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

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

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

Abstract: Business processes are facing increasing pressure to quickly and flexibly adapt to changes in the process context. Moreover, microservices are becoming increasingly popular as an architectural style for partitioning business logic into small services accessible with lightweight mechanisms, leading to increasing pressure for a more dynamic integration of information services with processes. Process-aware information systems must thus increasingly incorporate the ability to react to unforeseen changes during process enactment, facing difficulties in pre-modelling all the possible process variations and enactment circumstances for larger process models. 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. The evaluation results show the approach's potential in lightweight resource utilization, investigates its scalability, and compares its automation to common manual workflow modeling and enactment.

1 INTRODUCTION

In many areas of society today, a trend towards increased automation can be observed. One area in particular is that known as business processes or workflows. 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). 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.

Moreover, there is an increasing trend toward applying the microservice architecture style (Fowler, and 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 et al., 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 indicates that it is achieved by only 11% of Semantic Web applications (Heitmann et al., 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.

Oberhauser R.
Microflows: Lightweight Automated Planning and Enactment of Workflows Comprising Semantically-Annotated Microservices.
DOI: 10.5220/0006223001340143
In *Proceedings of the Sixth International Symposium on Business Modeling and Software Design (BMSD 2016)*, pages 134-143
ISBN: 978-989-758-190-8

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. 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 et al., 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 et al., 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 and 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.

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 et al., 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 et al., 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 and Gütl, 2012) and Hydra (Lanthaler, 2013) (Lanthaler and 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.

3 SOLUTION APPROACH

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

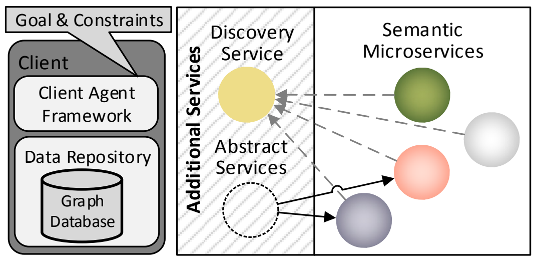


Figure 1: Solution concept.

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 et al., 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 constraints.

Microservice Discovery Service Principle (Optional): awe 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.

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 Figure.

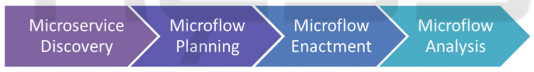


Figure 2: Microflow lifecycle.

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 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 (Figure) and explained below.

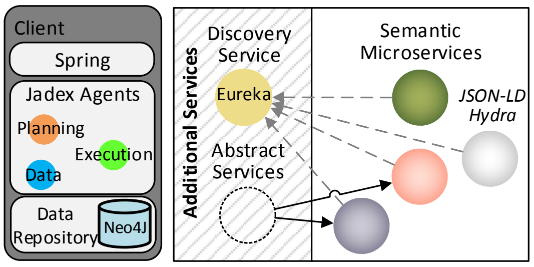
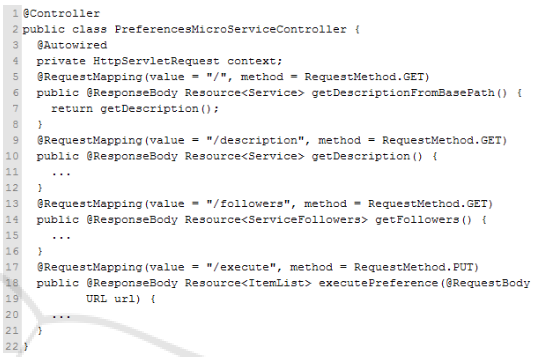


Figure 3: Microflow solution realization technologies.

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.

4.1 Microservices

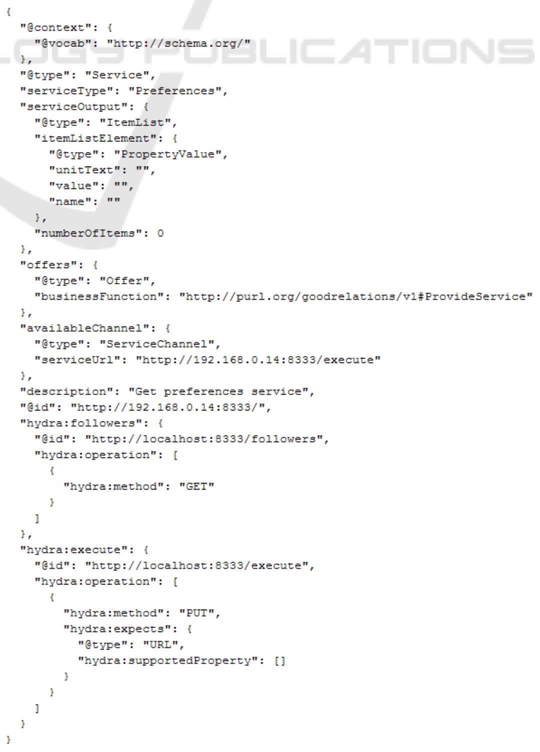
A code snippet of the Spring-based controller for the microservices is shown in Figure 4. Followers was explicitly provided to avoid having to know how to invoke domain-specific microservice operations when only the potential followers are of interest.



```
@Controller
public class PreferencesMicroServiceController {
  @Autowired
  private HttpServletRequest context;
  @RequestMapping(value = "/", method = RequestMethod.GET)
  public @ResponseBody Resource<Service> getDescriptionFromBasePath() {
    return getDescription();
  }
  @RequestMapping(value = "/description", method = RequestMethod.GET)
  public @ResponseBody Resource<Service> getDescription() {
    ...
  }
  @RequestMapping(value = "/followers", method = RequestMethod.GET)
  public @ResponseBody Resource<ServiceFollowers> getFollowers() {
    ...
  }
  @RequestMapping(value = "/execute", method = RequestMethod.PUT)
  public @ResponseBody Resource<ItemList> executePreference(@RequestBody
         URL url) {
    ...
  }
}
```

Figure 4: Example microservice Spring controller.

An example microservice description using JSON-LD and Hydra is shown in Figure 5.



```
{
  "@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": []
        }
      }
    ]
  }
}
```

Figure 5: Example microservice description with Hydra.

To support a larger-scale evaluation of the prototype, we created virtual microservices that differentiate themselves semantically but provide no real functionality. As a REST-based discovery service, Netflix's open source Eureka (Eureka, 2016) v. 1.1.147 is used.

4.2 Microservice Client

For the client, Jadex v. 3.0-SNAPSHOT is used as a BDI agent framework (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. For the client's Data Repository, Neo4j and Neo4j-Server v. 2.3.2 is used.

4.3 Microflow Lifecycle

The goals and constraints are referred to as PathParameters and consist of the startServiceType (e.g., preferences), endServiceType (e.g., payment), and constraint tuples in JSON as shown in Figure 6. Each constraint tuple consists of the target of the constraint (the service type affected), the constraint, and a constraint type (required, beforeNode, afterNode). For instance, target = "Book Hotel", constraint = "Search Hotel", and constraint type = "afterNode" would be read as: "BookHotel" after "Search Hotel", implying the microflow sequencing must ensure that "Search Hotel" precedes "Book Hotel" (but must not be directly before it).

```
{ "startServiceType":"Preferences",
  "endServiceType":"Payment",
  "constraints":[
  {"type":"RequiredNode","target":"Flight Search"},
  {"type":"AfterNode","target":"Payment","constraint":"Book Hotel"},
  {"type":"BeforeNode","target":"Hotel Search","constraint":"Book Hotel"},
  {"type":"AfterNode","target":"Payment","constraint":"Book Flight"}
  ]}
```

Figure 6: Goal and constraints inputs in JSON.

These set of constraint tuples are analyzed, 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.

We used a PathWrapper because of occasional issues incurred when passing Path objects in the Neo4J format between agents.

4.3.1 Microservice Discovery Stage

The *Microservice Discovery* stage involves the interactions shown in Figure 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.

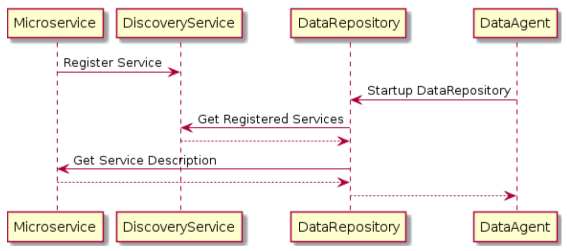


Figure 7: Microservice Discovery stage interactions.

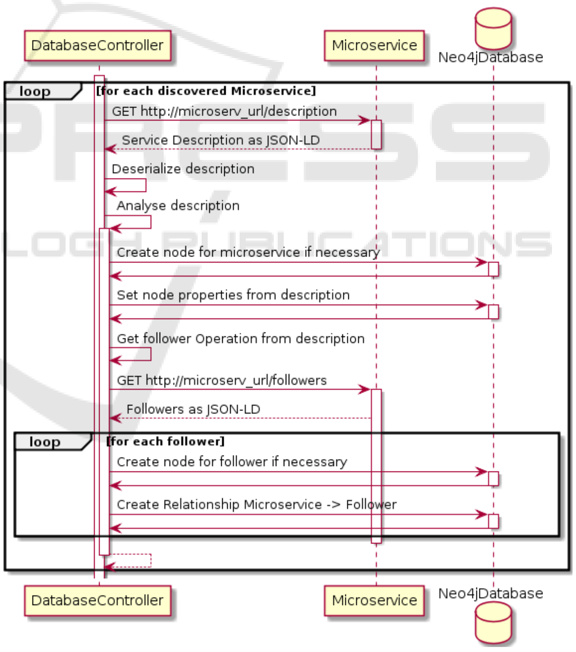

DatabaseController
Microservice
Neo4jDatabase
loop
[for each discovered Microservice]
GET http://microserv_url/description
Service Description as JSON-LD
Deserialize description
Analyse description
Create node for microservice if necessary
Set node properties from description
Get follower Operation from description
GET http://microserv_url/followers
Followers as JSON-LD
loop
[for each follower]
Create node for follower if necessary
Create Relationship Microservice -> Follower
DatabaseController
Microservice
Neo4jDatabase

Figure 8: Microservice description collection interactions.

In Figure 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.

4.3.2 Microservice Planning Stage

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

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

Figure 9: Microflow planning triggering.

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 Figure 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;
```

Figure 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.

4.3.3 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 Figure 9, the ExecutionAgent's plan is triggered by a change to the path variable (by the PlanningAgent), as shown in Figure 11.

```
@AgentArgument
@Belief
protected PathWrapper pathwrapper;
@Belief
protected int currentNode = -1;
@Belief
protected Path path;

@Plan(trigger = @Trigger(factchangeds = "path") )
public void startPathExecution(ChangeEvent event) {
```

Figure 11: ExecutionAgent (snippet).

The microflow enactment algorithm is shown in Figure 12. Line 8 shows that abstract nodes are skipped. Line 14 is a loop for the case when a microservice takes more than one input. In Line 17 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 19 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).

```
// toBeExecutedNode = current node index in workflow/path
// possibleInputList = available inputs from previous services

if(!isNodeValid(toBeExecutedNode))
  return;
serviceDescription = getServiceDescription(toBeExecutedNode);
if(isNodeAbstract(serviceDescription)){
  toBeExecutedNode++; // continue with next node in workflow
  return;
}
if(!isValidInputAvailable(serviceDescription))
  return;

foreach(input in getValidInputList(serviceDescription)){
  response= executeServiceWithInput(serviceDescription, input);
  if (response.OK()){
      possibleInputList.add(response.getBody());
  }else{
      generateNewWorkflowAfterError();
      toBeExecutedNode = 0;
      return;
  }
}
toBeExecutedNode++;
```

Figure 12: Microflow enactment algorithm (pseudocode).

Figure 13 shows the interactions when a microflow is enacted. Within a loop, a PUT is used to invoke each virtual microservice for testing purposes, but this would be adjusted for real microservices.

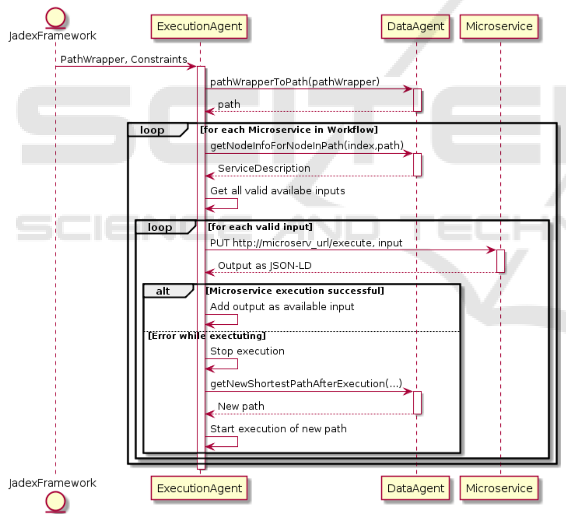

JadexFramework
ExecutionAgent
DataAgent
Microservice
PathWrapper, Constraints
pathWrapperToPath(pathWrapper)
path
loop
[for each Microservice in Workflow]
getNodeInfoForNodeInPath(index,path)
ServiceDescription
Get all valid availabe inputs
loop
[for each valid input]
PUT http://microserv_url/execute, input
Output as JSON-LD
alt
[Microservice execution successful]
Add output as available input
[Error while executing]
Stop execution
getNewShortestPathAfterExecution(...)
New path
Start execution of new path
JadexFramework
ExecutionAgent
DataAgent
Microservice

Figure 13: Microflow enactment interactions.

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.

5 EVALUATION

To evaluate the solution approach, we investigated if the resource usage of prototype was relatively lightweight, if it could execute the equivalent workflow of a BPMS, and determine if it shows any potential advantage in labor overhead.

The configuration used for the evaluation consisted of a PC with Windows 10 Pro x64, Intel Core i5-4460@3.2 GHz, 8 GB RAM, Java JRE 1.8.0_66-b18. Unless noted, the average of 10 consecutive measurements is given.

5.1 Resource Utilization

To determine the resources utilized by Neo4J, the number of microservices was scaled using 29 different configurations ranging from 100 to 6400 microservices, an extract of which is shown in Table 1. Only one outgoing edge for each microservice was used to keep that variable constant. The drop in RAM usage after 1600 microservices may be a result of garbage collection, and we intend to repeat these measurements with more control over that factor. These measurements show that a very large number of microservices can be supported with relatively little additional RAM or disk impact.

Table 1: Neo4j resource usage.

Number of Microservices	Diskspace (MB)	RAM (MB)
100	0.211	115
400	0.315	120
800	0.716	150
1600	0.741	175
3200	1.28	110
6400	2.39	115

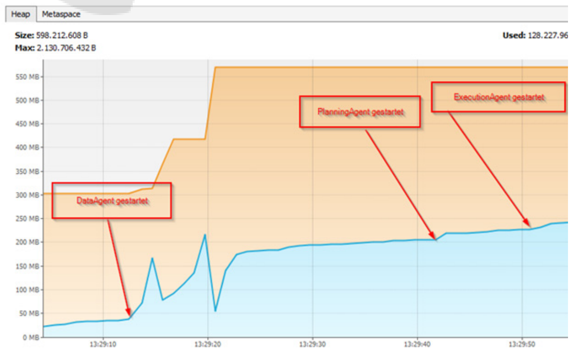


Figure 14: RAM profiling showing agent starting points.

To determine if the Jadex agents have a significant impact on RAM usage, profiling with the VisualVM was performed as shown in Figure 14. The DataAgent was started first (RAM use was in accordance with Table 1), while the PlanningAgent and ExecutionAgent had no major RAM impacts.

To investigate the performance and scalability of the microflow planning stage, a small program was written that generates z layers of microservices, each layer of which contains m microservices, and each microservice of a layer z has an edge to every microservice of the layer z+1.

Neo4j does not explicitly name the algorithm implemented for shortest path, but let us assume it is at least as good as the Dijkstra algorithm, which it also offers and appears to use Fibonacci Heaps (Makrai, 2015), yielding a complexity:

$$O(v \cdot \log v + e) \qquad (1)$$

where v are the vertices and e the edges. If n is the number of constraints, then n+1 segments have to be computed between the start and end vertex. Let x_i be the possible number of start vertices and y_i the possible number of end vertices for a segment i where $0 \leq i \leq n$, then in every segment there are a maximum of $x_i \cdot y_i$ shortest paths, resulting in:

$$O(\sum_{i=0}^{n}(x_i \cdot y_i) \cdot (v \cdot \log v + e)) \qquad (2)$$

Thus increasing the number of possible starting or ending nodes has a greater influence on performance.

We performed an experiment comparing the shortest path and Dijkstra algorithm performance, and the shortest path was faster in all cases, so we continued with shortest path.

As expected, when we increased the possible number of starting or ending nodes while keeping the total number of microservices constant, we observed a much larger impact on performance than any increase to the number of segments.

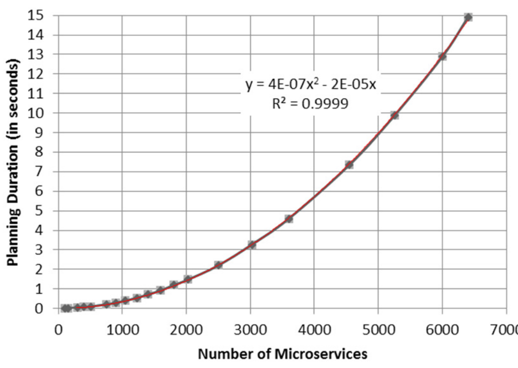


Figure 15: Microflow planning performance of planning duration vs. number of microservices.

Figure 15 shows the planning performance impact as the total number of microservices increase with a best-fit equation shown. While this may not be ideal, it may suffice for pragmatic usage in non-time critical situations where sufficient CPU resources are available and when the total number of microservices to be considered is limited.

5.2 Microflow Vs. Workflow

To attempt to provide insight into a pragmatic comparison of microflows to standard workflows, a user familiar with our microflow concept and somewhat fairly familiar with BPM, using the microflow described in Section 4.3 and shown in Figure 16 as a basis, and modeled its equivalent as a workflow in AristaFlow BPM Suite (representing a BPMS). The workflow consisted of 12 nodes and 13 edges: Start, Flight Search, Hotel Search, Book Flight, Book Hotel, Booking Error Check (which loops back to start on an error), followed by a conditional Branch to either Pay by CreditCard or Pay by Bank, than a Merge and then an End node.

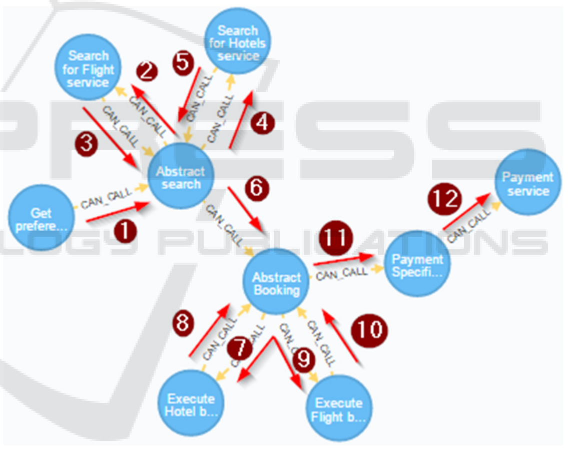


Figure 16: Microflow shown in Neo4j.

For the microflow, manually preparing the microflow constraints and starting the Jadex and DataAgent involved 4:24 minutes; the automatic planning took 3.9 seconds; and the enactment of the virtual microservices 4.7 seconds. For the equivalent workflow using empty activities that do not actually invoke services, manual creating the process schema took 19:39 minutes while the enactment of the workflow took 8 seconds.

This was not necessarily a "fair" comparison - since a BPMS supports many more capabilities such as correctness checks and the user was not independent. Nevertheless, the point of this exercise was to show that the input constraints needed for microflows (cp. Figure 6) can be more lightweight, and that the utilization of the dynamic planning capability can reduce the labor overhead of manual planning of workflows in the microservice space, especially if these are expected to vary often. Analogous to the more heavyweight EJB containers vs. more lightweight containers, perhaps the more lightweight form of microflows could be beneficial when the more complete BPMS functionality is not needed.

To validate its exception handling and replanning capability, we manually created situations where certain microservice returned an error, and observed that the agent triggered a replanning consisting of the error segment plus the remaining segments, providing some resilience.

Note that performance was intentionally not optimized in order to provide an indicator of the default viability and investigate how lightweight the approach is. In future work, we plan optimizations.

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 described. Based on a realization, the evaluation showed that the approach is lightweight, while still offering optimization potential. Although its scalability is impeded, depending on the environmental performance constraints and deployment configuration, the automatic planning may be viable for typical workflow scenarios using a limited set of microservices. Further, the evaluation showed that its automated planning offers efficiency benefits vs. manual modelling, and that its enactment performance can be on par with that of commercial BPMS systems.

One advantage we see in the Microflow approach is that the workflow (or plan) is not thoroughly adhoc and dynamic, so that validation and verification checks can be performed before execution and one is assured that an the workflow is executable as planned. For instance, if all microservices were there, but a payment service 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. However, enhanced support for verification and validation of the correctness of the microflow is still needed for users to entrust the automatic planning.

Future work includes integrating advanced verification and validation techniques, optimizing resource usage, integrating semantic support in the discovery service, transactional workflow support, support for gateways, supporting compensation and long-running processes, and enhancing the declarative and semantic support and capabilities.

ACKNOWLEDGEMENTS

The author thanks Florian Sorg for his 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.

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 Berlin Heidelberg.

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.