Planning and Execution of Heterogeneous Service Compositions

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

***Abstract*—Current Web-based Services are highly heterogeneous not only on data but also with regard to service interaction. Despite their heterogeneity, composition of these services is required in order to achieve additional functionality. Semantic descriptions and composition algorithms for heterogeneous services have been recently proposed. However, existing techniques do not take the Publish/Subscribe paradigm in consideration or do not offer sufficient support for interaction through hypermedia controls as required in the REST architectural style. This paper presents a composition architecture and two techniques, re-planning, and replication, for support of REST and Publish/Subscribe services by composition algorithms. We apply these techniques to a state of the art graph-based composition algorithm and evaluate the impact on performance.**

***Keywords*-service composition; REST; publish-subscribe; heterogeneous systems.**

I. Introduction

With respect to Web Services implementations, there is a technological and architectural division between services based on the SOAP (Simple Object Access Protocol) and those that employ the REST (REpresentational State Transfer) architectural style [1], even if partially. Analyzing keywords, as of August 2016 [1], only 10.03% of services registered on the ProgrammableWeb[2] directory reference SOAP in their descriptions. In contrast, 73.63% of service descriptions qualify as HTTP based services, a key characteristic of REST services (there is no data refering to actual adoption of REST constraints by the services). For the remaining descriptions, 3.28%, belong to both groups, 7.34% to none, and 5.72% have insufficient data.

The constraints defined by the REST architectural style are aimed at achieving architectural benefits. Therefore, some constraints apply not only to the design of services, but also to the behavior of clients and intermediary elements, such as caches and proxies. One of such constraints is HATEOAS (Hypertext As The Engine Of Application State). It requires the service to expose possible actions as hypermedia controls, which the client must take into consideration, as opposed to simply executing a static workflow. Failure to employ this constraint, both at server or client, will avoid independent evolution of client and server software and can potentially lead the client to encounter faults or to ignore data.

Additionally, while SOAP services are highly influenced by request-response interactions, and REST is based on this communication pattern, ESBs (Enterprise Service Buses) and many publicly-available services (e.g. Google Calendar, Twitter) support more complex interactions such as the Publish/Subscribe communication paradigm [2]. In addition to *ad hoc* implementations, such support may be backed by some application-independent protocol specification (e.g. Atom Publishing Protocol[3]), but there is no definitive specification for Publish/Subscribe services.

The integration of SOAP and REST services of varying maturity levels has been tackled by researchers in several forms. Representative examples are mediators [3], extended process engines [4] and the use of a common description language [5], [6]. Publish/Subscribe services are often considered in isolation, but [7] proposes integration with SOAP services in an ESB. Simultaneous integration of SOAP, REST and Publish/Subscribe services remains an open issue.

Automated service composition algorithms are one form of service integration. Given a set of service descriptions, an initial and goal states, the composition algorithm tries to orchestrate a subset of the services in order to achieve the goal state. During orchestration, constraints imposed by services that are part of this composition, such as HATEOAS, must be respected. Verborgh et al. [8] propose a composition and execution algorithm specifically for REST services supporting the HATEOAS constraint. In addition, recent SOAP composition algorithms have achieved high performance marks [9]. On the other hand, Publish/Subscribe services are treated in complete isolation with techniques such as CEP (Complex Event Processing) [10]. The goal of this paper is to create a single composition algorithm, able to compose SOAP, REST and Publish/Subscribe services.

In summary, this paper offers as contribution a composition architecture (featuring an intermediate description) and two techniques to adapt a graph-based composition algorithm into one that supports both HATEOAS and Publish/Subscribe services. Current composition algorithms and architectures do not support both simultaneously. In addition, architectures for Publish/Subscribe systems do not cover

[1]https://morph.io/IvanGoncharov/ProgrammableWeb
[2]http://www.programmableweb.com/
[3]https://tools.ietf.org/html/rfc5023

 DOI: 10.1109/iscc.2017.8024654

automated service composition, and HATEOAS is fully supported by only one algorithm [8]. We adapt a SOAP-oriented composition algorithm [9], replacing A* with D* Lite [11], and evaluate a prototype of our architecture using the the Web Services Challenge 2008 (WSC'08) [12] dataset and the benchmark in [8]. Despite added complexity of the platform, our prototype was faster for the majority of cases in both benchmarks, demonstrating the feasibility of our architecture in supporting HATEOAS for non time-critical applications.

The remainder of this paper is structured as follows. Related Work is discussed in section II. The intermediate description and the composition architecture are presented in section III and section IV. The composition algorithm is presented in section V, with experiments discussed in section VI and concluding remarks in section VII.

II. Related Work

The problem of integrating SOAP services and REST services has been handled by research literature in the last decade. Proposed solutions involve mediators [3], extended process engines [4], or a common service description. In the last group, the MSM (Minimum Service Model) [5] is proposed to integrate discovery and composition of both SOAP and REST services. The MSM is a generalization of WSDL (Web Service Description Language) concepts, to which REST interaction concepts are mapped: Groups of resources are mapped to a Service, resource operations to an operation and I/O representations are mapped to messages.

In addition to the lack of support for Publish/Subscribe, MSM has two other issues. First, it delegates most details of message structure to other description languages, such as hRESTS [6], which is problematic for REST services due to the large number of such languages. Second, while it is suggested that hypermedia controls be transported in message data and extracted by an implementation that is aware of them [6], a HATEOAS conformant composition must consider hypermedia controls in planning and change the plan according to the available controls [8].

Hypermedia controls can be seen as documentation of state transitions within a REST service, as per the HATEOAS constraint [1]. Therefore, composition of REST services must be driven by hypermedia. However, there is no guarantee that hypermedia controls will be offered for a particular resource. Verborgh et al. [8] propose a proof-guided, hypermedia-driven algorithm that optimistically assumes the presence of hypermedia controls and,after every request, verifies their presence and adapts the composition.

Other composition algorithms also have taken service faults into account. Alves et al. [13] applies non-deterministic planning to create workflows with contingency plans. Although planning for failures beforehand saves execution time, the generation of all possible alternative plans is a time-consuming process. In contrast, if there are multiple services under frequent changes, dynamic fault tolerance may be more appropriate.

Several service composition approaches are based on planning techniques [9], [14], [15]. One of such is ComposiIT [9], which reduces the problem of service composition to graph path-finding, modeling each vertex of the path as a combination of services. Employing optimizations for service and state size reduction, the algorithm presented high performance. However, the algorithm is not able to handle HATEOAS, nor Publish/Subscribe.

With respect to Publish/Subscribe services, the work of Georgantas et al. [7] tackles the issue of multiple communication paradigms [2] through an abstract Enterprise Service Bus. Adaptation of multiple communication paradigms is based on the notion of send an receive as common primitives. However, this is not sufficient to support HATEOAS.

III. Intermediate Descriptions

Many differences among service interaction models can be solved through the use of a description that abstracts the differences. Considering popularity and versatility of RDF[4] (Resource Description Framework) in exposing data on the web, we adopt the RDFS [5] (RDF Schema) vocabulary to define an intermediate description vocabulary. These intermediate descriptions are automatically produced by translators from existing service descriptions, such as SAWSDL[6] (Semantic Annotations for WSDL), Atom Service documents and the many description languages for REST services. During this process, translators can include in the descriptions references to other RDF vocabularies (e.g. HTTP[7] that are used during execution.

The two main concepts of the vocabulary are *Message* and *Variable*. Messages represent the high-level messages exchanged during a service invocation, e.g. request and response, or subscription and notifications. Messages contain technical information that pertains to the protocols involved and parts in the form of *Variable* instances. The vocabulary allows a Variable to be *part* of another, and to be *part* of something but be extracted from another location that is not the parents' location

A *Variable* may have a type, a representation and a value. The type of a variable is assumed to be a semantic class of which a Variable value is instance of, and representation denotes the format of the value. This distinction is necessary specially in REST services, in which a Resource of a semantic type may be represented by a URI, XML document or RDF, among others.

Figure 1 shows a fragment of a response message converted from an Hydra [16] description. The message is also a `http:Response`, from the HTTP vocabulary, and represents

[4]http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/
[5]http://www.w3.org/TR/2014/REC-rdf-schema-20140225/
[6]http://www.w3.org/TR/2007/REC-sawsdl-20070828/
[7]https://www.w3.org/TR/HTTP-in-RDF/

```
1  _:response a http:Response, u:Message;
2    u:part [ u:variable _:user;
3      u:partModifier [ a ux-http:PartModifier;
4        ux-http:httpProperty http:body;
5        ux-http:httpResource _:response ] ];
6    u:when [ u:reactionTo _:request;
7           u:cardinality u:one ].
```

Figure 1. Example intermediate description for a REST response message

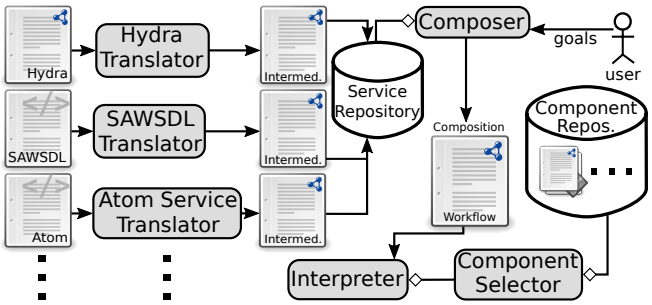

Figure 2. Overview of the service composition and execution architecture

the service output as an `_:user` variable, contained in the response body. The fact that `_:response` is a response is expressed conditioning its receipt to a previous message, `_:request` (lines 6-7 in Figure 1). Since cardinality is `u:one`, the response message will be received only once. In case of notification messages, cardinality would be `u:many`.

IV. Heterogeneous Composition Architecture

Figure 2 summarizes the proposed composition architecture where service descriptions are converted to an intermediate language, used by a composition algorithm to output interpretable workflows. Abstraction via intermediate description is achieved by *Translators*. The *Service Repository* indexes and manages the intermediate descriptions. The *Composer* outputs workflows from the intermediate descriptions and a user goal. The Workflows are executed by the *Interpreter*, which uses components selected by the *Component Selector* to handle remaining heterogeneities. Component selection uses specific details present in the intermediate description (which were ignored by the *Composer*) to select the components required to interact with the services (e.g., a SOAP client or a plain HTTP client). The *Composer* itself may also request components to the *Component Selector*, although such usage is specific to the composition algorithm.

The components stored in the *Component Repository* implement all tasks involved in sending and receiving messages. To send a *Message*, headers and a payload must be assembled from the values of the message variables, and then the message is sent using the relevant transport protocol. When receiving a *Message*, values are extracted from the actual message and bound to the *Message* variables. The assembly of messages, and implementation of transport protocols is responsibility of the *ActionRunner* class of components. The mapping between *Variable* values in RDF and actual representations transmitted in messages is done by components of classes *Renderer* and *Parser*. Component selection is determined by the specificities documented in the *Messsage* and *Variable* instances, such as third-party vocabularies (`http:Response` in Figure 1).

Component selection, performed by the *Component Selector*, consists in, given a tuple of RDF resources, $(r_1, \ldots, r_n)$, finding a component description x s.t. $r_1 \,\mathtt{a}\, c_1(x) \wedge \cdots \wedge r_n \,\mathtt{a}\, c_n(x)$. The c_n functions denote OWL[8] (Web Ontology Language) class expressions that x describes as supported for each of the r_n resources provided by the *Interpreter* or *Composer*. Action runners have only one of these classes, denoted with the `actionClass` property. Renderers and parsers, on the other hand, have two, `representationClass` and `valueClass`, which respectively specify the Variable's representation class and the value class (or datatype URI, in case of a literal value). The use of OWL class expressions in $c_1, \ldots, c_n$ allows fine-grained control of the range of resources supported by the component.

V. Replicated and Adaptive Composition

This section presents two techniques applied to a composition algorithm. The first, Replication, provides basic support for composition of Publish/Subscribe services. All composition and execution state is stored on an object that can be replicated and used to spawn independent executions for each notification message of Publish/Subscribe services. For example, consider a fictitious scientific publisher system. As researchers submit articles to journals, events are generated on a Publish/Subscribe service. Replication allows a composition algorithm to be planned so that for each submission (event), a service composition for assigning at least three reviewers is planned and executed. The second approach, Adaptation, adds support for the HATEOAS constraint. The composition plan is re-evaluated against the actual hypermedia controls, after every service interaction. Adaptation requires a graph-based composition algorithm, which operates on a *state graph* that has the following characteristics:

1) States contain a set (or singleton) of services;
2) The (u_1, u_2) edge corresponds to invoking u_1 services, achieving the necessary conditions so that u_2 services may be later invoked;
3) There are lists of alternatives to services and states.

An overview of a composition using an algorithm with these techniques is depicted in Figure 3. Upon client request, the Composer creates an execution object from the composition problem with all necessary data structures (including the state graph) for planning the composition. A plan is

[8]http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/

 DOI: 10.1109/iscc.2017.8024654

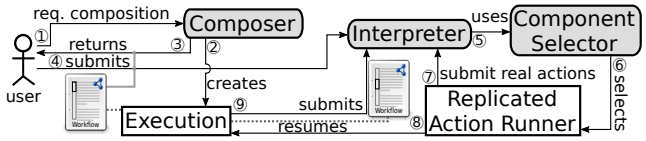

Figure 3. Overview of an adaptive composition execution.

obtained from the state graph, as a list of state graph edges.Each edge originates a sequence of actions executable by the architecture (copy, send, receive), and this sequence is further divided into *slices*. A slice contains at most one receive action, that if present, is the last action.

The descriptions of all services, and other elements of the problem (e.g., known inputs and wanted outputs), are stored on a single RDF graph that serves as working memory of the composition. Service inputs and outputs are present in the graph as unbound variables, which will be bound to values as Action Runners (described in section IV) that execute the planned actions. After every received message, results must be confronted with expectations so that the plan is adapted, if necessary. Therefore, only the first slice must be executed before control returns to the composition algorithm. The composer returns a workflow with only the first slice, wrapped with a Replicated Action, and with a pointer back to the execution object. When interpreting this action, as depicted in Figure 3, the Interpreter will use the Component Selector to select the Replicated Action Runner, which will return control to the composition algorithm for validation, adaptation and execution of the next slice.

The runner selected by the Component Selector for a replicated action works as shown in Figure 4. First, all actions except the receive (r) are executed, producing side effects on the execution's context (line 2). If there is a receive and it will result in multiple messages (line 5), each one is received into a replicated execution (line 8) and resumed in another thread (line 10). Single-shot messages are received into the current context and resumed directly (lines 13-15).

The RESUME procedure of the execution object displayed in Figure 5 closes the cycle, validating the results against the current plan of the execution object (e). If the edge of the state graph had its last slice executed, the effects on $context(e)$ are validated against the executed edge (line 4), resulting in a set of failed services. If valid, the execution object is advanced to the next edge of the composition path. Otherwise, the state graph is adapted in order re-plan the composition (lines 7-8). If the last slice was not the last one for the current edge ($remainingEdgeActions(e) \neq \emptyset$), the composition is simply advanced to the next slice (line 10) When ADVANCEEDGE meets the end of the path, it sets the $isEnded$ flag which causes the composition execution to deliver a result and finish (lines 12-13). If there is still work to do, the current slice is wrapped, and executed with the current interpreter (line 4 of Figure 4).

```
1:  procedure RunWrapper(w, c)
2:      r ← ExecuteUntilReceive(w, context(e))
3:      e ← execution(w)
4:      SetInterperter(e, interpreter(c))
5:      if r ≠ null ∧ cardinality(r) ≠ one then
6:          for all 1 ≤ i ≤ cardinality(r) do
7:              e′ ← Replicate(e)
8:              Run(interpreter(c), r, context(e′))
9:              begin thread
10:                 Resume(e′)
11:             end thread
12:     else
13:         if r ≠ null then
14:             Run(interpreter(c), r, context(e))
15:         Resume(e)
```

Figure 4. Replicated Action Runner implementation

```
1:  procedure Resume(e)
2:      if remainingEdgeActions(e) = ∅ then
3:          F ← Validate(e, currEdge(e))
4:          if F = ∅ then
5:              AdvanceEdge(e)
6:          else
7:              Adapt(e, F)
8:              Plan(e)
9:      else
10:         AdvanceSlice(e)
11:     if isEnded(e) then
12:         r ← CreateResult(e)
13:         DeliverResult(interpreter(e), r)
14:     c ← WrapAsReplicated(e, currSlice(e))
15:     Run(interpreter(e), c, context(e))
```

Figure 5. Resume Execution algorithm.

The procedure for adapting the state graph upon edge failure is shown in Figure 6, where G stands for the state graph, (u_1, u_2) is the failed edge, and $F \subseteq u_1$ is the set of failed services. A state a is constructed to replace u_1, without including services from F. Each failed service has a list of alternatives, denoted by $alts(f)$. Since not all alternatives are equivalent, a helper function $\text{COMPAT}(alts(f), u_1)$ filters $alts(f)$ to only services that can be invoked using the already bound inputs (and other state) of u_1.

There are three possible constitutions of the alternative state a. First, if all failed services in u_1 have invocable alternatives, they will constitute a (line 6). Second (lines 7-13), if there is suitable alternative state u_a to u_1 which does not include any node from $\hat{F}_{u_1}$. $\hat{F}_{u_1}$ is the union of all sets F for u_1 and all previous states (alternatives and the original) that were considered in this position of the state graph. Third, a will be an empty state distinct from any

 DOI: 10.1109/iscc.2017.8024654

```
1: procedure ADAPT(G, (u_1, u_2), F)
2:     I ← {(u_0, u_1) s.t. (u_0, u_1) ∈ E(G)}
3:     O ← {(u_1, u_3) s.t. (u_1, u_3) ∈ E(G)}
4:     l ← ∞
5:     minimize l
6:         a ← {chs(COMPAT(alts(f), u_1)) s.t. f ∈ F}
7:         if null ∈ a then
8:             a ← null
9:             for all u_a ∈ alts(u_1) s.t. u_a ∩ F̂_{u_1} = ∅ do
10:                a ← COMPAT(u_a, u_1)
11:                if a ≠ null ∧ |a| = |u_a| then
12:                    break
13:            a ← null
14:        if a = null then
15:            a ← {CREATEUNIQUEEMPTYNODE(G)}
16:        else
17:            ASSIGNINPUTS(a, u_1)
18:            l ← ADAPTEDGES(a, u_1, I)
19:            l ← l + ADAPTEDGES(a, u_1, O)
20:    end minimize
21:    CREATENOOP({(a, u_0) s.t. (u_0, u_1) ∈ I})
22:    REMOVENODE(u_1)
```

Figure 6. Algorithm for graph adaptation

other empty state already in G (line 15).

Once a has been computed the inputs of its services are bound with values already present for inputs of u_1 (line 17). The incoming (I) and outgoing edges (O) of u_1 originate edges to and from a (lines 18-19), this time adapting the assignments so that a's inputs and outputs are used in place of those of u_1. ADAPTEDGES also verifies any additional criteria that applies to edges of the state graph[9]. Depending on the alternative chosen by chs, more or less edges may be lost (l). For brevity, we omit exploration of these alternatives with a **minimize** block (line 5) to symbolize that only the side effects of the iteration that yielded the smallest l remain. An efficient implementation of this scheme relies on specific knowledge of the alternatives list properties.

A. Adapting a graph-based composition algorithm

We apply the aforementioned techniques to the composition algorithm proposed by Rodriguez-Mier et al. [9] within the composition architecture of section IV. In this algorithm, a composition problem consists of a set of services, a set of known inputs (bound variables in the intermediate description) and wanted outputs (unbound variables).

When processing a composition request, the first data structure to be created is the dependency graph. An example of such graph is shown in Figure 7, where variables v_1, v_2, v_3 are known inputs and v_{15} is the wanted output. Each node

[9]For example, if edges are determined by pre-/post-conditions, the satisfaction of pre-conditions must be rechecked during edge adaptation

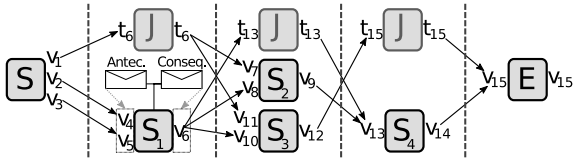

Figure 7. An example Service Dependency Graph.

```
1: function VALIDATE(e, (u_1, u_2))
2:     F ← ∅                                  ▷ set of failed services
3:     for all a ∈ actions((u_1, u_2)) s.t. a : Copy ∧
       target(a) ∈ I(u_1) do
4:         if ¬isBound(target(a)) then
5:             r ← FINDREPLACEMENT(target(a), u_1)
6:             if r ≠ null then
7:                 COPY(r, target(a))
8:             else
9:                 F ← F ∪ {provider(source(a))}
10:    REVERTASSIGNMENTS(F)
11:    return F
```

Figure 8. Algorithm for execution validation.

S_i in this graph corresponds to a service, which has a pair of antecedent and consequent messages. Variables part of the antecedent form the inputs of the node and those part of the consequent, the outputs. Edges connect the output variables in a layer with compatible inputs of the next layer: an output o with type t_o and representation r_o matches an input of type t_i and representation r_i if, and only if, $t_o \sqsubseteq t_i \wedge r_o = r_i$. Special jump nodes (J) ensure that only nodes of neighboring layers are connected. However, it is more efficient to create jump nodes on demand while exploring the state graph.

If two nodes s_1 and s_2 have the same predecessors, but s_2 successors are a subset of those of s_1, s_2 is said to dominate s_2 and s_1 can be removed. If instead, their successors sets are equal, only one of them must remain in the dependency graph. These removals, named *Offline Service Compression* [9], have no effect on the ability to find the optimal composition path. However, at runtime, the selected service may be unavailable, it might not contain an hypermedia control, or it may miss some relevant output. A topological sort of the dominates relation is used to identify a representative node and compile a sorted list of its $alternatives$ allowing the implementation to forgo the **minimize** block in Figure 6. Additionally, as all alternatives have the same predecessors, COMPAT is always true.

The state graph for this composition algorithm is also a layered graph, where a state is a combination of services that share the same layer on the dependency graph. An edge (u_1, u_2) exists between two states if, and only if, all outputs of u_2 can be assigned from outputs of u_1. The graph is materialized on demand during backward search, and the *Online Node Reduction* optimization [9]

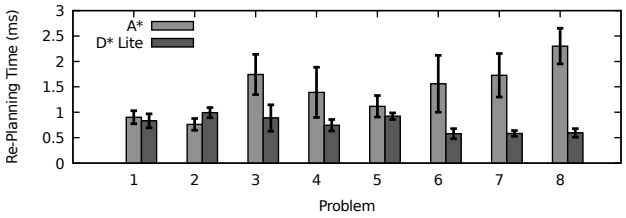

Figure 9. Re-planning due to induced fault on first invoked service.

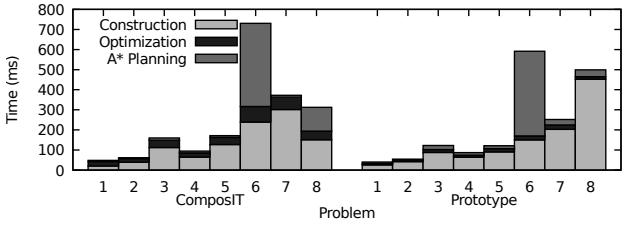

Figure 10. Time decomposition for ComposIT and the Prototype.

detects and removes equivalent states. As in *Offline Service Compression*, removed states are kept as $alternatives$ of the representative state, but no sorting is required.

Validation of an execution context against an edge (u_1, u_2), shown in Figure 8, checks for the inputs of u_2 that were targets of a copy action but remained unbound (lines 3-4). For any unbound input, the algorithm attempts to find a replacement source (lines 5-7), or otherwise adds the original provider service to the set F of failed services. Finally, assignments from failed services are undone (line 10).

VI. Experiments

A proof of concept implementation[10] of the architecture (section IV) and the results[11] shown here are available online. OpenJDK 1.8 was used, with Apache Jena 3.1.1 for RDF processing, and Hipster [17] for the A* implementation. D* Lite [11] was implemented based on the original paper, with an optimization to keep links to the successors of a node that determined its rhs-value. All experiments were performed in random order with 9 replications on a 2.5GHz Intel i5 running Linux with a maximum JVM heap of 4 GB.

As in [9], the WSC'08 dataset [12] was used to compare the performance of the adapted composition algorithm. To analyze how failures impact performance, a failure on the first invoked service for each one of the eight WSC'08 problems was induced. This resulted in a planning phase (where the state graph was partially materialized) and a subsequent re-planning. Figure 9 shows the re-planning times of A* and D* Lite. These times do not include the validation and adaptation procedures, which are performed independently from the search algorithms. D* Lite re-planning averages range

[10]https://bitbucket.org/alexishuf/unserved-testbench/

[11]https://bitbucket.org/alexishuf/unserved-testbench-iscc2017/

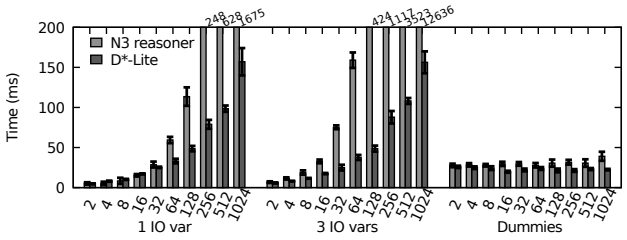

Figure 11. N3 reasoner time (Pragmatic Proof) versus D*-Lite (Prototype).

from 0.578 (problem 6) to 0.993 (problem 2), in the best case (problem 6) requiring only 37% of the time of a full A* replan. Problems 1, 2 and 5 presented overlapping confidence intervals, indicating no conclusive advantage for D* Lite. These problems have short composition paths ($l = 4, 4, 6$) and simple state graphs ($states/edges = 5/4, 5/4, 10/9$). In contrast, problem 3 is long ($l = 24$, $s/e = 25/24$), 6 and 7 are complex ($l = 8, 13$, $s/e = 17/15, 23/19$), and problem 8 is both long and slightly complex ($l = 21$, $s/e = 51/57$).

Figure 10 shows a breakdown of times for both our prototype and ComposIT [9]. Adding the three phases, averages for our prototype ranged from 68% to 92% of those from ComposIT, except problem 8 which had an overhead of 59% with its 8120 services that imposed high memory usage (due to the more verbose intermediate description) and increased graph construction by 199%. For other problems, memory was not an issue and the overhead of the intermediate description and executable state graph was surpassed by improvements on construction and optimization phases. Averages for construction, mainly due to computing closures of superclasses instead of a match table, ranged from 63% to 78% of ComposIT averages for half of the problems (in addition to 8, overheads between 1% and 27% were observed for problems 1,2 and 4). On Offline Service Compression, storing forward adjacencies on the dependency graph caused the averages to be between 26% and 45% of ComposIT.

Finally the Prototype was evaluated with the benchmark used by Pragmatic Proof (PP) [8], the only related work to support HATEOAS, and the results are displayed in Figure 11. To make the comparison against PP fair, we discounted the N3 reasoner (EYE) spawn time (36.32 ms in average), parsing of descriptions and the N3 proof conversion into a workflow. For the main scenarios described in [8], with 1 and 3 I/O variables, the 95% confidence interval ceases overlapping when chain length surpasses 64 for one variable and 4 for 3 variables. On the dummies scenario, in which the actual composition has a chain length of 32 with one I/O variable, the intersection occurs for chains smaller than 16 or equal to 64. In addition to shorter absolute times, our prototype is less sensitive to number of I/O variables than PP. For PP, taking l as the chain length and t_i as the reasoning time with i I/Os, $log_2(t_3 - t_1) = (1.63 \pm 0.12) log_2(l)^2 + (7.28 \pm 0.12) log_2(l) + 1 \pm 0.01$ with

95% confidence. No similar linear or quadratic model fits the data from our prototype, as experimental error dominates.

VII. Conclusion

In service oriented computing, research on automated service composition is often restricted to only SOAP, only REST or only Publish/Subscribe services. While there are proposals for integration of SOAP and REST [3]–[6] and for Publish/Subscribe and SOAP [7], the simultaneous integration of all three remained an open issue. We proposed a composition architecture and an adaptation technique to support these interaction models. The results show that the prototype was faster for 7 out of 8 scenarios when compared with ComposIT [9], and that absolute values are tolerable for most non time-critical applications. When comparing against [8], we observed large speed improvements on all scenarios and asserted that our prototype is less sensitive to the number of I/O variables. The re-planning time improvement also hints at promising performance in long-running mixed Publish/Subscribe and HATEOAS compositions.

The re-planning approach only applies to graph-based composition algorithms, and the gains obtained through D* Lite require an algorithm that reduces service composition to path-finding. Another limitation is that no compensation actions are applied if the execution backtracks. Finally, the replication algorithm spawns totally isolated execution threads that have no means of communication. Any composition that would require communication between replicated threads must have the requirements split in two parts, automatically composed, but with join code manually written to start the second composition from results of the first.

In future work we intend to add the ability of documenting the compensation needs of services to the intermediate language and to the algorithm, so that backtracked sub-paths can be compensated, if needed. Another related possibility is to model application state and consider action semantics during service composition. One challenging scenario for actions and state is device control in the Internet of Things, in which physical state is of key concern to applications.

References

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

[2] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[3] R. Battle and E. Benson, "Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST)," *Web Semantics: Sci., Services and Agents on the World Wide Web*, vol. 6, no. 1, pp. 61–69, Feb. 2008.

[4] J. Lee, S.-j. Lee, and P.-f. Wang, "A Framework for Composing SOAP, Non-SOAP and Non-Web Services," *IEEE Transactions on Services Comput.*, vol. 8, no. 2, pp. 240–250, 2015.

[5] C. Pedrinaci, D. Liu, M. Maleshkova, D. Lambert, J. Kopecky, and J. Domingue, "iServe: a linked services publishing platform," in *Proc. of the 1st Workshop on Ontology Repositories and Editors for the Semantic Web*, vol. 596. CEUR Workshop Proc., 2010.

[6] D. Roman, J. Kopecký, T. Vitvar, J. Domingue, and D. Fensel, "WSMO-Lite and hRESTS: Lightweight semantic annotations for Web services and RESTful APIs," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 31, pp. 39–58, 2015.

[7] N. Georgantas, G. Bouloukakis, S. Beauche, and V. Issarny, "Service-oriented distributed applications in the future internet: The case for interaction paradigm interoperability," in *Service-Oriented and Cloud Comput., 2nd European Conf. on*, vol. 8135 LNCS, 2013, pp. 134–148.

[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] P. Rodriguez-Mier, M. Mucientes, J. C. Vidal, and M. Lama, "An Optimal and Complete Algorithm for Automatic Web Service Composition," *Intl. Journal of Web Services Research*, vol. 9, no. 2, pp. 1–20, 2012.

[10] M. Potocnik and M. B. Juric, "Towards Complex Event Aware Services as Part of SOA," *IEEE Transactions on Services Comput.*, vol. 7, no. 3, pp. 486–500, 2014.

[11] S. Koenig and M. Likhachev, "Improved fast replanning for robot navigation in unknown terrain," in *Robotics and Automation, 2002. Proc.. ICRA '02. IEEE Intl. Conf. on*, vol. 1, 2002, pp. 968–975 vol.1.

[12] 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 Conf. on E-Commerce Technol. and the Fifth IEEE Conf. on Enterprise Comput., E-Commerce and E-Services*, 2008, pp. 351–354.

[13] J. Alves, J. Marchi, R. Fileto, and M. A. R. Dantas, "Resilient composition of Web services through nondeterministic planning," in *2016 IEEE Symp. on Computers and Comm. (ISCC)*, 2016, pp. 895–900.

[14] M. Klusch and A. Gerber, "Semantic web service composition planning with owls-xplan," in *Proc. of the 1st Int. AAAI Fall Symp. on Agents and the Semantic Web*, 2005, pp. 55–62.

[15] G. C. Hobold and F. Siqueira, "Discovery of semantic web services compositions based on sawsdl annotations," in *2012 IEEE 19th Intl. Conf. on Web Services*, 2012, pp. 280–287.

[16] M. Lanthaler and C. Gütl, "Hydra: A Vocabulary for Hypermedia-Driven Web APIs," in *Proc. of the 6th Workshop on Linked Data on the Web (LDOW2013) at the 22nd Intl. World Wide Web Conf. (WWW2013)*. CEUR-WS, 2013.

[17] P. Rodriguez-Mier, A. Gonzalez-Sieira, M. Mucientes, M. Lama, and A. Bugarin, "Hipster: An open source Java library for heuristic search," in *2014 9th Iberian Conf. on Inform. Systems and Technologies (CISTI)*, 2014, pp. 1–6.