Microflows: Automated Planning and Enactment of Dynamic Workflows Comprising Semantically-Annotated Microservices

Roy Oberhauser

Computer Science Department, Aalen University, Aalen, Germany
roy.oberhauser@hs-aalen.de

Abstract. Businesses are under increasing pressure to quickly and flexibly adapt their business processes to external and internal software and other changes. Furthermore, to address the rapid change and deployment of software functionality, microservices have emerged as a popular architectural style for partitioning business logic into small services accessible with lightweight mechanisms, resulting in a higher degree of dynamic integration of information services with processes. Current process-aware information systems tend to rely on manually pre-configured static process models and during process enactment exhibit challenges in reacting to unforeseen dynamic changes. This paper presents Microflows, an automatic lightweight declarative approach for the workflow-centric orchestration of semantically-annotated microservices using agent-based clients, graph-based methods, and the lightweight semantic vocabularies JSON-LD and Hydra. A case study shows approach's advantages for automating workflow modeling and enactment in a dynamic microservice environment.

Keywords: Workflow Management Systems, Microservices, Service Orchestration, Agent Systems, Semantic Technology.

1 Introduction

A trend towards increased automation can be observed in many areas of society today. One area affecting organizations and businesses in particular is the area of business processes or workflows. The automation of a business process according to a set of procedural rules is known as a workflow (WfMC, 1999). In turn, a workflow management system (WfMS) defines, creates, and manages the execution of workflows (WfMC, 1999). These workflows are often rigid, and while adaptive WfMS can handle certain adaptations, they usually involve manually intervention to determine the appropriate adaptation. As one indicator of its importance to business, spending on Business Process Management Systems (BPMS) is forecast at $2.7 billion in 2015 (Gartner, 2015).

Moreover, there is an increasing trend toward applying the microservice architecture style (Fowler & Lewis, 2014) for an agile and loosely coupled partitioning of business logic into small services accessible with lightweight

mechanisms. They can be deployed independently of each other and conform to a bounded context. As the dynamicity of the service world grows, the need for more automated and dynamic approaches to service orchestration becomes evident.

Service orchestration represents a single executable process that uses a flow description (such as WS-BPEL) to coordinate service interaction orchestrated from a single endpoint, whereas service choreography involves a decentralized collaborative interaction of services (Bouguettaya, Sheng, & Daniel, 2014), while service composition involves the static or dynamic aggregation and binding of services into some abstract composite process.

While automated and dynamic workflow planning can remove the manual overhead for workflow modeling, a fully automated semantic integration process remains challenging, with one study indicating that only 11% of Semantic Web applications achieve it (Heitmann, Cyganiak, Hayes, & Decker, 2012). Rather than pursuing the fairly heavyweight service-oriented architecture (SOA) and semantic web standards, we chose to investigate the viability of a lightweight approach. Analogous to microservices principles, we use the term Microflow to mean lightweight workflow planning and enactment of microservices, i.e. a lightweight service orchestration of microservices.

This paper explores an approach we call Microflows for automatically planning and enacting lightweight dynamic workflows of semantically annotated microservices. It uses a declarative paradigm with cognitive agents leveraging current lightweight semantic and microservice technology and investigates its viability. It extends (Oberhauser, 2016) and contributes more detail on its branching and error handling capabilities. Note that this approach does not intend to address all facets of BPMS support, but is focused on a narrow area addressing the automatic orchestration of dynamic workflows given a multitude of microservices using a pragmatic lightweight approach rather than a theoretical treatise.

This paper is organized as follows: the next section discusses related work. Section 3 and 4 describe the solution approach and its realization respectively. Section 5 describes the evaluation, followed by the conclusion.

2 Related Work

While the term Microflow has been used in IBM business process manager documentation to mean a transient non-interruptible BPEL process (IBM, 2015), in our terminology a Microflow is independent of any specific BPMS or any choreography or orchestration language.

Work related to the orchestration of microservices includes (Rajasekar, Wan, Moore, & Schroeder, 2012), who describe the integrated Rule Oriented Data System (iRODS) for large-scale data management, which uses a distributed event-condition-action rule engine to orchestrate micro-services into conditional chain-oriented workflows, maintaining transactional properties through recovery micro-services. (Alpers, Becker, Oberweis, & Schuster, 2015) describe a microservice architecture for BPM tools, highlighting a Petri Net editor to support humans with BPM.

As to web service composition, (Sheng et al., 2014) provides a survey of current research prototypes and standards in the area of web service composition. While the web service composition using the workflow technique (Rao & Su, 2004) can be viewed having similarity to ours, our approach does not explicitly create an abstract composite service but rather can be viewed as automated dynamic web service orchestration using the workflow technique.

Declarative approaches for process modeling include DECLARE (Pesic, 2007). A DECLARE model is mapped onto a set of LTL formulas that are used to automatically generate automata that support enactment. Adaptations with verification during enactment are supported, typically via GUI interaction with a human, whereby the changed model is reinitiated and its entire history replayed. As to inputs, DECLARE facilitates the definition of different constraint languages such as ConDec and DecSerFlow.

Concerning the combination of multi-agent systems and microservices, (Florio, 2015) proposes a multi-agent system for decentralized self-adaptation of autonomous distributed components (Docker-based microservices) to address scalability, fault tolerance, and resource consumption. These agents known as selfLets mediate service decisions using partial knowledge and exchanging messages. (Toffetti, Brunner, Blöchlinger, Dudouet, & Edmonds, 2015) provide a position paper focusing on microservice monitoring and proposing an architecture for scalable and resilient self-management of microservices by integrating management functions into the microservices, wherein service orchestration is cited to be an abstraction of deployment automation (Karagiannis et al., 2014), microservice composition or orchestration are not addressed.

Related standards include OWL-S (Semantic Markup for Web Services), an ontology of services for automatic web service discovery, invocation, and composition (Martin et al., 2004). Combining semantic technology with microservices, (Anderson, Suarez, Xu, & David, 2015) present an OWL-centric framework to create context-aware applications, integrating microservices to aggregate and process context information. For a more lightweight semantic description of microservices, JSON-LD (Lanthaler & Gütl, 2012) and Hydra (Lanthaler, 2013) (Lanthaler & Gütl, 2013) provide a lightweight vocabulary for hypermedia-driven Web APIs and enable the creation of generic API clients.

In contrast to the above work, our contribution specifically focuses on microservices, proposing and investigating an automatic lightweight declarative approach for the workflow-centric orchestration of microservices using agent-based clients, graph-based methods, and lightweight semantic vocabularies like JSON-LD and Hydra. Furthermore, adaptations during enactment do not require a complete reinitiation of the workflow and branching logic can be included to support automation.

3 Solution Approach

The principles and process constituting the solution approach described below reference the solution architecture of Fig. 1.

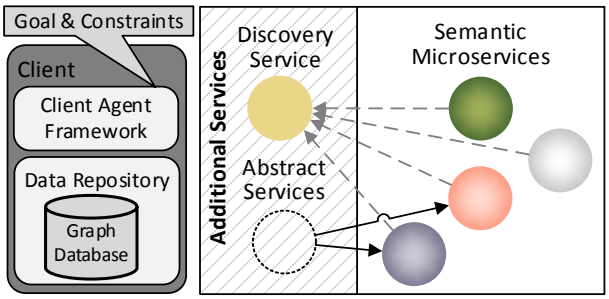

Fig. 1. Solution concept.

3.1 Microflow Solution Principles

The solution approach consists of the following principles:

Semantic self-description principle: microservices provide sufficient semantic metadata to support autonomous client invocation. For example, in our realization this was done using by using JSON-LD with Hydra.

Client agent principle: Intelligent agents exhibit reactivity, proactiveness, and social ability, managing a model of their environment and can plan their actions and undertake goal-oriented behavior (Wooldridge, 2009). Nominal WfMS are typically passive, executing a workflow according to a manually determined plan (workflow schema). Because of the expected scale in the number of possible microservices, the required goal-oriented choices in workflow modeling and planning, and the autonomous goal-directed action required during enactment, agent technology seems appropriate. Specifically, we chose Belief-Desire-Intention (BDI) agents (Bratman, Israel, & Pollack 1988) for the client realization, providing belief (knowledge), desire via goals, and intention utilizing generated plans that are the workflow.

Graph of microservices principle: microservices are mapped to nodes in a graph and can be stored in a graph database. Nodes in the graph are used to represent any workflow activity, such as a microservice. Nodes are annotated with properties. Directed edges depict the directed connections (flows) between activities annotated via properties. To reduce redundant resource usage via multiple database instances, the graph database could be shared by the clients as an additional microservice.

Microflow as graph path principle: A directed graph of nodes corresponds to a workflow, a sequence of operations on those microservices, and is determined by an algorithm applied to the graph, such as shortest path. The enactment of the workflow involves the invocation of microservices, with inputs and outputs retained in the client and corresponding to the client state.

Declarative principle: any workflow requirement specifications take the form of declarative statements, such as the starting microservice type, end microservice type, and constraints such as sequencing or branch logic constraints.

Microservice discovery service principle (optional): we assume a microservice landscape to be much more dynamic in microservices coming and going than

heavyweight services, and therefore utilize a microservice registry and discovery service. This could be deployed in different ways, including centralized, distributed, or having it embedded within each client, and utilize voluntary microservice-triggered registration or multicast mechanisms. For security purposes, there may be a wish to avoid discovery (of undocumented microservices) and thus maintain a whitelist. Clients may or may not have a priori knowledge of a particular microservice. Various broadcast services could be used.

Abstract microservices principle (optional): microservices with similar functionality (search, hotel booking, flight booking, etc.) can be grouped behind an abstract microservice. This provides an optional level of hierarchy to allow concrete microservices to only provide a client with link to the next abstract microservice(s), since the actual concrete microservices can be numerous and rapidly change, while determining exactly which one is appropriate can best be done by the client in conjunction with the abstract microservice.

Path weighting principle (optional): any followers of a service, be it abstract or concrete, can be weighted with a potentially dynamic cost that helps in quantifying and comparing one path with another in the form of relative cost. This also permits the navigation from one to another to be dynamically adjusted should that path incur issues such as frequent errors or slow responses. The planning agent can determine a path that minimizes the cost.

Logic principle (optional): if the path weighting is insufficient and more complex logic is desired for assessing branching or error conditions, these can be provided in the form of constraints referencing scripts.

Note that the Data Repository and Graph Database could readily be shared as a common service, and need not be confined to the Client.

3.2 Microflow Lifecycle

The Microflow lifecycle involves three stages as shown in Fig. 2.

Fig. 2. Microflow lifecycle.

The *Microservice Discovery* stage involves utilizing a microservice discovery service to build a graph of nodes containing the properties of the microservices and links to other microservices. This is analogous to mapping the landscape.

In the *Microflow Planning* stage, an agent takes the goal and other constraints and creates a plan known as a Microflow, finding an appropriate start and end node and using an algorithm such as shortest path to determine a directed path.

In our opinion, a completely dynamic enactment without any planning (no schema) could readily lead to dead-end or circular paths causing a waste of unnecessary invocations that do not lead to the desired goal and can potentially not be undone. This is analogous to following hyperlinks without a plan, which do not lead to the

goal and require backtracking. Alternatively, replanning after each microservice invocation involves planning resource overhead (CPU, memory, network), and since this is unlikely to dynamically change between the start and end of this lifecycle, we chose the pragmatic and hopefully more lightweight approach from the resource utilization perspective: plan once and then enact until an exception occurs, at which point a necessary replanning is triggered. Further advantages of our approach in contrast to a thoroughly adhoc approach is that the client is assured that there is at least one path to the goal, and validation of various structural, semantic, and syntactic aspects can be readily performed.

In the *Microflow Enactment* stage, the Microflow is executed by invoking each microservice in the order of the plan, typically sequentially but it could involve parallel invocations. A replanning of the remaining Microflow can be performed if an exception occurs or if notified by the discovery service of changes to the set of microservices. A client should retain the Microflow model (plan) and be able to utilize the service interfaces and thus have sufficient semantic knowledge for enactment.

The *Microflow Analysis* stage involves the monitoring, analysis, and mining of execution logs in order to improve future planning. This could be local, in a trusted environment, or this could be distributed. Thus, if invocation of a microservice has often resulted in exceptions, future planning for this client or other clients could avoid this troublesome microservice. Furthermore, the actual latency incurred for usage of a microservice could be tracked and shared between agents and taken into account as a type of cost in the graph algorithm.

4 Realization

A realization of the solution concept as a prototype involved mapping technology choices onto the solution concept (Fig. 3) and explained below.

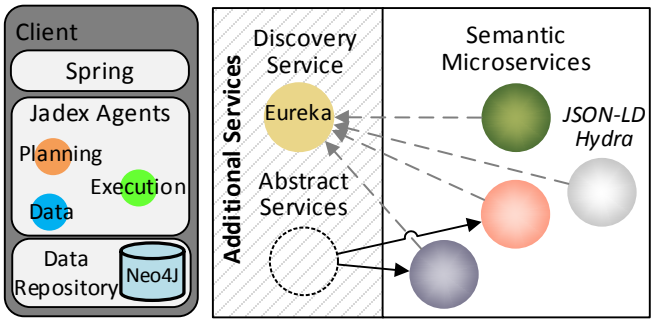

Fig. 3. Microflow solution realization technologies.

The prototype integrates the following, especially for REST (REpresentational State Transfer) and HATEOAS support (Fielding, 2000): Spring-boot-starter-web v. 1.2.4, which includes Spring boot 1.2.4, Spring-core and Spring-web v. 4.1.6,

Embedded Tomcat v. 8.0.23; Hydra-spring v. 0.2.0-beta3; and Spring-hateoas v. 0.16. For JSON (de)serialization Gson v. 2.6.1 is used. Unirest v. 1.3.0 is used to send HTTP requests. As a REST-based discovery service, Netflix's open source Eureka (Eureka, 2016) v. 1.1.147 is used.

4.1 Microservices

To support a larger-scale evaluation of the prototype, we created virtual microservices that differentiate themselves semantically but provide no real functionality. Microservice descriptions use Hydra based on JSON-LD, an example of which is shown in Fig. 4.


```
{
  "@context": {
    "@vocab": "http://schema.org/"
  },
  "@type": "Service",
  "serviceType": "Preferences",
  "serviceOutput": {
    "@type": "ItemList",
    "itemListElement": {
      "@type": "PropertyValue",
      "unitText": "",
      "value": "",
      "name": ""
    },
    "numberOfItems": 0
  },
  "offers": {
    "@type": "Offer",
    "businessFunction": "http://purl.org/goodrelations/v1#ProvideService"
  },
  "availableChannel": {
    "@type": "ServiceChannel",
    "serviceUrl": "http://192.168.0.14:8333/execute"
  },
  "description": "Get preferences service",
  "@id": "http://192.168.0.14:8333/",
  "hydra:followers": {
    "@id": "http://localhost:8333/followers",
    "hydra:operation": [
      {
        "hydra:method": "GET"
      }
    ]
  },
  "hydra:execute": {
    "@id": "http://localhost:8333/execute",
    "hydra:operation": [
      {
        "hydra:method": "PUT",
        "hydra:expects": {
          "@type": "URL",
          "hydra:supportedProperty": []
        }
      }
    ]
  }
}
```


Fig. 4. Example microservice description with Hydra.

Abstract microservices with similar functionality (hotel searching, payment, etc.) can be grouped behind an abstract microservice. This provides an optional level of

hierarchy to allow concrete microservices to only provide a client with link to the next abstract microservice(s), since the actual concrete microservice can change.

4.2 Microservice Clients

Microservice Client Agents. For an agent framework, the microservice clients uses the BDI agent framework Jadex v. 3.0-SNAPSHOT (Pokahr, Braubach, & Lamersdorf, 2005). Jadex's BDI nomenclature consists of Goals (Desires), Plans (Intentions), and Beliefs. Beliefs can be represented by attributes like lists and maps. Three agents were created: the DataAgent is responsible for providing for and maintaining data repository, the PlanningAgent generates a path through the graph as a Microflow, while the ExecutionAgent communicates directly with microservices to invoke them according to the Microflow. Neo4j and Neo4j-Server v. 2.3.2 is used as a client Data Repository.

Microflow Goals and Constraints. The goals and constraints for a MicroFlow are referred to as PathParameters and consist of the startServiceType (e.g., preferences), endServiceType (e.g., payment), and constraint tuples in JSON . Each constraint tuple consists of the target of the constraint (the service type affected), the constraint, and a constraint type (required, beforeNode, afterNode).

As an example, let us assume we have a process in mind such as that depicted in simplified form using Business Process Modeling Notation (BPMN) in Fig. 5 (whereby a BPMN model is not used or required by our solution). Let us assume that a search should determine the minimum total price for both flight and hotel within +/- 2 days of a given starting and ending date before booking. Here we assume the Search, Booking, and PaymentSpecification services are abstract and that one or more concrete services of their respective types are available (in BPMN subprocesses (not shown) could be used to determine these). After the booking, the type of payment service used should be determined based on total price (e.g., if > $2000 should use a billing service, between $500-$2000 should use a credit card service, and otherwise prepayment).

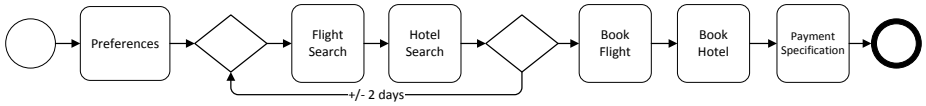

Fig. 5. Equivalent BPMN process desired.

The Microflow textual constraint specification for this example is shown in Fig. 6. For instance, target = "Hotel Search", constraint = "Book Hotel", and constraint type = "beforeNode" would be read as: "Hotel Search" before "BookHotel", implying the Microflow sequencing must ensure that "Search Hotel" precedes "Book Hotel" (but must not be directly before it).

```
{  "startServiceType":"Get preferences",
   "endServiceType":"Completion",
   "constraints":[
      { "type": "RequiredNode", "target": "Book Flight"},
      { "type": "RequiredNode", "target": "Book Hotel"},
      { "type": "BeforeNode", "target": "Flight Search", "constraint": "Book Flight"},
      { "type": "BeforeNode", "target": "Hotel Search", "constraint": "Book Hotel"},
      { "type": "AfterNode", "target": "Payment Specification", "constraint": "Abstract Booking"},
      { "type": "BranchAfterExecution", "target": "Abstract search", "constraint": "<PathToGroovyScript>"},
      { "type": "BranchAfterExecution", "target": "Payment Specification", "constraint": "<PathToGroovyScript>"},
      { "type": "IncreaseCostOnError", "target": "All", "constraint": "4"}
   ]}
```

Fig. 6. Goal and constraints inputs in JSON.

By default, branching occurs based on a minimization of total path cost across all path segments. However, to flexibly support branching logic independent of the microservice implementation, branching constraints can be provided consisting of the type "BranchAfterExecution", the target node, with the constraint consisting of the path to a Groovy script consisting of program logic that dynamically determines and returns a target node to branch to as a JSON string. As input, the script is passed all next follower nodes (potential branch targets) as a JSON string and the workflow state (consisting of the inputs and outputs from all previously invoked nodes) as a map. For example, the groovy script could be used to branch

In order to specify error or exception handling, a constraint type "IncreaseCostOnError" is provided, with either a specific target node or "All", and a constraint consisting of a multiplication factor with which to dynamically adjust the cost weighting of a problematic segment of a path to that node (e.g., the credit card has expired, or the node did not respond in time, etc., that can be used to constrain the number of retries). This factor can also be set to "MaxInt" to essentially ensure a problematic segment will not be retried. HTTP response status code of 5xx server errors cause a Microflow to navigate back to the previous abstract node and permit another attempt from there, under the assumption that the client is OK and the that particular service had an issue. Furthermore, if an "IncreaseCostOnError" factor exists it penalizes that segment's cost weighting by the given multiplication factor, and then reinvokes any branching logic script if any was specified, otherwise it triggers a replanning that minimizes the total remaining path cost. On 4xx client errors, a restart of the Microflow from the beginning is initiated, under the assumption that perhaps some changes in the resource environment are causing the client to make erroneous requests, and the entire plan and its enactment should be reinitialized. An "OnError" type constraint with a groovy script can be used for more complex error handling scenarios where increasing a path segment alone is insufficient.

The set of constraint tuples are analyzed on Microflow initialization, whereby any AfterNode is converted to a BeforeNode by swapping target and constraint, then ordered, and then checked if any constraint is redundant. Then RequiredNode constraints are also converted to BeforeNode constraints. A PathWrapper is used because of occasional issues incurred when passing Path objects in the Neo4J format between agents.

4.3 Microflow Lifecycle

We now describe the various microflow lifecycle stages.

Microflow Discovery Stage. The *Microservice Discovery* stage involves the interactions shown in Fig. 7, where Microservices first register themselves with the DiscoveryService. On client initialization, the DataAgent has the DataRepository fetch (via its DatabaseController) the registered services from the DiscoveryService and retrieve the service description from each microservice rather than a central repository. This avoids the issues of the discovery service retaining duplicate or incorrect (stale) semantic data.

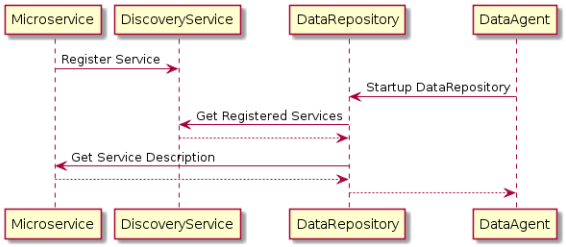

Fig. 7. Microservice Discovery stage interactions.

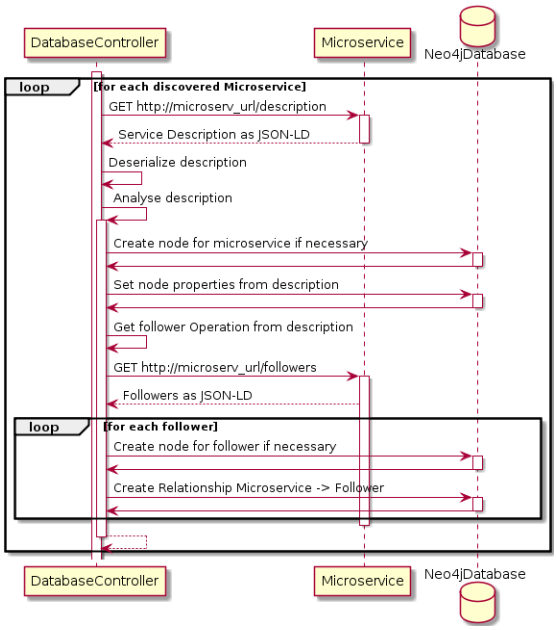

Fig. 8. Microservice description collection interactions.

In Fig. 8, the semantic description of the microservice is retrieved and, if a node does not yet exist, a node is inserted in the graph along with its properties. All followers are also inserted (if not already) and their association with this microservice is annotated as a directed edge. If any microservices are detected that were not (yet) registered with the discovery service, these are also tracked in a list.

Microservice Planning Stage. During the *Microservice Planning* stage, the PlanningAgent plans a Microflow. It has two Beliefs: PathParameters (the input) and the Path (see Fig. 9). The annotations show that anytime PathParameters changes, Jadex triggers a planning.

```
1 @Plan(trigger = @Trigger(factchangeds = "pathParameters"))
2 public void pathParametersChanged(ChangeEvent event) {
3  if (algorithmService != null) {
4    validPaths = algorithmService.getShortestPaths(param);
```

Fig. 9. Microflow planning triggering in Jadex.

Although Neo4J offered native graph algorithms, they did not completely fulfill our requirements. While we utilize them, we generate Microflows with our own algorithm as shown in Fig. 10. After converting the constraints (Line 1-3) as described above, the set of possible starting microservices matching the starting type are determined (Line 4). Then this set is iterated over using the shortestPath algorithm, trying to find a path to the start of the next pathPart, which is either the target of the next constraint or the endServiceType, which is iterated (Line 7) since multiple nodes are possible. Then a recursive calculation of pathParts is initiated (Line 10), which either ends due to a deadend (Line 17) or the path to a valid endServiceType being found (Line 15).

```
1  constraints = analyze(constraints);
2  // list of constraints
3  cList = orderConstraints(constraints);
4  startList = findServicesForServiceType(startType);
5  foreach(service in startList) {
6    nextTargetList = findServicesForServiceType(cList[0]);
7    for(target in nextTargetList){
8      path = findPath(service, target);
9      if(path.isValid()){
10       pathPartList = calculateNextPathPart(target,
11         cList[1...cList.length()-1]);
12       if(pathPartList.isValid()){
13         pathPartList.prepend(path);
14         possiblePathsList.add(pathPartsToPath(pathPartsList));
15         break; // Stop, valid path from a start found
16       }
17     }
18   }
19 }
20 calculatedPath = findBestPath(possiblePathsList);
21 return calculatedPath;
```

Fig. 10. Microflow generation algorithm (pseudocode).

The Microflow schema is currently only applicable for the current enactment, so that future enactments involve a replanning. However, the Microflow schema (sequence plans) could be retained and reused if desired - for instance, if nothing changed in the environment. If multiple clients and thus agents coexisted in a trusted environment, then they could utilize their social communication ability to request and share Microflows.

Although support for gateways (forking and merging) and intermediate events are feasible in this approach, are prototype did not yet realize this functionality at this time. Support for using costs with graph paths is implemented but not utilized in our evaluation, since with virtual microservices it appeared artificial for the focus of our investigation.

In focusing on a lightweight approach, and not requiring interoperability, we chose to avoid the XML-centric BPEL and BPMN, which would only have added extra overhead in our case study without any benefit.

Microservice Enactment Stage. For the *Microflow Enactment* stage, the ExecutionAgent is primarily responsible. It has three beliefs: pathWrapper, currentNode (points to which node is either active or about to be executed), and path (the planned Microflow), and similar to Fig. 9, the ExecutionAgent's plan is triggered by a change to the path variable (by the PlanningAgent), as shown in Fig. 11.

```
1 @AgentArgument
2 @Belief
3 protected PathWrapper pathwrapper;
4 @Belief
5 protected int currentNode = -1;
6 @Belief
7 protected Path path;
8
9 @Plan(trigger = @Trigger(factchangeds = "path") )
10 public void startPathExecution(ChangeEvent event) {
```

Fig. 11. ExecutionAgent (snippet).

The Microflow enactment algorithm is shown in Fig. 12. Line 7 shows that abstract nodes are skipped. Line 12 is a loop for the case when a microservice takes more than one input. In Line 15 the output of this invocation is retained for possible input as client state during further Microflow processing. Because the microservice invocations are asynchronous, a Java CountDownLatch is used for synchronization purposes. Line 21 shows that a new Microflow planning starting with the current node is triggered when an error occurs with avoidance of the problematic microservice if possible (e.g., if other identical microservice types are available) - otherwise a retry can be attempted. In addition, the initial constraints are readjusted since certain constraints may no longer be applicable (e.g., if they were already fulfilled in the partial Microflow already executed). Line 28 determines a new workflow plan after a branch. For the non-branching case the next node in the plan is taken (Line 30).

```
// toBeExecutedNode = index of node in workflow/path
// possibleInputList = list of available inputs generated by previous services
if (!isNodeValid(toBeExecutedNode))
  return; // stop execution
serviceDescription = getServiceDescription(toBeExecutedNode);
if (isNodeAbstract(serviceDescription)){
  toBeExecutedNode++; // continue with next node in workflow
  return;
}
if (!isValidInputAvailable(serviceDescription))
  return; // stop execution
for each (input in getValidInputList(serviceDescription)){
  response = executeServiceWithInput(serviceDescription, input);
  if (response.OK()){
    possibleInputList.add(response.getBody());
  }else{
    // On 5xx error, go back to the last abstract node, else restart WF
    // The new WF contains all previously visited nodes up to the last abstract one
    // Optionally: transition to current node is penalized by given cost factor
    // Keep inputs up to the last abstract node
    toBeExecutedNode = generateNewWorkflowAfterError(toBeExecutedNode);
    return;
  }
}
// The branching is encoded in the constraints.
// The new WF contains the already visited nodes up to the current one.
if(isBranchingNode(serviceDescription, constraints))
  toBeExecutedNode = generateNewWorkflowAfterBranching(toBeExecutedNode);
else // normal non-branching case is the next node in the plan
  toBeExecutedNode++;
```

Fig. 12. Microflow enactment algorithm (pseudocode).

Fig. 13 shows the interactions when a Microflow is enacted. Within a loop a PUT is used to invoke each virtual microservice in the Microflow for testing purposes, this would be adjusted for real microservices. When an error occurs, a replanning is done from the prior abstract node, and the transition path segment cost to the current node is penalized by a factor if given. If a branch is involved, a new path is calculated from the branch taken.

While the service description could be retrieved directly from the microservice, we currently use the internal copy stored during the discovery stage to avoid the additional network and microservice overhead of retrieving this information again. If the description is expected to be highly dynamic, the current description could be retrieved from the microservice during enactment.

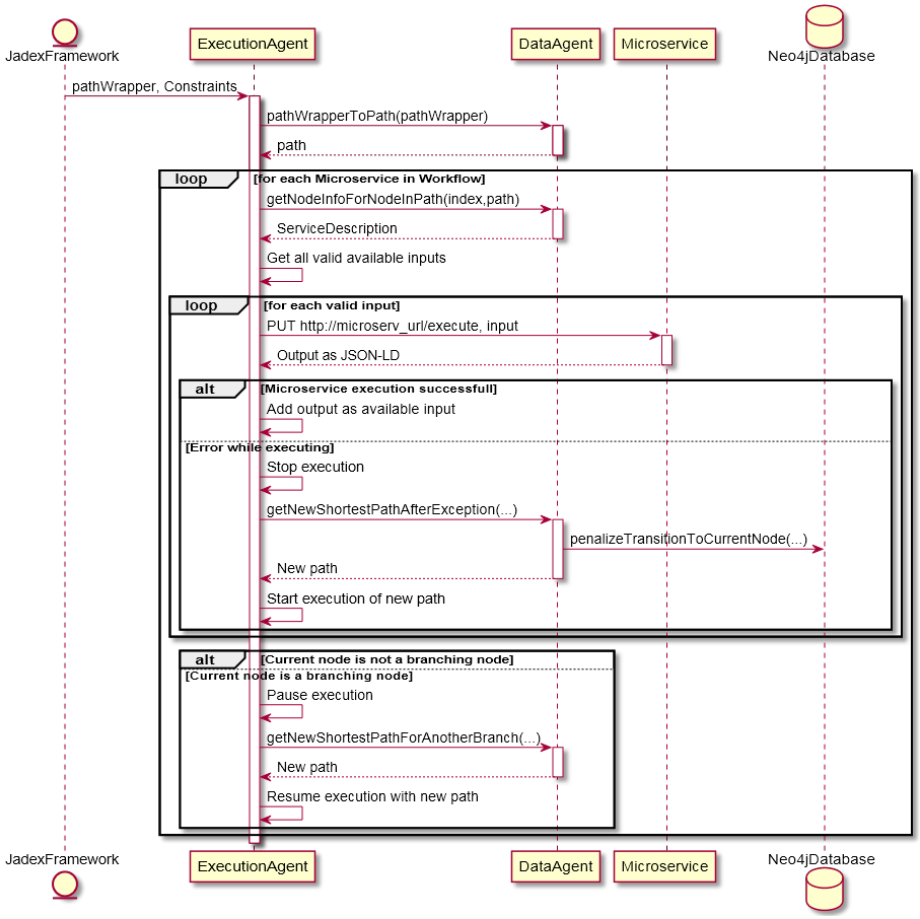

Fig. 13. Microflow enactment interactions.

5 Evaluation

In our prior paper (Oberhauser, 2016), we evaluated resource utilization and scalability to validate the lightweight nature and performance of Microflows. We also compared the typical standard BPM modeling approach vs. the constraint-based declarative automated planning approach of Microflows as far as modeling efficiency. Here we provide a case study that illustrates the solution approach including the use of path costs and branching.

5.1 Case Study

In this case study, we utilize a Microflow example comparable to standard workflows based on the previously described BPMN model Fig. 5 and the Microflow equivalent constraints of Fig. 6. Based on the Neo4j graph, the annotated Microflow is shown in Fig. 16. The Microflow consists of 12 nodes: Get preferences (node 1), Abstract search (node 2), Search for Flight service (node 3), Search for Hotel service (node 4), Abstract Booking (node 5), Execute Flight Booking (node 6), Execute Hotel booking (node 7), Payment Specification (node 8), which can be followed by a conditional Branch to either Prepayment (node 9), Bill Payment (node 10), or Credit Card Payment (node 11), followed by a Merge to the Completion (node 12). Let us assume the following path segment costs: a=30, b=6, c=8, d=2, e=10, f=7, g=4, h=3, i=19, j=20, k=3, l=10, m=30, n=20, o=10, p=10, q=10. Prepayment is the cheapest and least risky form of transaction, credit card is a more expensive transaction with little risk, while billing the customer involves the most risk and longest wait for funds.

If branching logic is provided in the constraints, then this logic in the form of a groovy script determines the next node to attempt. If branch logic is provided by "BrachAfterExecution" after Abstract search via a groovy script, then a loop of additional search invocations to determine the total minimum cost within two days before and after the given date preference can be attempted before continuing to Abstract Booking. If branch logic is provided at Payment Specification, then it determines the choice (e.g., based on total price, if > $2000 should use a billing service, between $500-$2000 should use a credit card service, and otherwise prepayment, or based on various risk factors associated with the customer profile).

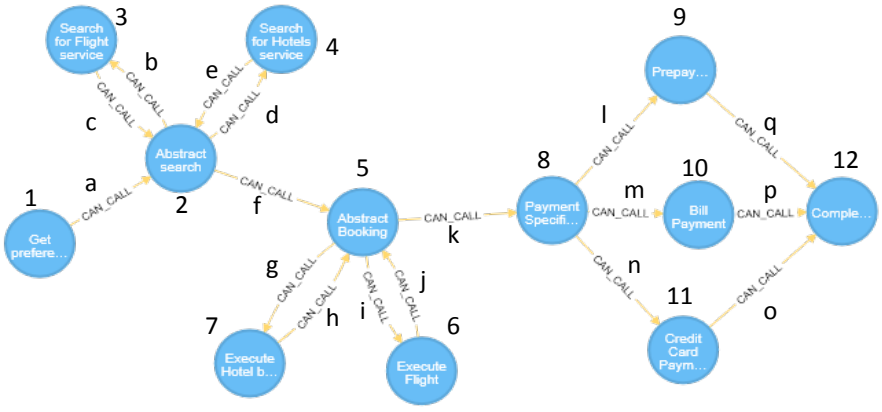

Fig. 14. Microflow example shown in Neo4j as nodes and call options (annotated with node numbers and path segment letters).

Without branching logic, the path costs are used. The total path cost up to node 8 (Payment Specification) is 112. Absent any such branching logic, then Payment Specification is an abstract node that provides the possible followers. In this case, since Prepayment (node 9) is the least costly (cost 10), it invokes it but receives an

error (e.g., due to insufficient funds). Because of the given "IncreaseCostOnError" factor 4, the new cost is calculated to be l=40. The least cost path is now calculated to be via node 11, which is invoked and also receives an error (e.g., credit card expired). The cost of the path segment is increased by the factor to n=80. The cheapest path is now via node 10 (Bill Payment), and this is invoked and succeeds with a cost of 30. The total path costs were 182 including the erroneous invocation paths taken.

6 Conclusions

We described Microflows, an automatic lightweight declarative approach for the workflow-centric orchestration of semantically-annotated microservices using agent-based clients, graph-based methods, and lightweight semantic vocabularies. Microflow principles and its lifecycle were presented and its prototype realization described. A case study demonstrated the branching, path segment cost weighting, and error handling capabilities.

Compared to a naive Restful approach, one advantage we see in the Microflow approach is that the plan (or workflow) is not thoroughly adhoc and dynamic, so that validation and verification checks can be performed before execution and one is assured that at least one path to the goal exists before beginning with enactment. For instance, if all microservices were there, but a payment service of the required type is missing, then a client without this knowledge would work its way through and realize at the very end that it has no way to pay.

Thus, future work includes integrating advanced verification and validation techniques for the automatic planning, optimizing resource usage, integrating semantic support in the discovery service, supporting compensation and long-running processes, and enhancing the declarative and semantic support and capabilities.

Acknowledgments. The author thanks Florian Sorg and Sebastian Stigler for their assistance with the design, implementation, evaluation, and diagrams.

References

Alpers, S., Becker, C., Oberweis, A. and Schuster, T. (2015). Microservice based tool support for business process modelling. In Enterprise Distributed Object Computing Workshop (EDOCW), 2015 IEEE 19th International (pp. 71-78). IEEE.

Anderson, C., Suarez, I., Xu, Y., & David, K. (2015). An Ontology-Based Reasoning Framework for Context-Aware Applications. In Modeling and Using Context (pp. 471-476). Springer International Publishing.

Bouguettaya, A., Sheng, Q.Z. and Daniel, F. (2014). Web services foundations. Springer.

Bratman, M.E., Israel, D.J. and Pollack, M.E. (1988). Plans and resource-bounded practical reasoning. Computational intelligence, 4(3), pp.349-355.

Eureka (2016). Retrieved April 20, 2016 from: https://github.com/Netflix/eureka/wiki

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine.

Florio, L. (2015). Decentralized self-adaptation in large-scale distributed systems. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (pp. 1022-1025). ACM.

Fowler, M. & Lewis, J. (2014). Microservices a definition of this new architectural term. Retrieved April 15, 2016 from: http://martinfowler.com/articles/microservices.htm

Gartner (2015). Gartner Says Spending on Business Process Management Suites to Reach $2.7 Billion in 2015 as Organizations Digitalize Processes. Press release. Retrieved April 15, 2016 from: https://www.gartner.com/newsroom/id/3064717

Heitmann, B., Cyganiak, R., Hayes, C. & Decker, S. (2012). An empirically grounded conceptual architecture for applications on the web of data. Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, 42(1), 51-60.

IBM (2015). IBM Business Process Manager V8.5.6 documentation. Retrieved May 2, 2016 from: http://www.ibm.com/support/knowledgecenter/SSFPJS_8.5.6/com.ibm.wbpm.wid.bpel.doc/topics/cprocess_transaction_micro.html

Karagiannis, G., Jamakovic, A., Edmonds, A., Parada, C., Metsch, T., Pichon, D., ... & Bohnert, T. M. (2014). Mobile cloud networking: Virtualisation of cellular networks. In Telecommunications (ICT), 2014 21st International Conference on (pp. 410-415). IEEE.

Lanthaler, M. (2013). Creating 3rd generation web APIs with hydra. In Proceedings of the 22nd international conference on World Wide Web companion. International World Wide Web Conferences Steering Committee, pp. 35-38.

Lanthaler, M., & Gütl, C. (2012). On using JSON-LD to create evolvable RESTful services. In Proceedings of the Third International Workshop on RESTful Design (pp. 25-32). ACM.

Lanthaler, M. and Gütl, C. (2013). Hydra: A Vocabulary for Hypermedia-Driven Web APIs. In Proceedings of the 6th Workshop on Linked Data on the Web (LDOW2013) at the 22nd International World Wide Web Conference (WWW2013), vol. 996.

Makrai, G. (2015). Experimenting with Dijkstra's algorithm. Retrieved May 2, 2016 from: https://gabormakrai.wordpress.com/2015/02/11/experimenting-with-dijkstras-algorithm/

Martin, D. et al. (2004). OWL-S: Semantic markup for web services. W3C member submission, 22, pp.2007-04.

Oberhauser, R. (2016). Microflows: Lightweight Automated Planning and Enactment of Workflows Comprising Semantically-Annotated Microservices. In Proceedings of the Sixth International Symposium on Business Modeling and Software Design (BMSD 2016) (pp. 134-143). SCITEPRESS.

Pesic, M., Schonenberg, H., & van der Aalst, W. M. (2007). Declare: Full support for loosely-structured processes. In Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International (pp. 287-287). IEEE.

Pokahr, A., Braubach, L., & Lamersdorf, W. (2005). Jadex: A BDI reasoning engine. In Multi-agent programming (pp. 149-174). Springer US.

Rajasekar, A., Wan, M., Moore, R., & Schroeder, W. (2012). Micro-Services: A Service-Oriented Paradigm for. Data Intensive Distributed Computing. In: Challenges and Solutions for Large-scale Information Management (pp. 74-93). IGI Global.

Rao, J. and Su, X. (2004). A survey of automated web service composition methods. In Semantic Web Services and Web Process Composition (pp. 43-54). Springer.

Sheng, Q. Z. et al. (2014). Web services composition: A decade's overview. Information Sciences, 280, 218-238.

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F., & Edmonds, A. (2015). An architecture for self-managing microservices. In Proceedings of the 1st International Workshop on Automated Incident Management in Cloud (pp. 19-24). ACM.

WfMC (1999). Workflow Management Coalition: Terminology & Glossary. WFMC-TC-1011, Issue 3.0.

Wooldridge, M. (2009). An introduction to multiagent systems. John Wiley & Sons.