A Query Language for Workflow Instance Data

Philipp Baumgärtel*, Johannes Tenschert, and Richard Lenz

Institute of Computer Science 6,
University of Erlangen-Nuremberg
{philipp.baumgaertel,johannes.tenschert,richard.lenz}@fau.de

Abstract. In our simulation project ProHTA (Prospective Health Technology Assessment), we want to estimate the outcome of new medical innovations. To this end, we employ agent-based simulations that require workflow definitions with associated data about workflow instances. For example, to optimize the clinical pathways of patients with stroke we need the time and associated costs of each step in the clinical pathway. We adapt an existing conceptual model to store workflow definitions and instance data in RDF. This paper presents a query language to aggregate and query workflow instance data. That way, we support domain experts in analyzing simulation input and output. We present a heuristic algorithm for efficient query processing. Finally, we evaluate the performance of our query processing algorithm and compare it to SPARQL.

1 Introduction

ProHTA (Prospective Health Technology Assessment) is a simulation project aimed at estimating the potential of innovative healthcare technologies at a very early stage. To this end, new types of hybrid and modular simulation systems are employed to simulate the effects of new healthcare technologies [5]. For example, one of our simulations concerns mobile stroke units [5]. For stroke, the time between onset and treatment is crucial for the treatment process. Mobile stroke units enable diagnosis and treatment of stroke patients on site, therefore reducing the time between onset and treatment. Hence, in our simulation models, we pay great attention to the diagnosis and treatment workflows of stroke patients and the time of individual steps in these workflows.

Besides the problem of simulation modeling, simulation input data management is an important concern [11]. Because medical and statistical simulation data in our project stems from several heterogeneous sources, a generic and flexible conceptual model is required. We developed a multidimensional conceptual model using RDF (Resource Description Framework) to cope with the heterogeneity [2].

In our simulation project, we are already using workflow definitions in form of activity diagrams [9] as a first step towards simulation models. Therefore, it is natural to organize our simulation input data according to these workflows. To

* On behalf of the ProHTA Research Group

this end, activity diagrams need to be stored in the data management system. Then, simulation input and output data can be stored and linked to the activity diagrams.

Our simulation practitioners and domain experts want to query and analyze data. However, it is hard for scientists to write non-trivial SQL or SPARQL queries [10]. Therefore, we propose using a domain specific query language. In our stroke example, the simulation estimates the time between onset and treatment gained by implementing mobile stroke units. Then, the domain experts could use a domain specific query language to compare the simulation outcomes of different settings.

Together with our simulation experts, we identified several requirements for such a domain specific query language:

1. *Query aggregated data for a specific part of a workflow*
2. *Query data for individual steps of a workflow*
3. *Query aggregated data for the most probable paths through a workflow*

In this paper, we present a conceptual model to organize data according to workflow definitions. Additionally, we develop a domain specific query language to query and analyze the data.

2 Conceptual Model

Dumas and Hofstede [7] evaluated UML activity diagrams as a specification language for workflows. Despite some imprecise semantics, they satisfy all of our requirements. As we are already using RDF to store multidimensional data [2], we decided to store UML activity diagrams using RDF. Dolog's OWL ontology allows for storing UML state machines in RDF [6].As UML activity diagrams can be mapped to UML state machines, we use a simplified version of Dolog's ontology.

The basic elements of activity diagrams are depicted in Fig. 1. There are states and transitions, for example "A" and "C" are simple states. Transitions are depicted as arrows. Additionally, there are initial and final states. Composite states can be used to construct hierarchical structures and may contain parallel regions. Branches like "B" can be used to indicate alternatives.

We decided to omit forks and joins because the well-formedness of diagrams containing forks and joins is non-trivial [7]. However, parallel execution can still be achieved without losing expressiveness by using parallel regions in composite states. The execution of a composite state is complete when it reaches all final states in the parallel regions of the composite state. A transition between a state inside a composite state and a state on the outside interrupts the execution of the composite state. For example, the transition between "B" and "C" interrupts the execution of the composite state in Fig. 1.

We extended Dolog's ontology with optional probabilities for outgoing transitions of branches. Additionally, we can store the time and different types of costs of states and transitions. These costs consist of a name and a numerical value. In addition to the workflow definition with aggregated data, we store data

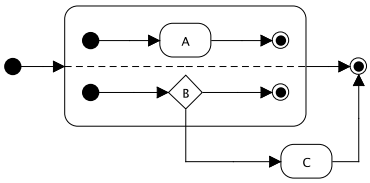


Fig. 1. UML activity diagram

about workflow instances. For example, we store data about the treatment of a patient. This fine-grained data can be used for query evaluation instead of the preaggregated data stored alongside the workflow definition.

3 Workflow Query Language

In this section, we present the main features of the WQL (Workflow Query Language). To this end, we introduce the ESTIMATE query type that can be used to aggregate time and costs in workflow definitions. We provide a formal definition of workflow data aggregation semantics online[1]. The scheme of an ESTIMATE query is shown in listing 1.1. Paths with a denoted start and end are examined in order to aggregate time and costs.

```
[ CONTEXT <URI> ]
  ESTIMATE time, costs, probability, state, path
  OF <workflow>
[ FROM <start> ] [ TO <end> ]
[ USING INSTANCE named instance ]
[ USING INSTANCES named instances for average times ]
[ USING ALL INSTANCES ]
[ WITH { variables, times, decisions } ]
[ GROUP BY state, path ]
[ ORDER BY time, costs, probability ASC/DESC ]
```

Listing 1.1. Scheme of an ESTIMATE query

To prevent the user from having to write the same prefix of URIs multiple times, the CONTEXT statement allows an abbreviated form similar to 'PREFIX' in SPARQL. ESTIMATE queries allow multiple column definitions in the ESTIMATE clause, e.g. time and different types of costs. Also, states, paths and the probability of paths can be queried if states or paths are part of the GROUP BY clause. Then, GROUP BY works like its SQL counterpart.

The optional FROM and TO clauses specify the beginning and end of the considered paths. Initial and final states of the examined workflow are the default values for FROM and TO. States can be identified either by URI or by unique

[1] `http://www6.cs.fau.de/people/philipp/wql_semantics.pdf`

names. The USING (ALL) INSTANCE(S) and WITH clauses specify workflow instances as described in Sect. 2 for the aggregation of time and costs. The ORDER BY statement triggers sorting of results with the desired sort order.

4 Query Processing

As the expressiveness of SPARQL is not sufficient, we cannot translate ESTIMATE queries to SPARQL. Therefore, we use SPARQL only to load data and activity diagrams and provide a custom query processing algorithm. In this section, we present the path-finding algorithm to process ESTIMATE queries. Since loops and decisions can produce an infinite number of paths, finding all of them is impossible. Hence we present a heuristic approach for finding the most likely paths. First, we present the basic algorithm that is not able to handle parallel sections. After that, we describe the extensions to support parallelism.

Since our path–finding algorithm is a heuristic, three parameters are provided to limit processing: the minimal probability of a path $p_{\min}$, the maximal number of results $r_{\max}$, and the maximal number of states in a path $n_{\max}$ Algorithm 1 shows a simplified version of our algorithm. In the following, we call the transitions of the activity diagram edges. The function `suitableEdges(`state`)` returns all outgoing transitions of a state excluding transitions to states with no path to a final state and transitions excluded by conditions. Therefore, we need to mark each state that reaches the end in advance.

Algorithm 1 Path-finding heuristic

```
List result, PriorityQueue pq
setCapacity(pq, r_max)
enqueue(pq, start, priority = 1)
while ¬empty(pq):
    path = pop(pq)
    edges = suitableEdges(last(path))
    ∀edge ∈ edges:
        if length(path + edge) > n_max ∨ probability(path) · probability(edge) < p_min:
            continue
        if reachEnd(path + edge):
            append(result, path + edge)
            setCapacity(pq, r_max- length(result))
        else:
            enqueue(pq, path + edge, priority = probability(path) · probability(edge))
```

The priority of a path in the priority queue is simply the probability of the path. Therefore, we try all suitable outgoing edges of the last state in the path with the highest probability. If none of our aforementioned limits is exceeded, we create new paths for each outgoing edge of the last state in that path. We append these new paths to the priority queue if they do not reach the end. Otherwise, we append them to the result list.

Our algorithm is used recursively for each parallel compound state to support nested parallel compound states. Resulting paths are no longer sequences as we have to store the states in each parallel region of the compound state. We store for each path whether it interrupts parallel execution or not. All non-interrupting paths are put into the priority queue.

For each interrupting path the paths of all other parallel regions have to be aborted at a certain point. To determine this point, we use the time of the interrupting path and search for all paths in the parallel regions with a shorter time span.

5 Evaluation

The evaluation of our heuristic is divided into two parts. First, we present an acyclic worst-case scenario and evaluate it against SPARQL property paths[2]. Afterwards, we evaluate a cyclic activity diagram and assess the precision of results. Our prototype is written in Python. SPARQL queries are processed by Fuseki 0.2.1.

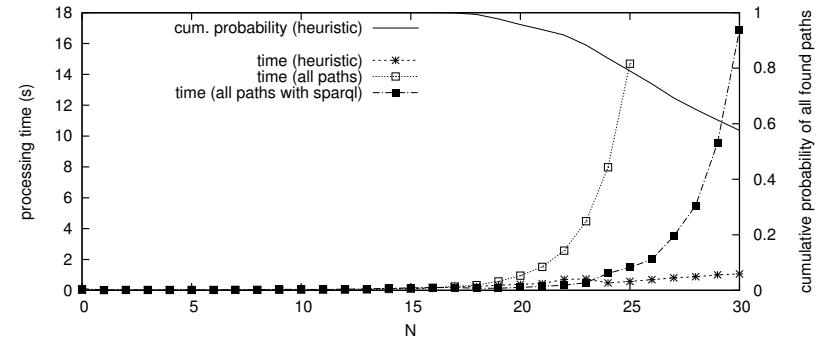
As SPARQL supports property paths to query paths in RDF graphs, we want to compare them to our path finding heuristic. These property paths are like regular expressions for RDF properties and can be used to query paths of arbitrary length in an RDF graph. SPARQL only finds matching endpoints to a property path and does not search for all paths between these two endpoints. Therefore, property paths aren't suitable for finding all paths in activity diagrams. However, for evaluating acyclic activity diagrams without parallelism with SPARQL we can simulate the search for paths. To this end, we enumerate all paths between two endpoints and store each path with separate synthetic endpoints in RDF. Then, we can write SPARQL queries using property paths that find and return the endpoints of all paths. Hence, the complexity of processing these SPARQL queries can be compared to our path finding algorithm. However, even with this extension, SPARQL property paths would not be applicable to cyclic diagrams or parallel regions.

Fig. 2(b) shows an example for a "Fibonacci activity diagram". In these synthetic diagrams with N states, each state is connected to it's two successors. We evaluated ESTIMATE queries asking for all paths that start at the initial state and end at the final state of the activity diagrams. The number of paths for each diagram with N states is the corresponding Fibonacci number, so an exponentially growing quantity of paths is generated. We define that transitions to direct successors (e.g. B to C) of a state have a probability of 90%.

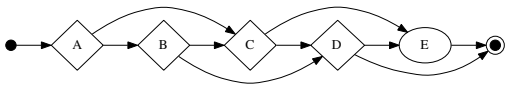
Our heuristic tries to find $r_{\max} = 2000$ most probable paths. Fig. 2(a) shows the cumulative probability of all found paths and the time to process each query. As expected, at some point the cumulative probability of the found paths decreases.

Fig. 2(a) shows that processing all possible paths is only appropriate to a certain extent. The time to find all paths using our SPARQL workaround or our

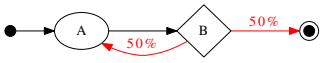
[2] `http://www.w3.org/TR/sparql11-property-paths/`



(a) Comparison of processing times



(b) Fibonacci Workflow ($N = 5$)



(c) Loops

Fig. 2. Processing times and evaluated workflows

path finding algorithm (searching for all paths) grows exponentially. SPARQL-only processing with property paths is faster than an exhaustive search for all paths with our algorithm as our implementation in python is not very fast. Despite of this, a heuristic search with our algorithm is much faster as it tries to find only the most probable paths. By defining the limits of the heuristic, the user is able to balance processing time and path coverage, which is depicted as cumulative probability of found paths.

Not all paths are equally important for results. Therefore, our heuristic tries to find the $r_{\max}$ most probable paths. For cycles, longer paths usually have less probability. Hence, in cyclic diagrams a few paths cover the most probable scenarios.

Fig. 2(c) shows an example of a loop with $cost(A) = 10$ and $cost(B) = 0$. Processing is limited by $r_{\max}$. Since this is a simple example, the precise average cost of all paths can be determined:

$$\text{cost} = \sum_{i=1}^{\infty} \frac{1}{2^i} \cdot 10i = 10 \sum_{i=1}^{\infty} i \cdot 2^{-i} = 10 \cdot 2 = 20 \tag{1}$$

By accumulating the weighted cost of the 10 most important paths, a relative error of $5.86 \cdot 10^{-3}$ remains. At 25 paths, the relative error is $4.02 \cdot 10^{-7}$ and therefore negligible. This is because for cycles longer paths usually have less probability. The query for 25 paths took $0.042s$. Hence, our heuristic is well-suited for cyclic diagrams.

6 Related Work

Awad [1] reviews existing query languages for business processes and classifies them according to three categories:

1. Querying business process definitions
2. Querying running instances of business processes
3. Querying execution history (logs) of completed business processes

The query languages in the first category are concerned with querying the structure of business processes. The approaches in the second category monitor running business processes. The third category is known as workflow or process mining. As our approach is concerned with querying the definition of a workflow, we consider the WQL to be in the first category. However, we also need to query data associated with the workflow definition. Therefore, we review some existing approaches in that category in addition to the literature already listed by Awad.

Awad [1] proposes a visual query language to search repositories of business processes for certain patterns. Deutch and Milo [4] provide a comprehensive formalism to study process modeling and querying. They use this formalism to evaluate the BPQL (Business Process Query Language) [3]. Francescomarino and Tonella [8] define the semantics of a visual query language for business processes by translating queries to SPARQL. They deal with the problem of querying paths between two elements. However, they assume each pair of elements that is connected by a path to be directly connected by an RDF property `p:isConnectedTo`. Hence, their solution is much simpler than our path finding algorithm. The aforementioned approaches including the approaches listed by Awad do not consider the data perspective and do not allow for aggregation of data. Therefore, our approach provides some unique contributions in this regard.

7 Conclusions and Future Work

In this paper, we presented a language to query aggregated data. To this end, we adapted an existing conceptual model to store workflow definitions as activity diagrams in combination with workflow instance data in RDF. We developed a heuristic query processing algorithm and evaluated it in comparison with pure SPARQL. Our evaluation shows that our heuristic algorithm is well suited for complex workflow definitions and is able to cope with cyclic graphs. The query language enables our domain experts to analyze simulation input and output data. Additionally, we can use this query language to load input data into our simulation models. Therefore, our query language renders both the input data management process and the evaluation of simulation output data more efficient.

In future work, we are planning to implement a semi-automatic transformation from activity diagrams to agent-based simulation models. Therefore, the conceptual model, aggregation formalism and query language will become more integrated with our simulation tools. We need to extend our conceptual model to support probability densities instead of fixed times and costs. Moreover, we

will investigate how to combine this conceptual model with our existing multidimensional conceptual model [2] to support data that depends on various factors, e.g. the age of a patient.

Acknowledgements. This project is supported by the German Federal Ministry of Education and Research (BMBF), project grant No. 13EX1013B.

References

1. Awad, A.: BPMN-Q: A Language to Query Business Processes, vol. 119, pp. 115–128. Citeseer (2007)
2. Baumgärtel, P., Lenz, R.: Towards data and data quality management for large scale healthcare simulations. In: Conchon, E., Correia, C., Fred, A., Gamboa, H. (eds.) Proceedings of the International Conference on Health Informatics. pp. 275–280. SciTePress - Science and Technology Publications (2012), iSBN: 978-989-8425-88-1
3. Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying business processes. In: Proceedings of the VLDB 2006 (2006)
4. Deutch, D., Milo, T.: A structural/temporal query language for business processes. Journal of Computer and System Sciences 78(2), 583 – 609 (2012)
5. Djanatliev, A., Kolominsky-Rabas, P., Hofmann, B.M., Aisenbrey, A., German, R.: Hybrid simulation approach for prospective assessment of mobile stroke units. In: SIMULTECH 2012 - Proceedings of the 2nd International Conference on Simulation and Modeling Methodologies, Technologies and Applications. pp. 357 – 366 (2012)
6. Dolog, P.: Model-driven navigation design for semantic web applications with the uml-guide. In: Matera, M., Comai, S. (eds.) Engineering Advanced Web Applications. Rinton Press (2004)
7. Dumas, M., ter Hofstede, A.: Uml activity diagrams as a workflow specification language. In: Gogolla, M., Kobryn, C. (eds.) UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science, vol. 2185, pp. 76–90. Springer Berlin / Heidelberg (2001)
8. Francescomarino, C., Tonella, P.: Crosscutting concern documentation by visual query of business processes. In: Ardagna, D., Mecella, M., Yang, J. (eds.) Business Process Management Workshops, Lecture Notes in Business Information Processing, vol. 17, pp. 18–31. Springer Berlin Heidelberg (2009)
9. Gantner-Bär, M., Djanatliev, A., Prokosch, H.U., Sedlmayr, M.: Conceptual modeling for prospective health technology assessment. In: Proceedings of the XXIV Conference of the European Federation for Medical Informatics (2012)
10. Howe, B., Cole, G., Souroush, E., Koutris, P., Key, A., Khoussainova, N., Battle, L.: Database-as-a-service for long-tail science. In: Bayard Cushing, J., French, J., Bowers, S. (eds.) Scientific and Statistical Database Management, Lecture Notes in Computer Science, vol. 6809, pp. 480–489. Springer Berlin / Heidelberg (2011)
11. Skoogh, A., Johansson, B.: A methodology for input data management in discrete event simulation projects. In: Proceedings of the 40th Conference on Winter Simulation. pp. 1727–1735. WSC '08, Winter Simulation Conference (2008)