Scalable precondition-aware service composition with SPARQL

Alexis Huf, Frank Siqueira
Graduate Program in Computer Science, Department of Informatics and Statistics
Federal University of Santa Catarina
Florianópolis, Brazil
alexis.huf@posgrad.ufsc.br, frank.siqueira@ufsc.br

***Abstract*—Functional service composition builds a plan that fulfills some user-provided goal using available services. The literature describes several approaches for description of services and goals, with different levels of expressiveness, ranging from the ones based only on inputs and outputs to the more expressive logics-based solutions. The former provide better scalability and performance, while the latter allow for increased expressivity in preconditions and effects of both services and goals. The approach proposed in this paper aims to achieve a balance between expressivity and performance of functional service composition. To this end, a graph-based composition algorithm is extended to support preconditions and effects described with a small subset of SPARQL. Experiments compare the performance of this extended algorithm with two state-of-the-art algorithms that support preconditions and effects and demonstrate better scalability. In one case, a maximum speedup of 29 times was achieved for the problems that could be expressed with the SPARQL subset. In another, problems that a state-of-the-art algorithm could not solve after 5 minutes where solved in seconds.**

***Keywords*-service composition, automated planning, preconditions and effects, semantic web**

I. Introduction

Most service description languages currently in use model services by their name, inputs and outputs [1]. For stub generation and manual composition of these services into new services and applications, this level of detail is sufficient. If service composition is done by a specialist, any missing information from the descriptions can be obtained from human-readable documentation or by testing invocations of the service.

If the composition is performed by an algorithm that seeks fulfillment of functional requirements, a process called *automatic functional composition* [2], then issues may arise due to low expressivity of descriptions. As a first example, consider a service called *Pay* that takes an *Order* as input and returns a *Receipt* as output. If requested to *Pay* an *Order*, a composition algorithm is able to discover and invoke such service. But if it is requested to *Ship* an *Order*, an issue arises: How does the composition algorithm will determine if it needs to pay an order before shipping it? Since the order may have already been paid, calling *Pay* again could result in an error. As a second example, consider a service that receives a *Location* (city or neighborhood) and a *BusinessSector* and returns a list of *Business*. Are the returned businesses located there, just operate in the location, or were founded by a resident of the location? Do the returned businesses operate on the given sector or are they suppliers to other businesses in it?

The issues identified above can be solved by formalizing the preconditions and effects of services so that they can be used by a composition algorithm. In the first case, "the *Receipt* is for the *Order*" is an effect of *Pay* and "the *Order* has a *Receipt*" is a precondition of *Ship*. In the second example, the effect is "*Business* operating on *Sector* and located in *Location*". While presented here informally, such preconditions and effects must be formally described to be processed by the composition algorithm.

There are efforts for service discovery based on preconditions and effects, among other attributes of services [3]. There are also approaches focusing on describing or inferring the relation between Inputs and Outputs (I/Os) of a service [4], [5]. Such relations can then be used as preconditions and effects by composition algorithms proposed in the literature [6], [7], [8], [9]. However, when experiments involving such composition algorithms are reported, the magnitude of composition times is often higher than that of times observed for I/O-only composition algorithms [10], [11], despite experimental scenarios with few services. Another issue is that approaches that employ general-purpose reasoners [8], [7] generate logical problems with a specific structure (e.g., in [8] a service is an implication with a description of the HTTP request as antecedent and of the response as the consequent), for which general-purpose reasoners may not be optimized.

The goal of this paper is to design a composition algorithm – and associated description for preconditions and effects – that achieves performance similar to I/O-only composition algorithms. This algorithm is implemented and evaluated in the Unserved service composition architecture (originally designed for composition of heterogeneous services), described in our previous work [12].

This paper presents an efficient composition algorithm that supports preconditions and effects containing binary predicates ($P(x, y)$), conjunction and disjunction. This algorithm is obtained extending the I/O-only algorithm proposed in [12]. Datasets that were used by related work for evaluation are converted into Unserved descriptions, allowing comparison of performance results.

The remainder of this paper is structured as follows. Section II surveys state-of-the-art composition algorithms that consider preconditions and effects. An overview of the Unserved architecture is given in Section III and the precondition-aware algorithm is described in Section IV. Evaluation experiments are described in Section V, and the concluding remarks are given in Section VI.

II. Related Work

Mohr et al. [6] propose a composition algorithm that supports a subset of First-Order Logic (FOL) to describe preconditions and effects. This subset does not support disjunction ($\vee$) nor quantification. A global knowledge base, limited to Horn clauses[1] can be provided as input along with service descriptions. The composition algorithm explores a graph of services, starting from the goal formula. Any service whose post-conditions partially satisfy preconditions is a predecessor node and retains unsatisfied preconditions of the successor as their own. This rule defines a graph in which a path from the goal node to a node without unsatisfied preconditions is a solution.

Alves et al. [7] tackle the problem of composing non-deterministic services, whose actual effects may be desirable or not. Preconditions are modeled as conjunction and effects as the disjunction of all possible effects, each modeled by a conjunction. The algorithm reduces the composition problem to a non-deterministic planning problem, which is then reduced to the boolean satisfiability problem (SAT). The planning output is a contingency tree which determines how to proceed if, during execution, a service produces undesirable effects. This approach allows generating a Business Process Execution Language (BPEL) process that superimposes all alternative plans. The drawback of this approach is that all combined failure possibilities must be explored on the contingency tree, even if few or no failures occur.

Some approaches [8], [9] for service composition describe services using the Resource Description Framework (RDF). One of the advantages of RDF is the ability to refer to ontologies when conceptually describing service I/Os. RDF structures data as a graph, defined by a set of subject-predicate-object triples that are analogous to $P(subject, object)$ in FOL. Other FOL-like constructs, such as disjunction and quantified variables, can be expressed using RDF extensions, such as N3Logic [13], or through SPARQL RDF Query Language (SPARQL)[2] queries. While both approaches can be used to describe preconditions and effects of services, SPARQL has the advantage of being the standard querying language for RDF [14].

Verborgh et al. [8] propose an algorithm that interleaves planning and execution in order to support the Hypermedia As The Engine Of Application State (HATEOAS) constraint of the Representational State Transfer (REST) architectural style[3] [15]. Planning is reduced to theorem proving under *N3Logic* [13] by modeling services as implications. Preconditions and effects are restricted to conjunction of binary predicates (i.e., triple patterns such as `?x a foaf:Person`). After every interaction with a service, results are added to a knowledge base and the proof is re-done, yielding an updated plan. The authors themselves constructed a benchmark, which consists in linking available services in an invocation chain. However, service I/Os are described in such a way that each service has only a single suitable predecessor, yielding a linear search space.

Serrano et al. [9] propose a middleware where every service is described as an RDF graph, with inputs and outputs of the service mapped to nodes of this graph. The end user requests a SPARQL query, which yields a desired graph. Employing graph isomorphism, any service whose graph intersects with the desired graph is selected. If the selected services have unknown inputs, the algorithm expands the desired graph to satisfy them and recurses until the desired graph is completely satisfied. The level of support of preconditions and effects in this approach is similar to that in [8]. The authors evaluate several properties of the middleware and demonstrate its applicability, but do not evaluate the performance of a prototype.

Most functional service composition approaches that do consider preconditions and effects, do so by reducing composition to general-purpose theorem provers or planners, the only exception being [6]. This is often accompanied by two difficulties: (1) efficient interaction with the general-purpose tool, that may require spawning a new process, feeding it with input data and parsing its output; and (2) exploring search space reduction strategies specific to the particular encoding of the composition problem as a logic problem. Another observation is that each approach uses a distinct benchmark, making direct comparisons of performance difficult. Nevertheless, by the description of benchmarks and the provided results, it can be concluded that composition considering preconditions and effects is more computationally challenging than I/O-only composition [10]. These observations motivated the methodology adopted in this paper of adding support for preconditions and effects to an efficient I/O-only algorithm.

III. Unserved

The composition algorithm presented in this paper was implemented inside the Unserved software architecture, whose main goal is to support composition (i.e., planning and execution) of heterogeneous services, mainly SOAP,

[1]Horn clauses have the form $p_1 \wedge \ldots \wedge p_n \rightarrow q$ and are the basis for logic programming.

[2]http://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[3]Under such constraint, the client of a RESTful service must determine its next action from hypermedia controls contained in the representation received after interacting with a resource.

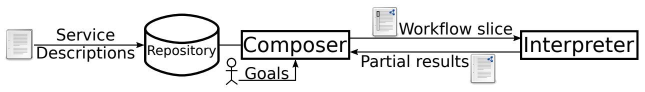

Figure 1: Overview of service composition under Unserved.

RESTful and event-oriented services [12]. This platform provides an abstraction on service interaction and interpretation of service results, which makes it a suitable starting point for comparing different composition algorithms. Another benefit of this platform is that it already implements some state-of-the-art service composition algorithms, such as ComposIT [10] and some of its variants [11]. Figure 1 provides an overview of how composition is done in Unserved. Service descriptions are transformed into an intermediate description and stored, as RDF, into a service repository. A composer synthesizes a workflow using those services to achieve user-provided goals. The workflow is split into slices, allowing the composer to change it during execution if a service fails to deliver an expected result.

The main composition algorithm in the Unserved implementation is an extension of the ComposIT [10] algorithm. ComposIT models a service composition as a path within a state graph, derived from the relations between I/Os of services. Its main innovations were optimizations to prune the graph before applying a path-finding algorithm. Its implementation under Unserved retains the same rules for forming the state graphs and to prune it. However, it was extended to support features of the architecture, such as adapting the plan during execution.

IV. Precondition-aware Composition

Preconditions and effects of services are modeled in the intermediate description as `ASK` SPARQL queries and are encoded in RDF using SPARQL Inferencing Notation (SPIN)[4]. For preconditions, all variables in the query must be inputs of the service, whereas, for effects, both inputs and outputs are allowed. The SPARQL queries are treated as if under the RDF Schema (RDFS) entailment regime. This ensures that, like in most automated functional service composition literature that employs semantics [1], superclass/subclass are considered when checking satisfiability of preconditions.

Similarly to the ComposIT-based composition algorithm presented in [12], the precondition-aware algorithm follows the same general steps:

1) from the known inputs and assumed preconditions, build a layered graph of service dependencies;
2) remove services from the dependency graph which are not required to obtain any requested output or effect;
3) remove redundant services from the graph, storing them as alternatives to explore during adaptation;
4) build a goal state representing all goals being achieved and a start state representing all known inputs;
5) find a path from this goal state to the start state, where each state in between is a set of services originating from the same layer in the dependency graph.

[4]http://spinrdf.org/spin.html

```
procedure BuildDepGraph($I_k, \Phi_k, O_w, \Psi_w, \zeta_I$)
    $start \leftarrow \langle(\emptyset,\emptyset) \rightarrow (I_k, \Phi_k)\rangle$
    $i \leftarrow 0, L_0 = \{start\}, \zeta_O \leftarrow \emptyset, \zeta_\Psi \leftarrow \emptyset$
    IndexOutputsAndEffects($\zeta_o, \zeta_\Psi, L_0$)
    while $\neg$Satisfied($\zeta_O, \zeta_\Psi, \langle(O_w, \Phi_w) \rightarrow (\emptyset,\emptyset)\rangle$) do
        $L_+ \leftarrow \bigcup_{j=0}^{i} L_j$
        $i \leftarrow i + 1$
        $L_i \leftarrow$ PartialylStatisfiedBy($\zeta_I, L_i$)
        $L_i \leftarrow L_i \setminus (L_* \cup \{c \text{ s.t. } \neg Satisfied(\zeta_O, \zeta_\Psi, c)\})$
        IndexOutputsAndEffects($\zeta_O, \zeta_\Psi, L_i$)
```

Figure 2: Dependency graph construction.

Steps 2 and 3 will not be discussed in detail since they correspond to pruning strategies based purely on the set of predecessors and successors of a node in the graph [10]. Due to this generality, the same pruning strategy can be applied to IO-only and to the precondition-aware algorithm. The service dependency graph is built according to the procedure in Figure 2, from the set of known inputs (I_k), preconditions already satisfied (Φ_k), wanted outputs (O_w), wanted effects (Ψ_w) and service input index (ζ_I). This latter index acts as an interface to the service repository, providing the set of services that have at least one input satisfied by a given output. In the algorithm, a service is represented as $\langle(Inputs, Preconditions) \rightarrow (Outputs, Effects)\rangle$.

The dependency graph is forward-built, layer by layer, from the start node. In addition to ζ_I, two indices are queried and built during the process: ζ_O gives the set of services whose outputs satisfy a particular input and $\zeta\Psi$ gives the set of services whose effects satisfy a given precondition. The Satisfied function determines if all inputs and preconditions of a service are satisfiable using the services from previous layers. When a service has multiple preconditions, all of them must be satisfied. In the case of preconditions employing `UNION`, at least one branch needs to be satisfied. The edges between services in the dependency graph are derived from the results of querying ζ_I and $\zeta\Psi$ with each input and precondition of a service.

The states graph, like the dependency graph, is also layered. A state $S = \langle i, \hat{S}, \mathring{S}\rangle$ at layer i consists of a set of nodes ($\hat{S} \subseteq L_i^*$) and equivalence sets ($\mathring{S}$) between I/Os of the nodes. L_i^* is a split into three types of nodes: service nodes at layer L_i of the dependency graph; jump nodes with the form $\langle(O,\Psi) \rightarrow (O,\Psi)\rangle$ that refer to services in any L_j with $j < i$; and indexed nodes which are duplicates of the previous two types introduced to avoid violations of equivalence sets. The first two types of nodes are denoted by L_i^+. The equivalence sets determine which I/Os are known to share the same value during execution. An edge

```
1:  procedure PREDECESSORS($\langle i, \hat{S}, \mathring{S} \rangle$)
2:      $E \leftarrow \emptyset$
3:      for $m_\phi \in \{\{\varphi \to (\psi \in \Psi(s), s \in L^{+}_{i-1})\}\}$ do
4:          $m'_v \leftarrow \emptyset$
5:          for $(\varphi \to (\psi, s = \langle (I, \Phi) \to (O, \Psi) \rangle)) \in m_\varphi$ do
6:              if $s$ is a jump node then
7:                  for $v \in vars(\varphi)$ do
8:                      $m'_v(v) \leftarrow (v, s)$
9:              else
10:                 for $(v, u) \in unify(\varphi, \psi)$ do
11:                     if $u \in O$ then
12:                         $m'_v(v) \leftarrow (u, s)$
13:         $F \leftarrow inputs(\hat{S}) \setminus keys(m_v)$
14:         for $m_F \in \{\{v \in F \to (u, s)\}\}$ do
15:             $m_v \leftarrow m'_v \cup m_F$
16:             $\hat{T}' \leftarrow \{s \text{ s.t. } \exists (x \to (y, s)) \in (m_v \cup m_\varphi)\}$
17:             $\mathring{T} \leftarrow Equivalences(\mathring{S}, m_\varphi, m_v)$
18:             $\hat{T} \leftarrow \hat{T}'$
19:             for $t \in \hat{T}'$ do
20:                 for $(v, j) \in$ CONFLICTS$(t, \mathring{S}, m_v)$ do
21:                     $\mathring{T} \leftarrow \eta_j(t)$
22:                     $m_v(v) \leftarrow \eta_j(m_v(v))$
23:             $n_T \leftarrow |t \in \hat{T}$ s.t. $t$ is not a jump node$|$
24:             $E \leftarrow (\langle i-1, \hat{T}, \mathring{T} \rangle, \langle i, \hat{S}, \mathring{S} \rangle, \langle n_T, m_v, m_\varphi \rangle)$
        return $E$
```

Figure 3: Compute edges to predecessor states.

$(T, S, n_T, m_v, m_\varphi)$ from T to S contains a cost attribute n_T and two mappings: m_v maps every input in $\hat{S}$ to a output in $\hat{T}$, while m_φ maps every precondition in $\hat{S}$ to an effect in $\hat{T}$.

Figure 3 shows the procedure to obtain edges to a state from all its direct predecessors. The first step (line 2) is to generate all mappings from all preconditions of S to some effect of a service at a previous layer, using jump nodes as necessary. A mapping m_φ also implies a mapping m'_v of some I/Os (lines 4–12). For non-jump nodes, the mappings are determined by Prolog-like unification of the precondition with the effect. Since m'_v may be undefined for some inputs of S, possible mappings for those inputs are explored and integrated into the final m_v (lines 13-15). When $F = \emptyset$, the single possible value for m_F is $\emptyset$.

The final step to obtaining an edge from a predecessor to the given successor state is to update the equivalence sets $\mathring{T}$ and to split nodes into indexed nodes, if required by $\mathring{S}$. Equivalences in $\mathring{T}$ are implied by unification between preconditions and effects in m_φ and by transitivity in $\mathring{S}$: if equivalent inputs a and b map to c and d, all four I/Os will belong to the same equivalence class. If a single output a provided by a node s is mapped by m_v from two inputs that are distinct in $\mathring{S}$, then s must be split and one of the inputs must map to the output $\eta_1(a)$ of the new node $\eta_1(s)$. As this situation may occur with n equivalent classes CONFLICTS(r)eturns the input that must be re-mapped and a counter j that identifies to which of the $n-1$ splits the input must be mapped.

V. EXPERIMENTS

For evaluating the performance of the precondition-aware composition algorithm, the available datasets employed by state-of-the-art precondition-aware approaches were used. Among all approaches discussed in Section II, only ConfigMate [6] and Pragmatic Proof [8] had their implementations and datasets available. Our algorithm is compared with ConfigMate using the composition problems that do not rely on negation support. Experiments comparing with PragmaticProof employ an extension of the Web Services Challenge 2008 (WSC'08) synthetic benchmark [16] that includes preconditions and effects. All experiments[5] were run on an Intel i5 at 2.5GHz with 8GB of RAM.

A. *WSC'08*

The WSC'08 dataset by Bansal et al. [16] is often used as a benchmark for I/O-only service composition algorithms [10], [17], [11]. It comprises 8 synthetic problems, each consisting of a type taxonomy, a set of known inputs described by their types, a set of wanted outputs, also described by their types, and a service repository, with services described by their inputs and outputs. The goal of each problem is to chain the available services, according to I/O compatibility, to obtain the wanted outputs from the known inputs.

To include preconditions and effects, three new variants of the dataset were added. The first, called *2x I/Os*, serves as the basis for others and consists in shadowing all I/Os, preserving the problem structure. For every input or output variable x with type t of service s, using a renaming function η, the service s will receive an new I/O $\eta(x)$ with type $\eta(t)$. The type taxonomy is also shadowed: for any superclass u in the original taxonomy, $\eta(t)$ will be a subclass of $\eta(u)'$. The second variant, *Conjunction (single)*, is obtained adding a precondition $P(x_1, \eta(x_1)) \wedge \cdots \wedge P(x_n, \eta(x_n))$ to every service s with inputs (or outputs) $x_1, \ldots, x_n$, where P is a constant predicate. The third variant, *Conjunction (shadow)* is obtained in the same manner, but using a different predicate P_{x_i} for every input (or output) x_i.

The WSC'08 benchmark, as well as the extensions above, can also be expressed as problems in N3 Logic, making the Pragmatic Proof algorithm [8] applicable. Figure 4 compares the time required to plan a service composition using the EYE reasoner (as used by the Pragmatic Proof algorithm [8]) to the time required by the Unserved+SPARQL algorithm. Every combination of Scenario, problem and algorithm had the time measured 9 times. Furthermore, the time required to spawn the EYE process and to parse service is not included,

[5]Scripts and data at https://bitbucket.org/alexishuf/sac-2019-experiments/

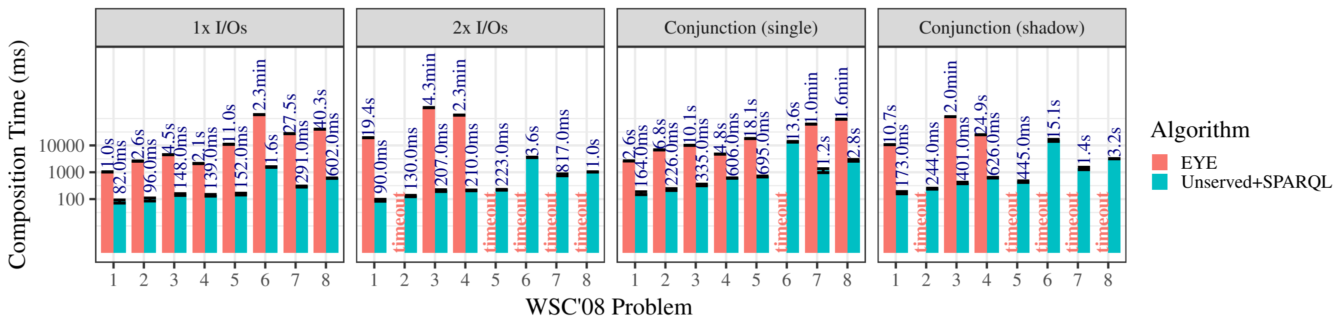

Figure 4: Composition Time in precondition-aware scenarios derived from WSC'08 for Unserved+SPARQL and for the EYE reasoner used in Pragmatic Proof [8].

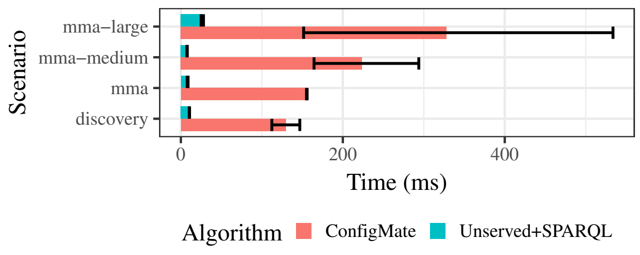

Figure 5: Composition time of ConfigMate (first solution) and Unserved+SPARQL.

since these steps are avoided in the Unserved+SPARQL implementation.

Figure 4 shows that unlike Unserved+SPARQL, for many scenarios, the EYE reasoner [8] is unable to find a solution within 5 minutes. A second important difference is that EYE is slower when the I/Os are shadowed (*2x I/Os*) than when predicates connect the inputs and the outputs (*Conjunction (single)*). Unserved+SPARQL is faster in all scenarios for all problems. The smallest speedup is of 7.85 for problem 4 in scenario *Conjunction (single)* and the largest is 1234.45 for problem 3 in scenario *2x I/Os*. Across all scenarios, the distribution of speedups has a long tail to the right: the median speedup is 39.76, with 5% of the problems below 12.53 and 5% above 642.42. The right tail is caused by large speedups for problems that yield large service chains from small repositories in the *2x I/Os* and *Conjunction (shadow)* scenarios, which were problematic for EYE. Nevertheless, speedups on these two scenarios are, respectively, 51.12 and 28.27 times on average.

B. *ConfigMate*

Among the problems present in the benchmark used in the proposal of the ConfigMate algorithm [6], only four could be successfully converted to Unserved descriptions. The other problems use features such as negation and rules, which are not supported by our proposed algorithm.

Figure 5 compares the original implementation of ConfigMate (without the graphical user interface) with the precondition-aware composition under Unserved for these four problems. The ComposIT algorithm alone is not able to handle these composition problems and, due to this limitation, it is not considered in this evaluation. Problem `discovery` contains 9 services without correspondence to a practical problem. Problems `mma`, `mma-medium` and `mma-large` model a travel ticket purchase scenario, varying in the number of services (2, 102 and 1002, respectively). For all problems, the Unserved implementation had better performance, with speedups ($\frac{T_{configmate}}{T_{unserved}}$), respectively, of 12.30, 18.51, 29.22 and 12.32.

VI. Conclusions

This paper presented a precondition-aware composition algorithm as an extension of the ComposIT-based algorithm implemented in the Unserved architecture. Preconditions and effects are modeled as SPARQL **`ASK`** queries. The planning algorithm provides support for conjunction (`.`) and predicates (`?x :p ?y`), while adaptive execution by Unserved achieves disjunction (**`UNION`**) support. We extended an I/O-only composition algorithm to create service compositions that achieve goals expressed with conjunction and disjunction of binary predicates. Earlier algorithms that achieve similar expressivity [6], [8], [9], [7] required more processing time or had no implementation able to handle large scenarios such as the WSC'08 benchmark.

When compared with state-of-the-art approaches handling preconditions and effects, this algorithm presented better scalability. In comparison with [6], speedups between 12.3 and 29.22 times were achieved for the problems that could be converted. In experiments using the WSC'08 benchmark, composition times remained tractable in all problems for all benchmark variants. In contrast, some WSC problems, when converted to N3 descriptions could not be solved under 5 minutes with the best reasoner used in [8]. Furthermore, the Unserved+SPARQL algorithm was faster in all problems of all scenarios, with speedups of at least 7.85.

A limitation of this work is that the flexibility provided by a complete logic framework, such as linear logic [18] or Answer Set Programming [19], is lost in favor of performance. A second limitation is that compensation of past service invocations during adaptation in the Unserved architecture is still not implemented. Two main research directions focused on the Unserved+SPARQL algorithm are the inclusion of negation and inclusion of arbitrary rules for inference during composition planning and execution. These two features, present in ConfigMate, enable tackling additional composition problems.

While this work improves the viability of precondition-aware automatic service composition, important questions remain unanswered. First, none of the algorithms evaluated generate constructs such as looping and recursion as part of the composition plans. Second, most research on service composition is evaluated using hand-crafted or mechanically generated composition problems. Of special interest are benchmarks with a stronger grounding in real use cases of composition. Another research opportunity is to use precondition/effects extraction techniques, such as [5], to improve the correctness of service composition given incomplete service descriptions.

ACKNOWLEDGEMENTS

This study was partially financed by the Brazilian research funding agencies CAPES and FAPESC.

REFERENCES

[1] Y.-Y. Fanjiang, Y. Syu, S.-P. Ma, and J.-Y. Kuo, "An overview and classification of service description approaches in automated service composition research," *IEEE Transactions on Services Computing*, vol. 10, no. 2, pp. 176–189, Mar. 2017.

[2] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, "Web services composition: A decade's overview," *Information Sciences*, vol. 280, pp. 218–238, october 2014.

[3] M. Klusch, P. Kapahnke, S. Schulte, F. Lecue, and A. Bernstein, "Semantic web service search: A brief survey," *KI - Künstliche Intelligenz*, vol. 30, no. 2, pp. 139–147, June 2016.

[4] M. L. Sbodio, D. Martin, and C. Moulin, "Discovering Semantic Web services using SPARQL and intelligent agents," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 8, no. 4, pp. 310–328, 2010.

[5] M. L. Mouhoub, D. Grigori, and M. Manouvrier, "Towards an automatic enrichment of semantic web services descriptions," in *On the Move to Meaningful Internet Systems. OTM 2017 Conferences: Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part I*. Cham, Switzerland: Springer-Verlag, 2017, pp. 681–697.

[6] F. Mohr, A. Jungmann, and H. K. Büning, "Automated Online Service Composition," in *2015 IEEE International Conference on Services Computing*. Washington, USA: IEEE, 2015, pp. 57–64.

[7] J. Alves, J. Marchi, R. Fileto, and M. A. R. Dantas, "Resilient composition of Web services through nondeterministic planning," in *2016 IEEE Symposium on Computers and Communication (ISCC)*. Washington, USA: IEEE, 2016, pp. 895–900.

[8] R. Verborgh, D. Arndt, S. Van Hoecke, J. De Roo, G. Mels, T. Steiner, and J. Gabarro, "The pragmatic proof: Hypermedia API composition and execution," *Theory and Practice of Logic Programming*, vol. 17, no. 1, pp. 1–48, 2017.

[9] D. Serrano, E. Stroulia, D. Lau, and T. Ng, "Linked REST APIs: A Middleware for Semantic REST API Integration," in *2017 IEEE International Conference on Web Services (ICWS)*. Washington, USA: IEEE, jun 2017, pp. 138–145.

[10] P. Rodriguez-Mier, C. Pedrinaci, M. Lama, and M. Mucientes, "An Integrated Semantic Web Service Discovery and Composition Framework," *IEEE Transactions on Services Computing*, vol. 9, no. 4, pp. 537–550, 2016.

[11] S. Chattopadhyay, A. Banerjee, and N. Banerjee, "A Scalable and Approximate Mechanism for Web Service Composition," in *2015 IEEE International Conference on Web Services*. Washington, USA: IEEE, 2015, pp. 9–16.

[12] A. Huf, I. Salvadori, and F. Siqueira, "Planning and execution of heterogeneous service compositions," in *2017 IEEE Symposium on Computers and Communications (ISCC)*. Washington, USA: IEEE, Jul. 2017, pp. 987–993.

[13] T. Berners-Lee, D. Connolly, L. Kagal, Y. Scharf, and J. Hendler, "N3Logic: A logical framework for the World Wide Web," *Theory and Practice of Logic Programming*, vol. 8, no. 03, pp. 249–269, 2008.

[14] M. Wylot, M. Hauswirth, P. Cudré-Mauroux, and S. Sakr, "Rdf data storage and query processing schemes: A survey," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 84:1–84:36, Sep. 2018.

[15] R. T. Fielding and R. N. Taylor, "Principled Design of the Modern Web Architecture," *ACM Transactions Internet Technology*, vol. 2, no. 2, pp. 115–150, 2002.

[16] A. Bansal, M. B. Blake, S. Kona, S. Bleul, T. Weise, and M. C. Jaeger, "WSC-08: Continuing the Web Services Challenge," in *2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*. Washington, USA: IEEE, 2008, pp. 351–354.

[17] A. S. da Silva, Y. Mei, H. Ma, and M. Zhang, "Evolutionary computation for automatic web service composition: an indirect representation approach," *Journal of Heuristics*, vol. 24, no. 3, pp. 425–456, Jun 2018.

[18] X. Zhao, E. Liu, G. J. Clapworthy, N. Ye, and Y. Lu, "RESTful web service composition: Extracting a process model from Linear Logic theorem proving," in *2011 7th International Conference on Next Generation Web Services Practices*. Washington, USA: IEEE, october 2011, pp. 398–403.

[19] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits, "Combining answer set programming with description logics for the semantic web," *Artificial Intelligence*, vol. 172, no. 12, pp. 1495 – 1539, 2008.