Formalizing the Specification and Execution of Workflows using the Event Calculus

Nihan Kesim Cicekli[1], Ilyas Cicekli[2]
[1]Department of Computer Engineering, METU, Ankara, Turkey
[2]Department of Computer Engineering, Bilkent University, Ankara, Turkey
nihan@ceng.metu.edu.tr, ilyas@cs.bilkent.edu.tr

Abstract. The event calculus is a logic programming formalism for representing events and their effects especially in database applications. This paper proposes the event calculus as a logic-based methodology for the specification and execution of workflows. It is shown that the control flow graph of a workflow specification can be expressed as a set of logical formulas and the event calculus can be used to specify the role of a workflow manager through a set of rules for the execution dependencies of activities. The proposed framework for a workflow manager maintains a history of events to control the execution of activities. The events are instructions to the workflow manager to coordinate the execution of activities. Based on the already occurred events, the workflow manager triggers new events to schedule new activities in accordance with the control flow graph of the workflow. The net effect is an alternative approach for defining a workflow engine whose operational semantics is naturally integrated with the operational semantics of a deductive database. Within this framework it is possible to model sequential and concurrent activities with or without synchronization. It is also possible to model agent assignment and execution of concurrent workflow instances. The paper, thus, contributes a logical perspective to the task of developing formalization for the workflow management systems.

Keywords: Event calculus, workflow formalization, temporal reasoning.

1. Introduction

A workflow is a collection of cooperating, coordinated activities designed to accomplish a completely or partially automated process. An activity in a workflow is performed by an agent that can be a human, a device or a program. A workflow management system provides support for modeling, executing and monitoring the activities in a workflow. There are many commercial products to model and execute workflows [1,3,22,34] and there have been many formal models proposed for the analysis and reasoning about the workflows [9,16,17,26]. The most common frameworks for specifying workflows are graph-based, event-condition-action rules, and logic-based methods.

Graph-based approaches provide a good way to visualize the overall flow of control, where nodes are associated with activities and edges with control or data flow between activities. Petri nets and state charts are graph-based general-purpose process specification formalisms that have been applied to workflow specifications [23,31]. Event-condition-action rules have been widely used in active databases and they have been adopted in the specification of workflows as well [5,12]. However, their expressive power is not as general as control flow graphs. Logic-based formalisms, on the other hand, use the power of declarative semantics of logic to specify the properties of workflows and the operational

semantics of logical systems to model the execution of workflows. Logic-based approaches mostly deal with the verification of workflows with global constraints [2,24].

We believe that logic-based methods have the benefit of well-defined declarative semantics and well-studied computational models. In this paper we also propose a logic-based framework for the specification and execution of workflows. We use a logic programming approach for the specification of control flow graphs, execution dependencies between activities and scheduling of activities within a workflow. The paper formalizes some important properties of workflow systems. These properties include the specification of main types of flow controls, such as sequential, concurrent, alternative and iterative execution of activities. The paper also presents deductive rules for scheduling activities and assigning agents to perform these activities. As another important issue, the paper deals with the execution of concurrent workflow instances. Other issues such as representing the transactional properties of workflows, or temporal constraints (global constraints) between workflow activities are out of the scope of this paper.

The proposed approach is based on the Kowalski and Sergot's Event Calculus [18]. Event Calculus, abbreviated as EC, is a simple temporal formalism designed to model situations characterized by a set of *events*, whose occurrences have the effect of initiating or terminating the validity of determined *properties*. Given a description of when these events take place and of the properties they affect, EC is able to determine the maximal validity intervals over which a property holds uninterruptedly. The EC uses a polynomial algorithm for the verification or calculation of the maximal validity intervals and its axioms can easily be implemented as a logic program [6].

EC provides mechanisms for storing and querying the history of all known events. Once the event occurrences until time *t* are known, the state of the system can be computed at any point of time until *t*. In order to be able to model the invocation of activities in a workflow, we need to be able to represent that certain type of event invariably follows a certain other type of event, or that a certain type of event occurs when some property holds. In our framework events are treated as triggers that denote the start or end times of activities. We also consider a set of external events, which might be recorded by the activities themselves or by the user externally. Once we know the history of all events either explicitly recorded or automatically generated by the system, the modeling of workflow execution becomes the computation of new events from the history and thus executing new activities until the end of the workflow is reached. The most important result made possible by this approach is the definition of the operational semantics of event detection, condition verification and activity scheduling in terms of a well-defined semantics, which can be computed by that of a deductive system and queries.

The paper presents a simple scheduling algorithm in which it is possible to model agents as separate entities and assign agents to certain activities based on their cost. The workflow manager is designed to choose the best agent to perform the next scheduled activity among all available agents qualified to do that activity. The representation of events, activities and agents in this framework makes it also possible to model the execution of concurrent workflow instances over a single workflow specification.

The main contribution of the paper is to present the use of event calculus approach in the formalization of an important set of properties of workflow systems. The approach allows the user to specify sequential and concurrent execution of activities; conditional

transitions between activities; and also iteration of activities. The given specification can be executed by means of some deductive rules and queries. The proposed framework has been easily implemented as a logic program. It can be used as a quick tool in the simulation, and testing of experimental workflows. It can be used to analyze the behavior of workflows for different control flows with different number of agents and workflow instances. It may also serve the need for querying some piece of information in the process history. Or it may serve the need for querying the history of the workflow to analyze and assess the efficiency, accuracy and the timeliness of the activities by deriving the state of the workflow at any time in the past.

To the best of our knowledge, we are not aware of any other *logic-based* formalism in which it is possible to specify all the activity execution routings that we support in this paper and to execute concurrent workflow instances with appropriate agent assignments within the same uniform framework. In the preliminary versions of this paper [15,16], we propose an outline of the use of the event calculus as a basis for complex workflow specifications where concurrent activities, agents and concurrent workflow instances can be modeled. However many of the axioms were application specific and a large set of rules must be written to capture the different aspects of the workflow at hand. In this paper we overcome these difficulties by proposing general rules that will be applicable to any workflow specification that includes the set of activity dependencies covered by our formalism.

The rest of the paper is organized as follows. Section 2 summarizes the basics of the event calculus by presenting the major axioms that are used in this paper. Section 3 discusses control flow graphs, relationship between events and activities, and also proposes a naming convention to uniquely identify events and activities to support concurrent workflow instances. Section 4 presents the rules for the local execution dependencies of sequential, concurrent, alternative and iterative activities in a workflow. The functionality of the workflow manager is described in Section 5 by presenting rules to start and end activities and assign agents to activities in concurrent workflow instances. The computational issues are discussed in Section 6 which also describes the implementation of the proposed framework. Section 7 presents a conceptual architecture of a workflow management system for a more realistic implementation of the framework. Section 8 discusses the related work by comparing them with the proposed approach in this paper. The paper is concluded by summarizing the features of the proposed framework and possible future extensions in Section 9.

2. Event Calculus

The event calculus is a logic programming formalism for representing events and their effects, especially in database applications [18]. A number of event calculus dialects have been presented since the original paper [13,14,25]. The one described here is based on a later simplified version presented in [19]. In contrast with the definition in [19], two assumptions are made in this version of the event calculus: The events have no extended duration, and the properties that events initiate, hold in the period initiated by the event and contain the said event. These assumptions simplify the formulation and implementation of the event calculus, but, otherwise nothing essential depends on them.

The event calculus is based on general axioms concerning notions of events, properties and the periods of time for which the properties hold. The events initiate and/or terminate periods of time in which a property holds. As events occur in the domain of the application, the general axioms imply new properties that hold true in the new state of the world being modeled, and infer the termination of properties that no longer hold true from the previous state.

The main axiom (also called the *persistence axiom*) used by the event calculus to infer that a property holds true at a time is described as follows[1]:

holds_at(*P, T*) ←
    happens(*E, T1*) , *T1* ≤ *T* , *initiates*(*E, P*), *not broken*(*P, T1, T*).

Here, the predicate *holds_at*(*P, T*) represents that property *P* holds at time *T*; the predicate *happens*(*E, T*) represents that the event *E* occurs at time *T*; the predicate *initiates*(*E, P*) represents that the event *E* initiates a period of time during which the property *P* holds; and the predicate *broken*(*P, T1, T2*) represents that the property *P* ceases to hold between *T1* and *T2* (inclusive) due to an event which terminates it. The time points are ordered by the usual comparative operators. The *not* operator is interpreted as negation-as-failure. The use of negation-as-failure gives a form of default persistence into the future. Thus, the persistence axiom states that once a property *P* is initiated by an event *E* at time *T1*, it holds for an open period of time containing time point *T1* (i.e. [*T1, T*)), unless there is another event happened at some point of time after *T1*, that breaks the persistence of property *P*.

Other axioms used in the body of this axiom are defined as follows. The axiom for *happens*(*E, T*) is usually defined as an extensional predicate symbol that records the happening of the event *E* at time point *T*. A particular course of events that occur in the real world being modeled is represented with a set of such extensional predicates. The axiom for *broken*(*P, T1, T2*) is defined by the following clause:

broken(*P, T1, T2*) ←
    happens(*E, T*), *terminates*(*E, P*), *T1* ≤ *T* ≤ *T2*.

That is, the persistence of the property *P* is broken at time point *T2* if a distinct event *E* that happened at time *T* between *T1* and *T2* terminates the persistence of *P*. Here the predicate *terminates*(*E, P*) represents that the event *E* terminates any ongoing period during which property *P* holds.

Finally the axioms for *initiates* and *terminates* are specific to the application at hand. The problem domain is captured by a set of *initiates* and *terminates* clauses. For instance, the following rule describes the effect of an event of promoting an employee:

initiates(*E, rank*(*Employee, Title*)) ←
    event(*E*), *act*(*E, promote*), *actor*(*E, Employee*), *role*(*E,Title*).

Here the property *rank*(*Employee, Title*) denotes a property in the application's database that starts to hold after the occurrence of the event E. The details of the event specification can be given as a set of binary predicates (semantic networks) as described in [18].

[1] The usual convention of using uppercase letters to represent logical variables is followed throughout the paper.

When an employee leaves the job, the property *rank*(*Employee, Title*) ceases to hold. This is described by the following rule in which the anonymous variable underscore in logic programming is used in place of *Title*, since the title value is not used in the body of the rule:

terminates(*E, rank*(*Employee, _*)) ←
    event(*E*), *act*(*E, lay_off*), *actor*(*E, Employee*).

EC is defined as the collection of all types of axioms described above. Once the event occurrences until time *t* are known, the state of the system can be computed at any point of time until *t* using the *holds_at* predicate. The event occurrences are recorded as an extensional database and snapshots of the database state can be derived at any time using this history of events.We can extend the EC by adding the definition of other predicates such as *holds_for*(*Property, TimePeriod*) to find out the period of time for which a property holds:

holds_for(*P, T1, T2*) ←
    happens(*E1, T1*), *initiates*(*E1, P*), *happens*(*E2, T2*),
    terminates(*E2, P*), *not broken*(*P, T1, T2*).

Alternatively, as in [11] we can define *holdsNow*(*Property*) to point implicitly to the current state, under the assumption that *Now* can be initiated with the time point that corresponds to the system clock at invocation time.

holdsNow(*Property*) ←
    clock(*Now*), *holds_at*(*Property, Now*).

In [11], the event calculus is used to formalize a large set of syntactic and semantic aspects of active databases. The approach to the formalization is centered on the idea of using a *history* as defined in the EC, to define event occurrences, database states, and actions on these. A history is a particular form of an extensional database containing representations of event occurrences. The authors show how the history is used with the event calculus to give rise to a sequence of extensional databases in the application. Broadly, event and condition specifications are given a Datalog-related operational semantics, while action specifications denote the addition of new axioms to the logical theory that is the representation of the history.

In this paper, we show how the event calculus can be used in the specification and the execution of workflows. That is, we show not only the activation of event-condition-action rules but also other forms of activity invocations. A workflow process definition contains a collection of activities and the order of activity invocations or conditions under which activities must be invoked (i.e. control flow) and also data flow between the activities. This paper proposes a formalization of workflow process definitions and their executions within the framework of the event calculus. In the proposed approach, events denote the start and end time points of activities and the state of the workflow is described by properties. Thus, events will be used to specify the control flow and the effects of the events are used to describe the data flow within the workflow.

3. Workflow Concepts

In this section we briefly provide the definitions of basic concepts of workflow systems that are used throughout the paper. Then, the basic concepts of workflow systems are associated with the constructs of the event calculus.

3.1. Basic Definitions

A *workflow* is a process involving the coordinated execution of multiple activities performed by different processing entities. Examples of workflows are processing of purchase orders over the Internet, processing of insurance claims, mail routing in an office etc. An *activity* (*task*) defines some work to be done. Examples of tasks include updating a database, generating a bill, mailing a form. An *agent* is a processing entity that performs the activities. An agent may be a person, a hardware device or a software system (e.g. a mailer, an application program, a database management system). Human tasks include interacting with computers such as providing input commands. A *workflow instance* is an enactment of a workflow. It is possible that several instances of a workflow can run concurrently. For example, a workflow manager can execute several processing orders at the same time.

Specification (or design) of a workflow involves describing those aspects of its constituent activities and the agents that execute them. It also defines the relationships among activities and their execution requirements. *Execution* of the multiple activities by different agents may be controlled by a human coordinator or by a software system called a *workflow management system*. In this paper we are interested in designing a workflow manager within the framework of the event calculus. For this purpose we first discuss the specification of workflows in a logical framework. We then provide the rules to specify the execution requirements of workflows.

3.2. Specification of Workflows

The Workflow Management Coalition (WfMC) defines a reference model that describes the major components and interfaces within a workflow architecture [35]. In a workflow, activities are related to one another via flow control conditions (transition information). It is possible to design workflow with many different transition patterns [33]. Accordingly we identify the following basic routings among the activities:

1. *Sequential*: Activities are executed in sequence (i.e. one activity is followed by the next activity.)
2. *Parallel*: Two or more activities are executed in parallel. Two building blocks are identified: (a) AND-split and (b) AND-join. AND-split enables two or more activities to be executed concurrently after another activity has been completed. The AND-join synchronizes the parallel flows, one activity starts only after all activities in the join have been completed.
3. *Conditional*: One of the alternative activities is executed. In order to model a choice among two or more alternatives two blocks can be used: (a) XOR-split and (b) XOR-join. In XOR-split, based on a condition check, only one of several branches is chosen. In XOR-join it is assumed that none of the alternative branches is ever executed in parallel.

4. *Iteration*: It may sometimes be necessary to execute an activity or a set of activities multiple times.

Among the most common frameworks for specifying workflows, control flow graphs are most appropriate for showing the execution dependencies of the activities in a workflow. It provides a good way to visualize the overall flow of control. In a control flow graph the vertices identify the names of activities. The edges represent the successor relation on the activities. A typical graph specifies the initial and the final activities in a workflow, the subsequent activities for each activity in the graph, and whether all of these subsequent activities must be executed concurrently, or it is sufficient to execute only one branch depending on a condition.

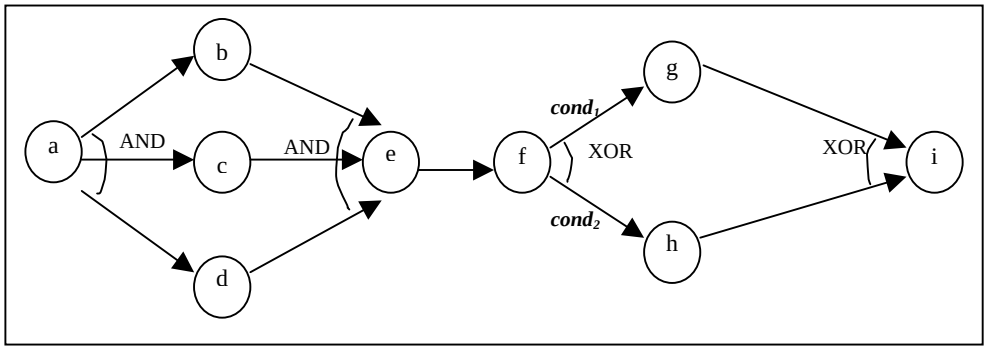

Fig. 1. An example control flow graph

Fig. 1 illustrates a control flow graph where the activity **a** is the initial task, and **i** is the final activity. After the activity **a**, the subsequent activities **b**, **c** and **d** are executed concurrently, which is indicated with the "AND" label. Activity **e** can only start after activities **b**, **c**, and **d** are completed. After the activity **e** is completed the activity **f** can start. The splitting branch labeled as "XOR" indicates that when activity **f** is finished, there is a choice of executing **g** or **h**. By the definition of XOR-split, only one of the conditions $cond_1$ or $cond_2$ will be true, and either activity **g** or activity **h** will start executing depending on which condition holds. The conditions are based on workflow control data and apply to the current state of the workflow. The conditions can depend on some logical status, or output generated by some prior activity in the workflow, or on the value of some external variable (e.g. time). Activity **i** is enabled immediately after either one of the activities **g** or **h** is completed.

As a real example, the control graph shown in Fig. 1 can be viewed as the workflow of paper reviewing process. When a paper is submitted electronically (external event), the workflow starts with the initial activity **a**, say *select reviewers*. The agent of this activity is a person (the editor). Once three reviewers are selected the paper is distributed to the reviewers and the reviewers (person agents) *review* the paper concurrently (the concurrent activities **b,c,d**). The subsequent activity **e**, say *combine reviews* is activated only when three reviews are completed. The agent of this activity can be a computer program which notifies the editor via email. Then the next activity **f** of *decision making* is done by the

Table 1. Successor relationships between activities

Predicate	Description
initial_activity(A)	A is the first activity in the workflow
sequential(A1,A2)	A2 follows A1 unconditionally
and_split(A,L)	A is followed by a list of activities L
xor_split(A,ActCondPairs)	A is followed by A_i in ActCondPairs if condition $Cond_i$ is true
and_join(L, A)	A starts after all the list L of activities complete
xor_join(L, A)	A starts after one of the list L of activities completes
final_activity(A)	A is the last activity in the workflow

editor and a decision of reject or accept will be made. If the decision is accept the next activity **g** will be *prepare an accept* letter. If the decision is reject the next activity **h** will be *prepare a reject* letter. Only one of the alternatives will be executed. Finally, the review is forwarded to the author of the paper (activity **i**).

3.2.1. Control flow graph described as a set of logical formulas

A given control flow graph can be represented as a set of predicates in first-order logic. In this paper, we consider five different successor relations between activities. We represent these relations with separate predicate symbols which are described in Table 1. For instance, the workflow depicted in Fig. 1 can be described by a set of predicates as follows:

initial_activity(*a*).
and_split(*a*,[*b,c,d*]).
and_join([*b,c,d*], *e*).
sequential(*e, f*).
xor_split(*f,* [(*g, cond*$_1$), (*h, cond*$_2$)]).
xor_join([*g,h*],*i*).
final_activity(*i*).

This example does not include an iterative execution structure. The specification of iteration is described in Section 4.4 separately.

This set of predicates maps the formal structure of the control flow graph directly into a set of logic formulas. The actual execution order of activities is determined by the workflow manager. The workflow manager uses execution dependency rules to determine which activity needs to be scheduled next. The execution dependency rules are various scheduling preconditions and they are described as axioms within the framework of the event calculus (see section 4, axioms AxS 1-8). However before introducing the axioms for execution dependencies, we first describe the relationship between activities in a workflow and the events in the event calculus.

3.2.2. Events and Activities

In the event calculus, events have no duration. The occurrences of events are considered as instantaneous happenings that are recorded in the database. Activities in a workflow, however, have duration. Agents need time to carry out their tasks. The period of time necessary to complete an activity can be either fixed or varying depending on the nature of the activity. For instance if the activity involves a mechanical task its duration may be fixed. However if the activity is performed by a human the duration of the activity can be varying. In workflow systems, a workflow specification is generally not concerned with the details of the internal operations of the activities, but rather with the way the activities are sequenced. A workflow manager is concerned only with those aspects of an activity that are externally visible on the workflow level. Thus for a workflow manager, an activity can be in one of the possible execution states (such as initial, executing, committed etc.) and state transitions are enabled in terms of externally observable events, such as start and commit. In our framework each activity is initiated by an event and its termination is regarded as another event that records the completion of that activity. Thus each activity ***A*** has a starting event ***start***(***A***) and an ending event ***end***(***A***). Once we know the times of these events, the duration of the activity can be derived easily. The relationship between the activities and events is described in Fig. 2. Notice that, between these two special events, the activity is in execution state and the internal operation of the activity is unknown to the workflow manager. We do not model the internal behavior of the activities in the event calculus.

Activities are executed by agents. The workflow manager assigns activities to agents and agents execute the activities. The workflow manager maintains the states of activities by recording their starting and ending times. The starting time of the activity corresponds to the time of its start event which is triggered by the workflow manager. The ending time of the activity corresponds to its end event which is sent by the agent to the workflow manager. If it is a fixed duration activity (e.g. agent is a hardware device and performs an automatic task), the end event will be sent by the agent within a predefined period of time. If the duration of the activity varies, then its execution time period may depend on some conditions or occurrences of some external events. The conditions that describe the end of the activity may be produced by the agent performing the activity. For instance, the activity may be a computer program and it may finish only when the user of the program fills in and submits a form. Such an input can be considered as an external event. Then the agent

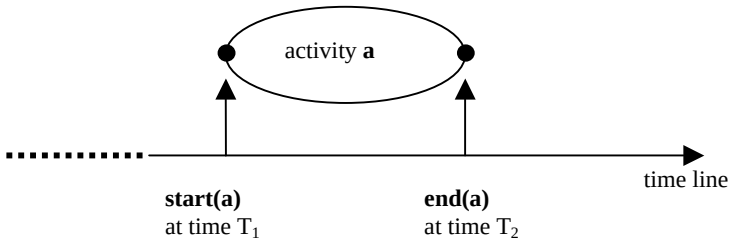

Fig. 2. Events start and end activities.

will terminate its execution by sending end activity event to the workflow manager. The execution duration of an activity is therefore application dependent and the activity must be designed to inform the workflow manager of its completion.

In this paper we view the activities as independent modules executed by proper agents and the implementation details of activities are out of the scope of this paper. We consider only their interfaces with the workflow manager in terms of their starting time, ending time and any relevant data that they generate to affect the workflow execution. In the event calculus, the interaction of activities with the workflow manager is *simulated* by the use of axioms AxH 3 and AxH 4 that are presented in Section 5.2.1.

3.2.3. Concurrent workflow instances and naming conventions

One of the objectives of this paper is to express the execution of concurrent workflow instances over the same workflow specification. For instance, if the workflow describes the activities in an order processing application, there may be more than one order being processed at the same time. In order to be able to model such concurrent instances of a given workflow and the execution of the same activities for different workflow instances, we use a special naming convention.

Each workflow instance is given a unique name (identity). This unique identity is an atomic term and it can be generated by the system when the workflow instance is started. Since each activity is executed at different times for different workflow instances, their names must be associated with an execution id to identify each of these executions. In its simplest form, this execution id will be the workflow instance id. For example, an execution of activity **e** in Fig. 1, in a workflow instance *w1* can be represented by the term *act*(*e,w1*), and when it is completed it can trigger the execution of the activity **f** with the same workflow id, i.e. *act*(*f,w1*).

In a workflow specification, one may also use iteration of activities in the specification of a workflow. An activity in an iteration block can be executed more than once, and each execution of that activity should be uniquely identified. The block name together with an iteration number can be used to uniquely identify each execution of an activity in the iteration block. This means that the naming convention should be general enough to express the different executions of the same activity in different iterations.

In order to be able to successfully address these issues, we use the following naming convention for identifying the different executions of activities: Each activity execution is represented by a term *act*(*ActName,EID*) where *ActName* is the name of the activity given by the user at the specification, and *EID* is the execution id of the activity generated by the system. An execution id *EID* of an activity is defined as follows:

i. *EID* can be an atomic term, which is simply the workflow instance id. In this case, the activity execution is identified with the activity name and the workflow instance id only.
ii. In case of specifying the execution of an activity within an iteration block, *EID* can be a functional term of the form *b*(*ParentEID,BlockName,IterationNo*) where *ParentEID* is the execution identity of the activity after which this iteration block is started, *BlockName* is the name of the iteration block, and *IterationNo* represents the iteration number for that block. The use of *ParentEID* allows us to uniquely identify the

executions of activities at any nesting level in the iteration blocks, as described in Section 4.4.

For example, the workflow in Fig. 1 is actually translated into the following predicates in our framework, using the naming conventions described here:

initial_activity(*act*(*a,EID*)).
and_split(*act*(*a,EID*), [*act*(*b,EID*), *act*(*c,EID*), *act*(*d,EID*)]).
and_join([*act*(*b,EID*), *act*(*c,EID*), *act*(*d,EID*)], *act*(*e,EID*)).
sequential(*act*(*e,EID*), *act*(*f,EID*)).
xor_split(*act*(*f,EID*),[(*act*(*g,EID*), $cond_1$), (*act*(*h,EID*), $cond_2$)].
xor_join([*act*(*g,EID*), *act*(*h,EID*)], *act*(*i,EID*)).
final_activity(*act*(*i,EID*)).

We need to identify the event occurrences uniquely too. In a workflow system, each activity is carried out by an agent and several agents may qualify to execute one activity. The same activity may be executed by different agents in different instances of the workflow. Thus, agent assignment is another consideration in naming the events. We use the following naming convention in describing the events that start and end an activity: The starting event for an activity *A* that is to be carried out by the agent *Ag* in a workflow instance *W* is described as *start*(*A,Ag,W*), and the ending event is identified as *end*(*A,Ag,W*). The workflow instance id is already included in the naming of the activity, however it is separately held in the naming of events too, because it simplifies the rules that we describe below.

4. Execution Dependencies of Activities

This section presents a logic-based formalization for the execution dependencies of activities in a workflow. The execution order of activities depends on the successor relation among activities, and conditions that are currently satisfied on the system state. Since we support the execution of multiple workflow instances, we include the workflow number in establishing the local execution dependencies between the activities within the same workflow instance.

The execution dependencies between the activities are described by rules for defining the four argument predicate *follows*. The semantics of a formula in the form: *follows*(*Act1,Act2,W,T*) represents the fact that, *Act2* follows *Act1* in the workflow instance *W* at time *T*. In the following subsections we present the rules for the predicate *follows* for each successor relation that we consider in this paper. These rules, mainly, describe the scheduling preconditions of activities and therefore they are named as axioms for scheduling (**AxS** in short).

4.1. Sequential Activities

Fig. 3 shows a graphical representation of sequential routing of activities. When activity a_i finishes, the next activity a_j can start unconditionally. For sequential activities, we can write the following execution dependency rule:

follows(*Act1, Act2, W, T*) ← **(AxS 1)**
sequential(*Act1, Act2*), *happens*(*end*(*Act1, _, W*), *T*).

i.e. *Act2* follows *Act1* in a workflow instance *W* at a time *T* when *Act1* finishes in that workflow instance *W* at the time *T*. The anonymous variable underscore is used in place of the agent name to denote that the rule is valid for any agent.

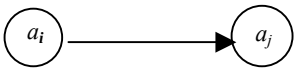

Fig. 3. Activity a_j starts when a_i finishes.

4.2. AND-split and AND-join

In a workflow, activities after an AND-split are scheduled to be executed concurrently. Fig. 4.a illustrates an AND-split. When the activity a_i finishes, activities $a_1, a_2, \ldots a_n$ can start concurrently. Fig. 4.b illustrates AND-join. Here the activity a_j can start when all the preceding activities $b_1, b_2, \ldots b_m$ finish.

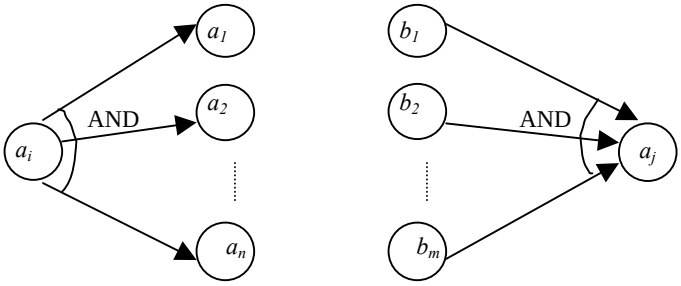

Fig. 4. a) AND-split b) AND-join

When the end of activity a_i is recorded, all subsequent activities are scheduled. Similarly, the activity a_j can be scheduled only when the ending events of all its predecessor activities are recorded. Thus we represent the execution dependency of an AND-split with the following rule:

follows(*Act1, Act2, W, T*) ← **(AxS 2)**
and_split(*Act1, ActList*), *happens*(*end*(*Act1, _, W*), *T*), *member*(*Act2, ActList*).

Here, predicate *member* will be true when *Act2* is a member of the activity list *ActList* in AND-split. The rule expresses the fact that every member of this list must follow the activity at the branch.

The following rule is used to represent the execution of an AND-join of activities:

follows(*Act1, Act2, W, T*) ← (**AxS 3**)
and_join(*ActList, Act2*),
findActEndTimePairs(*ActList, W, ActEndTimePairs*),
actWithMaxEndTime(*ActEndTimePairs, Act1, T*).

The rule uses the predicate *findActEndTimePairs* that holds when all predecessor activities in *ActList* are completed in a workflow instance *W*. If this predicate holds, *ActEndTimePairs* will be the list of all predecessor activities together with their ending times. Then the predicate *actWithMaxEndTime* picks the predecessor activity with the latest ending time. In Fig. 4.b, activity a_j must wait for the completion of all predecessor activities b_1 .. b_m. The last conjunct in this rule ensures that a_j is scheduled at the time of the last ending activity among activities $b_1,\ldots,b_m$. The definitions of predicates *findActEndTimePairs* and *actWithMaxEndTime* are given in the appendix.

4.3. XOR-split and XOR-join

In an XOR-split only one of the alternative activities is executed depending on the evaluated condition. The important point here is that only one of the conditions should hold true at the time of the decision in order to guarantee that only one execution path is chosen.

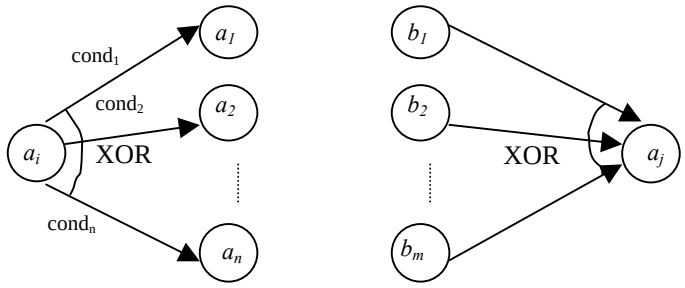

Fig. 5. a) XOR-split b)XOR-join

In an XOR-split (Fig. 5.a), when the activity a_i ends, one of the activities a_1, a_2, ..., a_n can start depending on the condition satisfied at that time. The conditions may be a state check (i.e. a *holds_at* predicate) to verify that some property holds either in the underlying database or in the workflow state.

follows(*Act1, Act2, W, T*) ← **(AxS 4)**
xor_split(*Act1, ActCondPairs*), *happens*(*end*(*Act1, _, W*), *T1*),
member((*Act2,Cond2*), *ActCondPairs*),
initiates(*Ev, Cond2*), *happens*(*Ev, T2*), *max*([*T1,T2*], *T*),
holds_at(*Cond2, T*).

Here we assume that one of the conditions at the split should evaluate to true. If none of the conditions hold then none of the execution paths can be chosen. The idea is to consider each alternate path one-by-one and check if its condition is true. This is achieved by the predicate *member* which is used to retrieve activity-condition pairs one by one from the list of activities in the XOR-split. The picked activity *Act2* will be scheduled in a workflow instance *W* at time *T* only if *T* is the later of the two time points: (i) the ending time of *Act1*, and (ii) the time of the event that initiates the condition *Cond2* for *Act2*. We must also check that *Cond2* still holds at time *T*.

In an XOR-join (Fig. 5.b) if any one of the incoming activities is finished, the activity at the join can start executing. Given that no parallel execution of alternative threads can occur, this pattern corresponds to a simple merge. Thus we represent the XOR-join by the following rule: -

follows(*Act1, Act2, W, T*) ← **(AxS 5)**
xor_join(*ActList, Act2*), *findOneActEndTimePair*(*ActList, W, Act1, T*).

The rule uses the predicate *findOneActEndTimePair* which holds when one of predecessor activities in *ActList* is completed in a workflow instance *W*. If this predicate holds, *Act1* will be the completed predecessor activity and *T* will be its ending time. Thus, the subsequent activity is scheduled at time *T* of the first ending activity. The definition of predicate *findOneActEndTimePair* is given in the appendix.

4.4. Iteration

In some workflow applications it might be necessary to execute a group of activities one or more times. The number of times these activities are executed may depend on some workflow state, or it can be a fixed number. Fig. 6 sketches a control flow graph which includes such a loop structure. The graph illustrates a post-condition checking loop structure. That is, the activities a_1 to a_n are executed at least once, then the iteration condition is checked. While the condition holds, the activities are executed again. The activities a_1 through a_n can be arranged in any of the transition types that we have discussed above.

4.4.1. Specification of the loop structure

In our framework, the body of the loop structure is considered as a block and each block is given a unique name. We use the predicate *serial* in the specification of the workflow, in order to describe that a block follows an activity, or a block is followed by an activity. Each block has an initial and final activity. Since the activities within the block are executed several times within a workflow instance, each execution must be identified uniquely within the history of events. For this purpose, we use the naming conventions for the activities described in Section 3.2.3 for the loop structures while translating the iteration into a set of logic formulas. Each execution of an activity in a loop is identified

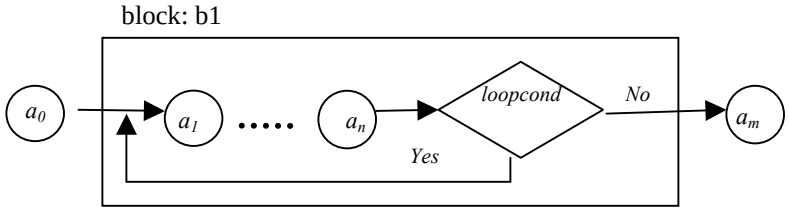

Fig. 6. Activities a_1 to a_n are executed while the condition is true.

with a term of the form:

act(*ActName,b*(*ParentID, BlockName, IterationNo*))

where *ActName* is the user defined name for the activity, and *b*(*ParentID, BlockName, IterationNo*) is the execution id of this activity. For instance, the activity a_1 is represented with the term *act*(a_1, *b*(*w1* ,*b1, I*)), where *w1* is the workflow-id of the workflow instance which starts the iteration block *b1*, and *I* represents the iteration number during execution. Thus, the specification of block *b1* in Fig. 6 includes the following formulas:

serial(*act*(a_0, *EID*), *block*(*b1,EID*)).
serial(*block*(*b1,EID*), *act*(a_m,*EID*), *loopcond*).
initial(*block*(*b1,EID*), *act*(a_1,*b*(*EID, b1, I*)).
final(*block*(*b1,EID*), *act*(a_n, *b*(*EID, b1, I*))).

The set of logical formulas above for the iteration block *b1* indicates that after the activity a_0 with an execution id *EID*, the iteration block *b1* with the same execution id *EID* will start. The activity a_m will start with the same execution id (*EID*) after the block *b1* if the condition *loopcond* does not hold at the time when the last activity of this block is completed. If the execution id of the block is w_1, the execution ids of all activities in this block will be *b*(w_1, *b1, I*). The predicates used for the representation of blocks in a workflow graph are listed in Table 2. The control flow structures between activities within the block are still described with the predicates that we introduced in Table 1 using the naming conventions described in Section 3.2.3. For instance, a sequential transition between two activities, say *a1* and *a2*, in the block is described as:

sequential(*act*(*a1,EID*), *act*(*a2, EID*)).

Since *EID*'s carry the block name and the iteration number, activity *a2* in block *b1* follows *a1* at every iteration sequentially.

Table 2. Predicates to represent blocks

Predicate	**Description**
initial(B, Act)	Act is the first activity in block B
serial(Act, B)	Block B is subsequent to activity Act
serial(B, Act, Cond)	Subsequent to block B is activity Act with the loop condition Cond
final(B, Act)	Act is the last activity in block B

4.4.2. Rules for the execution dependency of a block

In the following we introduce three rules to describe the execution dependency of a block in a workflow in our framework. The first rule is used to start the first activity in a block with iteration number 1:

follows(*Act1, InitAct, W, T*) ← **(AxS 6)**
serial(*Act1, B*), *happens*(*end*(*Act1, _, W*), *T*),
initial(*B, InitAct*), *setIterationNo*(*InitAct,1*).

The rule states that, after activity *Act1*, the next activity is the initial activity *InitAct* of block *B* if block *B* is in sequence with activity *Act1* in the workflow *W* at time *T*. The iteration number for the initial activity *InitAct* is set to 1 since this is going to be its first execution in the current workflow instance (see the appendix for the definition of the predicate *setIterationNo*). The next rule represents the case of exiting the block:

follows(*FnlAct, Act2, W, T*) ← **(AxS 7)**
serial(*B, Act2, Cond*), *final*(*B, FnlAct*),
happens(*end*(*FnlAct, _, W*), *T*), *not holds_at*(*Cond, T*).

The rule states that in a workflow instance *W*, the next activity after the final activity of a block is activity *Act2*, if the block is followed by activity *Act2* and the loop condition does not hold at the time of the final activity is completed. Finally, we describe the iteration of the activities in the block with the following rule:

follows(*FnlAct, InitAct, W, T*) ← **(AxS 8)**
initial(*B, InitAct*), *final*(*B, FnlAct*), *serial*(*B, _, Cond*),
happens(*end*(*FnlAct, _, W*), *T*), *holds_at*(*Cond, T*), *getIterationNo*(*FnlAct, I*),
J = I + 1, setIterationNo(*InitAct, J*).

This rule states that if the final activity of a block with the iteration number *I* is completed in the workflow *W* at time *T*, the initial activity of that block can start with iteration number *I* +1 if the loop condition holds at time *T* (see the appendix for the definition of the predicate *getIterationNo*).

We assume that the condition of a loop can be initiated and terminated by either external events or system-generated events for activities. In this section we described the specification of a post-condition checking loop structure. However, it is possible to describe pre-condition checking loop structures in a similar fashion.

The naming convention used in identifying the execution of activities within a block allows us to represent nested loop structures in a control flow graph too. The execution id of an activity in the nested block will carry the execution id of the activities in its outer loop (parent block). Thus an activity in an inner loop will be initiated with an id which includes the id of this parent block activities. This allows us to uniquely identify the execution of the activities in the inner loop(s). For instance, assuming that there is another block, say *b2*, defined inside block *b1* in Fig. 6, the execution id of an activity in block *b2* in a workflow instance *w1* will be: *b*(*b*(*w1, b1, I*), *b2, J*). Here *b*(*w1,b1,I*) is the execution id of the activity in the parent block *b1* after which *b2* is started and *I* represents the correct iteration number during execution; *b2* is the current block name, and *J* is the iteration within the inner loop. This nesting of execution ids through the parent id makes it possible to nest several loop structures within the same workflow.

The specifications and rules for the iterative structures that we discussed in this section can be extended to represent sub-workflows in a workflow. A sub-workflow can be viewed as a block with a unique name. A sub-workflow can start after an activity of the workflow instance, and when that sub-workflow is completed another activity in that workflow instance can start. Each activity in a sub-workflow instance can be uniquely identified with the unique name assigned to that sub-workflow instance and the unique name assigned to the instance of the workflow that started that sub-workflow instance. The naming

Table 3. Execution states of activities

State of Activity	***Meaning***	***Initiating event***
active(Act, Ag, W)	Activity Act is being executed by agent Ag in workflow instance W	start(Act, Ag, W)
completed(Act, Ag, W)	Activity Act is completed in workflow instance W	end(Act, Ag, W)
waiting(Act2, Ag2, W, T)	Activity Act2 is in the worklist of agent Ag2 in W with timestamp T	release(Ag1, Act1, W)

convention described in this section can easily be extended to cover sub-workflows too. However we will not discuss the details of executing sub-workflows any further in this paper.

5. Workflow Management

A workflow management system must permit the specification and execution of activities. We have so far presented the axioms necessary for the specification of workflow activities and the description of scheduling preconditions among the activities within the current logical framework. In this section, we explain the execution semantics of the workflows through the event calculus. We first describe the representation of the system state maintained by the workflow manager; next we present the rules for the execution of activities by appropriate agents.

5.1. Workflow State

At any time the execution state of a workflow can be defined as a collection of states of its constituent activities and agents. As event occurrences are recorded and activities are executed, the state of the workflow changes. The state of the workflow is derived through the event calculus axioms. The workflow manager keeps track of agent assignment and schedules new activities according to the workflow specification. At any point in time, it is desirable to check which activities are being executed, which ones are completed, which agents are idle and which ones are assigned to tasks, etc.

Each activity is characterized with a set of executable states and transitions between these states. An activity may be in either of the following states: an initial state (*waiting*), executing state (*active*) and done state (*completed*). When the workflow manager determines the next activity to be executed, it puts the activity into the worklists of all agents that can perform that activity; and the activity enters in *waiting* state. When an agent retrieves the activity from its worklist and starts executing it, the activity enters in *active* state. When the agent finishes executing the activity, it enters the *completed* state.

Each agent is associated with a worklist that shows which activities are waiting for that agent. The property *waiting* is also used to represent the worklists of agents since it includes the information about which activity is waiting for which agent. The property *waiting*(*Act, Agent, W, T*) describes that activity *Act* is waiting for agent *Agent* in a particular workflow instance *W*. The time variable *T* denotes the point of time at which the activity started waiting for the agent. An agent can be in either of the following two states: *idle* or *assigned*. An agent is in *idle* state when there is no activity in the worklist of the agent and the agent is not assigned to any activity. The agent is in *assigned* state when an

activity is in active state with that agent. We describe these two states of an agent with two predicates: *idle*(*Agent*) and *assigned*(*Agent, Activity, W*). The state of the agent may be changed by two events: *assign*(*Agent, Activity, W*) and *release*(*Agent, Activity, W*).

In addition to the time dependent description of the workflow state, there are also static properties of the workflow. The agent definitions, the activities for which they are qualified, the cost of each agent for each activity are static properties of the workflow and they are defined in the workflow specification. For simplicity, we assume that the cost of an agent is the amount of time that an agent requires to perform an assigned activity. In order to represent the relationship between the activities and agents we use the predicate *qualified*(*Ag, Act, Cost*) which is true when it takes *Cost* units of time for an agent *Ag* to finish the activity *Act*.

The time-dependent states for activities and agents together with the events causing the transitions between these states are summarized in Table 3 and Table 4 respectively. The time dependent states of activities and agents are initiated and terminated by events occurring in the workflow system. The third columns in the tables show these events. The axioms of the event calculus will be used in reasoning with these events and their effects. In the following we present the rules to describe how these events cause state transitions and these rules are named as axioms for initiates/terminates (**AxIT** in short) for reference purposes.

An activity becomes active in a workflow instance when its starting event is recorded in the database. An event recording the end of an activity sets up a completed state for that activity, terminating its active state. Thus we write,

initiates(*start*(*Act, Ag, W*), *active*(*Act, Ag, W*)). **(AxIT 1)**

initiates(*end*(*Act, Ag, W*), *completed*(*Act, Ag, W*)). **(AxIT 2)**

terminates(*end*(*Act, Ag, W*), *active*(*Act, Ag, W*)). **(AxIT 3)**

When an activity starts being executed by an agent, the agent is not idle any more and it is assigned to that activity until it finishes the activity. When the activity is finished, the agent is released and it is ready to execute the next activity. Thus, we write the following rules:

terminates(*assign*(*Ag, _, _*), *idle*(*Ag*)). **(AxIT 4)**

initiates(*assign*(*Ag, Act, W*), *assigned*(*Ag, Act, W*)). **(AxIT 5)**

When an agent finishes its task and it is released, it becomes idle. If the worklist of the agent is empty, the agent remains in the idle state. If there are one or more activities waiting for that agent in the agent's worklist, the agent is assigned to the next activity in its worklist. The assignment of the agent to the next activity is described in Section 5.2.2 (see axioms AxH 5 and AxH 6). Here, we present the rules that describe the effects of the event *release* on the system state.

Table 4. States of agents

State of Agent	***Meaning***	***Initiating event***
idle(Ag)	Agent Ag is idle	release(Ag, Act, W)
assigned(Act, Ag, W)	Agent Ag is carrying out the activity Act in workflow instance W	assign(Ag, Act, W)

initiates(*release*(*Ag, _, _*), *idle*(*Ag*)) . **(AxIT 6)**

terminates(*release*(*Ag, Act, W*), *assigned*(*Ag, Act, W*)) . **(AxIT 7)**

The use of the property *waiting*(*Act, Agent, W, T*) is twofold. First, it is used to represent the state of an activity. Second it is used to represent the worklists of agents. An is released when it completes an activity and the subsequent activity is enabled by the workflow manager (using the axioms AxS 1-8). The subsequent activity is inserted to the worklists of all agents qualified to do that activity. The following axiom is describing this behavior:

initiates(*release*(*Ag1,Act1,W*), *waiting*(*Act2, Ag2, W, T*)) ← **(AxIT 8)**
follows(*Act1, Act2, W, T*), *qualified*(*Ag2, Act2, _*).

The rule states that when an agent *Ag1* is released from an activity *Act1* in a workflow *W* at time *T1*, the subsequent activity *Act2* is made waiting for all qualified agents in the workflow instance *W*, with the timestamp *T*.

When an activity is assigned to an agent, the activity is no longer in waiting state. It is removed from all worklists:

terminates(*assign*(*_, Act, W*), *waiting*(*Act, _, W, _*)). **(AxIT 9)**

This rule has the effect of removing the activity from all worklists, because it is used to terminate the property *waiting*(*Act, _, W, _*) which represents the set of all agents that the activity *Act* in workflow instance *W* has been waiting.

5.2. Workflow Execution

A critical issue in workflow management is the assignment of activities to appropriate agents in order to execute the workflow. Many different scheduling and optimizing algorithms may be proposed for this purpose. In this paper, we formalize a simple agent assignment algorithm. The activity is assigned to the best agent among all available agents qualified to perform that activity. The best agent is determined by comparing the estimated costs of the candidate agents. When an agent pulls the activity from its worklist, the activity is removed from the worklists of all other agents too (see axiom AxIT 9). Choosing always the best available agent may not result in an optimized execution of the workflow, however, optimizing the execution of a workflow is out of the scope of this paper.

In this section, we first present the rules to start the execution of activities and to record the end of activities. We, then, present the rules for actually assigning tasks to agents and rules to release agents. The rules listed below, describe the generation of new events to trigger the desired functionalities. They are used to record new event occurrences in the history through the predicate *happens*. Therefore we name these rules as *axioms for happens* (**AxH** in short).

5.2.1. Rules for triggering events

The execution of an activity can start only when an agent is assigned to that activity. As soon as the agent is assigned, the starting event of the activity is generated, which is described by the following rule:

happens(*start*(*Act, Ag, W*), *T*) ← **(AxH 1)**
happens(*assign*(*Ag, Act, W*), *T*).

This rules states that when the event of assigning the agent *Ag* to activity *Act* in workflow instance *W* happens, the starting event of activity *Act* happens at the same time. The event *assign*(*Ag, Act, W*) is generated by the workflow manager as described in Section 5.2.2 (see axioms AxH 5 and 6).

When an activity is completed, the ending event of the activity is recorded and the agent that completed the activity is released. This is represented by the following rule:

happens(*release*(*Ag, Act, W*), *T*) ← **(AxH 2)**
happens(*end*(*Act, Ag, W*), *T*).

In a real workflow, the end of an activity would be sent to the workflow manager by the agent performing that activity, and the end of that activity is saved in the database. Some activities may be completed in a fixed amount of time. For some other activities, the duration may not be predicted; the end of the activity may depend on the occurrence of an external event. The application must include rules to determine the end of the activity. In our framework, in order to simulate the execution of fixed time and varying time activities we write rules AxH 3 and AxH 4. In AxH 3 we assume that the time required for a fixed duration activity is determined by the assigned agent. Thus, we write the following rule for fixed-time activities:

happens(*end*(*Act, Ag, W*), *T*) ← **(AxH 3)**
happens(*start*(*Act, Ag, W*), *Ts*), *fixed_activity*(*Act*),
qualified(*Ag, Act, Td*), *T = Ts + Td.*

That is, the agent *Ag* finishes the activity *Act* in *Td* time units after the starting event of the activity. For varying time activities, we assume that an external event (e.g. a user input) is waited to finish the activity. The end of the activity depends on the time required by the assigned agent and the time of the occurrence of the external event. The end of the activity is described as the time whichever happens later.

happens(*end*(*Act, Ag, W*), *T*) ← **(AxH 4)**
happens(*start*(*Act, Ag, W*), *Ts*), *varying_activity*(*Act*), *end_event*(*Act, ExtEvent*),
happens(*ExtEvent, Te*), *qualified*(*Ag, Act, Td*), *Tf is Ts + Td, max*([*Te, Tf*], *T*).

5.2.2. Rules for assigning agents to activities

The scheduled activities wait in the worklists of the qualified agents. An agent keeps checking its worklist when it is idle or when it is released after the completion of an activity. If worklist is not empty, the agent pulls the activity that has been waiting for the longest time from the list. The following rule describes the assignment of an agent to the longest waiting activity as soon as it is released from another activity.

happens(*assign*(*Ag, Act, W*), *T*) ← **(AxH 5)**
happens(*release*(*Ag, _, _*), *T*), *holds_at*(*waiting*(*Act, Ag, W, T1*), *T*),,
holds_at(*idle*(*Ag*),*T*), *not waiting_longer*(*Act, Ag, T1, T*),
not better_agent(*Ag, Act, T*).

If there are two or more activities waiting for the agent with the same timestamp, the conjunct *holds_at*(*idle*(*Ag*), *T*)) in the body of the rule guarantees that we assign the agent to only one of these waiting activities. This condition will be true before any assignment, but it will not hold at the time immediately after the first assignment.

The rule for *waiting_longer* checks for any other activity in the worklist of the agent that has been waiting longer than this activity. It looks up the system state at time *T* to find out which activities are waiting for this agent and compares their timestamps:

waiting_longer(*Act, Ag, T1, T*) ←
    holds_at(*waiting*(*Act2, Ag, W, T2*), *T*), *Act* ≠ *Act2, T2* < *T1.*

The check for *better_agent* is necessary in order not to assign the same task to different agents. Since one or more qualified agents may be available at the same time, we make sure that the activity is assigned to one of them (the best available one) only. The rule for *better_agent* checks if there are other less costly agents qualified for the activity. If two agents have the same cost, the first considered one is selected.

better_agent(*Ag1, Act, T*) ←
    qualified(*Ag1, Act, C1*), *qualified*(*Ag2, Act, C2*), *C2* < *C1,*
    holds_at(*idle*(*Ag2*), *T*).

As long as the worklist of an agent is empty, the agent stays in the idle state. However, when an activity is inserted into its worklist, it is assigned to the activity if there is no better agent to do that activity. An activity may be placed into the worklist of an agent at any time. As discussed in Section 5.1, the property *waiting* is initiated for an activity when the workflow manager determines that activity to be the subsequent activity (see AxIT 8). The agent checks its worklist at every time point that it is released from an activity (see AxH 5). If there is no activity in its worklist, it continues to be idle. There must be a way of triggering the agent to check its worklist when it is idle. This is achieved by the following rule, which triggers the event *assign* every time an activity is placed into an empty worklist of an idle agent:

happens(*assign*(*Ag, Act, W*), *T*) ← **(AxH 6)**
    initiates(_, *waiting*(*Act, Ag, W, T*)), *holds_at*(*waiting*(*Act, Ag, W,T*), *T*),
    holds_at(*idle*(*Ag*), *T*), *not better_agent*(*Ag, Act, T*).

Here, the condition *initiates*(_, *waiting*(*Act, Ag, W, T*)) is necessary to find the time point *T* at which the activity is placed into the worklist. The anonymous variable represents any event that may initiate the property *waiting*. As soon as such an event happens, the idle agent is assigned to the waiting activity. There may be one or more activities that have been inserted to the worklist of an agent at the same time when the agent is in idle state. The conjunct *holds_at*(*idle*(*Ag*), *T*) is used to make sure that the agent is assigned to only one of these activities.

5.3. Starting a workflow instance

The workflow manager is an interpreter to generate events that start and assign agents to activities through the event generation rules. In order to start generating the events (and

thus, start the execution of workflow instances), the manager needs to know what initiates the workflow and also the initial state of the system. In our framework there must be an external event to start the workflow. For instance, in an order processing workflow, the initial event may be the submission of an order request form by the user. This initial event must be defined in the workflow specification. In addition, all agents are in idle state at the beginning.

In order to set all agents idle initially, we define an event, called *free_agent*(*Ag*), whose effect is to initiate the idle property for all agents. This can be represented by the rule:

initiates(*free_agent*(*Ag*), *idle*(*Ag*)). (**AxIT 10**)

If we assume that the time is set to zero initially, we can set all agents idle with the following rule:

happens(*free_agent*(*Ag*), *0*) ← *agent*(*Ag*). (**AxH 7**)

The manager starts a workflow instance when an initial external event happens (e.g. submit an order). When that starting external event is recorded, the manager schedules the first activity of the workflow by inserting it into the worklists of all agents qualified to perform that activity. Once the first activity is inserted into the worklists, the event generation rules (AxH 1-6) will be activated so that it is assigned to the best qualifying agent. The workflow manager will keep scheduling the next activity for each completed activity using the execution dependency rules (AxS 1-8) and event generation rules (AxH 1-6) until the end of the workflow is reached (or until the current time). In order to start this process, we write the following rule, so that when the initial event happens, the first activity can be scheduled:

initiates(*Ev, waiting*(*Act, Ag, W, T*)) ← **(AxIT 11)**
initial_activity(*Act*), *starts*(*Ev, W*), *happens*(*Ev, T*),
setEID(*Act,W*), *qualified*(*Ag, Act, _*).

The starting event is defined with the predicate *starts*. The predicate *starts* also generates a unique workflow instance id *W*. Thus, this rule represents that when the event which starts the workflow instance *W* happens at time *T*, the first activity of the workflow starts waiting for all qualified agents. The predicate *setEID* sets the execution id of the initial activity of the workflow instance to the workflow id *W*. The workflow manager will assign the first activity to one of the agents through the rule AxH 6 in Section 5.2.

6. Implementation Issues

In this section we first discuss the computational aspects of the logical description given in this paper. We then present a case study to illustrate the capabilities of the system.

6.1. Implementation of the theory

The theory can be implemented in several different ways. One approach is to write the axioms more or less directly in Prolog. However as they stand, the general structure of the search space that would be explored by SLDNF resolution is riddled with non-terminating

loops and redundancy. Because the definition of *holds_at* includes calls to *happens* and the definition of *happens* includes calls to *holds_at*, this can cause non-terminating loops. Similarly, the definitions of *happens*, *initiates* and *follows* also include calls to *happens* that can cause non-terminating loops.

The major reason of the problem of getting infinite loops is that, in the execution of *holds_at*, after finding a relevant event, all events (past or possible future events) must be searched again in order to show that there is no other event affecting the established relation. This is because of the negation in the formulation of *holds_at*. Therefore we must restrict the search space in such a way that only the past relevant events (i.e. events which have occurred) should be searched.

We have overcome this problem by rewriting the axioms so that they are more suitable for SLDNF resolution. We rewrite the clauses so that a Prolog interpreter can proceed forwards in time from the earliest known event, maintaining a list of ongoing events. Since we know the causality relation between the events (i.e. which events will occur after which events), we can compute the entire history given the initial event(s). We proceed roughly in a bottom-up manner: we compute what events the initial events cause in the history, then compute what events these new events cause in the history, and so on.

In order to achieve this, we replace all calls to *happens* in the bodies of the rules for *holds_at*, *happens, initiates*, and *follows* with calls to a new predicate called *happened*. The *happened* predicate represents all events that are known to have happened in the history. The history of happened events is populated by using *happens* rules level by level. With these clauses, a Prolog interpreter proceeds forwards in time from the earliest event, maintaining a list of all occurred events. For example, the new version of AxH 1 is rewritten as follows:

```
happens(start(Act, Ag, W), T) ←
    happened(assign(Ag, Act, W), T).
```

Likewise, all occurrences of the predicate *happens* in the bodies of rules AxH (2-7), AxS (1-8), and AxIT (1-11) are replaced with the predicate *happened*. The new definition of *holds_at* is now given as follows:

```
holds_at(State, Time) ←
    initiates(Ev, State), happened(Ev, T1),
    T1 ≤ Time, not broken(State, T1, Time).

broken(State, T1, T2)  ←
    terminates(Ev, State), happened(Ev, T), T1 ≤ T ≤  T2.
```

Instead of searching all events, the new definition searches only the past events which are known to have occurred already (i.e. represented by the `happened` relation). These new axioms can be directly translated into a Prolog program. After all the events in the system are generated, it is possible to ask queries of the form

```
? - holds_at(State, t1).
```

to find out the state of the system at a time t_1 after the given initial event.

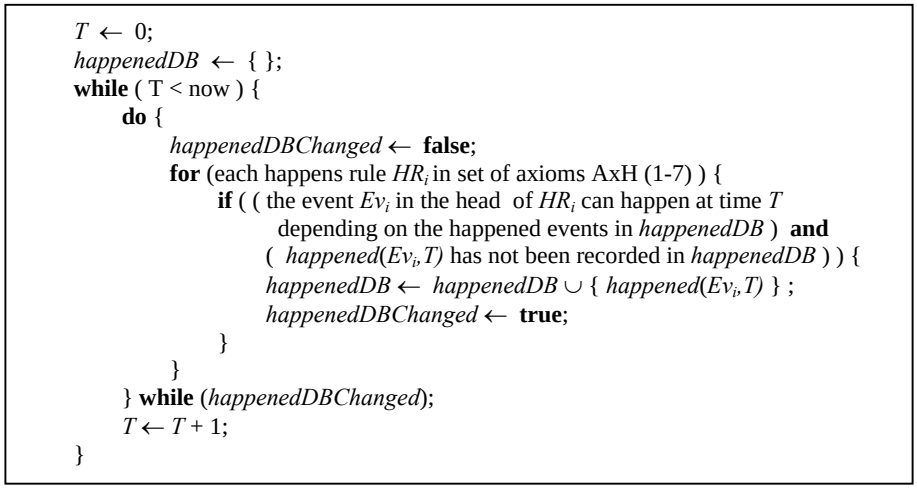

Fig. 7. Algorithm to find happened events

The algorithm in Fig. 7 explains the behavior of the Prolog interpreter to find all happened events and record them in an extensional database of history of events. All happened events are found level by level. First, we find all happened events at time 0, then at time 1, and so on. The outer loop in the algorithm quits when all possible events are generated and recorded in the history. The inner do-while loop finds all happened events at time T. The innermost for-loop checks whether each of possible events described by the axioms for happens (AxH 1-7) can happen at time T depending on the conditions induced by the already happened events in the happened database. This algorithm and all axioms presented in this paper are implemented in Prolog and tested in the simulation of some prototype workflow systems.

6.2. Case Study

We illustrate the use of the axioms presented so far with a case study. Consider an order processing system shown in Figure 8. Activity a_1 takes the order. Activity a_2 processes the order by updating the inventory. Activities a_3 and a_4 then start concurrently. Activity a_3 removes the product from the warehouse and packages the item. Activity a_4 performs the billing function. After both activities are completed, activity a_5 arranges shipping by initiating either activity a_6 or activity a_7. Finally when the delivery is successful, the database is updated to indicate that the order has been fulfilled.

In order to model and manage the execution of this workflow in our framework first the workflow graph definition must be given using the predicates shown in Table 1. Thus the example workflow is translated into the following:

```
initial_activity(act(order_collection, EID)).
sequential(act(order_collection, EID), act(order_processing, EID)).
and_split(act(order_processing, EID), [act(package,EID), act(billing,EID)]).
and_join([act(package,EID), act(billing,EID)], act(arrange_shipping,EID)).
```

```
xor_split(act(arrange_shipping,EID), [(act(by_air,EID),selection(EID,air)),
                    (act(surface_mail, EID),selection(EID,surface))] ).
xor_join([act(by_air,EID),act(surface_mail, EID)],act(archive,EID)).
final_activity(act(archive,EID)).
```

The list of qualified agents must be given with their timing constraints. In our prototype implementation, the agent information is defined as follows:

```
qualified(agent1,act(order_collection,_),1).
qualified(agent2,act(order_processing,_),2).
qualified(agent3,act(order_processing,_),5).
qualified(agent4,act(billing,_),1).
qualified(agent5,act(package,_),8).
qualified(agent6,act(arrange_shipping,_),2).
qualified(agent7,act(by_air,_),2).
qualified(agent8,act(by_surface,_),1).
qualified(agent6,act(archive,_),3).
```

In this example we assume that activities a_1, a_2, a_4, a_5 and a_8 are computer programs that execute in fixed period of time. Activities `package`, `by_air`, and `surface_mail` are varying time activities. These activities need human interference, thus their termination need some external event such as waiting for the user to enter some data. For instance, activity `package` needs the operator to input data that the packaging is finished. The actual shipment of the package (by air or surface mail) is done by a person, thus the completion of this activity must be recorded by an input and this is considered as an external event. In order to simulate the end of varying time activities, the external events that finish those activities must be given to be used by the axiom AxH 4.

```
end_event(act(package,EID), finish_packing(EID)).
end_event(act(by_air,EID), sent(EID)).
end_event(act(surface_mail,EID), sent(EID)).
```

In addition, for each external event, its occurrence time must be recorded with a `happened` clause. The workflow is initiated by an external event which is the submission of an order request form. Every time this event is entered to the system a new workflow instance is started. The following rule is used to specify the initialization of a workflow instance:

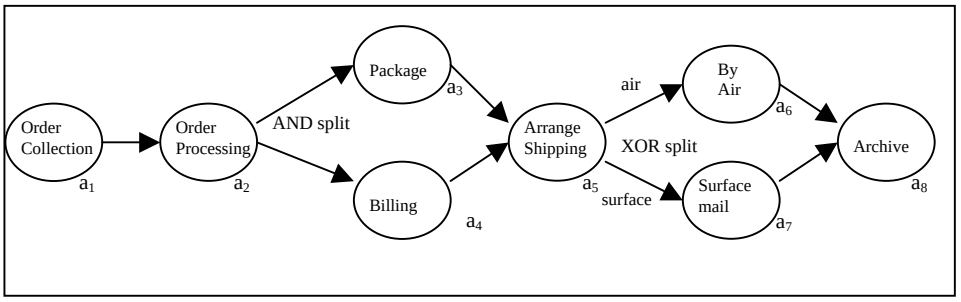

Fig. 8 Order Processing Workflow

```
starts(Ev, Wno) :-
    ext_event(Ev),
    Ev = submit(OID, CustID,  CName, Caddress, ProductID, Qty),
    Wno = OID.
```

where external event `submit` includes the information about the customer, product and order. The order id (OID) is assumed to be unique for each order, and therefore it is used as the workflow instance id (which is equivalent to the EID). The time of this submit event should also be recorded by a `happened` clause. Note that we simplified event representations to simple atomic terms. In a real application the details of events can be specified using several binary predicates [18].

The workflow specification for the given graph is now complete. In addition we have the axioms presented in this paper: AxS 1-8, AxIT 1-11, AxH 1-7. The external events to initiate the workflow instances, to end varying activities must be input to the system at various points in time. Then, the deductive framework allows us to query the system in different ways. Some possible queries are:

```
?- happened(Ev, T).                    % list all events in the history
?- holds_at(active(Act,Ag,W), now).    % list all currently active tasks
?- holds_at(idle(Ag), t).              % list idle agents at some time t
?- holds_at(waiting(Act,Ag,W,_), t). % worklists of all agents at some time t
?- holds_for(assigned(Act,Ag,W),P)     % working periods for all agents
```

Thus, given a set of predicates for a workflow graph specification, external events and qualified agents, the axioms that are presented in this paper can be used to answer queries such as finding out the system state at a specific time, or the period of time for which a certain property holds (e.g. how long an agent remains idle). By querying the history of events the actual order and occurrence times of all activities can be derived.

7. Architecture

The main concern of this paper is to present a new class of logic-based workflow systems based on the notion of a history that underlies the event calculus. Nevertheless, we describe a conceptual architecture to indicate how the logic-based workflow management system might be used at the implementation level. The proposed architecture is similar to the history-centered active database architecture of [11]. The main contribution of their architecture is to provide a logic-based integration of deductive databases and active databases. In their architecture, history serves to determine database states and it underlies the definition of event detection, condition verification and action execution.

We propose to extend the history–centered active database architecture of [11] by incorporating the workflow manager, which is responsible for scheduling the activities and assigning the agents. Fig. 9 depicts the components of our conceptual architecture for a logic-based workflow system.

The workflow state is described as a deductive database. The records of event occurrences are considered to be an extensional database, called the history. The intentional database includes the event calculus rules, workflow specification and activity execution dependency rules, and workflow execution rules. The set of known events and the set of

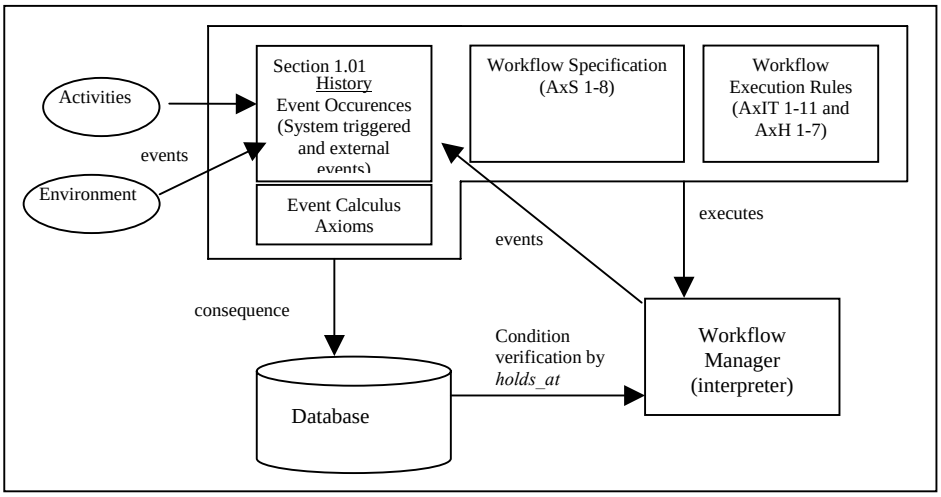

Fig. 9. Conceptual architecture of a workflow management system in the Event Calculus Framework

possible workflow states are immediately characterized in terms of the set of all logical consequences of this deductive database.

Conceptually speaking the database states need not be independently stored, since they follow logically from the history. The history only needs appending event occurrences to, in order to record that some event has happened in the modeled reality.

A typical cycle in this architecture can be described as follows. The environment notifies the system the start of a new workflow instance by appending an external event that initiates the workflow. Since the set of known events (i.e. history) now includes at least one event, the interpreter reacts to this change by scheduling the first activity in the workflow. The first activity is placed to the worklists of qualified agent(s). The agent assignment rules will be used to assign the best agent to the activity. When the end of the first activity is recorded in the history, the interpreter uses the execution dependency rules and agent assignment rules to start the next activity. Meanwhile, the environment may record the beginning of another workflow instance, or the executed activities may insert new (external) events to the history. The interpreter proceeds to coordinate the activities by reacting these new happenings until a saturation state is reached in which all known events have been derived.

8. Related Work and Discussion

A considerable amount of work has been done on formalizing workflow systems across the fields of computer supported cooperative work and advanced transaction models. This section gives a brief overview of formal specification methods used in products and research prototypes of workflow systems and compares our framework with other proposals. We also summarize a related research area, namely web service composition, and discuss the application of our proposed framework in the semantic web.

8.1. Net-based methods

Petri-nets and state and activity charts are net-based methods, which have a formal foundation. When a graphical visualization of workflow specifications is the top priority, state and activity charts and Petri nets are good choices. State and activity charts have originally been developed for software engineering applications, especially for specifying reactive systems, but they have been also used as a formal tool for workflow specifications. In [23] it is shown that state and activity charts can be used for the specification of workflows, verification of workflow properties and the distributed execution of workflows. Although most execution dependencies can be formally specified in this framework, iterative execution of activities is not modeled. Although concurrent activities within a workflow can be executed through a partitioning algorithm, it is not clear how concurrent instances of the same workflow are executed.

Petri nets are general purpose process specification formalisms. Petri net variants are widely used as a workflow modeling technique [31,32,33]. A workflow process specified in terms of a Petri net has a clear and precise definition, because the formal semantics of the classical Petri net and its enhancements (color, time, hierarchy). Petri nets have also the advantage of the availability of many analysis techniques. Our work differs from the Petri net approach radically since Petri nets are graphical and sate-based whereas our approach is declarative and event-based. Our aim is to show the use of the event calculus as a workflow modeling specification and execution tool in a logic programming framework.

8.2. Logic-based methods

These methods attempt to establish a formal specification model with a well-defined semantics to be used in the analysis and reasoning about workflows. In [9,24], Concurrent Transaction Logic (CTR) is used as the language for specifying, analyzing and scheduling of workflows. In this framework, both local and global properties can be represented as CTR formulas and reasoning can be done with the use of the proof theory and the semantics of this logic. Like in all logic programming systems the proof theory of CTR is also a run-time environment for executing workflows. Within their framework, it is possible to represent control flow graphs with transition conditions, triggers, concurrent execution of activities and a set of temporal constraints. The proposed system does not cover the specification of loops and iteration of activities and they do not address the problem of agent assignment and concurrent instances of workflows. They are mainly concerned with the development of an algorithm for consistency checking of workflows and for their property verification. The algorithm compiles global constraints on workflow execution into the control flow graph. This compile technique also helps optimize the run-time scheduling of workflow events. In our framework, we do not study specification and verification of global (and temporal) constraints on workflow activities. Instead we concentrate on the representation of different routings of activities (including loops and iteration), agent assignment and concurrent workflow instances within a logic programming framework. Instead of proposing a new logic and its proof theory, we use the well-known SLD-NF procedure, which makes our framework simpler.

Another logic-based approach is presented in [2]. They propose treating workflows as a collection of cooperative agents and use recent results on reasoning about actions to

formalize correctness of a workflow. They also discuss the automatic verification and construction of reactive condition-action rules that specify the workflow control. The workflow specification is defined as a conditional program where each transition is described as a sequential transition with condition. In other words, each possible path execution depends on the explicit specification of conjunction of conditions. (In [17], a more direct way of the mapping of different routings of activities to action description language C is shown). The main purpose is to use this framework as a formal tool for testing the correctness of a workflow. It can also be used to explore the behavior of "what-if" scenarios during the construction of ad-hoc workflows. Our framework can be used for the same purposes. In addition, we can represent more complex workflows, concurrent processes and several candidate agents to perform a single task. Testing different versions of the workflow specification with different number of agents can be more easily done to explore what-if scenarios in our framework.

8.3. Algebraic methods

Process algebras have been considered but not widely known in the field of workflow management. In [10] the specification language, which is based on process algebra, is extended towards workflow management. The main drawback of process algebras is that they are often not intuitive and hard to understand.

In [26], an event algebra is presented for specifying and scheduling workflows. In this event algebra activity execution dependencies can be declaratively expressed and these dependencies are symbolically processed to determine which events occur and when. This algebra can model most features of control flow graphs, but it is not sufficient to express transition conditions attached to edges.

8.4. Event-Condition-Action Rules

Event-Condition-Action Rules, shortly termed ECA rules, are used in active database systems and have been used in defining workflows [5]. ECA rules, are used to specify the control flow between activities. However, this specification is not as general as control flow graphs. They are not sufficiently expressive to represent all possible routings among activities. The graphical visualization of ECA rules is not easy either. Large sets of ECA rules are hard to handle and verify. Also, some workflow properties such as loops and sub-workflows cannot be represented in this approach.

In [11], a logic-based approach is presented for the integration of deductive and active databases. Although this work does not consider workflows at all, it is still very closely related to our framework, because event calculus is used to define event occurrences, database states and actions on these. The formalization of an active and deductive database is built on the idea of a history-centered architecture. In this architecture, the component data store is a deductive database, where the extensional database keeps the history (i.e. set of event occurrences) and the semantics of event detection, condition verification and action execution are defined in terms of querying and updating deductive database. Although this work and our framework seem to have a lot in common, the objectives are different. In [11], the main objective is to present a logic-based approach to the formalization of a large set of syntactic and semantic aspects of active databases. There is

no notion of defining a workflow or coordination of activities to accomplish a certain task. Therefore, issues like activity scheduling, agent assignment, concurrent execution of activities or workflow instances are not addressed. The only similarity is our architecture which is actually an extension of the architecture proposed in [11].

8.5. Web Services

A currently much related research area is web services and their composition. The literature on web services and the semantic web is abundant [29]. Therefore the need for a more rigorous formal foundation is widely discussed.

Web services are platform and language independent software components that can be invoked on the web to fulfill some goals. The composition problem for web services is to figure out how a set of given services could be invoked to complete a given task. A number of flow languages have emerged for web services, such as WSFL [20], XLANG [30], BPEL4WS [8]. The composition of the flow is still manually obtained [27]. It is argued that automatic web service composition can be seen as a planning problem where given a domain description of the available services and user goals, the planner generates a flow (plan) [21,28]. In [21] the Golog logic programming framework is used to generate service compositions (plans) for goals based on plan templates. Golog builds on top of the situation calculus by providing a set of extralogical constructs for assembling primitive actions into complex actions. In [28], a heuristic search planner is used to solve the planning problem created from the description of the available web services and the requirement of the composite service. The resulting plan can be generated in any web service composition language and executed by the corresponding flow execution engine.

Event calculus has been used in planning [25]. Planning in the event calculus, is an abductive reasoning process through resolution theorem prover. Given the domain knowledge (i.e. a set of *initiates* and *terminates* axioms), event calculus axioms and the conjunction of *holds_at*(*X*, *t*) formulae, where *t* is the time point, we are interested in, the event calculus planner generates the set of events that lead to the specified state at time *t*. We argue that our current framework can be adapted to web service composition problem by viewing the flow of activities as a plan generation problem. The activities will represent web services and events which start activities will correspond to messages received to invoke web services. Flow constructs that we presented in this paper will be used to specify the flow of service composition. In order to model the fact that there might be several web services to perform the same task, we can extend the concept of agents to represent the candidate web services. The current framework must be extended with a service discovery mechanism to select the best agent depending on the preconditions of the desired goal. This needs to be further studied.

9. Conclusions and Future Directions

This paper demonstrates the use of the event calculus to describe the specification and execution of activities in a workflow. The main axioms of the event calculus are integrated with a set of activity execution dependency rules and a set of agent assignment rules for the formalization of workflow systems. It is shown that major types of activity routings in a workflow (namely sequential, concurrent, conditional and iterative) can be expressed in a

declarative way. It is also illustrated that agent assignments and concurrent workflow instances can be modeled within the framework of the event calculus. In addition, a conceptual architecture of a workflow management system is presented as a basis for a more realistic implementation of this logic-based approach. For a quick simulation of a workflow, the user needs merely to specify the atomic formulas to describe the control flow graph and if there are any, the external events and their possible effects on the underlying database. The rest of the workflow management is done by the rules presented in this paper.

The proposed logic-based approach can be used as a quick tool in prototyping applications and/or simulations of workflows. Due to its additional temporal dimension, it provides facilities for querying the history of all activities, thus providing opportunities to analyze the execution of the workflows. It can be used as an easy tool to simulate and verify the execution of a prototype workflow system. The workflow might be executed with different number of agents and assignments. The behavior of the workflow can be analyzed by querying the history of events and the snapshots of the workflow state at different times.

In this paper we did not consider the workflows where some activities do not terminate successfully. Some of the activities can abort and therefore they need to be compensated or some kind of exception handling mechanism must be applied. As a future work, the set of execution dependency rules can be extended to cover such control flows. These extensions do not require substantial changes to the proposed architecture. Broadly speaking, what needs to be done is to define additional scheduling rules to the set of axioms AxS, so that when an activity does not end, the execution is diverted to another route of activities, which will be used either to abort the workflow or compensate the failed activity.

Other extensions are possible to the implementation of the system to ease its use. For instance a graphical tool might be integrated to the architecture to provide the user with the facility of drawing the control-flow graph of the workflow. Then another application might map this graph into a set of atomic formulas presented in this paper automatically.

The paper presents a simple agent assignment algorithm where each activity is assigned to the best (i.e. the least costly) available agent by the scheduler. This simplistic view of agent assignment might be changed to implement more sophisticated algorithms in order to test the behavior of a certain workflow so that the execution can be optimized. This and other optimization problems for the execution of workflows are open problems. Likewise, addition of global constraints and reasoning with them using the axioms of event calculus is another interesting research topic.

Workflow has moved inexorably towards Web services in the last two years. Web services provided by various organizations can be inter-connected in order to implement business collaborations, leading to composite web services. The composition of the web services is still manually obtained. The semantic web community draws on AI planning for automatically composing services [21]. The notion of event calculus can also be viewed as an opportunity to take advantage of the latest developments in web services. Modern workflow engines will be asynchronous, with the process enactment driven by the arbitrary arrival of messages from different sources. Event calculus could be a way to specify and reason about web composition where the actual process model is not known.

Appendix

The appendix presents the definitions of the auxiliary predicates that have been used in this paper. The definitions are presented as Prolog-style rules.

The 3-argument predicate *findActEndTimePairs* (used in AxS 3) finds the ending times of all predecessor activities in an AND-join. The third argument is a list of (activity, ending time) pairs if all the incoming activities have completed their executions.

findActEndTimePairs(*ActList, W, ActTimePairs*) ←
    findall((*Act, EndTime*),
        (*member*(*Act, ActList*), *happens*(*end*(*Act, _, W*), *EndTime*)),
        ActTimePairs),
    length(*ActList, ActListLen*), *length*(*ActTimePairs, ActTimePairsLen*),
    ActListLen = ActTimePairsLen.

The 3-argument predicate *actWithMaxEndTime* (used in AxS 3) simply calls its 4-argument definition in order to find the maximum ending time in the list of (activity, ending time) pairs. The subsequent activity in an AND-join can start execution only if all incoming activities are completed. Therefore the maximum ending time is found to determine the starting time of the subsequent activity.

actWithMaxEndTime([*FirstPair* | *ActEndTimePairs*], *Act, EndTime*) ←
    actWithMaxEndTime(*ActEndTimePairs, FirstPair, Act, EndTime*).

actWithMaxEndTime([], (*Act, EndTime*), *Act, EndTime*).
actWithMaxEndTime([*CurrPair* | *Rest*], *CurrMax, Act, EndTime*) ←
    CurrPair = (*Act1, T1*), *CurrMax* = (*Act2, T2*), *T1* > *T2,*
    actWithMaxEndTime(*Rest, CurrPair, Act, EndTime*).
actWithMaxEndTime([*CurrPair* | *Rest*], *CurrMax, Act, EndTime*) ←
    CurrPair = (*Act1, T1*), *CurrMax* = (*Act2, T2*), *T1* ≤ *T2,*
    actWithMaxEndTime(*Rest, CurrMax, Act, EndTime*).

The 3-argument predicate *findOneActEndTimePair* (used in AxS 5) finds the predecessor activity that has been completed in an XOR-join, with its ending time. It simply checks each activity in the XOR-join with the predicate *member* to see whether it has been finished.

findOneActEndTimePair(*ActList, W, Act, EndTime*) ←
    member(*Act, ActList*), *happens*(*end*(*Act, _, W*), *EndTime*).

The predicate *setIterationNo* (see AxS 6) is used to generate the execution id of an activity in a block in the next iteration. The functional term representing the execution id of an activity includes the workflow instance number, block name and the iteration number. Therefore this predicate simply changes the third argument of the functional term representing the execution id. The user should note that *IterationNo* is an unbound variable when this predicate is invoked.

setIterationNo(*Act, N*) ←
    Act = *act*(*_, b*(*_, _, IterationNo*)), *IterationNo* = *N.*

Similary, the predicate *getIterationNo* (see AxS 6) is used to extract the iteration number in the execution id of a given activity.

getIterationNo(*Act, IterationNo*) ←
    Act = *act*(_, *b*(_, _ , *IterationNo*)).

References

[1] G. Alonso, C. Mohan, WFMS: The next generation of distributed processing tools, in: S. Jajodia and L. Kerschberg (Eds.) Advanced Transaction Models and Architectures, Kluwer Academic Publishers, 1997, pp. 35-62.

[2] C. Baral, J. Lobo, G. Trajcevski, Formalizing and reasoning about the requirements specifications of workflow systems, International Journal of Intelligent Information Systems 10(4) (2001) 483-507.

[3] D. Barbara, S. Mehrotra, M. Rusinkiewicz, INCAs: Managing dynamic workflows in distributed environments, Journal of Database Management 7(1) (1996) 5-15.

[4] C. Bettini, X. Wang, S. Jajodia, Temporal reasoning in workflow systems, Distributed and Parallel Databases 11(3) (2002) 269-306.

[5] C. Bussler, S. Jablonski, Implementing agent coordination for workflow management systems using active database systems, in: Proceedings of Fourth International Workshop on Research Issues in Data Engineering, Houston, 1994, pp. 53-59.

[6] I. Cervesato, M. Franceschet, A. Montanari, A guided tour through some extensions of the event calculus, Computational Intelligence 16(2) (2000) 307-347.

[7] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web services description language (WSDL) 1.1., Available from <http://www.w3.org/TR/wsdl>.

[8] Business process execution language for web services version 1.1, Available from <http://www-128.ibm.com/developerworks/library/specification/ws-bpel>.

[9] H. Davulcu, M. Kifer, C.R. Ramakrishnan, I.V. Ramakrishnan, Logic based modeling and analysis of workflows, in: Proceedings of ACM Symposium on Principles of Database Systems, Seattle, Washington, ACM Press, 1998, pp. 25-33.

[10] S.J. Even, F.J. Faase, R.A. de By, Language features for cooperation in an object-oriented database environment, International Journal of Cooperative Information Systems 5(4) (1996) 469-500.

[11] A. A. Fernandez, M. H. Williams, N.W. Paton, A logic-based integration of active and deductive databases, New Generation Computing 15 (1997) 205-244.

[12] A. Geppert, M. Kradolfer, D. Tombros, Realization of cooperative agents using an active object-oriented database system, in: Proceedings of the Second International Workshop on Rules in Database Systems (RIDS), Athens, Greece, 1995, pp. 327–341.

[13] F. N. Kesim, M. Sergot, A logic programming framework for modelling temporal objects, IEEE Transactions on Knowledge and Data Engineering 8(5) (1996) 724-741.

[14] F. N. Kesim, M. Sergot, Implementing an object-oriented deductive database using temporal reasoning, Journal of Database Management 7(4) (1996) 21-34.

[15] N. Kesim-Cicekli, A temporal reasoning approach to model workflow activities, in: R.Y. Pinter, S. Tsur (Eds.), Proceedings of NGITS'99, LNCS, vol. 1649, Springer-Verlag, Berlin, 1999, pp. 256-266.

[16] N. Kesim-Cicekli, Y. Yildirim, Formalizing workflows using the event calculus, in: M. Ibrahim, J. Kung, N. Revell (Eds.), The 11th International Workshop on Database and Expert Systems Applications (DEXA'00), LNCS, vol. 1873, Springer-Verlag, Berlin, 2000, pp. 222-231.

[17] P. Koksal, N.K. Cicekli, H. Toroslu, Specification of workflow processes using the action description language C, in: AAAI Spring 2001 Symposium Series: Answer Set Programming, Palo Alto, California, 2001, pp. 103-109.
[18] R.A. Kowalski, M. J. Sergot, A logic-based calculus of events, New Generation Computing 4 (1986) 67-95.
[19] R.A. Kowalski, Database updates in the event calculus, Journal of Logic Programming 12(1-2) (1992) 121-146.
[20] F. Leymann, Web services flow language (WSFL 1.0), Available from <http://www4.ibm.com/software/solutions/webservices/ pdf/WSFL.pdf>.
[21] S. McIlraith, T.C. Son, Adapting Golog for composition of semantic web services, in: D. Fensel, F. Giunchiglia, D. McGuinness, M.-A. Williams (Eds.), Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02), 2002, pp. 482-496.
[22] J. A. Miller, D. Palaniswami, A. P. Sheth, K. Kochut, H. Singh, WebWork: METEOR 2 's web-based workflow management system, Journal of Intelligent Information Systems 10(2) (1998) 185-215.
[23] P. Muth, D. Wodtke, J. Weissenfels, G. Weikum, A. K. Dittrich, Enterprise-wide workflow management based on state and activity charts, in: A. Dogac, L. Kalinichenko, T. Özsu, A. Sheth (Eds.), NATO ASI Series: Workflow Management Systems and Interoperability, Springer Verlag, 1998, pp. 281-303.
[24] P. Senkul, M. Kifer, I. H. Toroslu, A logical framework for scheduling workflows under resource allocation constraints, in: Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002), Hong Kong, China, 2002, pp. 694-705.
[25] M.P. Shanahan, An abductive event calculus planner, Journal of Logic Programming 44 (2000) 207–239.
[26] M.P. Singh, G. Meredith, C. Tomlinson, P.C. Attie, An event algebra for specifying and scheduling workflows, in: Proceedings of the Fourth International Conference on Database Systems on Advanced Applications (DASFAA'95), Singapore, 1995, pp. 53-60.
[27] B. Srivastava, J. Koehler, Web service composition-current solutions and open problems, in: Proceedings of ICAPS 2003 Workshop on Planning for Web Services, 2003, pp. 28-35.
[28] B. Srivastava, Automatic web services composition using planning, in: Proceedings of International Conference on Knowledge Based Computer Systems (KBCS-2002), Navi Mumbai, India, 2002, pp. 467-477.
[29] S. Staab, Web services: been there, done that?, IEEE Intelligent Systems 18(1) (2003) 72-85.
[30] S. Thatte, XLANG: Web services for business process design, Available from <http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm>.
[31] W.M.P. van der Aalst, The application of petri nets to workflow management, The Journal of Circuits, Systems and Computers 8(1) (1998) 21-66.
[32] W.M.P. van der Aalst, K.M. van Hee, Workflow management: models, methods, and systems, MIT press, Cambridge, MA, 2002.
[33] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski,. A.P. Barros, Workflow patterns, Distributed and Parallel Databases 14(1) (2003) 5-51.
[34] H. Wachter, A. Reuter, The ConTract model, in: A.K. Elmagarmid (Ed.), Database Transaction Models for Advanced Applications, Morgan Kaufmann, 1992, pp. 220-263.
[35] D. Hollingsworth, Workflow management coalition the workflow reference model, Available from <http://www.wfmc.org/standards/docs/tc003v11.pdf>.