Formalizing Workflows Using the Event Calculus

Nihan Kesim Cicekli, Yakup Yildirim

Department of Computer Engineering,
METU, Ankara, Turkiye
{nihan,yakup}@ceng.metu.edu.tr

Abstract. The event calculus is a logic programming formalism for representing events and their effects especially in database applications. This paper presents the use of the event calculus for specifying and simulating workflows. The proposed framework maintains a representation of the dynamic world being modeled on the basis of user supplied axioms about preconditions and effects of events and the initial state of the world. The net effect is that a workflow specification can be made at a higher level of abstraction. Within this framework it is possible to model sequential and concurrent activities with synchronization when necessary. It is also possible to model agent assignment and concurrent workflow instances. A logic programming approach to the computational problem is adopted.

1 Introduction

A workflow is a collection of coordinated activities designed to accomplish a well-defined complex process such as a business process in a large enterprise, health-care information systems, or a student registration system. An activity in a workflow may be executed by different processing entities like 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 [e.g. 1,3,13,14,19] but it has been realized that a formal specification model is required for the analysis and reasoning about the workflows. The most common frameworks for specifying workflows are control flow graphs [10], event-condition-action rules [2,8] and temporal constraints[17,18]. There is also a logic-based formalism which proposes a concurrent transaction logic for specifying, analyzing and scheduling of workflows [7].

In this paper we propose a framework for specifying and executing workflows based on the Event Calculus [11]. The Event Calculus provides a framework for temporal reasoning by using the first-order predicate logic. We argue that the specification of workflow processes can be enhanced by integrating them with temporal databases and/or with temporal logic programming systems. Addition of a temporal dimension to workflows can enhance their facilities in different ways. It will provide mechanisms for storing and querying the history of all processes. This may serve the need for querying some piece of information in the process history. Or it

may serve the need for mining the history of the workflow to analyze and assess the efficiency, accuracy and the timeliness of the processes.

In [4] we proposed a formulation of the event calculus to describe simple workflow specifications where one activity follows another in a sequential manner. In this paper we investigate the ways in which the event calculus can be used as a basis for more complicated workflow definitions where concurrent activities, agents and concurrent workflow instances can also be modeled. We show how an extended version of the event calculus can be used to model a workflow enhanced with temporal reasoning. We also show that it will be possible to ask queries like which activities follow which ones, or when they did/will happen, and also the state of the system can be derived at any time in the past or future.

The rest of the paper is organized as follows. Section 2 reviews the workflow process definition by the Workflow Management Coalition. Section 3 summarizes the basics of the event calculus. The modeling of workflow processes using the event calculus is described in Section 4. The proposed framework is extended to include workflow manager, agents and concurrent workflows in Section 5. The computational issues are discussed in Section 6. We conclude the paper by summarizing the features of the proposed system in Section 7.

2 An Overview of Workflow Management

Workflow Management Coalition (WfMC) is a grouping of companies who have joined together to define the standards to enable different workflow management products to work together. WfMC defines a 'reference model' which describes the major components and interfaces within a workflow architecture [20]. A workflow involves several activities, each performed by an agent – human or automated. The main activity which corresponds to the 'workflow engine' in the terminology of the WfMC can be considered to be performed by a manager agent whose job is to maintain the control flow information, monitor other agents, recover from failures, answer queries about the status of the workflow, etc.

In a workflow, activities are related to one another via flow control conditions (transition information). According to this reference model we identify four 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. To model parallel routing, two building blocks are identified: (a) AND-split and (b) AND-join. The 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. Here no synchronization is required.

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

In the rest of the paper, we discuss how these features of a workflow management system can be modeled in the framework of the event calculus. However, we first give a brief summary of the event calculus.

3 Event Calculus

The event calculus was introduced by Kowalski and Sergot as a logic programming formalism for representing events and their effects, especially in database applications [11]. A number of event calculus dialects have been presented since this original paper[5,6,9,12,15,16]. The one described here is based on a later simplified version presented in [12].

The event calculus is a logical framework in which it is possible to infer what is true when, given a set of events at certain time points and their effects. The 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 which hold true in the new state of the world being modeled, and infer the termination of properties which no longer hold true from the previous state.

The main axioms used by the event calculus to infer that a property holds true at a time are as follows:

$$
\begin{aligned}
&holds\_at(P, T) \leftarrow \\
&\qquad happens(E, T1)\ ,\ T1 < T\ , \\
&\qquad initiates(E, P), \\
&\qquad not\ interrupted(P, T1, T).
\end{aligned}
$$

$$
\begin{aligned}
&interrupted(P, T1, T2) \leftarrow \\
&\qquad happens(E', T'), \\
&\qquad terminates(E', P), \\
&\qquad T1 < T',\ T' <= T2.
\end{aligned}
$$

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 time points are ordered by the usual comparative operators. The formula *initiates(E, P)* represents that the event *E* initiates a period of time during which the property *P* holds, and *terminates(E, P)* represents that the event *E* terminates any ongoing period during which property *P* holds. The *not* operator is interpreted as negation-as-failure. The use of negation-as-failure gives a form of default persistence. The formula *interrupted(P, T1, T2)* represents that the property *P* ceases to hold at some time between *T1* and *T2* due to an event which terminates it.

The problem domain is captured by a set of *initiates* and *terminates* clauses. For example in the blocks world, blocks and their places can be described by the following clauses. The term *on(X, Y)* names the property that block *X* is on top of block *Y* or at location *Y*; and the term *clear(X)* names the property that block or location *X* has nothing on top of it. The term *move(X, Y)* names the event of moving *X* onto block or location *Y*.

$$initiates(move(X, Y), on(X, Y)).$$
$$initiates(move(X, Y), clear(Z)) \leftarrow$$
$$happens(move(X, Y), T),$$
$$holds\_at(on(X, Z), T), Z \neq Y.$$

A particular course of events to represent that a block *a* was moved to location *x* and then to location *y* can be written as:

$$happens(move(a, x), t_1).$$
$$happens(move(a, y), t_2).$$

where t_1 is less than t_2.

These axioms can be used deductively to predict the locations of blocks at different times by querying the system with the *holds_at* predicate. For instance with the following, we can query the position of the block *a* at time t_2.

?-holds_at(on(a, X), t_2).

The event calculus can also compute the periods of time for which a property holds [11,12]. For instance we can query how long the block *a* stayed on block *x* by the query:

?-holds_for(on(a, x), P).

The result will be the period P = $(t_1, t_2]$. We omit the axioms for *holds_for* in this paper.

We want to use the event calculus in the specification of a workflow process. A workflow process 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. In the event calculus framework, events will denote the start and end time points of activities. The state of the workflow will be described by the properties. In other words events will specify the control flow and the effects of the events are used to describe the data flow within the workflow.

4 Specification of Workflow Processes in the Event Calculus

This section presents the event calculus as a first-order formalism for the specification of workflow processes. Once the event occurrences till time *t* are known, the state of the system can be computed at any point of time until *t*. Thus modeling can be regarded as the computation of event occurrences.

Each activity is initiated by an event and its termination starts one or more activities. We model the starting time and ending time of activities by events. In other words, for each activity there is an event which starts that activity and there is an event which finishes that activity. Thus each activity ***A*** has a starting event ***start(A)*** and a last event ***end(A)***. Between these two events there may be several other sub-events defined for the activity.

4.1 Sequential Activities

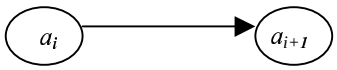

Fig. 1. Activity a_{i+1} starts when a_i finishes.

Figure 1 shows a graphical representation of sequential routing of activities. Circles represent an activity. The activity a_{i+1} follows a_i in a sequential manner. When the activity a_i finishes, the next activity a_{i+1} starts. This can be formulated in the event calculus as follows:

$$happens(start(a_{i+1}), T) \leftarrow happens(end(a_i), T). \quad (1)$$

When the last event of the activity a_i happens (e.g. commit) the starting event of the next activity is triggered. In this rule when the event a_i commits the event a_{i+1} starts immediately (at the same time point T). This can be modified if necessary by delaying the start time of the next activity by a specific time period. For instance, if a_{i+1} starts after x time units we can write:

$$happens(start(a_{i+1}), T2) \leftarrow happens(end(a_i), T1), T2 \text{ is } T1 + x. \quad (2)$$

An activity begins executing when its start event occurs. It may have a predefined duration or its execution time period may depend on some conditions or occurrences of other sub-events. We use the term sub-event to refer to the events occurring within the activity. Thus, the last event of an activity can happen either after some predefined time or after some condition is met. This and some other sequencing among activities can be described in the event calculus as follows:

$$happens(end(a), T) \leftarrow happens(start(a), T1), T = T1 + y. \quad (3)$$

$$happens(end(a), T) \leftarrow happens(a_{sub}, T). \quad (4)$$

$$initiates(start(A), active(A)). \quad (5)$$

$$terminates(end(A), active(A)). \quad (6)$$

In rule (3) y is a constant used to denote the predefined time period between events *start(a)* and *end(a)*. This value can be given statically at the time of the specification of the workflow or it may be determined by the current state of the workflow. Or as in rule (4), the end of an activity may be determined by the occurrence of a sub-event (a_{sub}) within the activity. Rules (5) and (6) show how the effects of the starting events can be represented. For instance the rule (5) states that, if the starting event of an activity happens, it initiates the time period for which that activity is active. The property *active(a)* holds between the happening times of the events *start(A)* and *end(A)*.

4. 2 Concurrent Activities

In a workflow, some activities are executed concurrently. Here we describe *AND*-split and *AND*-join transitions between activities. Figure 2.a illustrates AND-split. When the activity a_i finishes, activities $a_1, a_2, \ldots a_n$ start and they execute concurrently. Figure 2.b illustrates AND-join. Here the activity a_j starts when all the preceding activities finish.

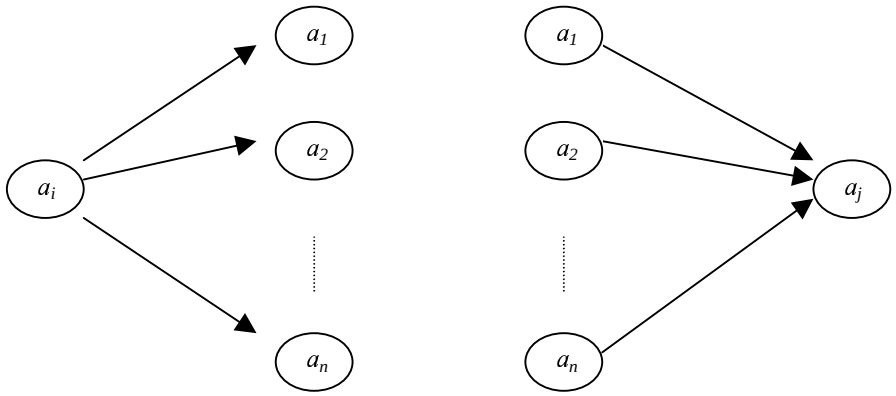

Fig. 2. (a) AND-split (b) AND-join

In the event calculus we represent AND-split with a sequence of rules as the following:

$$happens(start(a_1), T) \leftarrow happens(end(a_i), T). \quad (7)$$
$$happens(start(a_2), T) \leftarrow happens(end(a_i), T).$$
$$\vdots$$
$$happens(start(a_n), T) \leftarrow happens(end(a_i), T).$$

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

$$happens(start(a_j), T) \leftarrow happens(end(a_1), T_1), \quad (8)$$
$$happens(end(a_2), T_2), \ldots,$$
$$happens(end(a_n), T_n),$$
$$T = max(T_1, T_2, \ldots, T_n).$$

Activity a_j waits for the completion of all activities $a_1 .. a_n$. The last conjunct in the rule (8) is to start the activity at the time of the last ending activity among activities $a_1 .. a_n$.

4.3 Conditional Activities

In a workflow the execution of some activities may depend on certain conditions. If the conditions are satisfied those activities are executed; otherwise they are not

executed. This kind of execution specification can be represented in a similar fashion as in Fig. 2, except that the arrows are labeled with conditions. (Fig. 3.a)

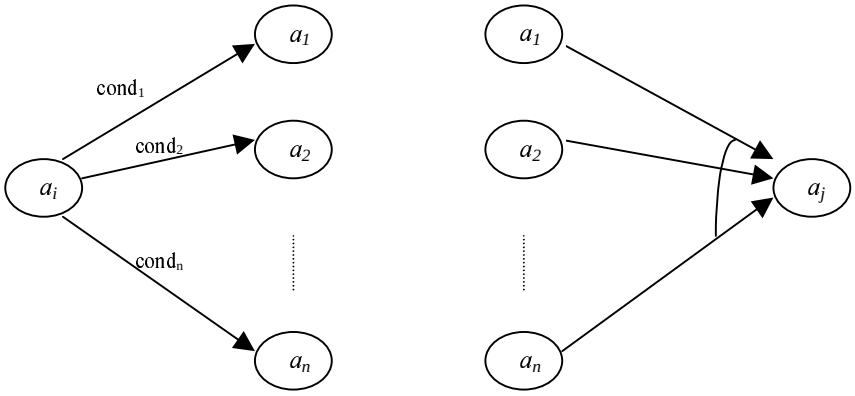

Fig. 3. a) XOR-split b)XOR-join

Such transitions can be used to describe XOR-splits. In these transitions only one of the alternative activities is executed depending on the evaluated condition. The important point here is that the conditions must be exclusive. In other words only one of the conditions should be true at the time of the decision in order to guarantee that only one execution path is chosen. We can represent the conditional activities in the event calculus as follows:

$$happens(start(a_1), T) \leftarrow happens(end(a_i), T), holds\_at(condition_1, T). \quad (9)$$
$$happens(start(a_2), T) \leftarrow happens(end(a_i), T), holds\_at(condition_2, T).$$
$$\vdots$$
$$happens(start(a_n), T) \leftarrow happens(end(a_i), T), holds\_at(condition_n, T).$$

When the activity a_i ends, the activity a_1 or a_2 …or a_n starts according to the condition satisfied at that time. The conditions may be a state check (i.e. a *holds_at* predicate) or another predicate *happens* checking the occurrence of another event. Only one of the conditions must become true at the time of the execution. If none of the conditions is satisfied at the end of the activity a_i, then the following activity cannot start. In order to prevent this problem an additional path can be provided as an "otherwise" option. This can be represented in the event calculus as follows:

$$happens(start(a_{n+1}), T) \leftarrow happens(end(a_i), T), not\ happens(start(a_1), T), not\ happens\ (start(a_2), T), \ldots, not\ happens(start(a_n), T).$$

In XOR-join (Fig.3b) if any one of the incoming activities is finished, the activity at the join can start executing. Here there is no need for the synchronization of the incoming activities. The completion of one of the incoming activities is sufficient to start the joined activity. The description of XOR-join in the event calculus needs some care. It may be seen as simply listing *n* rules in the form:

$happens(start(a_j), T) \leftarrow happens(end(a_k) T).$

for each $k = 1, \ldots, n$. However this formulation will not yield the desired execution, because the follow-up activity a_j will be started every time an incoming activity is finished (i.e. *n* times). We have to ensure that a_j is started only once. In order to achieve this we represent the XOR-join by the following rules:

$happens(start(a_j), T) \leftarrow$ (10)
$happens(end(a_1) T), endsameorafter(a_2,T), \ldots, endsameorafter(a_n, T).$

$happens(start(a_j), T) \leftarrow$ (11)
$happens(end(a_2) T), endafter(a_1,T), endsameorafter(a_3,T), \ldots,$
$endsameorafter(a_n, T).$

⋮

$happens(start(a_j), T) \leftarrow$ (12)
$happens(end(a_n) T), endafter(a_1,T), endafter(a_2,T), \ldots,$
$endafter(a_{n-1},T).$

Rule (10) represents that the end of the activity a_1 starts a_j only if the activities $a_2 \ldots a_n$ end at the same time as a_1 or after. Rule (11) states that the end of the activity a_2 starts a_j only if the activities listed before a_2 (i.e. a_1) end after a_2 and the activities listed after a_2 (i.e. $a_3 \ldots a_n$) end at the same time or after a_2. Finally, rule (12) represents that the end of the activity a_n starts a_j if the activities listed before a_n end after a_n. The predicates *endsameorafter(A, T)* checks if the activity *A* ends at the time instant *T* or after. Similarly, the predicate *endafter(A, T)* checks if the activity *A* ends strictly after time *T*. These predicates are defined as follows:

$endsameorafter(A, T) \leftarrow happens(end(A), T1), T < T1.$
$endsameorafter(A, T) \leftarrow not\ happens(end(A), \_).$

$endafter(A, T) \leftarrow happens(end(A), T1), T < T1.$
$endafter(A, T) \leftarrow not\ happens(end(A), \_).$

The second clauses are used to handle the case that the end of the activity has not happened at all.

4.4 Iteration

Sometimes we may need to execute a group of activities one or more times. We start with an activity, execute others and after the last activity of the group we return to the first activity again. This loop may be executed for a certain number of times.

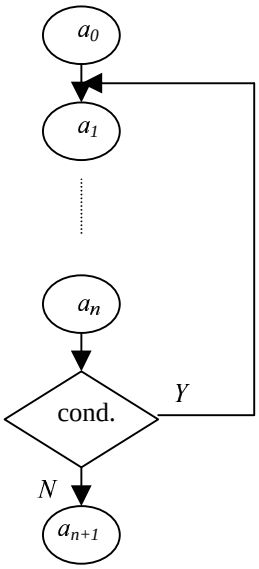

Fig. 4. Activities a_1 to a_n are executed several times.

In the event calculus, we may describe the iteration of activities as follows:

$$happens(start(a_1), T) \leftarrow happens(end(a_0), T). \quad (13)$$

$$happens(start(a_1), T) \leftarrow happens(end(a_n), T), holds\_at(loopcondition, T).$$

$$happens(start(a_{n+1}), T) \leftarrow happens(end(a_n), T), not\ holds\_at(loopcondition, T).$$

The activities between the activities a_1 and a_n, can be arranged in any of the transition types that we mentioned before. The number of the iterations can be controlled with a loop count variable that is increased by the last event of the iteration or with any condition which checks the state of the system by the time of the last event in the iteration.

5 Workflow Management

A workflow management system consists of a scheduler (manager) agent and task agents. A task agent controls the execution of an activity. The workflow manager is an agent that coordinates the execution of the activities according to the workflow specification. It knows which activities follow which ones under what conditions.

When an activity is taken, the manager must assign an agent that will execute that activity. Thus the manager must keep track of available agents as well. This section discusses how the manager design is done within the event calculus.

5.1 Agent Assignment

Every task agent is represented as a property in the modeled system. Each agent can perform one or more activities; and each activity can be executed by one or more agents. Which agents can perform which activities is specified with the predicate *qualified(Agent, Activity)*. When an activity is to be executed, an agent that is qualified for that activity is selected and the activity is assigned to that agent if the agent is idle. If the agent is not idle either another available, qualified agent is selected or the activity is kept waiting for the agents to finish their jobs. For instance, in a sequential execution of activities (Fig.1), the assignment of the agent is expressed by the following rule:

$$\begin{aligned} &happens(assign(Agent, a_{i+1}), T) \leftarrow \\ &\quad happens(end(a_i, \_), T), \\ &\quad qualified(Agent, a_{i+1}), holds\_at(idle(Agent), T), \\ &\quad not\ anysmalleragent(Agent, a_{i+1}, T). \end{aligned} \tag{14}$$

$$\begin{aligned} &anysmalleragent(Agent, A, T) \leftarrow \\ &\quad qualified(Agent2, A), Agent2 < Agent, \\ &\quad holds\_at(idle(Agent2), T). \end{aligned} \tag{15}$$

Rule (14) represents that the activity a_{i+1} is assigned to the qualified agent *Agent* at the completion of the activity a_i. If *Agent* is idle at that time and there is no other agent, with a smaller number, can run the activity. We must ensure that the same activity is not assigned to more than one agent at the same. We can achieve this by ordering the numbers of agents and checking for the minimum numbered idle agent to assign the activity to (rule 15). Here, the number used in the comparison of agents is an abstraction. It may denote the cost of executing the activity on the agent or it may denote a priority in assigning the activity to an agent in the definition of the predicate *qualified.*

When an agent *Agent* is assigned to an activity *A*, it starts executing that activity *A.* This can be represented by the following rule:

$$happens(start(A, Agent), T) \leftarrow happens(assign(Agent, A), T). \tag{16}$$

Here, we specify the agent explicitly in the event name.

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 becomes idle again, ready to execute the next activity. We describe these two states of an agent with two predicates: *idle(Agent)* and *assigned(Agent, Activity)*. The state of the agent may be changed by two events: *assign(Agent, Activity)* and *release(Agent, Activity).* Thus, we write the following rules:

initiates(assign(Agent, Activity), assigned(Agent, Activity)).
terminates(assign(Agent, Activity), idle(Agent)) .

happens(release(Agent, Activity),T) ← *happens(end(Activity, Agent) , T).*
terminates(release(Agent, Activity), assigned(Agent, Activity)).
initiates(release(Agent, Activity), idle(Agent)).

There are cases in which a task needs an agent but all agents are busy. In this case the activity must wait until one of the agents becomes idle. For instance, assume that in a sequential routing, after the last event of an activity a_i, activity a_{i+1} looks for an idle agent to access on. If there is no idle agent, activity a_{i+1} starts waiting for an appropriate agent:

initiates(end(a_i,_), waiting(a_{i+1}, Agent, T)) ← (17)
happens(end(a_i,_), T), qualified(Agent, a_{i+1}),
holds_at(assigned(Agent, C), T), C ≠ a_{i+1},
not anyotheragent(Agent, a_{i+1}, T).

anyotheragent(Agent, a_{i+1}, T) ← (18)
qualified(Agent2, a_{i+1}), Agent ≠ *Agent2,*
holds_at(idle(Agent2), T).

The activity a_{i+1} will be kept waiting if all the agents qualified for this activity are assigned to other activities. Rules (17) and (18) check the availability of all the qualified agents for a given activity.

It is also possible that several activities may wait for the same agent. If the agent becomes idle then the activity that has waited longest (i.e. the one with the smallest timestamp) is assigned to that agent (rule 19). This rule also deals with the problem of having more than one agent becoming idle at the same time. If that is the case, the activity is assigned to the minimum numbered agent:

happens(assign(Agent, Act), T) ← (19)
happens(release(Agent, _), T),
holds_at(waiting(Act, Agent, T1), T),
not waitinglonger(Act, Agent, T1, T),
not releasedotheragent(Agent, Act, T).

waitinglonger(Act, Agent, T1, T) ← (20)
holds_at(waiting(Act2, Agent, T2), T), Act ≠ *Act2, T2 < T1.*

releasedotheragent(Agent, Act, T) ← (21)
qualified(Agent2, Act), Agent2 < Agent,
happens(release(Agent2, _),T).

Rule (20) checks for any other activity that has waited longer for the currently released agent. And rule (21) checks for any other qualified agent being released at

the same time. When an activity is assigned to an agent that activity will not wait for any agents any more (rule 22).

$$terminates(assign(Agent, Act), waiting(Act, \_, T1)) \quad (22)$$

5.2 Concurrent Workflow Instances

So far, we have represented how to specify workflow activities and execute a single instance of a defined workflow within the event calculus framework. However in an application, there can be several instances of the same workflow executing at the same time. The available agents can perform the activities of any of the workflow instances. Thus the problem is how to represent more than one workflow instance concurrently within the current framework.

In order to model concurrent workflow instances, we number the workflow instances. Each workflow instance is assigned a unique number and every activity in a workflow is associated with a workflow instance. Thus we add the number of the workflow instance to the rules as well. For instance in the following rule we specify a sequential routing of activities using workflow instances:

$$happens(start(a_j, Wno), T) \leftarrow happens(end(a_i, Wno), T). \quad (23)$$

When an activity a_i in a workflow instance Wno finishes the next activity in the same workflow starts. In this rule we omit the agent assignment for simplicity. If concurrent workflow instances are modeled, the number of the workflow instance must be added to all the previous rules that we have discussed earlier.

5.3 Developing Workflow Management Utility

The manager actually runs the workflow specification (rules 1-13). It knows which activities follow which ones under what conditions. It also knows which agents can do which activities and which agents are available at all times. The manager agent starts a workflow process when an initial event is happened. Once that is recorded, the manager starts the other activities according to the workflow definition and the available agents. In our framework the workflow manager has a centralized control over the agents. All the agents inform the manager agent of the end of the activity (by *end(A)*) when they finish it. The manager starts an activity at an agent by recording the event *assign(Agent, Act)*. If no available agents are found for an activity, then the activity is kept waiting. When an agent is released one of the waiting activities is assigned to that activity.

We have given rules to describe the task of the manager agent with sequential routing of activities (rules 14-23). These rules can be applied to other routings of the activities that were specified in Section 4 in a similar way.

5.4 Querying the system

The state of the system can be queried at any time using the *holds_at* predicate. One can query the state of the system at a certain time *t* as:

?- holds_at(P, t).

The answer to this query will be all properties that can be derived using the rules of the system. In addition to this general querying, one can ask more specific queries like

?- holds_at(idle(A), t).
?- holds_at(waiting(Act, Agent, T), t).
?- holds_at(assigned(Agent, Activity), t).

One can also find out the starting and/or ending time of the activities by:
?-happens(start(a, Agent), T).

In this paper we have only demonstrated deriving the state of the system at a certain time point. However, in the event calculus it is also possible to derive the time periods for which a property is true, using the predicate *holds_for(Property, Period)* [11]. The existing framework can be extended to deal with time periods as well. Thus we can query the history of all processes in order to analyze the efficiency of the workflow. For instance, we may query the time period P to see how long an activity is kept waiting by:
?-holds_for(waiting(act, Agent, T), P).

6 The Computational Problem

The theory can be implemented in several different ways. One approach is to use a general-purpose theorem prover directly with the event calculus. Since the axioms presented are all Horn clauses, they can be used more or less directly with 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. For example consider the execution of the query *holds_at(assigned(Agent, Act), t).* A direct translation of the axioms into a Prolog program will cause an infinite loop, because the definition of *holds_at* includes calls to *happens* and that in turn includes calls to *holds_at.*

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 (but perhaps less declarative) [4]. In order to achieve this we consider the causality of events. 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 which events 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 cause in the history, and so on. The same approach is used in the implementation of the currently proposed framework.

7 Conclusion

We have demonstrated how the event calculus might be extended to describe the specification and execution of activities in a workflow. The major types of activity routings in a workflow can be expressed using the axioms of the event calculus in a declarative way. The proposed framework can be used as a quick tool in prototyping applications and/or simulations. Due to its additional temporal dimension, it provides facilities for querying the history of all activities, thus providing opportunities to analyze the efficiency of the workflows. The event calculus as presented exhibits three conceptual differences over the proposed formalisms.

First of all, it is purely declarative. Programs in most traditional formalisms usually contain side-effect causing operations such as event scheduling (insertion) and unscheduling (deletion) upon an event queue. We are able to compute event occurrences without any use of event queues, scheduling or unscheduling.

Second, there is no explicit state. Traditional programs keep past object states explicitly as a collection of tuples (P, V, T) each meaning that the value of state parameter P is V at time T. In the event calculus however only the sequence of events are retained instead of explicit states. The state at any time can be derived using the axioms of the event calculus.

Finally, a general definition of event is introduced. An event can be any real world event that occurs when a proposition becomes true. A wide range of happenings can be regarded as events and thus different domains can be modeled in a similar fashion.

The current framework is currently being extended to include exception handling in the workflows. We also investigate the ways of modeling rollback and compensation activities using declarative rules.

References

1. Alonsa, G., D. Agrawal, A. El Abdabi, and C. Mohan. Functionalities and Limitations of Current Workflow Management Systems. In *IEEE-Expert (Special Issue on Cooperative Information Systems)*, 1997.
2. Baral C., J. Lobo. Formalizing workflows as collections of condition-action rules. Dynamics '97, Workshop in ILPS, 1997..

3. Barbara, D., S. Mehrotra and M. Rusinkiewicz. INCAs: Managing Dynamic Workflows in Distributed Environments. *In Journal of Database Management*, vol.7, no.1, 1996.
4. Cicekli-Kesim, N. A Temporal Reasoning Approach to Model Workflow Activities, Lecture Notes in Computer Science, no. 1649, ed. R.Y. Pinter and S. Tsur, NGITS'99, Zikhron-Yaakov, Israel, July 1999.
5. Kesim, N. and M. Sergot. A Logic Programming Framework for Modelling Temporal Objects. In the IEEE Transactions on Knowledge and Data Engineering, vol. 8, no. 5, October 1996.
6. Kesim, F.N. and M. Sergot. Implementing an Object-Oriented Deductive Database Using Temporal Reasoning. In the Journal of Database Management, Vol. 7, No. 4, 1996.
7. Davulcu, H., M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic Based Modeling and Analysis of Workflows. In *ACM Symposium on Principles of Database Systems*; Seattle, Washington, ACM Press, 1998.
8. Dayal, U., M.Hsu and R. Ladin. Organizing Long_Running Activities With Triggers and Transactions. *In ACM SIGMOD Conference on Management of Data*, 1990.
9. Evans C. The Macro-Event Calculus: Representing Temporal Granularity. Technical Report, Imperial College, London, 1989.
10. Harel, D. StateCharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8:231-274,1987.
11. Kowalski, R.A. and M. J. Sergot. A Logic-based calculus of events. *New Generation Computing*, 4, pp. 67-95, 1986.
12. Kowalski, R.A. Database Updates in the Event Calculus. *Journal of Logic Programming* 12(1-2): 121-146 (1992).
13. Krishnakumar, N. and A. Sheth. Managing Heterogeneous Multi-System Tasks to Support Enterprise-wide Operations. In *Distributed and Parallel Databases*, Vol.3, No.2, April 1995.
14. Mohan, C., G. Alanso, R. Gunthor and M. Kamath. Exotica: A Research Perspective on Workflow Management Systems. In *Data Engineering*, Vol.18, No.1, March 1995.
15. Shanahan, M. Representing Continuous Change in the Event Calculus. *ECAI*, pp. 598-603, Stockholm, Sweden, 1990.
16. Shanahan, M. A Simple Logical Framework for Prediction Problems, Technical Report, Logic Programming Group, Imperial College, November 1988.
17. Singh, M.P. Semantical Considerations on Workflows: An algebra for Intertask Dependencies. *In Proceedings of the International Workshop on Database Programming Languages*, Gubbio, Umbria, Italy, September 6-8 1995.
18. Singh, M.P. Synthesizing Distributed Constrained Events From Transactional Workflow Specifications. In *proceedings of 12-th IEEE Intl. Conference on Data Engineering* , pp. 616-623, New Orleans, LA, February 1996.
19. Wachter, H. and A. Reuter. The ConTract Model. *In Transaction Models for Advanced Database Applications*, Chapter 7, Morgan_Kaufmann, February 1992.
20. Workflow Management Coalition. Terminology and Glossary. Technical Report (WFMC-TC-1011), Workflow Management Coalition, Brussels, 1996.