A Logic Programming Approach for Planning Workflows Evolutions

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. Most of the formalism for modeling workflow schemes are based on graphical representations. Even though such approaches lead to intuitive specifications, they usually lack of the ability of specifying complex structural properties.
This paper illustrates how a recent extension of DATALOG, enriched with features for dealing with events and nondeterministic choice, is well suited for modeling workflows and expressing complex properties and constraints on executions. Moreover, the language provides a querying mechanism for simulating executions, by fixing an initial state and an execution scenario, and for planning future choices in order to achieve a given goal.

1 Introduction

A great deal of recent research concerns the task of modelling 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. In particular, the most common approach is the use of a *control flow graph*, in which the workflow is represented by a labelled directed graph whose nodes correspond the task to be performed, and the arcs describe the precedences among them.

As pointed out by many authors, see e.g [?], the essential drawback of approaches based on the *control flow graph* is their limited expressive power: they are only able to specify *local* dependencies whereas properties such as synchronization, concurrency, or serial execution of tasks, also called in the literature *global constraints*, cannot be described. The current trend in workflow management system is to left unstated all the complex constraints (thus delivering an incomplete specification) or to eventually expressed them using other formalisms, e.g., some form of logics.

In this paper, we present an overview of a system which realizes a logic based formalism which combines a rich graphical representation of workflow schemes with simple (i.e., locally stratified), yet powerful `DATALOG` rules to express complex properties and *global constraints* on executions. Both the graph representation and the `DATALOG` rules are mapped into a unique program in $\texttt{DATALOG}^{ev!}$, that is a recent extension of `DATALOG` for handling events. A distinguished feature of our formalism is the ability to instantiate several times the same task so that complex flows can be captured by succinct definitions. This formalism very much simplifies the model proposed in [?] and provides a powerful ground for an efficient implementation of the system using the **DLV** system [?], with the aim of obtaining an effective tool for simulating and reasoning on workflows. Concerning the translation of $\texttt{DATALOG}^{ev!}$ programs into **DLV** programs, we mention that in [?] some of the authors have already shown how to compile them into a classical logic programming framework.

2 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: $\langle A, E, a_0, F, A^{\wedge}_{in}, A^{\vee}_{in}, A^{*}_{in}, A^{*\wedge}_{in}, A^{*\vee}_{in}, A^{\wedge}_{out}, A^{\vee}_{out}, A^{L}_{out}, E^L, \lambda, L\rangle$ where:

- $\langle\{a_0\}, A^{\wedge}_{in}, A^{\vee}_{in}, A^{*}_{in}, A^{*\wedge}_{in}, A^{*\vee}_{in}\rangle$ and $\langle F, A^{\wedge}_{out}, A^{\vee}_{out}, A^{L}_{out}\rangle$ are two partitions of A — the nodes a in A, the unique initial task a_0 and the final tasks b in F are defined by the predicates `task(a)`, `startTask(a₀)`, `finalTask(b)`, respectively; every task returns a label in L after its execution that is used for the possible activation of labelled arcs — a special label is "fail" which notifies an abnormal execution of the task;
- each task a can be either: (i) a *regular* task, defined by `regularTask(a)`, which can be executed only once for each occurrence of the workflow, or (ii) a *replicated* task, defined by `replicatedTask(a)`, which admits several executions for the same occurrence of the workflow; the replicated tasks are all the nodes in $A^{*}_{in} \cup A^{*\wedge}_{in} \cup A^{*\vee}_{in}$, whereas all other nodes are regular; regular tasks are identified by their name and the identifier of the workflow instance whereas replicated tasks need an additional identifier to distinguish the various instances, i.e., the key is the triple (`WorkflowID, TaskName, TaskID`);
- the arcs in E are defined by the predicates `arc(PrecTask, NextTask)`;
- $E^L \subseteq E$ are arcs labelled with symbols in L by the function $\lambda : E^L \to L$ and are defined by the predicate `arcLabel(PrecTask, NextTask, Label)` (these arcs can be activated only if the outcome of the task `PrecTask` coincides with the label of the arc; we require the source node in a labelled arc to belong to A^{L}_{out});
- each task `a` in $A^{\wedge}_{in}$, defined by `inAND(a)`, acts as a synchronizer (also called a *and-join* task in the literature), thus, `a` cannot be started until after all its incoming arcs are activated; observe that, in the case of an incoming arc leaving a replicated task, say `p`, the arc must be activated for each replication of `p` before `a` can be started;
- each task `a` in $A^{\vee}_{in}$, defined by `inOR(a)`, is a *or-join* task and can be started as soon as one of its incoming arcs is activated; notice that, in the case of an incoming arc leaving a replicated task, say `p`, the task `a` is started even if the arc is activated for only one replication of `p`;

- each task a in A^*_{in}, defined by `inRep(a)`, has exactly one preceding task, say p, and p is not replicated; once the arc (p, a) is activated, a number of instances for a are started according to specific criteria specified for each workflow instance; for each replicated task b, *start**(b) defines the task a $\in A^*_{in}$ such that there is a path P from a to b consisting of all replicated tasks, and no other path P' can extend P with other replicated tasks — it can be shown that for each replicated task b there exists exactly one task in A^*_{in} which satisfies the above condition;
- each task a in $A^{*\wedge}_{in}$, defined by `inRepAND(a)`, has all preceding tasks replicated and for each two of its preceding tasks, say p_1 and p_2, *start**(p_1) = *start**(p_2); the task a may have several instances, one for each instance of *start**(a), say with task identifier *tid*, and the task identifier of each instance of a coincides with *tid*; an instance of a with identifier *tid* is actually started if each incoming arc leaving a task, say p, is activated for the instance of p with identifier *tid*;
- each task a in $A^{*\vee}_{in}$, defined by `inRepOR(a)`, has all preceding tasks replicated and for each two of its preceding tasks, say p_1 and p_2, *start**(p_1) = *start**(p_2); the task a may have several instances, one for each instance of *start**(a), say with task identifier *tid*, and the task identifier of each instance of a is *tid*; an instance of a with identifier *tid* is actually started as soon as one of its incoming arcs leaving a replicated task, say p, is activated for for the instance of p with identifier *tid*;
- each task a in $A^{\wedge}_{out}$, defined by `outAND(a)`, activates all its outgoing arcs; if a is replicated, every arc is activated several times, one for each instance of a;
- each task a in $A^{\vee}_{out}$, defined by `outOR(a)`, activates exactly one of its outgoing arcs, that is non-deterministically chosen; if a is replicated, the activation of one arc is repeated for each instance of a and two instances of a may activate different arcs, thus the instances of a make their non-deterministic choice independently from each other;
- each task a in A^{L}_{out}, defined by `outLabel(a)`, activates those outgoing arcs whose labels coincide with the label returned by a after completion; if a is replicated, an arc may be activated several times, one for each instance of a and the label of the arc must be checked against the outcome of the related instance.

Example 1. An example of workflow schema is shown in Figure 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* will activate both outgoing arcs after completion. Note that the task *VerifyAvailability* is a task in A^*_{in} and hence it is instantiated for each store. Then, each instance, characterized by a unique task identifier, either notifies to the task *OneAvailable* that the requested amount

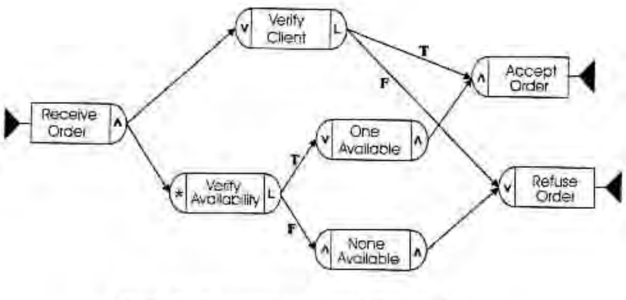

Fig. 1. Example of workflow schema.

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. Indeed, both the tasks *NoneAvailable* and *OneAvailable* have the effect of dropping the quantifications over the stores. Finally, 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. □

An *instance* $\mathcal{WI}$ of a workflow schema $\mathcal{WS}$ is a directed graph whose nodes correspond to the nodes in $\mathcal{WS}$ and are denoted by a pair $\langle a, ID\rangle$, where a is the corresponding node in $\mathcal{WS}$ and ID is the task identifier if a is replicated or 1 otherwise. Moreover, each node in $\mathcal{WI}$ is marked as: (i) *activated* if the corresponding task has been activated by at least one incoming arc but it did not start because of some missing condition on its precedences, or (ii) *executed* with an output label from L. The following conditions on $\mathcal{WI}$ hold:

- the node $\langle a_0, 1\rangle$ is in $\mathcal{WI}$ and is marked as executed;
- for each task a in $A^{\wedge}_{in}$, the node $\langle a, 1\rangle$ is in $\mathcal{WI}$ if there is at least one incoming arc $(\langle p, tid\rangle, \langle a, 1\rangle)$ in $\mathcal{WI}$; moreover, the node $\langle a, 1\rangle$ is marked executed (and not just activated) if for each $(p, a) \in A$, either (i) the node $\langle p, 1\rangle$ and the arc $(\langle p, 1\rangle, \langle a, 1\rangle)$ are in $\mathcal{WI}$ if p is regular or (ii) otherwise, there is at least one node $\langle r, tid\rangle$ in $\mathcal{WI}$, where $r = start^*(p)$, and tid is a task identifier, and for each node $\langle r, tid\rangle$ in $\mathcal{WI}$, the node $\langle p, tid\rangle$ and the arc $(\langle p, tid\rangle, \langle a, 1\rangle)$ are in $\mathcal{WI}$; if some of the above condition is not satisfied but
- for each task a in $A^{\vee}_{in}$, the node $\langle a, 1\rangle$ is in $\mathcal{WI}$ if there exists an arc $(p, a) \in A$ such that a node $\langle p, tid\rangle$ and an arc $(\langle p, tid\rangle, \langle a, 1\rangle)$ are in $\mathcal{WI}$; note that such nodes are always marked executed as there are no preconditions;
- for each task a in A^{*}_{in}, given a a task identifier tid, the node $\langle a, tid\rangle$, is in $\mathcal{WI}$ and is marked executed as well if both the node $\langle p, 1\rangle$ and the arc $(\langle p, 1\rangle, \langle a, tid\rangle)$ are in $\mathcal{WI}$;
- for each task a in $A^{*\wedge}_{in}$, the node $\langle a, tid\rangle$ is in $\mathcal{WI}$ if there is at least one incoming arc $(\langle p, tid\rangle, \langle a, tid\rangle)$ in $\mathcal{WI}$; moreover, the node $\langle a, tid\rangle$ is marked

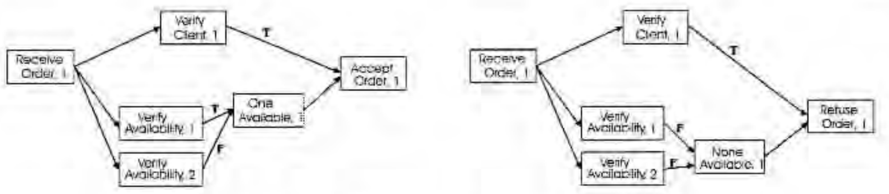

Fig. 2. Example of workflow instances.

executed if for each $(p, a) \in A$, the $\langle p, tid\rangle$ and the arc $(\langle p, tid\rangle, \langle a, tid\rangle)$ are in $\mathcal{WI}$;

- for each task a in $A_{in}^{*\vee}$, the node $\langle a, tid\rangle$ is in $\mathcal{WI}$ and is marked executed as well if there exists an arc $(p, a) \in A$ such that the $\langle p, tid\rangle$ and the arc $(\langle p, tid\rangle, \langle a, tid\rangle)$ are in $\mathcal{WI}$;
- for each node $\langle a, tid\rangle$ in $\mathcal{WI}$ that is marked executed, say with output label l, (i) if a is in $A_{out}^{\wedge}$, then for each arc $(a, b) \in E$, the arc $(\langle a, tid\rangle, \langle b, tid\rangle)$ is in $\mathcal{WI}$; (ii) if a in $A_{out}^{\vee}$, then there exactly one arc in $\mathcal{WI}$ leaving $\langle a, tid\rangle$, say $(\langle a, tid\rangle, \langle b, tid\rangle)$, and the arc (a, b) is in $\mathcal{WS}$; (iii) if a in A_{out}^{L}, then for each arc $(a, b) \in E$ with label l, the arc $(\langle a, tid\rangle, \langle b, tid\rangle)$ is in $\mathcal{WI}$.

Example 2. Two examples of workflow instances for the schema in Figure 1 are shown in Figure 2, where we only report the tasks that have been executed. In particular, we assume that the company has two stores. On the left, the order has been accepted, as the requested amount is in the first store, while on the right the order has been rejected since none of the store has enough availability. □

3 Description of Servers and Task Executions

In the previous section we have described the concept of workflow instance, under a "static" perspective, as the tasks can be either activated or executed. We next enrich the model by also considering the servers that are allowed to execute the tasks, and consequently we provide a more detailed analysis of the possible states of the tasks during their execution.

In the following, we associate to each workflow instance $\mathcal{WI}$ a unique identifer *WID*, and, in order to simplify the presentation, a predicate p(*WID*, X), where X is a generic list of arguments, is denoted by $\mathtt{p}_{WID}$(X).

We are given a set of servers which (or who) are defined by the predicate `server(ServerName)` and that will be used for the executions of the various tasks. Actually the predicate `executable(S,T,D)` states that the server `S` can execute the task `T` and the execution will have the duration `D`. Moreover, the predicate `outOfOrder(S)` states that the server `S` cannot be temporally used for any execution. If not out of order, a server `S` is available for a new task execution if it is not busy — the availability is checked with the following `DATALOG` rule:

$$\begin{aligned}\texttt{available(Server)} \leftarrow\ & \texttt{executable(Server,}\,\_,\_\texttt{)},\ \neg\,\texttt{outOfOrder(Server)},\\ & \neg\,(\ \texttt{startRunning}_{WID}\texttt{(Task, TaskIdentifier, Server,}\,\_\texttt{)},\\ & \quad \neg\,\texttt{executed}_{WID}\texttt{(Task, Quantifiers,}\,\_,\_\texttt{)}\ ).\end{aligned}$$

where startRunning and ended are predicates defined below in this section. Note that in order to characterize the task that a server is running, we explicitly need to consider not only its name (Task), but also its identifier (TaskIdentifier), eventually different from 1 in the case of replicated node.

In the above rule, 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 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. We shall use this notation also in the rest of the paper.

A task (or a task instance if it is replicated) is in one of the following states: (1) *idle*, thus the task is not yet activated as none of its incoming arcs are active; (2) *activated*, thus the task has received the notification for its execution from at least one incoming arc but it is not yet started as it needs the activation of some additional incoming arcs; (3) *ready*, i.e., the task is ready for execution and has been started but it is waiting for the assignment of a server; (4) *running*, i.e., the task is currently executed by a server; (5) *executed*, i.e., the task has been terminated.

Given a workflow instance *WID*, the state of the execution of a task T with identifier TID is kept by means of the following relations: $\texttt{startActive}_{WID}(\texttt{T},\texttt{TID},\texttt{Time})$, storing the time when the task $\langle\texttt{T},\texttt{TID}\rangle$ was activated; $\texttt{startReady}_{WID}(\texttt{T},\texttt{TID},\texttt{Time})$, storing the time when the task $\langle\texttt{T},\texttt{TID}\rangle$ was declared ready for execution; $\texttt{startRunning}_{WID}(\texttt{T},\texttt{TID},\texttt{S},\texttt{Time})$, storing the time a server S has started its execution; $\texttt{executed}_{WID}(\texttt{T},\texttt{TID},\texttt{Time},\texttt{Output})$, storing the time when the execution of the task $\langle\texttt{T},\texttt{TID}\rangle$ is completed and the result Output of the execution — recall that Output is a label in L. The state of a task $\langle\texttt{T},\texttt{TID}\rangle$ can be derived using simple DATALOG rules and will be acced with the predicate $\texttt{state}_{WID}(\texttt{T},\texttt{TID},\textit{stateType})$.

Finally the fact that an instance of an arc (Prec, Next) has been activated from an instance TIDP of the task Prec to an instance TIDN of the task Next is stored in the predicate activeArc(Prec, TIDP, Next, TIDN).

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

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. The first event, called init, is an external event which starts a new workflow instance at a certain time. The predicate $\texttt{started}_{WID}$ that is used for keeping trace of the fact that a new instance *WID* has been started.

$$\begin{array}{l}\mathtt{r}_1: \ [\mathtt{init(WID)@(T)}] \\ \qquad \mathtt{run()\text{++}},\ \mathtt{started}_{WID}(),\ \mathtt{startReady}_{WID}(\mathtt{ST},\mathtt{1},\mathtt{T}) \leftarrow \mathtt{startTask(ST)}.\end{array}$$

Every time the event $\mathtt{run()@(T)}$ 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 $\mathtt{unsat}_{WID}()$ 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 $\mathtt{executed}_{WID}()$ is true if the workflow instance has already entered a final state so that no other task needs to be performed.

$$\begin{array}{lll}\mathtt{r}_2: \ [\mathtt{run()@(T)}] & & \\ \quad \mathtt{evaluate}_{WID}(\mathtt{Task},\mathtt{L},\mathtt{Duration})\text{++} & & \\ \quad \mathtt{startRunning}_{WID}(\mathtt{Task},\mathtt{TID},\mathtt{Server},\mathtt{T}) & \leftarrow & \neg\mathtt{unsat}_{WID}(),\ \neg\mathtt{executed}_{WID}(), \\ & & \mathtt{state}_{WID}(\mathtt{Task},\mathtt{TID},\mathit{ready}), \\ & & \mathtt{available(Server)}, \\ & & \mathtt{executable(Server,Task,Duration)} \\ & & \otimes\ \mathtt{choice((Task),(Server))} \\ & & \otimes\ \mathtt{choice((Server),(Task))}.\end{array}$$

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.

$$\begin{array}{l}\mathtt{r}_3: \ [\mathtt{evaluate}_{WID}(\mathtt{Task},\mathtt{TID},\mathtt{Duration})\mathtt{@(T)}] \\ \quad \mathtt{complete}_{WID}(\mathtt{Task},\mathtt{L},\mathtt{Output})\mathtt{+(Duration)},\ \leftarrow \mathbf{evaluation}_{WID}(\mathtt{Task},\mathtt{TID},\mathtt{Output}).\end{array}$$

The predicate $\mathbf{evaluation}_{WID}(\mathtt{Task},\mathtt{L},\mathtt{Output})$ 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 $\mathtt{T}+\mathtt{Duration}$. As described in the next event, after the completion of a task, the selection of which of its successor tasks to 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 $\mathtt{unsat}_{WID}(\mathtt{T})$ is added only if the predicate $\mathbf{unsatGC}_{WID}(\mathtt{Task},\mathtt{L})$ 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. For a final task, if the global constraints are satisfied then we can register the successful execution of the workflow instance.

```
r4 : [complete_WID(Task, TID, Output)@(T)]
     run()++, executed_WID(Task, TID, T, Output).
     unsat_WID()                        ← unsatGC_WID(Task, L).
     executed_WID()                     ← finalTask(Task), ¬ unsatGC_WID(Task, TID).
     activateTask_WID(Next, TID, Task)++,  ← outOR(Task), Output ≠ "fail", arc(Task, Next)
                                          ⊗ ChoiceAny().
     activateTask_WID(Next, TID, Task)++   ← outAND(Task), Output ≠ "fail", arc(Task, Next).
     activateTask_WID(Next, TID, Task)++   ← outLabel(Task), Output ≠ "fail",
                                          arcLabel(Task, Next, Label), Label = Output.
```

The event activateTask is used for activating the target task in an arc. If the task is in $A^{\vee}_{in} \cup A^{*}_{in} \cup A^{*\vee}_{in}$ the activation also implies that the task is ready for the execution; in the other cases we have to check more elaborated conditions by means of the event checkForReadY. In the case *Task* is in A^{*}_{in}, then for each NewTID stored in the relation quantifyTID, a new instance of the task is activated and becomes immediately ready. We also keep track of all activated arcs.

```
r5 : [activateTask_WID(Task, TID, Prec)@(T)]
     run()++,
     activeArc_WID(Prec, TID, Task, NewTID),
     startActive_WID(Task, NewTID, T),
     startReady_WID(Task, NewTID, T),      ← inRep(Task), quantifyTID(Task, NewTID).
     activeArc_WID(Prec, TID, Task, TID)   ← inRepOR(Task).
     run()++,
     startActive_WID(Task, TID, T),
     startReady_WID(Task, TID, T)          ← inRepOR(Task), ¬state_WID(Task, TID, active).
     activeArc_WID(Prec, TID, Task, 1)     ← inOR(Task).
     run()++,
     startActive_WID(Task, 1, T),
     startReady_WID(Task, 1, T)            ← inOR(Task), ¬state_WID(Task, _, active).
     checkForReady_WID(Task, TID)++,
     activeArc_WID(Prec, TID, Task, TID)   ← ¬activeArc_WID(Prec, TID, Task, TID), inRepAND(Task).
     startActive_WID(Task, TID, T)         ← ¬state_WID(Task, TID, active), inRepAND(Task).
     checkForReady_WID(Task, 1)++,
     activeArc_WID(Prec, TID, Task, 1)     ← ¬activeArc_WID(Prec, TID, Task, 1), inAND(Task).
     startActive_WID(Task, 1, T)           ← ¬state_WID(Task, 1, active), inAND(Task).
```

The event checkForReady decides whether a given activated task corresponding to an and-join is ready for execution.

```
r6 : [checkForReady_WID(Task, TID)@(T)]
     run()++,
     startReady_WID(Task, TID, T)     ← inANDRep(Task), ¬state_WID(Task, TID, ready),
                                        ¬ (arc(Prec, Task), ¬ activeArc_WID(Prec, TID, Task, TID)).
     startReady_WID(Task, 1, T)       ← inAND(Task), ¬state_WID(Task, 1, ready),
                                        ¬ (arc(Prec, Task), possInstance(Prec, TIDP),
                                        ¬ activeArc_WID(Prec, TIDP, Task, 1)).
```

where

```
possInstance(Task, 1)   ← regular(Task).
possInstance(Task, TID) ← replicated(Task), start* (Task, TaskR),
                          startReady(TaskR, TID, _).
start* (Task, TaskR)    ← arc(Prec, Task), replicated(Prec), start* (Prec, TaskR).
start* (Task, Task)     ← inRep(Task).
```

5 Global Constraints

As shown in [?], a number of *global constraints* and additional constraints on the scheduling of the activities can be translated into a DATALOG$^{\mathrm{ev!}}$ program $\mathbf{P_{Constr}}(\mathcal{WS})$. In this section, we complete the model by showing how to specify global constraints. Let us first formalize the notion of *global constraint* over a workflow schema $\mathcal{WS}$: (i) for any $a \in A$, $!a$ (resp. $\neg!a$) is a *positive* (resp., *negative*) *primitive* global constraint, (ii) given two positive primitive global constraints c_1 and c_2, $c_1 \prec c_2$ is a *serial* global constraint, (iii) 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 also mapped into a set of DATALOG$^{\mathrm{ev!}}$ rules as follows:

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

$$\begin{array}{l}\texttt{unsatGC1}_{WID}(\texttt{c},\texttt{gs}) \leftarrow \texttt{ended}_{WID}(\texttt{a},\_,\_,\texttt{O}), \texttt{O} = \text{“}\texttt{fail}\text{”}.\\ \texttt{unsatGC1}_{WID}(\texttt{c},\texttt{gs}) \leftarrow \neg\,\texttt{ended}_{WID}(\texttt{a},\_,\_,\_).\end{array}$$

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:

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

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

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

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

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

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

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

Let us now define the predicate $\mathbf{unsatGC}_{WID}(\texttt{Task}, \texttt{L})$ 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 :

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

Observe that we do not check satisfaction for constraints which are only used as sub-expressions; moreover, we point out 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 $\mathbf{unsatGC}_{WID}(a_2, L) \leftarrow \mathbf{unsatGC1}_{WID}(c, g)$. 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 $\texttt{executed}_{WID}()$ or $\texttt{unsat}_{WID}()$, 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:

$$\begin{aligned}
&\texttt{failed}_{WID}() \leftarrow \texttt{unsat}_{WID}().\\
&\texttt{failed}_{WID}() \leftarrow \texttt{started}_{WID}(),\ \neg\,\texttt{executed}_{WID}(), \neg\,\texttt{working}_{WID}().\\
&\texttt{working}_{WID}() \leftarrow \texttt{state}_{WID}(\texttt{T}, \_, \texttt{ready})).\\
&\texttt{working}_{WID}() \leftarrow \texttt{state}_{WID}(\texttt{T}, \_, \texttt{running})).
\end{aligned}$$

6 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, AND C. Hagen. Flexible Exception Handling in the OPERA Process Support System. In *18th International Conference on Distributed Computing Systems (ICDCS)*, pages 526–533, 1998.
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. 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.
5. G. Greco, A. Guzzo and D. Saccà. Reasoning on Workflow Executions. Proc. ADBIS Conference,, September 2003.
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. Eiter T., Leone N., Mateis C., Pfeifer G. and Scarcello F.. A Deductive System for Non-monotonic Reasoning. *Proc. LPNMR Conf.*, 363-374, 1997.
9. 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.
10. P. Senkul, M. Kifer and I.H. Toroslu. A logical Framework for Scheduling Workflows Under Resource Allocation Constraints. In *VLDB*, pages 694-705, 2002.
11. 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.
12. The Workflow Management Coalition, *http://www.wfmc.org/*.
13. Zaniolo, C., Transaction-Conscious Stable Model Semantics for Active Database Rules. In *Proc. Int. Conf. on Deductive Object-Oriented Databases*, 1995.
14. 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.
15. 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.