한국 *CAD/CAM* 학회 논문집
제 *15* 권 제 *1* 호 *2010* 년 *2* 월 *pp. 11-23*

제품개발환경을 지원하기 위한 Event Calculus 기반의 워크플로우 모델링

이희정*, 서효원**

Workflow Modeling for Product Development Environments based on Event Calculus

Heejung Lee* and Hyo-Won Suh**

ABSTRACT

A flexible and correct model of the activity flows is required for workflows in product development environments. In particular, the design activity flows are not known until run-time, and conventional approaches have limit to handle this situation because they cannot predefine all the potentially reachable paths. Thus, the structure of the workflow model must be flexible enough to describe variety in workflow design and accommodate dynamic changes during workflow execution. In this paper, we provide the general primitive axioms and change patterns based on event calculus for dynamic workflow specification and execution mechanisms in product development environments. Also, we describe how to execute the workflow dynamically based on the workflow specification and workflow change patterns using abductive planning technique.

Key words : workflow modeling, dynamic change, event calculus

1. Introduction

Workflow management systems (WfMS) are currently the leading technology to cope with the today's challenging environment. A workflow as the automation of a process involves a set of partially ordered activities to be undertaken by a set of procedural rules[1,3]. Typically workflow design aims at representing different flow of activities, capturing all possible situations with normal activity flows, without adjustment to a new situation.

Business activities and environments, as well as many engineering branches in general, are highly dynamic and subject to change. As the business climate is increasingly dynamic and competitive worldwide, redesign and optimization of existing business processes become essential in most organizations to gain better efficiency and effectiveness in the rapidly changing environments. Between radical redesigns, business processes often have to be adjusted over and over again. In addition to that, software systems are confronted with evolution requirements caused by technical advances. Technical advances often lead to systems reconfiguration, such as, replacement, updating of software components, addition of new components, and change in component interface. However, today's workflow systems are often built in the same way as traditional approach systems. That is, they are targeted at definite scenarios and not designed to cope with a rapidly and dramatically changing environment. To cope with this problem, the flexible workflow management issues have been a hot research topic.

Thus, our research objective is to develop a framework for managing workflows in a formal and orderly manner while allowing flexibility. Consequently, this paper includes following topics:

- workflow specification based on event calculus
- workflow modification rules
- workflow execution based on abductive planning and validation.

*정회원, 대구대학교 산업시스템공학과
**종신회원, 한국과학기술원 산업공학과
- 논문투고일: 2009. 09. 28
- 논문수정일: 2009. 11. 18
- 심사완료일: 2009. 12. 01

The remainder of this paper is organized as follows: Section 2 summarizes the related work and technical background. Section 3 suggests the formal workflow modeling method based on the logic programming, i.e., event calculus. Section 4 provides the workflow modification rules. Section 5 proposes workflow execution mechanism applying abductive planning technique, and also introduces the correctness and soundness problems, which are likely to be arisen in the logic programming. Section 6 contains the implementation of the proposed approach, and finally concluding remarks are described in section 7.

2. Related Work and Background

2.1 Product Development Environments

A product development process (PDP) is a set of activities beginning with the perception of a market opportunity and ending in the production, sale, and delivery of a product. In the PDP many decisions have to be made under uncertainty because of the insufficient accuracy level of data and the iterative feature. In addition, the PDP is for the most part human-based creative work that depends greatly on the specific knowledge of the people participating in the process[3]. The predictability and repeatability that can be found in the manufacturing process are not presented in the same degree in the PDP[4]. PDP projects are implemented as a means of achieving an organization's strategic plan and differ from each other in operation because each project is unique. Unique means that the product or service is different in some distinguishing way from all other products or services. Therefore, when we implement a PDP, a new process needs to be designed for each case[4]. We call this a one-of-a-kind process, and every case has its own process.

While the PDP is one of the most important business processes for the sustainable success of enterprise, the characteristics of the PDP make it a challenge to manage it in an effective and autonomous way. Workflow management systems (WfMS) have been suggested as a potential solution to deal with this, but only a few of them have focused on the PDP, focusing rather on general business processes, which are relatively simple, repetitive, and uncoupled. For these reasons, flexible workflow management issues, such as providing the ability of the workflow processes to react to changes in the environment in a consistent way, have been a hot research topic for the last few years.

2.2 Workflow Flexibility

Traditional workflow management systems, including *production workflow systems*, are process-oriented and aim at structured workflows. In addition, it is widely recognized that workflow management systems should also provide flexibility. Much research on flexible and adaptive workflow management systems has been carried out by diverse approaches. We summarize below three approaches in workflow flexibility.

Process adaptation: Process adaptability focuses on the ability of the workflow processes to react to exceptional circumstances. Usually workflow changes can take place at both workflow *schema* and the workflow *instance* level, so workflow flexible management should support both cases. One of the well-established frameworks for adaptive process management is the ADEPT2 change framework[5], which adequately deals with process changes during run time by supporting the following fundamental change requirements-support of structural adaptations at both the workflow schema and the workflow instance level, enabling a high level of abstraction when defining process changes; of change operations; and correctness of changes. CAKE2[6] and WASA2[7] support structural flexibilities at run time at the workflow instance level. Both approaches support only primitive changing such as adding or removing, while ADEPT2 provides support for a wide range of high-level change operations.

Other approaches include a case handling and a rule-based approach[3,8]. While the traditional workflow management deals primarily with the work item and control flows, the case handling approach[8] focuses on the case itself, e.g., the evaluation of a job application or the decision on a traffic violation. The central concept for case handling is the case and not the activities or the routing. The case acts as a primary driver to determine which activities are enabled. On the other hand, AgentWork[3], based on the rule-based approach, specifies exceptions and necessary workflow flexibilities, using temporal estimates to determine which remaining parts of running workflows are affected by an exception, and is able to perform suitable predictive adaptations.

Built-in flexibility: This approach deals with workflow flexibility by leaving process fragments unspecified at build time and by specifying the missing parts during run time. This is more useful in cases where the process can be structured with partial information by deferring uncertainty to run time. Worklets[4] is an approach for dynamic flexibility and evolution in workflows based on accepted ideas of how people actually work (called *Activity Theory*). In

Worklets, each task of a process instance may be linked to a repertoire of actions, one of which is contextually chosen at run time to carry out the task. Pockets of flexibility[9] allow ad hoc changes and/or building of workflows for highly flexible processes, providing the ability to execute based on a partially specified model, where the full specification of the model is made at run time. A constraint-based workflow model[10] combines the advantages of a declarative style of modeling and allows ad hoc and evolutionary changes, which makes it is possible both to avoid the need for unnecessary changes and restrictions using a more declarative style and to provide support changes at the schema and instance level.

Artificial Intelligence Planning: In the meantime, methods from the artificial intelligence planning community enable composition, adaptation, and synthesis of processes, thus providing the means to expand predefined process libraries to accommodate new situations and requirements. In addition, this community provides techniques for modifying activated processes in response to run-time failures and unexpected events. Two automated plan generation methods lend themselves most naturally to the synthesis of new processes from libraries of previously defined processes. Hierarchical task network (HTN) planning[11] synthesizes plans using libraries of processes (referred to as *task network*) defined over multiple levels of abstraction. Planning consists of incrementally refining tasks at high levels of abstraction by applying more refined task networks, eventually bottoming out in a set of directly executable tasks. HTN planning is well-suited to workflow management, given the similarity between processes and task networks. Case-based planning[12,13] generates new plans for a given situation and task by retrieving solutions for similar problems from a previously defined *case library*, and then adapting them to meet the requirements of the current situation. As such, case-based planning methods provide a way to build an experience with previously defined processes, providing adaptation to suit new conditions and requirements.

2.3 Event Calculus

The event calculus is based on axioms concerning notions of events, properties, and the time points at which the properties hold. The following primitives present the essentials of the event calculus[14].

- *holds* $\subseteq \mathbf{P} \times \mathbf{T}$
: *holds(p, t)* means that property $p \in \mathbf{P}$ is true at time $t \in \mathbf{T}$.
- *happens* $\subseteq \mathbf{E} \times \mathbf{T}$
: *happens(e, t)* means that event $e \in \mathbf{E}$ occurs at time $t \in \mathbf{T}$.
- *initiates* $\subseteq \mathbf{E} \times \mathbf{P} \times \mathbf{T}$
: *initiates(e, p, t)* means that if event $e \in \mathbf{E}$ occurs at time $t \in \mathbf{T}$, it will initiate property $p \in \mathbf{P}$.
- *terminates* $\subseteq \mathbf{E} \times \mathbf{P} \times \mathbf{T}$
: *terminates(e, p, t)* means that if event $e \in \mathbf{E}$ occurs at time $t \in \mathbf{T}$, it will terminate property $p \in \mathbf{P}$.
- *clipped* $\subseteq \mathbf{T} \times \mathbf{P} \times \mathbf{T}$
: *clipped*(t_1, p, t_2) means that property $p \in \mathbf{P}$ is terminated between times $t_1 \in \mathbf{T}$ and $t_2 \in \mathbf{T}$.
- *initially* $\subseteq \mathbf{P}$
: *initially(p)* means that property $p \in \mathbf{P}$ holds from time $0 \in \mathbf{T}$.
- $< \subseteq \mathbf{T} \times \mathbf{T}$
: standard order relation for time $t \in \mathbf{T}$.

Based on these primitives, the following axiom can be defined,

Axiom 1. *(Basic Event Calculus)*
- $holds(p, t) \leftarrow initially(p) \wedge \neg clipped(0, p, t)$.
- $holds(p, t) \leftarrow happens(e, t_0) \wedge initiates(e, p, t_0) \wedge t_0 \leq t \wedge \neg clipped(t_0, p, t)$.
- $clipped(t_0, p, t) \leftarrow happens(e', t') \wedge terminates(e', p, t') \wedge t_0 < t' \leq t$.

Axiom 1 means that a property p holds at the time t if p holds initially or for the period after an event e happens at time t_0, and there exists no such an event e' which happens between t_0 and t and terminates the property p.

3. Workflow Specification based on Event Calculus

The various workflow modeling techniques differ slightly in the extent to which they provide the ability to model different domain and system perspectives[15]. The *control flow* perspective describes activities and their execution ordering through different constructors, which permits execution-flow control. The *data* perspective deals with business and processing data, which is layered on top of the control perspective. The *resource* perspective provides an organizational structure anchor to the workflow in the form of human and device roles responsible for executing activities. The *operational* perspective describes element actions executed by activity, where the actions map into underlying applications. Ideally, what might be needed is the development of a single and holistic technique that could effectively represent all modeling perspectives in a thorough and concise form and hence be applicable in all modeling situation.

Clearly, the control flow perspective provides an essential insight into a workflow specification's effectiveness. In this paper, we are interested in catching the control flow perspectives of workflow and propose a formal framework for specifying and executing workflows based on the event calculus. To the best author's knowledge, the framework for specifying and executing workflows based on the event calculus was first proposed by Cicekli and Yildirim[16]. They have demonstrated how the event calculus might be extended to describe the specification and execution of activities in a workflow. However, they only deal with the routing rules without activity's state transition and dynamic environment.

The event calculus is the formalism reasoning about time and change[17]. It uses general rules to derive that a new property holds as the result of the event. With the narrative basis of the event calculus[18] we can cope with the abnormal situation during workflow enactment as well as standard workflow representation. An important feature of the event calculus is that it can be extended, without too much difficulty, to deal with some problems that are extremely hard to represent using other formal languages.

The workflow specification can be described by the basic axiom (*Workflow Event Calculus : WEC*), the state transition (W_{state}), and the routing control ($W_{routing}$). The action in W_{state} means any event that is considered relevant in a process involving routing of a workflow. The state in W_{state} means the property that is effect from relevant action. The routings in $W_{routing}$ are the temporal relationship or associations among actions, and consist of *happens* clauses and ordering of time points. Workflow Event Calculus (*WEC*) can be defined as follows.

Axiom 2. *(WEC)*

- *holds(state(activity), t)* $\leftarrow$ *initially(state(activity))* $\wedge \neg$ *clipped(0,state(activity), t).*
- *holds(state(activity), t)* $\leftarrow$ *happens(action(activity), t_0), initiates(action (activity), state(activity), t_0)* $\wedge$ $t_0 \leq t$ $\wedge \neg$ *clipped(t_0, state(activity), t).*
- *clipped(t_0, state(activity), t)* $\leftrightarrow$ *happens(action' (activity), t'), terminates(action' (activity), state (activity), t')* $\wedge$ $t_0 < t' \leq t$.

WEC means that a *state(activity)* holds at time t if a *state(activity)* holds initially or for the period after an *action(activity)* happens at time t_0, and there exists no such an *action'(activity)* which between t_0 and t and terminates the *state(activity)*. Example 1 means that the *activity* is *running* at time t if it holds initially or for the period after *run* action at time t_0, and there is no action which happens between t_0 and t and terminates its *running* state.

Example 1.

- *holds(running(activity), t)* $\leftarrow$ *initially(running(activity))* $\wedge \neg$ *clipped(0, running(activity), t).*
- *holds(running(activity), t)* $\leftarrow$ *happens(run (activity), t_0)* $\wedge$ *initiates(run(activity), running(activity), t_0)* $\wedge$ $t_0 \leq t$ $\wedge \neg$ *clipped(t_0, running(activity), t).*
- *clipped(t_0, running(activity), t)* $\leftrightarrow$ *happens(suspend (activity), t')* $\wedge$ *(terminates(suspend(activity), running(activity), t')* $\vee$ *terminates(abort(activity), running(activity), t'))* $\wedge$ $t_0 < t' \leq t$.

3.1 State Transition (W_{state})

The individual activity instance of a workflow will change its state in response to the actions. The descriptions of the states are as follows.

- *not_initiated*: if a workflow instance has been created, then all activities in the workflow are initially set to the *not_initiated* states.
- *initiated*: an activity instance has been created, but the activity has not yet fulfilled the conditions to cause it to start execution.
- *running*: an activity instance is processing.
- *completed*: an activity instance has fulfilled the conditions for completion.
- *suspended*: an activity instance is quiescent.
- *aborted*: the execution of an activity instance has been stopped before its normal completion.

Fig. 1 shows how the related actions (*i.e., trigger, start, restart, complete, suspend, abort*) change the state of an activity, and which state transitions are permissible.

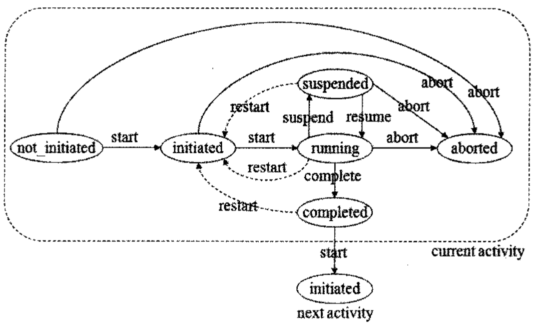


Fig. 1. State transition diagram.

In this paper, *initiates* and *terminates* predicates are used to specify how the state of an activity can change. In addition to defining which states they initiate or terminate, the required preconditions for activating these predicates can be specified using *holds* predicate. Therefore all state transitions can be specified in terms of a set of *initiates*, *terminates*, and *holds* clauses as follows.

- *(S_1) initiates(action(activity), state(activity), t)* *← holds(precondition(activity), t)*
- *(S_2) terminates(action(activity), state(activity), t)* *← holds(precondition(activity), t).*

(S_1) or (S_2) means that if a *precondition(activity)* holds at time *t*, then *action(activity)* occurs at the same time and will initiate *state(activity)* or terminates *state(activity)* respectively. The functional arguments in (S_1) and (S_2) are represented as *[action_name, state_name, precondition_name]*. We formalize all state transitions using (S_1) and (S_2) in axioms 3~8.

Axiom 3. *(not_initiated state : N_{state})*
•∀ *i, initially(not_initiated(activity$_i$)).*
•*(S_2) [action, state, precondition]* = *[(start, abort), not_initiated, not_initiated].*
: All activities are in *not_intiated* states from initial time. If an *activity* is in a *not_initiated* state and one of following actions *trigger* or *abort* occurs at time *t*, then the action terminates *not_initiated(activity).*

Axiom 4. *(initiated state : I_{state})*
•*(S_1) [action, state, precondition]* = *{[start, initiated, not_initiated], [rework, initiated, (running, completed, suspend)]}.*
• *(S_2) [action, state, precondition]* = *[(run, abort), initiated, initiated].*
: If an *activity* is in a *not_initated* state and an action *trigger* occurs at time *t*, then the action initiates *initiated(activity)*. If an *activity* is in one of following states, *running* or *suspended* state and an action *restart* occurs at time *t*, then the action initiates *initiated(activity)*. If an *activity* is in an *initiated* state and one of following actions *start* or *abort* occurs at time *t*, then the action terminates *initiated(activity).*

Axiom 5. *(running state : R_{state})*
• *(S_1) [action, state, precondition] = {[run, running, initiated], [resume, running, suspended]}.*
• *(S_2) [action, state, precondition] = [(suspend, complete, abort, rework), running, running].*
: If an *activity* is in an *initiated* or a *suspended* state and respectively an action *start* or *resume* occurs at time *t*, then the action initiates *running (activity)*. If an *activity* is in a *running* state and one of actions following *suspend, complete, abort,* or *restart* occurs at time *t*, then the action terminates *running(activity).*

Axiom 6. *(suspended state : S_{state})*
• *(S_1) [action, state, precondition] = [suspend, suspended, running].*
• *(S_2) [action, state, precondition]* = *[(abort, resume, rework), suspended, suspended].*
: If an *activity* is in a *running* state and an action *suspend* occurs at time *t*, then the action initiates *suspended(activity)*. If an *activity* is in a *suspended* state and one of following actions *abort, resume,* or *restart* occurs at time *t*, then the action terminates *suspended(activity).*

Axiom 7. *(completed state : C_{state})*
• *(S_1) [action, state, precondition]* *= [complete, completed, running].*
• *(S_2) [action, state, precondition]* = *[rework, completed, completed].*
: If an *activity* is in a *running* state and an action *complete* occurs at time *t*, then the action initiates *completed(activity)*. Notice that there is no action terminating *completed(activity).*

Axiom 8. *(aborted state : A_{state})*
• *(S_1) [action, state, precondition]* = *[abort, aborted, (not_initiated, initiated, running, suspended)].*
: If an *activity* is in one of following states, *not_initiated, initiated, running,* or *suspended* and an action *abort* occurs at time *t*, then the action initiates *aborted(activity)*. Notice that there is no action terminating *aborted(activity).*

3.2 Workflow Routing Controls ($W_{routing}$)

Sequences and dependences among activities can be specified in several ways: two activities can be directly connected, with the meaning that, as soon as the predecessor is completed, the successor is ready for execution. In all other cases, connections among activities are performed by special-purpose routing rules: *splits* and *joins*. Compositions of splits and joins may be used to represent iterations or other complex routing structures[14].

A split is preceded by one activity, i.e., predecessor, and followed by many activities, i.e., successors. Splits are classified as *And-Split* and *Or-Split.*

- *And-Split*: after the predecessor is completed, all successors are to run.
- *Or-Split*: each successor is associated with a

branch condition, and after the predecessor is completed, conditions are evaluated and only successors with a *true* condition are to run.

A Join is preceded by many activities, i.e., predecessors, and followed by one activity, i.e., successor. Joins are classified as *And-Join* and *Or-Join*.

- *And-Join*: only after all predecessors are completed, a successor is to run.
- *Or-Join*: the successor is to run every time whenever a predecessor is completed.

Axiom 9. *(Serial)*

- $initiates(trigger(Y), initiated(Y), t)\ holds(path(X, Y), t) \leftarrow holds(completed(X), t)$.

: There is a path between a predecessor X and a successor Y, and Y is initiated as soon as X is completed (Fig. 2-a).

Axiom 10. *(And-Split)*

- $initiates(trigger(Y_1), initiated(Y_1), t) \leftarrow holds(path(X, Y_1), t)\ holds(completed(X), t)$.
- $initiates(trigger(Y_2), initiated(Y_2), t) \leftarrow holds(path(X, Y_2), t)\ holds(completed(X), t)$.

 …

- $initiates(trigger(Y_n), initiated(Y_n), t))\ holds(path(X, Y_n), t) \leftarrow holds(completed(X), t)$.

: There are paths between a predecessor X and successors $Y_1, Y_2, \ldots, Y_n$ and all $Y_1, Y_2, \ldots, Y_n$ are initiated after X is completed (Fig. 2-b).

Axiom 11. *(Or-Split)*

- $initiates(trigger(Y_1), initiated(Y_1), t) \leftarrow holds(path(X, Y_1), t) \wedge holds(condition(XY_1), t) \wedge holds(completed(X), t)$.
- $initiates(trigger\ (Y_2), initiated(Y_2), t) \leftarrow holds(path(X, Y_2), t) \wedge holds(condition(XY_2), t) \wedge holds(completed(X), t)$.

 …

- $initiates(trigger\ (Y_n), initiated(Y_n), t) \leftarrow holds(path(X, Y_n), t) \wedge holds(condition(XY_n), t) \wedge holds(completed(X), t)$.

: There are paths between a predecessor X and successors $Y_1, Y_2, \ldots, Y_m$, and after the predecessor X is completed, $condition(XY_i)$ are evaluated and only successors with a *true* condition are initiated (Fig. 2-c).

Axiom 12. *(And-Join)*

- $initiates(trigger(Y), initiated(Y), max(t_1, t_2, \ldots, t_n))$
$\leftarrow holds(path(X_1, Y), t_1) \wedge holds(path(X_2, Y), t_2) \wedge, \ldots, \wedge holds(path(X_m, Y), t_n) \wedge holds(completed(X_1), t_1) \wedge holds(completed(X_2), t_2) \wedge, \ldots, \wedge holds(completed(X_n), t_n)$.

: There are paths between predecessors $X_1, X_2, \ldots, X_n$ and successors Y, and only after all predecessors $X_1, X_2, \ldots, X_n$ are completed, a successor Y is initiated (Fig. 2-d).

Axiom 13. *(Or-Join)*

- $initiates(trigger(Y), initiated(Y), t)$
$\leftarrow holds(path(X_1, Y), t) \wedge holds(path(X_2, Y), t) \wedge, \ldots, \wedge holds(path(X_m, Y), t) \wedge (holds(completed(X_1), t) \vee holds(completed(X_2), t) \vee, \ldots, \vee holds(completed(X_n), t))$.

: There are paths between predecessors $X_1, X_2, \ldots, X_n$ and successor Y, and the successor Y is initiated every time whenever any predecessor X_i $(i = 1, 2, \ldots, n)$ is completed. Generally Or-Join definition is not easy in workflow context, and many applications use the Or-Join term as exclusive join (XOR-Join) instead. In this paper, Or-Join is considered as iterative join in that the activities arriving later at joining point cannot be started immediately but in the initiated states until the earlier (in running) will be completed, suspended, or aborted (Fig. 2-e).

Axiom 14. *(Loop)*

- $initiates(trigger(Y_1), initiated(Y_1), t)$
$\leftarrow holds(path(X_n, Y_1), t) \wedge holds(loopcondition(X_nY_1), t) \wedge holds(completed(X_n), t)$.
- $initiates(trigger(X_1), initiated(X_1), t)$
$\leftarrow holds(path(Y_n, X_1), t) \wedge holds(completed(Y_n), t)$.

: Like *Or-Split*, a branch activity X_n in a loop can be associated with a loop condition, which is evaluated whenever the activity is completed. A merge activity X_1in a loop can be considered as a successor in *Or-Join*. In a loop, the merge activity is in a *completed* state, because the merge activity has already been done (Fig. 2-f).

Now we look at how the routing axioms ($W_{routing}$) can be used to specify the workflow process by an example. Considering the workflow in Fig. 3, there are 12 activities indexed by from A to L, and flow and the routing controls such as the *And-Split*, *Or-*

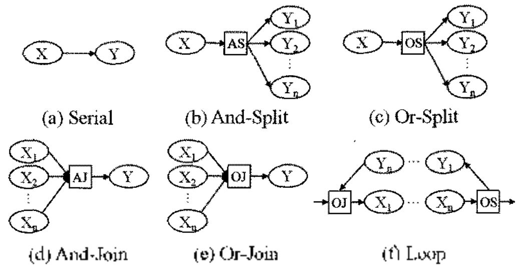


Fig. 2. Workflow routings.

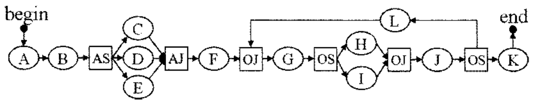


Fig. 3. Workflow example.

```
/*Serial*/
initiates(start(A),initiated(A),t0)
←holds(path(begin,A),t0) ∧ holds(completed(begin),t0).
initiates(start(B),initiated(B),t1)
←holds(path(A,B),t1) ∧ holds(completed(A),t1).
initiates(start(G),initiated(G),t6)
←holds(path(F,G),t6) ∧ holds(completed(F),t6).
initiates(start(end),initiated(end),t11)
←holds(path(K,end),t11) ∧ holds(completed(K),t11).
/*And-Split*/
initiates(start(C),initiated(C),t2)
←holds(path(B,C),t2) ∧ holds(completed(B),t2).
initiates(start(D),initiated(D),t2)
←holds(path(B,D),t2) ∧ holds(completed(B),t2).
initiates(start(E),initiated(E),t2)
←holds(path(B,E),t2) ∧ holds(completed(B),t2).
/*And-Join*/
initiates(start(F),initiated(F),max(t3,t4,t5))
←holds(path(C,F),t3) ∧ holds(path(D,F),t4)
 ∧ holds(path(E,F),t5) ∧ holds(completed(C),t3)
 ∧ holds(completed(D),t4) ∧ holds(completed(E),t5)
/*Or-Split*/
initiates(start(H),initiated(H),t7) ←holds(path(G,H),t7)
 ∧ holds(condition(GH),t7) ∧ holds(completed(G),t7).
initiates(start(I),initiated(I),t7) ←holds(path(G,I),t7)
 ∧ holds(condition(GI),t7) ∧ holds(completed(G),t7).
/*Or-Join*/
initiates(start(J),initiated(J),t8)
←holds(path(H,J),t8) ∧ holds(path(I,J),t8)
 ∧ (holds(completed(H),t8) ∨ holds(completed(I),t8)).
/*Loop*/
initiates(start(K),initiated(K),t9) ←holds(path(J,K),t9)
 ∧ holds(condition(JK),t9) ∧ holds(completed(J),t9).
initiates(start(L),initiated(L),t9) ←holds(path(J,L),t9)
 ∧ holds(condition(JL),t9) ∧ holds(completed(J),t9).
initiates(start(G),initiated(G),t10)
←holds(path(L,G),t10) ∧ holds(completed(L),t10)
```

Fig. 4. Workflow specification example.

Spit, *And-Join*, and *Or-Join* to model serial, conditional, parallel and loop routing. While parallel routing normally commences with *And-Split* and concludes with *And-Join*, conditional and loop routing commence with *Or-Split* and with *Or-Join*. Fig. 4 shows the workflow specification for the Fig. 3.

4. Workflow Change Patterns

To design workflows in changing and dynamic environments, a flexible, correct, and rapid realization of models of the activity flow is required. We are concerned with dynamic structural change. 'Structural' means that we are concerned with changes to the structure of workflows; we are not concerned in this paper such as changes to the value of an application data variable. 'Dynamic' means that we are required to make the change 'on the fly' in the midst of continuous execution of the changing procedure. Not only the correctness and consistency before and after dynamic change, but also the state of an activity in a workflow is the major criteria for deciding whether a specific structural change can be applied to it or not. As an example, the new addition of an activity between the two successive activities should not be permitted if the successor was already completed.

In particular, techniques are needed to design workflows capable of adapting themselves effectively when dynamic situation occurs during process execution. In this section, we present an approach to flexible workflow design based on patterns applying *WEC*, W_{state}, and $W_{routing}$ of the previous section. Now we define patterns that can frequently occur in workflow modeling.

When a new activity is inserted into an existing workflow, new paths must be added (*by initiating the new path*) and old paths must be removed (*by terminating the old path*) while maintaining the correctness and consistency of the workflow.

Let be a finite set of patterns Π $\{P_a, P_d, P_r, P_{os}, P_{oj}\}$. A pattern $P_j \in \Pi$ is defined as 2-tuple, P_j = <*Name*, *Template*>, where, the name of a pattern should be meaningful to indicate its purpose and be unique to identify the pattern, and the template of a pattern is defined as a non empty set of event calculus.

4.1. Add Pattern: P_a

Description: A change is of type P_a if a new activity is introduced between the exiting activities in serial, parallel, or conditional routings.

P_a = <Add, T_a> (see Fig. 5)
/ Initial condition */*
holds(path(X, Y), t) ∧ (holds(not intiated(Y), t) ∨ holds(initiated(Y), t)).
/ Change template : T_a */*
step 1 : initiates(add(X, New), path(X, New), t).
step 2 : initiates(add(New, Y), path(New, Y), t).
step 3 : terminates(delete(X, Y), path(X, Y), t).

State constraints: The add pattern can be applied to following situations: serial, parallel (*And-Split* and *And-Join*) or conditional (*Or-Split* and *Or-Join*). Furthermore, the applicability of the add pattern depends on the state of the activities in a workflow. To avoid the addition of a new activity as a not-accepted state such as *running* or *completed*, we require that all successors of activity *New* must be in one of the states *not_initiated* or *initiated*. The predecessor of activity *New* may be in an arbitrary state.

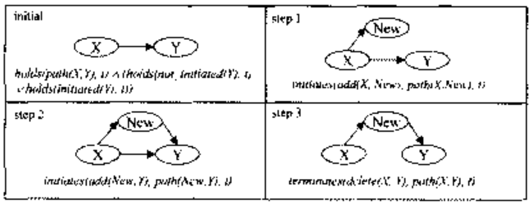

initial
X
Y
holds(path(X,Y), t) ∧ (holds(not_initiated(Y), t) ∨ holds(initiated(Y), t))
step 1
New
X
Y
initiates(add(X, New), path(X,New), t)
step 2
New
X
Y
initiates(add(New,Y), path(New,Y), t)
step 3
New
X
Y
terminates(delete(X, Y), path(X,Y), t)

Fig. 5. Add procedure.

4.2. Delete Pattern: P_d

Description: A change is of type P_d if an activity is removed between the exiting activities in serial, parallel, or conditional routings.

P_d = <*Delete*, T_d> (see Fig. 6)
/* *Initial condition* */
holds(path(X, Y), t) ∧ *holds(path(Y, Z), t)* ∧ *(holds(not_intiated(Y), t)* ∨ *holds(initiated(Y), t))*.
/* *Change template* : T_d */
step 1 : initiates(add(X, Z), path(X, Z), t).
step 2 : terminates(delete(X, Y), path(X, Y), t).
step 3 : terminates(delete(Y, Z), path(Y, Z), t).

State constraints: The delete pattern can be also applied to following situations: serial, parallel (And-Split and And-Join) or conditional (Or-Split and Or-Join) routings. Furthermore, the applicability of the delete pattern also depends on the state of the activities in a workflow. The deletions of an activity *Y* is possible, if the activity which is going to be deleted is either in the state *not_initiated* or *initiated*.

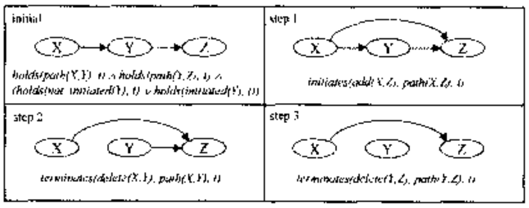

initial
X
Y
Z
holds(path(X,Y), t) ∧ holds(path(Y,Z), t) ∧ (holds(not_initiated(Y), t) ∨ holds(initiated(Y), t))
step 1
X
Y
Z
initiates(add(X,Z), path(X,Z), t)
step 2
X
Y
Z
terminates(delete(X,Y), path(X,Y), t)
step 3
X
Y
Z
terminates(delete(Y,Z), path(Y,Z), t)

Fig. 6. Delete procedure.

4.3. Replace Pattern: P_r

Description: A change is of type replace (composition of add and delete) if an activity is replaced by another activity in serial, parallel, or conditional routings. The replace pattern always can be composed of P_a and P_d pattern.

P_r <*Replace*, T_r>
$T_r \equiv T_a \wedge T_d$

4.4. Reorder Pattern: $P_o \equiv P_{os} \wedge P_{op}$

Description: A change is of type P_o if activities are reordered without addition or removal. Reorder pattern is comprised of the followings: serializing activities that were previously allowed to run in parallel (P_{os}) and parallelizing activities that were previously allowed to run in serial (P_{op}). Notice that conditionalizing activities that previously were constrained to execute in parallel and vice versa are not involved in the structural change.

P_{os} = <*Reorder*$_s$, T_{os}> *(see Fig. 7)*
/* *Initial condition* */
holds(path(X, Y), t) ∧*holds(path(X, Z), t)* ∧ *holds(path(Y, W), t)* ∧*holds(path(Z,W), t)*
∧ ¬*holds(completed(Z), t).*
/* *Change template* : T_{os} */
step 1 : initiates(add(Y, Z), path(Y, Z), t).
step 2 : terminates(delete(X, Z), path(X, Z), t).
step 3 : terminates(delete(Y, Z), path(Y, W), t).

State constraints: In Fig. 7, a workflow, which originally does activity Y and Z at the same time, makes a dynamic change to its procedure by performing activity Z after activity Y. Although the procedure looks safe after the change, there are problems that could potentially surface during the change[14,19]. For example, after activity X is completed, the successors Y and Z start to run in parallel, and there may happen that the activity Y is still running and the activity Z is already completed after some while. If the dynamic change P_{os} on the fly occurs at this time, the activity W will be started to run without completing the activity Y due to the routing rules, and this is an undesired result. This undesirable bug can be avoided by simply writing *holds(completed(Z), t).*

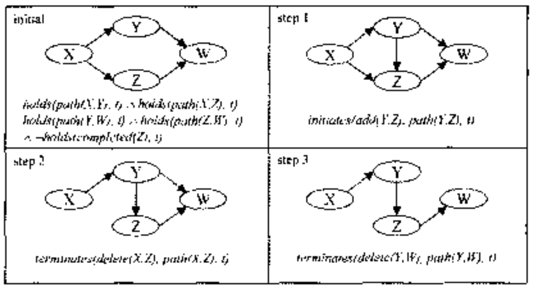
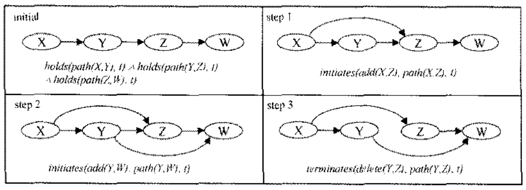

initial
X
Y
Z
W
holds(path(X,Y), t) ∧ holds(path(X,Z), t)
holds(path(Y,W), t) ∧ holds(path(Z,W), t)
∧ ¬holds(completed(Z), t)
step 1
X
Y
Z
W
initiates(add(Y,Z), path(Y,Z), t)
step 2
X
Y
Z
W
terminates(delete(X,Z), path(X,Z), t)
step 3
X
Y
Z
W
terminates(delete(Y,W), path(Y,W), t)

Fig. 7. Reorder procedure (Serializing).

P_{op} <$Reorder_p$, T_{op}> (see Fig. 8)
/ Initial condition */*
holds(path(X, Y), t) holds(path(Y, Z) holds(path(Z, W), t).
/ Change template : T_{op} */*
step 1 : initiates(add(X, Z), path(X, Z), t).
step 2 : initiates(add(Y, W), path(Y, W), t).
step 3 : terminates(delete(Y, Z), path(Y, W), t).

State constraints: In Fig. 8, a workflow, which originally executes activity Y and Z in sequential order, makes a dynamic change to its procedure by performing activity Y and Z in parallel, i.e., there is no ordering relationship between the activities Y and Z. In this change, there is no problem and it is always possible to transfer.


initial
X
Y
Z
W
holds(path(X,Y), t) ∧ holds(path(Y,Z), t) ∧ holds(path(Z,W), t)
step 1
initiates(add(X,Z), path(X,Z), t)
step 2
initiates(add(Y,W), path(Y,W), t)
step 3
terminates(delete(Y,Z), path(Y,Z), t)
Fig. 8. Reorder procedure (Parallelizing).

5. Workflow Execution and Planning

The workflow management system is composed of two main components: the process definition and workflow enactment service. The process definition is used in build-time to generate a computerized and executable definition of a workflow. In commercial workflow management systems, it provides graphical modeling tools and helps the designer to design, test, and validate workflow process. In this paper, the process definition is characterized by the workflow specification applying event calculus proposed in section 3. The workflow enactment service is composed of a set of software modules for creating and controlling instances of processes during run-time, and provides the run-time environment in which workflow management systems also manage the execution and sequencing of the various activities of the workflow.

5.1. Abductive Planning

We describe how to execute the workflow dynamically based on the workflow specification (in section 3) and workflow change patterns (in section 4) using abductive planning technique. For the dynamic workflow execution, we apply the abductive planning to the event calculus, and abductive planning will produce the ordering of actions which comprise the workflow. We firstly give a brief introduction to the abductive inference rule. At first the rule of inference called *resolution* which is based on *modus ponens* augmented with *unification* is as follows: '*from p ← q and q, we can infer p*'. While our workflow execution is based on the abduction inference rule: '*from p ← q and p, we can infer q*'

That is, given a rule $p \leftarrow q$ and a fact p, abduction generates an explanation for p by stating the fact q. This is not a sound inference rule, however, because q does not logically follow from $p \leftarrow q$ and p. Therefore, q must be seen a hypothetical explanation. If the implication $p \leftarrow q$ corresponds to the notion of causality, then abduction will generate plausible explanations.

Now let's explore how this abductive inference rule can be applied to the dynamic workflow execution. A plan consists of the set of facts defined by the predicates happens and temporal ordering formulae (<). This set can represent a plan because it defines the actions that happen and the time ordering between actions.

Definition 1. A plan P is a solution for a goal G with respect to the theory T and history H such as $T \wedge H \wedge P$ *infer* G, where, $T = WEC \wedge W_{state} \wedge W_{routing} \wedge TL$

This definition means that any planning problem P consists of finding one or more valid and possible paths can be generated from the workflow specification theory T and the current history H to a given goal G. History H is initially empty and will be updated as $H \wedge P$ every time plan P is generated by a dynamic goal state. If a desired goal G is related to the modifications of the existing workflow, then the structure of a workflow can be changed by a set of primitive provided patterns.

We present examples to show how the dynamic workflow paths and related activity states are generated and stored through the abductive planning. Let's see the workflow example in Fig. 3 again. Initially all activities are in the *not initiated* states and *initiated(begin)* sets to true. If our desired goal state is *holds(completed(B), t)*, then one possible plan P (there can be so many solutions including repetition of *happens(suspend(X), t_i)* and *happens(resume(X), t_j))* is the conjunction of the following happens and temporal ordering formulae (<).

P = [happens(trigger(A),t1), happens(start(A),t2), happens(complete(A),t3), happens(trigger(B),t3), happens (start(B),t4), happens(complete(B),t5), t1<t2, t2<t3, t3<t4, t4<t5]

Fig. 9 is the abductive proof tree of the above example, representing the top down search space of possible paths using dynamic workflow specification rules. This search space will be terminated until they found the initial condition or adequate history *H*.

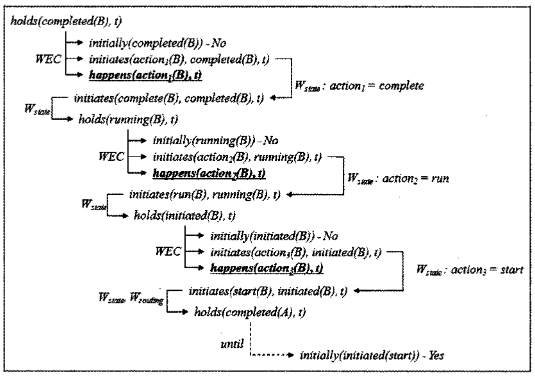

holds(completed(B), t)
initially(completed(B)) - No
WEC
initiates(action1(B), completed(B), t)
happens(action1(B), t)
Wstate : action1 = complete
initiates(complete(B), completed(B), t)
Wstate
holds(running(B), t)
initially(running(B)) - No
WEC
initiates(action2(B), running(B), t)
happens(action2(B), t)
Wstate : action2 = run
initiates(run(B), running(B), t)
Wstate
holds(initiated(B), t)
initially(initiated(B)) - No
WEC
initiates(action3(B), initiated(B), t)
happens(action3(B), t)
Wstate : action3 = start
Wstate, Wrouting
initiates(start(B), initiated(B), t)
holds(completed(A), t)
until
initially(initiated(start)) - Yes

Fig. 9. Abductive proof tree.

Now let's assume that we are at time t_{now}(= t_5 + some time), i.e., both activities A and B were completed, and we are now performing the parallel activities C and D, and we have completed E, i.e., *holds(running(C),* t_{now}*), holds(running(D),* t_{now}*),* and *holds(completed(E),* t_{now}*).* To make a dynamic change by performing activity E after D serially instead of parallel processing will fail because change pattern P_{os} will preserve reasonable change (Fig. 10). Without losing control, one can evaluate the dynamic situation at workflow running time and cope with any dynamic change.

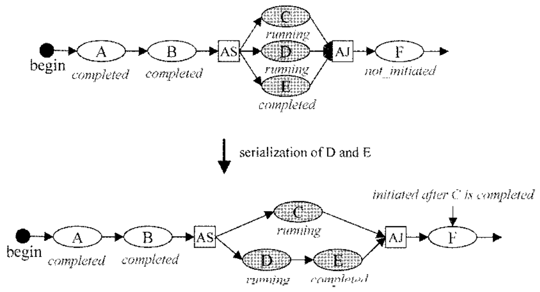


Fig. 10. Example of dynamic change.

Given *H* which is updated as *P* ∧ *holds(running (C),* t_{now}*)* ∧ *holds(running(D),* t_{now}*)* ∧ *holds(completed (E),* t_{now}) (Fig. 11), we assume next goal state *G* = *holds(completed(G), t)*, then the valid plan *P'* will be generated (Fig. 12) as follows.

P = [happens(complete(C),t1), happens(complete(D), t2), happens(trigger(F), t3=max(t1,t2)), happens(start (F), t4), happens(complete(F),t5), happens(trigger(G), t5), happens(start(G),t6), happens(complete(G),t7), t3<t4, t4<t5, t5<t6, t6<t7].

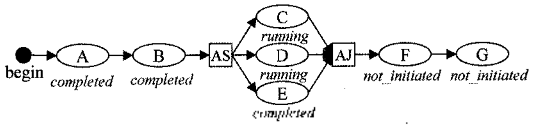


Fig. 11. Example of history.

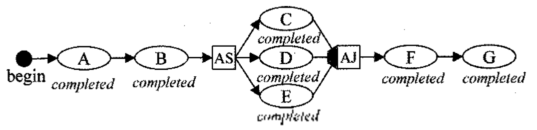


Fig. 12. Example of planning result.

If our desired goal state *G* is like *holds (completed(K), t),* which is for the last activity in the workflow, abductive plan *P* is considered as workflow scheduling to generate event notices for all activities that can be performed.

5.2. Correctness and Soundness

Event calculus is an express representation language for planning problem. Consider the following example, taken from[20].

initially(r).
initiates(e_1, p, t).
initiates(e_2, q, t).
terminates(e_1, r, t) ← holds(q, t).
terminates(e_2, r, t) ← holds(p, t).

: the terminates rules define the context dependent effect, *i.e.,* e_1 *terminates r if q holds, and* e_2 *terminates r if p holds.* Given the goal *G = [holds(p, t)* ∧ *holds(q, t)* ∧ *holds(r, t)]*, the abductive planning would find the following solution considered incorrect.

$$P = [happens(e_1, t_1), happens(e_2, t_2), t_1 < t, t_2 < t].$$

This example shows that correctness problems occur in the case of context dependent terminating events, and all non-linear planners that allow for such effects are incomplete and generally incorrect. However, we can exclude such events because there is no action in our workflow model that interferes with each other in a context dependent way.

An abductive planning must meet certain constraints to ensure the correct execution of the workflow at run-time.

First, an arbitrary goal state *holds(completed (activity), t)* should be true in a workflow instance by the appropriate plan. Consider the following example,

- *initiates(trigger(B), initiated(B), t)*
  $\leftarrow$ *holds(path(A, B), t)* $\wedge$ *holds(condition(AB), t)* $\wedge$ *holds(completed(A), t).*
- *initiates(trigger(C), initiated(C), t)*
  $\leftarrow$ *holds(path(A, C), t)* $\wedge$ *holds(condition(AC), t)* $\wedge$ *holds(completed(A), t).*
- *initiates(trigger(D), initiated(D), t)*
  $\leftarrow$ *holds(path(B, D), t)* $\wedge$ *holds(path(C, D), t)* $\wedge$ *holds(completed(B), t)* $\wedge$ *holds(completed(C), t).*

Given the goal G = *holds(completed(D), t)*, the abductive planning would find the failing intermediate solution. As shown in Fig. 13, in order that *holds (completed(D), t)* should be true, both *holds(completed (B), t)* and *holds(completed(C), t)* are to be true, However, this could never happen, when *path(A, B)* and *path(A, C)* are alternative, so only one of *condition(AB)* and *condition(AC)*should hold at the time activity *A* is *completed.*

Second, the workflow will be finished eventually and at that moment there should not be any activity that is running. Consider the following example,

- *initiates(trigger(B), initiated(B), t)*
  $\leftarrow$ *holds(path(A, B), t)* $\wedge$ *holds(completed(A), t).*
- *initiates(trigger(B), initiated(B), t)*
  $\leftarrow$ *holds(path(C, B), t)* $\wedge$ *holds(completed(C), t).*
- *initiates(trigger(C), initiated(C), t)*
  $\leftarrow$ *holds(path(A, C), t)* $\wedge$ *holds(completed(A), t).*
- *initiates(trigger(D), initiated(D), t)*
  $\leftarrow$ *holds(path(C, D), t)* $\wedge$ *holds(completed(C), t).*

Given the goal G = *holds(completed(D), t)*, the abductive planning would find the valid solution. But after the workflow is correctly completed, there still exists any live activity in the completed workflow. As shown in Fig. 14, after activity *C* is *completed*, the action *happens(trigger(B), t)* and *happens(trigger (D), t)* will be fired concurrently, and then the activity *D* will be *running* and then *completed* normally. However activity *B* and *C* in the loop are can be still in *running* states.

These two results are not desired, and we can avoid these problems by the following methods. Cleary the parallel flows started by an And-Split should not be joined by an Or-Join. The conditional flows created via an Or-Split should not be joined by an And-Join. In Summary, an And-Split should be completed by an And-Join and an Or-Split should be complemented by an Or-Join. Furthermore, the loop is controlled by Or-Split and Or-Join instead of And-Join and And-splitto preserve the finite iteration.

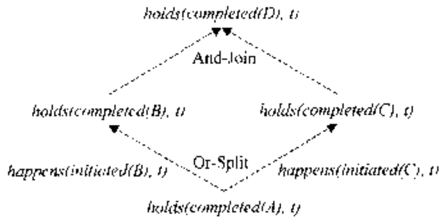


Fig. 13. Soundness Problem (1)

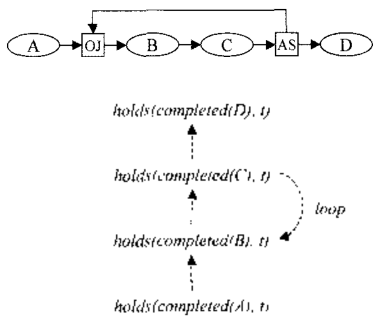


Fig. 14. Soundness Problem (2).

6. Implementation

The proposed approach can be implemented in several different ways. One approach is to write the axioms directly in *Prolog*. However, this will cause an infinite loop, because the definition of *holds* predicate includes calls to *initiates* or *terminates* predicate that in turn includes calls to *holds*. One such abductive event calculus planner avoiding this problem is due to[21]. The following's behavior is equivalent to that of the *vanilla* meta-interpreter with the object-level clause for λ_0 :- λ_1, λ_2, ..., λ_n.

demo([λ_0 | G_1]*) :- axiom(*λ_1, G_2*), append(*G_2, [λ_2, ..., λ_n | G_1], G_3*), demo(*G_3*)*

This is an abductive meta-interpreter for abduction with negation-as-failure, with built-in features for handling event calculus queries; we modify this event calculus planner here to demonstrate how possible paths can be generated and stored. As the number of activities in workflow increases, the search space in this program also expands. We can reduce this problem space by introducing the facts of history *H*.

The workflow manager actually runs the workflow specification. In normal situations, the manager starts a workflow process by an initial plan for an initial goal. Once the abnormal situation happens, the workflow manager queries the new goal to cope with this exceptional situation, and reschedule the workflow process. The proposed approach can be also used as a quick tool in prototyping applications or simulations, thus providing opportunities to analyze the efficiency of the workflows.

The ideas discussed in this paper have been implemented with *SWI-Prolog*. The example workflow as shown in Fig. 15 can be expressed in *Prolog* syntax and the workflow planning solution *R* will be obtained by the query template, *e.g. abdemo([holds (state(activity),t), ... holds(state(activity),t)],R),* where, *R* is solution which consists of actions and temporal orders. For example, after execution of queries such as '*abdemo([holds(completed(f), t)], R)*' and '*abdemo ([holds(path(f, r)), t), holds(path(r, g)), t), holds(neg (path(f ,g)), t)], R)*', which can be interpreted as applying *state transition* and *change pattern* respectively, solution *R* will be displayed like Fig. 16. First solution depicts the state transition results of activities C, D, and F, and second solution shows the path change. The *Prolog* file for abductive planner about this example is partially shown in Appendix.

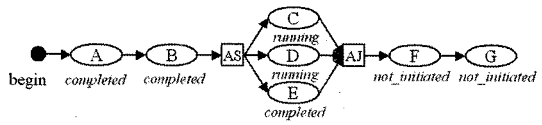
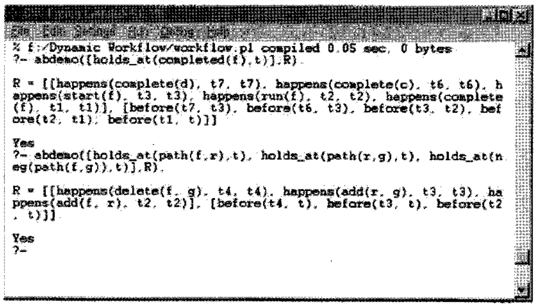


Fig. 15. Example workflow.



```
% f:/Dynamic Workflow/workflow.pl compiled 0.05 sec, 0 bytes
?- abdemo([holds_at(completed(f),t)],R).

R = [[happens(complete(d), t7, t7), happens(complete(c), t6, t6), h
appens(start(f), t3, t3), happens(run(f), t2, t2), happens(complete
(f), t1, t1)], [before(t7, t3), before(t6, t3), before(t3, t2), bef
ore(t2, t1), before(t1, t)]]

Yes
?- abdemo([holds_at(path(f,r),t), holds_at(path(r,g),t), holds_at(n
eg(path(f,g)),t)],R).

R = [[happens(delete(f, g), t4, t4), happens(add(r, g), t3, t3), ha
ppens(add(f, r), t2, t2)], [before(t4, t), before(t3, t), before(t2
, t)]]

Yes
?-
```

Fig. 16. Workflow planning solution.

7. Conclusion

In this paper, we have presented an approach to the dynamic workflow specification and execution applying event calculus extensions. The approach is based on modeling the general flow of activities and the dynamic behavior separately. Major contribution of the approach is related to the design of workflows in changing and dynamic environment, while preserving flexible and correct models of the activity flow. The general primitive axioms and change patterns for dynamic specification and execution mechanisms in workflow management systems can provide a good degree of flexibility and can be used in many applications.

In this paper we have also shown that the activity state transition and major types of routings *(i.e., Serial, And-Split, Or-Split, And-Join, Or-Join, Loop)* can be expressed and how the dynamic workflow paths and related activity states are generated and stored through the abductive planning.

This paper focuses on flexible specifications and evolutionary changes concerning the flow structure, i.e., definition of the sequence in which activities should be executed within a workflow. Future work in this area includes the extensions of the analysis to modification of other workflow characteristics such as data control, information handling, or organizational operations, which can be done by considering such predicates as: *activity(agent, input, output)*, *role(agent, activity)*.

Appendix

<Basic structure of abductive planner in *Prolog* syntax>

// WEC //

- *demo([holds(S, T2)|G1]) :- axiom(initiates(A, S, T1),G2), axiom(happens(A,T1),G3), axiom(before (T1,T2), []), demo([not(clipped(T1,S,T2))]), append (G3,G2,G4), append(G4,G1,G5), demo(Gs5).*

// Workflow State Transition Rule //

- *Axiom(initiates(action(Activity), state(Activity), T), [holds(condition(Activity), T)]).*
- *Axiom(terminates(action(Activity), state(Activity), T), [holds(condition(Activity), T)]).*

// Workflow Routing Rule //

- *Axiom(initiates(start(Activity), initiated(Activity), T)), [holds(path(_,_),T),...,holds(path(_,_),T), cond (_),...,cond(_), holds(completed(Activity),T),...,holds (completed(Activity),T)]).*

// Workflow Changing Rule for Add //

- *Change(X,New,Y,pa) :- abdemo(holds(path(X,New), t), holds(path(New,Y),t), holds(neg(path(X,Y)),t)],R).*

// Workflow Changing Rule for Delete //

- *Change(X,Old,Y,pd) :- abdemo(holds(path(X,Y),t), holds(neg(path(X,Old),t)), holds(neg(path(Old,Y)),t)], R).*

// Workflow Changing Rule for Replace //

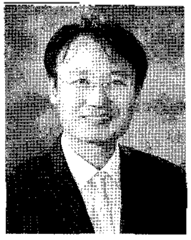
- *Change(X,Old, New,Y,pr) :- Change(X,New,Y,pa), Change(X,Old,Y,pd)*

// Workflow Changing Rule for Reorder //

- *Change(X,Y,Z,W,pos) :- abdemo(holds(path(Y,Z),t), holds(neg(path(X,Z)),t), holds(neg(path(Y,W)),t)],R).*
  // Workflow Initial Condition and History //
- *axiom(initially(state(Activity)),[ ]).*
- *axiom(initially(path(Activity,Activity)),[ ]).*
- *axiom(cond(Activity,Activity)),[ ]).*

References

1. WfMC, *The Workflow Reference Model*, WFMC-TC-1003, Workflow Management Coalition, 1995.
2. WfMC, *Terminology & Glossary*, WFMC-TC-1011, Workflow Management Coalition, 1996.
3. Müller, R., Greiner, U. and Rahm, E., "AgentWork: A Workflow System Supporting Rule-based Workflow Adaptation", *Data and Knowledge Engineering*, Vol. 51, pp. 223-256, 2004.
4. Adams, M., Hofstede, A., Edmond, D. and van der Aalst, W. M. P., "Worklets: A Service-oriented Implementation of dynamic flexibility in Workflows", *CoopIS'06*, pp. 291-308, 2006.
5. Reichert, M. and Dadam, P., *Realizing Adaptive Process-aware Information Systems with ADEPT2*, Ulmer Informatik-Berichte, Nr.2013-08, 2008.
6. Minor, M., Schmalen, D., Koldehoff, A. and Bergmann, R., "Structural Adaptation of Workflows Supported by a Suspension Mechanism and by Case-based Reasoning", *WETICE'07*, 2007.
7. Weske, M., *Workflow Management Systems: Formal Foundation, Conceptual Design, Implementation Aspects*, University of Münster, Habilitation Thesis, 2000.
8. van der Aalst, W. M. P., Weske, M. and Grünbauer, D., "Case Handling: A New Paradigm for Business Process Support", *Data and Knowledge Engineering*, Vol. 53, pp. 129-162, 2005.
9. Sadiq, S., Sadiq, W. and Orlowska, M., "Pockets of Exibility in Workflow Specifications", *In: ER'01*, pp. 513-526, 2001.
10. Pesic, M., Schonenberg, H., Sidorova, N. and van der Aalst, W. M. P., "Constraint-based Workflow Models: Change Made Easy", *CoopIs'07*, pp. 77-94, 2007.
11. Erol, K., Hendler, J. and Nau, D. S., *Semantics for Hierarchical Task-network Planning*, Technical Report CS-TR-3239, Computer Science Department, University of Maryland, 2004.
12. Hammond, K., *Case-Based Planning: Viewing Planning as a Memory Task*, Academic Press, 1989.
13. Velso, M.M ., *Learning by Analogical Reasoning in General Problem Solving*, Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, PA, 1992.
14. Casati, F., Ceri, S., Pernici, B. and Pozzi, G., "Workflow Evolution", *Data & Knowledge Engineering*, Vol. 24, pp. 211-238, 1998.
15. van der Aalst, W. M. P. and van Hee, K., *Workflow Management: Models, Methods, and Systems*, The MIT Press, 2002.
16. Cicekli, N. K. and Yildirim, Y. "Formalizing Workflows Using the Event Calculus", *LNCS*, Vol. 1873, pp. 222-231, 2000.
17. Kowalski, R. A. and Sergot, M. J., "Logic-based Calculus of Events", *New Generation Computing*, Vol. 4, pp. 67-95, 1986.
18. Shanahan, M. T., *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*, MIT Press, Cambridge, 1997.
19. van der Aalst, W. M. P., "Exterminating the Dynamic Change Bug: A Concrete Approach to Support Workflow Change", *Information Systems Frontier*, Vol. 3, No. 3, pp. 297-317, 2001.
20. Missiaen, L., Bruynooghe, M. and Denecker, M., "CHICA: A Planning System based on Event Calculus", *The Journal of Logic and Computation*, Vol. 5, No. 5, pp. 579-602, 1995.
21. Shanahan, M. T., "An Abductive Event Calculus Planner", *The Journal of Logic Programming*, Vol. 44, pp. 207-239, 2000.



이 회 정

1996년 한양대학교 산업공학과 학사
1998년 한국과학기술원 산업공학과 석사
2007년 한국과학기술원 산업공학과 박사
2003년 8월~2008년 8월 삼성전자 개발혁신팀 책임연구원
2008년 9월~현재 대구대학교 산업시스템공학과 전임강사
관심분야: Workflow Management, PLM

서 효 원

1981년 연세대학교 기계공학과 학사
1983년 한국과학기술원 기계공학과 석사
1991년 West Virginia University 산업공학과 박사
1983년~1987년 대우중공업(주) 중앙연구소 주임연구원
1992년~1995년 생산기술연구원 생산시스템센터 수석연구원
1996년~현재 한국과학기술원 산업공학과 교수
관심분야: CE/PDM/CPC/PLM, Workflow Management/BPM, Ontology/Knowledge Based System