the interface or the

matching function can be evaluated (the result is $match$ or $nomatch$).

The algorithm assumes that the rules of the data-fetch interface can be executed independently. In particular this means that if there is a symbol referencing a concept in the knowledge base and there is a rule which can fetch an instance for that concept, there will be no other rule which needs to be executed first.

The algorithm uses independent memory (memory M) from the knowledge base (B_{gw}) for processing the data-fetch interface. This allows that already-obtained data cannot be removed from the knowledge base while, at the same time, correct processing of the interface is ensured. The memory M is used not only for data but also for control of interface processing (in general, the content of the memory does not need to always reflect the content of the knowledge base).

4 Implementation and Evaluation

We extended our implementation for service discovery, including runtime data fetching, to tackle the problems of simple composition presented by the Challenge. The composition is described as simple because it assumes that the order in which the services are invoked is unimportant. The criteria for the tests are that the constraints on individual services and those defined in the tasks (represented as goals) are met and that the constraints between particular services are also satisfied.

We define a common ontology to define shared concepts used in the descriptions of goals and services, such as $Location$, $Notebook$, $DockingStation$, etc. In addition, we use this ontology to specify named relations. Specific ontologies for goals and services declare axioms that define the relations to represent their conditions. As before for discovery, we define a set of relations in the common ontology which represent the axioms that a service may need to define. Axioms provide the definitions for these relations with specific conditions. For example, listing 1 shows the simple declaration for the $isCompatible$ relation in the common ontology and how it can be defined in the service ontology.

```
1   /* isCompatible relation in the domain ontology */
2
3   relation do#isCompatible (ofType do#Notebook, ofType do#
        DockingStation)
4
5   /* implementation of the isCompatible relation in the service ontology */
6
7   axiom isCompatibleDef definedBy
8       ?notebook[do#GTIN hasValue ?gtinX]
9       memberOf do#Notebook and
10      ?dockingstation[do#supportsGTIN hasValue ?gtinY]
11      memberOf do#DockingStation
12      and ?gtinX = ?gtinY) implies
13      do#isCompatible(?notebook, ?dockingstation).
```

Listing 1. $isCompatible$ **relation**

The variables $?gtinX$ and $?gtinY$ hold Global Trade Identication Numbers (GTIN)[1] for products, a standard specification used by the Challenge. The axiom $isCompatible$ is true if the notebook sold by the service provider can be used with a discovered DockingStation. This axiom can be used in the goal query to check compatibility of the two components.

We create WSMO semantic descriptions of the services provided by the Challenge. Service descriptions include data-fetch interfaces, for the dynamic properties services may have, that are only available on request at run-time. The capability of service descriptions may use their own logical definition of the *isComaptible* relation to specify constraints of the service (e.g. IBM laptops only fit IBM docking stations). For each of the tasks specified by this part of Challenge, we specified a WSMO goal. An example for the correlated composition goal is shown below in listing 2. This specifies that we are looking for a correlated notebook and docking station with specified constraints on memory and hard-drive capacity.

```
1   Goal GoalPurchaseHardware
2       ...
3       capability GoalPurchaseHardwareCapability
4       postcondition
5           definedBy
6               (?x[po#price hasValue ?priceX, po#GTIN hasValue ?gtinX,
7                   po#hddGB hasValue ?hddGBX, po#memoryMB hasValue ?
                        memMBX] memberOf po#Notebook
8                   and ?memMBX >= 512
9                   and ?hddGBX >= 40 and
10              ?y[po#price hasValue ?priceY, po#supportsGTIN hasValue ?
                    gtinY] memberOf po#DockingStation
11              and isCompatible(?gtinX, ?gtinY)
12              and ?price = (?priceX + ?priceY)
13          ).
14      ...
```

Listing 2. User Goal in WSMO

For each goal of the service requester (may be multiple if decomposition invoked) discovery is carried out. For the first goal only the ontologies of that goal, and each candidate service in turn, form the knowledge base for the discovery. In our example, we look for a service selling IBM laptops first. Once a matching service is found, its ontologies are stored (including the constraint that it only operates with IBM docking stations), and the process moves on to the next goal (search for docking stations). Now the knowledge base used for discovery is constructed from the ontologies of the goal and each Web service in turn *and* the ontologies of the Web service discovered earlier. During this discovery step, Web services that match the goal will only be considered if they do not break the constraints specified by services discovered beforehand e.g. a HP docking station will not be a successful match.

From a functional perspective, the solution provided was able to successfully complete all composition tasks speci-

---

[1]http://www.gtin.info/

fied by the SWS Challenge. The focus was on being able to handle both constraints on the services themselves and constraints on the goals that represented client requests. Additionally, it was possible to handle correlation constraints specified between services in a simple composition.

5 Related Work

Sheth identifies four types of semantics (data, functional, non-functional and execution) for Web services in [9] and there is ongoing discussion on this topic. We use WSMO-lite as it does not contradict this discussion and provides a defined formal semantics.

The service discovery aspect of this paper focuses on the use of instance data that may be fetched from service interfaces at run-time. This is in contrast to the static-description only approach of many semantic and non-semantic service discovery initiatives. For example in [7], subsumption reasoning is used over the concepts representing the post-conditions and effects of service requests and descriptions. Keyword-based discovery using ebXML and UDDI is described in [3]. Additionally in the course of our work in the Challenge we have detailed a comparison between our approach and that submitted by DEI Politecnico di Milano, and CEFRIEL Milano in [12].

For the purposes of the Challenge, we provide a solution for constrained service composition with the assumption that invocation order was unimportant. For business processes, this is usually not the case and there is a wealth of research in AI planning, workflow and business process management examining rich composition techniques. These include the work of McIlraith et al. [5] modelling requests and services using first-order situation calculus, McDermott [4] in planning using PDDL and DAML-S, Van der Aalst using workflow patterns [1], and Osman et al. [6] on bridging workflow with Semantic Web service based composition. There is also much research on constraint-based service composition such as [2]. The progression of our work is that the instance data on which constraints are evaluated can be fetched from services at run-time rather than extracted from a static service registry.

Acknowledgments

References

[1] W. M. P. v. d. Aalst, M. Dumas, and A. H. M. t. Hofstede. Web service composition languages: Old wine in new bottles? In *29th EUROMICRO Conference 2003*, Belek-Antalya, Turkey, 2003.

[2] R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint driven web service composition in meteor-s. In *SCC '04: Proceedings of the 2004 IEEE International Conference on Services Computing*, pages 23–30, Washington, DC, USA, 2004. IEEE Computer Society.

[3] A. Dogac, Y. Tambag, P. Pembecioglu, S. Pektas, G. Laleci, G. Kurt, S. Toprak, and Y. Kabak. An ebXML infrastructure implementation through UDDI registries and RosettaNet PIPs. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 512–523, 2002.

[4] D. McDermott. The 1998 ai planning systems competition. *The AI Magazine*, 21(2):35–55, 2000.

[5] S. McIlraith and T. Son. Adapting golog for composition of semantic web services. In *8th International Conference on Principles of Knowledge Representation and Reasoning.*, 2002.

[6] T. Osman, D. Thakker, and D. Al-Dabass. Bridging the gap between workflow and semantic-based web services composition. In *Business Process Management Workshop 2005*, Compiegne, France, 2005.

[7] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *1st International Semantic Web Conference (ISWC)*, pages 333–347, 2002.

[8] D. Roman, U. Keller, H. Lausen, J. d. Bruijn, R. Lara, M. Stollberg, A. Polleres, D. Fensel, and C. Bussler. Web service modeling ontology. *Applied Ontology Journal*, 1(1):77–106, 2005.

[9] A. P. Sheth. Semantic web process lifecycle: Role of semantics in annotation, discovery, composition and orchestration, invited talk, workshop on e-services and the semantic web, www, 2003. http://lsdis.cs.uga.edu/lib/presentations/www2003-esswinvitedtalk-sheth.pdf. Technical report, 2003.

[10] T. Vitvar, J. Kopecky, and D. Fensel. WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web. In *The 5th IEEE European Conference on Web Services, ECOWS 2007*. IEEE, 2007.

[11] T. Vitvar, M. Zaremba, and M. Moran. Dynamic Service Discovery through Meta-Interactions with Service Providers. In *Proceedings of the 4th European Semantic Web Conference (ESWC 2006)*, June 2007.

[12] M. Zaremba, T. Vitvar, and M. Moran. Towards semantic interoperability (in-depth comparison of two approaches to solving semantic web service challenge mediation tasks). In *Proceedings of the 9th International Conference on Enterprise Information Systems*, Funchal, Madeira - Portugal, 2007., 2007.