# A Logic Framework for Reasoning on Workflow Executions

G. Greco, A. Guzzo, D. Saccà

**RT-ICAR-CS-04-11** **Settembre 2004**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

# Mining and Reasoning on Workflows

G. Greco[1], A. Guzzo[1], D. Saccà[1,2]

***Rapporto Tecnico N.:***
**RT-ICAR-CS-04-11**

***Data:***
**Settembre 2004**

[1] Università degli Studi della Calabria, DEIS, Via P. Bucci 41C, Rende (CS)
[2] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Cosenza, Via P. Bucci 41C, 87036 Rende(CS)

*I rapporti tecnici dell'ICAR-CNR sono pubblicati dall'Istituto di Calcolo e Reti ad Alte Prestazioni del Consiglio Nazionale delle Ricerche. Tali rapporti, approntati sotto l'esclusiva responsabilità scientifica degli autori, descrivono attività di ricerca del personale e dei collaboratori dell'ICAR, in alcuni casi in un formato preliminare prima della pubblicazione definitiva in altra sede.*

# A Logic Framework for Reasoning on Workflow Executions [1]

Gianluigi Greco [a], Antonella Guzzo [a], Domenico Saccà [a,b,*]

[a] *DEIS, University of Calabria, Via Pietro Bucci 41C, 87036 Rende, Italy*

[b] *ICAR, CNR, Via Pietro Bucci 41C, 87036 Rende, Italy*

**Abstract**

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. 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. The high expressive power of both the graphical formalism and the $\texttt{DATALOG}^{\texttt{ev!}}$ rules provides the designer with powerful mechanisms for reasoning on workflows: (i) modeling a workflow schema with the possibility of expressing many types of constraints on the executions, (ii) defining various execution scenarios (i.e., sequences of workflow executions for the same schema) and (iii) simulating the actual behavior of the modeled scheme by fixing an initial state and an an execution scenario and querying the state after such executions. As a scenario may include a certain amount of non-determinism, the designer can also verify under which conditions a given (desirable or undesirable) goal can be eventually achieved.

*Key words:* Logic Programming, Work Flow Management, Modelling Languages, Intelligent Systems

* **Corresponding Author.** Phone: +39 0984 494750. Fax. +39 0984 839054
*Email addresses:* ggreco@si.deis.unical.it (Gianluigi Greco), guzzo@si.deis.unical.it (Antonella Guzzo), sacca@icar.cnr.it (Domenico Saccà).

[1] A preliminary version of part of this paper appeared in the proceedings of the 7th Int. Conf. on Advances in Databases and Information Systems, ADBIS'03.

## 1 Introduction

Workflow management systems (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. In order to give a simple and intuitive description of the workflow structure, most of such formalisms are based on graphical representations, such as the *control flow graph*, in which the workflow is represented by a labelled directed graph whose nodes represents the tasks to be performed, and whose arcs describe the precedences among them. Moreover, Workflow Management Coalition (WfMC [28]) has also identified additional controls, such as loops and sub-workflows.

An example of control flow which will be used throughout the rest of the paper, modelling a typical process for a selling company, is shown in Figure 1 by exploiting a notation whose meaning is informally presented below.

**Example 1** A customer issues a request to purchase a certain amount of a given product by filling in a request form on the browser (task *ReceiveOrder*). After completion, the task will activate both its outgoing arcs as it is denoted by the symbol $\wedge$ in output. Then the request is forwarded both 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 *VerifyAvailability* (marked in input with $*$) is instantiated for each store, and, hence, each instance, characterized by a unique task identifier, either notifies to the task *OneAvailable* that 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*, denoted by the symbol $\vee$ in input, is started as soon as one notification of availability is received, whereas the task *NoneAvailable*, denoted by $\wedge$ in input, 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. In parallel, after the execution of *VerifyClient*, if the client turns out not to be reliable, it is checked whether the reliability of the client should be further checked by more detailed investigations. Eventually a final decision will be made on whether the client is reliable for the order or not. 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. □

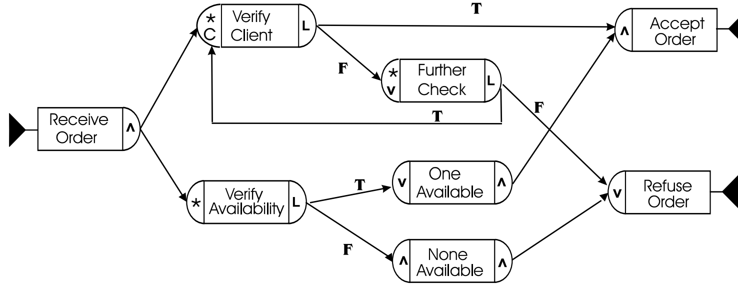


Fig. 1. Running Example: Sales Ordering Process.

Besides to an intuitive graphical notation, current workflow management systems do not provide other mechanisms for helping the designer in the modeling phase of a complex process. Thus, the usage of most of the available systems for treating real world cases with processes made of hundreds of tasks may be complicated for two major reasons:

(1) The languages adopted for the specifications are not enough expressive for modelling a number of properties, which cannot be captured by a graph. In fact, as pointed out by many authors (e.g., see [5]), the essential limitation of the approach based on the *control flow graph* lies in the ability of specifying *local* dependencies only, while complex properties, also called in the literature *global constraints*, are left unstated. For instance, in our running example, a natural global constraint is that the company will try to satisfy the request by looking at the store nearest to the client, in order to reduce transportation costs.
(2) There is no way for assessing and reasoning on the properties of the specification. For instance, prior to the enactment of the workflow it would be very useful to simulate the actual behavior of the modelled 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.

In this paper, we face these problems by proposing a logic based environment, which combines a rich graphical representation of workflow schemas with simple (i.e., stratified), yet powerful DATALOG rules to express complex properties and *global constraints* on executions. Both the graph representation, the DATALOG rules as well as all the data needed for the instantiation of the process are mapped into a unique program in $\texttt{DATALOG}^{\texttt{ev!}}$[11,10], that is a recent extension of DATALOG for handling events and temporal properties.

The attractive feature of this approach is that the resultant logic program may serve as *executable* logical specification, well suited for **reasoning** on its possible enactments and for being used as a run-time environment for

the **simulation**. The simulation is carried out by equipping $\texttt{DATALOG}^{\texttt{ev!}}$ with a powerful querying mechanism, leading the ability of supporting reasoning about events and actions. Thus, given a scenario, the problem consists in verifying whether there exist particular sequences of further event occurrences that eventually satisfy a given goal, and in returning a possible evolution which satisfies it.

**Example 2** Let us return to the selling company example. A typical scenario of execution consists of a number of requests that are planned in a certain period of time. For instance, by adopting the syntax of our language, the list $\texttt{H} : [\texttt{ReceiveOrder}(\texttt{id}_1, \texttt{c}_1, \texttt{i}_1, 5)@(0), \texttt{ReceiveOrder}(\texttt{id}_2, \texttt{c}_2, \texttt{i}_1, 10)@(2)]$ specifies that two orders of clients $\texttt{c}_1$ and $\texttt{c}_2$ are planned at times $\texttt{0}$ and $\texttt{2}$, consisting of the request of the item $\texttt{i}_1$ in quantity $\texttt{5}$ and $\texttt{10}$, respectively.

Since for each order, the requested products with the desired quantity are assumed to be taken from a single store, it is obvious that not all the possible schedules will eventually lead to the satisfaction of all the orders. One important feature of our language is the powerful querying mechanism, that can be used for planning and scheduling purposes; for instance, by simple supplying a query of the form $\exists^{@(\texttt{t})}\texttt{AcceptOrder}(\texttt{id}_2)$ for a suitable value of $\texttt{t}$, we could check whether there exists a workflow execution that lead to acceptance of $\texttt{id}_2$. Obviously, in the case the query is evaluated false, we are ensured that there is no way for satisfying such an order, and, hence, we can think at rejecting it in advance, or at planning a new production. □

### *1.1 Overview of the Proposal*

Our logic framework for reasoning on workflows has been implemented into a prototype system, consisting of a lightweight tool which can be put on the top of any pre-existing commercial workflow engine, thus providing some add-on functionalities that currently lacks in these products. The conceptual architecture of the system, shown in Figure 2, evidences the fact that it is completely independent on the particular *Enactment Engine* used in the organization. This can be achieved by the use of a wrapper which translates specifications written in the standards BPML and XPDL into $\texttt{DATALOG}^{\texttt{ev!}}$ programs (and vice-versa).

In the architecture we evidence the *User Interface* used for both designing workflow schemas and specifying a list of envisioned events, used for simulation purposes. The *WF-model wrapper* provides the ability of mapping workflows specified both in our graphical interface or in external engines into the system. Moreover, in the case an external workflow schema (e.g., written in BPML) is loaded, this module notifies the user interface for a proper displaying. All the information on the workflow model are stored in the *Metadata Repository* by using the internal representation language, $\texttt{DATALOG}^{\texttt{ev!}}$.

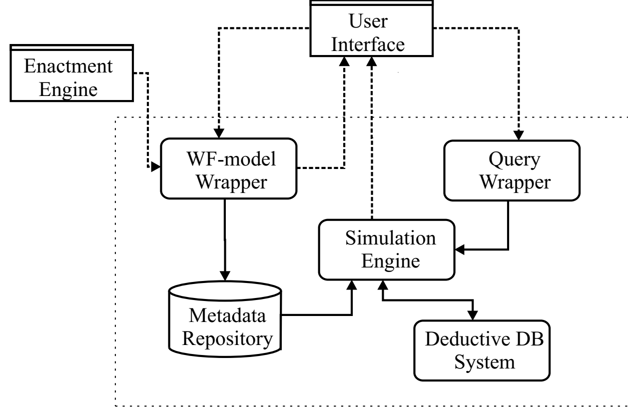


Fig. 2. Architecture of the prototype.

As it will be later discussed in more details, the $\texttt{DATALOG}^{\texttt{ev!}}$ program $\mathcal{P}(\mathcal{WS})$ modeling a workflow $\mathcal{WS}$ consists of three distinct elements:

- A set $\texttt{DB}_{ws}$ of facts used for modeling (i) the static aspects, i.e., the definition of the control flow prescribing the relationships of precedence among activities, (ii) the actual status of dynamic ones, i.e., the definition of the set of servers available and their actual scheduling on the tasks, and (iii) some other additional information needed for the run-time support;
- A set $\texttt{KDB}_{ws}$ of *dynamic rules* which are used for modeling the way in which the knowledge in the database $\texttt{DB}_{ws}$ is updated, thus, keeping trace of the evolution of an execution. For instance, in the sales ordering process we need some rule for choosing a store and for updating its quantity of products, after the selling;
- A set $\texttt{EV}_{ws}$ of *event activations* rules, which are used for modeling the enactment engine. Notice that this part is essentially independent of the particular type of workflow, since it prescribes in a declarative manner the procedures that must be followed while executing an instance, by specifying the way the tasks are scheduled on the servers and the rules according to which a task become ready for its execution; under this perspective, these rules provide the formal semantics for our specifications.

The specification $\mathcal{P}(\mathcal{WS})$ can be further extended by specifying, by means of other $\texttt{DATALOG}^{\texttt{ev!}}$ rules: *global constraints* and additional constraints on the scheduling of the activities, denoted by $\mathbf{Constr}(\mathcal{WS})$. Interestingly, the resulting program is still an executable specification (after a proper automatic translation which is performed by the *Simulation Engine* module according to the technique presented in [11]) into *Deductive Database Engines* such as DLV [14] or Smodels [18]. Thus, each time a simulation is required by the user, the module *Query wrapper* translates the requirements in a proper query, which is evaluated by the engine - in our application the DLV system, and the result is eventually supplied to the user interface.

### 1.2 Organization

The rest of the paper is organized as follows. A discussion on the basic concepts of workflow specification, together with the overview of the formal model we adopted, is provided in Section 2. The syntax and semantics of $\texttt{DATALOG}^{\texttt{ev!}}$ rules is briefly overview in Section 3. Then, in Section 4, we provide the formalization of the logic framework for workflow specification by discussing in details all the components of the program $\mathcal{P}(\mathcal{WS})$, for a given workflow $\mathcal{WS}$. The query language providing simulation capabilities, together with some complexity results assessing the 'intrinsic' difficulty in reasoning on workflow, is reported in Section 5. Finally, in Section 6 we compare our approach with other proposals in the literature and we draw our conclusions.

## 2 Formal Foundation of Workflows

A workflow is a partial or total automation of a business process, in which a collection of *activities* must be executed by some *servers* (humans or machines), according to certain *procedural rules*. Thus, any specification must focus both on some *static aspects*, i.e., the description of the relationships among activities not depending from a particular instance, and on some *dynamic aspects*, i.e., the description of workflow instances whose actual executions depend on status of the system (available servers and other resources).

### 2.1 Workflow Schema

We next introduce the notion of workflow schema which is the formal foundation for representing processes, by specifying a number of elementary tasks, along with flow relationships among them. We have decided not to refer to any particular formalisms proposed in the literature, even though the careful reader will notice that our specifications have many features in common with the syntax of the major commercial standards. In addition, we introduce the quite original concept of *replicated task*, i.e., of a task which admits several instantiations in the same execution, and whose usage may reduce the complexity of the modelling complex processes, by providing compact representations – this concept is actually available in the activity diagrams of UML. For instance, in our running example the task *Verify Availability* is instantiated for each store no matter of the number of stores, thus providing a more flexible definition of the process. Moreover, replicated tasks provide a formal way for defining the notion of cyclic sequence of tasks, where each task involved in a cycle may be instantiated different times.

**Definition 3 (Workflow Schema)** A *workflow schema* $\mathcal{WS}$ is a tuple $\langle \mathtt{A}, \mathtt{E}, \mathtt{a_0}, \mathtt{F}, \mathtt{A}^{\wedge}_{\mathtt{in}}, \mathtt{A}^{\vee}_{\mathtt{in}}, \mathtt{A}^{*}_{\mathtt{in}}, \mathtt{A}^{*c}_{\mathtt{in}}, \mathtt{A}^{*\wedge}_{\mathtt{in}}, \mathtt{A}^{*\vee}_{\mathtt{in}}, \mathtt{A}^{\wedge}_{\mathtt{out}}, \mathtt{A}^{\vee}_{\mathtt{out}}, \mathtt{A}^{\mathtt{L}}_{\mathtt{out}}, \mathtt{E}^{\mathtt{L}}, \lambda, \mathtt{L}\rangle$, where

- $\langle \mathtt{A}, \mathtt{E}\rangle$ is a graph, whose nodes $A$ are the tasks and whose arcs $E$ are the relationships of precedences among tasks;
- $\langle \{\mathtt{a_0}\}, \mathtt{A}^{\wedge}_{\mathtt{in}}, \mathtt{A}^{\vee}_{\mathtt{in}}, \mathtt{A}^{*}_{\mathtt{in}}, \mathtt{A}^{*c}_{\mathtt{in}}, \mathtt{A}^{*\wedge}_{\mathtt{in}}, \mathtt{A}^{*\vee}_{\mathtt{in}}\rangle$ and $\langle \mathtt{F}, \mathtt{A}^{\wedge}_{\mathtt{out}}, \mathtt{A}^{\vee}_{\mathtt{out}}, \mathtt{A}^{\mathtt{L}}_{\mathtt{out}}\rangle$ are partitions of $A$;
- $\mathtt{a_0}$ is the unique initial task;
- $\mathtt{F}$ is a set of final tasks;
- $\mathtt{L}$ is a set of labels, $\mathtt{E}^{\mathtt{L}}$ is a subset of $\mathtt{E}$ also called *labelled* arcs s.t. $\forall(\mathtt{a}, \mathtt{b}) \in \mathtt{E}^{\mathtt{L}}$, $\mathtt{a} \in \mathtt{A}^{\mathtt{L}}_{\mathtt{out}}$, and $\lambda : \mathtt{E}^{\mathtt{L}} \rightarrow \mathtt{L}$ is a function assigning a label to each arc in $E^L$.

All the nodes in $\mathtt{A}^{*}_{\mathtt{in}} \cup \mathtt{A}^{*c}_{\mathtt{in}} \cup \mathtt{A}^{*\wedge}_{\mathtt{in}} \cup \mathtt{A}^{*\vee}_{\mathtt{in}}$ are called *replicated tasks*, whereas all the others are called *regular tasks*. The following constraints hold:

- every task in $A-\{a_0\}$ has at least one incoming arc and every task in $a-F$ has at least one outgoing arc;
- each task $\mathtt{a} \in \mathtt{A}^{*}_{\mathtt{in}}$ has exactly one preceding task, say $\mathtt{p}$, and $\mathtt{p}$ is not replicated;
- each task $\mathtt{a} \in \mathtt{A}^{*c}_{\mathtt{in}}$ has exactly one preceding task $\mathtt{p}$ which is not replicated, and one preceding task $\mathtt{r}$ which is in $A^{*\wedge}_{in} \cup A^{*\vee}_{in}$ - every arc $(r, a)$ is called a *replication arc*; we require that the graph obtained from $\mathcal{WS}$ by removing all replication arcs be acyclic;
- for each task $a$ in $A^{*\wedge}_{in} \cup A^{*\vee}_{in}$ and for each $(b, a) \in E$, $b$ is replicated;
- for each task $b$ in $A^{*\wedge}_{in} \cup A^{*\vee}_{in}$ there exists a unique task $a$ in $A^{*}_{in} \cup A^{*c}_{in}$, denoted by $start*(b)$, such that there is a path from $a$ to $b$ consisting of all replicated tasks. $\Box$

Notice that we are assuming that each task returns a label in $\mathtt{L}$ 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. Then, an arc $(\mathtt{a}, \mathtt{b})$ in $\mathtt{E}^{\mathtt{L}}$ can be activated only if the outcome of the task $\mathtt{a}$ coincides with the label of the arc (we also require the tasks $\mathtt{a}$ to belong to $A^L_{out}$). The activations of the arcs determine the tasks that can be executed.

The informal semantics for the tasks (whose associated symbols adopted in our graphical formalism are shown in Figure 3) is as follows:

- Each task $\mathtt{a}$ in $\mathtt{A}^{*}_{\mathtt{in}}$ has exactly one preceding task, say $\mathtt{p}$, and $\mathtt{p}$ is not replicated; once the arc $(\mathtt{p}, \mathtt{a})$ is activated, a number of instances for $\mathtt{a}$, distinguished by a proper task identifier *tid*, are started according to specific criteria specified for each workflow instance;
- Each task $\mathtt{a}$ in $\mathtt{A}^{*c}_{\mathtt{in}}$ has exactly one preceding task $\mathtt{p}$ which is not replicated, and one preceding tasks $\mathtt{r}$ which is replicated; once the arc $(\mathtt{p}, \mathtt{a})$ is activated, an instance for $\mathtt{a}$ is started with $tid = 1$. Later on, each time the arc $(\mathtt{r}, \mathtt{a})$ is activated by some instance of $\mathtt{r}$ with $tid = t$, a new instance of $\mathtt{a}$ is created with a different $tid$ $t' = t + 1$ - but this may happen only if all replicated tasks $b$ for which $a = start*(b)$ and with $tid = t$ have been executed;

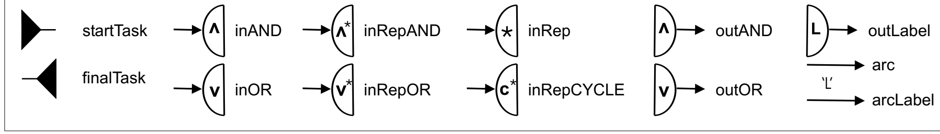


Fig. 3. Legend of the symbols for the graphical notation.

- Each task a in $\mathtt{A}^{\wedge}_{\mathtt{in}}$ 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 coming from a replicated task, say p, the arc must be activated for each replication of p before a can be started;
- Each task a in $\mathtt{A}^{\vee}_{\mathtt{in}}$ 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 $\mathtt{A}^{*\wedge}_{\mathtt{in}}$ has all preceding tasks replicated and for each two of its preceding tasks, say $\mathtt{p}_1$ and $\mathtt{p}_2$, $start^*(\mathtt{p}_1) = start^*(\mathtt{p}_2)$; the task a may have several instances, one for each instance of $start^*(\mathtt{a})$, 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 $\mathtt{A}^{*\vee}_{\mathtt{in}}$ has all preceding tasks replicated and for each two of its preceding tasks, say $\mathtt{p}_1$ and $\mathtt{p}_2$, $start^*(\mathtt{p}_1) = start^*(\mathtt{p}_2)$; the task a may have several instances, one for each instance of $start^*(\mathtt{a})$, 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 by the instance of p with identifier *tid*;
- Each task a in $\mathtt{A}^{\vee}_{\mathtt{out}}$ 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 $\mathtt{A}^{\wedge}_{\mathtt{out}}$ 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 $\mathtt{A}^{\mathtt{L}}_{\mathtt{out}}$ 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.

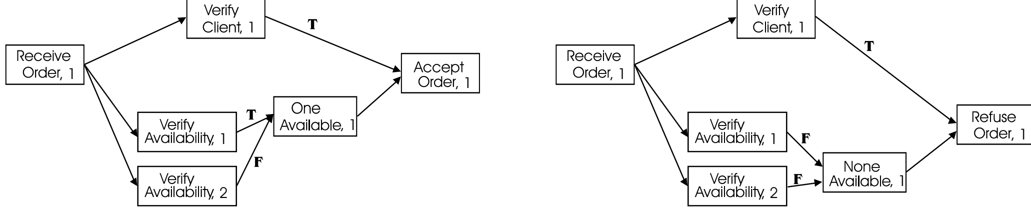


Fig. 4. Instances for the Sales Ordering Process.

### *2.2 Workflow Enactments*

We next turn to the dynamic aspects of workflow specification, and specifically in the way executions can be formalized.

In the following, given a task $\mathtt{a} \in \mathtt{A}$ of a workflow schema $\mathcal{WS}$, we denote by $\mathtt{t_a} = \langle \mathsf{a}, tid \rangle$ a generic *tasks instance* of $\mathtt{a}$, with $task(\mathtt{t_a}) = \mathtt{a}$, where *tid* is the task identifier (natural number) if $\mathtt{a}$ is replicated or 1 otherwise.

**Definition 4 (Instance)** An *instance ID* of a workflow schema $\mathcal{WS}$ over the nodes $\mathtt{A}$ and the labelled arcs $\mathtt{E}$ over $L$ is a tuple $\langle \mathtt{A}_{ID}, \mathtt{E}_{ID}, \lambda_{ID} \rangle$, where $task(\mathtt{A}_{ID}) \subseteq \mathtt{A}$, $\mathtt{E}_{ID} \subseteq \mathtt{A}_{ID} \times \mathtt{A}_{ID}$ s.t. $\forall (\mathtt{t_a}, \mathtt{t_b}) \in \mathtt{E}_{ID}$, $(task(\mathtt{t_a}), task(\mathtt{t_b})) \in \mathtt{E}$, $\lambda_{ID} : A_{ID} \mapsto L$. *ID* satisfies the following constraints:

- $\langle \mathsf{a}_0, 1 \rangle \in \mathtt{A}_{ID}$;
- $\forall \langle \mathsf{a}, 1 \rangle \in \mathtt{A}_{ID}$ with $\mathtt{a} \in \mathtt{A}^{\wedge}_{\mathtt{in}}$, and $\forall (p, a) \in E$, there exists either
  (i) $(\langle \mathtt{p}, 1 \rangle, \langle \mathsf{a}, 1 \rangle) \in \mathtt{E}_{ID}$, if $p$ is not replicated or otherwise
  (ii) for each $\langle \mathtt{r}, tid \rangle \in \mathtt{A}_{ID}$, where $\mathtt{r} = start^*(\mathtt{p})$, there exists $(\langle \mathtt{p}, tid \rangle, \langle \mathsf{a}, 1 \rangle)$ in $E_{ID}$;
- $\forall \langle \mathsf{a}, 1 \rangle \in \mathtt{A}_{ID}$ with $\mathtt{a} \in \mathtt{A}^{\vee}_{\mathtt{in}}$, there exists at least an arc $(\langle \mathtt{p}, tid \rangle, \langle \mathsf{a}, 1 \rangle)$ in $E_{ID}$;
- $\forall \langle \mathsf{a}, tid \rangle \in \mathtt{A}_{ID}$ with $\mathtt{a} \in \mathtt{A}^{*}_{\mathtt{in}}$, there exists the arc $(\langle \mathtt{p}, 1 \rangle, \langle \mathsf{a}, tid \rangle)$ in $E_{ID}$, where $p$ is the non-replicated task preceding $a$;
- $\forall \langle \mathsf{a}, tid \rangle \in \mathtt{A}_{ID}$ with $\mathtt{a} \in \mathtt{A}^{*\mathtt{c}}_{\mathtt{in}}$, there exists the arc $(\langle \mathtt{p}, 1 \rangle, \langle \mathsf{a}, tid \rangle)$ in $E_{ID}$, where $p$ is the non-replicated task preceding $a$; moreover, if $tid > 1$, the arc $(\langle \mathtt{b}, tid - 1 \rangle, \langle \mathsf{a}, tid \rangle)$ is in $E_{ID}$, where $b$ is the replicated task preceding $a$;
- $\forall \langle \mathsf{a}, tid \rangle \in \mathtt{A}_{ID}$ with $\mathtt{a} \in \mathtt{A}^{*\wedge}_{\mathtt{in}}$, and $\forall (\mathtt{p}, \mathtt{a}) \in \mathtt{E}$, $\langle \mathsf{a}, tid \rangle \in \mathtt{A}_{ID}$;
- $\forall \langle \mathsf{a}, tid \rangle \in \mathtt{A}_{ID}$ with $\mathtt{a} \in \mathtt{A}^{*\vee}_{\mathtt{in}}$, there exists $(\langle \mathtt{p}, tid \rangle, \langle \mathsf{a}, tid \rangle) \in \mathtt{E}_{ID}$;
- $\forall \langle \mathsf{a}, tid \rangle \in \mathtt{A}_{ID}$ with $\mathtt{a} \in \mathtt{A}^{\wedge}_{\mathtt{out}}$, it holds $(\langle \mathsf{a}, tid \rangle, \langle \mathtt{b}, tid \rangle) \in \mathtt{E}_{ID}$, $\forall (\mathtt{a}, \mathtt{b}) \in \mathtt{E}$;
- $\forall \langle \mathsf{a}, tid \rangle \in \mathtt{A}_{ID}$ with $\mathtt{a} \in \mathtt{A}^{\vee}_{\mathtt{out}}$, there exists exactly one arc, say $(\langle \mathsf{a}, tid \rangle, \langle \mathtt{b}, tid \rangle) \in \mathtt{E}_{ID}$, outgoing from $\langle a, tid \rangle$;
- $\forall \langle \mathsf{a}, tid \rangle \in \mathtt{A}_{ID}$ with $\mathtt{a} \in \mathtt{A}^{\mathtt{L}}_{\mathtt{out}}$ and $\lambda_{ID}(\langle \mathsf{a}, tid \rangle) = \mathtt{l}$, $\forall (\mathtt{a}, \mathtt{b}) \in \mathtt{E}$, with label $l$ the arc $(\langle \mathsf{a}, tid \rangle, \langle \mathtt{b}, tid \rangle)$ is in $E_{ID}$, . □

**Example 5** Two instances for the schema in Figure 1 are shown in Figure 4, 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

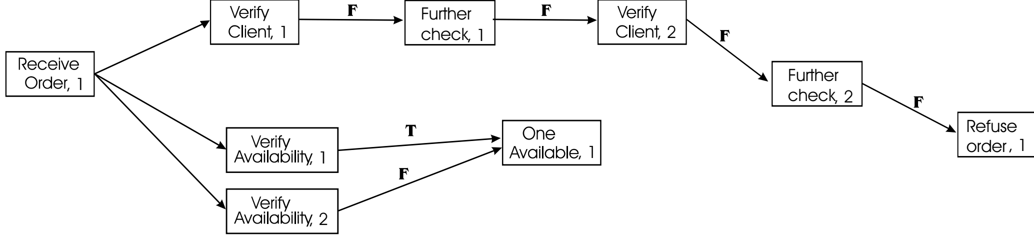


Fig. 5. Cyclic instanc for the Sales Ordering Process.

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. In both cases the client turned out to be reliable and both instances have replicated tasks. The case that the client is not considered reliable even after two checks is shown in Figure 5 — the instance is cyclic. □

Finally, we assume in our model that a number of servers are available to perform the various tasks of the workflow instances which are being executed in a given time period.

**Definition 6 (Servers)** Given a workflow schema $\mathcal{WS}$ over the set of tasks `A`, we denote by $\mathcal{S}_{ws}$ the set of *servers* (human and/or computers) who are appointed to execute various tasks. Then, the functions

- $\texttt{task} : \mathcal{S}_{ws} \mapsto 2^A$,
- $\texttt{duration} : \mathcal{S}_{ws} \times A \mapsto \mathbb{N}$, and
- $\texttt{stateServer} : \mathcal{S}_{ws} \mapsto \{available,\ busy,\ outOfOrder\}$

assign to each server the tasks that it may execute, the time required for the execution, and its current state, respectively. □

The current status of a server is determined by the fact that it is executing some tasks: Given an instantiation of the workflow, the state of a server is *busy* if it is executing a certain task, is *outOfOrder* if it is not allowed to execute any tasks, and is *available* if it is waiting for a task to execute. It is worth noting, that also the tasks are characterized by a particular state among the following: *idle*, thus the task is not yet activated as none of its incoming arcs are active; *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; *ready*, i.e., the task is ready for execution and has been started but it is waiting for the assignment of a server; *running*, i.e., the task is currently executed by a server; *executed*, i.e., the task has been terminated.

## 3 DATALOG$^{ev!}$ in a Nutshell

In this section we overview *Event Choice* DATALOG (short: DATALOG$^{ev!}$) [11,10], an extension of DATALOG that is able to deal with events and dynamic knowledge, and which is particulary suited for modelling and reasoning workflow evolutions.

Roughly speaking, DATALOG$^{ev!}$ is a language for modeling the evolution of knowledge states, triggered by events and guided by nondeterministic transition rules. Its main features are:

- **Event Activation Rules:** The language models the transitions among states of the world by exploiting the notion of event. The occurrence of an event enables the application of a rule that may modify the state by asserting or retracting some facts (*fluents*), and may trigger other events to occur in the future. The language also supports the interaction with external events. This latter feature is particularly useful for simulating and reasoning about possible scenarios, as we shall describe in our motivating examples.
- **Choice constructs:** The ability to deal with the nondeterminism has been recognized as a key feature of logic based languages. However, an undisciplined use of unstratified negation and/or disjunction leads to higher computational complexities and to hard-to-read programs. For this reason, DATALOG$^{ev!}$ programs are stratified, but their rules may contain *choice* atoms [20], that provide nondeterministic features. In particular, if we are not interested in a particular outcome (temporal evolution) of the program, the *choice* construct is able to model a *don't-care* form of nondeterminism.

Thus, DATALOG$^{ev!}$ combines the capability of the *choice* construct to express nondeterminism (mainly, *don't-care* nondeterminism), with the *event activation rules*, used for modeling events occurring at certain specified time instants.

### *3.1 Syntax of Event Event Activation Rules*

DATALOG$^{ev!}$ is based on DATALOG syntax, by enriching the predicates with an additional arguments that provides the multiple-time dimensions. In the paper, we shall deal with only two dimensions, i.e., we do not exploit all the modelling capabilities of the language. Hence, we assume each time instant to be a tuple $\langle t_1, t_2 \rangle$, where both $t_1$ and $t_2$ are natural numbers that can be thought of as the integer and the decimal part of the time, respectively.

External events arise at the integer time $\langle t_1, 0 \rangle$ and subsequent time instants at a finer scale are indicated by increasing the second component and are used for sequencing almost immediate internal events triggered by the system.

Whenever the second component instant does not matter, a time instant can be simply represented as $\mathtt{t_1}$.

The set $\mathcal{T}$ of all the pairs of natural numbers is the underlying *time domain* of any program, which is linearly ordered according to the usual lexicographic precedence relationship. Then, for any literal $\mathtt{p}$, and each time instant $\mathtt{t}$, $\mathtt{p@t}$ is true if $\mathtt{p}$ holds at time $\mathtt{t}$. Moreover, we define two *temporal functions* on time instants: given $\mathtt{t} = \langle \mathtt{t_1}, \mathtt{t_2} \rangle \in \mathtt{T}$, let $\mathtt{t{+}{+}} = \langle \mathtt{t_1}, \mathtt{t_2} + \mathtt{1} \rangle$, and let $\mathtt{t} + (\mathtt{D}) = \langle \mathtt{t_1} + \mathtt{D}, \mathtt{0} \rangle$ for any natural number $\mathtt{D}$.

We assume that three sets of constants, variables, and time variables symbols, $\sigma^{const}$, $\sigma^{vars}$, and $\sigma^{time_vars}$ are given, where the constants symbols are disjoint from the (time) variables symbols. A *term* $s$ is an element in $\sigma^{const} \cup \sigma^{vars}$. Moreover, let $\sigma^{\mathtt{EDB}}$, $\sigma^{\mathtt{DDB}}$, $\sigma^{\mathtt{IDB}}$, and $\sigma^{\mathtt{EV}}$ be disjoint sets of predicate symbols, with associated arity ($\geq 0$). Then, an EDB *atom* has a "classical" format $\mathtt{p}(\mathtt{s_1}, \ldots, \mathtt{s_n})$ where $p$ is a symbol in $\sigma^{\mathtt{EDB}}$ and $\mathtt{s_1}, \ldots, \mathtt{s_n}$ are terms. Instead DDB (dynamic extensional predicates), IDB (intensional predicates), and EV (event predicates) atoms are of the form $\mathtt{p}(\mathtt{s_1}, \ldots, \mathtt{s_n})@\mathtt{t}$, where $\mathtt{p}$ is a symbol in $\sigma^{\mathtt{DDB}}$, $\sigma^{\mathtt{IDB}}$, and $\sigma^{\mathtt{EV}}$, respectively, $\mathtt{n}$ is the arity of $\mathtt{p}$, $\mathtt{s_1}, ...\mathtt{s_n}$ are terms, and $\mathtt{t}$ is a time instant or a time variable in $\sigma^{time_vars}$. EV atoms can be also of the form $\mathtt{p}(\mathtt{s_1}, \ldots, \mathtt{s_n})@\mathtt{f}(\mathtt{t})$, where $\mathtt{f}$ is a temporal function over the domain $\mathcal{T}$.

An EDB,DDB, IDB, or EV *literal* is either an atom or its negation. The set of all the EDB literals (resp. DDB, IDB, EV), is denoted by $\mathcal{L}_{\mathtt{EDB}}$ (resp. $\mathcal{L}_{\mathtt{DDB}}$, $\mathcal{L}_{\mathtt{IDB}}$, $\mathcal{L}_{\mathtt{EV}}$). Furthermore, for any set of literals $\mathcal{L}$, $\mathcal{L}^+$ and $\mathcal{L}^-$ denote the sets of its positive and of its negative literals, respectively.

**Definition 7** A *dynamic* rule has the form $\mathtt{p}(\mathtt{X_1}, ..., \mathtt{X_n}) \leftarrow \mathtt{B_1}, ..., \mathtt{B_m}$. where $\mathtt{p}(\mathtt{X_1}, ..., \mathtt{X_n})@\mathtt{T} \in \mathcal{L}^+_{\mathtt{IDB}}$, $\mathtt{m} \geq \mathtt{0}$, and $\mathtt{B_1}, ..., \mathtt{B_m} \in \mathcal{L}_{\mathtt{EDB}} \cup \mathcal{L}_{\mathtt{DDB}} \cup \mathcal{L}_{\mathtt{IDB}}$.
An *event activation* rule has the format $[\mathtt{e}(\mathtt{X_1}, ..., \mathtt{X_n})@\mathtt{T}]\quad \mathtt{TR_1}\ \ ...\ \ \mathtt{TR_k}$, where $\mathtt{e}(\mathtt{X_1}, ..., \mathtt{X_m})@\mathtt{T} \in \mathcal{L}^+_{\mathtt{EV}}$, and $\mathtt{TR_1}, ..., \mathtt{TR_k}$ are *transition rules*. Each transition rule is of the form

$$!\mathtt{EV_1}@\mathtt{f_1}(\mathtt{T}), ..., !\mathtt{EV_n}@\mathtt{f_n}(\mathtt{T}), +\mathtt{A_1}, ..., +\mathtt{A_h}, -\mathtt{A_{h+1}}, ..., -\mathtt{A_\ell} \leftarrow \mathtt{B_1}, ..., \mathtt{B_m},\ \otimes\ \mathtt{C}.$$

where $\mathtt{n} + \ell > \mathtt{0}$, $\mathtt{EV_1}, ..., \mathtt{EV_n} \in \mathcal{L}^+_{\mathtt{EV}}$, $\mathtt{m} \geq \mathtt{0}$, $\mathtt{A_1}, ..., \mathtt{A_h}, \mathtt{A_{h+1}}, ..., \mathtt{A_\ell} \in \mathcal{L}^+_{\mathtt{DDB}}$, $\mathtt{B_1}, ..., \mathtt{B_m} \in \mathcal{L}_{\mathtt{EDB}} \cup \mathcal{L}_{\mathtt{DDB}} \cup \mathcal{L}_{\mathtt{IDB}}$, $\mathtt{f_1}, .., \mathtt{f_n}$ are temporal functions, and $\mathtt{C}$ is the optional choice atom *choiceAny*. □

Roughly speaking, the construct choiceAny() nondeterministically selects one consequence from the set of all consequences derivable from a rule $r$.

The informal semantics of an event activation rule is that, if the event $\mathtt{e}(\mathtt{X_1}, ..., \mathtt{X_n})$ occurs at time $\mathtt{t} \in \mathcal{T}$ and the body of the transition rule is evaluated true, then the facts $\mathtt{A_1}, ..., \mathtt{A_h}$ are asserted at time $\mathtt{t{+}{+}}$, the facts $\mathtt{A_{h+1}}, ..., \mathtt{A_\ell}$ are retracted at time $\mathtt{t{+}{+}}$ (thus their temporal argument is implicit and of-

ten omitted), and the events $\mathtt{EV}_1, ..., \mathtt{EV}_n$ are triggered to be executed at times $\mathtt{f}_1(\mathtt{T}), ..., \mathtt{f}_n(\mathtt{T})$. Finally, we shall also report a predicate in $\mathtt{B}_1, ..., \mathtt{B}_m$ without temporal argument, whenever it coincides with the time t in which the event is triggered.

### *3.2* DATALOG$^{ev!}$ *Programs*

A DATALOG$^{ev!}$ program $\mathcal{P} = \langle \mathtt{DB}, \mathtt{DKB}, \mathtt{EV} \rangle$ consists of (i) a set DB of both extensional and dynamic facts, called *database*, (ii) a set DKB of dynamic rules, called *dynamic knowledge base*, and (iii) a set EV of event activation rules.

DATALOG$^{ev!}$ programs can be naturally used for modelling ad-hoc workflow specifications, by exploiting the notion of event activation rule. Intuitively, the completion of each task is modelled by means of an event, which is triggered after the execution of its predecessors according to the workflow schema; moreover, an initial event occur for the starting of a new instance.

**Example 8** We next built a program $\mathcal{P}_s = \langle \mathtt{DB}_s, \mathtt{DKB}_s, \mathtt{EV}_s \rangle$ modelling our running example. The database $\mathtt{DB}_s$ contains the extensional predicates product(IDitem, Description) and store(IDstore, City), storing information about the stores, plus the dynamic predicate availability(IDstore, IDitem, QTY) storing the quantity of product available in each store, and selectedStore(IDStore, IDItem, Quantity) encoding the fact that a given quantity of an item is taken from a store in order to satisfy a request. Figure 6 shows the database of a given company, and some facts of the dynamic knowledge at the time instant $\langle 0, 0 \rangle$.

The event activation rules are as follows. When an order is received at time $\mathtt{t} = \langle \mathtt{t}_1, \mathtt{t}_2 \rangle$, the event ReceiveOrder(ID, IDClient, IDItem, Qty)@(t) is triggered, causing the enactment of a new workflow instance, with identifier ID, in which the order of the client IDClient requesting the quantity Qty of the item IDItem is processed.

At the instant $\langle \mathtt{t}_1, \mathtt{t}_2 + 1 \rangle$, the events VerifyClient(ID, IDClient, CheckNo), with CheckNo = 1, and VerifyAvailability(ID, IDStore, IDItem, Qty) are internally triggered, where the latter is invoked for each store in store(IDStore, City):

```
[ReceiveOrder(ID, IDClient, IDItem, Qty)@(T)]
    VerifyAvailability(ID, IDStore, IDItem, Qty)@(T++) ← store(IDStore, City).
    VerifyClient(ID, IDClient, 1)@(T++)
```

If there exists a store S with enough availability, the event OneAvailable(ID, S, I, Qty) is internally triggered and the dynamic predicate selectedStore(S, I, Qty) is asserted in order to keep trace of the quantity

availability@$\langle 0, 0\rangle$:

| IDStore | IDItem | QTY |
|---|---|---|
| $s_1$ | $i_1$ | 5 |
| $s_1$ | $i_2$ | 6 |
| $s_1$ | $i_3$ | 4 |
| $s_2$ | $i_1$ | 10 |
| $s_2$ | $i_2$ | 7 |

store:

| IDStore | City |
|---|---|
| $s_1$ | Rome |
| $s_2$ | Milan |

product:

| IDItem | Description |
|---|---|
| $i_1$ | $d_1$ |
| $i_2$ | $d_2$ |
| $i_3$ | $d_3$ |
| $i_4$ | $d_4$ |

Fig. 6. The status of $\mathtt{DB}_s$ at time $\langle 0, 0\rangle$, in the Sales Ordering Process.

of items taken from each store at different executions. Note that in both the above rules, we use the operator ++, and, hence, we increase the secondary auxiliary time component. In fact, the events `VerifyClient` and `VerifyAvailability` are internal and arise almost instantaneously.

Notice also that when more then one store has the desired availability one is nondeterministically chosen according to the `choiceAny` construct. Finally, in the case, no store is available, the event `NoneAvailable(ID)` is instead triggered:

```
[VerifyAvailability(ID, S, I, Qty)@(T)]

   OneAvailable(ID, S, I, Qty)@(T++),
   +selectedStore(S, I, Qty)          ← availability(S, I, AQty), AQty > Qty, ⊗ choiceAny().
   NoneAvailable(ID)@(T++)            ← ¬available(I, Qty).
```

where the rule $\mathtt{available(I, Qty)} \leftarrow \mathtt{availability(S, I, AQty)},\ \mathtt{AQty} > \mathtt{Qty}$. is added to the dynamic knowledge base $\mathtt{KDB_s}$.

The order is eventually refused by triggering `RefuseOrder(ID)` if either no store is available, or the client is not reliable – here we assumed the existence of predicates `reliable(IDClient, CheckNo)` and `maxCheckNo(CheckNo)` since we are not interested in the details of this aspect. Conversely, if the client is reliable we store the fact that he was verified (`verified(ID, IDClient)`), and we call the event `AcceptOrder(ID)`:

```
[NoneAvailable(ID)@(T)]

   RefuseOrder(ID)@(T++)

[OneAvailable(ID, IDStore, IDItem, Qty)@(T)]

   AcceptOrder(ID)@T(++)

[VerifyClient(ID, IDClient, CheckNo)@(T)]

   AcceptOrder(ID)@(T++), +verified(ID, IDClient) ← reliable(IDClient, CheckNo).
   FurtherCheck(ID, IDClient, CheckNo)@(T++)      ← ¬reliable(IDClient, CheckNo).

[FurtherCheck(ID, IDClient, CheckNo)@(T)]

   RefuseOrder(ID)@(T++)                          ← maxCheckNo(Nmax), CheckNo ≥ Nmax.
   VerifyClient(ID, IDClient, CheckNo1)@(T++)     ← maxCheckNo(Nmax),
                                                    CheckNo < Nmax, CheckNo1isCheckNo + 1.
```

The careful reader may have noticed that the event `AcceptOrder(ID)` is triggered both when we find a store with the requested availability and when the client is verified. Nonetheless, the actual acceptance may happen only when both events occurred. Thus, in the event activation rule we further verify that both the client is verified (`verified(ID, IDClient)`) and a store has been selected (`selectedStore(S, I, Qty)`). Then, the quantity of item is finally updated, by aserting `availability(S, I, AQty − Qty)` and retracting the old quantity `availability(S, I, AQty)`.

$[\mathtt{AcceptOrder}_{ID}()@(\mathtt{T})]$

$$\begin{array}{lll} -\mathtt{availability(S,I,AQty)}, & & \\ +\mathtt{availability(S,I,AQty-Qty)} & \leftarrow & \mathtt{verified(ID,IDClient)}, \\ & & \mathtt{selectedStore(S,I,Qty)}, \\ & & \mathtt{availability(S,I,AQty)}. \end{array}$$

□

We stress that the above example is just an application of $\mathtt{DATALOG}^{\mathtt{ev!}}$ for modelling a particular workflow, and it has been presented for making the reader familiar with its syntax. In Section 4, we shall generalize these ideas by presenting a technique for automatically encoding *any* workflow specification into a $\mathtt{DATALOG}^{\mathtt{ev!}}$ program.

### *3.3 Semantics*

The semantics of an $\mathtt{DATALOG}^{\mathtt{ev!}}$ program is given in terms of its temporal (stationary) models. As usual, we first introduce the notion of interpretation and then add the conditions allowing an interpretation to be a model.

Let $\mathcal{P} = \langle \mathtt{DB}, \mathtt{DKB}, \mathtt{EV} \rangle$ be a $\mathtt{DATALOG}^{\mathtt{ev!}}$ program. As usual, the *Herbrand Universe* $U_{\mathcal{P}}$ of a $\mathcal{P}$ is the set of all constants appearing in $\mathcal{P}$. A dynamic literal (resp., an event) in $\mathcal{L}_{\mathtt{DDB}} \cup \mathcal{L}_{\mathtt{IDB}}$ (resp. in $\mathcal{L}_{\mathtt{EV}}$) is ground if no variable occurs in it. The EDB (resp. DDB, IDB, EV) *Herbrand Base*, denoted by $\mathbf{B}_{\mathtt{EDB}}$ (resp. $\mathbf{B}_{\mathtt{DDB}}$, $\mathbf{B}_{\mathtt{IDB}}$, $\mathbf{B}_{\mathtt{EV}}$), is the set of all ground extensional (resp., dynamic fact, intensional, event) literals that can be constructed with the predicate symbols in $\sigma^{\mathtt{EDB}}$ (resp., $\sigma^{\mathtt{DDB}}$, $\sigma^{\mathtt{IDB}}$, $\sigma^{\mathtt{EV}}$), by replacing the variables in $\sigma^{vars}$ by constants in the Herbrand universe and the time variables in $\sigma^{time_vars}$ by time instants in T.

An *interpretation* for the program $\mathcal{P}$ consists of a pair $\langle S, E \rangle$, where $S$ is a set of ground literals and $E$ is a set of ground events, such that

$(i)$ $S \subseteq \mathbf{B}_{\mathtt{EDB}} \cup \mathbf{B}_{\mathtt{IDB}} \cup \mathbf{B}_{\mathtt{EV}} \cup \mathbf{B}_{\mathtt{DDB}}$ $\qquad$ $(ii)$ $E \subseteq \mathbf{B}^{+}_{\mathtt{EV}}$

The minimum temporal argument occurring in the events in $E$ is denoted by *nextTime*$(I)$, while the maximum temporal argument occurring in the predi-

cates in $S$ is denoted by $curTime(I)$. Finally, an interpretation $I$ is *feasible* if $E = \emptyset$ or $curTime(I) \prec nextTime(I)$.

Intuitively, a feasible interpretation $I$ determines a truth value for all the predicates preceding the time $curTime(I)$, and contains the information on the events that are currently triggered to occur in the future. In particular, a ground IDB or EDB predicate is true w.r.t. $I$ if it is an element of it; a dynamic ground fact $\mathtt{p@t}$ is true w.r.t. $I$ if there exists an element $\mathtt{p@t'}$ in $I$ such that $t' \preceq t$, and there is no literal $\neg\mathtt{p@t''} \in \mathtt{I}$ such that $t' \prec t'' \prec t$. Note that in the above definition, we assume that any DDB predicate asserted at a given time, remains valid till it is explicitly retracted from the database; indeed, the behavior of the DDB predicates is essentially *inertial*, while the truth value of the IDB predicates must be determined at each time instant. Finally, the special choice literals are defined to be always true w.r.t. to any possible interpretation $I$, regardless whether they occur or not in $I$.

**Example 9** In the program of Example 8, the pairs:

$I_1 = \langle\{\mathtt{selectedStore(s_2, i_3, 4)@5}\}, \{\mathtt{RefuseOrder(id_1)@8}\}\rangle$, and
$I_2 = \langle\{\mathtt{selectedStore(s_2, i_3, 4)@4}, \neg\mathtt{selectedStore(s_2, i_4, 6)@5}\},$
$\{\mathtt{RefuseOrder(id_1)@2}, \mathtt{AcceptOrder(id_2)@9}\}\rangle$

are both interpretations,but, the latter is not feasible since $nextTime(I_2) = 2$ and $curTime(I_2) = 8$. Moreover, note that in the former interpretation the predicate $\mathtt{selectedStore(s_2, i_3, 4)}$ is true in every time instant following 5, since it has been never retracted after its assertion at time 5. □

Given an interpretation $I = \langle S, E\rangle$, we denote by *triggered*$(E)$ the set of all events in $E$ having temporal argument $nextTime(I)$. Let $TR(I)$ be the subset of all transition rules such that all their activating events belong to *triggered*$(E)$, and $C(I)$ be the set of all choice predicates occurring in the rules in $TR(I)$. Moreover, let *ground_* $TR(I)$ be the set of all the ground instantiations $R$ of the rules in $TR(I)$ such that (i) all transition rules in $R$ are enabled, and (ii) the functional dependencies determined by the choice constructs in $C(I)$ are satisfied by $R$. Thus, *ground_* $TR(I)$ contains a set of enabled ground rules (coming from instantiations of the rules in $TR(I)$) for each possible way of enforcing the functional dependencies determined by the choices in $C(I)$.

Let *chosen_ tr* be any set of ground rules in *ground_* $TR(I)$. We denote by $\mathcal{A}_I(chosen_tr)$ the set of all the dynamic atoms $\mathtt{p}$ such that $+\mathtt{p}$ occurs in the head of some transition rule in *chosen_ tr* and $\mathtt{p}$ is false w.r.t. $I$. Such a dynamic atom $\mathtt{p}$ is said to be asserted. Similarly, $\mathcal{R}_I(chosen_tr)$ is the set of all the dynamic literals $\neg\mathtt{p}$ such that $-\mathtt{p}$ occurs in the head of some transition rule in *chosen_ tr* and $\mathtt{p}$ is true w.r.t. $I$. In this case, we say that $\mathtt{p}$ has been retracted. Finally, $\mathcal{E}_I(chosen_tr)$ is the set of the events triggered by all transition rules $r$ in *chosen_ tr* such that at least one dynamic atom is either asserted or

retracted because of $r$. In the sequel, the set of all the interpretations of a given program $\mathcal{P}$ is denoted by $\mathcal{I}_{\mathcal{P}}$, while the set of all the subsets of $\mathcal{I}_{\mathcal{P}}$ is denoted by $2^{\mathcal{I}_{\mathcal{P}}}$.

**Definition 10** Let $\mathcal{P} = \langle \mathtt{DB}, \mathtt{DKB}, \mathtt{EV} \rangle$ be an event choice Datalog program. Then, we define $\mathbf{T} : 2^{\mathcal{I}_{\mathcal{P}}} \mapsto 2^{\mathcal{I}_{\mathcal{P}}}$ to be the function that, given a set of interpretations $\mathcal{I}$, outputs a set of interpretations $\mathbf{T}(\mathcal{I})$ containing, for any $I = \langle S, E \rangle \in \mathcal{I}$ and any set of transition rules $chosen_tr \in ground_TR(I)$, all interpretations $\langle S', E' \rangle$ such that

$$S' \in \mathrm{SM}(\mathtt{DB} \cup \mathtt{DKB} \cup S \cup \mathcal{A}_I(chosen_tr) \cup \mathcal{R}_I(chosen_tr)) \cup triggered(E),$$

$$E' = E \cup \mathcal{E}_I(chosen_tr) - triggered(E).$$

□

Note that, for any given interpretation $I = \langle S, E \rangle$, this function computes the set of all feasible interpretations that can be obtained by triggering events and by asserting or retracting predicates, according to $I$. Note that any output interpretation $I' = \langle S', E' \rangle$ takes into account the consequences of the events triggered at the time $nextTime(I)$. All these events are removed from the set of envisioned events $E'$, while new events possibly planned to occur in the future are added to $E'$ through the set $\mathcal{E}_I(chosen_tr)$. The set $S'$ is any stable model of the dynamic knowledge base DKB evaluated over $\mathtt{DB} \cup S$ plus the asserted and retracted predicates, and including the recently occurred events.

We point out that, as a consequence of the non-deterministic choices constructs, the output of $\mathbf{T}$ applied on a singleton $\{I\}$ is in general a set of multiple alternative interpretations, even in the case the dynamic knowledge base is stratified. However, it deterministically outputs a unique interpretation (for the given $I$) if the program is stratified and there are no "active" choices, i.e., $C(I) = \emptyset$.

**Definition 11** Let $\mathcal{P} = \langle \mathtt{DB}, \mathtt{DKB}, \mathtt{EV} \rangle$ be a $\mathtt{DATALOG}^{\mathtt{ev!}}$ program, where EDB is the set of extensional predicates in DB, and H a list of ground events, also called *list of envisioned events*. The *evolution* of the program $\mathcal{P}$ given H (short: the evolution of $\mathcal{P}_{\mathtt{H}}$) is the succession of sets of interpretations $\widehat{\mathbf{T}}$ such that (i) $\widehat{\mathbf{T}}_{\mathbf{0}} = \{\langle \mathtt{EDB}, \mathtt{H} \rangle\}$, and (ii) $\widehat{\mathbf{T}}_{\mathbf{i+1}} = \mathbf{T}(\widehat{\mathbf{T}}_{\mathbf{i}})$. For every $j > 0$, any interpretation $M \in \widehat{\mathbf{T}}_j$ is called a *temporal model* for $\mathcal{P}_{\mathtt{H}}$. □

Note that the definition of temporal model refers to a list H of envisioned events, containing the events that are deterministically known to happen. Thus, H can be used for simulating the actual behavior of a system modelled with $\mathtt{DATALOG}^{\mathtt{ev!}}$. For instance, in Example 8, two orders $id_1$ and $id_2$ that arrive at time instants 0 and 3, respectively, can be encoded through the list of envisioned events $\mathtt{H} = [\mathtt{ReceiveOrder}(\mathtt{id}_1, \mathtt{c}_1, \mathtt{i}_1, 5)@0, \mathtt{receiveOrder}(\mathtt{id}_2, \mathtt{c}_2, \mathtt{i}_1, 10)@3]$.

Under an abstract perspective, the events in H are used for constraining the evolution of the $\texttt{DATALOG}^{\texttt{ev!}}$ program.

**Definition 12** Let $\mathcal{P}$ be a $\texttt{DATALOG}^{\texttt{ev!}}$ program, EDB be an input database, and H a list of ground events. A temporal model $M$ for $\mathcal{P}_{\texttt{H}}$ is a *stationary model* (for $\mathcal{P}_{\texttt{H}}$) if it is a fixpoint of $\mathbf{T}$, i.e., if $M \in \mathbf{T}(\{M\})$. Moreover, $curTime(M)$ is called the *converging time* of $M$. □

Finally, the set of all the temporal (resp. stationary) models of a given program $\mathcal{P}_{\texttt{H}}$ is denoted by $\mathcal{TM}(\mathcal{P}_{\texttt{H}})$ (resp. $\mathcal{TSM}(\mathcal{P}_{\texttt{H}})$).

## 4 Describing Workflow Evolutions in $\texttt{DATALOG}^{\texttt{ev!}}$

In this section, we provide an automatic mechanism for deriving logic specifications, which can be eventually used for simulating the behavior of the system with different execution scenarios. Specifically, let $\mathcal{WS}$ be a workflow schema, then we want to construct a $\texttt{DATALOG}^{\texttt{ev!}}$ program $\mathcal{P}(\mathcal{WS}) = \langle \texttt{DB}_{ws}, \texttt{KDB}_{ws}, \texttt{EV}_{ws} \rangle$ modelling $\mathcal{WS}$, by specifying both static and dynamic aspects, as well as the general event transition rules for the its enactment.

### *4.1 Database and Dynamic Knowledge Bases*

Given $\mathcal{WS} = \langle \texttt{A}, \texttt{E}, \texttt{a}_0, \texttt{F}, \texttt{A}^{\wedge}_{\texttt{in}}, \texttt{A}^{\vee}_{\texttt{in}}, \texttt{A}^{*}_{\texttt{in}}, \texttt{A}^{*\texttt{c}}_{\texttt{in}}, \texttt{A}^{*\wedge}_{\texttt{in}}, \texttt{A}^{*\vee}_{\texttt{in}}, \texttt{A}^{\wedge}_{\texttt{out}}, \texttt{A}^{\vee}_{\texttt{out}}, \texttt{A}^{\texttt{L}}_{\texttt{out}}, \texttt{E}^{\texttt{L}}, \lambda, \texttt{L} \rangle$, the database $\texttt{DB}_{ws}$ is such that

- Each node $a$ in $A$, the unique initial task $a_0$ and each final task $b$ in $F$ are defined by the predicates `task(a)`, `startTask(a`$_0$`)`, `finalTask(b)`, respectively;
- Each regular task $a$ is defined by `regularTask(a)`, whereas each replicated one is defined by `replicatedTask(a)`;
- The arcs in $E$ are defined by the predicates `arc(PrecTask, NextTask)`, while $\lambda : E^L \to L$ is defined by `arcLabel(PrecTask, NextTask, Label)`;
- Each task `a` in $\texttt{A}^{\wedge}_{\texttt{in}}$, $\texttt{A}^{\vee}_{\texttt{in}}$, $\texttt{A}^{*}_{\texttt{in}}$, $\texttt{A}^{*\texttt{c}}_{\texttt{in}}$, $\texttt{A}^{*\wedge}_{\texttt{in}}$, $\texttt{A}^{*\vee}_{\texttt{in}}$, $\texttt{A}^{\wedge}_{\texttt{out}}$, $\texttt{A}^{\vee}_{\texttt{out}}$, and $\texttt{A}^{\texttt{L}}_{\texttt{out}}$ is defined by `inAND(a)`, `inOR(a)`, `inRep(a)`, `inRepCYCLE(a)`, `inRepAND(a)`, `inRepOR(a)`, `outAND(a)`, `outOR(a)`, and `outLabel(a)` respectively.

**Example 13** In our running example, $\texttt{DB}_s$ contains 8 atoms defining the tasks, `startTask(ReceiveOrder)`, `finalTask(RefuseOrder)`, and `finalTask(AcceptOrder)`. The predicate `inAND` contains exactly the facts (`AcceptOrder`) and (`NoneAvailable`); `inOR` consists of the facts (`RefuseOrder`), and (`OneAvailable`).

We have the fact (VerifyAvailability) in inRep, the fact (VerifyClient) in inRepCYCLE and the fact (FurtherCheck) in inRepOR. The facts (ReceiveOrder), (OneAvailable) and (NoneAvailable) are in outAND. Finally the facts (VerifyClient), (FurtherCheck) and (VerifyAvailability) are in outLabel.

Concerning the arcs, we have that the predicate arcLabel consists of (VerifyAvailability, OneAvailable, T), (VerifyClient, AcceptOrder, T), (VerifyAvailability, NoneAvailable, F), (FurtherCheck, RefuseOrder, F), (FurtherCheck, VerifyClient, T), (VerifyClient, FurtherCheck, F). □

$\mathtt{DB}_{ws}$ contains also the information needed for the execution. In the following, we associate to each workflow instance a unique identifer *ID*, and, in order to simplify the presentation, a predicate $\mathtt{p}(ID, \mathtt{X})$, where X is a generic list of arguments, is denoted by $\mathtt{p}_{ID}(\mathtt{X})$. Then, given a workflow instance *ID*, the state of the execution of a Task with identifier TID is kept by means of the following relations:

- $\mathtt{startActive}_{ID}(\mathtt{Task}, \mathtt{TID}, \mathtt{Time})$, storing the time when $\langle\mathtt{Task}, \mathtt{TID}\rangle$ was activated;
- $\mathtt{startReady}_{ID}(\mathtt{Task}, \mathtt{TID}, \mathtt{Time})$, storing the time when the task $\langle\mathtt{Task}, \mathtt{TID}\rangle$ was declared ready for execution;
- $\mathtt{startRunning}_{ID}(\mathtt{Task}, \mathtt{TID}, \mathtt{S}, \mathtt{Time})$, storing the time a server S has started its execution;
- $\mathtt{executed}_{ID}(\mathtt{Task}, \mathtt{TID}, \mathtt{Time}, \mathtt{Output})$, storing the time when the execution of the task $\langle\mathtt{Task}, \mathtt{TID}\rangle$ is completed and the result Output of the execution — recall that Output is a label in L.

The state of a task $\langle\mathtt{Task}, \mathtt{TID}\rangle$ can be derived using simple DATALOG rules and will be accessed with the predicate $\mathtt{state}_{ID}(\mathtt{Task}, \mathtt{TID}, stateType)$. These rules are in the dynamic knowledge base $\mathtt{KDB}_{ws}$:

$$\begin{array}{lll}
\mathtt{state}_{ID}(\mathtt{Task}, \mathtt{TID}, idle) & \leftarrow & \neg\mathtt{startActive}_{ID}(\mathtt{Task}, \mathtt{TID}, _). \\
\mathtt{state}_{ID}(\mathtt{Task}, \mathtt{TID}, activated) & \leftarrow & \mathtt{startActive}_{ID}(\mathtt{Task}, \mathtt{TID}, _),\ \neg\,\mathtt{startReady}_{ID}(\mathtt{Task}, \mathtt{TID}, _). \\
\mathtt{state}_{ID}(\mathtt{Task}, \mathtt{TID}, ready) & \leftarrow & \mathtt{startReady}_{ID}(\mathtt{Task}, \mathtt{TID}, _),\ \neg\,\mathtt{startRunning}_{ID}(\mathtt{Task}, \mathtt{TID}, _, _). \\
\mathtt{state}_{ID}(\mathtt{Task}, \mathtt{TID}, running) & \leftarrow & \mathtt{startRunning}_{ID}(\mathtt{Task}, \mathtt{TID}, _, _),\ \neg\,\mathtt{executed}_{ID}(\mathtt{Task}, \mathtt{TID}, _, _). \\
\mathtt{state}_{ID}(\mathtt{Task}, \mathtt{TID}, executed) & \leftarrow & \mathtt{executed}_{ID}(\mathtt{Task}, \mathtt{TID}, _, _).
\end{array}$$

where, to simplify the notation, we used some syntactic sugar for writing negative literals in the body of the first of the above rules: $\neg\mathtt{a}(\mathtt{X})$, stands for $\neg\mathtt{a}'(\mathtt{Y})$, where $\mathtt{a}'$ is defined by the new rule: $\mathtt{a}'(\mathtt{Y}) \leftarrow \mathtt{a}(\mathtt{X})$, and Y is the list of all non-anonymous variables occurring in X.

Servers are stored by the predicate server(ServerName), while 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 dynamic rule in $\texttt{KDB}_{ws}$:

$$\begin{array}{l}\texttt{available(Server)} \leftarrow \texttt{executable(Server,_,_)}, \neg\,\texttt{outOfOrder(Server)}, \\ \qquad\qquad \neg\,(\,\texttt{startRunning}_{ID}(\texttt{Task},\texttt{TaskIdentifier},\texttt{Server},_), \\ \qquad\qquad\ \neg\,\texttt{executed}_{ID}(\texttt{Task},\texttt{Quantifiers},_,_)\,).\end{array}$$

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

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

### *4.2 Modelling the Enactment Engine*

he core of the program $\mathcal{P}(\mathcal{WS})$ lies in the definition of the event activation rules that guide the evolution of the workflow instances, and that enable to realize a simulation environment for workflow executions, which is quite intuitive, declarative in the spirit, yet so powerful to cover all the features of current workflow systems. The whole set of rules in $\texttt{EV}_{ws}$ is shown in Figure 7, where bold predicates are used for denoting predicates that can be customized, i.e., specialized for the need of any particular workflow designer.

The first event, called init, is an external event which starts a new workflow instance at a certain time $\langle \texttt{t}_1, \texttt{t}_2\rangle$. The dynamic predicate $\texttt{started}_{ID}$ is used for keeping trace of the fact that a new instance *ID* has been started. Notice that these event causes also the triggering of $\texttt{run}()@(\texttt{T}{+}{+})$, i.e., at a time instant $\langle \texttt{t}_1, \texttt{t}_2 + 1\rangle$. Thus, it is an event which is seen instantaneously with the init in the main (real) dimension, whereas it happens in the auxiliary internal dimension. Every time the event $\texttt{run}()@(\texttt{T})$ is internally triggered in rule $r_2$, 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 $\texttt{executed}_{ID}()$ is true if the workflow instance has already entered a final state so that no other task needs to be performed.

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 in rule $r_3$, where the predicate

```
r1 : [init(WID)@(T)]
       !run()@(T++),
       +started_ID(), +startReady_ID(ST, 1, T)          ← startTask(ST).

r2 : [run()@(T)]
       !evaluate_ID(Task, L, Duration)@(T++)
       +startRunning_ID(Task, TID, Server, T)           ← ¬ unsat_ID(), ¬ executed_ID(),
                                                          state_ID(Task, TID, ready),
                                                          available(Server),
                                                          executable(Server, Task, Duration)
                                                          ⊗ choiceAny().

r3 : [evaluate_ID(Task, TID, Duration)@(T)]
       !complete_ID(Task, TID, Output)@(T+(Duration))  ← evaluation_ID(Task, TID, Output).

r4 : [complete_ID(Task, TID, Output)@(T)]
       !run()@(T++), executed_ID(Task, TID, T, Output).
       +unsat_ID()                                      ← unsatGC_ID(Task, TID).
       +executed_ID()                                   ← finalTask(Task), ¬ unsatGC_ID(Task, TID).
       !activateTask_ID(Next, TID, Task)@(T++),         ← outOR(Task), Output ≠ "fail", arc(Task, Next)
                                                          ⊗ ChoiceAny().
       !activateTask_ID(Next, TID, Task)@(T++)          ← outAND(Task), Output ≠ "fail", arc(Task, Next).
       !activateTask_ID(Next, TID, Task)@(T++)          ← outLabel(Task), Output ≠ "fail",
                                                          arcLabel(Task, Next, Label), Label = Output.

r5 : [activateTask_ID(Task, TID, Prec)@(T)]
       !run()@(T++),
       +startActive_ID(Task, 1, T),
       +startReady_ID(Task, 1, T)                       ← inOR(Task), ¬ state_ID(Task, _, active).
       !run()@(T++),
       +activeArc_ID(Prec, TID, Task, NewTID),
       +startActive_ID(Task, NewTID, T),
       +startReady_ID(Task, NewTID, T),                 ← inRep(Task), quantifyTID(Task, NewTID).
       !run()@(T++),
       +startActive_ID(Task, NewTID, T),
       +startReady_ID(Task, NewTID, T)                  ← inRepCYCLE(Task), NewTID = TID + 1,
                                                          ¬ (start * (Task′, Task),
                                                          startActive_ID(Task′, TID, _),
                                                          ¬ executed_ID(Task′, TID, _)).
       !run()@(T++),
       +startActive_ID(Task, TID, T),
       +startReady_ID(Task, TID, T)                     ← inRepOR(Task), ¬ state_ID(Task, TID, active).
       +activeArc_ID(Prec, TID, Task, TID)              ← inRepOR(Task).
       !checkForReady_ID(Task, TID)@(T++),
       +activeArc_ID(Prec, TID, Task, TID)              ← inRepAND(Task).
       +startActive_ID(Task, TID, T)                    ← inRepAND(Task),
                                                          ¬ state_ID(Task, TID, active).
       !checkForReady_ID(Task, 1)@(T++),
       +activeArc_ID(Prec, TID, Task, 1)                ← inAND(Task).
       +startActive_ID(Task, 1, T)                      ← inAND(Task), ¬ state_ID(Task, 1, active).

r6 : [checkForReady_ID(Task, TID)@(T)]
       !run()@(T++).
       +startReady_ID(Task, TID, T)                     ← inRepAND(Task), ¬state_ID(Task, TID, ready),
                                                          ¬ (arc(Prec, Task),
                                                          ¬ activeArc_ID(Prec, TID, Task, TID)).
       +startReady_ID(Task, 1, T)                       ← inAND(Task), ¬state_ID(Task, 1, ready),
                                                          ¬ (arc(Prec, Task), possInstance(Prec, TIDP),
                                                          ¬ activeArc_ID(Prec, TIDP, Task, 1)).
```

Fig. 7. $\texttt{DATALOG}^{ev!}$ *program* $\mathcal{P}(\mathcal{WS})$*, modelling the behavior of a workflow system.*

$\mathbf{evaluation}_{ID}(\mathtt{Task}, \mathtt{TID}, \mathtt{Output})$ 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 $\mathtt{T} + \mathtt{Duration}$.

**Example 14** In our running example, the task *VerifyAvailability* must check whether a given store contains enough quantity of the required item. This behavior is captured by the following rules:

$$
\begin{array}{lcl}
\mathbf{evaluation}_{ID}(\mathtt{VerifyAvailability}, \mathtt{TID}, \text{“T”}) & \leftarrow & \mathtt{store}(\mathtt{TID}, \mathtt{City}),\ \mathtt{order}_{ID}(\mathtt{Item}, \mathtt{OQTY}), \\
 & & \mathtt{availability}(\mathtt{TID}, \mathtt{Item}, \mathtt{AQTY}),\ \mathtt{AQTY} \geq \mathtt{OQTY}. \\
\mathbf{evaluation}_{ID}(\mathtt{VerifyAvailability}, \mathtt{TID}, \text{“F”}) & \leftarrow & \mathtt{store}(\mathtt{TID}, \mathtt{City}),\ \mathtt{order}_{ID}(\mathtt{Item}, \mathtt{OQTY}), \\
 & & \mathtt{availability}(\mathtt{TID}, \mathtt{Item}, \mathtt{AQTY}),\ \mathtt{AQTY} < \mathtt{OQTY}.
\end{array}
$$

Note that, besides `availability` and `store`, we are also assuming the existence of the dynamic predicate $\mathtt{order}_{ID}(\mathtt{Item}, \mathtt{OQTY})$ storing information about client request. □

As described in the event $r_4$, after the completion of a task, the selection of which of its successor tasks to be activated depends on whether the task is in $\mathtt{A}^{\vee}_{\mathtt{out}}$, $\mathtt{A}^{\wedge}_{\mathtt{out}}$, or $\mathtt{A}^{\mathtt{L}}_{\mathtt{out}}$ 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 $\mathtt{unsat}_{ID}()$ is added only if the predicate $\mathbf{unsatGC}_{ID}(\mathtt{Task}, \mathtt{TID})$ 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.

The event `activateTask` in rule $r_5$ is used for activating the target task in an arc. If the task is in $\mathtt{A}^{\vee}_{\mathtt{in}} \cup \mathtt{A}^{*}_{\mathtt{in}} \cup \mathtt{A}^{*\vee}_{\mathtt{in}} \cup \mathtt{A}^{*\mathtt{c}}_{\mathtt{in}}$ the activation also implies that the task is ready for the execution. For tasks in $\mathtt{A}^{*}_{\mathtt{in}}$, we activate as many instances as specified in the predicate `quantifyTID`, whereas for tasks in $\mathtt{A}^{*\mathtt{c}}_{\mathtt{in}}$ we activate a new instance at time, but only after having checked that all replicated tasks created in the previous iteration have been executed. The predicate *start∗* is defined next:

$$
\begin{array}{lcl}
\mathtt{start*}\,(\mathtt{Task}, \mathtt{TaskR}) & \leftarrow & \mathtt{arc}(\mathtt{Prec}, \mathtt{Task}), \\
 & & \mathtt{replicated}(\mathtt{Prec}), \mathtt{start*}\,(\mathtt{Prec}, \mathtt{TaskR}). \\
\mathtt{start*}\,(\mathtt{Task}, \mathtt{Task}) & \leftarrow & \mathtt{inRep}(\mathtt{Task}). \\
\mathtt{start*}\,(\mathtt{Task}, \mathtt{Task}) & \leftarrow & \mathtt{inRepCYCLE}(\mathtt{Task}).
\end{array}
$$

Notice that we also keep track of all activated arcs through the predicate `startActive`. As for tasks in $\mathtt{A}^{\wedge}_{\mathtt{in}}$ and $\mathtt{A}^{*\wedge}_{\mathtt{in}}$ we have to check more elaborated conditions by means of the event `checkForReady`, defined by rule $r_6$, which decides whether a given activated task is ready for execution, by implementing the semantics informally described in Section 2. Finally, the predicate

possInstance is defined next:

```
possInstance(Task, 1)      ← regular(Task).
possInstance(Task, TIDR) ← replicated(Task), start∗ (Task, TaskR), inRep(TaskR),
                             startReady(TaskR, TIDR, _).
```

### 4.3 Global Constraints

We complete the model by showing how to specify global constraints on the executions. Actually, the workflow designer may define the predicate $\mathbf{unsatGC}_{ID}(\mathtt{Task}, \mathtt{TID})$, used within the event `complete`, in order to account for any desired behavior. Obviously, these constraints are all those expressible by means of DATALOG rules; nonetheless, some common types of constraints are next discussed.

**Definition 15** Given a workflow $\mathcal{WS}$, a *scheduling constraint* over $\mathcal{WS}$ is defined as follows: (i) for each task $a \in A$, $!a$ (resp. $\neg !a$) is a *positive*(resp., *negative*) *primitive* scheduling constraint, (ii) given two positive primitive scheduling constraints $c_1$ and $c_2$, $c_1 \prec c_2$ is a *serial* constraint, whereas given any pair of scheduling constraints $c_1$ and $c_2$, $c_1 \vee c_2$ and $c_1 \wedge c_2$ are *complex* constraints. □

Informally, a *positive* (resp., *negative*) *primitive* constraint specifies that a task must (resp., must not) be performed in any workflow instance. A *serial* 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. The set of global constraints over a workflow $\mathcal{WS}$, denoted by $\mathbf{Constr}(\mathcal{WS})$, can be also mapped into the program $\mathcal{P}(\mathcal{WS}) = \langle \mathtt{DB}_{ws}, \mathtt{KDB}_{ws}, \mathtt{EDB}\rangle$, and specifically in a suitable set a set of dynamic rules in $\mathtt{KDB}_{ws}$, denoted by $\mathcal{P}_{\mathbf{Constr}}(\mathcal{WS})$, constructed as follows.

- For each global constraint $c =!\langle \mathtt{Task}, \mathtt{TID}\rangle$, $\mathcal{P}_{\mathbf{Constr}}(\mathcal{WS})$ contains

  $\mathtt{unsatGC1}_{ID}(\mathtt{c}, \mathtt{gs}) \leftarrow \mathtt{executed}_{ID}(\mathtt{Task}, \mathtt{TID}, \mathtt{T}, \text{“}\mathtt{fail}\text{”}).$
  $\mathtt{unsatGC1}_{ID}(\mathtt{c}, \mathtt{gs}) \leftarrow \neg\, \mathtt{executed}_{ID}(\mathtt{Task}, \mathtt{TID}, \mathtt{T}, _).$

  where `gs` equals `s` if $c$ only occurs as sub-expression of a complex global constraint; otherwise, `gs` holds `g`.
- For each global constraint $c = \neg\, !\langle \mathtt{Task}, \mathtt{TID}\rangle$, $\mathcal{P}_{\mathbf{Constr}}(\mathcal{WS})$ contains

  $\mathtt{unsatGC1}_{ID}(\mathtt{c}, \mathtt{gs}) \leftarrow \mathtt{executed}_{ID}(\mathtt{Task}, \mathtt{TID}, \mathtt{T}, \text{“}\mathtt{fail}\text{”}).$
- For a global constraint $c =!\langle \mathtt{Task}_1, \mathtt{TID}_1\rangle \prec !\langle \mathtt{Task}_2, \mathtt{TID}_2\rangle$, $\mathcal{P}_{\mathbf{Constr}}(\mathcal{WS})$ contains

  $\mathtt{unsatGC1}_{ID}(\mathtt{c}, \mathtt{gs}) \leftarrow \mathtt{executed}_{ID}(\mathtt{Task}_2, \mathtt{TID}_2, _, \text{“}\mathtt{fail}\text{”})).$
  $\mathtt{unsatGC1}_{ID}(\mathtt{c}, \mathtt{gs}) \leftarrow \mathtt{executed}_{ID}(\mathtt{Task}_2, \mathtt{TID}_2, \mathtt{T2}, \mathtt{O2}), \mathtt{O2} \neq \text{“}\mathtt{fail}\text{”}$
  $\quad \mathtt{executed}_{ID}(\mathtt{Task}_1, \mathtt{TID}_1, \mathtt{T1}, \mathtt{O1}), \mathtt{O1} \neq \text{“}\mathtt{fail}\text{”}, \mathtt{T2} < \mathtt{T1}.$
- For each global constraint $c:\ c_1 \vee c_2$, $\mathcal{P}_{\mathbf{Constr}}(\mathcal{WS})$ contains

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

- For each global constraint $c: \ c_1 \wedge c_2$, $\mathcal{P}_{\textbf{Constr}}(\mathcal{WS})$ contains

```
unsatGC1_ID(c), gs ← unsatGC1_ID(c1, _)
unsatGC1_ID(c), gs ← unsatGC1_ID(c2, _).
```

- Finally, $\mathcal{P}_{\textbf{Constr}}(\mathcal{WS})$ contains

```
unsatGC_ID(Task, L) ← finalTask_ID(Task), unsatGC1_ID(_, g).
```

Notice, that the last rule we added enforces the check for scheduling constraints to be done after the completion of a final task. 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 $a_2$. An interesting optimization issue is to find out which global constraints could be effectively tested after the completion of each task.

**Unsuccessful Executions:** 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:

```
failed_ID()  ← unsat_ID().
failed_ID()  ← started_ID(), ¬executed_ID(), ¬working_ID().
working_ID() ← state_ID(T, _, ready)).
working_ID() ← state_ID(T, _, running)).
```

These predicates are definitively useful while querying the workflow in order to reconstruct the actual status of an execution.

## 5 Querying for an Evolution

After having introduced the model $\mathcal{P}(\mathcal{WS})$ for specifying structural and dynamic aspects of a workflow $\mathcal{WS}$, 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 example, 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 requests (with the corresponding arrival time), and it is denoted with (S). For instance, the scenario $[\texttt{init}(\texttt{id}_1)@0, \texttt{init}(\texttt{id}_2)@2, \texttt{init}(\texttt{id}_3)@4, \texttt{init}(\texttt{id}_4)@5]$ specifies a new instantiation of the workflow at time instants 0, 2, 4 and 5. This scenario is used for querying the $\mathcal{P}(\mathcal{WS})$ $\texttt{DATALOG}^{\texttt{ev!}}$ program.

Roughly, a query is a triple $\langle \mathtt{S}, \mathtt{G}, \mathtt{R} \rangle$, where $\mathtt{S}$ is a scenario, which lists all the events envisioned to happen, $\mathtt{G}$ is the goal we want to achieve (e.g., the scheduling of all the orders), and $\mathtt{R}$ is a list of results. Goals are formulae involving literals and special temporal quantifiers, as defined inductively below. Let $\mathtt{t}$ be a time instant and $\mathtt{C}$ a nonempty conjunction of ground literals. Then, $\exists^{@\mathtt{t}}\mathtt{C}$ is a goal. Moreover, let $\mathtt{t_1}$ and $\mathtt{t_2}$ be two time instants such that $\mathtt{t_1} < \mathtt{t_2}$, let $\mathtt{C}$ be a (possibly empty) conjunction of ground literals, and let $\mathtt{Q}$ be a goal, whose first quantifier is $\exists^{@\mathtt{t_2}}$. Then, $\exists^{@\mathtt{t_1}}(\mathtt{C} \wedge \mathtt{Q})$ is a goal.

Hereafter, given a model $M = \langle S, E \rangle$ for $\mathcal{P}(\mathcal{WS})$, and a time instant $\mathtt{t}$, we denote by $M@\mathtt{t} = \langle S', E \rangle$ the interpretation consisting of all the atoms having any temporal argument $\mathtt{t}' \leq \mathtt{t}$. Given two temporal models $M$ and $N$, we say that $N$ is an evolution of $M$ from time $\mathtt{t}$ if $M@\mathtt{t} = N@\mathtt{t}$. The set of all the evolutions of $M$ from time $\mathtt{t}$ is denoted by $\mathit{evols}^{@\mathtt{t}}(M)$.

The semantics of goals is as follows.

**Definition 16** Let $\mathcal{WS}$ be a workflow, and let $\mathcal{M}$ be a set of temporal models for $\mathcal{P}(\mathcal{WS})$. We say that *a goal* $\mathtt{G}$ *is true with respect to* $\mathcal{M}$ if one of the following conditions holds:

- $\mathtt{G} = \exists^{@\mathtt{t}}\mathtt{C}$, where $\mathtt{C}$ is a nonempty conjunction of ground literals, and there exists $M \in \mathcal{M}$ s.t. all the literals in $\mathtt{C}$ are in $M$ at time $\mathtt{t}$; or
- $\mathtt{G} = \exists^{@\mathtt{t}}(\mathtt{C} \wedge \mathtt{Q}')$, where $\mathtt{C}$ is a (possibly empty) conjunction of ground literals, and there exists $M \in \mathcal{M}$ s.t. all the literals in $\mathtt{C}$ are in $M$ at time $\mathtt{t}$ and $\mathtt{Q}'$ is true w.r.t. $\mathit{evols}^{@\mathtt{t}}(M) \cap \mathcal{M}$.

Otherwise, we say that $\mathtt{G}$ is false w.r.t. $\mathcal{M}$. □

We are now in the position of formalizing the notion of querying a workflow.

**Definition 17** Let $\mathcal{WS}$ be a workflow. A query on $\mathcal{WS}$ is an expression of the form $\langle \mathtt{S}, \mathtt{G}, \mathtt{R} \rangle$ where

- $\mathtt{S}$ is a scenario;
- $\mathtt{G}$ (*goal*) is a goal;
- $\mathtt{R}$ (*result*) is a list $[\mathtt{r_1}(\mathtt{X_1})@(\mathtt{t_1}), \ldots, \mathtt{r_m}(\mathtt{X_m})@(\mathtt{t_m})]$, $m > 0$ and $\mathtt{t_1} \leq \ldots \leq \mathtt{t_m}$, where $\mathtt{r_i}$ is any predicate symbol of the program $\mathcal{P}(\mathcal{WS})$, say with arity $\mathtt{k_i}$, and $\mathtt{X_m}$ is a list of $\mathtt{k_i}$ terms. □

The semantics of a query is as follows.

**Definition 18 (Query answers)** Let $\mathcal{WS}$ be a workflow, and $\mathtt{Q} = \langle \mathtt{S}, \mathtt{G}, \mathtt{R} \rangle$, where $\mathtt{R} = [\mathtt{r_1}(\mathtt{X_1})@(\mathtt{t_1}), \ldots, \mathtt{r_m}(\mathtt{X_m})@(\mathtt{t_m})]$, be a query on it. The *answer of* $\mathtt{Q}$ (resp. *stationary answer of* $\mathtt{Q}$), denoted by $\mathtt{Q}(\mathcal{WS})$ (resp. $\mathtt{Q}^{\mathtt{s}}(\mathcal{WS})$), is either

- the list of relations $[\mathbf{r_1}, \ldots, \mathbf{r_m}]$ such that $\mathbf{r_i} = \{\mathtt{x_i} | \mathtt{r_i}(\mathtt{x_i})@\mathtt{t_i} \in \mathtt{M} \wedge \mathtt{x_i} \text{ unifies with } \mathtt{X_i}\}$, where $\mathtt{M}$ is a model with $\mathit{curTime}(M) \geq \mathtt{t_m}$ in the set of temporal models $\mathcal{TM}(\mathcal{P}(\mathcal{WS})_{\mathtt{S}})$ (resp., of stationary models

$\mathcal{TSM}(\mathcal{P}(\mathcal{WS})_{\mathtt{S}})$, such that $\mathtt{G}$ is true in $\{\mathtt{M}\}$ or
- the empty list if $\mathtt{G}$ is false in $\mathcal{TM}(\mathcal{P}(\mathcal{WS})_{\mathtt{S}})$ (resp. $\mathcal{TSM}(\mathcal{P}(\mathcal{WS})_{\mathtt{S}})$).

In the former case the query is true, whereas in the latter is false. □

**Example 19** Assume, in our running example, the company has planned to have a number of requests, constituting a scenario $\mathtt{S}$. The designer want to know the possible evolutions; this aim can be achieved by supplying the query $\langle \mathtt{S}, \emptyset, \mathtt{R} \rangle$. Indeed, the list $\mathtt{R}$, in the case is not empty, stores the log of the executions that satisfy the goal $\mathtt{G}$, for a given scenario $\mathtt{S}$.

For the following, let $\mathtt{t}_{\mathtt{ws}}^{\mathtt{max}}$ be the sum of all the durations of the tasks, declared by any server. Observe that such $\mathtt{t}_{\mathtt{ws}}^{\mathtt{max}}$ is an upper bound on the completion time of any instance. Assume the requests $\mathtt{order}(\mathtt{id}_1, \mathtt{i}_1, 5)$ and $\mathtt{order}(\mathtt{id}_2, \mathtt{i}_1, 10)$ are given and that the database is the one shown in Figure 6.

Then, the query $\langle \mathtt{H}, \mathtt{G}, \mathtt{R} \rangle$, where
- $\mathtt{H} : [\mathtt{init}(\mathtt{id}_1)@0,\ \mathtt{init}(\mathtt{id}_2)@3]$,
- $\mathtt{G} : \exists^{@\mathtt{t}_{\mathtt{ws}}^{\mathtt{max}}}(\mathtt{executed}(\mathtt{id}_1) \wedge \mathtt{executed}(\mathtt{id}_2))$, and
- $\mathtt{R} : [\mathtt{availability}(\mathtt{X}, \mathtt{I}, \mathtt{Q})@\mathtt{t}_{\mathtt{ws}}^{\mathtt{max}}]$

will output the availability of products in each store after the satisfaction of the orders — the careful reader may check that these orders may be satisfied by selecting store $s_1$ for request $id_1$, and store $s_2$ for request $id_2$. Conversely, assuming an other order of the form $\mathtt{order}(\mathtt{id}_3, \mathtt{i}_3, 5)$, the query with the goal $\mathtt{G} : \exists^{@\mathtt{t}_{\mathtt{max}}}\ \mathtt{executed}(\mathtt{id}_3)$ and $\mathtt{H} : [\mathtt{init}(\mathtt{id}_1)@0,\ \mathtt{init}(\mathtt{id}_3)@2]$ will output the empty list since there is no way for selling the desired quantity of item $i_3$. □

### *5.1 Computational Complexity of Reasoning on Workflows*

We have just seen how to equip our framework of an interesting querying mechanism which enable to reason on possible execution. We next study the computational complexity of the most common reasoning tasks. In particular, following the *data complexity* approach [25], in all results stated below we will consider a given problem instance having as its input the temporal domain $\mathcal{T}$, the database $\mathtt{DB}$, and the list of envisioned events $\mathtt{H}$, while both the program $\mathcal{P}$ and (possibly) the query $\mathtt{Q}$ are fixed. Recall that the (data) complexity of computing a stable model of a $\mathtt{DATALOG}^{\neg_s}$ program on a given database $\mathtt{EDB}$ can be done in time polynomial on the size of $\mathtt{EDB}$, whereas it requires exponential time (unless $\mathbf{P} = \mathbf{NP}$) if the program is not stratified. In fact, in the latter case, deciding whether there exists a stable model or not is $\mathbf{NP}$-complete [16].

First, we discuss the problem of deciding the existence of temporal and stationary temporal models, evidencing how constraints represent an actual source of complexity.

**Theorem 20 (Temporal model existence)** *Let* $\texttt{H}$ *be a list of envisioned events. Then, deciding whether* $\mathcal{P}(\mathcal{WS})_{\texttt{H}}$*, for a given workflow* $\mathcal{WS}$*, has a temporal model is*

- **NP**-*complete, for general* $\mathcal{WS}$*. Hardness holds even if* $\texttt{H}$ *contains one event only, the schema* $\mathcal{WS}$ *is acyclic, and* $\mathbf{Constr}(\mathcal{WS})$ *contains one constraint only (no matter of the type).*
- *feasible in polynomial time if* $\mathbf{Constr}(\mathcal{WS}) = \emptyset$*.*

**PROOF.**

- *Membership.* Recall that an interpretation $I$ is a model for $\mathcal{P}(\mathcal{WS})$ iff there exist $j > 0$, such that $I \in \widehat{\mathbf{T}}_j$, where $\mathbf{T}$ is the function defined in Definition 11. We claim: $\widehat{\mathbf{T}}_1 \neq \emptyset \Leftrightarrow \mathcal{TM}(\mathcal{P}(\mathcal{WS})_{\texttt{H}}) \neq \emptyset$. In fact,
  ($\Rightarrow$) If there exists $M \in \widehat{\mathbf{T}}_1$, then $M$ is a temporal model (with $j = 1$);
  ($\Leftarrow$) Let $\texttt{t}$ be the first time which is triggered in the program $\mathcal{P}(\mathcal{WS})$, i.e., $\texttt{t} = \mathit{nextTime}(\langle \texttt{EDB}_{ws}, \texttt{H}\rangle)$. If there exists $M' \in \mathcal{TM}(\mathcal{P}(\mathcal{WS})_{\texttt{H}})$, then $M'@\texttt{t}$ must belong to $\widehat{\mathbf{T}}_1$, i.e., $M'@\texttt{t}$ is the first evolution that had lead to the model $M'$.

  It follows that the problem reduces to deciding the non-emptiness of $\mathbf{T}(\{I_0\})$, where $I_0 = \langle \texttt{EDB}_{ws}, \texttt{H}\rangle$. By definition, $\mathbf{T}(\{I_0\})$ contains for any set of transition rules $\mathit{chosen_tr} \in \mathit{ground_TR}(I_0)$, all interpretations $\langle S', E'\rangle$ such that (i) $S' \in \mathrm{SM}(\texttt{DB}_{ws} \cup \texttt{DKB}_{ws} \cup S \cup \mathcal{A}_{I_0}(\mathit{chosen_tr}) \cup \mathcal{R}_{I_0}(\mathit{chosen_tr})) \cup \mathit{triggered}(\texttt{H})$, and (ii) $E' = \texttt{H} \cup \mathcal{E}_{I_0}(\mathit{chosen_tr}) - \mathit{triggered}(\texttt{H})$. Hence, in order to check whether this set is empty we can equivalently check whether the program $\texttt{DB}_{ws} \cup \texttt{DKB}_{ws} \cup S \cup \mathcal{A}_{I_0}(\mathit{chosen_tr})$ have stable models. This latter task is feasible in **NP**[16].

  *Hardness.* Recall that, given a boolean formula $\Phi$ over variables $X_1, ..., X_m$ the problem of deciding whether it is satisfiable is **NP**-complete [6]. W.l.o.g. assume $\Phi$ to be in conjunctive normal form. Then, we define a workflow schema $\mathcal{WS}(\Phi) = \langle A, E, a_o, \{\mathit{UnSat}, \mathit{Sat}\}\rangle$, such that $A$ consists of an initial activity $a_0$ with $a_0 \in A^{\wedge}_{out}$, of the activities $X_i, TX_i, FX_i$ for each $0 < i \leq m$, of the activities $C_j$ and $\overline{C}_j$ for each distinct clause $j$ of $\Phi$, and of two final states $\mathit{UnSat}$ and $\mathit{Sat}$ such that $\mathit{Sat} \in A^{\wedge}_{in}$, and $\mathit{UnSat} \in A^{\vee}_{in}$. The set of precedences $E$ is defined as follows.
  - For each $X_i$, $(X_i, TX_i)$ and $(X_i, FX_i)$ are in $E$, with $X_i \in A^{\wedge}_{in} \cap A^{\vee}_{out}$, $TX_i \in A^{\wedge}_{in} \cap A^{\wedge}_{out}$, and $FX_i \in A^{\wedge}_{in} \cap A^{\wedge}_{out}$. Thus, each time the activity $X_i$ is executed, it is required to make a choice between its possible successors; note that in our encoding, $TX_i$ means that $X_i$ is **true**, while $FX_i$ means that $X_i$ is **false**. Finally an arc $(a_0, X_i)$ is in $E$.
  - For each $C_j$, we have that $(C_j, \mathit{Sat})$ is in $E$, with $C_j \in A^{\vee}_{in} \cap A^{\wedge}_{out}$. Moreover, we have $(TX_i, C_j) \in E$ in the case $X_j$ appears in the clause $C_j$, while we have $(FX_i, C_j) \in E$ in the case $X_i$ appears negated in the clause $C_j$.
  - For each $\overline{C}_j$, we have that $(\overline{C}_j, \mathit{UnSat})$ is in $E$, with $\overline{C}_j \in A^{\wedge}_{in} \cap A^{\wedge}_{out}$. Moreover, we have $(FX_i, \overline{C}_j) \in E$ in the case $X_j$ appears in the clause

$C_j$, while we have $(TX_i, C_j) \in E$ in the case $X_i$ appears negated in the clause $C_j$.

Finally let $\mathtt{H} = [\mathtt{init}(\mathtt{id_1})]$, i.e., we consider one instantiation of the above schema.

We claim that: $\Phi$ is satisfiable $\Leftrightarrow$ the instance $id_1$ may activate the task *Sat*. In fact, if $\Phi$ is satisfiable, then, it is possible to choice the successor of each $X_i$, in a way that all the activities $C_j$ can be executed. Hence, the activity *Sat* will be eventually reached. On the other side, if there exists a path leading to *Sat*, it can be easily mapped into a satisfying assignment for $\Phi$. Conversely, if the truth assignment of the variables do not satisfy $\Phi$, then all the activities $\overline{C}_j$ are executed leading to *UnSat*.

The theorem follows by observing that the reaching of the task *Sat* can be enforced both by constraint $!Sat$ and by $!Sat \prec !UnSat$.

- In the case no constraints are issued on the global schema, it easily follows that the program $\mathcal{P}(\mathcal{WS})$ is stratified modulo choice by construction. Thus, we can exploit the results in [20] and concluding the fact that we can compute any temporal model in polynomial time. $\square$

Note that the problem is harder, if we are not satisfied with any temporal model, and we require the model to be stationary.

**Theorem 21 (Stationary model existence)** *Let* H *be a list of envisioned events. Then, deciding whether* $\mathcal{P}(\mathcal{WS})_{\mathtt{H}}$ *has a stationary model is*

- **PSPACE**-*complete, for general* $\mathcal{WS}$*, and*
- **NP**-*complete, if the associated control flow is acyclic. In this case, if* $\mathbf{Constr}(\mathcal{WS}) = \emptyset$*, then* $\mathcal{P}(\mathcal{WS})$ *always have a stationary model, and any stationary model can be computed in polynomial time.*

**PROOF.**

- *Membership.* We can simply apply in a constructive way the procedure for computing a temporal model $\mathbf{T}$. We start with $I_0 = \langle \mathtt{EDB}_{ws}, \mathtt{H} \rangle$, and we nondeterministically compute (if exists) a model in $I_1 \in \mathbf{T}(\{I_0\})$. Similarly, at each step $j > 1$ we select $I_j \in \mathbf{T}(\{I_{j-1}\})$. Since the number of possible models is bounded by all the possible combinations of the literals in it, we can avoid to return two times in the same state, and hence after an exponential number of steps we can possibly reach a stationary model or deciding that there not exists any. The membership derives from the fact that **NPSPACE** = **PSPACE**.

  *Hardness.* Let us consider the language $Datalog_{1S}$ proposed in [4], for dealing with finite representation of infinite query answers. $Datalog_{1S}$ is an extension of DATALOG in which each predicate has a distinguished temporal argument on which the increment function (+1) can be possibly applied. It is im-

mediate to see that any $Datalog_{1S}$ rule can be reformulated in a suitable event activation rule in $\mathtt{DATALOG}^{\mathtt{ev!}}$ by preserving the underlying semantics. The results follows from the fact **PSPACE**-completeness of reasoning on $Datalog_{1S}$ programs.
- *Membership.* In the case of acyclic control flow, an upper bound on the completion time of any instance is provided by the sum of all the durations of the tasks, declared by any server. Let $\mathtt{t_{max}}$ this time. Then, we can guess an interpretation $I$ in which the time instants must be in the range $[\mathtt{0}..\mathtt{t_{max}}]$, and we can verify that $I$ is indeed a model in polynomial time [10].
  *Hardness.* It is sufficient to observe that in the case of acyclic control flow there exists a stable model if and only if there exists a stationary one. Thus, both the **NP**-hardness result and the tractable case for case $\mathbf{Constr}(\mathcal{WS}) = \emptyset$) easily follow from Theorem 20, which does not require the acyclicity of the graph. □

It turns out that acyclic workflows have an efficient implementation as far as as the computation of one temporal model is concerned. However, if we have to answer a given query $Q$, and hence we are interested in some "particular" temporal models, then the complexity becomes much higher.

**Proposition 22 (Query answering under temporal models)** *Let* $\mathcal{WS}$ *be a workflow, and* $\mathtt{Q} = \langle \mathtt{S}, \mathtt{G}, \mathtt{R} \rangle$ *be a query on it. Then, deciding whether the answer* $\mathtt{Q}(\mathcal{WS})$ *is true is* **NP**-*complete. Hardness holds even for acyclic workflows, with* $\mathbf{Constr}(\mathcal{WS}) = \emptyset$*, and* $\mathtt{G}$ *containing a literal only.*

**PROOF.** *Membership.* Let $\mathtt{t_{max}}$ be the maximum time instant occurring in the literals of $\mathtt{G}$. Then, we can guess an interpretation $I$ in which the time instants must be in the range $[\mathtt{0}..\mathtt{t_{max}}]$, and we can verify that $I$ is indeed a model in polynomial time [10].

*Hardness.* It easily derives from Theorem 20, by letting in the construction $\mathtt{G} = \exists^{@4} Sat$, assuming each activity to have unitary duration. □

Interestingly, query answering under stationary models is not more difficult then deciding the existence of a stationary model.

**Proposition 23 (Query answering under stationary models)** *Let* $\mathcal{WS}$ *be a workflow, and* $\mathtt{Q} = \langle \mathtt{H}, \mathtt{G}, \mathtt{R} \rangle$ *be a query on it. Then, deciding whether the answer of* $\mathtt{Q}^{\mathtt{s}}(\mathcal{WS})$ *is true is*
- **PSPACE**-*complete, for general* $\mathcal{WS}$*, and*
- **NP**-*complete, if the associated control flow is acyclic. Hardness holds even with* $\mathbf{Constr}(\mathcal{WS}) = \emptyset$.

**PROOF.**

- It easily derives from Theorem 21. In fact, the problem is at least hard as deciding the existence of a stationary model, which is **PSPACE**-hard. Moreover, it can be done in non-deterministic polynomial space with the same argument of the membership in Theorem 21.
- Membership in **NP** derives from the same observations in Theorem 21. For the hardness, it suffices to let $\mathtt{G} = \exists^{@4} Sat$ in the **NP**-hardness proof of Theorem 20 (which does not require the acyclicity of the graph). □

## 6 Comparison with Related Work and Conclusion

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.

Approaches with a slightly similar spirit have been already appeared in the literature. For instance, in [5] the use of the *Concurrent Transaction Logic* ($\mathcal{CTR}$) [3] is proposed 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 [19], 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. Among the other graphical formalisms, we mention the use of *Petri Nets* [23] 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 [24] 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.

However there are some important differences both in the spirit and in the technical solutions of these letter approaches w.r.t. our framework. First, $\mathcal{CTR}$ logic and Petri Nets have been used for solving different problems than the simulation on scenarios, which is, instead, the focus of the application of $\mathtt{DATALOG}^{\mathsf{ev!}}$. Specifically, they considered other 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. Nonetheless, neither the notion of querying the state of the workflow after some envisioned executions nor the ability of planning the actions to perform in order to satisfy some goal have been treated there.

Moreover, $\mathtt{DATALOG}^{\mathsf{ev!}}$ is a language for modelling not only the schema of the workflow but also the resources and the servers involved in the process, thus, providing a way for capturing both *static* and *dynamic* aspects of the modelling. Specifically, as for the dynamic aspects, such as checking whether a desired amount of products is available in a given store at a given time, $\mathtt{DATALOG}^{\mathsf{ev!}}$ provides a great flexibility since it allows the designer to explicitly introduce in the modelling the notion of time and of happening of events.

Finally, a very important and distinguishing feature of our $\mathtt{DATALOG}^{\mathsf{ev!}}$ language is the ability to deal with external events, other then internal ones. For instance, in our running example, natural external events are the closing of a store, and the purchasing of a given quantity of products to assign at a given store. All these events can be naturally modelled by means of very simple and intuitive rules, that can be incrementally inserted into the specifications without modifying any other part. This feature will eventually lead to modular specifications and results very important for real workflow systems.

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 detect 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] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] 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.

[3] 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.

[4] J. Chomicki and T. Imielinski. Relational Specifications of Infinite Query Answers. In *ACM SIGMOD International Conference on Management of Data*, pages 174–183. ACM Press, May 1989.

[5] 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.

[6] M.R. Garey and D.S. Johnson, *Computers and Intractability. A Guide to the Theory of* **NP**-*completeness*, Freeman and Comp., NY, USA, 1979.

[7] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[8] 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.

[9] 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.

[10] G. Greco, A.Guzzo, D. Saccà, and F. Scarcello. Event Choice Datalog: A Logic Programming Language for Reasoning in Multiple Dimensions. In *Proc. of Int. Conf. on Principles and Practice of Logic Programming*, ACM Press, Verona, Italy, 2004.

[11] A. Guzzo and D. Saccà. Modelling the Future with Event Choice DATALOG. Proc. AGP Conference,, pages 53-70, September 2002.

[12] 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.

[13] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A Deductive System for Non-monotonic Reasoning. *Proc. LPNMR Conf.*, 363-374, 1997.

[14] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV System for Knowledge Representation and Reasoning (2004) To appear. Available via *http://www.arxiv.org/ps/cs.AI/0211004*.

[15] J.W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 1984.

[16] V.W. Marek and M. Truszczyński. Autoepistemic Logic. *Journal of the ACM*, 38(3):588–619, 1991.

[17] 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.

[18] I. Niemelä and P. Simons. Smodels – An Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. In *Proc. of the 4th Int. Conf. on Logic Programming and Nonmonotonic Reasoning*, pages 420–429, 1997.

[19] P. Senkul, M. Kifer and I.H. Toroslu. A logical Framework for Scheduling Workflows Under Resource Allocation Constraints. In *VLDB*, pages 694-705, 2002.

[20] 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.

[21] 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.

[22] J. D. Ullman. *Principles of Database and Knowledge-Base Management System.* New York: Academic, 1988.

[23] 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.

[24] 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.

[25] M. Vardi. The Complexity of Relational Query Languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pp. 137–146, 1982.

[26] 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.

[27] 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.

[28] The Workflow Management Coalition, *http://www.wfmc.org/*.

[29] C. Zaniolo. Transaction-Conscious Stable Model Semantics for Active Database Rules. In *Proc. Int. Conf. on Deductive Object-Oriented Databases*, 1995.

[30] C. Zaniolo. 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.

[31] C. Zaniolo, N. Arni, and K. Ong. Negation and Aggregates in Recursive Rules: the LDL++ Approach, *Proc. 3rd Int. Conf. on Deductive and Object-Oriented Databases*, 1993.