ELSEVIER

Available online at www.sciencedirect.com
SCIENCE DIRECT®

Decision Support Systems 37 (2004) 49–69

Decision Support Systems

www.elsevier.com/locate/dsw

Consulting past exceptions to facilitate workflow exception handling

San-Yih Hwang[a,*], Jian Tang[b]

[a] Department of Information Management, National Sun Yat-Sen University, Kaohsiung 80424, Taiwan, ROC
[b] Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada A1B 3X5

Accepted 30 September 2002

Abstract

In this paper, we propose an architecture model that deals with both expected and unexpected exceptions in the context of workflow management. Expected exceptions and their handling approaches are specified by ECA rules, while cases of unexpected exceptions are characterized by their features and resolution approaches. The handling of unexpected exceptions is then assisted by the system providing information about how recent similar cases were resolved. The ways in which the previous exception cases were handled provides useful information in determining how to handle the current one. Quantifying the similarity of exception cases is described, and three algorithms for efficiently searching for similar exception cases are proposed and evaluated both theoretically and by experimenting with synthetic data sets.
© 2002 Elsevier B.V. All rights reserved.

Keywords: Workflow exceptions; Workflow management; Exception handling; Similarity matching

1. Introduction

Workflow management systems (WFMSs) support the execution of business processes. A business process has a market-centred aim of fulfilling a business contract or satisfying a customer's needs [13], and typically, it is controlled by many factors, including the description of the constituent activities, their control/data flow, the potential participants, the organization model and the referenced data [31]. A WFMS separates the specification of business processes, or so-called workflow types, from their execution and provides a convenient and powerful means of specifying a business process and controlling its executions. However, it is well recognized that defining a workflow that represents all properties of the underlying business process is difficult [10]. Moreover, since the formulation of business processes occurs at a high conceptual level, workflows have to adapt rapidly to a changing environment, resulting in executions that deviate from the predefined plan.

The process model defined at specification simply represents a standard case, and WFMSs should provide the flexibility to support run-time modification to the defined workflows so as to handle non-standard cases, or so-called exceptions. Some types of exceptions are expected because they are known to occur occasionally or periodically, and their char-

* Corresponding author.
E-mail address: syhwang@mis.nsysu.edu.tw (S.-Y. Hwang).

0167-9236/$ - see front matter © 2002 Elsevier B.V. All rights reserved.
doi:10.1016/S0167-9236(02)00194-X

acter and the associated way of handling them can be completely decided at build-time. Other exceptions are *unexpected* since they result from unpredictable changes in the environment, being unable to decide how to handle the exception, or from some other factor that simply cannot be predicted at design-time. It has been observed that exceptions occur rather frequently in real working environments [11,27]. This highlights the importance of exception handling in the context of workflow management.

1.1. Related work

The need for handling workflow exceptions has been identified by several researchers and research projects in recent years (e.g. EXOTICA [1], METEOR [27], ADOME [7,8], ADEPT [23,24], WAMO [11], and WIDE [6]). Most research has focused on the handling of expected exceptions whose character can be anticipated at build-time. To keep the logic of the normal process clean, exceptions and their handling approaches are usually not incorporated into standard workflow types. Instead, another more flexible mechanism is adopted to support explicit modelling of these exceptions. Two approaches that are typically used to implement this mechanism are the *extended transaction model* and *event–condition–action* (*ECA*) *rules*. An extended transaction model requires the workflow designer to specify some properties of the constituent activities and sub-workflows, such as compensatible, retriable, and alternating activities. When an exception occurs, the workflow system handles it according to the given attribute values of the involved activities or sub-workflows. A typical situation would be to roll-back activities to a particular point by executing the corresponding compensating activities in reverse order. An alternative path would then be taken when continuing the execution [1,12,21]. Another approach is to use ECA rules, or so-called triggers [6–8]. The ECA paradigm describes an exception type as a particular ECA rule. The event and the condition of an ECA rule describe the situation under which the associated exceptions occur, and the action part defines the operations that would resolve these exceptions. Possible operations include notifying responsible persons, ignoring exceptions, retrying the activity that causes exceptions, partial rollback followed by forward execution, adding some extra activities, deleting some planned activities, or any change to the part of the workflow definition that is not yet executed. Recently, Hagen and Alonso [14] proposed a framework that integrates rules[1] and an extended transaction model for handling workflow exceptions. Participants in the ADEPT project also proposed an approach which involved evaluating the correctness of changes to workflow schema [23,24]. While previous work has focused on providing flexible mechanisms for defining exceptions at build-time, there is a need to handle exceptions that are not defined at build-time. These exceptions are by no means uncommon and, in some cases, could be substantial.

Cases not specified by the defined workflow types require special treatment. Even though a WFMS may be capable of executing any exception resolution plan that is specified in a particular format, deriving an appropriate solution for handling a given exception is currently conducted in a manual, ad hoc manner, which involves numerous meetings and discussions with authorized and knowledgeable persons. We proposed [29] providing a query interface to enable users to browse the information related to the workflow instance to which an exception occurs. In the proposal, a query is specified in terms of attributes of constituent activities which include, for example, input and output values, the date and time, and details of the performers. By examining the attribute values returned by a number of queries, the user makes an appropriate decision on how to handle the current exception. Chiu et al. [8] proposed the *Human Interface Manager* for handling unexpected exceptions. This device handles exceptions by listing common approaches that serve as suggested resolutions, and allows users to visualize all the recent methods that have been used to resolve exceptions. However, exactly how to provide a list of suitable resolutions for a given exception in a changing environment was not discussed in detail.

[1] In their work, the exception model in C++ or Java, rather than ECA rules, is used to combine with an extended transaction model. However, we consider this exception model to be the same as ECA rules in spirit.

1.2. Contributions

To further facilitate the decision-making process, in this paper we propose an architecture model that deals with both expected and unexpected exceptions with emphasis on the handling of unexpected exceptions. Specifically, we propose an intelligent system that resolves an unexpected exception in the following way. It first searches on a set of prestored exception cases for the similar ones. Then by referencing these similar cases and how they were resolved previously, it determines the approach that is most appropriate for the current one. In this paper, we will concentrate on the first job, which is challenging in the workflow context. Specifically, we will describe the kind of information required about exceptions, the similarity metrics tailored to workflow exceptions, and the algorithms using these metrics for similarity searching. We also present results from applying the algorithms to synthetic data sets, which demonstrate the relative performance of the different algorithms.

The remainder of the paper is structured as follows. Section 2 illustrates the need for handling unexpected exceptions by showing a real-world example. Section 3 is devoted to the description of the architecture model adopted in this paper for handling both expected and unexpected exceptions. Section 4 describes the types of information about previous unexpected exceptions and the types of queries used to search these exceptions. In Section 5, we formally define a similarity measure between exceptions and discuss how to compute the similarity between two exceptions under various conditions. In Section 6, we develop algorithms for searching for those previous exception instances that are similar to a given one. We test and compare these algorithms in Section 7. We conclude the paper in Section 8 by summarizing the main results and identifying issues for further study.

2. A motivating example

The authors have been involved in the investigation of various business processes that are used by the National Health Insurance Bureau (NHIB) of Taiwan. The NHIB is a government-sector business whose primary mission is to execute the National Health Insurance Program in Taiwan, and consequently, the vast majority of clinics and hospitals in Taiwan have contracts with the NHIB. According to a recent survey, Taiwan has the highest average outpatient visits per capita in the world (see http://www.nhi.gov.tw/english/englsih2-7.htm). Thus, it is imperative to closely review the insurance claim process to prevent the misuse of medication. The current insurance claim process includes the following four sequential steps:

1. Receiving monthly insurance claims from a medical institute.
2. Correcting errors in these claims.
3. Evaluating these claims.
4. Filing a payment to the medical institute.

Note that each step is complicated and comprises many activities. Usually a single processing instance takes up to 60 days to complete. To ensure that medical institutes receive their payments as scheduled, the following prepayment policy has been enforced:

(A) If an insurance claim case does not complete within 30 days, the relevant medical institute automatically receives up to 90% of their total claims.
(B) If an insurance claim case does not complete within 60 days, the medical institute receives the remaining payments as they are claimed.

However, if after the completion of a case it is found that the total insurance claims amount to less than the prepayment, another process is invoked to request a refund from the medical institute.

It is clear that the insurance claim process is very important to the NHIB. On one hand, the entire process has to be conducted carefully (and is therefore time-consuming) so as to prevent the misuse of precious medical resources. On the other hand, the turnaround time of this process is expected to be short to meet the financial needs of the medical institutes and prevent invoking the prepayment and refunding processes. To achieve both goals, a promising approach is to employ WFMSs to both speed up and better coordinate the process. To do so, an

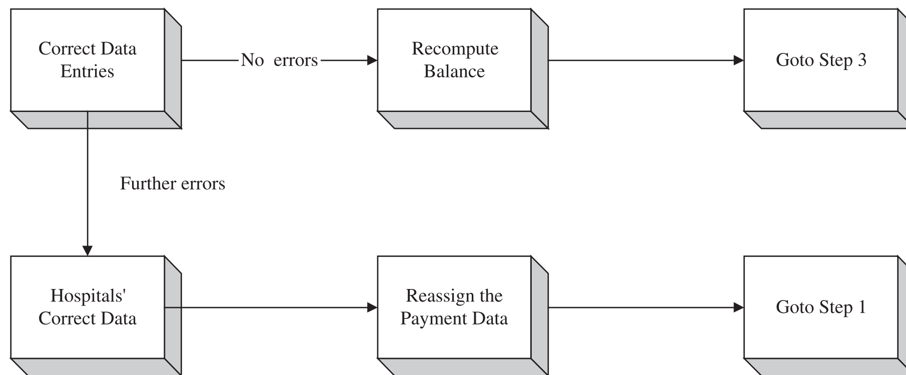

Fig. 1. Subprocess of "correcting errors" in the insurance claim process.

unambiguous specification of the activities as well as their coordination is vital. However, we have found that some activities simply cannot be specified clearly. As an example, consider the second "correcting errors" step of the payment process. This can be further decomposed into substeps shown in Fig. 1.

The staff members at the NHIB first try to correct errors that appear in the submitted claims. Possible errors include missing data entries, contradictory data entries (e.g. by switching the start and ending dates), and incompatible data formats. Errors are corrected, for example, by making telephone calls to the hospital to obtain the missing data, or by searching other sources to allow filling in or correcting of the data provided. However, if errors exist that cannot be resolved easily, all the claims are returned to the claiming hospital. The hospital is then required to correct the data themselves and return the corrected claims within a few days. In this case, the prepayment mentioned above has to be subsequently postponed. In fact, an official policy limits the duration of "Hospitals' Correct Data" to no more than 7 days—if the NHIB does not receive the corrected claims within 7 days, the entire case is closed. However, we noticed that NHIB staff members do not usually follow this rule strictly. By considering the location of the medical institute and the amount of errors incurred, they tend to extend the deadlines to up to 15 days. Furthermore, frequently one or two reminders are sent to the medical institute before the deadline. If the corrected claims are returned quickly enough, the payment date is often not even changed.

It is clear at this point that the ways to handle "Correct Data Entries" and "Hospitals' Correct Data" are flexible and therefore cannot be specified exactly. According to our recent survey at the south office[2] of the NHIB, there were about 1000 cases with erroneous data entries per month. Among them, about 10% cannot be corrected by NHIB staff and must be returned to the medical institutes for correction. We view these cases as exceptions, and because their character and handling approaches cannot be well defined, we further regard them as *unexpected* exceptions. These exceptions constitute a large amount of information if they are accumulated over several years, and this information can provide valuable insight into determining how to best handle future exceptions.

3. The architecture model

Fig. 2 depicts the reference architecture adopted in this paper. Administrators define a workflow type by describing its constituent activities and the (control/data) interdependencies between them via some workflow definition tool. This information is stored in the workflow database (WFDB). Formally, a workflow type is a tuple $\langle V, E, F_1, F_2, \ldots, F_k, P \rangle$, where V is a set of activities; E is a set of transitions, each of which embodies a control

[2] The NHIB has totally six branches in Taiwan. The information provided here concerns only one office.

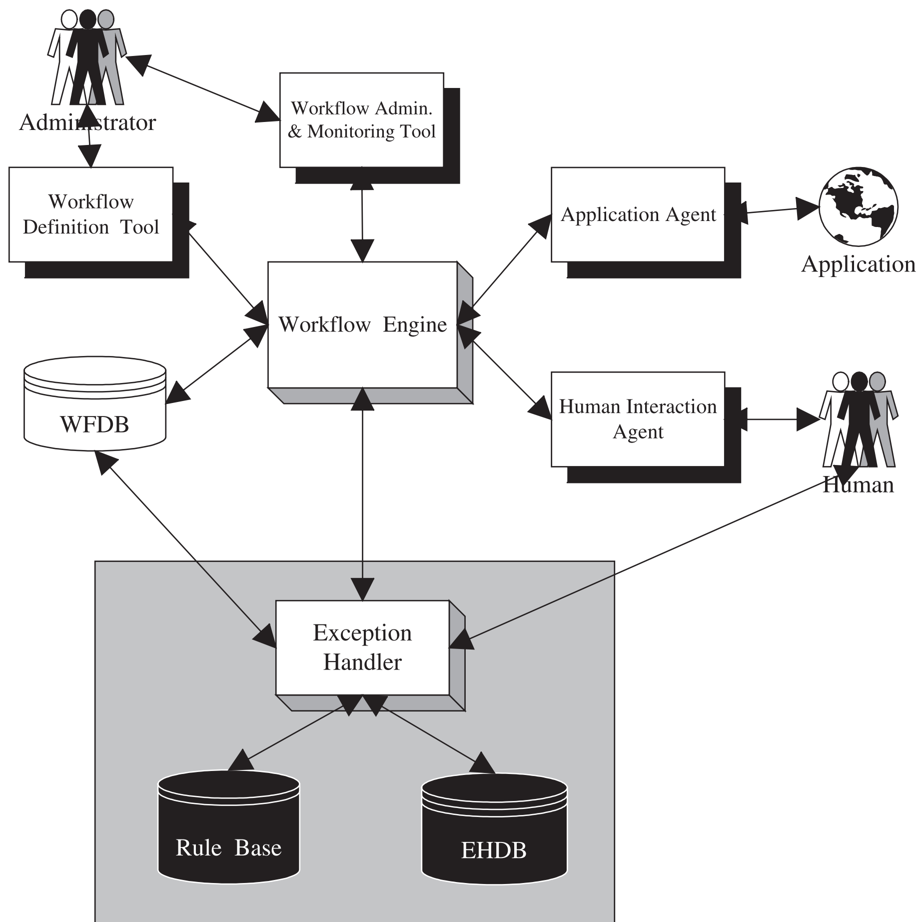

Fig. 2. The reference architecture (WFDB is the workflow database, and EHDB is the exception handler database).

dependency between a pair of activities; F_i, $1 \leq i \leq k$, is a function that maps an activity to its *i*th attribute value; and P is a function that maps a transition to its predicate. Possible attributes of an activity include qualified participants, applications capable of executing this activity, input data, output data, the precondition before the activity can be executed, and the postcondition that must be satisfied after the activity completes. Note that an activity can be defined as another workflow type, thereby forming a nested process definition. Another function F_j can be used to specify such a subworkflow definition. The predicate of a transition defines the condition under which the transition is to occur. The workflow engine coordinates the workflow execution in such a way that all constraints—as specified in F_i and P—are enforced. Note that an activity can be conducted by either a human operator or a computer program. The interaction between the performer of an activity and the workflow engine is enabled by either a human interaction agent (in the case of a human activity) or an application agent (in the case of a computerized activity). An agent monitors the execution of its activity and reports its status to the workflow engine when necessary. In addition to handling human activities, human interaction agents also help manage process steps, such as starting a new process instance, or suspending or terminating an existing process. When a user detects that relevant events have occurred in the outside world, it interacts with the workflow engine through a human

interaction agent to ensure the proper handling of the effects of these events.

Under normal circumstances, agents successfully execute their associated activities and report the execution results to the workflow engine. The workflow engine will store these results in the output data, evaluate the predicates associated with the transitions that follow, and invoke the activities that are next executed. However, when an agent detects the occurrence of an exception to the activity it executes or an event in the outside world that affects the ongoing process, it *throws* an exception [19]. To properly handle the exception, the workflow engine then uses the shaded component in Fig. 2, the *Exception Handler*, to determine a handling approach.

To determine a suitable approach for handling an exception, the exception handler performs a two-step procedure. First, it consults the rule base that contains a set of rules for handling exceptions. Each rule describes how to handle a specific exception type. More precisely, a rule is represented as a four-tuple ⟨*scope, event, condition, action*⟩. The *scope* specifies an activity (which could be an atomic activity or a subprocess) to which this rule is applied. The *event* denotes the type of exceptions that this rule is designed to handle. The *condition* describes the predicate which, when satisfied, triggers the process defined in the *action*. The process defined in the action part of a triggered rule is a workflow type that is designed to replace the activity specified in the *scope*. When the execution of an activity throws an exception, the workflow engine first searches for the rules with a *scope* equal to this activity and an event equal to the type of this exception. If no such rule exists or none of the rules satisfy their respective conditions, the workflow engine then searches another set of rules with a scope that is the immediate ancestor of this activity in the process definition. The same procedure continues until either a rule is triggered or no ancestor is found. After identifying a triggered rule, the exception handler sends a ⟨*scope, action*⟩ pair to the workflow engine, which defines an exception handling approach. The workflow type defined in the *action* serves as a *fixing process* that is used to fix the activity specified in the *scope*. Therefore, the workflow engine will first execute the workflow defined in the *action* and resume execution of the original process by assuming that the replaced activity specified in the *scope* has been successfully executed. To execute the fixing process specified in the *action*, some workflow data values changed during the failed activity specified in the *scope* may have to be undone to eliminate its effect. This would require a recovery procedure that traverses the workflow log in a manner similar to that used to handle transaction failure. Interested readers are referred to our previous work [29] for details about such a recovery. If, however, the execution of the fixing process throws a new exception, the same rule-searching procedure is conducted following the rules pertinent to this process. To avoid repeated spawning of exceptions (so that acceptable workflow completion time will not be jeopardized), one may choose to associate only those rules that handle general exceptions, such as machine failures, to a fixing process. In case any other exception occurs to the fixing process, the system resorts to humans for proper provision to ensure its semantics.

On the other hand, if none of the rules are triggered, the second step is taken. Assume that an exception handler database (EHDB) contains the feature and the handling approach for each unexpected exception that occurred previously. The exception handler will search the EHDB for previous exception cases that are similar to the current one in their features. The approaches used to resolve those exceptions will be presented to the user as alternatives for handling the current exception. Once an alternative is chosen, possibly with some modifications, it will be given to the workflow engine for implementation. The feature of this exception and its handling approach will be inserted into the EHDB as a new record. Formally, we define an exception record as a triplet ⟨*feature, action, scope*⟩, where the *feature* describes the characteristics of the exception, such as the type of the exception, the activity to which the exception occurs, and the time and place that this exception occurred, and the *action* specifies a workflow type that is used to fix the activity specified in the *scope*. By examining the ⟨action, scope⟩ pairs of the exceptions with similar features, the person responsible then determines a specific ⟨action, scope⟩ pair that is appropriate for handling the current exception. Finally, this ⟨action, scope⟩ pair is sent back to

the workflow engine for execution. The complete algorithm that the exception handler uses to handle a given exception is shown below.

```
function Search-RuleBase(t: Activity, e: Event): Rule
begin
    R := {r: r is a rule in Rule Base and r.scope = t and r.event=e};
    for each r in R do
    begin
        if r.condition is evaluated to be TRUE
        then return r
    end
    /* no rule in R is triggered */
    if t.parent = NULL /* if t is the top-level process */
    then return NULL;
    else return Search-RuleBase(t.parent, e);
end

Handling-Exception(x: Exception)
begin
    r =  Search-RuleBase(x.activity, x.event);
    if r ≠ NULL then send <r.action, r.scope> to Workflow Engine
   else /* no rule in rule base is triggered */
   begin
        E:={e: e is an exception in EHDB that is similar to x};
        Responsible person decides an appropriate <action, scope>
        by consulting exceptions in E;
        Send this <action, scope> pair to Workflow Engine;
   end
end
```

Referring to the above algorithm, to handle a given exception *x*, the Handling-Exception(*x*) procedure is invoked. We assume an exception is a structure that records information about the exception, including the activity occurring (*x.activity*) and the event type of exception (*x.event*). Handling-Exception(*x*) first searches the rule base by initiating the procedure Search-RuleBase(*x.activity, x.event*). Procedure Search-RuleBase(*t, e*) first searches for the rules with a scope and event equal to *t* and *e*, respectively. The conditions of these rules are then evaluated to determine whether any rule can be triggered. Note that we do not consider the possibility that more than one rule is triggered. When this does happen, we follow the *static priority* scheme, which is the most popular scheme among expert system implementations, in choosing a rule to fire [22]. The static priority scheme lets the designer determine an absolute value for each rule as its priority and fires the fireable rule with the highest priority. We feel that the static priority scheme is suitable in the context of workflow exception handling because of its deterministic behavior and simplicity. On the other hand, if no rule is triggered, a recursive call Search-RuleBase(*t.parent, e*) is invoked to search for the rules with a scope equal to the immediate ancestor of *t* in the workflow type. This procedure continues until either a rule is triggered or the top-level process is reached. If, however, the current exception cannot be handled by the rule base, this exception is designated an *unexpected exception*. In this case, we resort to using human experts for determining an appropriate ⟨scope, action⟩ pair, with support from the system that provides previous resolution approaches (*E*) in handling similar exception cases.

4. Characterizing unexpected exceptions and their queries

To locate similar previous exception cases, we have to elaborate the attributes associated with features in an exception record. An exception case can be described by many attributes, including:

1. *Status*: This records the status of a workflow instance and its constituent activities. The possible statuses of a workflow instance include initiated, running, suspended, terminated, active, and complete, while an activity may be in one of the following statuses: inactive, active, suspended, complete [31].
2. *Activity*: This attribute records the activity that causes the exception.
3. *Exception type*: This is the semantic information that describes the type of the exception, which could be described by a set of user-defined keywords.
4. *Time*: This denotes the time when the exception occurs.
5. *Participant*: This attribute indicates the performer of the activity that causes the exception.
6. *Characteristic attributes*: These include data values specific to a workflow type. For example, in the NHIB's claim handling process, there are several specific attributes, including *hospital*, *amount claimed*, and *number of records claimed*.

We call the first five attributes *generic* because they are applicable to each workflow type. The characteristic attributes, on the contrary, are specific to a workflow type. While the generic attribute values of exceptions are always recorded in NHDB, the characteristic attributes need not to be stored entirely due to performance consideration. When a workflow type is just deployed, only generic attributes about exceptions are recorded in NHDB. As time goes by and it is found that users often query some characteristic attributes values when determining the way to handle an unexpected exception, these attributes are added to the NHDB for subsequent exceptions. In other words, the set of characteristic attributes recorded in EHDB evolves.

Finding previous exception records that are similar to the current one could be a subjective task, and different agents may want to do different kinds of similarity matching. For example, when a junior level staff detects an exception, she may create an exception record with feature being ⟨(...),..., (level of staff: junior),...⟩, intending to see how her peers handled similar cases. But she may also be interested in finding previous records of the form ⟨(...),..., (level of staff: senior),...⟩ to see how senior staff handled such cases.

There are two approaches to conduct similarity matching. The first approach is that the agent itself specifies a query record that she would like to find a match for, and the exception handler just returns all the records that exactly match this query record. However, to make a proper decision, the agent probably has to create and submit many records. To further assist the decision making, we adopt a second approach that allows the agent to specify a criterion and let the exception handler to decide which exception records are considered similar based on this criterion. The issue here is how to specify a criterion.

To precisely define the concept of similarity, we incorporate *concept hierarchies* into our approach [15]. A concept hierarchy is a partial general-to-specific ordering of concepts. Several of the attributes mentioned above can have their own concept hierarchies. For example, the nested definition of workflow types provides a natural definition of the concept hierarchy for the *Activity*. As the semantics are often organized into a taxonomy of concepts that are partially ordered, this taxonomy could serve as the concept hierarchy for the *Exception type*. The organization hierarchy is an ideal candidate for the concept hierarchy of *Participant*. Finally, *Time* values can be further decomposed into year, month, day, and time. In the case where there is no obvious concept hierarchy associated with a quantitative attribute, a concept hierarchy can be computed using an approach that combines consecutive intervals in a recursive manner based on some measurement [16,26].

To specify a criterion for conducting similarity matching, a set of attributes that are considered relevant in making decisions is first selected. We call these attributes *candidate attributes*. The importance of each candidate attribute is then specified. The set of candidate attributes and their associated importance enable the exception handler to calculate the distance

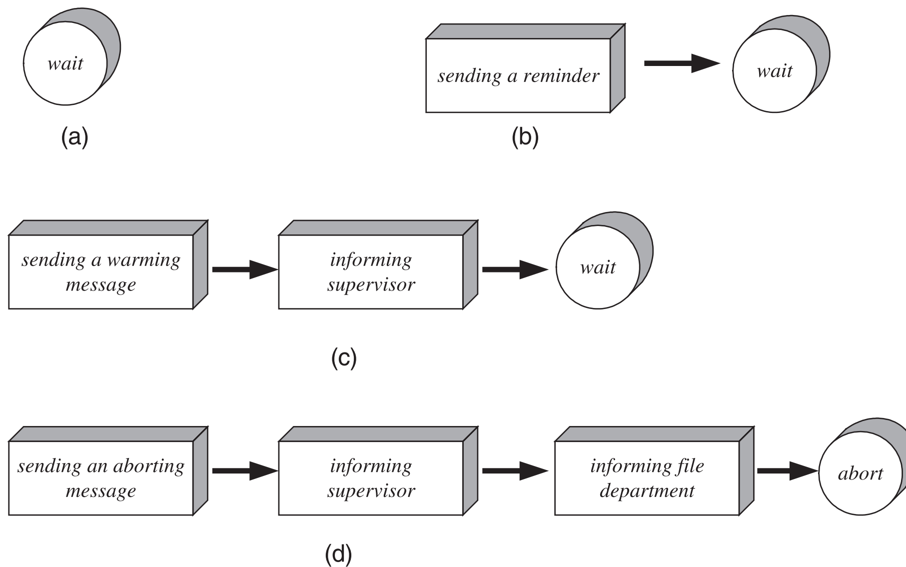

Fig. 3. Various approaches for handling "Claim returns expired" exceptions.

between two exception records. It can then rank the exception records based on their distances to the current exception record. Based on different criteria, two kinds of similarity matching will be used.

4.1. Closeness matching

A closeness matching searches for a predefined number of closest records to the current exception record. The issue here is how to define the distance between two records in the context of workflow exception handling, and how to do the search efficiently. We will discuss this issue in detail in Section 5.

4.2. Targeted-closeness matching (TC)

This kind of matching first specifies a subset of feature attributes, termed *target attributes*. For each target attribute, a set of values is specified. Only the records with the values in the specified set for the target attributes will be retrieved. A closeness matching is then followed only on the retrieved records. More precisely, a TC matching is a tuple $\langle A, c, T, P\rangle$ where A is the set of candidate attributes, c is the current exception record, $T=\{a_1,\ldots, a_v\}$ is the set of target attributes, and $P=\{p_1,\ldots, p_v\}$, where for all i, $1 \leq i \leq v$, p_i is a predicate defined on the domain of a_i. A record r is a target of a TC matching if for all i, $1 \leq i \leq v$, $p_i(r.a_i) = \text{true}$[3]. A target matching is useful when an agent wants to reference some records with specific values for certain attributes.

4.3. Example

Consider the exception handling at "Hospitals' Correcting Data," a step for correcting errors in NHIB's medical insurance claim handling process. To focus, we discuss only a particular type of exception, in which a hospital does not return their corrected claims within a specified period. This type of exception is termed "Claim returns expired," which is thrown periodically if the NHIB does not receive the corrected claims as scheduled. Since the official policy states that a hospital should correct errors and return the corrected claims within 7 days, we can design a rule that throws the "Claim returns expired" exception every two days after 5 days of sending claims back to a hospital.

[3] We assume that $\forall i\ 1 \leq i \leq v$, $\exists\ d \in \text{domain}(a_i)$, $[p_i(d) = \text{true}]$, since otherwise it is pointless for the violating a_i to be designated as a target attribute.

Table 1
A sample EHDB

Feature						Action	Scope
Exception	Time	Participant	Activity	Hospital	Days delayed		
...	...	...	...	...	...	...	...
Claim returns expired	2001-5-12	Anna	Hospitals' correcting data	HDH	13	Fig. 3c	Hospitals' correcting data
Claim returns expired	2001-9-3	Rita	Hospitals' correcting data	VMH	17	Fig. 3d	Claim handling
Claim returns expired	2001-11-3	Rita	Hospitals' correcting data	CYC	9	Fig. 3d	Claim handling
Claim returns expired	2001-11-4	John	Hospitals' correcting data	CSH	7	Fig. 3b	Hospitals' correcting data
Claim returns expired	2001-12-14	Ann	Hospitals' correcting data	VMH	5	Fig. 3a	Hospitals' correcting data
Claim returns expired	2001-12-20	Beth	Hospitals' correcting data	DLH	7	Fig. 3c	Hospitals' correcting data
Claim returns expired	2002-1-4	John	Hospitals' correcting data	HDH	7	Fig. 3b	Hospitals' correcting data
Claim returns expired	2002-1-13	Rita	Hospitals' correcting data	CSH	11	Fig. 3c	Hospitals' correcting data
Claim returns expired	2002-2-20	Mary	Hospitals' correcting data	HDH	11	Fig. 3d	Claim handling
...	...	...	...	...	...	...	...

There are four possible approaches to handle this exception:

1. Simply ignoring it and continuing waiting for the arrival of the corrected claims.
2. Sending a gentle reminder to the hospital and then continuing waiting for the arrival of the corrected claims.
3. Sending a warning message to the hospital, informing the supervisor, and then continuing waiting for the arrival of the corrected claims.
4. Closing the case of processing these claims. The hospital has to reinitiate the insurance claim handling process in the future.

These four approaches are depicted graphically in Fig. 3. However, exactly which options should be chosen under various conditions are not officially recognized, and therefore these handling approaches will not be summarized in the rule base.

Assume that previous experiences in handling exceptions have been stored in the EHDB, which is partly shown in Table 1.

The EHDB could store thousands of records about the features and handling approaches for *Claim returns expired*, and the agent is interested in only those records that have similar feature values.

Suppose a clerk Liza at NHIB is assigned to handle an emerging *claim returns expired* exception with

Table 2
Current exception and the relevant exception records

Feature						Action	Scope
Exception	Time	Participant	Activity	Hospital	Days delayed		
(a) The current exception							
Claim returns expired	2002-2-10	Liza	Hospitals' correcting data	CSH	7		
(b) Exception records listed in ascending order of their importances							
Claim returns expired	2001-11-4 (3)	John (3)	Hospitals' correcting data	CSH (0)	7 (0)	Fig. 3b	Hospitals' correcting data
*Claim returns expired	2002-1-4 (2)	John (3)	Hospitals' correcting data	HDH (1)	7 (0)	Fig. 3b	Hospitals' correcting data
Claim returns expired	2001-12-20 (3)	Beth (1)	Hospitals' correcting data	DLH (2)	7 (0)	Fig. 3c	Hospitals' correcting data
Claim returns expired	2001-12-14 (3)	Ann (1)	Hospitals' correcting data	VMH (2)	5 (1)	Fig. 3a	Hospitals' correcting data
*Claim returns expired	2002-1-13 (1)	Rita (3)	Hospitals' correcting data	CSH (0)	11 (3)	Fig. 3c	Hospitals' correcting data
*Claim returns expired	2002-2-20 (2)	Mary (1)	Hospitals' correcting data	HDH (1)	11 (3)	Fig. 3d	Claim handling
Claim returns expired	2001-5-12 (3)	Anna (3)	Hospitals' correcting data	HDH (1)	13 (3)	Fig. 3c	Hospitals' correcting data
Claim returns expired	2001-9-3 (3)	Rita (3)	Hospitals' correcting data	VMH (2)	17 (3)	Fig. 3d	Claim handling
Claim returns expired	2001-11-3 (3)	Rita (3)	Hospitals' correcting data	CYC (2)	9 (3)	Fig. 3d	Claim handling

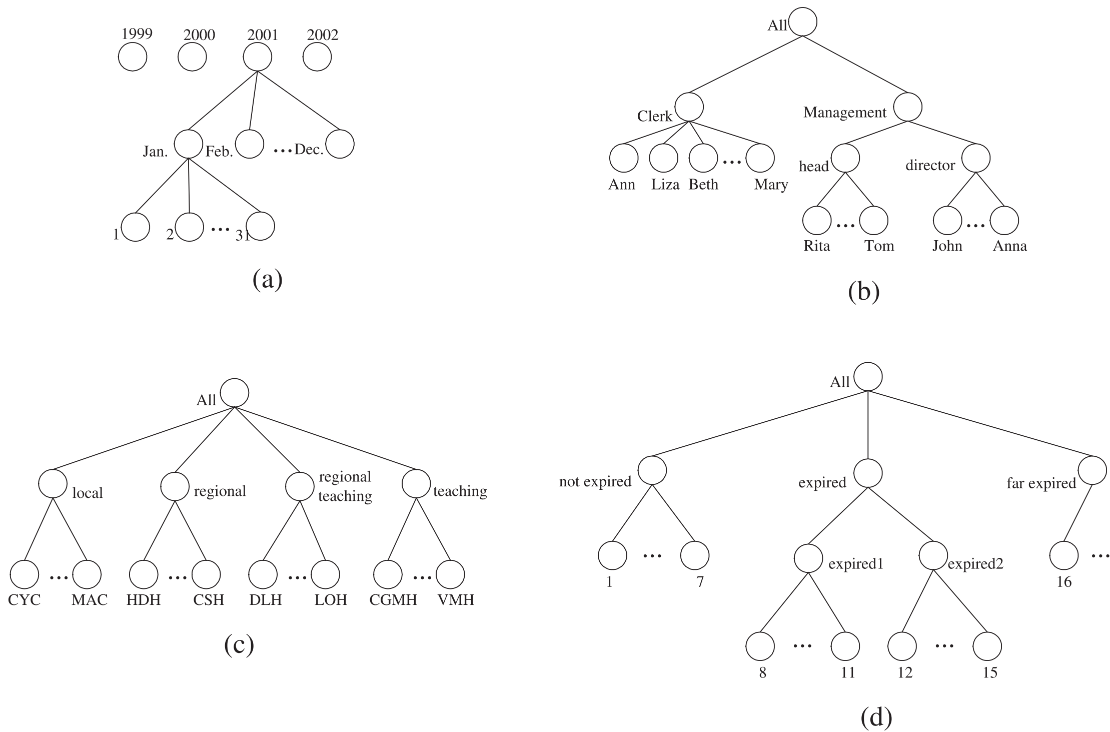

Fig. 4. Concept hierarchies of (a) Time, (b) Participant, (c) Hospital, and (d) Days delayed.

attributes shown in Table 2(a). To make a proper decision, she is interested in knowing how previous cases were handled. For this purpose, she has determined that two generic attributes: *Time* and *Participant*, and two characteristic attributes: *Hospital* and *Days delayed* are candidate attributes and that their importance are prioritized by the following order: *Days delayed, Hospital, Participant*, and *Time*. Fig. 4 depicts the example concept hierarchies of these four candidate attributes. The exception records are listed in ascending order of their importance (smaller values indicate higher importance), as shown in Table 2(b). Note that the number in the parenthesis next to an attribute value is the level number[4] of the least common ancestor of this value and the corresponding attribute value of the current exception. From the list displayed in Table 2(b), Liza decides to adopt the action by John, as the corresponding records are the closest to the current record. Thus, she handles this case by sending a reminder message to CSH (the approach depicted in Fig. 3b). This is an example of closeness matching.

Now suppose Liza instead wants to find out how her colleagues in other regional hospitals handle this kind of exceptions in the year of 2002. She therefore identifies *Hospital* and *Time* as the target attributes, and define the predicates on *Hospital* and *Time* as $p(h)$ = 'h is a regional hospital' and $p(t)$ = 'year(t) = 2002', respectively. She then prioritizes the three candidate attributes in the order: *Participant, Days delayed,* and *Time*. As a result, only the three exception records marked by asterisks in Table 2(b) will be retrieved. This is an example of targeted-closeness matching.

Note that the concept hierarchy pertaining to each attribute may vary over time. For example, if some day the official period for hospitals' correcting data is extended to 10 days, the concept hierarchy for *Days delayed* has to be changed accordingly. In summary, to enable an effective similarity matching, the agent

[4] We number levels from leaves upward. That is, the concept at a leaf node is designated as level 0, and that at the parent of a leaf node is assigned level 1, and so on.

has to specify a matching criterion as the input of the exception handler that includes:

1. the feature of the current exception,
2. (optional) a set of target attributes, each of which is associated with a predicate,
3. a set of candidate feature attributes,
4. a set of concept hierarchies, each associated to a candidate attribute, and
5. a set of importance values, each pertaining to a candidate attribute.

5. Retrieving similar exceptions

This section discusses how to quantify the similarity between exceptions. For the closeness matching, we select the k exceptions that are the most similar, where k is a constant specified by the user. For the TC matching, for each stored record in EHDB we first evaluate each predicate for the corresponding target attribute value. If all the predicates evaluate to true, we calculate the distance between that record and the current exception record based on the candidate attributes. These records are ranked in ascending order of their distances and the first k records are selected. Thus, for both kinds of matching, the main challenge lies in the calculation of the distance between records for a given set of (candidate) attributes.

The basic idea is to first compute the similarity of each selected attribute of the exception feature by considering the associated concept hierarchy. Each attribute similarity is then weighted by its importance when computing the entire exception similarity. Let m be the number of candidate attributes provided by the agent. Each exception record can be thought as a vector in m-dimensional space. Let the feature of current exception be $c=(c_1, c_2, \ldots, c_m)$, and that of the ith exception e_i in the EHDB be $e_i=(e_{i,1}, e_{i,2}, \ldots, e_{i,m})$.

5.1. Distance between two attribute values

While the distance between numeric attribute values can be easily determined by using a simple function such as subtraction, that between categorical attribute values is less obvious to specify. A common approach for computing distance between values of a categorical attribute is to construct a semantic ontology pertaining to the attribute, and the distance between two concepts is measured as the number of edges on the shortest path that connects one concept to the other in the ontology [5]. In our model, a concept hierarchy, which can be considered as a kind of ontology, is associated with each candidate attribute, and the attribute values of exception records locate at the lowest level of the hierarchy. Equivalently, we define the distance between the jth attribute c_j of the current exception and that of the ith exception $e_{i,j}$, denoted att_dist(c_j, $e_{i,j}$), as the level number of the least common ancestor of c_j and $e_{i,j}$ in the concept hierarchy of the ith attribute. Consider the two attributes *Days delayed* and *Hospital* in Example 1. The current exception is characterized as ⟨7, CSH⟩. The distances between attributes of the current exception and those in Table 1 are partially listed as follows:

$\text{att_dist}_1(7, 7) = 0$
$\text{att_dist}_1(7, 5) = 1$
$\text{att_dist}_1(7, 11)$[5] $= 3$
$\text{att_dist}_2(\text{CSH}, \text{CSH}) = 0$
$\text{att_dist}_2(\text{CSH}, \text{HDH}) = 1$
$\text{att_dist}_2(\text{CSH}, \text{VMH}) = 2$

5.2. Distance between two exceptions

There are a number of ways to compute the distance between two vectors. Here we use the Manhattan distance because its computation involves simply the summation of the distances of individual element pairs that can be computed by using att_dist( ) defined above [4]. Furthermore, the distance between two attribute values is weighted by the importance parameter provided by the agent. Specifically, the distance between two exceptions c and e_i, denoted dist(c, e_i), is given by

$$\text{dist}(c, e_i) = \sum_{j=1}^{m} w_j \text{att_dist}(c_j, e_{i,j}), \tag{1}$$

where w_i is the importance weight assigned to the ith attribute.

[5] Here the common ancestor is All, whose level is defined as 3 because it takes three edges to reach 11 from All.

5.3. Heterogeneous records

In the above discussion, we implicitly assume that for each record stored in the EHDB, all the attributes are present that are necessary to perform the matching with the current exception record. In reality, however, this is not always the case. Recall that an exception is associated with two kinds of attributes, generic and characteristic. While the set of generic attributes are stable, the set of characteristic attributes may vary. Exactly which characteristic attributes should be attached to the current exception is at the discretion of the agent who is responsible for handling that exception. Thus, it is possible that some stored records do not contain all the characteristic attributes selected for the current exception record, or they may contain more attributes than necessary. We call these records *heterogeneous records*. The existence of heterogeneous records has two implications. For closeness matching, a missing attribute renders it being impossible to calculate the distance between the heterogeneous record and the current record. For TC matching, if a target attribute is missing, then there is no way to evaluate the predicate on that attribute. In the following, we propose several approaches for handling heterogeneous records. They differ mainly in ways of filling out the missing attributes. Let r be a record stored in the EHDB, and c be the current exception record. Let a be a candidate or target attribute for c, but is missing from r.

5.4. Optimistic method

If a is a target attribute, a value v for a is inserted into r such that $p_a(v) = \text{true}$[6]. If a is a candidate attribute, we insert ca. The philosophy here is that the missing value is assumed to contribute to the likelihood that the heterogeneous record will be selected. Thus, the agent should use this method if the missing attribute is not important. This is desirable since the records that should have been retrieved had they contained the missing values would still be returned. Another situation where the optimistic method can be used is when the agent would like to examine many records to assist him/her in making a decision, or if the number of non-heterogeneous records expected to be returned does not provide a satisfactory reference.

[6] Recall that we assume the predicate is true for at least one value in the domain of the attribute on which it is defined.

5.5. Pessimistic method

This is just the opposite of the above method. If a is a target attribute, and there exists a value v such that $p_a(v) = \text{false}$, then insert v as the value for a into r, otherwise, insert any value in the domain of a into r. If a is a candidate attribute, insert value v such that the youngest common ancestor of v and $c.a$ is at level L, where L is the number of levels in the concept hierarchy for attribute a. We can see that filling out a value in this way contributes to the likelihood that the heterogeneous record will not be selected. If the missing attribute is important, then the heterogeneous records that should not be retrieved had they contained the missing value would indeed not be retrieved. This is desirable since the agent cannot possibly make a wrong reference. However, those records that should be retrieved may also get lost. The agent must decide which aspect is more important. For example, if the agent does not have an idea of how to handle the current exception (i.e. correct reference is important), he/she should use a pessimistic method, otherwise this may not be a good choice.

5.6. Statistical method

This method makes an educated guess of what the missing value would be. A simple way of doing this is to use a random number generator, R_a with a preselected distribution. When it is called, it returns a value from the domain of attribute a. The agent can determine the desired distribution. For example, we may simply use a uniform distribution, which returns any value in the domain of a equally likely. A more sophisticated method would be an algorithm that performs a supervised learning based on the values of some attributes in the stored records in EHDB. (There is a large number of works on supervised learning in the current literature. They can be selectively employed for our purpose.) A statistical method is less aggressive than an optimistic method, and therefore can retrieve less records with low reference values. It is also not as conservative as a pessimistic

method, and hence may obtain more records for references. It is appropriate for an agent who is able to handle the exception at hand with a certain degree of confidence.

6. Searching for similar exceptions

Similarity matching has been studied extensively in the past, and many methods have been proposed. Conceptually, however, they all employ similar strategies, i.e. evaluating the distance between the objects, and find the closest objects. The main differences lie in the metrics used to define the distances, and the data structures used to implement the algorithms. The problem in our context is similar to a class of similarity matching problems, termed *k-nearest neighbor query* (*k*-NN). In the current literature, the best-known data structure used for *k*-NN is R-tree [25]. An R-tree in some sense is similar to a B-tree, but the 'indexing' scheme is very different. The basic idea is that all the points are organized into groups, and each group is bounded by a geometric object (such as a rectangle). At each node, the distances between the query point and the bounding objects that belong to the node are calculated efficiently. These distances can then be used to guide the search, based on the knowledge that all the bounded points have a larger distance than the bounding object from the query point. Recently, some variation of R-tree, such as grid method [18] and SR-tree [17], are suggested for applications with different parameters. Using these data structures, the complexity of *k*-nearest neighbor query can be either logarithmic to the total number of objects stored or linear to the number of objects retrieved.

The techniques used in the R-tree (or its variations) are not applicable in our context, however. This is because, from the way we define the distance, the exception records do not preserve the basic Euclidian properties such as the sum of two sides is larger than the third side in any triangle. (The Euclidian properties are implicitly assumed in R-tree.) In this section, we present three methods, in the order of increasing sophistication, that are suited to our context. Since handling heterogeneous records can be viewed as orthogonal to *k*-NN, in the following we will assume that all the exception records have the same set of attributes.

6.1. SEQ-E

The most straightforward approach for identifying the k exception records that are closest to the current exception record is to sequentially traverse all of the exception records in the EHDB and compute their distances to the current exception according to Eq. (1). The k most-similar exceptions can then be identified at the end of the traversal. This approach is abbreviated as SEQ-E, standing for SEQuential matching algorithm on Exception records. Let N be the number of exception records in the EHDB and B the number of exception records contained in a disk block. The number of blocks accessed is $O(N/B)$, assuming that k is a constant.

6.2. SEQ-C

Rather than scanning every exception record in the EHDB, another approach is to enumerate all possible concept-level permutations in ascending order of their weights. A concept-level permutation is a m-tuple $\langle l_1, l_2,\ldots, l_m\rangle$, where $1 \le l_i \le L_i$ and L_i is the number of levels in the concept hierarchy of attribute i. The weight of $\langle l_1, l_2,\ldots, l_m\rangle$ is $\sum_{i=1}^{m} w_i l_i$. We say that $\langle l_1, l_2,\ldots, l_m\rangle$ is matchable if there exists an exception record e_i=($e_{i,1}$, $e_{i,2}$,..., $e_{i,m}$) in the EHDB such that $e_{i,j}$ and c_j meet at level l_j of the concept hierarchy of attribute j, $1 \le j \le m$. The traversal starts with the concept-level permutation of the lowest weight. If there are k_1 exception records that are targets with this permutation, where $k_1 < k$, we next inspect the concept-level permutation of the next weight to search for $k - k_1$ most-similar exception records. This procedure continues until the k exception records have all been identified. For example, suppose there are four candidate attributes (i.e. $m = 4$) and that the same weight is assigned to each. The traversal sequence on concept-level permutations is $\langle 0, 0, 0, 0\rangle$, $\langle 0, 0, 0, 1\rangle$, $\langle 0, 0, 1, 0\rangle$, $\langle 0, 1, 0, 0\rangle$, $\langle 1, 0, 0, 0\rangle$, $\langle 0, 0, 0, 2\rangle$,..., $\langle L_1 - 1, L_2 - 1, L_3 - 1, L_4 - 1\rangle$. We call this approach SEQ-C, standing for SEQuential matching algorithm on Concept-level permutations.

Since it is more convenient to store past exception records in a database, it would be nice if a simple SQL statement can be used to determine whether a permutation is matchable for the current exception. To do so, we number categorical attribute values based on their

positions in the corresponding concept hierarchies. Consider, for example, the concept hierarchy in Fig. 5a. We can translate the domain values into the scalar values, as shown in Fig. 5b.

Suppose that there are three candidate attributes (A1, A2, and A3), each of which is associated with a concept hierarchy having the same structure as that shown in Fig. 5a. The attribute values of the current exception are a9, a2, and a4, respectively, and we would like to determine if the permutation $\langle 1, 0, 2\rangle$ is matchable. The following query serves this purpose:

```
SELECT*
FROM EHDB
WHERE A1 BETWEEN 6 AND 9
  AND A2 = 2
  AND A3 BETWEEN 1 AND 9;
```

A non-empty set for the query result indicates that the permutation is matchable.

An index on the encoded attribute values can be created to improve the query processing time. In the above example, the query can be processed more efficiently if an index such as the following is constructed:

```
CREATE INDEX
ON EHDB(A1, A2, A3)
```

However, the popular B+-tree-based index structure that is widely implemented in today's commercial DBMSs may not be efficient when processing multi-attribute range queries [28] with a huge data volume. A number of index structures that are designed to support efficient range-query processing on large amounts of high-dimensional data have been proposed in the literature (e.g. see Refs. [2,3,17,20,30]). Another way of speeding up the processing of multi-attribute range queries is to employ parallelism by declustering the data set according to the attribute values of the queries. The readers are referred to Ref. [9] for a good survey on various data placement schemes. Comparison of the various approaches for efficiently answering range queries is beyond the scope of this paper.

In case of targeted-closeness matching, the set of records that are associated with a matchable permutation has to be further examined to obtain a subset that comprises the target records.

For now, let us assume that some tree-based index structure is employed and x is the number of levels in the index structure. The number of blocks accessed by SEQ-C is $O(\Pi_{i=1\text{ to }k}L_i x)$.

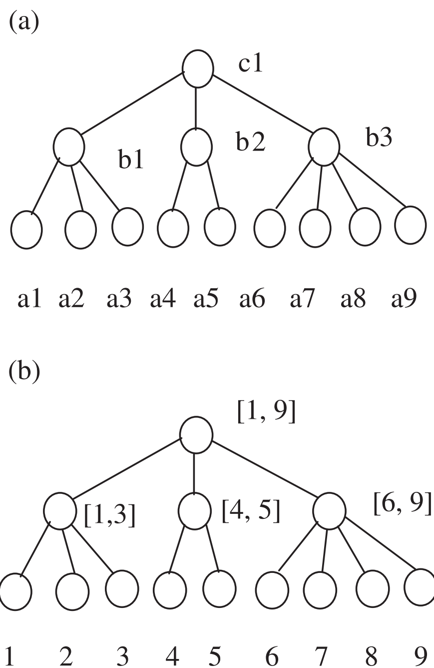

Fig. 5. (a) Original concept hierarchy, (b) Concept hierarchy after encoding.

6.3. BIN-C

To help reduce the search time relative to the above two approaches, we propose another algorithm that performs a binary search on the sequence of concept-level permutations. We call this approach BIN-C, standing for BINary searching algorithm on Concept-level records. Let C be the total number of concept-level permutations, i.e. $C = L_1 \times L_2 \times \ldots \times L_k$, and let $P[i]$ denote the ith permutation in the sequence sorted in ascending order of weights, $1 \le i \le C$. BIN-C first checks if $P[C/2]$ is matchable. If so, the permutations after $P[C/2]$ in the sequence are tentatively ignored, and the next permutation checked is $P[C/4]$; otherwise $P[3C/4]$ is checked. This procedure continues until a matchable permutation $P[j]$ is found such that all the subsequent permutations $P[j/2]$, $P[3j/4]$,. . ., $P[\![j(2^{\log j}$

Table 3
Running times of SEQ-E, SEQ-C, and BIN-C (N is the number of exceptions in the EHDB, B is the number of exception records contained in a disk block, x is the number of index levels)

	SEQ-E	SEQ-C	BIN-C
Best case	$O(N/B)$	$O(x)$	$O(\log(\Pi_{i=1 \text{ to } k} L_i)x)$
Worst case	$O(N/B)$	$O(\Pi_{i=1 \text{ to } k} L_i x)$	$O(\Pi_{i=1 \text{ to } k} L_i x)$
Average case	$O(N/B)$	$O(\Pi_{i=1 \text{ to } k} L_i x)$	not clear

$-1)/2^{\log j}$]] are not matchable. If P[j] is matchable with k_1 (target) records, the same procedure is conducted when searching for the next $k-k_1$ similar records on the permutations between P[$j+1$] and P[n]. In summary, BIN-C first locates a *minimum* matchable permutation and then tries the next matchable permutation, and so on.

Readers may have already noticed that the exception records returned by BIN-C may not be optimal. This is because BIN-C returns a permutation $P[j]$ only if $P[j]$ is matchable and all the following permutations $P[j(2^i-1)/2^i]$, $1 \le i \le \log j$, are not matchable. However, this does not prevent the existence of some matchable permutation $P[i]$, where $i<j$. We have conducted some experiments to compare the quality as well as the efficiency of these algorithms, and report the results in Section 7.

Similar to the case of SEQ-C, we assume that some tree-based index structure is employed and that x is the number of levels in the index structure. In the worst case, when every permutation has to be visited, the number of blocks accessed by BIN-C is also $O(\Pi_{i=1 \text{ to } k} L_i x)$. In the best case, when only a logarithm of the total number of permutations need to be visited, the number of blocks accessed by BIN-C is $O(\log(\Pi_{i=1 \text{ to } k} L_i)x)$. Table 3 summarizes the running times of the three algorithms under various conditions.

7. Performance evaluation

We are still left with the problem of assessing how the three algorithms perform in handling real-world data. Unfortunately, the workflow exception records of the insurance claim process in NHIB that we consulted to were not available to this work due to confidential reasons. Therefore, we generated synthetic data and applied the data to the proposed algorithms. Note that we compare the three algorithms only for closeness matching since they differ mainly in the ways they retrieve the most k similar exception records. These results are intended to reveal the relative performance of the three algorithms under various operation regions.

7.1. Parameter settings

Suppose there are N exception records stored in the EHDB, each of which is described by m attributes. To simplify the performance study, we assume that the concept hierarchy of every attribute is homogeneous in its structure. That is, each concept hierarchy is modelled as having L levels, and the degree of each non-leaf node is D. For the weighting scheme, we have tested several combinations. In this paper we show two representative ones, namely, *equal weighting* and *prioritized weighting*. The equal weighting scheme assigns the same weight to each attribute, and the prioritized weighting scheme prioritizes the attributes into a particular order. In our study, the weight tuple of the equal weighting scheme is $\langle 1,1,\ldots,1\rangle$. Let the priority sequence imposed by the prioritized weighting scheme be $w_1, w_2,\ldots, w_B$, where w_i has higher priority than w_j, $1 \le i<j \le B$. The weight tuple $\langle w_1, w_2,\ldots, w_B\rangle$ of the prioritized weighting scheme is such that $w_B=1$, and $w_i=w_{i+1}L$, $1 \le i<B$. This assignment guarantees that the permutation $\langle 0, 1, 0, 0\rangle$ has larger weight than $\langle 0, 0, L-1, L-1\rangle$. The exception records in the EHDB and in the current exception are randomly generated. These parameters and their values are summarized in Table 4.

In the following, we compare the relative performance of the three algorithms under various parameter settings. These experiments were conducted on a PC

Table 4
Parameter values

Parameter	Description	Value(s)
N	Number of exception records in the EHDB	10,000...1,500,000
m	Number of candidate attributes	4
L	Number of levels in a concept hierarchy	6
D	Degree of each internal node in a concept hierarchy	5
k	Desired number of returned exception records	1, 10, 100
W	Weighting scheme	Equal, Prioritized

comprising a 300-MHz Pentium processor coupled to 256 MB of RAM.

We generated a number of synthetic data sets and stored them in MS SQL Server 6.5. The three algorithms were implemented using Borland's Delphi Pascal. A B+-tree index on the conjunction of all attributes was created to improve the running time of both SEQ-C and BIN-C.

7.2. Impact on efficiency

Fig. 6a and b shows the running times of the three algorithms under the equal and prioritized weighting schemes, respectively. As expected, the running time of SEQ-E is approximately linearly proportional to the number of exception records in the EHDB. To stretch these algorithms to their limit, the number of exception records was set up to 1,500,000. It can be seen that the increase of the running time with the increase on the number of exception records is quite mild initially. However, when the number of exception records reaches more than 1,000,000, the running times of both SEQ-C and BIN-C incur a big jump. After the jump, the increase of running time remains mild again. This is because the height of the index structure (such as the B+-tree) is a step-like function

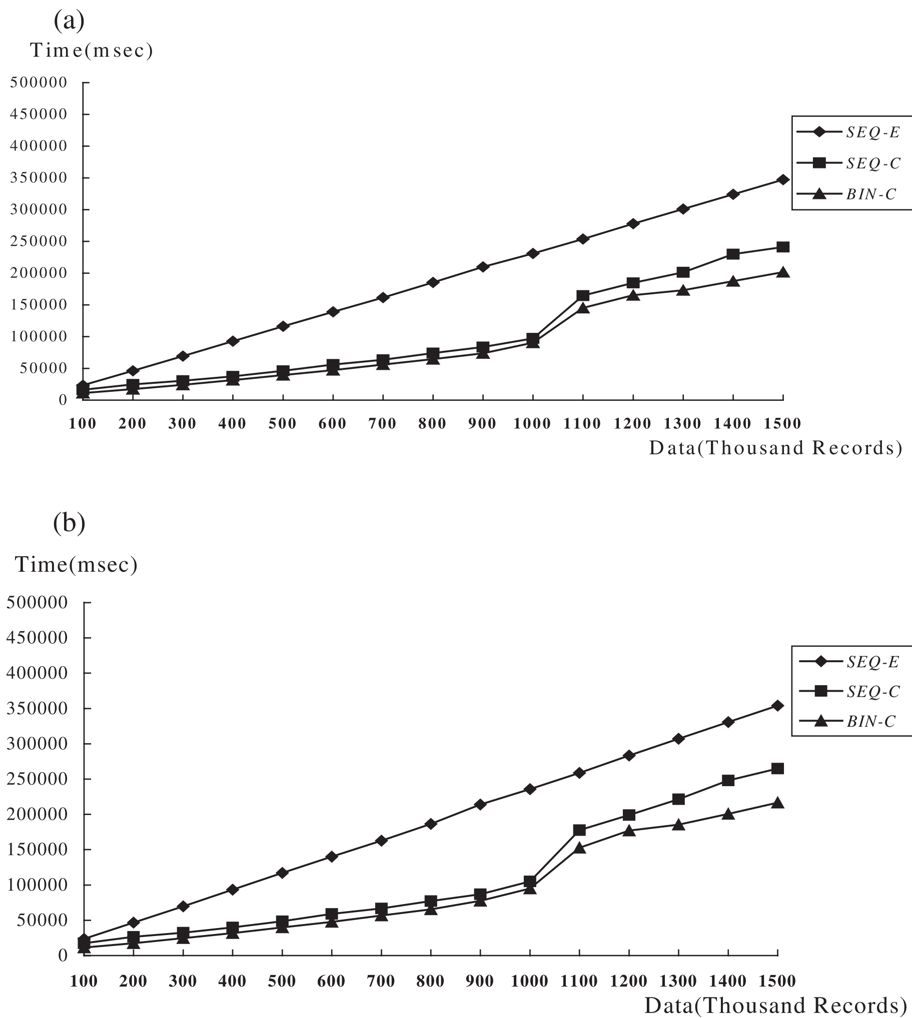

Fig. 6. (a) Running time for the equal weighting scheme, (b) Running time for prioritized weighting scheme.

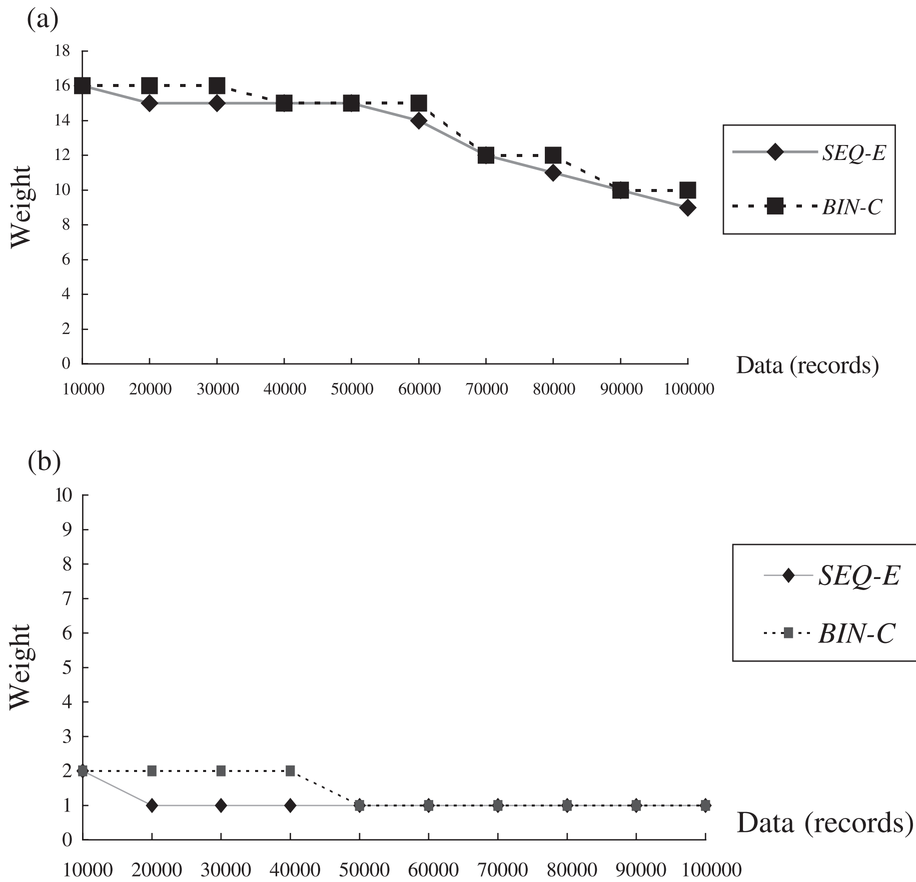

Fig. 7. (a) Weight of the matching exception records for the equal weighting scheme, (b) The first attribute value of the matching exception records for the prioritized weighting scheme.

on the number of records. In our experiment, when the number of exception records is approximately 1,000,000, the number of index levels increases by 1, which dramatically increases the running time. This explanation is consistent with the following index-level information provided by SQL Server 6.5, which shows that when the number of records is between 1,000,000 and 1,100,000, the index level is incremented (in the average case).

B+-tree levels

Data(Thousand Records)	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500
Min.	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Avg.	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3
Max.	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3

Fig. 6 also shows that BIN-C always performs better than SEQ-C. Although the difference may not seem significant in the diagram, a small difference in the diagram may result in a large time difference in the real world due to the longer execution time in the experiments (i.e. minutes). Moreover, SEQ-C performs much better than SEQ-E. We attribute the superior performance to the index structure employed by SEQ-C, which is much more efficient than the sequential scan performed in SEQ-E.

7.3. Impact on quality of the returned records

The exception records returned by SEQ-E or SEQ-C have the minimum weights, while those returned by BIN-C may be less than optimal. To see how the exception records returned by BIN-C differ from those returned by the other two in terms of weight, we conducted an experiment for $k=1$ (Fig. 7a). As expected, SEQ-E (or SEQ-C) always returns the exception record with the smallest weight, whose weight decreases as the number of records in the EHDB increases. This is because we fix the structure of the concept hierarchy, and thus as the number of exceptions records increases, there is a greater chance of finding a matching exception record with a smaller weight. Moreover, we can clearly see in Fig. 7a that the weight difference between the exception records returned by the two algorithms is very small, and in several instances, the results of BIN-C are as good as those of SEQ-E (or SEQ-C).

As the prioritized weighting scheme gives the first attribute the highest value, we conducted another experiment to determine—for each algorithm—the value of the first attribute of the returned exception records. The result is shown in Fig. 7b, which indicates that the performance of BIN-C and SEQ-E (or SEQ-C) is very similar in terms of the quality of the first attribute values under various operating regions.

Overall, we conclude that BIN-C is a promising approach for searching similar exception records. It offers higher efficiency at the cost of a minor decrease in the quality of the result. Since the result returned by the exception handler serves only as supporting information to the agent in deciding the proper approach to handling the current exception, this small deviation from the optimal result should be acceptable in most cases.

8. Conclusions

Handling the unexpected exceptions from workflows is a practical but difficult problem. In this paper, we have proposed an architecture model for handling such workflow exceptions. When an exception occurs, the exception handler first searches for a suitable ECA rule that can be triggered. If no such rule can be found, it then searches the previous exception cases and identifies a small subset that resembles the current exception. We characterized an exception by a set of attributes and allowed users to specify a criterion for retrieving matching records. This criterion includes a set of predicates on some attributes and a set of importance weights on candidate attributes. Exception records that satisfy these predicates are ranked according to a similarity measure defined by considering candidate attribute values and their importance weights. Three algorithms, namely, SEQ-E, SEQ-C, and BIN-C, have been derived for finding past exceptions that are similar to the current one. Their relative performance has been compared both theoretically and by experimenting with synthetic data sets. We concluded that BIN-C is an attractive approach, especially when the number of past exception records is huge. BIN-C provides considerable benefits in terms of the execution time, and sacrifices very little in terms of the quality of the returned exception record(s).

As we described earlier in this paper, unexpected exceptions can be handled through a two-step procedure in the real world. Firstly, the approaches proposed in this paper are employed in order to obtain a set of similar past exceptions. Secondly, these exceptions are reviewed and an appropriate approach is decided upon for handling the new exception. The focus of this paper has been on the first step. However, the second step is at least as important as the first one, since it usually involves a great deal of human input. Our future work includes the development of a structured way for performing this step. The preliminary idea is to first derive a naïve exception handler from these similar exceptions and then adopt case-based reasoning to adapt this naïve handler to handle the current exception.

Acknowledgements

This work is supported in part by the National Science Council in Taiwan under Grant no. NSC90-2213-E-110-022.

References

[1] G. Alonso, M. Kamath, D. Agrawal, A. E1 Abbadi, R. Gunthor, C. Mohan, Failure Handling in Large Scale Workflow Management Systems, IBM Tech. Report RJ9913, 1994.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seiger, The R*-tree: an efficient and robust access method for points and rectangles, Proc. of ACM SIGMOD Conf. on Management of Data, ACM Press, Atlantic City, NJ, 1990.

[3] S. Berchtold, D. Keim, H.-P. Kriegel, The X-tree: an index structure for high-dimensional data, Proc. of the 22nd Conf. on VLDB, Morgan Kaufmann, Mumbai (Bombay), India, 1996.

[4] M.J. Berry, G. Linoff, Data Mining Techniques: For Marketing, Sales, and Customer Support, Wiley, Indianapolis, 1997.

[5] K. Bradley, R. Rafter, B. Smyth, Case-based user profiling for content personalization, Proc. of the AAAI-98 Workshop on Recommender Systems, Springer, Trento, Italy, 1998.

[6] F. Casati, S. Ceri, S. Paraboschi, G. Pozzi, Specification and implementation of exceptions in workflow management systems, ACM Trans. Database Syst. 24 (3) (1999, Sep.) 405–451.

[7] D.K.W. Chiu, Q. Li, K. Karlapalem, A meta modeling approach to workflow management systems supporting exception handling, Inf. Syst. 24 (2) (1999, April) 159–184.

[8] D.K.W. Chiu, Q. Li, K. Karlapalem, Web interface-driven cooperative exception handling in adome workflow management system, Inf. Syst. 26 (2) (2001, April) 93–120.

[9] Special issue Data Placement for Parallelism, Bull. Tech. Comm. Data Eng. 17 (3) (1994, Sep.).

[10] T. Davenport, Process Innovation—Reengineering Work through Information Technology, Harvard Business School, Boston, MA, 1993.

[11] J. Eder, W. Liebhart, Contributions to exception handling in workflow systems, Proc. of EDBT Workshop on Workflow Management Systems, Valencia, Spain, Springer, Valencia, Spain, 1998.

[12] A.K. Elmagarmid, Database Transaction Models for Advanced Applications, Morgan Kaufmann Publishers, California, 1992.

[13] D. Georgakopoulos, M. Hornick, A. Sheth, An overview of workflow management: from process modeling to workflow automation infrastructure, Distrib. Parallel Databases 3 (3) (1995) 119–153.

[14] C. Hagen, G. Alonso, Exception handling in workflow management systems, IEEE Trans. Softw. Eng. 26 (10) (2000, Oct.).

[15] J. Han, Y. Cai, N. Cercone, Knowledge discovery in databases: an attribute-oriented approach, Proc. of the 18th VLDB Conf., Vancouver, British Columbia, Canada, Morgan Kaufmann, Vancouver, Canada, 1992, pp. 547–559.

[16] J. Han, Y. Cai, N. Cercone, Data-driven discovery of quantitative rules in relational databases, IEEE Trans. Knowl. Data Eng. 5 (1993, Feb.).

[17] N. Katayama, S. Satoh, The SR-tree: an index structure for high-dimensional nearest neighbor queries, Proc. of ACM SIGMOD Conf., ACM Press, Tuczon, AZ, 1997, pp. 369–380.

[18] E. Knorr, R. Ng, Finding intensinal knowledge of distance-based outliers, Proc. of 25th VLDB Conf., Morgan Kaufmann, Edinburgh, Scotland, 1999, pp. 211–222.

[19] A. Koenig, B. Stroustrup, Exception handling in C++, (revised) Proc. of USENIX C++ Conf., San Francisco, (1990) 149–176.

[20] K. Lin, H.V. Jagadish, C. Faloutsos, The TV-tree: an index structure for high-dimensional data, VLDB J. 3 (4) (1994, Oct.) 517–542.

[21] J.A. Miller, D. Palaniswami, A.P. Sheth, K.J. Kochut, H. Singh, WebWork: METEOR2's web-based workflow management system, J. Intell. Inf. Syst. 10 (2) (1998, March/April) 185–215.

[22] N.W. Paton, O. Diaz, Active database systems, ACM Comput. Surv. 31 (1) (1999, March).

[23] M. Reichert, P. Dadam, ADEPT-supporting dynamic changes of workflows without losing control, J. Intell. Inf. Syst. 10 (2) (1998, March/April).

[23] M. Reichert, P. Dadam, ADEPT-supporting dynamic changes of workflows without losing control, J. Intell. Inf. Syst. 10 (2).

[24] M. Reichert, C. Hensinger, P. Dadam, Supporting adaptive workflows in advanced application environments, EDBT workshop on Workflow Management System, Valencia, Spain, Springer, Valencia, Spain, 1998.

[25] N. Roussopoulos, S. Kelley, F. Vincent, Nearest neighbor queries, Proc. of ACM SIGMOD Conf. on Management of Data, ACM Press, San Jose, CA, 2000, pp. 427–438.

[26] R. Srikant, R. Agrawal, Mining quantitative association rules in large tables, Proc. of the ACM SIGMOD Conf. on Management of Data, ACM Press, Montreal, Canada, 1996, pp. 1–12.

[27] A. Sheth, K.J. Kochut, Workflow applications to research agenda: scalable and dynamic work coordination and collaboration systems, Proc. of NATO Advanced Study Institute on Workflow Management Systems and Interoperability, Istanbul, Turkey, Springer, Istanbul, Turkey, 1997, pp. 35–59.

[28] A. Silberschatz, H. Korth, S. Sudarshan, Database System Concepts, McGraw-Hill, New York, 1997.

[29] J. Tang, S.-Y. Hwang, A scheme to specify and implement adhoc recovery in workflow, Proc. of 6th Int'l. Conf. on Extending Database Technologies, Springer LNCS, Valencia, Spain, 1998, p. 1377.

[30] D.A. White, J. Jain, Similarity indexing with the SS-tree, Proc. of 12th Int. Conf. on Data Engineering, New Orleans, LA, IEEE Computer Society, New Orleans, LA, 1996, pp. 516–523.

[31] Workflow Management Coalition, The Workflow Reference Model, Workflow Management Coalition, Belgium, 1994.

San-Yih Hwang received his BS and MS degrees from National Taiwan University, Taiwan, in 1984 and 1988, respectively, and his PhD degree from the University of Minnesota, Minneapolis, in 1994, all in computer science.

He is presently an associate professor in the Department of Information Management, National Sun Yat-Sen University, where he initially joined in 1995. Between 1994 and 1995, he was with the Computer and Communication Laboratory, Industrial Technology Research Institute (CCL/ITRI), Taiwan. His current research interests include workflow systems, data mining, and data management aspects in mobile computing.

Jian Tang received his MS degree from the University of Iowa in 1982, and his PhD from the Pennsylvania State University in 1988, both from the Department of Computer Science. Since 1988, he has been a faculty member at the Department of Computer Science, Memorial University of Newfoundland, Canada, where he is currently a professor. He is currently working in the Department of Computer Science and Engineering, the Chinese University of Hong Kong, as a visiting scholar. His research interests include database systems, workflow management, data mining, e-commerce transactions.