Planning-based Generation of Service-Oriented Workflows

Carlos-Manuel López-Enríquez
Grenoble Institute of Technology
carlos.manuel.lopez@gmail.com

Víctor Cuevas-Vicenttín
University of California at Davis
victorcuevasv@gmail.com

Genoveva Vargas-Solar
CNRS, Grenoble Institute of Technology
genoveva.vargas@gmail.com

Christine Collet
Grenoble Institute of Technology
christine.collet@grenoble-inp.fr

José-Luis Zechinelli-Martini
Universidad de las Américas, Puebla
joseluis.zechinelli@udlap.mx

ABSTRACT

We address service-oriented workflows specification, enactment, and goal-based generation. Service-oriented workflows are specified in ASASEL (Abstract State Machines -ASM- Execution Language), a language based on the ASM formalism that is compatible with a workflow representation. A workflow specified in ASASEL consists of on-demand and streaming data services that produce data that are then processed by computation services. Our model and execution environment enables computation services to be built from the composition of more basic computation services. Workflows satisfying the given functional requirements are generated by adopting an approach of planning-based generation of a search space of equivalent workflows. The various workflows satisfying the functional requirements in such search space differ in respect to quality of service criteria represented by a goal. Together, our enactment engine and generation approach serve as a base for a highly flexible mechanism for managing service-oriented workflows, which can be employed to access and process information, business processes, or eScience.

Keywords

workflows, services, answer set planning, logic programming

1. INTRODUCTION

The advances in mobile computing and communication technologies bring promises of timely and inexpensive access to information, as well as of increasing interaction between people and software agents. Service-oriented architectures play a crucial role in these developments, since they enable to deal with the interoperability issues of the underlying systems.

We note in particular the proliferation of streaming and on-demand data services, which enable access to data pertaining to a multitude of domains, and possibly involving temporal and mobile properties. The availability of data services is accompanied by a democratization in access to computational resources. Nevertheless, users typically must rely on proprietary applications that delegate data processing to their backend, which makes it difficult to share resources and add new features.

We propose instead to build up systems from shared resources accessible as services via workflow specifications. Our work considers both on-demand and streaming data services in conjunction with the ability to construct composite computation services. Our language also has the advantage of being based on a solid formalism and having a fully congruent visual representation, also we consider it to be more user friendly than XML-based languages. In addition, we propose a workflow generation framework based on planning techniques to meet quality of service goals. An implementation of this framework is presented covering data-centric workflows.

2. EXAMPLE SCENARIO

We introduce our approach with a prototypical Friend Finder application, although our approach can be applied to a broader range of applications. Consider a scenario in which multiple users are in an urban area carrying GPS-enabled mobile devices that periodically transmit their location; furthermore, they have agreed to share some of their personal information. A user in this scenario may want to *Find friends which are no more than 3 km away from her, which are over 21 years old and that are interested in art.*

Data services produce data in one of two ways: on-demand in response to a given request, or continuously as a data stream. In either case, the data service exposes an interface, composed of several operations and supported by standardized protocols. The JavaScript Object Notation[1] is used to represent the data. Accordingly, objects are built from atomic values, nested tuples, and lists.

For instance, in our scenario the users' location is available by a stream data service with the interface

[1]JSON http://www.json.org/

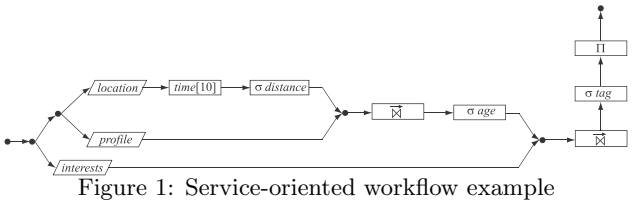


Figure 1: Service-oriented workflow example

`subscribe()` → ⌈`location` : ⟨`nickname`, `coor`⟩⌉

consisting of a subscription operation that after invocation will produce a stream of location tuples, each with a nickname that identifies the user and his/her coordinates. The rest of the data is produced by the next two on-demand data services, each represented by a single operation

`profile(nickname)` → `person` : ⟨`age`, `gender`, `email`⟩
`interests(nickname)` → [`s_tag` : ⟨`tag`, `score`⟩]

The first provides a single person tuple denoting a profile of the user, once given a request represented by his/her nickname. The second produces, given the nickname as well, a list of s tag tuples denoting the interests of the user by scored tags (*e.g.* 'music' with 8.5).

In order to obtain the desired result we need to give to it an executable form, in our case a workflow of activities implementing a service coordination. Workflows are built by the parallel and sequential composition of activities that are bound to data and computation services; the first provide the data, while the latter process them as required.

The workflow is specified textually as an Abstract State Machine (ASM) [3], which can be represented as a series-parallel graph. Such representation of a service coordination corresponding to our example application is given in Figure 1. It includes the location, profile, and interests data services, as well as computation services for various relational operations such as selections, joins, and a window bounding the location stream.

3. COMPUTATION SERVICES

Two kinds of computation services form part of our approach: simple computation services and composite computation services specified in the ASASEL language.

Simple computation services These involve a single service operation invocation to process data (see Figure 2a). The distance computation service relies on a `geo-distance` service, which provides the capability to calculate the geographical distance between two points, e.g., by Vincenty's formula.

Composite computation services These process data by multiple operation invocations, possibly from different services, and often also by the manipulation of local data. These tasks are organized in a service coordination specified in the ASASEL language and represented as a workflow, following a model in which we add data items as well as conditional and iteration constructs to our basic parallel and sequential composition workflow model illustrated in Figure 1.

Figure 2b depicts a composite computation service, it evaluates the join operator based on the symmetric hash-join algorithm and two instances of a stateful `hash-index` service. Several interrelated operation invocations on both service instances, as well as reads and updates on local data items, are used to find the tuple matches that form part of the join result.

4. SERVICES AND WORKFLOW MODEL

Next we describe how we model services and their associated workflows, along with the associated quality of service objectives, for our planning-based generation approach.

4.1 Workflow Quality of Service

In our framework users express their application needs along their QoS preferences. For instance, consider the application request "*Where are my friends at this moment?*" with the QoS preferences "*Privilege the execution price over the execution time and this in turn on the battery consumption*".

Services have QoS measures associated to their invocation and thus the overall execution of a workflow has an associated cost that is the aggregation of the QoS measures. For the service-oriented workflows under consideration we assume costs along three dimensions: execution time, execution price, and battery consumption.

A potentially large finite set of workflows produce the desired results but only a subset from these satisfies the cost requirements of the user. This is a combinatory optimization problem of minimizing the cost of a workflow W that coordinates a subset of activities A through its control-flow. In our workflow generation apporach we limit ourselves to workflows accessing on-demand data only. For example, "*What are the interests of my friend Joe*", where only on-demand data are considered.

4.2 Data and computing services

We characterize service APIs with the following structure

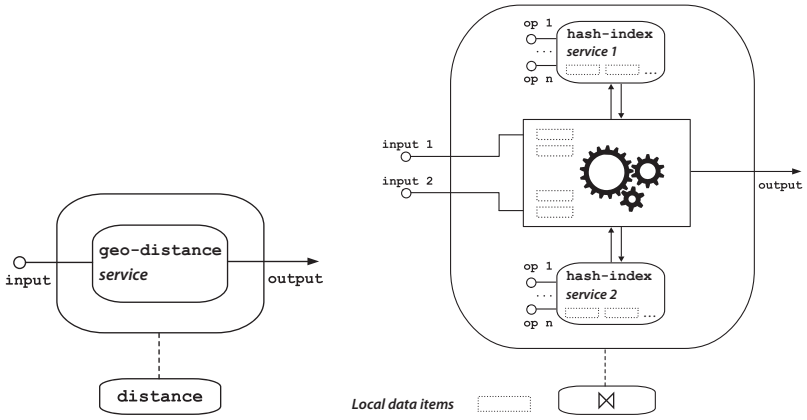
`s_name:m_name(`$\mathtt{B}_1$`:`$\mathtt{T}_1$`?,...,`$\mathtt{B}_m$`:`$\mathtt{T}_m$`?,`$\mathtt{F}_1$`:T'`$_1$`!,...,`$\mathtt{F}_n$`:T'`$_n$`!)`

where the service identified by `s_name` has a method named `m_name` with `m` bound parameters, each with an unique name $\mathtt{B}_i$ within the method header and associated with a type $\mathtt{T}_i$. Analogously there are `n` free parameters similarly named and typed. Bound and free parameters will be identified by question and exclamation marks, respectively.

From this general notion of service, we distinguish two service specializations as discussed earlier. (1) On-demand data services provide data in a request-response fashion through synchronous data operations. (2) Computing services provide a set of methods to perform operations over data (*e.g.*, distance estimation, ordering, correlation).

4.3 Service-oriented workflow

A service-oriented workflow represents a control flow among a set of activities. Each activity is a program that calls a data service method or a computing service method. The control flow is defined by the *sequence* and *parallel* workflow composition operators.



(a) Simple service (b) Composite service

Figure 2: Computation services

A service-oriented workflow W is modeled as a directed acyclic graph $W=(V, E, init, finish, A, O)$ where:

V	is a set of vertices
$E \subseteq V \times V$	is a set of edges
$A \subseteq V$	is a set of activities
$\{init, finish\} \subseteq A$	are the initial and final activities of W
$O \subseteq V$	is a set of workflow composition operators $\{seq_1, ..., seq_m, par_1, ..., par_n\}$

.

There are three activity specializations: the *activity* performs a service method invocation and always has a previous and a next activity, the *init* activity has no previous activity and its only goal is to launch the first *activity* of query workflow, the *finish* activity has no next activity and it stops the query workflow execution after the last *activity*.

The compositions of activities $E \subseteq V \times V$ are defined by two workflow operators: sequential and parallel composition. A sequential composition *seq* represents the total order of a pair of activities a_1, a_2 such that $e_1 = (a_1, a_2)$. It means that the second activity a_2 is executed after a_1. A parallel composition *par* represents the partial order of activities a_1, a_2, a_3, a_4 such that $e_1 = (a_1, a_2)$, $e_2 = (a_1, a_3)$, $e_3 = (a_2, a_4)$, $e_3 = (a_3, a_4)$. It means that a_2 and a_3 are independent from each other and are executed in parallel after a_1 and before a_4.

5. WORKFLOW GENERATION

Workflow generation is the process of enumerating all the valid workflows in order to have a search space for choosing a subset that satisfies the user's preferences. The generation is subject to query workflow constraints for composing activities. We model these constraints as action rules in the language DLV-K[2] that allows the sequential or parallel execution of query workflow activities.

In DLV-K, planning problems have a set of facts that represent the problem domain named background knowledge. The facts are predicates of static knowledge and are the input of the planning problem. Planning problems are modeled as state machines described by a set of fluents and a set of actions. A fluent is a property of an object in the world and is part of the states of the world [1]. Fluents may be true, false or unknown. An action is executable if a precondition holds in the current state. Once an action is executed, the fluents and thus the state of the plan are modified. The action rules define the subset of fluents that must be held before the execution of an action (*i.e.* preconditions) and the subset of fluents to be held after the execution (*i.e.* post-conditions). Finally, a goal is a set of fluents that must be reached at the end of plan. A goal is expressed by the conjunction of fluents and by a plan length $l \in \mathbb{Z}^+$.

[2]http://www.dbai.tuwien.ac.at/proj/dlv/k

Workflow	Planning problem
APIs, operations	Facts (background knowledge)
Workflow states	Fluents
Workflow activities	Actions
Result delivery	Goal: `finished?`$(l \in \mathbb{Z}^+)$

Table 1: Mapping to a planning problem for a workflow

The mapping from workflow generation to a planning problem is direct, as shown in Table 1. The APIs and the application requirements (operations on data) are modeled as facts of the background knowledge. The execution state of a workflow is modeled as fluents and workflow activities as actions.

Next we show, through a simple example, how we represent the background knowledge for workflow generation. Afterwards, we show how the workflow state and activities are expressed in DLV-K.

5.1 Background knowledge

The background knowledge is the input for workflow generation and it is represented by a set of facts. The facts are two-folded in (1) service interface representation and (2) operations representation.

Service interface Consider the following service methods:

`p:profl(str:nickname?,int:age!,str:sex!,str:email!)`

This method provides the user profile formed by `age, sex` and `email` of a given user with a `nickname`.

`i:interests(str:nickname?,str:tag!,real:score!)`

This method provides the interests of a given user identified by a `nickname`. An interest is represented by a `tag` and has a `score` that quantifies the interest over the tag.

The service interface and its methods are described by the facts `service_interface/1` and the `method/2` respectively. We distinguish between bound and free parameters with the facts `bound_p/4` and `free_p/4`. The normal form of a parameter is stated by `parameter/4` and this rule guarantees that the parameter name is unique in the context of the service method.

```
service_intereface(profile).                                    1
method(profile,      profile).                                  2
bound_p(profile, profile, nickname, string).                    3
free_p(   profile, profile, age,      int).                     4
free_p(   profile, profile, sex,      string).                  5
free_p(   profile, profile, email,    string).                  6
                                                                7
service_interface(interests).                                   8
method(interests,    interests).                                9
bound_p(interests, interests, nickname, string).                10
free_p(   interests, interests, tag,      string).              11
free_p(   interests, interests, score,    real).                12
                                                                13
parameter(DSN,ON,PN,T):- bound_p(DSN,ON,PN,T).                  14
parameter(DSN,ON,PN,T):- free_p(DSN,ON,PN,T).                   15
```

Operations Now consider the application requirement "*What are the interests of my friend Joe*" that is represented by the operation facts:

```
project_(p,nickname,n).                                         1
project_(i,score,s).                                            2
project_(i,tag,t).                                              3
retrieve_(profile,profile,p).                                   4
retrieve_(interests,interests,i).                               5
select_(p, nickname).                                           6
join_(p,nickname,i,nickname).                                   7
```

This requirement expresses the need of data over the methods `p:profl` and `i:interests` methods, the nickname of the profile is filtered and correlated with the nickname of interests. Finally, the parameters nickname, score and tag are projected. Observe that the selection over the nickname attribute is indicated only in intention because the equality operators are not significant for the workflow generation.

5.2 Workflow activities

Workflow activities are represented as actions in DLV-K. In general, the activities are predicates that hold if their facts from background knowledge are true. There are also activities that are independent from facts.

init and finish These activities have the special purpose to `init` and `finish` the workflow execution accordingly with our model in subsection 4.3. Thus the semantics is not associated with the application and there is no dependency with the background knowledge.

on-demand This activity establishes a connection with a data service method and requires a method of a service and the expressed need of the user to be fulfilled.

```
on_demand(DS) requires method(DSN,ON), retrieve_(               1
    DSN,ON,DS).
```

bind_selection This activity invokes and retrieves data from a service method. The invocation is done by a given bound parameter valid in the service interface definition. The workflow must express which data are required from this service method in respect to a selected bound parameter.

```
bind_selection(DS,BP) requires method(DSN,ON),                  1
        retrieve_(DSN,ON,DS), bound_p(DSN,ON,BP,                2
            _), select_(DS,BP).
```

bind_join The required correlation in the workflow is performed by this activity. Correlation is binary between data from two service methods on a parameter from each one. The parameter from the outer method must be bound. This activity is analogous to `bind_selection` but this one takes the value of a bound parameter from the output of another method (*i.e.* the inner method).

```
bind_join(DS1,P1,DS2,BP2)                                       1
   requires method(DSN1,ON1), retrieve_(DSN1,ON1,               2
       DS1),
            parameter(DSN1,ON1,P1,_),                           3
            method(DSN2,ON2), retrieve_(DSN2,ON2,               4
                DS2),
            bound_p(DSN2,ON2,BP2,_),                            5
            join_(DS1,P1,DS2,BP2).                              6
```

select The select activity performs the filtering over a parameter of a required service method.

```
select(DS,P) requires method(DSN,ON),                           1
                      retrieve_(DSN,ON,DS),                     2
                      parameter(DSN,ON,P,_),                    3
                      select_(DS,P).                            4
```

project This activity projects a parameter of a service method.

```
project(DS,P) requires project_(DS,P,_).                        1
```

The semantics of activities described above is completed with constraints that define their pre-conditions and post-conditions.

5.3 Workflow constraints

These constraints define the conditions associated to the execution of activities. A condition is a state of knowledge modifiable by the execution of the activities. Conditions also define when the activities can be executed. Hence, we talk about pre-conditions and post-conditions of activities. A state is composed by a set of fluents that occur during the execution of activities. The rules that define when a fluent occurs can be static or dynamic. Static rules are those that occur given the truth-value of a subset of fluents, and dynamic rules occur given the truth-value of a subset of fluents and after the execution of an activity.

Below we present the constraints that describe how the activities can be performed and how a query plan should be constructed.

init and finish The first executable activity in a query workflow is **init**. This activity has no previous activity, thus

its pre-condition is that the query workflow is not `initiated` and produces a new state `initiated`. The last activity is `finish` and there is no other activity executed after this one. The pre-condition to execute `finish` is that there is not evidence that the query workflow is `finished` and the data is already `delivered` (See `output` activity below for details about `delivered`). The post-condition of `finish` is `finished` and this is always the goal for the generation.

```
executable init if   -initiated.                          1
caused     initiated after init.                          2
executable finish if not finished, delivered.             3
caused     finished after finish.                         4
```

on-demand Once initiated the plan, the data services must be `connected(DS)`. This fluent is produced by the execution of `on_demand(DS)` activity.

```
executable on_demand(DS) if initiated.                    1
caused     connected(DS) after on_demand(DS).             2
```

During the execution, all data services must be connected. Therefore, there is a fluent `all_connected` that is false if there is not evidence that a data service is connected. Otherwise, it is true.

```
caused -all_connected if not connected(DS).               1
caused  all_connected if not -all_connected.              2
```

bind_selection One of the activities that implements data retrieval is bind selection. It is only executable if there is not evidence that the data service `DS` has already been retrieved and if there is a connection with `DS`. Once the bind selection is executed, the fluent `retrieved(DS)` is true. `retrieved(DS)` is an inertial fluent, thus the re-execution is not possible.

```
executable bind_selection(DS,BP)                          1
       if not retrieved(DS), connected(DS).               2
caused retrieved(DS) after bind_selection(DS,BP).         3
```

selection The filtering of data is done by the `selection` activity. It is executable if there is not evidence that the parameter `P` of `DS` has already been selected. There is also need that the data from `DS` has been retrieved and obviously the selection `select_(DS,P)` must be part of the workflow. The execution of the selection makes the fluent true `selected(DS,P)` and it is inertial, so the re-execution of the selection `select(DS,P)` is not possible.

```
executable select(DS,P) if not selected(DS,P),            1
                           retrieved(DS),                 2
                              select_(DS,P).
caused     selected(DS,P) after select(DS,P).             3
```

In order to be aware of the state of selection over all the required parameters of a method `DS`, the all_selected_from(DSOName) becomes true if there is no other parameter pending to be selected.

```
caused -all_selected_from(DS) if not selected(DS,P 1
    ), select_(DS,P).
caused  all_selected_from(DS) if not -                    2
    all_selected_from(DS), retrieved(DS).
```

Analogously, all parameters from all data services must be selected when they are required. Therefore, there is the fluent `all_selected`. This fluent is true if there is no other method `DS` pending to be selected.

```
caused -all_selected if -all_selected_from(DS),           1
    select_(DS,P).
caused -all_selected if -all_selected_from(DS),           2
    not select_(DS,P), parameter(DSN,ON,P,_),
    retrieve_(DSN,ON,DS).
caused  all_selected if not -all_selected.                3
```

projection This activity is executable if there is not evidence that the parameter `P` of `DSOName` has been projected. There is also need that the data from `DSOName` be retrieved. The execution of projection makes the fluent `projected(DSOName,P)` true and it is inertial, thus the re-execution of the projection is not possible.

```
executable project(DS,P) if not projected(DS,P),          1
    retrieved(DS), project_(DS,P,_).
caused     projected(DS,P) after project(DS,P).           2
```

The projected fluent is true once the action `project(DS,P)` is done. During the query workflow execution, all the required parameters must be projected. For `DS` grain the fluent `all_projected_from(DS)` is true if there is no other parameter from `DS` pending to be projected. For the entire query, the fluent `all_projected` is true if there is no other `DS` pending to be projected.

```
caused -all_projected_from(DS) if not projected(DS 1
    ,P), project_(DS,P,_).
caused  all_projected_from(DS) if not -                   2
    all_projected_from(DS) after project(DS,P).
                                                          3
caused -all_projected if -all_projected_from(DS),         4
    project_(DS,P,_).
caused  all_projected if not -all_projected.              5
```

output Once the operations are processed, data must be delivered by the activity `output`. In order to model this precondition, the fluent `query_processed` is true if all the other possible activities have been processed. Otherwise, the fluent is `-query_processed`.

```
caused -query_processed if not all_connected.             1
caused -query_processed if not all_retrieved.             2
caused -query_processed if not all_selected.              3
caused -query_processed if not all_projected.             4
caused  query_processed if not -query_processed.          5
```

Once the query is processed, the `output` activity delivers the result and the fluent `delivered` becomes true.

```
executable output if query_processed, not                 1
    delivered.
caused     delivered after output.                        2
```

6. EXPERIMENTS

We performed experiments with the various sets of operation instances described below. The objective of the experiments is to measure the size of the search space of query workflows given a fixed number of activities and a goal to define the length of the workflow.

6.1 Configuration

In order to test the performance of workflow generation, we have defined four test cases. All are based on the example of

Query	Data operator		Description	# activities
	select_(p,age)	join_(p,nickname,i.nickname)		
[Q1]	✗	✗	no additional operators	11
[Q2]	✓	✗	+unary data operator	12
[Q3]	✗	✓	+binary operator	13
[Q4]	✓	✓	+both unary and binary operator	14

Table 2: Workflow configuration

l	Queries			
	Q1	**Q2**	**Q3**	**Q4**
6	0	0	0	0
7	4	4	20	20
8	62	128	598	1192
9	278	956	5062	15820
10	578	3068	19822	90100
11	718	5368	43698	277800
12	718	6268	62358	525476
13	718	6268	62358	749840
14	718	6268	62358	749840

Table 3: Search space growth respect to the plan length

subsection 5.1. In order to make the generation more complex we add two data operations. The resulting workflows are configured as follows:

Another dimension for the test is the max length of plans. The generated plans depend on a required plan length. In DLV-K, one must specifies the desired length that determines the size of the search space of query workflows and thus the required time for the generation.

In the tests we used lengths from 6 to 14, and we will measure (1)the length of the generated plans, (2)the size of the search space, and (3) the execution time of the generation.

6.2 Results

The data retrieved from the experiments show (1) the behavior of the search space growth during the generation and (2) the time required for generating all the search space.

Each workflow requires a fixed number of activities accordingly with the number of data operators required to process its result. Given that size of query plans are subject to the length l, there are workflows that require a largest l than others. For example, $Q1$ with 11 activities, requires at least an $l = 7$ to get workflows with the must possible parallel compositions, and at most $l = 11$ to get completely sequential compositions. For space reasons we invite the reader to visit the URLs http://goo.gl/XKZuL and http://goo.gl/z4iu3 to appreciate examples of query workflows of 7-length and 11-length respectively. The growth of search space is shown in Table 3 and, as was expected, the size of search space tends to be stable once the maximum length is reached. For example, $Q1$ reach the maximum length with $l = 11$. This is analogous for $Q2$, $Q3$, and $Q4$. Nevertheless the size of search space for each one is considerably bigger than the others less complex. It can be noted that the search spaces have an exponential growth until the max length.

Besides the size of search space, the time for processing the workflow generation is also exponential and it is not feasible to generate completely the search space in the context of query optimization. Thus, this enumeration must be done implicitly in order to avoid the combinatorial explosion.

7. SYSTEM IMPLEMENTATION

The ASASEL system was developed on the Java platform. Workflows are entered textually via a GUI illustrated in Figure 3. For a given ASASEL service coordination, the GUI (relying on the JGraph[3] library) provides the user a workflow visualization. Data services are represented in yellow whereas computation services are represented in blue, both with their corresponding labels.

The enactment of a workflow is enabled by two main components. First, a scheduler determines which service is executed at a given time according to a predefined policy. Second, composite services are executed by an interpreter that implements the full ASASEL language. Computation service workflows can also be visualized through the GUI, as shown at the right part of the screenshot in Figure 3. The interpreter was developed using the ANTLR[4] parser generator.

During the execution of a workflow, data flows from the data services to several computation services via queues, as determined by the ASASEL specification. Computation services can be specified to output data in textual form in the GUI or to transmit it to another application. For instance, in our example application we output as a result data stream that denotes the tuples that are added and the tuples that are removed from the result dataset.

We developed a set of basic computation services that are used to build the operations for our example application. These services run on a Tomcat container supported by the JAX-WS reference implementation [5], which enables to create stateful services.

Additionally, we implemented two test scenarios and their corresponding data services to use with our computation services. The first one is the location-based application introduced in Section 2. The second scenario is an adaptation of the NEXMark benchmark[6] for XML stream query processing which we employed to obtain performance measurements. In brief, the measurements indicated a tolerable overhead for the use of services, which we consider outweighed by the advantages.

8. RELATED WORK

[3]http://www.jgraph.com/
[4]http://www.antlr.org/
[5]https://jax-ws.dev.java.net/
[6]http://datalab.cs.pdx.edu/niagara/NEXMark/

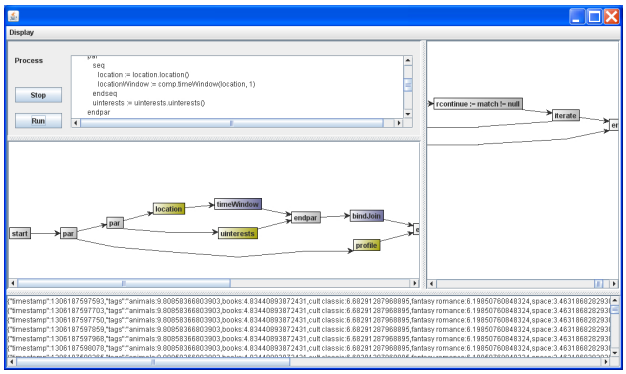


Figure 3: Caption of the ASASEL GUI

The optimization of workflows involving data services must tackle challenges such as autonomy, non-durability of data, absence of fine grain data statistics, etc. For instance, the evaluation of queries over Web services presented in [4] proposes an optimization approach by ordering the service calls in a pipelined fashion and by tunning the size of data. However, the control over data size (i.e. data chunks) and selectivity statistics is a strong assumption because of the autonomy of services and the absence of data statistics in many real scenarios.

Another aspect to consider during the optimization is the selection of the service instances, which represents a factor for the service coordination cost. In [2, 5] the optimal selection of services with a multidimensional cost is proposed for optimizing the complete service coordination. This selection is done by solving a multi-objective assignation problem for a set of abstract services. However, the possibility to modify the order of service invocations is not considered and this is a key issue in service-oriented workflow optimization.

9. CONCLUSIONS AND FUTURE WORK

In this paper we focused on the generation of the search space of data-centric workflows in service-oriented environments. We proposed a set of constraints modeled in an action language, specifically DLV-K, in order to characterize the construction of workflows with sequential and parallel compositions.This work is envisaged to be a foundation for incorporating a full cost model, leading to the selection of the most suitable workflow w.r.t. user's preferences. This is crucial because in the absence of constraints the search space is in the order of $n!$ permutations in the number of activities required for obtaining the result.

10. REFERENCES

[1] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[2] D. Claro Barreiro, P. Albers, and J.-k. Hao. Selecting web services for optimal composition. In *International Conference on Web Services (ICWS05)*, 2005.

[3] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. B*backslash*"orger, editor, *Specification and Validation Methods*, chapter Evolving A, pages 9—-36. Oxford University Press, Inc., New York, NY, USA, 1995.

[4] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. *VLDB '06, Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006.

[5] H. Wada, P. Champrasert, J. Suzuki, and K. Oba. Multiobjective Optimization of SLA-Aware Service Composition. In *2008 IEEE Congress on Services - Part I*, pages 368–375. IEEE, July 2008.