“ALEXANDRU IOAN CUZA” UNIVERSITY OF IAȘI

FACULTY OF COMPUTER SCIENCE

Master’s Thesis

Composition Algorithm for Relational Semantic Model

by

Ștefania-Liana Țucăr

Master Programme: Advanced Studies in Computer Science
July, 2019

Scientific Coordinator *Prof. Dr. Croitoru Cornelius*

Contents

1 Introduction **2**

1.1 Contribution 2

2 Motivating Example **4**

3 Problem Description **6**

3.1 Problem Definition 6

4 Algorithm **9**

4.1 Algorithm Description 9

4.1.1 Main Idea 9

4.1.2 Construction Phase 10

4.1.3 Search for service calls 12

5 Evaluation of the algorithm **13**

5.1 Testset Generator 13

5.2 Generated Tests 14

5.3 Tests from Competition 15

5.4 Evaluation Conclusion 15

6 Conclusion and Future Work **17**

1 Introduction

Service Oriented Architectures are very popular these years as it is a modular architecture. Services usually have their specific functionality and more complex scenarios use the services as building blocks. This way the same implementation of a functionality can be used multiple times in different scenarios, this being an improvement in production time as well as resources utilization.
Having a repository of web services, they can be combined in different ways to serve requests that otherwise could not be solved using only one service. As repository usually have a big number of services and requests should usually be answered fast, automating this process is a real need. This can be solved by automatic web service composition.
Among information about a specific service the following is of interest in the context of service composition: input parameters that can be used to call the service and preconditions (i.e. properties that objects used to instantiate the parameters should have); output parameters that are obtained after the service is called and postonditions (i.e. properties that are guaranteed that the output objects have).
Different models for representing services and matching parameters have been proposed in the context of web service composition. In the model presented WSC2005 competition the parameters are matched only by their names. The model is improved from the semantic point of view in the WSC2008 competition [1] where parameters are described using concepts from an ontology.

1.1 Contribution

A model that aims to leverage semantic information is proposed in [2]. Parameters are described by concepts from an ontology and also by relations between them. Preconditions that enable calling a service as well as postconditions (properties satisfied by the output of the service call) are expressed by binary relations between parameters. This provides a significant improvement in the expressivity of the model. Thus the composition obtained after modeling real services and user queries in the proposed way is supposed to be more valuable than the ones provided after modeling in a more simplistic way, because more semantic information is used in this model.
In [2] an algorithm for solving the composition problem (in the relational model) is described. A new version of the algorithm is presented in section 4 that proves to be faster and to obtain a shorter composition. In the new proposed algorithm the decision to call services is made tacking into consideration the semantic context of the objects thus resulting in a shorter composition and a faster algorithm.
A more detailed description of the problem in the proposed model is presented in section 3. A proposed algorithm for solving the web service composition problem under the proposed model is explained in section 4.

In section 5 the testset generator that was implemented to evaluate the algorithm is described as well as results of the proposed algorithm on the generated testcases and on public testcases that were modified to be compatible with the proposed model.

2 Motivating Example

Following is presented an example (presented also in [2]) that showcases the increase in expressiveness of the proposed model compared to the models based only on a hierarchy of concepts. The enhancement comes from adding relations between objects/parameters and inference rules that can be used to deduct more relations between objects.
The scenario is described below. A teacher is trying to schedule a meeting at some collaborating university. The following information is provided: identifier of the person, university where the teacher works, and the name of the university he wants to visit. The person is an instance of **Person** concept and both universities are instances **University** concept.
Two inference rules are defined: **locatedAtWorkRule**: if **pers** is an employee of **univ** and **univ** is located at **location** then **pers** is located in **location**; **destinationGenRule** as described in the Figure 1.

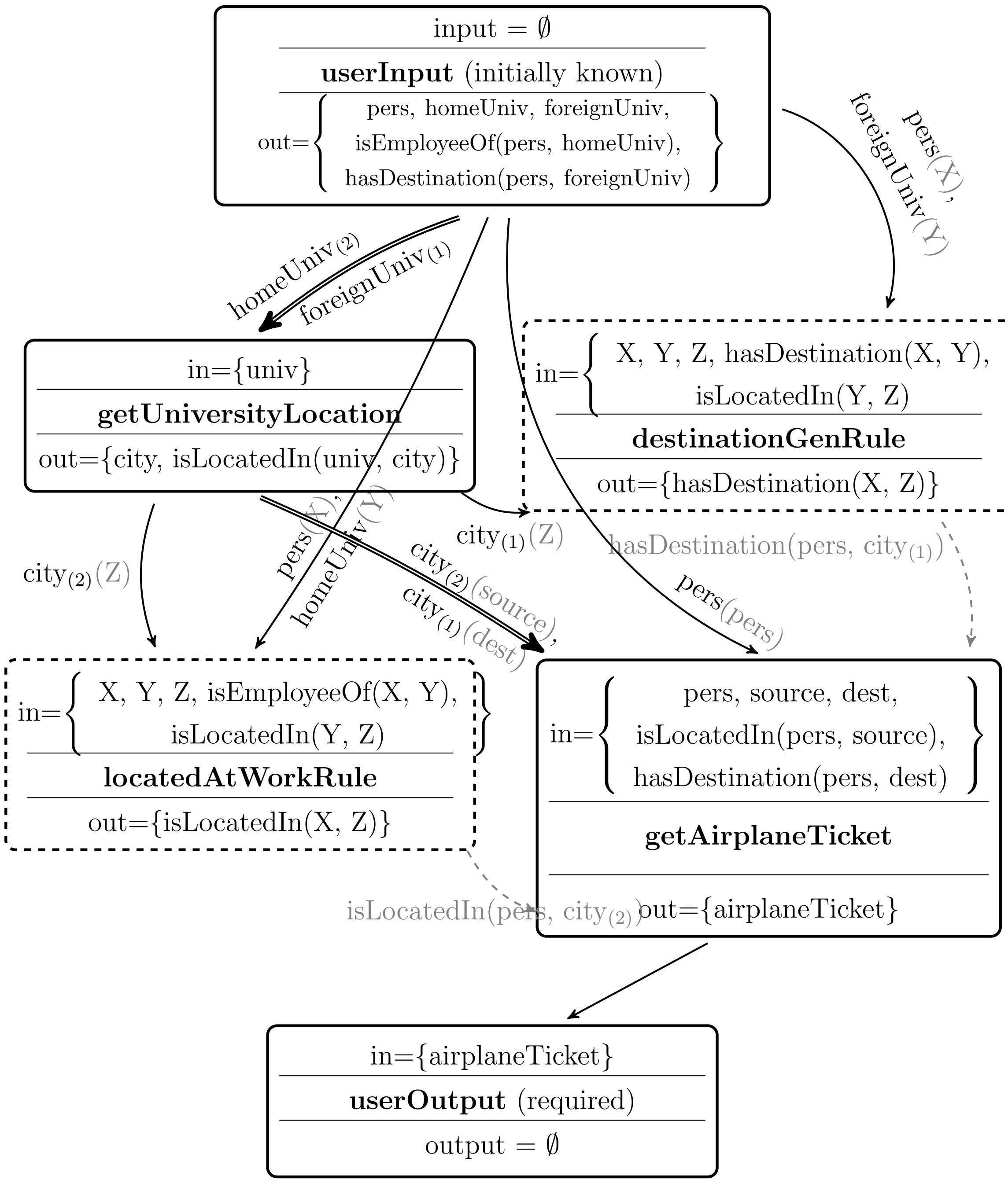
In Figure 1, solid boxes represent services and the user request (one for input - the initially known concepts and their relations, and one for the required output). Dashed boxes represent inference rules. Edges show information flow: solid edges - parameters and the dashed edges - relationships among them. Not all possible relations are used. Parameters are matched to rule variables (rule "parameters"), or other service parameters, based on the specification in gray in parenthesis. Multiple calls to the same service can be handled and are shown with double edges.

One composition representing a valid solution for the described instance would be the following list ordered list of service invocations:
getInput$(\emptyset) \Longrightarrow$ pers, homeUniv, foreignUniv, isEmployeeOf(pers, homeUniv), hasDestination(pers, foreignUniv);
getUniversityLocation$\big($homeUniv$\big) \Longrightarrow$ homeCity, isLocatedIn(homeUniv, homeCity);
getUniversityLocation$\big($foreignUniv$\big) \Longrightarrow$ foreignCity, isLocatedIn(foreignUniv, foreignCity);

The two cities: *homeCity* and *foreignCity* are differentiated based on their relations, the names are not relevant for the composition (there is no restriction on what names they get if they would be automatically created; i.e. any distinct strings would work).
locatedAtWorkRule$\big($pers, homeUniv, homeCity, isEmployeeOf(pers, homeUniv), isLocatedIn(homeUniv, homeCity)$\big) \Longrightarrow$ isLocatedIn(pers, homeCity);
destinationGenRule$\big($pers, foreignUniv, foreignCity, hasDestination(pers, foreignUniv), isLocatedIn(foreignUniv, foreignCity)$\big) \Longrightarrow$ hasDestination(pers, foreignCity);
getAirplaneTicket$\big($pers, homeCity, foreignCity, isLocatedIn(pers, homeCity), hasDestination(pers, foreignCity) $\Longrightarrow$ airplaneTicket;

In this example the possible need of different objects of the same type can be easily noticed - there are two different universities as instance of the same concepts, but each has a different semantic context described by relations: person **isEmployeeOf** Univ1 and person **hasDestination** Univ2.

The example could be modified to show the use of a hierarchy of objects, but for the



input = ∅
userInput (initially known)
out={pers, homeUniv, foreignUniv, isEmployeeOf(pers, homeUniv), hasDestination(pers, foreignUniv)}
pers(X), foreignUniv(Y)
homeUniv(2)
foreignUniv(1)
in={univ}
getUniversityLocation
out={city, isLocatedIn(univ, city)}
in={X, Y, Z, hasDestination(X, Y), isLocatedIn(Y, Z)}
destinationGenRule
out={hasDestination(X, Z)}
city(1)(Z)
hasDestination(pers, city(1))
city(2)(Z)
pers(X), homeUniv(Y)
city(2)(source), city(1)(dest)
pers(pers)
in={X, Y, Z, isEmployeeOf(X, Y), isLocatedIn(Y, Z)}
locatedAtWorkRule
out={isLocatedIn(X, Z)}
in={pers, source, dest, isLocatedIn(pers, source), hasDestination(pers, dest)}
getAirplaneTicket
out={airplaneTicket}
isLocatedIn(pers, city(2))
in={airplaneTicket}
userOutput (required)
output = ∅

Figure 1: Example motivating the need for relations and rules in composition language.
Service parameter types are not shown for simplicity.

sake of simplicity it is kept this way. It could for example have a hierarchy of concepts where University is a subconcept of Builing and the service getUniversityLocation is modified as getBuildingLocation; so the getBuildingLocation could be called with university as input parameter for building because university is a building (according to the hierarchy of concepts).

3 Problem Description

Given a repository of services and an user query the objective is to find a composition that can answer the query. Each service is described by an identifier, a lists of input parameters, a list of preconditions (relations that need to be satisfied to call the service), a list of output parameters and a list of postconditions.
For representing the possible types and relations between objects (and parameters) an ontology is defined. The ontology is defined as a hierarchy of concepts, each concept being identified by its' name; a list of relation names and a list of inference rules.
A composition is an ordered list of service invocations. More precisely it is a list of service identifiers and if more details are useful, the objects used to call each service can be included.

3.1 Problem Definition

Let $Names$ be a list of all possible identifiers used to differentiate between concepts, parameters, services. It can be simply a list of strings over a predefined alphabet.

Concepts and Parameters Let $\mathcal{C}$ be the set of all concepts that can be used to describe object and parameter types. Each concept is identified by a name.
$Params \subseteq Names \times \mathcal{C}$. Let $Params$ be the set of all possible parameters. A parameter has a name and a type from the concepts hierarchy.

Ontology $Ontology = (Hierarchy, Relations, Rules)$;
$Hierarchy = (\mathcal{C}, subType)$ is a tree of concepts.
$subType \subseteq \mathcal{C} \times \mathcal{C}$ transitive acyclic relation that describes the types hierarchy.
$Relations \subseteq \mathcal{P}(Names)$, where $\mathcal{P}$ represents the power set.
$Rules$ = set of inference rules over Relations.

Repository $Rep = \{\mathcal{S}erv_1, ..., \mathcal{S}erv_n\}$.
The *repository* is a set of services identified by names.
$\mathcal{S}erv = (name, inputParams, outputParams, preConditions, postConditions)$;
$inputParams, outputParams \subseteq \mathcal{P}arams$;
$preConditions \subseteq \mathcal{P}(Relation \times inputParams \times inputParams)$
$postConditions \subseteq \mathcal{P}(Relation \times Par)$;, where $Par = (inputParams \cup outputParams) \times (inputParams \cup outputParams) \setminus (inputParams \times inputParams)$;

Each *service* is defined by a name, two lists of parameters (input and output) and two lists of relations between parameters (preconditions and postconditions). The preconditions are relations only between input parameters while the postconditions are relations between input and output parameters or output and output parameters.

Query $Query = (providedParams, requiredParams, providedRels, requiredRels)$;
$providedParams, requiredParams \subseteq Params$;
$providedRels \subseteq \mathcal{P}(Relation \times providedParams \times providedParams)$
$requiredRels \subseteq \mathcal{P}(Relation \times Par)$;, where $Par = (providedParams \cup requiredParams) \times (providedParams \cup requiredParams) \setminus (providedParams \times providedParams)$;
The query is defined by the provided parameters and relations which describe the types of the objects provided and the relations between them and by the required parameters and relations which describes the types and relations that the objects obtained after the composition need to satisfy.
It has a similar structure with a service, but replacing input parameters/relations with provided parameters/relations and output parameters/ relations with required parameters/ relations. Their meaning is somehow reversed.

Knowledge $Knowledge = (Objects, Relations)$;
$Objects = \{Obj_1, ..., Obj_k\} \subseteq Names \times Concepts$;
$Relations \subseteq Relations \times Objects \times Objects$.
The Knowledge is a tuple formed by a set of objects and a set of relations defined on those objects. It represents the set of all objects obtained after service calls or directly provided from the user query together with the known relations that hold between objects.
Each object has a specific type (defined by a concept) and a name. There can be multiple objects of the same type.

Service Call $ServCall = (Serv, objAssignation)$;
$Serv = (name, inputParams, outputParams, preConds, postConds)$;
$objAssignation : inputParams \rightarrow Knowledge$.
A service can be called assigning an object to each of its input parameters. The objects assigned need to "match". An object matches to a parameter if the type of the object is equal or more particular to the type of the parameter (according to the types hierarchy defined in ontology). Besides this condition the objects need to satisfy all the preconditions the service defines so all the relations from preconditions between parameters need to hold for the objects assigned to parameters.
The service together with the objects assigned for the input parameters form a "service call".

Composition $Comp = (ServCall_1, ..., ServCall_m)$;
A composition is an ordered list of service calls (services and objects used to call the services). Each service call needs to be possible i.e. objects from knowledge can be assigned to all input parameters.

Problem Instance Input: $(Rep, Ontology, Query)$;
Output: *Composition* that solves the *Query*.

Given a repository of web services, the ontology and an user query, a valid composition that solves the user query should be returned. The composition solves the query if all the required parameters can be found in the knowledge obtained after all service calls (within the composition) are made. Meaning that for every required parameter (from query) a corresponding object with matching type needs to be found in the knowledge. Besides this condition, the required relations (from query) between parameters need to hold for objects used to instantiate them.

4 Algorithm

4.1 Algorithm Description

4.1.1 Main Idea

During the construction of the composition a set of objects and relations between objects is kept in memory. This set is called "knowledge".
In the beginning of the algorithm the knowledge is formed only of the objects and relations from the query provided information (query.providedParams and query.providedRels) - this being the information provided by the user in order to obtained the desired information (query.required).
As the composition grows (i.e. services can be called and added to composition and new objects and relations are obtained as a consequence) objects and relations are added to the knowledge until no more services can be called or until the query can be answered (i.e. the required parameters and relations are obtained).
To handle inference rules in a simple manner, in the initialization phase fake services are added in the repository. For each inference rule a corresponding fake service is created and added in the repository. This step does not significantly increase the execution time of the algorithm as the number of inference rules is usually smaller than the number of initial services.
During all the following algorithm the repository, ontology, query, knowledge and composition are considered global values (for writing simplicity).

```
Data: repository, ontology, query
Result: Composition that solves the query if it finds one; "Not Solved"
        otherwise.
initialize data structures and create fake services for inference rules;
$changed \leftarrow True$;
while $\neg canAnswerQuery(query)$ $And\ changed = True$ do
    $(changed\ ,\ composition,\ knowledge) \leftarrow constructionPhase()$;
end
if $canAnswerQuery(query)$ then
    return $composition$;
end
else
    return $Not\ Solved$;
end
```

Algorithm 1: Main function

During the algorithm there is a need to know if the query is solved. Multiple calls to this function are made.
The query is said to be solved if objects with the required properties (i.e. proper types

and relations between them) can be found in the knowledge. A fake service is constructed with the property that being able to call that service is equivalent to solving the query. The service is not added in the repository. It is just used for checking if the query is solved.
The service has as input parameters all parameters from the query and as preconditions all the relations from the query. Despite the fact that query.providedParams are always in the knowledge they still need to be added in the fake service input parameters. This happens because some of the required relations from the query can be relations between input and output parameters. So to be able to check all required relations, all the parameters from the query need to be added as service input parameters.
If the constructed fake service is callable, then the query can be answered (and the algorithm will stop).

```
Data: query, knowledge
Result: $True$ if the query is solved; $False$ otherwise.
Function canAnswerQuery($query, knowledge$):
    $serv \leftarrow newService()$ ;
    $serv.inParams \leftarrow query.providedParams \ \cup \ query.requiredParams$ ;
    $serv.preconditions \leftarrow query.providedRels \ \cup \ query.requiredRels$ ;
    $serv.outParams \leftarrow \emptyset$ ;
    $serv.postconditions \leftarrow \emptyset$ ;
    return $canCallService(serv, knowledge)$ ;
```
Algorithm 2: Function that checks if the query is solved using objects in the knowledge

4.1.2 Construction Phase

In the construction phase all the services (including the fake ones created for handling the inference rules) from the repository are iterated and for each service all the possible calls are searched. A service can be called multiple times with different input parameters. From all the possible calls of a service only the ones that provide "useful" information are added to the composition.
If at least one service call was added in the composition, the composition is updated accordingly and the function returns "True" (for the calling function to continue the algorithm run).

```
Data: repository, ontology, query
Result: True if services were added to the composition; False otherwise
Function constructionPhase():
    $compositionUpdated \leftarrow False$ ;
    foreach $service \in repository$ do
        $possibleCalls \leftarrow searchForPossibleCalls(service)$ ;
        foreach $servCall \in possibleCalls$ do
            if $providesUsefulInformation(servCall)$ then
                $makeCall(servCall)$ ;
                $compositionUpdated \leftarrow True$ ;
            end
        end
    end
    return $compositionUpdated$
```

Algorithm 3: constructionPhase

When a service call is made new objects and relations corresponding with the service output and postconditions are created and added in the knowledge.

If for all the new objects that would be created after a service call other objects that are similar or better than them already exist in the knowledge than the service call should not be made. If the service call would be made than no useful result is obtained, but the knowledge size would grow.

A service call is useless if all the new objects obtained are "semantically similar" with other already present in the knowledge. To obtain the semantic context of an object the connected component from the knowledge that would contain the object (if added in the knowledge) needs to be extracted.

To check if the service call provides useful information a fake service is created and checked if it can be called. The fake service has as input parameters and preconitions the semantic context of all objects obtained after the service call (i.e. the connected components of the objects as explained above).

If the fake service created (but not added to the repository) can be called, then the service call would be useless (all the information obtained already exists in the knowledge).

```
Data: servCall, knowledge
Result: $True$ if the service call provides useful information; $False$ otherwise.
Function providesUsefulInformation($servCall$):
    $params \leftarrow servCall.service.inParams \cup servCall.service.outParams$ ;
    $objects \leftarrow \{obj \mid obj = servCall.assignment[param]; param \in params\}$ ;
    $rels \leftarrow \{rel \mid rel \in knowledge; rel.o1, rel.o2 \in objects\}$ ;
    $serv \leftarrow newService()$ ;
    $serv.inParams \leftarrow \{newParam(type = obj.type) \mid obj \in objects\}$ ;
    $serv.preconditions \leftarrow \{newRelation(rel.name, o1 = x, o2 = y) \mid$
     $rel \in rels; x = paramToObj[rel.x]; y = paramToObj[rel.y]\}$ ;
    $serv.outParams \leftarrow \emptyset$ ;
    $serv.postconditions \leftarrow \emptyset$ ;
    return $\neg canCallService(serv, knowledge)$ ;
```
Algorithm 4: Function that checks if the service call provides "new" information

4.1.3 Search for service calls

Given a service, the task of finding all possible service calls means finding all combinations of objects that can be used as input parameters for the service. This involves finding for each input parameter a corresponding object that has a type that is equal or more general with the parameter type. Besides this condition regarding the types, the objects found need to satisfy all declared preconditions from the service definition (i.e. all relations from preconditions need to hold between corresponding objects used to call the service).
This problem is equivalent to finding all labeled subgraph isomorphisms in the following constructed problem instance:

- the "pattern" graph: for each input parameter a corresponding vertex with the label equal to the type of the parameter. For each relation from preconditions a corresponding directed edge between vertices correspondent to parameters involved in relation is added in the graph.

- the graph in which the pattern searched: for each object in the knowledge a vertex with the label equal to the type of the parameter. For each relation that holds between objects from the knowledge a corresponding directed edge is added in the graph.

The problem is known to be NP-Complete (as stated for example in [3]).

A backtracking procedure was implemented to solve this problem. For each input parameter all the known objects of matching types are iterated and all the preconditions involving the parameter and another parameter that has an object already chosen are checked. If all these relations match, than the object is chosen for this parameter and the backtracking procedure is recursively called for the next parameters.

5 Evaluation of the algorithm

5.1 Testset Generator

In order to evaluate the algorithm presented in Chapter 4, a test set generator was implemented. The generator is a nondeterministic program that produces: a repository, ontology and user query that has a high probability of being solvable. These elements form a problem instance.

In the first phase of generation, it produces the repository: a set of services that have the property that a composition that uses more than one service can be produced using them.

To generate such services, the following observation is useful: as the composition grows (i.e services are "called" meaning that they are added in the composition), new objects and relations between objects are obtained so the knowledge is enhanced. So, to generate the repository, we start by generating and saving stages of the knowledge (that would correspond to the knowledge growth as services ar added to the composition).

To obtain the repository we need to create services that can generate these stages. Between each two consecutive stages K_i, K_{i+1} we generate a "layer" of services with each service having as input parameters objects and relations between them from K_i and output parameters in a similar way from K_{i+1}, as shown in Figure 2. The repository is formed by all the services produced in this step (saved in an arbitrary order).

The ontology is formed by all the concepts produced while generating the knowledge stages. To obtain a taxonomy of concepts, they are arranged in a random hierarchy having a most general concept as the root of the produced tree. Relations and inference rules are generated and added to the ontology.

The user query is generated also using the knowledge stages: the provided parameters and relations of the query are a subset of the first stage of the knowledge, while the required parameters and relations are a subset of the last stage.

To make the problem instance harder to solve more concepts, services and relations are added in the repository and ontology. This can be viewed as a noise adding phase.

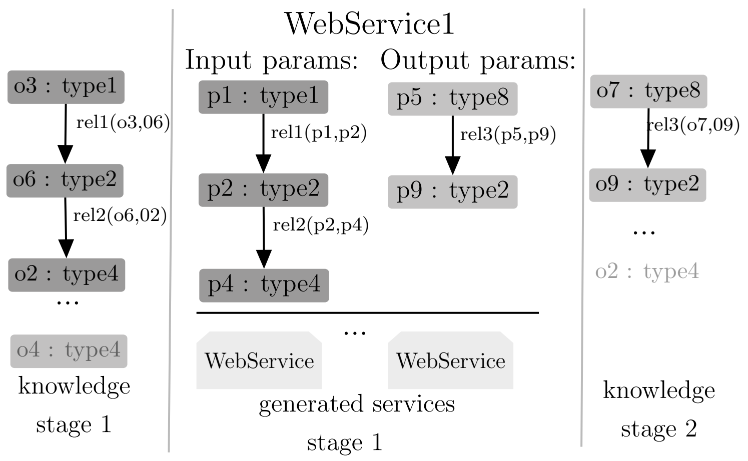


Figure 2: Consecutive stages of knowledge and services generated between the stages.

5.2 Generated Tests

In Table 1 results on some of the generated tests are presented. The algorithms compared are: the proposed algorithm without the use of inference rules (Alg 1); the proposed algorithm (Alg 2); the algorithm presented in [2] (Alg 3).
Times are measured in milliseconds.

Looking at test number 5 the use of inference rules can be noticed (as the algorithm

	Alg 1			Alg 2			Alg 3		
test	#serv	#rules	exec t	#serv	#rules	exec t	#serv	#rules	exec t
1	3	0	6	3	1	7	4	3	11
2	2	0	67	2	0	66	1	0	6
3	8	0	14	8	2	15	31	9	73
4	7	0	27	7	2	3	35	11	2433
5	N/A	N/A	N/A	7	3	152	37	74	272
6	5	0	72129	5	2	71530	N/A	N/A	N/A

Table 1

without inference rules does not solve the test while the ones that use inference rules do). On the most of the test cases the proposed algorithm (Alg 2) obtains a shorter composition and in a less amount of time than Alg 3. This happens in five out of six tests presented. However on the second test Alg 3 behaves better.
Test number 6 is the most "difficult" one: the size of the repository is the biggest. Alg 3 doesn't solve it under two minutes (that's the maximum time the evaluator lets the algorithms run on each test case).

5.3 Tests from Competition

To see how the algorithm performes compared to other algorithms public tests are used. Some of the tests were used in a public competition [4] to evaluate the algorithms.
As the model proposed is more general than the one in the WSC08 competition, the tests can be transformed to be compatible with the proposed model. One problem instance is transformed in the following way. The services in the repository are the same (input and output parameters are the same as in the original test; preconditions and posctonditions are the empty set). The ontology is formed by the hierarchy of concepts from the original test and no rules and relations are defined. The query is kept the same (input/output parameters are the same; preconditions and postconditions are the empty set). The algorithm is compared with the algorithm presented in [2] (Alg 1) and with the algorithm presented in [5] (Alg min). The latter is an algorithm that finds the composition that contains the minimum number of services possible. As it can be noticed in table 2 the

	Alg 1		Proposed		Alg min	
	#serv	exec time	#serv	exec time	#serv	exec time
WSC 01	57	171	33	17	10	91
WSC 02	46	133	35	33	5	123
WSC 03	461	6772	104	291	40	1929
WSC 04	95	418	43	101	10	314
WSC 05	180	1198	91	132	20	6356
WSC 06	274	2535	186	288	42	777
WSC 07	269	2367	124	465	20	9835
WSC 08	556	9820	122	1105	30	6398

Table 2

proposed algorithm is faster than Alg 1 and obtains a shorter composition on all testcases. Both algorithms lack a phase of reduction the size of the composition as their primarily objective is not trying to optimize the size of the composition. But if they would implement such a functionality the proposed algorithm seems to have a head start.
Comparing the results of the proposed algorithm with "Alg min" the latter finds a shorter composition (as it was optimized to do so), but the first one is faster.

5.4 Evaluation Conclusion

The algorithm is fast, obtaining a composition in the least amount of time compared with the presented algorithms on most of the test cases. Even though the algorithm was designed for a more general model than the one that only contains a hierarchy of concepts, it still runs faster than an algorithm specially optimized for that model.
The size of the composition is bigger than the size of the composition obtained by the

algorithm presented in [5], but of comparable size (tacking into consideration that the algorithm was not optimized for this dimension and was constructed on a more general model).
As noticed from table 1 the algorithm that uses inference rules and the one doesn't handle them have a comparable speed. This shows that the way the inference rules are implemented (using fake services) does not significantly affect the time complexity of the algorithm.

6 Conclusion and Future Work

The presented model aims to preserve contextual information about service parameters with the use of relations. This model is more expressive (as it can be also noticed from the motivating example presented) than models that use only a hierarchy of concepts.
In the presented model the problem of finding a composition becomes computationally harder as to decide if a service can be called is now a NP-Complete problem while in the models that only use a hierarchy of concepts polynomial algorithms are presented in multiple articles (for example in [6]).
However the problem still seems solvable in a decent amount of time as the evaluation of the presented algorithm shows. This result is also due to the optimizations made in the algorithm (for example tacking in consideration the semantic context of objects to decide if the information obtained after making a service call is useful).
There are many paths of continuation. One would be maturing the algorithm: optimizing it to run faster and also to minimize the composition. For both optimizations a scoring heuristic can be implemented and a reduction phase similar with the ones presented in article [7].
Tacking into consideration Quality of Services [8] and optimizing accordingly when construction the composition would be another useful improvement.

References

[1] S. L. G. C. S. Department, *Semantic Web Services Challenge: Proceedings of the 2008 Workshops.* 2009.

[2] A. N. Paul Diac, Liana Tucar, *Relational Model for Parameter Description in Automatic Semantic Web Service Composition.* following publication in 23rd International Conference on Knowledge-Based and Intelligent Information Engineering Systems, 2019.

[3] C. S. M. V. L. P. Cordella, P. Foggia, *A (sub) graph isomorphism algorithm for matching large graphs.* IEEE transactions on pattern analysis and machine intelligence, vol. 26, no. 10, pp. 1367–1372, 2004.

[4] K. G. Steffen Bleul, Thomas Weise, *Proceedings of the Workshop on Service-Oriented Computing.* KIVS, 2009.

[5] M. L. Pablo Rodriguez-Mier, Manuel Mucientes, *Automatic web service composition with heuristic-based search algorithm.* IEEE International Conference on Web Services, 2011.

[6] P. D. Liana Tucar, *Semantic Web Service Composition based on Graph Search.* Knowledge-Based and Intelligent Information and Engineering Systems: Proceedings of the 22nd International Conference, KES-2018, Belgrade, Serbias, 2018.

[7] P. Diac, *Engineering Polynomial-Time Solution for Automatic Web Service Composition.*

[8] M. B. B. Srividya Kona, Ajay Bansal, *WSC-2009: A Quality of Service-Oriented Web Services Challenge.* IEEE, 2009.