Automatic Control of Workflow Processes Using ECA Rules

Joonsoo Bae, Hyerim Bae, Suk-Ho Kang, and Yeongho Kim

Abstract—Changes in recent business environments have created the necessity for a more efficient and effective business process management. The workflow management system is software that assists in defining business processes as well as automatically controlling the execution of the processes. This paper proposes a new approach to the automatic execution of business processes using Event-Condition-Action (ECA) rules that can be automatically triggered by an active database. First of all, we propose the concept of blocks that can classify process flows into several patterns. A block is a minimal unit that can specify the behaviors represented in a process model. An algorithm is developed to detect blocks from a process definition network and transform it into a hierarchical tree model. The behaviors in each block type are modeled using ACTA formalism. This provides a theoretical basis from which ECA rules are identified. The proposed ECA rule-based approach shows that it is possible to execute the workflow using the active capability of database without users' intervention. The operation of the proposed methods is illustrated through an example process.

Index Terms—Workflow management, ECA rules, active database, business process.

1 Introduction

For the last several years, companies have been experiencing many changes in their business environments. One is an internal change caused by the ever-increasing pressure for the need to satisfy various customer needs. In order to meet the diverse customer needs, companies may have to diversify their business processes. Another change faced by companies today is an external one resulting from the increase in strategic alliance and e-Business. This change compels a company to become involved in the business processes of other companies [2]. Not only have such internal and external changes caused for many new business processes to be created, but they have also increased the complexity of the processes.

The changes in business environments have created the necessity of technology and tools to ensure efficient and effective process management. As a result, there have been numerous attempts to enhance information systems towards providing advanced functions of process management beyond simple manipulation of independent tasks. A WorkFlow Management System (WFMS), a software tool to define, manage, and enact complex business processes [16], [23], [25], [31], [33], presents a new solution to the necessity of process management technology and tools [27].

Consider the business process presented in Fig. 1. This example shows a process of credit card application, which is composed of a number of activities, such as "application form filling" and "form scanning." A WFMS usually uses such a graphical representation to describe the business logic. The model represents the precedence relations among activities and some structural relations, such as activities proceeding in serial order or parallel. The representation also includes detailed specifications of activity, such as task performers, related documents, and necessary applications. More examples are presented in [2], [16], [23], [31].

A typical WFMS has a process design tool to create a process model and a workflow engine to control the execution of the model. In almost all the previous WFMSs, a process model has been translated into a format that can be understood by the workflow engine. In this kind of system, the workflow engine plays a key role in the execution and control of the process model. The system assures a coordinated progress of human activities, such as "application form filling" and "applicant information input" in the case of the example above, and automated activities that are carried out by application systems, such as "examining applicant's qualification" and "card issuance."

This paper presents a new approach to automatic execution of workflow processes. We propose a method of using Event-Condition-Action (ECA) rules in controlling business processes. The ECA rules are extracted from traditional process models, which can then be executed by the active capability of database. The issues to be discussed can be summarized as follows:

Classification of process patterns. Process flows are classified into several patterns, each of which, referred to as block in this paper, describes a certain behavior that is distinguished from other patterns.

Block detection. An algorithm is developed to detect blocks from a given process model.

- *J. Bae is with the Department of Industrial and System Engineering, Chonbuk National University, Jeonju, 561-756 Republic of Korea. E-mail: jsbae@hanmir.com.*
- *H. Bae is with the Department of Internet Business, Dongeui University, Busan, 614-714 Republic of Korea. E-mail: hrbae@dongeui.ac.kr.*
- *S.-H. Kang and Y. Kim are with the Department of Industrial Engineering, Seoul National University, Seoul, 151-742 Republic of Korea. E-mail: {shkang, yeongho}@cybernet.snu.ac.kr.*

Manuscript received 26 Nov. 2001; revised 31 Dec. 2002; accepted 6 June 2003.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 115433.

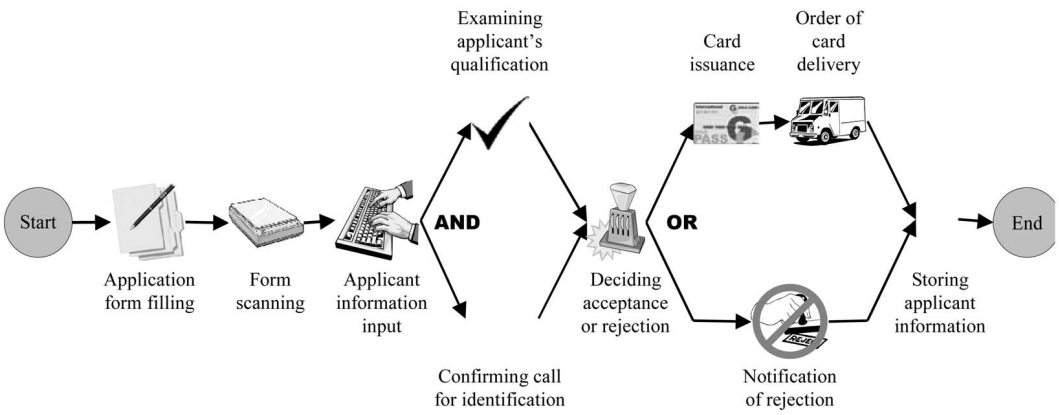

Fig. 1. An example business process.

Tree representation of process models. By reorganizing the blocks detected, a process model is transformed into a hierarchical tree representation.

Identification of ECA rules. For every block type, a set of ECA rules is defined, so that it is used by the active database in executing a process.

The concept of block forms the underlying basis of our approach. Dogac et al. [14] introduce this concept in WFMS for concurrently executing workflows in a distributed environment. They propose a block structured workflow specification language and develop a workflow scheduling mechanism. Our block definition is similar to the previous approach, but the differences are found in the automatic identification of blocks from process models and the use of ECA rules on the basis of the blocks.

This paper is organized as follows: Section 2 provides a review of literature on process modeling and application of ECA rules to WFMS. Block types are described in Section 3, and a block detection algorithm is presented in Section 4. In Section 5, a process model is transformed into a hierarchical tree model. Section 6 covers the ECA rules for workflow control. An operational example of the proposed methodology is presented in Section 7, followed by the summary and conclusions.

2 Related Literature

Our research is mainly concerned with process modeling and applications of ECA rules to WFMS. Related literature on each of the above is reviewed.

2.1 Workflow Process Model

Workflow is defined as "a business process that will be automatically executed by the computer," and a workflow management system is "a software system that defines a workflow, controls the execution and sequence of the defined workflow, and manages all the processes" [20]. For the last decade, much research work on the workflow, including [4], [5], [14], [15], [16], [21], [22], [24], [25], [31], [32], have been conducted.

A process is composed of a set of tasks, and each task has a specific objective that contributes somehow to attaining the process goal. The tasks progress following certain procedures that are usuallypredefined by a set of business rules. A process model defines the tasks to execute, their sequence, task performers, and work contents, and also specifies input and output conditions for each task [30].

This paper deals with the structural aspects of process models, that is, the precedence relationships determining the order of task execution and the task arrangement indicating sequential or parallel process flows. Other attributes, including task performers, required resources, work contents, execution time, etc. [20], are not considered in this paper. The structural aspect of process models is represented using a directed graph [1], [3], called process definition network.

An example process definition network is presented in Fig. 2. A circular node denotes a component task, and an arc connecting two nodes indicates their precedence relationships. For example, task T_2 should precede tasks T_3 and T_4. Some tasks are serially connected while others are parallelly connected. In the figure, tasks T_4, T_8, and T_{10} show an example of serial connection. As for parallel connection, task T_3 is split into T_5, T_6, and T_7, and these are merged into task T_9. Once the process is launched in WFMS, a task can start only after all of its preceding tasks are completed. The task states change during the process execution; that is, some are executed, while others are already completed or waiting to be executed. The state change model will be further described in a later section.

A graphical representation of process modelprovides a visual means through which users can have an easier understanding of the semantics of process models. However, it is not in a form that can be read by a machine

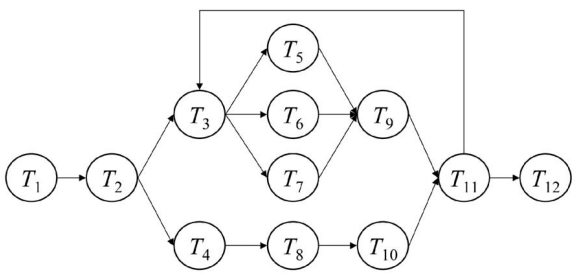

Fig. 2. Process definition network.

directly. Therefore, the graphical representation must be translated into a machine-readable language before launching it in a WFMS.

2.2 ECA Rules in Workflow

An ECA (Event-Condition-Action) model is originally used in active database systems [26], [34]. If an event occurs and a condition turns out to be true, then the active database executes a corresponding action. The event is a change of database contents, the condition is associated with a database query to check it, and the action corresponds to a set of statements that may trigger other changes in the database. All these are carried out automatically without any intervention from users or external applications.

It is an interesting approach to use ECA rules for controlling workflow processes. Dayal et al. [13] are those who have made one of the first attempts in applying ECA rules to WFMS. Casati et al. [8] consider various rules necessitated for workflow management at a conceptual level, and propose a classification of the rules. Casati et al. [7], [9], and Chiu et al. [10], [11] present a rule-based approach to exception handling in WFMS. Geppert et al. [17] implement a rule-based workflow engine. They maintain a list of event history, where an event is described in a logic-based form. The list records all the events that occur during workflow execution. By controlling the workflow, the workflow engine tries to match the history information with rules in a rule-base. Goh et al. [18] report the use of ECA rules to support workflows in product development.

The ECA rule has a sound theoretical basis. Once a set of ECA rules required for process execution is prepared, we can take advantage of the theoretical basis. However, it is not easy to visualize the meaning of the rules, unlike the graphical representation and, thus, it is very difficult for users to understand and manage the rules. In addition, the previous approaches require a considerable amount of manual efforts in generating the rules. In other words, the ECA rules entail many difficulties in dealing with complex processes. This is in fact the main reason why the ECA rule-based approach has not been a popular choice among commercial WFMSs.

We propose a method of combining graphical process representation and ECA rules. A graphical processmodel, though it is convenient for a human user to grasp the actual process, is not readily machine-readable. We transform the graphical model into a set of ECA rules, so that our workflow system is able to control its execution automatically. In order to do this, a systematic method of reducing a process model into a simple form is developed. This leads us to a formalization of process models that is suitable for ECA rule-based control. Existing approaches, however, do not provide any generic method of process simplification, and they cannot completely formalize process models with the ECA rules.

3 Block

A block is a unit of representation that can minimally specify the behavioral pattern of process flow. The behavioral patterns found in process models are classified into iterative, serial and parallel ones, each of which is illustrated in Fig. 3. Our discussion in this paper is confined to such networks that can be built by combining those patterns. Notice that the example network in Fig. 2 contains all the patterns in Fig. 3.

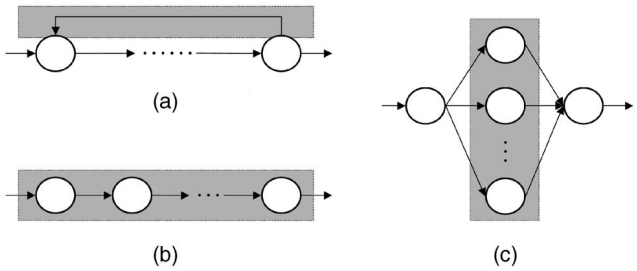

Fig. 3. Block types. (a) Iterative block. (b) Serial block. (c) Parallel block.

First, consider an iterative pattern, called iterative block. The iterative pattern forms a cycle as in Fig. 3a. Such a pattern only appears when some tasks can be carried out repeatedly. An iterative block is defined with start and end nodes, and iteration arc that directs from the start node to the end node. The iterative block in Fig. 2 starts at T_{11} and ends at T_3. The definition also includes an iteration condition that specifies when the iteration is needed. The condition is associated with the start node of iterative block.

Second, a serial pattern is shown in Fig. 3b. This pattern is simple in that it involves no iteration and has no split and merge in its task flow. The pattern must have at least two tasks that are connected consecutively, and each of the tasks has only one preceding task and only one succeeding task. Therefore, all the tasks should be executed consecutively. It is only after a task is completed successfully that the succeeding task can be started.

Finally, a parallel pattern is such a flow that a node splits into two or more branches, the branches proceed in parallel, and merge into a node. Fig. 3c is an illustration of this kind of pattern. The pattern is further subdivided into four types: AND, OR, POR, and COR-parallel. With an AND-parallel pattern, all of its component tasks are executed concurrently. Successful completion of all the tasks initiates the next task. If any task fails, the whole process fails. This last restriction is relaxed in OR-parallel pattern. If any parallel task succeeds, the following task begins. With a POR-parallel pattern, every task is associated with a priority, and the task with the highest priority is executed first. When this task fails, the task having the next highest priority is commenced. On the other hand, if a task succeeds, all the other tasks are ignored, and the following task is commenced. A COR-parallel pattern has some conditions on the branches. Only the task that meets the condition is executed. This pattern can represent exclusive OR split that has the condition that only one component task must be executed.

Although all the parallel patterns are different in terms of their semantics, they have the same graphical structure. This is because the graphical objects of nodes and arcs deal with only the split-and-merge relations of tasks. The semantics distinguishing the parallel patterns are usually specified on the split or merge nodes.

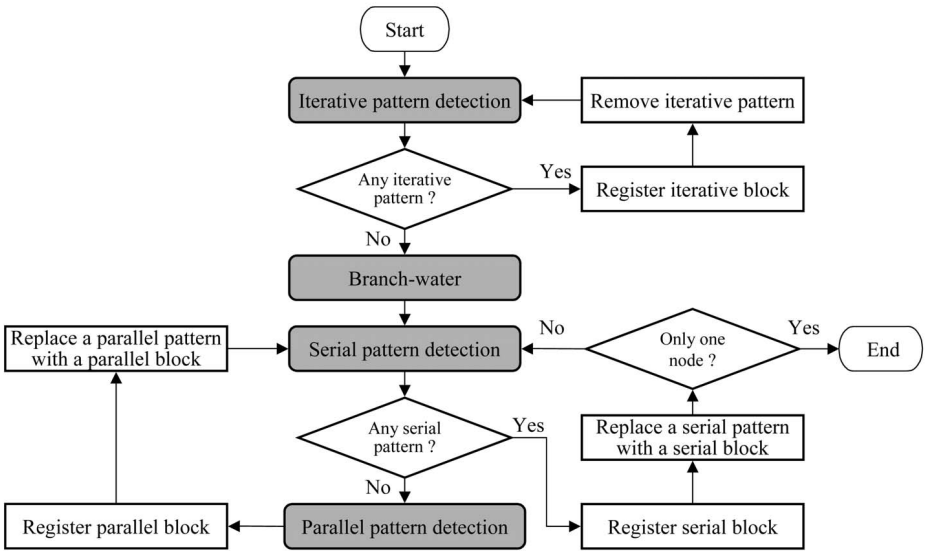

Fig. 4. Block detection algorithm.

4 Block Detection Algorithm

An algorithm is developed to detect blocks from a given process definition network. Listed below is the notation used in the algorithm.

- G: process definition network.
- N: the set of nodes in G.
- A: the set of arcs in G.
- v, v': a node in N.
- s: the start node of G.
- $pred(v)$: the set of nodes immediately preceding v.
- $succ(v)$: the set of nodes immediately succeeding v.
- $w(v)$: the water-level of v.
- $w\text{-}list$: the set of water-levels.

Fig. 4 shows the overall flow of the algorithm, where rounded rectangles indicate major procedures, each of which is further described in this section.

Fig. 5 is an illustrative example showing how our algorithm works. The algorithm first detects iterative blocks, like the one shown in Fig. 5a. This iterative block is registered and removed from the network. This leads the example network into the one in Fig. 5b. The branch-water procedure is used to simplify the remaining procedures, the details of which will be described later. Then, the algorithm identifies serial and parallel patterns. These two procedures may be alternated because the replacement of a block for several parallel tasks leads to a new serial pattern, and the replacement of a serial block for several serial tasks leads to a new parallel pattern. Figs. 5b, 5c, 5d, 5e, and 5f illustrate the alternating procedures.

The first procedure shown below, called iterative-block-detection, identifies iterative patterns in a given process definition network.

PROCEDURE Iterative-block-detection
(**in** G, **out** (the start node, the end node))
  QUEUE := {s};
  while (QUEUE $\neq \phi$) **do**
    let v be the first element of QUEUE;
    remove v from QUEUE;
    mark v;
    for (all $v' \in succ(v)$ **do**
      if (v' is marked) **then** return (v, v');
      if (all $pred(v')$ are marked) **then** append v' to QUEUE;
    end
  end
  return null;
end Iterative-block-detection

When there is an arc that turns back the process flow, it forms a directed cycle in a network. The iterative-block-detection procedure discovers such an arc and returns the arc's start and end nodes. The arc detected is registered as an iterative block with the start and end nodes and the iteration-condition specified on the start node. Then, the arc is removed from the network. This procedure is repeated until there is no more cycle.

Prior to detecting the other block types, our algorithm preprocesses the graph with the following branch-water procedure.

PROCEDURE Branch-water (**in** G, **out** $w\text{-}list$)
 for all v, **do** $w(v) := 0.0$;
 $w(s) := 1.0$; $w\text{-}list := \{1.0\}$; QUEUE := {s};
 while (QUEUE $\neq \phi$) **do**
  let v be the first element of QUEUE;
  remove v from QUEUE;
  mark v;
  for (all $v' \in succ(v)$) **do**
   $w(v') := w(v') + \frac{w(v)}{|succ(v)|}$;
   if ($w(v') \notin w\text{-}list$) **then** add $w(v')$ to $w\text{-}list$;
   if (all $pred(v')$ are marked) **then** append v' to QUEUE;
  end
 end
end Branch-water

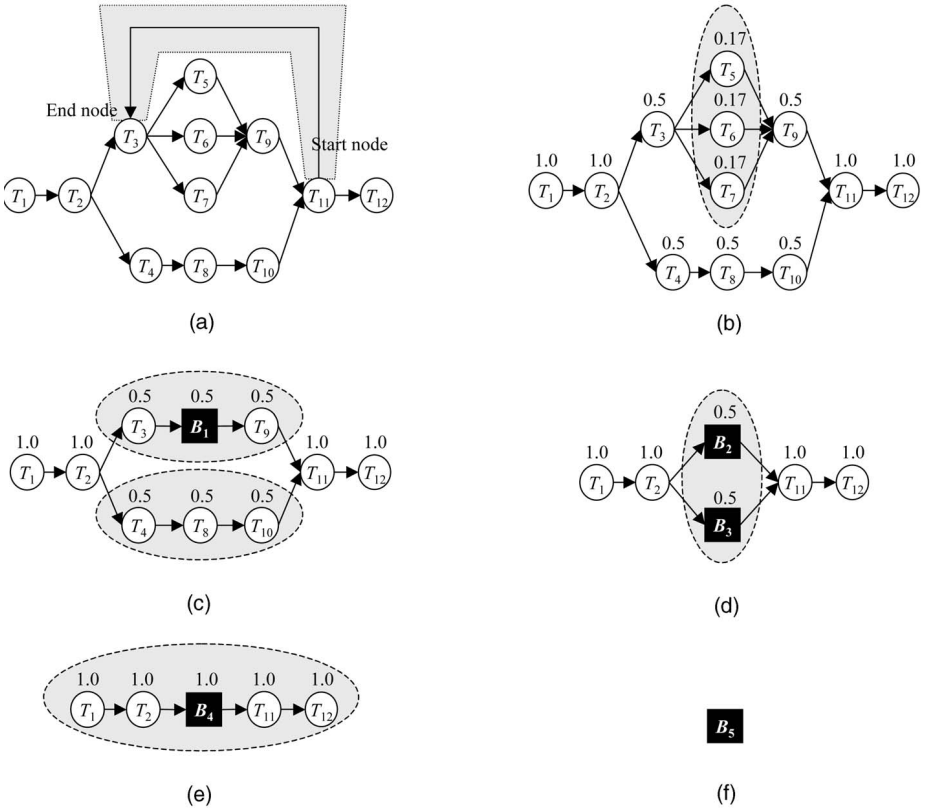

Fig. 5. Example application of block detection algorithm. (a) Iterative block. (b) Branch-water and parallel block. (c) Serial block. (d) Parallel block. (e) Serial block. (f) Final.

This procedure assigns a number to each node. We consider a process definition network as a networked pipeline, and water is poured into the pipeline. While the water flows through the pipeline network, it branches and merges in the network. The number, called water-level, indicates the level of water at every node. The procedure first assigns an initial number to the beginning node of the network. This number propagates through the arcs of the network as the water flows into the pipeline. Consider a node whose water-level is r. If the node is branched into k nodes, then r/k is propagated into each of its immediately succeeding nodes. A node's water-level is the sum of the numbers propagated from all of its immediately preceding nodes.

Using the water-levels, it becomes straightforward to identify the inner most block in the next two procedures. Consider the example presented in Fig. 5b, where the water-levels assigned to the nodes are indicated above every circle. It is clear that the parallel pattern (T_5, T_6, T_7) having the minimum water-level is the inner most block. Now, the algorithm alternates the search of serial patterns and parallel patterns.

The pseudo code presented below is the procedure of serial block detection.

PROCEDURE Serial-block-detection (**in** G, **out** $(SB, w\text{-}list)$)
$b := \min(w\text{-}list)$; LOOP := T; QUEUE := $\{s\}$;
while (LOOP $= T$) **do**
if (QUEUE $= \phi$) **then return** null;
let v be the first element of QUEUE;
remove v from QUEUE;
if $(w(v) = b \&\& |succ(v)| = 1 \;\&\&\; |pred(succ(v))| = 1)$ **then**
LOOP := F;
$SB :=$ SerialFrom(v);
w(SB) := w(v);
else append $succ(v)$ to QUEUE;
end
end Serial-block-detection

This procedure returns a set of nodes contained in a serial pattern. It starts the search at the beginning node of G and traverses the other nodes until the first node of serial block is identified. Once the first node is recognized, it is easy to identify the other nodes in the serial block because their in and out-degrees are all equal to 1 and they are connected consecutively starting from the first node. This is performed by SerialFrom in the above procedure. Since the algorithm uses the minimum water-level, it always finds the inner most serial pattern.

All the nodes in a serial pattern, each of which has the same water-level, are reduced to one serial block. Our algorithm registers the block and modifies the graph by replacing the nodes with the serial block. The new serial block's water-level is equal to its component nodes' water-level. If there is no further serial pattern, the algorithm proceeds to the parallel-block detection procedure as follows:

PROCEDURE Parallel-block-detection
(**in** G, **out** $(PB, w\text{-}list)$)

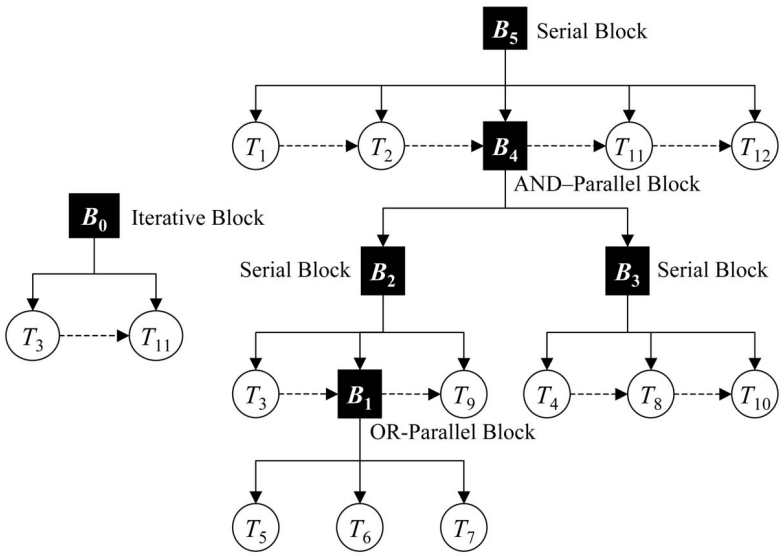

Fig. 6. Tree representation of process model.

$b := \min(w\text{-}list)$; LOOP $:= T$; QUEUE $:= \{s\}$;
while (LOOP $= T$) **do**
  let v be the first element of QUEUE;
  remove v from QUEUE;
  if $(w(v) = b)$ **then**
    LOOP $:= F$;
    $PB := succ(pred(v))$;
    w(PB) := w(b) $*$ |succ(pred(v))|;
    update $w\text{-}list$;
  else append $succ(v)$ to QUEUE;
end
end Parallel-block-detection

The parallel-block detection procedure again uses the minimum water-level to detect the inner most parallel pattern. The procedure's encountering a node with the minimum level indicates that there exists a parallel pattern. This is because all the serial patterns associated with the minimum level have already been detected previously. The procedure collects all the nodes that are parallel to the node. Our algorithm reduces those nodes to one parallel block, and registers it. To obtain the detailed classification, i.e., AND, OR, POR, and COR-parallel blocks, it can consult the split or merge conditions. Then, the algorithm proceeds to another round of serial block detection.

All the procedures are based on the well-known breadth-first search algorithm. This algorithm is modified taking into account the purpose of each procedure. Notice that the last two procedures always reduce the size of network, and finally making it into one block, which implies the end of the algorithm. The block detection algorithm is very efficient because its complexity is $O(m^2)$, where m is the number of arcs in the network [6].

5 Tree Representation of Process Model

In this research, a process definition network is restructured as a tree form. We adopt the concept of a nested process model in [21]. The nested process model is originally proposed to build up a process model in a top-down manner. A general process model is first created in a very abstract level, and then some of its activities are deployed into more specific subprocesses. The nested model provides several advantages over conventional flat ones, which includes the convenience in process modeling and the extensibility of process models. A more detailed discussion is found in [21].

With regard to the construction of process model, the tree representation in this paper shows a bottom-up approach. This is so in the sense that a complete process model is first created and transformed into a nested model. Tracing back the results of the block detection algorithm in the previous section gives the tree representation.

For example, Fig. 6 shows a tree representation of the process definition network in Fig. 5. The root node, B_5, is identical to the node shown in Fig. 5f. The node is expanded into the serial block of (T_1, T_2, B_4, T_{11}, T_{12}), which is shown in Fig. 5e. The dotted arrow indicates that the nodes are executed serially. The AND-parallel block, B_4, again spreads out two nodes, B_2 and B_3, both of which are serial blocks. Similarly branching out the blocks, we can obtain the tree representation in Fig. 6. The iterative block is registered separately from the tree.

In this representation, a process is an assembly of blocks and unit tasks that cannot be further broken down. Another interesting feature of the representation is that serial and parallel blocks appear alternately. Such a nested model makes it easy to control the process execution.

In the representation, the root node is an abstraction of a whole process. This means that execution of the whole process is identical to the execution of the components contained in the node. In our example, the root node (B_5) is a serial block composed of four unit tasks (T_1, T_2, T_{11}, T_{12}) and one parallel block (B_4). A method is needed to control the execution of the serial block. In the midst of controlling the serial block, a parallel block is encountered which requires a different control mechanism. The parallel block is then decomposed into two serial blocks (B_2,B_3). This

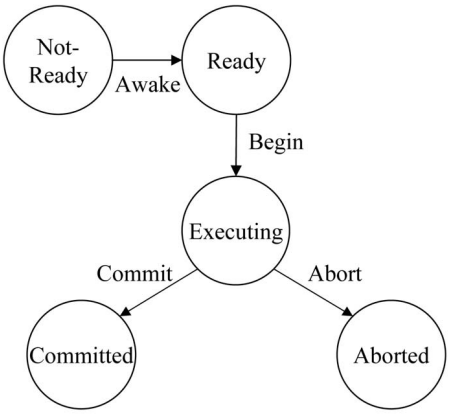

Fig. 7. State transition diagram of task processing.

continues until no block appears. If we have a method of automatic control for each block type, then the whole process can be controlled automatically. The control of blocks is described in the next section.

6 Derivation of ECA Rules

This section describes the ECA rules for every block type. An active database uses the ECA rules for automatic control of process execution.

6.1 Dependency Relations and ACTA Formalism

The basic mechanism of process control is often described with the state transition of component tasks [14], [19], [24], [28], [29]. Fig. 7 shows the state transition model used in this paper.

The state of a task at a point of time is either one of "Not-Ready," "Ready," "Executing," "Committed," and "Aborted." Every task is initialized as a "Not-Ready" state. The task state becomes "Ready" as soon as its preceding tasks are all completed. A task can be set up with some preconditions that should be satisfied before execution. When all the preconditions are satisfied, the task begins and the state changes to "Executing." A task without any precondition can be executed immediately. All tasks end with one of the two exclusive states, "Committed" and "Aborted." The labels, such as "Begin" and "Commit," on the arcs represent events that cause state transition as shown in the figure.

Since a block consists of a certain number of tasks as described in Section 3, the state of a task affects those of other tasks and its block. To represent the interrelations among tasks and a block, we provide dependency relations for different block types.

In a serial block, all the tasks are linearly interrelated, and the first task begins as soon as the block begins. If a preceding task commits, the next task begins executing. When the last task eventually commits, the entire block commits. While executing a block, if a task aborts, the entire block aborts. In this case, the results of preceding tasks that are already committed are required to be undone. For each of such tasks, we define an additional compensation task, which restores it to the previous states. An iterative block behaves similarly except it repeats till a certain condition meets.

In an AND-parallel block, all subtasks execute concurrently. If all tasks commit successfully, the block commits. In the case of any one of the tasks aborting, the block aborts after compensation tasks are executed for tasks that are already committed.

An OR-parallel block also starts with all tasks executing concurrently. However, committing of any task makes the entire block commit, and if all the tasks abort, the block aborts. A POR-parallel block and a COR-parallel block have the same condition for committing and aborting as an OR-parallel block. The two block types are different only in that all the tasks may not execute at the beginning of the block. In a POR-parallel block, tasks execute one after another in a predefined order, and in a COR-parallel block, only tasks that meet a certain condition can begin.

In this paper, blocks are organized hierarchically and a block contained in another block can be handled as a type of task. Therefore, we consider that a block and a compensation task are also kinds of tasks that describe the dependency relations.

To represent the interrelations of the state transitions, we adopt the ACTA (which means "actions" in Latin) formalism [8], [12]. The original ACTA formalism utilizes several predefined dependency relations to define the interrelations among transactions in database systems.

The dependency relations that are used are listed in Table 1. A state transition is triggered by an event and, thus, a dependency relation between tasks is defined as the relationship between their events. Consider, for example, "t_j CD t_i," which is commit dependency between t_j and t_i. This means that if both tasks t_i and t_j commit, then the commitment of t_i precedes the commitment of t_j. The symbol, $\Rightarrow$, denotes logical implication, and $<$ is a predicate representing a precedence relation between two events, i.e., $e < e'$ is true if event e precedes event e'. H is a finite set of all the events that have occurred during workflow execution. That is, $e \in H$ indicates that event e has already occurred. The dependency relations are used to express how a task state transition affects the state transition of other tasks. Then, ECA rules can be derived mechanically from the dependency relations, as addressed in the subsequent sections.

6.2 ACTA Formalism for Block Representation

For every block type, the dependency relations are represented using the ACTA formalism. Since one block type is different from the others in its structure and semantics, the representations of dependency relations are also different from each other. The dependency relations of each block type are presented in Fig. 8.

Consider, for example, the dependency contained in an AND-parallel block, which is shown in Fig. 8. In this block, a certain number of tasks are processed concurrently. Prior to processing the block, the block and each of its tasks has a begin dependency (BD), which means that every task can begin immediately after the block begins. This dependency is shown in the first diagram of the figure.

The second diagram shows the dependency relations after all the tasks in the block have begun. Notice that the block has a commit dependency (CD) with every task. Therefore, if there is any task that has not committed, the

TABLE 1
Dependency Relations in the ACTA Formalism

Dependency Relation	Name	Definition	Description
t_j CD t_i	Commit Dependency	(Commit $t_j \in H$) $\Rightarrow$ ((Commit $t_i \in H$) $\Rightarrow$ (Commit $t_i <$ Commit t_j))	If both t_i and t_j commit, then the commitment of t_i precedes the commitment of t_j.
t_j SCD t_i	Strong-Commit Dependency	(Commit $t_i \in H$) $\Rightarrow$ (Commit $t_j \in H$)	If t_i commits, then t_j commits.
t_j AD t_i	Abort Dependency	(Abort $t_i \in H$) $\Rightarrow$ (Abort $t_j \in H$)	If t_i aborts, then t_j aborts.
t_j WAD t_i	Weak-Abort Dependency	(Abort $t_i \in H$) $\Rightarrow$ ($\neg$(Commit $t_j <$ Abort t_i) $\Rightarrow$ (Abort $t_j \in H$))	If t_i aborts and t_j has not yet committed, then t_j aborts.
t_j BD t_i	Begin Dependency	(Begin $t_j \in H$) $\Rightarrow$ (Begin $t_i <$ Begin t_j)	Task t_j cannot begin executing until t_i has begun.
t_j BCD t_i	Begin-on-Commit Dependency	(Begin $t_j \in H$) $\Rightarrow$ (Commit $t_i <$ Begin t_j)	Task t_j cannot begin executing until t_i commits.
t_j BAD t_i	Begin-on-Abort Dependency	(Begin $t_j \in H$) $\Rightarrow$ (Abort $t_i <$ Begin t_j)	Task t_j cannot begin executing until t_i aborts.
t_j CAD t_i	Commit-on-Abort Dependency	(Abort $t_i \in H$) $\Rightarrow$ (Commit $t_j \in H$)	If t_i aborts, then t_j commits.
t_j WCD t_i	Week-Begin-on-Commit Dependency	(Begin $t_j \in H$) $\Rightarrow$((Commit $t_i \in H$) $\Rightarrow$(Commit $t_i <$ Begin t_j))	If t_i commits, t_j can begin executing after t_i commits.

block cannot commit. The abort dependency (AD) specifies that aborting any task causes the block to also abort. If the block aborts, all tasks that have not yet committed will abort by weak abort dependency (WAD).

In the diagram, task T_i is associated with a new task CT_i, called compensation task. Once T_i begins, CT_i is set up internally. Suppose that a task is committed but the AND-parallel block encompassing the task is aborted. Since in this case the commitment of the task has nothing to do with the successful completion of the overall process, it may need to withdraw the commitment of the task. CT_i carries out such compensation actions. The begin-on-commit (BCD) dependency between a task and its compensation task indicates that the compensation task cannot be processed unless the task commits.

When one task (say, task T_1, without loss of generality) commits, the dependency relations become as illustrated in the third diagram. Then, T_1 can no longer affect the state of the block. However, the compensation task, CT_1, establishes begin-on-abort dependency (BAD) and commit-on-abort dependency (CAD) with the block. This implies that the compensation task become effective when the block aborts. Whenever a task commits, the dependency relations are modified in a similar manner.

After all the tasks commit, the block immediately commits by strong-commit dependency (SCD), as illustrated in the fifth diagram. If the block aborts at any time during this procedure, it triggers the compensation operations for committed tasks. The order of processing the compensation is reversed to that of the committing tasks. This is represented by weak-begin-on-commit dependency (WCD), as illustrated in the fourth and fifth diagrams.

In this paper, we assume that all tasks are compensatable. However, this is not the case in the real world. It is impossible to return to the original state once an activity changes or reforms something physically, such as stamping documents and sending postal mail. In another case, the compensation would be difficult if the workflow system invokes an application system. The compensation considered in this paper is confined within the compensation of the task states stored in the workflow database. For such noncompensatable tasks, the workflow system may send notification of the compensation to the corresponding task performer or application system.

Compensation for blocks is a little more complex because a block, with the exception of the root node, is always nested to another block. The compensation task for a nested block activates, recursively the compensation for its component tasks. For example, consider a block A that is an AND-parallel block containing components $\{a1, a2, B, a3\}$, and its nested block B that is a serial block containing components $\{b1, b2\}$. Suppose that $a1$ and B are COMMITTED, $a2$ EXECUTING, and $a3$ READY. If A is aborted, the compensation tasks of $a1$ and B are activated and the compensation of block B immediately activates the compensation tasks of $b1$ and $b2$.

6.3 Derivation of ECA Rules

From the dependency relations represented in ACTA formalism, ECA rules are extracted for each block type. An ECA rule is composed of event, condition, and action. The generation of a rule is equivalent to identifying these three elements.

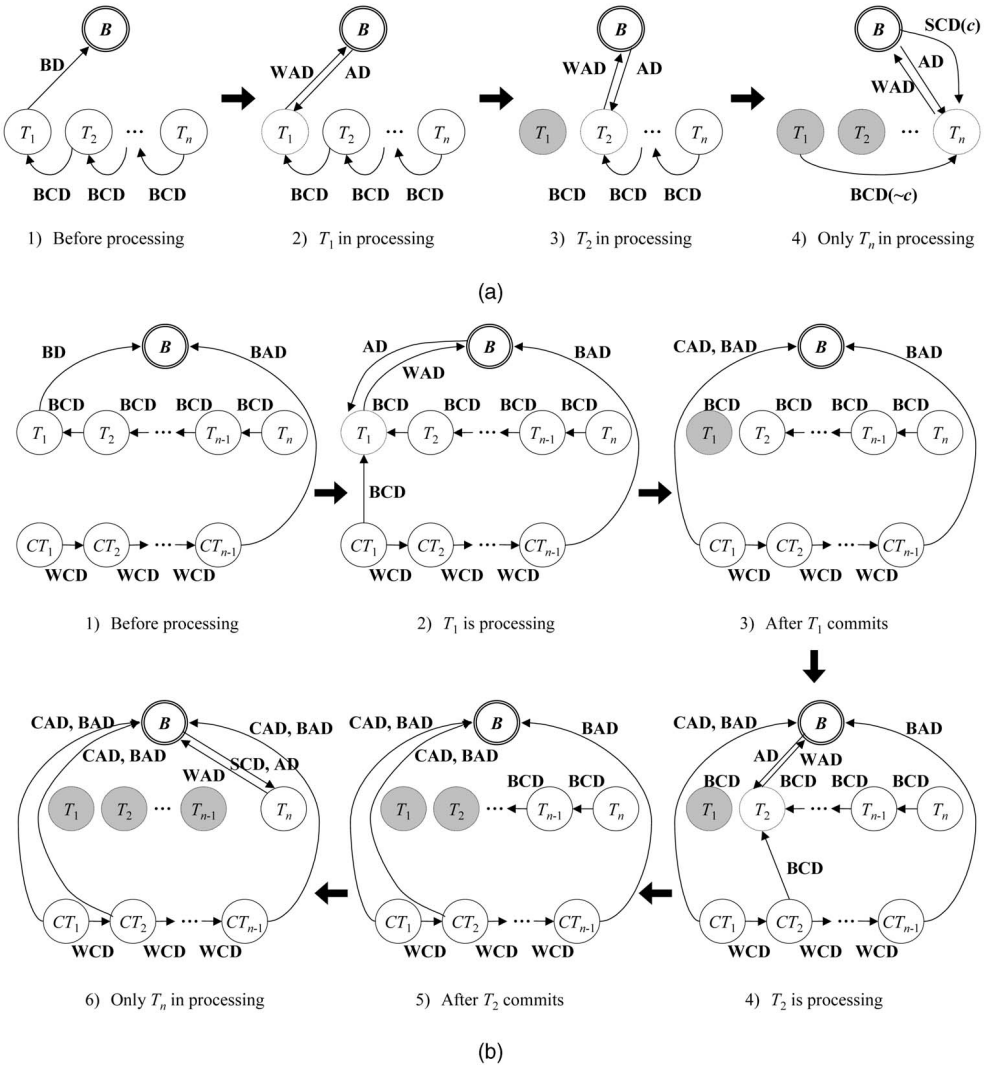

Fig. 8. ACTA formalism for each block type. (a) Iterative block. (b) Serial block.

An ECA rule is implemented as a stored procedure. Fig. 9 presents an example of ECA rule implemented in Oracle 8i. The stored procedure, which is actually a program, is made up of three components: event, condition, and action, each of which corresponds to that in ECA rules. The event is a "triggering statement," which specifies events that the procedure can detect automatically. The events are table operations, such as insert, delete, and update. The condition is "trigger condition," which is a logical expression that must be satisfied in order to activate the action part. The action is "triggered statement," which involves SQL statements and codes to be executed.

The example ECA rule is an implementation of R15 (commit_block_by_all_tasks). This rule implements the CD and SCD dependency relations in the AND-parallel block. When a commit event occurs at a task in an AND-parallel block, this is recorded into a database table. (In our prototype implementation, the table is WF_TASKINST.) This update operation triggers R15 immediately. The event part of the rule states that "event" is an update of ComponentStatus in WF_TASKINST. The active database detects this event, and then it checks the condition part. The condition part says that the block type should be AND-parallel and the updated ComponentStatus be "Committed." Finally, the action part changes the block state (BlockStatus) to "Committed" when the number of "Committed" components of the current block instance is equal to the number of block components. Other rules are implemented similarly, and are summarized in Table 2.

7 Prototype Implementation and Operational Example

We have implemented a prototype of WFMS. Fig. 10 presents a simplified architecture of the system. The active database carries out the role of traditional workflow engine. The tables having stored procedures are shown in the active database. When the database detects an event in any of the tables, it identifies and triggers a relevant rule taking into account the corresponding block type and task states. While controlling a process instance, the database communicates with users and application programs through the external event manager. Listed below are the external events that they exchange.

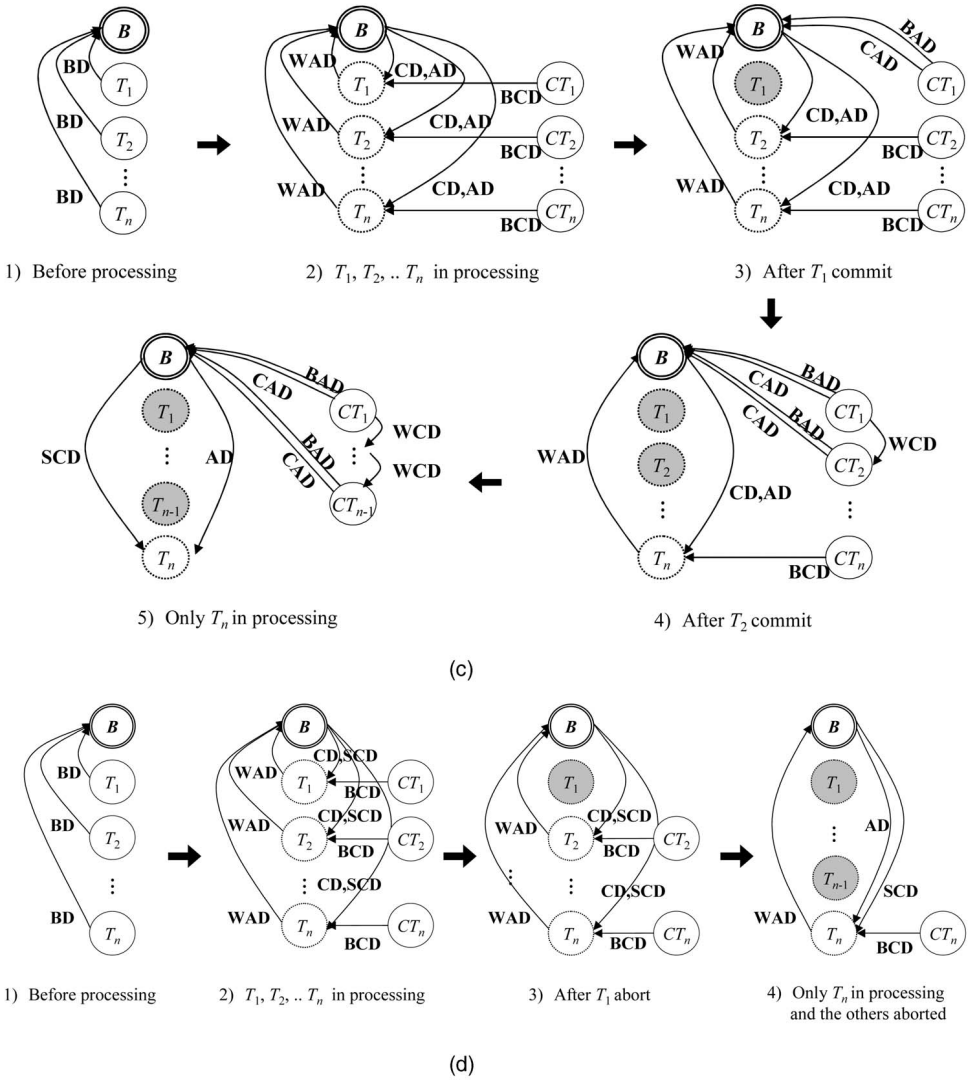

Fig. 8 (continued). ACTA formalism for each block type. (c) AND-parallel block. (d) OR-parallel block.

- START(Process Instance): This event occurs when a client launches a process instance.
- DELIVER(Task): This is an event that assigns a unit task to a task performer that can be a human user or standalone program.
- SUCCESS(Task): This event indicates a client's successful completion of task.
- FAIL(Task): This event indicates that a user or application program has failed in completing a task.

In our system, WF_EVENT maintains those external events. WF_BLOCKINST and WF_TASKINST maintain the current state information of blocks and tasks, respectively. While processing a rule, the tables interact with each other. In the example of ECA rule in the previous section, the action part changes the block state in WF-BLOCKINST. This triggers another rule defined in the table. Such chains of rule triggering combined with event detection controls the process execution. The necessary ECA rules for each table are listed in the three tables in Fig. 10.

Fig. 10 also shows a sequence of rule applications for the illustrative example presented in Fig. 11a. The example demonstrates how the proposed approach works. Applying our block detection algorithm to the example, identifies one parallel block and one serial block. Then, the network is transformed into the tree representation in Fig. 11b. Once this process model is stored in a database, an authorized user can use it whenever the user needs to launch an actual process instance following the definition.

Upon launching the process instance, our system first picks up the tree's root node (B_2). This node is a serial block and, thus, rules for serial block are applied to it. The serial block is composed of three tasks, $B_2.t_1$, $B_2.t_2$, and $B_2.t_3$ (which denote T_1, B_1, and T_4, respectively), as shown in Fig. 11c. While executing the block, it encounters another block, B_1, which has two parallel tasks $B_1.t_1$ and $B_1.t_2$ (which denote T_2 and T_3, respectively) as shown in Fig. 11d. To control the execution of B_1, we need to apply the rules for AND-parallel block.

8 Summary and Conclusions

We propose an ECA rule-based WFMS that makes it possible to execute business processes using an active database. The existing approaches to adopting ECA rules in WFMSs use the rules to manage exceptional situations or

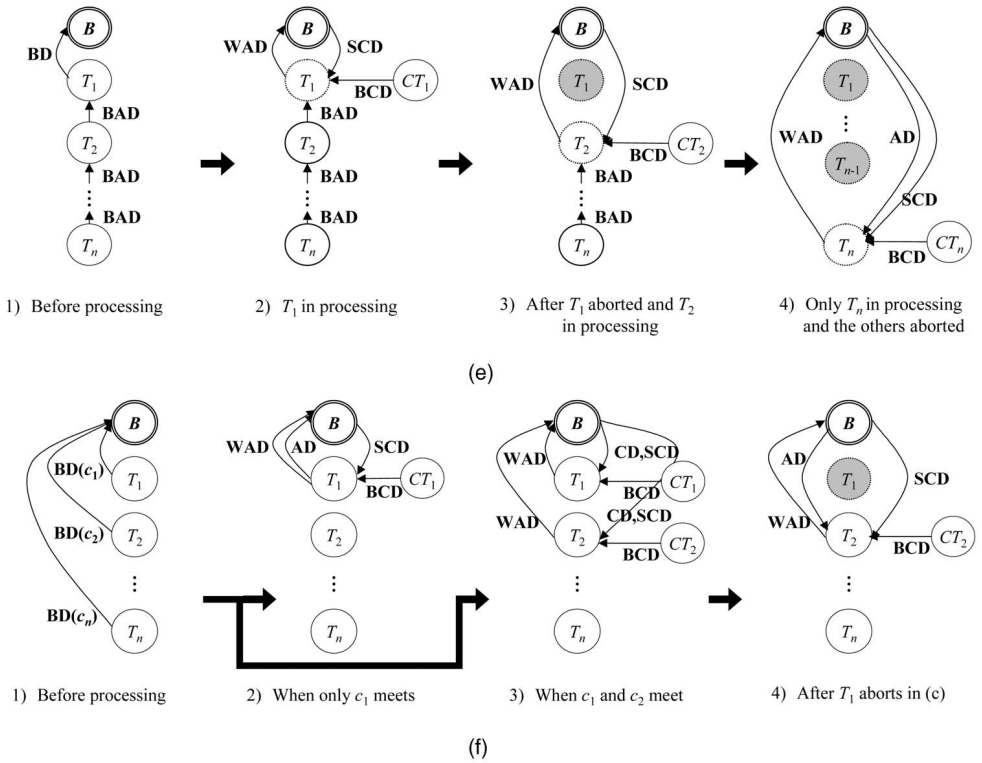

Fig. 8 (continued). ACTA formalism for each block type. (e) POR-parallel block. (f) COR-parallel block.

Event

```
CREATE OR REPLACE TRIGGER Commit_block_by_all_tasks
AFTER UPDATE OF ComponentStatus ON WF_TASKINST
FOR EACH ROW
```

Condition

```
WHEN (new.BlockType = 'AND-parallel') AND (new.ComponentStatus = 'Committed')
```

Action

```
DECLARE
    noOfComponent       NUMBER(3);
    noOfCommitted       NUMBER(3);
    blockinst_var       WF_BLOCKINST%rowtype;
BEGIN
    blockinst_var.ProcessID := :new.ProcessID;
    blockinst_var.ProcessInstID := :new.ProcessInstID;
    SELECT MAX(ComponentSequence) INTO noOfComponent FROM WF_BLOCK
        WHERE   ProcessID = :new.ProcessID AND
                BlockID = blockinst_var.BlockID;
                    /* Get the number of  block components from WF_BLOCK. */
    SELECT MAX(ComponentSequence) INTO noOfCommitted FROM WF_BLOCKINST
        WHERE   ProcessID = :new.ProcessID AND
                BlockID = blockinst_var.BlockID;
               /* Get the number of committed components from WF_BLOCKINST. */
    IF noOfComponent = noOfCommitted THEN
                                        /* Check if those two numbers are equal. */
        UPDATE  WF_BLOCKINST SET BlockStatus = 'Committed'
        WHERE   ProcessID = :new.ProcessID AND
                ProcessInstID = :new.ProcessInstID AND
                TaskID = :new.TaskID;
                    /* Update the status of the current block into 'Committed'.*/
    END IF;
END;
```

Fig. 9. An implementation example of ECA rules.

domain-specific rules to deal with task contents. This means that the approaches cannot be used as a general method of process control. The method proposed in this paper could fully replace existing workflow engines.

Our original contributions are as follows: First, we propose the use of blocks that can classify process flows into several patterns. The blocks become the basic unit of representing process models and identifying ECA rules.

TABLE 2
ECA Rules for Each Block Type

	Dependency	Event	Condition	Action	ID
Common	(EXTERNAL)	START(PI)		Begin B	R1
	(EXTERNAL)	SUCCESS(t_i)	Begin $t_i \in H$	Commit t_i	R2
	(EXTERNAL)	FAIL(t_i)	Begin $t_i \in H$, Commit $t_i \notin H$	Abort t_i	R3
	(INTERNAL)	Begin t_i	$t_i = B$	Begin B	R4
	(INTERNAL)	Commit B	$B = t_i$	Commit t_i	R5
	t_i WAD B	Abort B	Abort $B \in H$, Commit $t_i \notin H$	Abort t_i	R11
	ct_i BAD B ct_i BCD t_i	Abort B	Abort $B \in H$, Commit $t_i \in H$	Begin ct_i	R13
Serial	t_1 BD B	Begin B	Begin $B \in H$	Begin t_1	R6
	B AD t_i	Abort t_i	Abort $t_i \in H$	Abort B	R9
	t_{i+1} BCD t_i	Commit t_i	Commit $t_i \in H$	Begin t_{i+1}	R12
	ct_i WCD ct_{i+1} ct_i BCD t_i	Commit ct_{i+1}	Commit $ct_{i+1} \in H$, Commit $t_i \in H$, Abort $B \in H$	Begin ct_i	R14
	B SCD t_n	Commit t_n	Commit $t_n \in H$	Commit B	R16
AND-parallel	t_i BD B	Begin B	Begin $B \in H$	Begin t_i	R7
	B CD t_i B SCD t_n	Commit t_i	$\forall i$ (i=1,2,…,n), Commit $t_i \in H$	Commit B	R15
	B AD t_i	Abort t_i	Abort $t_i \in H$	Abort B	R9
	ct_i WCD ct_{i+1} ct_i BCD t_i	Commit ct_{i+1}	Commit $ct_{i+1} \in H$, Commit $t_i \in H$, Abort $B \in H$	Begin ct_i	R14
OR-parallel & COR-parallel	t_i BD B	Begin B	Begin $B \in H$	Begin t_i	R7
	B AD t_i	Abort t_i	$\forall j$ (j =1, …, n), Abort $t_j \in H$	Abort B	R10
	B SCD t_i	Commit t_i	Commit $t_i \in H$	Commit B	R16
	t_i BD(c_j) B	Begin B	Begin $B \in H$, c_j=True	Begin t_i	R8
POR-parallel	t_1 BD B	Begin B	Begin $B \in H$	Begin t_1	R6
	t_{i+1} BAD t_i	Abort t_i	Abort $t_i \in H$	Begin t_{i+1}	R18
	B SCD t_i	Commit t_i	Commit $t_i \in H$	Commit B	R16
	B AD t_n	Abort t_n	$\forall j$ (j=1,…,n), Abort $t_j \in H$	Abort B	R10
Iterative	t_1 BD B	Begin B	Begin $B \in H$	Begin t_1	R6
	t_{i+1} BCD t_i	Commit t_i	Commit $t_i \in H$	Begin t_{i+1}	R12
	B AD t_i	Abort t_i	Abort $t_i \in H$	Abort B	R9
	B SCD(c) t_n	Commit t_n	Commit $t_n \in H$, c= True	Commit B	R17
	t_1 $BCD(\neg c)$ t_n	Commit t_n	Commit $t_n \in H$, c= False	Begin t_1	R19

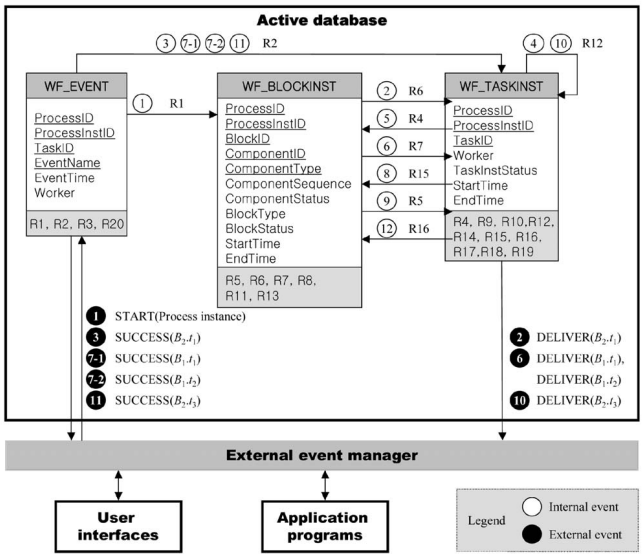

Fig. 10. System architecture and sequence of rule applications.

Second, an algorithm is developed to identify the blocks from process definition networks. Third, a flat network is transformed into a hierarchical tree representation using the blocks. This allows us to take the advantage of modularity in controlling the processes. Finally, for each block type, the control logic is modeled using ACTA formalism. This provides a theoretical basis for using ECA rules. Overall, our approach can become a basis for purely rule-based WFMS. Since most of the recent DBMSs possess active capability, the proposed method can be installed anywhere a DBMS is available.

There are several further research issues to be dealt with. Although the rules in our current approach deal with only the structural aspects of process models, it can be extended to domain specific rules. Another interesting issue is at generalizing the block types. In some process definition networks, two or more patterns may be interlinked. For example, one task in a parallel pattern is connected to another task in other parallel patterns. The current approach cannot be applied to such complicated cases. In addition, it is important to evaluate the efficiency of the

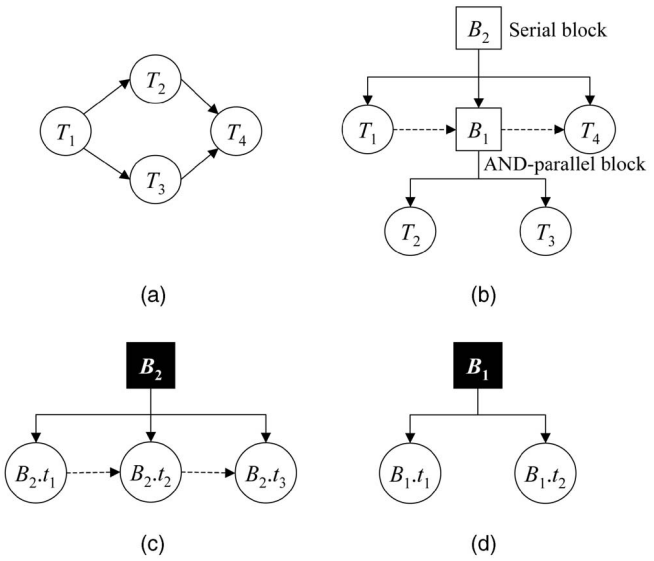

Fig. 11. Illustrative example. (a) Process definition network. (b) Tree representation. (c) Serial block. (d) AND-parallel block.

proposed approach in order to see how many workflows can be run and controlled at the same time.

ACKNOWLEDGMENTS

This research was partly supported by the program of the National Research Laboratory granted from the Korea Institute of Science and Technology Evaluation and Planning. This work was also supported by grant no. R01-2002-000-00155-0 from the Basic Research Program of the Korea Science and Engineering Foundation. The authors would like to thank the anonymous referees for their helpful and constructive comments.

REFERENCES

[1] W.M.P. van der Aalst, "Formalization and Verification of Event-Driven Process Chains," *Information and Software Technology,* vol. 41, no. 10, pp. 639-650, July 1999.

[2] W.M.P. van der Aalst, "Process-Oriented Architectures for Electronic Commerce and Interorganizational Workflow," *Information Systems,* vol. 24, no. 8, pp. 639-671, Dec. 1999.

[3] W.M.P. van der Aalst and A.H.M. ter Hofstede, "Verification of Workflow Task Structures: A Petri-Net-Based Approach," *Information Systems,* vol. 25, no. 1, pp. 43-69, 2000.

[4] G. Alonso, C. Hagen, D. Agrawal, A.E. Abbadi, and C. Mohan, "Enhancing the Fault Tolerance of Workflow Management Systems," *IEEE Concurrency,* vol. 8, no. 3, pp. 74-81, July 2000.

[5] I.B. Arpinar, U. Halici, S. Arpinar, and A. Dogac, "Formalization of Workflows and Correctness Issues in the Presence of Concurrency," *Distributed and Parallel Databases,* vol. 7, no. 2, pp. 199-248, 1999.

[6] J.S. Bae, "Automatic Enactment of Workflow Process Using Active Databases," PhD dissertation, Seoul Nat'l Univ., Seoul, Korea, 2000.

[7] F. Casati, S. Castano, M. Fugini, I. Mirbel, and B. Pernici, "Using Patterns to Design Rules in Workflows," *IEEE Trans. Software Eng.,* vol. 26, no. 8, pp. 760-785, Aug. 2000.

[8] F. Casati, S. Ceri, B. Pernici, and G. Pozzi, "Deriving Active Rules for Workflow Enactment," *Proc. Seventh Int'l Conf. Database and Expert Systems Applications,* pp. 94-110, 1996.

[9] F. Casati, M. Fugini, and I. Mirbel, "An Environment for Designing Exceptions in Workflows," *Information Systems,* vol. 24, no. 3, pp. 255-273, May 1999.

[10] D.K.W. Chiu, Q. Li, and K. Karlapalem, "A Meta Modeling Approach to Workflow Management Systems Supporting Exception Handling," *Information Systems,* vol. 24, no. 2, pp. 159-184, Apr. 1999.

[11] D.K.W. Chiu, Q. Li, and K. Karlapalem, "Web Interface-Driven Cooperative Exception Handling in ADOME Workflow Management System," *Information Systems,* vol. 26, no. 2, pp. 93-120, Apr. 2001.

[12] P.K. Chrysanthis and K. Ramamritham, "ACTA: The SAGA Continues," *Database Transaction Models for Advanced Applications,* pp. 354-397, Morgan Kaufmann, 1995.

[13] U. Dayal, M. Hsu, and R. Ladin, "Organizing Long-Running Activities with Triggers and Transactions," *Proc. 1990 ACM SIGMOD Conf.,* pp. 204-214, 1990.

[14] A. Dogac, E. Gokkoca, S. Arpinar, P. Koksal, I. Cingil, B. Arpinar, N. Tatbul, P. Karagoz, U. Halici, and M. Altinel, "Design and Implementation of a Distributed Workflow Management System: METUFlow," *Proc. NATO Advanced Study Institute (ASI) Workshop Workflow Management Systems and Interoperability,* pp. 61-66, Aug. 1997.

[15] D. Georgakopoulos, M. Hornick, and F. Manola, "Customizing Transaction Models and Mechanisms in a Programmable Environment Supporting Reliable Workflow Automation," *IEEE Trans. Knowledge and Data Eng.,* vol. 8, no. 4, pp. 630-649, Aug. 1996.

[16] D. Georgakopoulos, M. Hornick, and A. Sheth, "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure," *Distributed and Parallel Databases,* vol. 3, no. 2, pp. 119-153, 1995.

[17] A. Geppert, D. Tombros, and K.R. Dittrich, "Defining the Semantics of Reactive Components in Event-Driven Workflow Execution with Event Histories," *Information Systems,* vol. 23, nos. 3/4, pp. 235-252, May 1998.

[18] A. Goh, Y.-K. Koh, and D.S. Domazet, "ECA Rule-Based Support for Workflows," *Artificial Intelligence in Eng.,* vol. 15, no. 1, pp. 37-46, 2001.

[19] C. Hagen and G. Alonso, "Exception Handling in Workflow Management Systems," *IEEE Trans. Software Eng.,* vol. 26, no. 10, pp. 943-958, Oct. 2000.

[20] D. Hollingsworth, "The Workflow Reference Model," Technical Report WFMC-TC-1003, 1.1, Workflow Management Coalition, Brussels, 1994.

[21] Y. Kim, S.H. Kang, D.S. Kim, J.S. Bae, and K.J. Joo, "WW-Flow: A Web-Based Workflow Management System Supporting Run-Time Encapsulation," *IEEE Internet Computing,* vol. 4, no. 3, pp. 55-64, May/June 2000.

[22] A. Kumar and J.L. Zhao, "Dynamic Routing and Operational Controls in Workflow Management Systems," *Management Science,* vol. 45, no. 2, pp. 253-272, 1999.

[23] G. Mentzas, C. Halaris, and S. Kavadias, "Modelling Business Processes with Workflow Systems: An Evaluation of Alternative Approaches," *Int'l J. Information Management,* vol. 21, no. 2, pp. 123-135, Apr. 2001.

[24] J. Miller, D. Palaniswami, A. Sheth, K. Kochut, and H. Singh, "WebWork: METEOR's Web-Based Workflow Management System," *J. Intelligent Information Systems,* vol. 10, no. 2, pp. 185-215, 1998.

[25] C. Mohan, "Tutorial: State of the Art in Workflow Management System Research and Products," *Proc. Fifth Int'l Conf. Extending Database Technology,* Mar. 1996, and *Proc. ACM SIGMOD Int'l Conf. Management of Data,* June 1996.

[26] N.W. Paton, *Active Rules in Database Systems.* Springer-Verlag, 1998.

[27] M. Pérez and T. Rojas, "Evaluation of Workflow-Type Software Products: A Case Study," *Information and Software Technology,* vol. 42, no. 7, pp. 489-503, May 2000.

[28] J. Puustjarvi, H. Tirri, and J. Veijalainen, "Reusability and Modularity in Transactional Workflows," *Information Systems,* vol. 22, nos. 2/3, pp. 101-120, Apr. 1997.

[29] M. Rusinkiewicz and A. Sheth, "Specification and Execution of Transactional Workflows," *Modern Database Systems: The Object Model, Interoperability, and Beyond,* pp. 592-620, Addison-Wesley, 1995.

[30] C. Schlenoff, A. Knutilla, and S. Ray, "Unified Process Specification Language: Requirements for Modeling Process," *Nat'l Inst. of Standards and Technology,* 1996.

[31] G. Shegalov, M. Gillmann, and G. Weikum, "XML-Enabled Workflow Management for E-Services Across Heterogeneous Platforms," *The Very Large Databases J.,* vol. 10, pp. 91-103, 2001.
[32] A. Sheth and K. Kochut, "Workflow Applications to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration Systems," *Proc. NATO Advanced Study Institute (ASI) Workshop Workflow Management Systems and Interoperability,* Aug. 1997.
[33] S.K. Shrivastava and S.M. Wheater, "Workflow-Management Systems," *IEEE Concurrency,* vol. 7, no. 3, pp. 16-17, July-Sept. 1999.
[34] C.W. Tan and A. Goh, "Implementing ECA Rules in an Active Database," *Knowledge-Based Systems,* vol. 12, no. 4, pp. 137-144, Aug. 1999.

Joonsoo Bae received the PhD, MS, and BS degrees in industrial engineering from Seoul National University, Korea in 2000, 1995, and 1993, respectively. He also completed the advanced software engineering program in the School of Computer Science at Carnegie Mellon University at 2002. He is an assistant professor in the Department of Industrial and System Engineering at Chonbuk National Univertsity. He was with the SCM and CRM Department of LG-EDS as a technical consultant from 2000 to 2002. He is interested in the areas of system design in manufacturing field, system integration in management information system, and e-business technology. His research topics include control of business processes using workflow systems, e-business security, process improvement of software engineering, and supply chain analysis.

Hyerim Bae received the PhD, MS, and BS degrees in industrial engineering from Seoul National University. He is an assistant professor in the Department of Internet Business at Dongeui University. He was a manager of the Information Planning Team at Samsung Card Corporation before joining the Dongeui University. His research interests include information system design, engineering databases, Web-based application development, workflow management system, and multiorganizational application integration. His recent research topics are document version control, XML data management, and collaborative process integration in e-commerce.

Suk-Ho Kang received the BS degree in physics from Seoul National University, the MS degree from the University of Washington, and the PhD degree from Texas A&M University, both in industrial engineering. He is a professor in the Industrial Engineering Department at Seoul National University. His research interests are in intelligent manufacturing systems and B2B e-commerce.

Yeongho Kim received the PhD degree from North Carolina State University and the BS and MS degrees from Seoul National University. He is an associate professor in the Department of Industrial Engineering at Seoul National University. He is interested in the areas of Internet applications, concurrent engineering and collaborative design, engineering databases, and AI applications in manufacturing. His research topics cover a wide variety of areas including workflow management systems, product data management, and computer supported collaborative work on the Internet. He is a member of the editorial board of the *International Journal of Industrial Engineering—Applications and Practice*, the *International Journal of Management Systems*, and the *International Journal of CAD/CAM*.