A Planning-Based Service Composition Approach for Data-Centric Workflows

Carlos-Manuel López-Enríquez[2,4], Víctor Cuevas-Vicenttín[3], Genoveva Vargas-Solar[1,2], Christine Collet[2], and José-Luis Zechinelli-Martini[4]

[1] CNRS
[2] Grenoble Institute of Technology
BP. 72, 38402, Saint Martin d'Hères Cedex, France
[3] Universidad Panamericana Campus Guadalajara
Calzada Circunvalación Poniente No. 49, Ciudad Granja 45010
Zapopan, Jalisco, Mexico
[4] Universidad de las Américas Puebla
Exhacienda Sta. Catarina Mártir s/n, 72820
San Andrés Cholula, Puebla, Mexico
carlos.manuel.lopez@gmail.com,victorcuevasv@gmail.com,genoveva.vargas@imag.fr,
christine.collet@grenoble-inp.fr,joseluis.zechinelli@udlap.mx

Abstract. This paper presents a planning-based approach for the enumeration of alternative data-centric workflows specified in ASASEL (Abstract State mAchineS Execution Language), which define the coordination of data and computation services for satisfying data requirements. The optimization of data-centric workflows is associated to the exploration of the parallelization of the workflow activities. We address the exploration of parallelism formalizing the enumeration problem in the DLV-k language. Together, our ASASEL language and enactment engine along with our enumeration approach provide the foundation for a highly flexible mechanism for managing data-centric workflows.

Keywords: workflows, services, answer set planning, logic programming

1 Introduction

We witness a proliferation of streaming and on-demand data services for accessing data pertaining to a multitude of domains, 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.

Therefore we propose ASASEL (Abstract State Machines Execution Language) to build up systems from shared resources accessible as services via data-centric workflow specifications. Our work considers both on-demand and streaming data services producing complex values, operations on these data, and the

ability to construct composite computation services to process them. In addition, we propose a workflow transformation framework based on planning techniques to meet quality of service goals. We present a concrete implementation of this framework covering parallelization through the workflow structure.

The remainder of this paper is structured as follows. Section 2 presents our workflow model and language, while Section 3 introduces our complex values data model and related operations. In Section 4 we present a planning-based workflow transformation framework, whose experimental results are presented in Section 5. Our system implementation is discussed in Section 6. Section 7 discusses related work. Finally, we present our conclusions and discuss future work in Section 8. The material in Section 2 is also presented in [**?**], which however does not cover the contents of Section 3 onwards.

2 Data-Centric Workflows

Consider a Friend Finder application in which multiple users carry mobile devices that periodically transmit their location. Assume that they have agreed to share some of their personal information. A user in this scenario may want to *Find friends recently located no more than 3 km away from me, 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 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

$\texttt{subscribe}() \rightarrow \lceil \texttt{location} : \langle \texttt{nickname}, \texttt{coor} \rangle \rceil$

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

$\texttt{profile}(\texttt{nickname}) \rightarrow \texttt{person} : \langle \texttt{age}, \texttt{sex}, \texttt{email} \rangle$

$\texttt{interests}(\texttt{nickname}) \rightarrow [\texttt{s\_tag} : \langle \texttt{tag}, \texttt{score} \rangle]$

The first provides a single person tuple denoting a profile of the user, once given a request represented by 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.

2.1 Workflow Model

The workflow is specified as an Abstract State Machine (ASM) [4], which can be represented as a series-parallel graph. The ASM specification of the service coordination corresponding to our example application is presented in Listing 1.1, while its workflow representation 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 time-based window bounding the location stream to recent data (e.g. location notifications obtained within the last 10 minutes).

```
seq
   par
      seq
         par
            seq
               location := l.location()
               locWin := comp.timeWin(location,10)
                distSel := comp.funCallSel(locWin,
                 d.dist(lat,lon,48.85,2.29)<3.0 )
            endseq
            profile := profile.profile()
         endpar
         lp := comp.bindJoin(distSel,profile,nickname=nickname)
         ageSel := comp.selection(lp,age > 21)
      endseq
      interests := i.interests()
   endpar
   lip := comp.bindJoin(lp,interests,nickname=nickname)
   tagSel := comp.selection(lip,tag='art')
   output := comp.output(tagSel)
endseq
```

Listing 1.1: ASM specification for example application

Fig. 1: Data-centric workflow for example application

A workflow W is modeled as a directed acyclic graph $W = (V, E, in, out, A, C)$ where:

$$
\begin{aligned}
V &\text{ is a set of vertices} \\
E \subseteq V \times V &\text{ is a set of edges} \\
A \subseteq V &\text{ is a set of activities} \\
\{in, out\} \subseteq A &\text{ are the initial and final activities of } W \\
C \subseteq V &\text{ is a set of composition operators } \{par_1, ..., par_n\}
\end{aligned}
$$

.

There are three types of vertices: *activities* perform a service method invocation and always have ancestor and descendant vertices, *in* vertices have no ancestors and their only goal is to launch the first *activity* of the workflow, *out* vertices have no descendants and stop the workflow execution after the last *activity*. A series of construction rules enable to generate a workflow graph from a given ASM, which are detailed in [2].

2.2 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 involve a single service operation invocation to process data. For instance, a distance computation service that 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 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.

The specification of a time-based window composite service in ASASEL is presented in Listing 1.2, based on a simple `calendar-queue` service. It has a corresponding workflow representation as detailed in [2].

```
if( ctl_state = 'active')
  seq
     inTuple := readTuple()
     if(inTuple = nil)
        skip
     else
        seq
           oldTuple := cq.peekFirst()
           iterate(oldTuple != nil)
              if(oldTuple.ts + range < inTuple.ts)
                 seq
                    oldTuple.sign := -1
                    oldTuple.ts := oldTuple.ts + range
                    output(oldTuple)
                    cq.removeFirst()
                    oldTuple := cq.peekFirst()
                 endseq
           pq.enqueue(inTuple)
           output(inTuple)
        endseq
  endseq
```

Listing 1.2: ASM specification for the time-based window

3 Complex Values Data Model

Our workflow model is complemented by a data model consisting of complex values and operations to flexibly manipulate them. Due to space restrictions we only specify two representative operators while the full specification and semantics of the model is given in [2]. Concretely, we first define complex values and then present a recursive operator and a nesting operator over them.

The set $\mathbf{T}$ of all complex value types over a set $\mathbb{A}$ of type names is defined inductively as follows.

1. if D is a domain, then $A : D$ is an atomic type named A, where $A \in \mathbb{A}$;

2. if $\hat{t}$ is a type, then $A : \{\hat{t}\}$ is a set type named A;
3. if $\hat{t}_1, ..., \hat{t}_n$ are types with distinct names, then $A : \langle \hat{t}_1, ..., \hat{t}_n \rangle$ is a tuple type named A and each $\hat{t}_i$ is an attribute type.

3.1 Recursive complex value operators

Inspired in the traditional relational operators, they apply to complex values in a recursive manner; meaning that through an expression it is possible to apply the operator to structures nested within a complex value. In particular, we present next the specification of the projection operator.

Projection Enables to retrieve certain data elements in a complex value instance. Such data elements may be nested and multivalued. The data elements to retrieve are specified in a (possibly recursive) projection expression π_{exp}, which is applied to the input complex value instance s.

- *Notation*: $\pi_{exp}(s)$
  Projection expressions π_{exp} are constructed as follows, we use A to represent type names that occur in the complex value instance

$$\begin{array}{lll} \pi_{exp} & ::= & \pi \ (\ list\ ) \\ list & ::= & term \ \mid \ term\ ,\ list \\ term & ::= & A \ \mid \ \pi_{exp} \end{array}$$

- *Operation type*: $\pi : \hat{t} \rightarrow \hat{t'}$, where $\hat{t'}$ is defined below
- *Semantics*: $\pi_{exp}(s)$ is defined as follows.
  First, we define the function $eval(A : v, L)$, where $A : v$ is a tuple complex value of the form $A : \langle ..., A' : v', ... \rangle$ and L an expression term (as defined by the notation third rule above).
  1. If L is of the form A' then $eval(A : v, L) = A' : v'$
  2. If L is of the form $\pi(A', L'_1, ..., L'_n)$ then $eval(A : v, L) = \pi(A', L'_1, ..., L'_n)(A' : v')$

  The value of $\pi_{exp}(s)$ is then given by
  1. If $s = A : \langle A_1 : v_1, ..., A_n : v_n \rangle = A : v$, i.e. s is a tuple complex value, and $\pi_{exp} = \pi(A, L_1, ..., L_n)$, then
     $\pi_{exp}(s) = A : \langle eval(A : v, L_1), ..., eval(A : v, L_n) \rangle$ and
     $\hat{t'}$ is $A : \langle type(eval(A : v, L_1)), ..., type(eval(A : v, L_n)) \rangle$
  2. If $s = A : \{A' : v_1, ..., A' : v_m\}$, i.e. s is a set complex value, and $\pi_{exp} = \pi(A, \pi_{exp'})$ with $\pi_{exp'}$ of the form $\pi(A', L'_1, ..., L'_n)$, then
     $\pi_{exp}(s) = A : \{\pi_{exp'}(A' : v_i) | A' : v_i \in val(s)\}$ and
     $\hat{t'}$ is $A : \{type(\pi_{exp'}(A' : v_j))\}$ for an arbitrary $A' : v_j \in val(s)$

Consider the following complex value

s = person:⟨ sex:'M', nick:'Charles', email:'charles@gmail.com', age:40,

interests:{*stag:*⟨ *tag:'art', score:6.5* ⟩, *stag:*⟨ *tag:'sports', score:7.5* ⟩}⟩

The expression $\pi(person, nick, age, \pi(interests, \pi(stag, score)))(s)$ produces the value

person:⟨ *nick:'Charles', age:40,interests:*{*stag:*⟨ *score:6.5* ⟩, *stag:*⟨*score:7.5* ⟩}⟩

3.2 Nesting and unnesting operations

These operators take into consideration common values occurring in several tuples, therefore facilitating grouping or ungrouping them (which gives the operators their names). The specification of the group operator is presented next.

Group. Intuitively, grouping a set of tuple complex values R over a set of attributes X implies aggregating the tuples that are equal in all attributes except those in X to create a single tuple. This tuple will contain a new set attribute with new tuples containing all of the X-values of the aggregated input tuples. This set attribute is given a new name, as are the tuples built from the X attributes that are contained in it; both of which are specified in the group expression.

- *Notation*: $group_{exp}(R)$
  Group expressions exp are constructed as follows, we use A to represent the type names that occur in the complex value instances, and B and B' to represent the new names of the grouped tuples set and its constituent tuples, respectively

  $$\begin{array}{lll} exp & ::= & group\ (A,\ B : list\ [B']\ ) \\ list & ::= & A\ \mid\ A\ ,\ list \end{array}$$

- *Operation type*:
  $group: \{A : \langle \hat{a}_1, ..., \hat{a}_m, \hat{b}_1, ..., \hat{b}_n \rangle\} \rightarrow \{A : \langle \hat{a}_1, ..., \hat{a}_m, B : \{B' : \langle \hat{b}_1, ..., \hat{b}_n \rangle\}\rangle\}$
- *Semantics*:
  $group_{exp}(R) =$
  $\{A : \langle A_1 : v_1, ..., A_m : v_m, B : w \rangle \mid ($
  $\exists t \in R \mid \forall_{i \mid 1 \leq i \leq m}\ t.A_i = v_i \wedge w =$
  $\{B' : \langle B_1 : u_1, ..., B_n : u_n \rangle | A : \langle A_1 : v'_1, ..., A_m : v'_m, B_1 : u_1, ..., B_n : u_n \rangle$
  $\{B' : \langle B_1 : u_1, ..., B_n : u_n \rangle | A : \langle A_1 : v'_1, ..., A_m : v'_m, B_1 : u_1, ..., B_n : u_n \rangle$
  $\in R \wedge \forall_{i \mid 1 \leq i \leq m}\ t.A_i = v'_i\}$
  $)\ \}$
  where all values $A_i : v_i$ and $A_i : v'_i$ are of type $\hat{a}_i$ and all values $B_i : u_i$ are of type $\hat{b}_i$.

  Consider the following set of tuple complex values

  $R = \{$ *person:*⟨ *nickname:'Bob', tag:'sports', score:6.5* ⟩
  person:⟨ *nickname:'Bob', tag:'cars', score:8.0* ⟩

person:⟨ nickname:'Alice', tag:'fashion', score:7.0 ⟩
person:⟨ nickname:'Alice', tag:'novels', score:8.5 ⟩ }

The expression $group(person, interests : tag, score[s\_tag])(R)$ thus yields:

$R' =$ { *person:⟨ nickname:'Bob',*
interests:{s_tag:⟨ tag:'sports', score:6.5 ⟩,
s_tag:⟨tag:'cars', score:8.0 ⟩ },
person:⟨ nickname:'Alice',
interests:{s_tag:⟨ tag:'fashion', score:7.0 ⟩,
s_tag:⟨tag:'novels', score:8.5 ⟩ } }

4 Workflow enumeration

This section decribes the process of enumerating all the equivalent workflows that satisfy the same functional requirements given by an ASASEL specification. The enumeration leads to a search space of workflows with increasing levels of parallelism in their structure. The levels of parallelism can privilege the cost preferences such as response time or the communication cost. The enumeration is subject to constraints for composing the required activities by the ASASEL specification. In order to make a proof of concept, we model these constraints as action rules in the language DLV-K[5].

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. 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.* pre-conditions) 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 the plan. A goal is expressed by the conjunction of fluents and by a plan length $l \in \mathbb{Z}^+$.

The mapping from workflow enumeration to a planning problem is shown in Table 1. The APIs and the required activities by the ASASEL specification are modeled as facts of the background knowledge. The execution state of a workflow is modeled as fluents and the activities to perform as actions.

Next we show, through an example, how we represent the background knowledge for workflow enumeration. Afterwards, we show how the workflow state and activities are expressed in DLV-K rules. Given such rules, the DLV-K engine performs the workflow enumeration.

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

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

Table 1: Mapping to a planning problem

4.1 Background knowledge

The background knowledge contains a set of facts that serve as the input for the workflow enumeration. It includes (1) the service methods and (2) the required activities derived from the ASASEL specification.

Service methods are represented by the facts `method/2`. The bound and free attributes associated to such a method are represented by the facts `bound_p/4` and `free_p/4`. The rule `att/4` represents the normal form of an attribute.

```
method(p,profl).
bound_a(p,profl,nickname,str).
free_a(p,profl,age,int).
free_a(p,profl,sex,str).
free_a(p,profl,email,str).

att(DSN,ON,PN,T):- bound_a(DSN,ON,PN,T).
att(DSN,ON,PN,T):- free_a(DSN,ON,PN,T).
```

Required activities are derived from the ASASEL workflow specification and represented through facts (with the underscore at the end). The required activities derived from a workflow implementing "*What are the interests of my friend Joe*" are represented by the following facts.

```
project_(p1,nickname,n).
project_(i1,score,s).
project_(i1,tag,t).
retrieve_(p,profl,p1).
retrieve_(i,interests,i1).
filter_(p1,nickname).
join_(p1,nickname,i1,nickname).
```

These required activities express the need over the methods `p:profl` and `i:interests`. Both data are retrieved by `retrieve_/3` and represented by `p1` and `i1`. The nickname attribute of the profile is filtered by `filter_/2` and correlated by `join_/4` interests through the nickname attribute. The attributes nickname, score and tag are projected. Observe that the filter over the nickname attribute is only indicated as the equality operators are not relevant for the workflow transformation.

4.2 Workflow activities

Workflow activities are represented as actions in DLV-K. Such actions are predicates that require facts from the background knowledge to be true. There are also activities that are independent from facts.

init and finish These activities have the special purpose to initialize and terminate the workflow execution. Thus their semantics is not associated with the application and there is no dependency with the background knowledge.

data_service establishes a connection with a data service method. It requires from the knowledge base a service method and the expressed need to retrieve data from it.

```
data_service(DS) requires method(DSN,ON), retrieve_(DSN,ON,DS).
```

bind_selection invokes a service method and retrieves data from it. The invocation is done by providing a bound attribute.

```
bind_selection(DS,BP) requires method(DSN,ON),
   retrieve_(DSN,ON,DS), bound_a(DSN,ON,BP,_), filter_(DS,BP).
```

bind_join correlates data from two service methods w.r.t. an attribute from each one. The attribute from the outer method must be bound. This activity is analogous to `bind_selection` but it takes the value from another method attribute.

```
bind_join(DS1,P1,DS2,BP2) requires
   method(DSN1,ON1), retrieve_(DSN1,ON1,DS1),
   att(DSN1,ON1,P1,_), method(DSN2,ON2), retrieve_(DSN2,ON2,DS2),
   bound_a(DSN2,ON2,BP2,_), join_(DS1,P1,DS2,BP2).
```

filter performs the filtering over an attribute of a required service method.

```
filter(DS,P) requires method(DSN,ON), retrieve_(DSN,ON,DS),
                      att(DSN,ON,P,_), filter_(DS,P).
```

project projects an attribute of a service method.

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

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

4.3 Workflow constraints

The workflow constraints define the pre-conditions and post-conditions associated to the execution of the workflow activities. A condition is a state of knowledge modifiable by the execution of activities. Through the satisfaction of such conditions, the workflows are transformed. In the following, we present the intuition of these constraints along with their rules in DLV-k.

init and finish The **init** activity has no previous activity and its pre-condition is that the workflow has not been `initiated`. As post-condition, it produces the state `initiated`. The last activity is `finish` and there is no other activity to be executed afterwards. Its pre-condition is that there is not evidence that the workflow is `finished` and the result has been `delivered` (See `output` activity

below for details about `delivered`). The post-condition of `finish` is `finished` and this is the goal to be reached for the workflow transformation.

```
executable init if   -initiated.
caused     initiated after init.
executable finish if not finished, delivered.
caused     finished after finish.
```

data_service Once initiated the workflow, the data services must be `connected(DS)`. This fluent is produced by the execution of the `data_service(DS)` activity.

```
executable data_service(DS) if initiated.
caused     connected(DS) after data_service(DS).
```

In order to retrieve all the required data, all data services should be connected. The 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).
caused  all_connected if not -all_connected.
```

bind_selection It is only executable if there is not evidence that data from the data service `DS` have been retrieved and if there is a connection with `DS`. Once the bind selection is executed, the fluent `retrieved(DS)` is true.

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

filter It is executable if there is not evidence that the attribute `P` of `DS` has been filtered. It is required that the data from `DS` have been retrieved and the activity `select_(DS,P)` must be required. The execution of the filter makes the fluent `filtered(DS,P)` true.

```
executable filter(DS,P) if not filtered(DS,P),
           retrieved(DS),filter_(DS,P).
caused filtered(DS,P) after filter(DS,P).
```

As might several filter activities over `DS` are required, the `all_filtered_from` becomes true if there is no other attribute pending to be filtered.

```
caused -all_filtered_from(DS) if not filtered(DS,P), filter_(DS,P).
caused  all_filtered_from(DS) if not -all_filtered_from(DS),
                                 retrieved(DS).
```

There is the fluent `all_filtered` that becomes true if there is no other attribute of the method `DS` pending to be filtered.

```
caused -all_filtered if -all_filtered_from(DS), filter_(DS,P).
caused -all_filtered if -all_filtered_from(DS), not filter_(DS,P),
                        att(DSN,ON,P,_), retrieve_(DSN,ON,DS).
```

project This activity is executable if there is not evidence that the attribute `P` of `DS` has been projected. The execution of projection makes the fluent `projected` true.

```
executable project(DS,P) if not projected(DS,P),retrieved(DS),
                            project_(DS,P,_).
```

During the workflow execution, all the projection activities have to be performed. For the method `DS`, the fluent `all_projected_from` is true if there is no other attribute from `DS` pending to be projected. The fluent `all_projected` is true if there is no other `DS` with an attribute pending to be projected.

```
caused -all_projected_from(DS) if not projected(DS,P), project_(DS,P,_).
caused  all_projected_from(DS) if not -all_projected_from(DS)
                               after project(DS,P).
caused -all_projected if -all_projected_from(DS), project_(DS,P,_).
```

output Once all the required activities are performed, the result is delivered by the activity `output`. To model this pre-condition, the fluent `activities_performed` is true if all the required activities have been processed. Otherwise, the fluent is false `-activities_performed`.

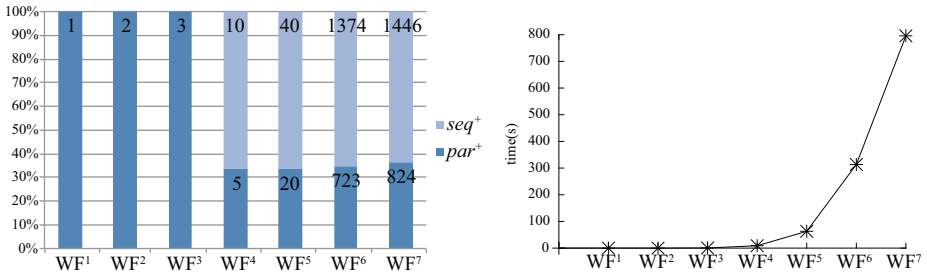
```
caused  all_projected if not -all_projected.
caused -activities_performed if not all_connected, not all_retrieved,
                                not all_filtered, not all_projected.
caused  activities_performed if not -activities_performed.
```

Once the result is delivered by `output`, the fluent `delivered` becomes true and the workflow can be finished (*cf.* `finish` pre-conditions).

```
executable output if activities_performed, not delivered.
caused    delivered after output.
```

5 Experiments

We performed experiments to measure the amount of alternative workflows with sequential compositions and with parallel compositions for a given ASASEL workflow. We setup seven different ASASEL workflows $WF^1, ..., WF^7$ with increasing number of activities and different potential grades of parallelism. The generated workflows were classified by analyzing the data dependencies among activities and their structures. A workflow whose independent activities are composed in parallel is classified as par^+, otherwise it is classified as seq^+. The charts in Figure 2 show the classified space of alternatives and the required time for each workflow $WF^1, ..., WF^7$.



(a) Space of alternatives (b) Required time

Fig. 2: Enumeration of the space of alternative workflows with different grade of parallelism

In Figure 2a, the spaces of the workflows $WF^1 - WF^3$ only contain par^+ workflows because they have few activities and there are no independent activi-

ties. The spaces of the workflows $WF^4 - WF^7$ have 1/3 of par^+ workflows and 2/3 seq^+ workflows. This correspondence is not constant and depends on the data dependencies among activities, *e.g.* a workflow with many activities may have only sequential alternatives if there are no independent activities.

The par^+ workflows represent better opportunities for improving time related costs while the seq^+ ones privilege the resource usage (*e.g.* network, cpu). This classification can be used for improving the enumeration performance (*cf.* Figure 2b) by incorporating user/application preferences for obtaining only par^+ or seq^+.

6 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 provides the user a workflow visualization by applying the construction rules outlined in Section 2.1.

The system interacts with DLV-K through intermediate input and output files generated and parsed as required. The enactment of a selected workflow is supported 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.

During the execution of a workflow, data flows from the data services to complex value operators as well as several computation services via queues, as determined by the ASASEL specification. These computation services run on a Tomcat container supported by the JAX-WS reference implementation, which enables to create stateful services. Additional output 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 a data stream that denotes the tuples that are added and the tuples that are removed from the result dataset.

We implemented two test scenarios and their corresponding data and computation services. The first one is the location-based application introduced in Section 2. The second scenario is an adaptation of the online auctions 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.

Fig. 3: Caption of the ASASEL GUI

[6] http://datalab.cs.pdx.edu/niagara/NEXMark/

7 Related work

Data-centric workflows involving services share some similarities with queries over Web services as presented in [9]. There the authors propose an optimization approach by ordering the service calls in a pipelined fashion and by tuning the size of service call batches. An algebraic approach for the optimization of workflows with relational and map-reduce operations is presented in [7]. Our approach is to enable workflows with a broader variety of operations defined through service compositions, thus requiring alternative optimization techniques.

Planning techniques have been applied for automatic service composition, for instance in [6] and [8]. The problem addressed in those works is to create a service composition from atomic actions (services) based on a propositional goal. The Roman Model [1] alternatively employs finite state transition system descriptions for the available and target services, but with the same basic objective in mind. However, we use planning techniques instead for the optimization of a workflow that includes possibly composite computation services.

Alternative formalisms for the specification of workflows include, for example, process algebras [3] and petri nets [5]. The use of ASMs provides a formal semantics, as in the aformentioned formalisms, but also fully compatible text and workflow representations that are easy to specify. Although ASMs have been used to study and model the properties of workflows, less effort has been given to using them in a fully operational manner.

8 Conclusions and future work

In this paper we presented a language and system for the specification and enactment of data-centric workflows based on service composition. In addition, we introduced a planning-based approach for the generation of the search space of workflows implementing requirements specifications. Concretely, we proposed a set of constraints modeled in an action language, specifically DLV-K, in order to characterize the transformation of workflows with sequential and parallel compositions. This work is envisaged to be a foundation for incorporating a full cost model that covers the specification of composite computation services, leading to the selection of the most suitable workflow w.r.t. the user's preferences. Future work also includes validating the practicality of ASASEL for the specification of data-centric workflows for diverse users, which would require a more sophisticated GUI-based editing tool than our current prototype.

References

1. Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M., Patrizi, F.: Automatic service composition and synthesis: the roman model. IEEE Data Eng. Bull. 31(3), 18–22 (2008)
2. Cuevas-Vicenttín, V.: Evaluation of hybrid queries based on service coordination. Ph.D. thesis, University of Grenoble (May 2005), `http://tel.archives-ouvertes.fr/tel-00630601`

3. Curcin, V., Missier, P., De Roure, D.: Simulating taverna workflows using stochastic process algebras. Concurr. Comput. : Pract. Exper. 23(16), 1920–1935 (Nov 2011)
4. Gurevich, Y.: Specification and validation methods. chap. Evolving Algebras 1993: Lipari Guide, pp. 9–36. Oxford University Press, Inc., New York, NY, USA (1995)
5. Hidders, J., Kwasnikowska, N., Sroka, J., Tyszkiewicz, J., Van den Bussche, J.: Dfl: A dataflow language based on petri nets and nested relational calculus. Inf. Syst. 33(3), 261–284 (May 2008)
6. McIlraith, S.A., Son, T.C.: Adapting golog for composition of semantic web services. In: Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22-25, 2002. pp. 482–496 (2002)
7. Ogasawara, E.S., de Oliveira, D., Valduriez, P., Dias, J., Porto, F., Mattoso, M.: An algebraic approach for data-centric scientific workflows. PVLDB 4(12), 1328–1339 (2011)
8. Sirin, E., Parsia, B., Wu, D., Hendler, J., Nau, D.: Htn planning for web service composition using shop2. Web Semant. 1(4), 377–396 (Oct 2004)
9. Srivastava, U., Munagala, K., Widom, J., Motwani, R.: Query optimization over web services. In: Proceedings of the 32Nd International Conference on Very Large Data Bases. pp. 355–366. VLDB '06, VLDB Endowment (2006)