A Logic-Based Formalism to Model and Analyze Workflow Executions

Gianluigi Greco[1], **Antonella Guzzo**[1], and **Domenico Saccà**[1,2]

DEIS[1], University of Calabria, Via Pietro Bucci 41C, 87036 Rende, Italy
ICAR, CNR[2], Via Pietro Bucci 41C, 87036 Rende, Italy
{ggreco,guzzo}@si.deis.unical.it, sacca@icar.cnr.it

Abstract. Workflow management systems are a key technology for effectively modeling, executing and monitoring business processes in several application domains such as finance and banking, healthcare, telecommunications, manufacturing and production. Many research works deal with the phase of modeling workflow schemes and several formalisms for specifying structural properties have been already proposed to support the designer in devising all admissible execution scenarios. Most of such formalisms are based on graphical representations in order to give a simple and intuitive description of the workflow structure. This paper presents a new formalism which combines a rich graph representation of workflow schemes with simple (i.e., stratified), yet powerful DATALOG rules to express complex properties and constraints on executions. The graph representation allows one to specify quantified tasks which are instantiated in a number of occurrences during an execution, on the basis of the actual values that can be assigned to the quantified variables. Both the graph representation and the DATALOG rules are mapped into a unique program in $\texttt{DATALOG}^{\texttt{ev!}}$, that is a recent extension of DATALOG for handling events. This mapping enables the designer to simulate the actual behavior of the modeled scheme by fixing an initial state and an execution scenario (i.e., a sequence of executions for the same workflow) and querying the state after such executions. As the scenario includes a certain amount of non-determinism, the designer may also verify under which conditions a given (desirable or undesirable) goal can be eventually achieved.

1 Introduction

A *workflow* is as a collection of activities that must be performed by one or more software systems, by one or a team of humans, or by a combination of these[4], in order to accomplish some business process.

Workflow management system (WFMs) represent today a key technological infrastructure for effectively managing business processes in several application domains including finance and banking, healthcare, telecommunications, manufacturing and production. Many research works deal with the phase of modeling workflow schemes and several formalisms for specifying structural properties have been already proposed to support the designer in devising all admissible execution scenarios. Most of such formalisms are based on graphical representations in order to give a simple and intuitive description of the workflow structure.

The most common formalism is the *control flow graph*, in which the workflow is represented by a labelled directed graph whose nodes represents the task to be performed, and whose arcs describe the precedences among them. Moreover, Workflow Management Coalition [16] has also identified additional controls, such as loops and sub-workflows.

As pointed out by many authors, see e.g [3], the essential limitation of the approach based on the *control flow graph* lies in the ability of specifying *local* dependencies only; indeed, properties such as synchronization, concurrency, or serial execution of tasks cannot be expressed. These latter properties are called in the literature *global constraints*. Such properties, which cannot be captured by a graph, either are left unstated (thus delivering an incomplete specification) or are eventually expressed using other formalisms, e.g., some form of logics to specify elaborated execution constraints.

In this paper, we propose a logic based environment which combines a rich graph representation of workflow schemes with simple (i.e., stratified), yet powerful DATALOG rules to express complex properties and *global constraints* on executions. In particular, the traditional graph representation is enriched with *quantified* tasks which are instantiated in a number of occurrences during an execution, on the basis of the actual values that can be assigned to the quantified variables — e.g., the request of a certain amount of an item is executed by several tasks inquiring for item availability, one for each store of the selling company. Both the graph representation and the DATALOG rules are mapped into a unique program in $\texttt{DATALOG}^{\texttt{ev!}}$, that is a recent extension of DATALOG for handling events.

We must point out that a similar approach has been adopted in [3], by using the *Concurrent Transaction Logic* ($\mathcal{CTR}$) [2], in order to provide a way to both

describe and reason about workflow, by introducing a rich set of constraints. An implementation of the technique is in [9], in which a compiler, named *Apply*, accepts a workflow specification that includes a control graph, the triggers and a set of temporal constraints; as result of the compilation process, an equivalent specification in $\mathcal{CTR}$ is provided.

It is worth noting that $\mathcal{CTR}$ logic has been used for solving central problems in the workflow management, such as the **consistency**, i.e., deciding whether a workflow graph is consistent w.r.t. some global constraints, and the **verification**, i.e., deciding whether any legal execution satisfies the global constraints.

The framework we propose is, instead, well suited for being used as a run-time environment for the **simulation**; in fact, the mapping into $\texttt{DATALOG}^{\texttt{ev!}}$enables the designer to simulate the actual behavior of the modeled scheme by fixing an initial state and an execution scenario (i.e., a sequence of executions for the same workflow) and querying the state after such executions. As the scenario includes a certain amount of non-determinism, the designer may also verify under which conditions a given (desirable or undesirable) goal can be eventually achieved.

Related Work. A different type of specification consists in the use of *triggers*, in order to define *ECA* rules for describing transitions among states. This approach has been used in the *state chart model* [14], while in [15] a *state and activity chart* is used into a distributed runtime executable workflow environment. The interesting contribution is that, by providing a semantic partitioning of the state charts, this approach produces components that can be executed on different processing units. However, the system does not provide an efficient mechanism to describe the synchronization and concurrency between tasks. This formalization has bee adopted in two different project, i.e., OPERA and MENTOR [8, 1].

In [7] a workflow is modelled by integrating *ECA* rules with object-oriented concepts; this approach is also know as *active object oriented*.

Finally we mention the *Process algebra* of [11], and the use of *Petri Nets* [12] for modeling and analyzing workflows; this latter formalism has a deep formal foundations, and is profitably used for investigating different interesting properties for the process, such as liveness, and boundness. A recently work [13] uses the Petri-net theory and tools to analyze workflow graphs. The approach consists in translating workflow graphs into so-called *workflow-nets*, which are are a class of Petri nets tailored towards workflow analysis.

Organization. The rest of the paper is organized as follows. Section 2 presents some preliminaries on logic programming and the logic language $\texttt{DATALOG}^{\texttt{ev!}}$. Section 3 deals with the basic concepts related to workflow management, while Section 4 describes our proposal for modelling static and dynamic aspect of workflow

specification. Section 5 illustrates the ability of the formalism to handle global constraints and Section 6 introduce powerful non-deterministic query mechanisms which allow the workflow designer to simulate and analyze meaningful workloads, consisting of temporal sequences of workflow instances. Section 7 presents the conclusion and discusses further work.

2 Preliminaries on DATALOG$^{ev!}$

A DATALOG$^{\neg}$rule r is a clause of the form $a \leftarrow b_1, \cdots, b_k, \neg b_{k+1}, \cdots, \neg b_{k+m}$ where $k, m \geq 0$, and $a, b_1, \cdots, b_{k+m}$ are function-free atoms. If $m = 0$, then r is *positive*.

A DATALOG$^{\neg}$ program $\mathcal{P}$ is a finite set of DATALOG$^{\neg}$ rules. Predicate symbols can be either extensional (i.e. defined by the facts of a database — *EDB predicate symbols*), also called *base predicates*, or intensional (i.e. defined by the rules of the program — *IDB predicate symbols*), also called *derived predicates*.

Positive DATALOG programs have a nice semantics and a very efficient implementation but not enough expressive power. Stratified negation is an important but yet limited step forward. A drastic solution is to remove the condition that there is no recursion through negation. Unfortunately, the usage of unrestricted negation in programs is often neither simple nor intuitive, and, for example, might lead to writing programs that have no total stable models. As argued in [5], a promising compromise is to extend stratified DATALOG, with only predefined types of non-stratified negation, hardwired into ad-hoc constructs. A first construct for capturing a controlled form of unstratified negation was the *choice*, whose semantics was defined in terms of stable models in [10] and which exploits the nondeterminism implicit in the notion of stable model. Datalog with stratified negation and *choice* rules is denoted by DATALOG$^{\neg s,c}$.

A second form of unstratified negation is represented by XY-stratification which was first introduced in [19] and has later been used to model updated and active rules [17, 18]. The recursive predicates of an XY-stratified program have a temporal argument which is used to enforce local stratification.

Choice and a variation of XY-stratification, called XYZ-stratification, have been recently combined into the language *Event Choice* DATALOG (DATALOG$^{ev!}$) in [6] in order to deal with events, In addition to classical DATALOG predicate symbols, the language includes *event* predicate symbols having an additional argument which provides the time dimension. An event predicate atom has the format $p(X)@(T)$, where X is a list of arguments and T is the time argument stating that the event occurs at the time T with the properties described in X.

A DATALOG$^{ev!}$ program comprises: (i) the *static definition* that is a DATALOG$^{\neg s}$ program, and (ii) a number of *event definitions*. An event definition consists of

the *event declaration* within brackets and of one or more *transition rules*. It has the following format:

$$[e(X)@(T)] \quad t_1 \quad \cdots \quad t_k$$

where $e(X)@(T)$ is the event which enables the transition rules t_i ($1 \leq i \leq k$, $k > 0$) as soon as it occurs. A transition rule is of the form:

$$E_1,\ \ldots\ ,\ E_n,\ A_1,\ \ldots\ ,\ A_m \leftarrow B, \otimes C_1 \otimes\ \ldots \otimes\ C_s.$$

where

1. E_i ($1 \leq i \leq n$, $n \geq 0$) is an event atom that is triggered if the body of the transition rule is true; E_i has one of the the two following formats:
   - $g(X)$`++` — informally, the event g will occur at the time $T + \delta$ where δ is a sub-unit of time that is used to enable micro transitions for a finer tuning of the program evolution; we stress that sub-units of time cannot be directly handled but only incremented using the above syntax;
   - $g(X)$`+`(T') — informally, the event g will occur at the time $T + T'$ where now T' is not a sub-unit of time but it is measured in the same scale as T;
2. A_i ($1 \leq i \leq m$, $m > 0$) is a `DATALOG` atom that is made true when the rule body is true — observe that at least one `DATALOG` atom must be present in the transition rule head;
3. B is a conjunction of `DATALOG` literals (negation is allowed in the body of the transition rule);
4. C_i ($1 \leq i \leq s$, $s \geq 0$) is a *selection* atom (i.e., $choice((Y),(Z))$, $choiceAny()$, $choiceLeast((Y),Z)$, $choiceMin(Z)$, $choiceMost((Y),Z)$, $choiceMax(Z)$, $choiceCond((Y),(D))$ or $prefer(D)$); the selections are applied in the order they appears in the rule (i.e., as it often happens during an evolution, ordering is relevant).

It is worth noting that any $\texttt{DATALOG}^{\texttt{ev!}}$ program P can be easily traduced into an equivalent $\texttt{DATALOG}^{\neg_s,c}$ program with XYZ-stratification, called the *standard version* $sv(P_D)$. Informally, the rewriting consists in expliciting a time parameter for each predicate symbol. The interested reader can find more details in [6]. As for the usage of the language in this paper, a few intuitions and a number of exemplifications will be sufficient to grasp its semantics (or at least we hope so).

3 A Logic Based Environment for Analyzing Workflow Executions

In this section we introduce some basic concepts regarding the specification of a workflow, by focusing the attention both on the *static aspects*, i.e. the description

of the relationships among activities not depending from a particular instance, and on the *dynamic aspects*, the description of workflow instances whose actual executions depends on status of the system (available servers and other resources).

Any workflow system should provide facilities for modelling the *control flow* graph, for defining the *servers* and some politic on their use, and for giving the user the ability of define more involved *constraint*. The approach we propose is summarized in Figure 3.1, where these three components are mapped into a $\texttt{DATALOG}^{\texttt{ev!}}$ program. We make use of three databases:

- $\mathbf{DB}_{CF}(\mathcal{WS})$, storing the control frow structure,
- $\mathbf{DB}_{WE}(ID)$, storing information on the instance evolution, such as the status of the tasks and of the servers, and
- $\mathbf{DB}_{I}(\mathcal{WS})$, storing additional information needed to the execution.

Note that, $\mathbf{DB}_{CF}(\mathcal{WS})$ and $\mathbf{DB}_{I}(\mathcal{WS})$ are shared among the different instances. Details on each component will be given below.

Moreover, all the *global constraints* and additional constraints on the scheduling of the activities can be translated into a $\texttt{DATALOG}^{\texttt{ev!}}$ program $\mathbf{P}_{\mathbf{Constr}}(\mathcal{WS})$ over the predicates contained in the above databases. Finally, the run-time execution mechanism can be also defined in term of $\texttt{DATALOG}^{\texttt{ev!}}$ rules ($\mathbf{P}(\mathcal{WS})$).

The translation we make, can be profitably used for simulating executions and for reasoning on the instances. Note that, the approach proposed in [3] make also use of a similar mapping, but is not suited for reasoning on case of the executions. Indeed, the CTR logic is useful as a *model checking* mechanism, but at the notion of time is implicit in the specifications, it cannot be used for reasoning on actual scenarios.

3.1 Workflow Schema

A *workflow schema* $\mathcal{WS}$ is a directed graph whose nodes are the tasks and the arcs are their precedences. More precisely, $\mathcal{WS}$ is defined as a tuple $\langle A,\ E,\ a_0,\ F,\ A^{\wedge}_{in},\ A^{\vee}_{in},\ A^{c}_{in},\ A^{\wedge}_{out},\ A^{\vee}_{out},\ A^{L}_{out},\ E^{Q},\ E^{L},\ \lambda,\ L\rangle$ where

- A is the set of tasks, $a_0 \in A$ is the initial task, $F \subseteq A$ is the set of final tasks and L is a set of labels; after completion, each task returns a value in L or "fail" in case of an abnormal execution;
- $E \subseteq (A - F) \times (A - \{a_0\})$ is an acyclic relation of precedences among tasks; after its completion, a task *activates* some (or all) of its outgoing arcs, and in turn a task is *started* if some (or all) of its incoming arcs are activated, according to the properties of the involved tasks and arcs, as described next;

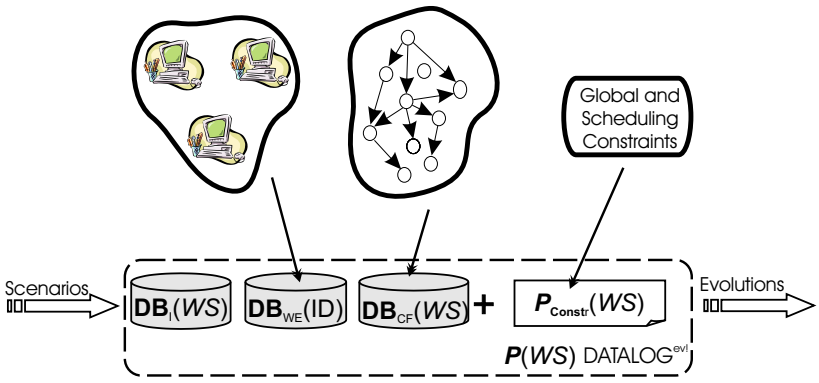

Fig. 1. The run-time environment for workflow executions and simulations.

- $E^Q \subseteq E$ denotes a set of quantified arcs; each arc $(a, b) \in E^Q$ has associated a formula $\forall X : p(X)$, where $p(X)$ is a relation storing the state of the workflow execution, and the task b has no other incoming arc; once activated, (a, b) instantiates several instances of the task b, one for each value assigned to X, and all such tasks are immediately started;
- $E^L = \{(a, b) | \, (a, b) \in E \wedge a \in A^L_{out}\}$ are all the arcs leaving the tasks in A^L_{out} and λ is a labelling function from E^L to L; an arc $(a, b) \in E^L$, say with label o, is activated after the completion of the task a only if the output returned by a coincides with o;
- $A^\wedge_{in} \subseteq A - \{a_0\}$ are the tasks which act as synchronizer (also called a *join* tasks in the literature), thus, a task in $A^\wedge_{in}$ cannot be started until after all its incoming arcs are activated;
- $A^\vee_{in} \subseteq A - \{a_0\}$ are the tasks which can be started as soon as at least one of its incoming arcs is activated;
- $A^c_{in} \subseteq A - \{a_0\}$ are the tasks whose conditions for starting involve more elaborated properties on their incoming arcs — note that $A^\wedge_{in}$, $A^\vee_{in}$ and A^c_{in} form a partition of $A - \{a_0\}$;
- $A^\wedge_{out} \subseteq A - F$ are the tasks which activate all their outgoing arcs;
- $A^\vee_{out} \subseteq A - F$ are the tasks which activate exactly one of their outgoing arcs, that is non-deterministically chosen;
- $A^L_{out} \subseteq A - F$ are the tasks which activate those outgoing arcs whose labels coincide with the label returned by them after completion — note that $A^\wedge_{out}$, $A^\vee_{out}$ and A^L_{out} form a partition of $A - F$.

For a better understanding, let us consider the following example, that will be used throughout the rest of the paper. The example describes a typical process for a selling company.

Example 1. A customer issues a request to purchase a certain amount of given product by filling in a request form on the browser (task *ReceiveOrder*). The request is forwarded to the financial department (task *VerifyClient*) and to each company store (task *VerifyAvailability*) in order to verify respectively whether the customer is reliable and whether the requested product is available in the desired amount in one of the stores. The task *ReceiveOrder* is include in $A^{\wedge}_{out}$ as it must activate both outgoing arcs after completion.

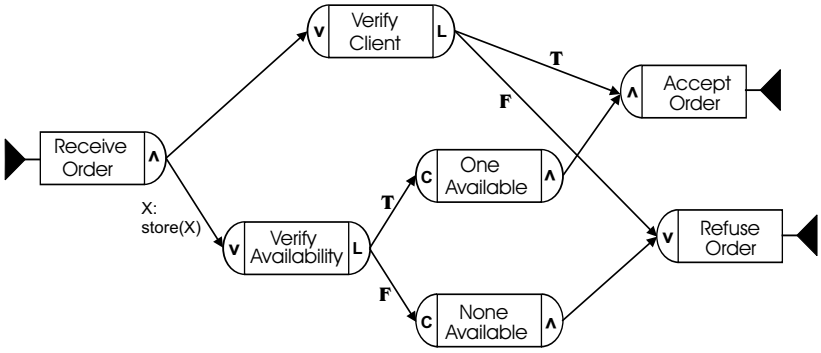

Fig. 2. Example of workflow.

As the task *VerifyAvailability* must be instantiated for each store, the incoming arc must be included in E^{Q}, i.e., it is "quantified". By the activation of the corresponding arcs, each instance of *VerifyAvailability* either notifies to the task *OneAvailable* that there the requested amount is available (label 'T') or otherwise it notifies the non-availability to the task *NoneAvailable* (label 'F'). Observe that the task *OneAvailable* is started as soon as one notification of availability is received whereas the task *NoneAvailable* needs the notifications from all the stores to be activated. The order request will be eventually accepted if both *OneAvailable* has been executed and the task *VerifyClient* has returned the label 'T'; otherwise the order is refused. □

A workflow schema $\mathcal{WS}$ is represented by suitable tuples in the database $\mathbf{DB}_{CF}(\mathcal{WS})$, called *control flow* database, whose relation schemes are:
$\texttt{task}(\texttt{A})$, $\texttt{arc}(\texttt{A},\texttt{A})$, $\texttt{startTask}(\texttt{a}_0)$, $\texttt{finalTask}(\texttt{F})$, $\texttt{qArc}(\texttt{A},\texttt{A})$, $\texttt{inOR}(\texttt{A}^{\vee}_{\texttt{in}})$, $\texttt{inAND}(\texttt{A}^{\wedge}_{\texttt{in}})$, $\texttt{inCond}(\texttt{A}^{\texttt{c}}_{\texttt{in}})$, $\texttt{outOR}(\texttt{A}^{\vee}_{\texttt{out}})$, $\texttt{outAND}(\texttt{A}^{\wedge}_{\texttt{out}})$, $\texttt{outLabel}(\texttt{A}^{\texttt{L}}_{\texttt{out}})$, $\texttt{lArc}(\texttt{A}^{\texttt{L}}_{\texttt{out}},\texttt{A},\texttt{L})$.

Example 2. The *control flow* database of the workflow schema of Example 1 is organized as follows. The relation `task` contains 7 tuples, the tuple in `startTask` is (`ReceiveOrder`) and in `finalTask` there are (`RefuseOrder`) and (`AcceptOrder`).

The relation `inAND` contains the tuple (`AcceptOrder`); `inOR` consists of the tuples (`VerifyClient`), (`VerifyAvailability`) and (`RefuseOrder`); in `inCond` we have the tuples (`OneAvailable`) and (`NoneAvailable`). The tuple (`ReceiveOrder`) is in `outAND` and the tuples (`VerifyClient`) and (`VerifyAvailability`) are in `outLabel`. We insert the tuples (`OneAvailable`) and (`NoneAvailable`) in the relation `outAND` but they could instead be included in `outOR` as they have only one outgoing arc.

Concerning the arcs, we have that the relation *arc* contains 8 tuples, `qArc` consists of the tuple (`ReceiveOrder`, `VerifyAvailability`), and the tuples in the relation `lArc` are:
(`VerifyAvailability`, `OneAvailable`), (`VerifyAvailability`, `NoneAvailable`), (`VerifyClient`, `AcceptOrder`), (`VerifyClient`, `RefuseOrder`). □

3.2 Description of task executions and servers

The second aspect of the specification of a workflow concerns the the description of how the tasks can be executed.

First of all, we are given a set of servers (human and/or computers) who are appointed to execute various tasks, one at the time, in a given amount of time – the execution duration of the same task is not necessarily the same for all servers. Data about the servers are stored in the relation `executable`(`Server`, `Task`, `Duration`) at the start of the execution of workflow instances.

A server is in one of the following states: *available*, *busy*, *outOfOrder*. The latter state is registered into the relation $outOfOrder(Server)$ whose tuples are added or removed during the executions; the other states are derived using simple rules as we shall show later in this section.

A *workflow instance* (also called *workflow enactment*, or *case* in the literature) is activated at a certain time by an external event which starts the initial task. Then subsequent tasks are executed in a order consistent with the precedences and the constraints of the workflow graph — obviously not all tasks in the graph are activated and a task after a quantified arc may be instantiated several times. Thus a workflow instance is a subgraph of the workflow graph only if no quantified arcs are activated.

During the execution of a workflow instance, each task may be in one of the follows states:

1. *idle*, thus the task is not yet started as it needs that some incoming arc be activated;
2. *ready*, i.e., the task is ready for execution and has been started but it is waiting for the assignment of a server;

3. *running*, i.e., the task is currently executed by a server;
4. *ended*, i.e., the task has been terminated.

Given an instance *ID*, the database $\mathbf{DB}_{WE}(ID)$ keeps trace of the state of the execution by means of the following relations:

- `startReady(ID, Task, Quantifiers, Time)`, storing the time when the `Task` was started; observe that `Quantifiers` is a stack of values that are used to instantiate tasks activated by quantified arcs — as several quantified arcs (possibly none) may be activated in cascade, their values are collected in a stack (possibly empty);
- `startRunning(ID, Task, Quantifiers, Server, Time)`, storing the time a server has started its execution;
- `ended(ID, Task, Quantifiers, Time, Output)`, storing the `Time` when the execution of `Task` is completed and the `Output` of the execution.

The state of a task can be derived using simple DATALOG rules, such as:

```
state(ID, Task, Quantifiers, ready)   ← startReady(ID, Task, Quantifiers, _),
                                        ¬ startRunning(ID, Task, Quantifiers, _, _).
state(ID, Task, Quantifiers, running) ← startRunning(ID, Task, Quantifiers, Server, _),
                                        ¬ ended(ID, Task, Quantifiers, _, _).
```

To simplify the notation, we used some syntactic sugar for writing negative literals in the body of the first of the above rules: $\neg a(X)$, stands for $\neg a'(Y)$, where a' is defined by the new rule: $a'(Y) \leftarrow a(X)$, and Y is the list of all non-anonymous variables occurring in X. We shall use this notation also in the rest of the paper.

The following DATALOG rule is used to derive whether a server is available to start some task:

```
available(Server) ← executable(Server, _, _), ¬ outOfOrder(Server),
                    ¬ ( startRunning(ID, Task, Quantifiers, Server, _),
                      ¬ ended(ID, Task, Quantifiers, _, _) ).
```

In the above rule we have further simplified the notation for writing negated conjunctions in the body of a rule r: $\neg(C)$, where C is a conjunction, stands for $\neg c(X)$, where c is defined by the new rule: $c(X) \leftarrow C$, and X is the list of all variables occurring in C which also occur in r.

In addition to the control flow database $\mathbf{DB}_{CF}(\mathcal{WS})$, each workflow has also associated an internal database $\mathbf{DB}_{I}(\mathcal{WS})$ storing information to be shared among the different instances, and used for computations purposes.

Example 3. In our running example, the company may want to store information about the quantity of products available in each store into the relations:

$\texttt{store}(\texttt{IDstore}, \texttt{City})$, $\texttt{product}(\texttt{IDitem})$, and $\texttt{availability}(\texttt{IDstore}, \texttt{IDitem}, \texttt{QTY})$. Note that the relation `store` is the one used in the quantified arc from `ReceiveOrder` to `VerifyAvailability`. Finally a relation $\texttt{order}(\texttt{ID}, \texttt{IDitem}, \texttt{QTY})$ may be used for storing the quantity of a product required in a workflow instantiation by a customer living in a city. □

4 Describing the Workflow Evolution in DATALOG$^{\texttt{ev!}}$

The aim of this section is to present a logic framework for the specification of the executions of a workflow for a given scenario of instances. This framework can be thought of as a simulation environment for workflow executions.

We use the setting of DATALOG$^{\texttt{ev!}}$ for defining events in the logic description. In order to simplify the presentation, in the following, a predicate $\texttt{p}(\texttt{ID}, \texttt{X})$, where `X` is a generic list of arguments, will be denoted by $\texttt{p}_{ID}(\texttt{X})$.

The first event, called `init`, is an external event which starts a new workflow instance at a certain time.

$$\texttt{r}_1: [\texttt{init}(\texttt{ID})@(\texttt{T})] \quad \texttt{run}()\texttt{++}, \texttt{started}_{ID}(), \texttt{startReady}_{ID}(\texttt{ST}, [\,], \texttt{T}) \leftarrow \texttt{startTask}(\texttt{ST}).$$

Every time the event $\texttt{run}()@(\texttt{T})$ is internally triggered, the system tries to assign the ready tasks to the available servers — as we do not use a particular policy for scheduling the servers, the assignment is made in a nondeterministic way. The predicate $\texttt{unsat}_{ID}()$ is true if it has been already checked that the workflow instance does not satisfy possible constraints on the overall execution – this check is performed during the event `complete`, described below. The predicate $\textit{executed}_{ID}()$ is true if the workflow instance has already entered a final state so that no other task need to be performed.

$$\begin{aligned}\texttt{r}_2: [\texttt{run}()@(\texttt{T})]\quad & \texttt{evaluate}_{ID}(\texttt{Task}, \texttt{L}, \texttt{Duration})\texttt{++}, \\ & \texttt{startRunning}_{ID}(\texttt{Task}, \texttt{L}, \texttt{Server}, \texttt{T}) \leftarrow \neg\texttt{unsat}_{ID}(), \neg\texttt{executed}_{ID}(), \\ & \qquad \texttt{state}_{ID}(\texttt{Task}, \texttt{L}, \texttt{Ready}), \texttt{available}(\texttt{Server}), \\ & \qquad \texttt{executable}(\texttt{Server}, \texttt{Task}, \texttt{Duration}) \\ & \qquad \otimes \texttt{choice}((\texttt{Task}), (\texttt{Server})) \\ & \qquad \otimes \texttt{choice}((\texttt{Server}), (\texttt{Task})).\end{aligned}$$

Once the tasks are assigned to servers, their executions start. So information on the assigned servers and the execution starting time are stored; moreover, an event `evaluate` is triggered for each execution.

$$\texttt{r}_3: [\texttt{evaluate}_{ID}(\texttt{Task}, \texttt{L}, \texttt{D})@(\texttt{T})] \quad \texttt{complete}_{ID}(\texttt{Task}, \texttt{L}, \texttt{OG})\texttt{+}(\texttt{D}), \leftarrow \textbf{evaluation}_{ID}(\texttt{Task}, \texttt{L}, \texttt{Output}).$$

The predicate $\mathbf{evaluation}_{ID}(\texttt{Task}, \texttt{L}, \texttt{Output})$ is used to model the function performed by each task, typically depending on both the execution and internal databases — this predicate must be suitably specified by the workflow designer. The event for completing the task is triggered at the time $\texttt{T} + \texttt{D}$, where D is the duration of the task for the assigned server.

Example 4. In our example, the task *VerifyAvailability* must check whether a given store contains enough quantify of the required item. The task behavior is captured by the following rules:

$$\begin{aligned}
&\mathbf{evaluation}_{ID}(\texttt{VerifyAvailability}, [\texttt{X}], \text{“T”}) \leftarrow \texttt{store}(\texttt{X}),\ \texttt{order}_{ID}(\texttt{Item}, \texttt{QTY}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{enoughItem}(\texttt{Store}, \texttt{Item}, \texttt{QTY}).\\
&\mathbf{evaluation}_{ID}(\texttt{VerifyAvailability}, [\texttt{X}], \text{“F”}) \leftarrow \texttt{store}(\texttt{X}),\ \texttt{order}_{ID}(\texttt{Item}, \texttt{QTY}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \neg\ \texttt{enoughItem}(\texttt{X}, \texttt{Item}, \texttt{QTY}).\\
&\texttt{enoughItem}(\texttt{Store}, \texttt{Item}, \texttt{QTY}) \qquad\qquad\qquad \leftarrow \texttt{availability}(\texttt{X}, \texttt{Item}, \texttt{QTYavail}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{QTYavail} \geq \texttt{QTY}.
\end{aligned}$$

Note that these rules make use of the internal database $\mathbf{DB}_I(\mathcal{WS})$. □

As described in the next event, after the completion of a task, the selection of which of its outgoing arcs be activated depends on whether the task is in $A^{\vee}_{out}$, $A^{\wedge}_{out}$, or A^{L}_{out} and can be done only if the task execution is not failed. The two actions of registering data about the completion and of triggering the event run to possibly assign the server to another task are performed in all cases. The fact $\texttt{unsat}_{ID}(\texttt{T})$ is added only if the predicate $\mathbf{unsatGC}_{ID}(\texttt{Task}, \texttt{L})$ is true. This predicate is defined by the workflow designer to enforce possible *global constraints* — if not defined then no global constraints are checked after the completion of the task. We shall return on the definition of this predicate for typical global constraints in the next section. Observe that in the case of a final task, if the global constraints are satisfied then we can register the successful execution of the workflow instance.

$$\begin{aligned}
\texttt{r}_4:\ &[\texttt{complete}_{ID}(\texttt{Task}, \texttt{L}, \texttt{Output})@(\texttt{T})]\\
&\texttt{run}()\texttt{++}, \texttt{ended}_{ID}(\texttt{Task}, \texttt{L}, \texttt{T}, \texttt{Output}).\\
&\texttt{unsat}_{ID}() \qquad\qquad\qquad\qquad\qquad \leftarrow \mathbf{unsatGC}_{ID}(\texttt{Task}, \texttt{L}).\\
&\texttt{executed}_{ID}() \qquad\qquad\qquad\qquad\quad \leftarrow \texttt{finalTask}(\texttt{Task}),\ \neg\, \mathbf{unsatGC}_{ID}(\texttt{Task}, \texttt{L}).\\
&\texttt{activateArc}_{ID}(\texttt{Task}, \texttt{L}, \texttt{Next})\texttt{++} \quad \leftarrow \texttt{outOR}(\texttt{Task}),\ \texttt{Output} \neq \text{“}\texttt{fail}\text{”},\ \texttt{arc}(\texttt{Task}, \texttt{Next})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \otimes\, \texttt{ChoiceAny}().\\
&\texttt{activateArc}_{ID}(\texttt{Task}, \texttt{L}, \texttt{Next})\texttt{++} \quad \leftarrow \texttt{outAND}(\texttt{Task}),\ \texttt{Output} \neq \text{“}\texttt{fail}\text{”},\ \texttt{arc}(\texttt{Task}, \texttt{Next}).\\
&\texttt{activateArc}_{ID}(\texttt{Task}, \texttt{L}, \texttt{Next})\texttt{++} \quad \leftarrow \texttt{outLabel}(\texttt{Task}),\ \texttt{Output} \neq \text{“}\texttt{fail}\text{”},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{arcLabel}(\texttt{Task}, \texttt{Next}, \texttt{Label}),\ \texttt{Label} = \texttt{Output}.
\end{aligned}$$

The event activateArc performs two distinct actions, depending on whether the arc involved in the notification is quantified or not. In the latter case, the

arc is immediately activated. Otherwise, first the quantification is evaluated by means of the user defined predicate $\texttt{evaluateQ}(\texttt{Task}, \texttt{Next}, \texttt{X})$ and, then, for each value x returned for the variable X, a new instantiation of the arc is both created and activated by pushing x into the stack L. Each activated arc is then stored into the relation $\texttt{activatedArc}_{ID}(\texttt{Task}, \texttt{Next}, \texttt{L})$. Finally the event `activatedArc` is triggered in order to the verify whether the activation will make some task ready for starting.

$$
\begin{array}{ll}
\texttt{r}_5: & [\texttt{activateArc}_{ID}(\texttt{Task}, \texttt{L}, \texttt{Next})@(\texttt{T})] \\
& \quad \texttt{checkNext}_{ID}(\texttt{Next}, [\texttt{X}|\texttt{L}])\texttt{++} \\
& \quad \texttt{activatedArc}_{ID}(\texttt{Task}, \texttt{Next}, [\texttt{X}|\texttt{L}]) \leftarrow \texttt{qArc}(\texttt{Task}, \texttt{Next}),\ \mathbf{evaluateQ}(\texttt{Task}, \texttt{Next}, \texttt{X}). \\
& \quad \texttt{checkNext}_{ID}(\texttt{Next}, \texttt{L})\texttt{++} \\
& \quad \texttt{activatedArc}_{ID}(\texttt{Task}, \texttt{Next}, \texttt{L}) \quad \leftarrow \neg\texttt{qArc}(\texttt{Task}, \texttt{Next}).
\end{array}
$$

Example 5. In our running example, the quantification for the arc from the task *ReceiveOrder* to *VerifyAvailability* is defined by the following rule:

$$\mathbf{evaluateQ}(\text{“}\texttt{ReceiveOrder}\text{”}, \text{“}\texttt{VerifyAvailability}\text{”}, \texttt{X}) \leftarrow \texttt{store}(\texttt{X}).$$

stating that the task *VerifyAvailability* must be “multiplied” for each store `X` of the company. □

The event `checkNext` stores each task which starts now and triggers the event `run` in order to possibly assign a server for its execution.

$$
\begin{array}{ll}
\texttt{r}_6: & [\texttt{checkNext}_{ID}(\texttt{Task}, \texttt{L})@(\texttt{T})] \\
& \quad \texttt{run}()\texttt{++} \quad \leftarrow . \\
& \quad \texttt{startReady}_{ID}(\texttt{Task}, \texttt{L}, \texttt{T}) \leftarrow \texttt{inOr}(\texttt{Task}),\ \neg\texttt{state}_{ID}(\texttt{Task}, \texttt{L}, \texttt{ready}). \\
& \quad \texttt{startReady}_{ID}(\texttt{Task}, \texttt{L}, \texttt{T}) \leftarrow \texttt{inAnd}(\texttt{Task}), \\
& \qquad\qquad \neg\,(\texttt{arc}(\texttt{Prec}, \texttt{Task}), \neg\,\texttt{notifiedArc}_{ID}(\texttt{Prec}, \texttt{Task}, \texttt{L})). \\
& \quad \texttt{startReady}_{ID}(\texttt{Task}, \texttt{L}', \texttt{T}) \leftarrow \texttt{inCond}(\texttt{Task}),\ \neg\texttt{state}_{ID}(\texttt{Task}, \texttt{L}, \texttt{ready}), \\
& \qquad\qquad \mathbf{declaredInCond}_{ID}(\texttt{Task}, \texttt{L}, \texttt{L}').
\end{array}
$$

The predicate $\texttt{declaredInCond}_{ID}(\texttt{Task}, \texttt{L}, \texttt{L}')$ is to be defined by the workflow designer.

Example 6. In our running example, we have:

$$
\begin{array}{l}
\texttt{declaredInCond}_{ID}(\texttt{OneAvailable}, [\texttt{X}], [\texttt{X}]) \leftarrow \texttt{evaluation}_{ID}(\texttt{VerifyAvailability}, [\texttt{X}], \text{“}\texttt{T}\text{”}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \otimes\ \texttt{choiceAny}(). \\
\texttt{declaredInCond}_{ID}(\texttt{NoneAvailable}, [\texttt{X}], []) \leftarrow \neg\ \texttt{evaluation}_{ID}(\texttt{VerifyAvailability}, \_, \text{“}\texttt{T}\text{”}).
\end{array}
$$

Note that the second rule has the effect of dropping the quantification of the stores if no store has the required quantity, thus collapsing all occurrences of the task `NoneAvailable` into one. At most one occurrence is retained also for the task `OneAvailable` because of the selection made by the `choiceAny` construct in the first rule. □

Finally we have two external events for putting a server out of order and for resuming her/his/it availability.

$$\begin{aligned} \mathtt{r}_7: \ & [\mathtt{setOutOfOrder(Server)@(T)}] \\ & \mathtt{outOfOrder(Server)}. \\ \mathtt{r}_8: \ & [\mathtt{setInOrder(Server)@(T)}] \\ & \mathtt{run()++}, \neg\mathtt{outOfOrder(Server)}. \end{aligned}$$

Definition 1. Let $\mathcal{WS}$ be a workflow schema, and *ID* be an instance. Then, the logic program $\mathcal{P}(\mathcal{WS})$ modelling the behavior of a workflow system, consists of the facts in $\mathbf{DB}_{CF}(\mathcal{WS})$, and of the set of $\mathtt{DATALOG}^{\mathtt{ev!}}$ rules $\{r_1, \cdots, r_8\}$, over the schema $\mathbf{DB}_{WE}(ID) \cup \mathbf{DB}_I(\mathcal{WS})$. □

5 Adding Global Constraints

In this section, we complete the model by showing how to specify global constraints. First, we formalize the types of constraints we want to model.

Definition 2. Given a workflow $\mathcal{WS}$, a *global constraint* over $\mathcal{WS}$ is defined as follows:

- for any $a \in A$, $!a$ (resp. $\neg!a$) is a *positive* (resp., *negative*) *primitive* global constraint,
- given two positive primitive global constraints c_1 and c_2, $c_1 \prec c_2$ is a *serial* global constraint,
- given any two global constraints c_1 and c_2, $c_1 \vee c_2$ and $c_1 \wedge c_2$ are *complex* global constraints. □

Informally, a *positive* (resp., *negative*) *primitive* global constraint specifies that a task must (resp., must not) be performed in any workflow instance — obviously a negative constraints makes sense only as a sub-expression of a complex global constraint. A *serial* global constraint $c_1 \prec c_2$ specifies that the event specified in the global constraint c_1 must happen before the one specified in c_2. The semantics of the operators $\vee$ and $\wedge$ are the usual.

Global constraints can be mapped into a set of $\mathtt{DATALOG}^{\mathtt{ev!}}$ rules as follows:

- for each global constraint $c = !a$, we introduce the rules:

$$\begin{aligned} &\mathtt{unsatGC1}_{ID}(\mathtt{c}, \mathtt{gs}) \leftarrow \mathtt{ended}_{ID}(\mathtt{a}, \_, \_, \mathtt{0}), \mathtt{0} = \text{“}\mathtt{fail}\text{”}. \\ &\mathtt{unsatGC1}_{ID}(\mathtt{c}, \mathtt{gs}) \leftarrow \neg\, \mathtt{ended}_{ID}(\mathtt{a}, \_, \_, \_). \end{aligned}$$

  where `gs` equals `s` if c only occurs as sub-expression of a complex global constraint; otherwise (i.e., c is a global constraint), gs holds `g`.

– for each global constraint $c = \neg\, !a$, we introduce the rule:

$$\texttt{unsatGC1}_{ID}(\texttt{c},\texttt{gs}) \leftarrow \texttt{ended}_{ID}(\texttt{a},\_,\_,\texttt{O}),\texttt{O} \neq \text{“}\texttt{fail}\text{”}.$$

– the rules for a global constraint $c = !a_1 \prec !a_2$ are:

$$\begin{array}{l}\texttt{unsatGC1}_{ID}(\texttt{c},\texttt{gs}) \leftarrow \texttt{ended}_{ID}(\texttt{a}_2,\_,\_,\texttt{O2}),\ \texttt{O2} \neq \text{“}\texttt{fail}\text{”} \\ \qquad\qquad\qquad\qquad\ \ \texttt{ended}_{ID}(\texttt{a}_1,\_,\_,\text{“}\texttt{fail}\text{”}). \\ \texttt{unsatGC1}_{ID}(\texttt{c},\texttt{gs}) \leftarrow \texttt{ended}_{ID}(\texttt{a}_2,\_,\texttt{T2},\texttt{O2}),\texttt{O2} \neq \text{“}\texttt{fail}\text{”} \\ \qquad\qquad\qquad\qquad\ \ \texttt{ended}_{ID}(\texttt{a}_1,\_,\texttt{T1},\texttt{O1}),\ \texttt{O1} \neq \text{“}\texttt{fail}\text{”},\ \texttt{T2} < \texttt{T1}.\end{array}$$

– for each global constraint $c:\ c_1 \vee c_2$, we have the rule:

$$\texttt{unsatGC1}_{ID}(\texttt{c},\texttt{gs}) \leftarrow \texttt{unsatGC1}_{ID}(\texttt{c}_1,\_),\ \texttt{unsatGC1}_{ID}(\texttt{c}_2.\_).$$

– for each global constraint $c:\ c_1 \wedge c_2$, the rules are:

$$\begin{array}{l}\texttt{unsatGC1}_{ID}(\texttt{c}) \leftarrow \texttt{unsatGC1}_{ID}(\texttt{c}_1,\_) \\ \texttt{unsatGC1}_{ID}(\texttt{c}) \leftarrow \texttt{unsatGC1}_{ID}(\texttt{c}_2,\_).\end{array}$$

Let us now define the predicate $\mathbf{unsatGC}_{ID}(\texttt{Task},\texttt{L})$ used inside the event `complete`. The problem is selecting the time for checking global constraints. Obviously this check must be done after the completion of a final task. So we can use the following definition :

$$\mathbf{unsatGC}_{ID}(\texttt{Task},\texttt{L}) \leftarrow \texttt{finalTask}_{ID}(\texttt{Task}),\ \texttt{unsatGC1}_{ID}(\_,\texttt{g}).$$

Observe that we do not check satisfaction for constraints which are only used as sub-expressions.

Note that some global constraint check can be anticipated. For instance, the global constraint $c = !a_1 \prec !a_2$ can be checked just after the execution of the task $a2$; so we may introduce the rule:

$$\mathbf{unsatGC}_{ID}(\texttt{a}_2,\texttt{L}) \leftarrow \texttt{unsatGC1}_{ID}(\texttt{c},\texttt{g}).$$

An interesting optimization issue is to find out which global constraints could be effectively tested after the completion of each task.

Observe that, as discussed in the previous section, a successful or unsuccessful completion for a workflow instance *ID* is registered by means of the predicate $\texttt{executed}_{ID}()$ or $\texttt{unsat}_{ID}()$, respectively. If both predicates are not true, then the are two case: either (i) the execution is not yet finished for some task is currently ready or running, or (ii) non more tasks are scheduled even though a final task was not reached. The latter case indeed corresponds to an unsuccessful completion of the workflow instance and can be modelled as follows:

$$\begin{array}{l}\texttt{failed}_{ID}() \ \leftarrow \texttt{unsat}_{ID}(). \\ \texttt{failed}_{ID}() \ \leftarrow \texttt{started}_{ID}(),\ \neg\,\texttt{executed}_{ID}(), \neg\,\texttt{working}_{ID}(). \\ \texttt{working}_{ID}() \leftarrow \texttt{state}_{ID}(\texttt{T},\_,\texttt{ready})). \\ \texttt{working}_{ID}() \leftarrow \texttt{state}_{ID}(\texttt{T},\_,\texttt{running})).\end{array}$$

6 Querying for an evolution

Once introduced a model for specifying structural and dynamic aspects of a workflow, the next step is to provide a mechanism for querying the model in order to obtain information on its (possible) evolutions. For instance, in our running, the designer may be interested in knowing whether (and when) a given task has been executed for a given pre-defined scenario.

The scenario is modelled by means of a list containing all the request (with the corresponding arrival time), and it is denoted with $(\mathcal{S})$. For instance, the scenario $[\mathtt{init}(\mathtt{id}_1)@(0), \mathtt{init}(\mathtt{id}_2)@(2), \mathtt{init}(\mathtt{id}_3)@(4), \mathtt{init}(\mathtt{id}_4)@(5)]$ specifies a new instantiation of the workflow at time 0, 2, 4 and 5. This scenario is used for querying the $\mathcal{P}(\mathcal{WS})$ $\mathtt{DATALOG}^{\mathtt{ev!}}$ program.

Definition 3. A query on a $\mathtt{DATALOG}^{\mathtt{ev!}}$ program P is of the form $\langle \mathtt{E}, \mathtt{G}, \mathtt{R} \rangle$ where
- S is a scenario — say that t^S_{max} is the last time of the events in S;
- G (*goal*) is a list of query conditions of the form $[\mathtt{g}_1@(\mathtt{t}_1), \ldots, \mathtt{g}_\mathtt{n}@(\mathtt{t}_\mathtt{n})]$ ($n \geq 0$), where $0 \leq t_1 < t_2 < \ldots < t_n \leq t^S_{max}$; each g_i can be:
  - $\exists(A)$, where A is a conjunction of ground DB literals;
  - $\forall(A)$, where A is a (possibly negated) conjunction of ground DB literals;
  - $opt(X : A)$, where $opt = min$ or max, X is a variable and A is a conjunction of literals containing no variables except X;
- R (*result*) is a list $[\mathtt{r}_1(\mathtt{X}_1)@(\mathtt{t}_1), \ldots, \mathtt{r}_\mathtt{m}(\mathtt{X}_\mathtt{m})@(\mathtt{t}_\mathtt{m})]$, $m > 0$ and $t_1 \leq \ldots \leq t_m \leq t^S_{max}$, where r_i is any predicate symbol, say with arity k_i, and X_m is a list of k_i terms. □

Given a query $\mathtt{Q} = \langle \mathtt{S}, \mathtt{G}, \mathtt{R} \rangle$ where $\mathtt{R} = [\mathtt{r}_1@(\mathtt{t}_1), \ldots, \mathtt{r}_\mathtt{m}@(\mathtt{t}_\mathtt{m})]$, on a $\mathtt{DATALOG}^{\mathtt{ev!}}$ program P, and a database D, an (nondeterministic) *answer* of Q on D, denoted by $Q(D)$, is either:

1. the list of relations $[\mathbf{r}(X_1)_1, \ldots, \mathbf{r}(X_i)_m]$ such that $\mathbf{r}_i = \{x_i | r'_i(x_i, t_i, max_{subT}(M, t_i)) \in M$ and x_i unifies with $X_i\}$, where r' is the temporal version of the predicate symbol r and M is a *Q-filtered* stable model of $sv(P_D) \cup sv(S)$,
2. or (ii) the empty list if there is no Q-filtered stable model.

Assume, in our running example, the company has planned to have a number of requests, constituting a scenario $\mathcal{S}$. The designer want to know the possible evolutions; this aim can be achieved by supplying the query $\langle \mathcal{S}, \emptyset, \mathtt{R} \rangle$. Indeed, the list R, in the case is not empty, stores the log of the executions that satisfy the goal G, for a given scenario S.

Let t_{max} be the sum of all the durations of the tasks, declared by any server. Observe that such t_{max} is an upper bound on the completion time of any instance,

t^S_{max}. The following proposition states that our model is sound and complete w.r.t. the satisfaction of the global constraints (the predicate `executed(ID)` is the one defined in the previous section).

Proposition 1. *Let $\mathcal{WS}$ be a workflow schema,* **Constr**$(\mathcal{WS})$ *a set of global constraint, and Q be the query* $\langle \mathtt{init}(\mathtt{id_1})@(0), \exists(\mathtt{executed}(\mathtt{id_1})@(\mathtt{t_{max}}), \mathtt{R}\rangle)$ *on the program* $\mathcal{P}(\mathcal{WS}) \cup \mathcal{P}_{\mathbf{Constr}}(\mathcal{WS})$*. Then, R is empty if and only if there exists no instance that satisfies the given constraints.* □

7 Conclusions and Further Work

We have presented a new formalism which combines a rich graph representation of workflow schemes with simple (i.e., stratified), yet powerful DATALOG rules to express complex properties and constraints on executions. We have shown that our model can be used as a run-time environment for workflow execution, and as a tool for reasoning on actual scenarios. The latter aspects gives also the designer the ability of finding bugs in the specifications, and of testing the system's behavior in real cases.

On this way, our long-term goals is to devise workflow systems that automatically fix "improperly working" workflows (typically, a workflow systems supply, at most, warning message when detecty such cases). In order to achieve this aim, we shall investigate formal methods that are able to understand when a workflow system is about to collapse, to identify optimal scheduling of tasks, and to generate improved workflow (starting with a given specification), on the basis of some optimality criterion.

References

1. G. Alonso, C. Hagen and A. Lazcano. Processes in electronic commerce. In *ICDCS Workshop on Electronic Commerce in Web-Based Applications*, pages ??-??, 1999.
2. A. Bonner. Workflow, Transactions, and Datalog. In *Proc. of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 294–305, 1999.
3. H. Davulcu, M. Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic Based Modeling and Analysis of Workflows. In*Proc. 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 25–33, 1998.
4. D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2), pages 119–153, 1995.
5. S. Greco, D. Saccà and C. Zaniolo. Extending Stratified Datalog to Capture Complexity Classes Ranging from P to QH. In *Acta Informatica*, 37(10), pages 699–725, 2001.
6. A. Guzzo and D. Saccà. Modelling the Future with Event Choice DATALOG. Proc. AGP Conference,, pages 53-70, September 2002.

7. G. Kappel, P. Lang, S. Rausch-Schott and W. Retschitzagger. Workflow Management Based on Object, Rules, and Roles. *Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society*, 18(1), pages 11–18, 1995.
8. P. Muth, J. Weienfels, M. Gillmann, and G. Weikum. Integrating Light-Weight Workflow Management Systems within Existing Business Environments. In *Proc. 15th Int. Conf. on Data Engineering*, pages 286–293, 1999.
9. P. Senkul, M. Kifer and I.H. Toroslu. A logical Framework for Scheduling Workflows Under Resource Allocation Constraints. In *VLDB*, pages ??–??, 2002.
10. D. Saccà and C. Zaniolo. Stable Models and Non-Determinism in Logic Programs with Negation. In *Proc. ACM Symp. on Principles of Database Systems*, pages 205–218, 1990.
11. M. P. Singh. Semantical considerations on workflows:An algebra for intertask dependencies. In *Proc. of the Int. Workshop on Database Programming Languages*, pages 6–8, 1995.
12. W. M. P. van der Aalst. The Application of Petri Nets to Worflow Management. *Journal of Circuits, Systems, and Computers*, 8(1), pages 21–66, 1998.
13. W. M. P. van der Aalst, A. Hirnschall and H.M.W. Verbeek. An Alternative Way to Analyze Workflow Graphs. In *Proc. of the 14th Int. Conf. on Advanced Information Systems Engineering*, pages 534–552, 2002.
14. D. Wodtke, and G. Weikum. A Formal Foundation for Distributed Workflow Execution Based on State Charts. In *Proc. 6th Int. Conf. on Database Theory (ICDT97)*, pages 230–246, 1997.
15. D. Wodtke, J. Weissenfels, G. Weikum, and A. Dittrich. The Mentor project: Steps towards enterprise-wide workflow management. In *Proc. of the IEEE International Conference on Data Engineering* ,pages 556–565, 1996.
16. The Workflow Management Coalition, *http://www.wfmc.org/*.
17. Zaniolo, C., Transaction-Conscious Stable Model Semantics for Active Database Rules. In *Proc. Int. Conf. on Deductive Object-Oriented Databases*, 1995.
18. Zaniolo, C., Active Database Rules with Transaction-Conscious Stable Model Semantics. In *Proc. of the Conf. on Deductive Object-Oriented Databases*, pp.55–72, LNCS 1013, Singapore, December 1995.
19. Zaniolo, C., Arni, N., and Ong, K., Negation and Aggregates in Recursive Rules: the LDL++ Approach, *Proc. 3rd Int. Conf. on Deductive and Object-Oriented Databases*, 1993.