Knowledge and Information Systems (2001) 3: 225–251
© 2001 Springer-Verlag London Ltd.

Knowledge and Information Systems

ActiveCBR: An Agent System That Integrates Case-Based Reasoning and Active Databases

Sheng Li and Qiang Yang

School of Computing Science, Simon Fraser University, Burnaby, BC, Canada

Abstract. Case-based reasoning (CBR) is an artificial intelligence (AI) technique for problem solving that uses previous similar examples to solve a current problem. Despite its success, most current CBR systems are passive: they require human users to activate them manually and to provide information about the incoming problem explicitly. In this paper, we present an integrated agent system that integrates CBR systems with an active database system. Active databases, with the support of active rules, can perform event detection, condition monitoring, and event handling (action execution) in an automatic manner. The integrated ActiveCBR system consists of two layers. In the lower layer, the active database is rule-driven; in the higher layer, the result of action execution of active rules is transformed into feature–value pairs required by the CBR subsystem. The layered architecture separates CBR from sophisticated rule-based reasoning, and improves the traditional *passive* CBR system with the *active* property. The system has both real-time response and is highly flexible in knowledge management as well as autonomously in response to events that a passive CBR system cannot handle. We demonstrate the system efficiency and effectiveness through empirical tests.

Keywords: Active databases; Agent system; Case-based reasoning

1. Introduction

Many case-based reasoning (CBR systems (Kolodner, 1993) follow a user-interaction model. In this model, a user provides all the information necessary for the CBR system to draw a conclusion. This '*passive*' nature of interactive CBR requires the direct involvement of human users in order to provide information about the incoming problem explicitly. This *passive mode* on system activation in some circumstances limits the system's ability to perform reasoning in a timely

Received 21 Apr 2000
Revised 12 Jun 2000
Accepted 14 July 2000

fashion, especially in applications such as forest fire protection and coastal salvage in which the group profile of events and data trigger the operation of cases in a CBR system.

To solve these problems, we integrate an active database system (Widom and Ceri, 1996) with a CBR system. An active database system is a database system that monitors situations of interest, and when they occur triggers an appropriate response in a timely manner. An active rule in active databases extends the expert system rules with the ability of autonomously responding to events, e.g., the modifications of data table such as *INSERT, DELETE,* and *UPDATE*. It generally follows the *Event-Condition-Action* paradigm.

on *event*	# event detecting
if *condition*	# condition monitoring
then *action*	# action executing

This paper presents an `ActiveCBR` agent architecture that builds a CBR subsystem on top of an active database. The system realizes problem solving based on data and events in a relational database. The `ActiveCBR` system developed under this architecture consists of two layers. In the lower layer, the active database subsystem is rule-driven. A CBR subsystem is located in the top level where the result of action execution of active rules is transformed into feature–value pairs required for the reasoning procedure. The resultant system possesses the reactive, autonomous and adaptive properties that give rise to an agent system (Jennings et al., 1998). To the best of our knowledge, this work presents the first attempt to integrate the active databases and CBR in an agent system.

The rest of the paper is organized as follows. Section 2 addresses the knowledge representation and subsystem algorithms of the proposed `ActiveCBR` system. Section 3 describes a two-layer architecture of the `ActiveCBR` system. In Section 4, system flexibility will be discussed. In Section 5, empirical test results are presented in order to evaluate the `ActiveCBR` architecture and underlying subsystems. Finally, Section 6 concludes the paper with a summary of our study. Some future research directions are presented to close the paper.

2. `ActiveCBR`: Representation and Algorithms

In this section, we present the detailed functionality of the `ActiveCBR` system. We focus our discussion on the broader areas of knowledge representation and the algorithms in subsystems. Examples are given to illustrate the discussions.

2.1. Active Databases

Active database, a database system with reactive behavior, has been the subject of extensive research recently (Widom and Ceri, 1996). The knowledge model of an active database system determines *what* can be said about active rules in the system. In contrast, the execution model indicates *how* a set of rules behaves at runtime. Diaz et al. (1994) discuss different *sources* of event that are monitored. The execution of the rules depends on the event–condition and condition–action *coupling modes*, which can be *immediate*, *deferred* or *detached* (Diaz and Jaime, 1997). The *transition granularity* describes the relationship between events and the rule execution. It can be *tuple-oriented* or *set-oriented*.

Termination is a key design principle for active rules. Due to the unexpected interactions between rules, termination is difficult to ensure even after a careful design. Triggering graphs are used for reasoning rules about termination. A rule set is *confluence* when any triggering of rules produces a unique final database state independent to the order of execution of the rules. A rule set guarantees *observable determinism* when all visible actions performed by rules are the same for any order of execution of the rules.

An additional area of future research is to explore the relationship between this work and that of agents (Jennings et al., 1998). It can be observed that the ActiveCBR system has the property that it is reactive, adaptive and autonomous. It would be interesting to explore dimensions along which the system can be made proactive as well. In this mode, the system will not only be adaptive to external events, but it will also proactively perform actions to meet high-level goals.

As an example of an active database system, the *Chimera* system (Ceri et al, 1996) integrates an object-oriented data model, a declarative query language, and an active rule language. It supports *display*, *generalize*, and *specialize* events in addition to traditional *create*, *delete*, and *modify* primitives to reflect object manipulation.

2.2. Knowledge Representation in the ActiveCBR System

In traditional CBR systems, knowledge is represented with cases. A case records a previous situation, problem pattern and/or solution. In rule-based systems, including the rule system in active databases, knowledge is abstracted in the form of inductive rules. A rule depicts the outcome under particular conditions. Since the ActiveCBR system integrates both CBR and rule-based methods, the knowledge under the ActiveCBR architecture is naturally represented by the combination of cases and active rules.

In this section, we consider the knowledge representation and algorithms for the active CBR system. This system consists of two subsystems – the case base (CB) subsystem is close to the user interface layer, and the active database (ADB) subsystem is close to the database system level. We consider the system architectures in the next section.

2.2.1. Case Representation

The representation of a case has various forms depending on different applications. In the ActiveCBR system, a case base in ActiveCBR system is a combination of $\{\mathscr{C}, \mathscr{F}, \mathscr{I}\}$, where $\mathscr{C}$, $\mathscr{F}$, and $\mathscr{I}$ are case space, feature space, and weight space, respectively. We describe the spaces $\{\mathscr{C}, \mathscr{F}, \mathscr{I}\}$ as follows.

- The case space $\mathscr{C} = \{c^m \mid m = 1, \ldots, M\}$ is the set of M case specifications. A specification of a case consists of *Name, Description, Threshold,* and *Solution.*
- The feature space $\mathscr{F} = \{(f_n, v_{n,k}) \mid n = 1, \ldots, N;\ k = 1, \ldots, K_n\}$, is the set of feature–value pairs, where N is the total number of features, and K_n is the number of possible values of feature f_n. In the case representation of the ActiveCBR system, all the feature values are symbolic. For the features with original numeric context, we transform them into discrete symbolic values.
- The weight space $\mathscr{I} = \{\omega(m, n, k) \mid m = 1, 2, \ldots, M;\ n = 1, \ldots, N;\ k = 1, \ldots, K_n\}$ is

Table 1. A sample case of the travel agent domain.

Name		TravelCase31
Description		#245
Threshold		85
Solution		Hotel Golden Coast, Attica
Feature	**Value**	**Weight**
JourneyCode	649	0
Price	$1,000-2,499	80
HolidayType	Recreation	35
NumOfPerson	1–2	70
Region	Germany	75
Transportation	By plane	45
Duration	5–7 days	85
Season	Summer	65
Accommodation	Luxury	70

the set of feature–value weights. A weight is a real number between 0 and 1.[1] Therefore, we can consider $\mathscr{I}$ as such a relation:

$$\mathbf{R} : \mathscr{C} \times \mathscr{F} \to [0, 1]$$

Having the definition of the case base, we can represent a case by two parts: the specification part from an element in $\mathscr{C}$ that describes the name, description, threshold, and solution of the case; and the weight part from all the elements in $\mathscr{I}$ that related to this case, which describe the similarity property of the case.

A *threshold* is introduced into the `ActiveCBR` system as a new field in case specification. It represents the minimum score to which the case is detected at runtime and should be fired accordingly. We will discuss the similarity and score computation in the next subsection.

Examples. An example case in AI-CBR's travel agents domain[2] is shown in Table 1.

The travel agents case base is used to help travel agents to recommend a hotel destination for customers based on their individual interest and requirement. The solution of each case is a hotel destination. The similarity property of the example case is described by the nine feature–value–weight triples. Note that all the feature values are symbolic. Some features, such as *Price*, *Duration*, and *NumOfPerson*, have original numeric values. In this case base, each feature has only one value with a positive weight. Feature *JourneyCode* is used for indexing purposes only and no positive weight is assigned. (We have omitted other feature–value pairs with zero weight in the table.)

Another example case is from a cable TV domain used by a cable company (shown in Table 2). In the cable domain, a case can have multiple positive weights for different values on a particular feature. For instance, both values '*no picture*' and '*reception*' of the feature '*ProblemType*' are related to the case '*Regional switch*

[1] In the internal representation of the `ActiveCBR` system, the weights are converted to integers between 0 and 100, and the threshold in case specifications is also defined as an integer between 0 and 100.

[2] The travel agents case base is downloaded from AI-CBR's case base archive (*http://www.ai-cbr.org/cases.html*).

Table 2. A sample case of the cable TV domain.

Name:		Regional switch (LB) problem
Description:		Low band regional switch is breakdown
Threshold:		78
Solution:		Generate ticket for technician
Feature	**Value**	**Weight**
ProblemType	no picture	75
	reception	65
	VCR problem	0
Channels	lower band	80
	upper band	0
	US channel	0
Duration	recent 24 hrs	70
	recent 1 week	45
	not specified	0
Location	particular	85
	not specified	0

Table 3. Trigger representation in SQL Server and Oracle.

SQL Server	Oracle
CREATE TRIGGER *trigger-name*	**CREATE TRIGGER** *trigger-name*
ON *table-name*	**BEFORE** | **AFTER** *trigger-event*
FOR *trigger-event*	**ON** *table-name*
	[FOR EACH ROW
	[WHEN (*condition*)]]
AS *Transact-SQL block*	*PL/SQL block*

(LB) problem', but the former has a higher possibility, so it is assigned a higher weight. We will discuss the meaning and usage of the feature weights further in Section 2.3.

2.2.2. Rule Representation

The representation of an active rule in the ADB subsystem depends on its underlying RDBMS. Both Oracle and SQL Server use triggers to perform the rule mechanism. A trigger in ADB subsystem is a special kind of stored procedure that is executed automatically when the specified data modification occurs on the specific table. One trigger can contain one rule or several rules raised by the same event. A rule can be an ECA rule with complete event–condition–action semantics, or an E-A rule, in which the condition is implicitly specified by the database query language in a trigger.

The creation of a trigger is shown in Table 3. In both contexts, the *trigger-event* could be one of the three data manipulation operations: *INSERT*, *UPDATE* and *DELETE*. An example of trigger on UPDATE events is shown in Section 2.2.3.

The trigger format in SQL Server is the minimum implementation of the active rules. There is no explicit clause to define rule conditions. In fact, all the conditions and actions are written in the Transact-SQL block in the **AS** clause. Since multiple rules, as long as they monitor the same event, can be

clustered into one trigger, this expression greatly simplifies the interaction among the triggers. However, one drawback of SQL Server's trigger syntax is its limited capacity to monitor complicated applications. Another drawback is its relatively low efficiency due to the mandatory execution of the Transact-SQL block even before the condition evaluation. The rule set of the `ActiveCBR` system is 'simple' enough to be handled by the trigger in SQL Server. We consider it simple because the action of the rules in the `ActiveCBR` system will not modify the triggering data table itself, so that we can guarantee the termination of the rule execution.

The capacity of triggers in Oracle is expanded in several dimensions:

- A trigger can be evaluated and executed either *before* or *after* the triggering operation event.
- It supports row-level granularity, in which triggering occurs for each row that is affected by the operation event.
- If the condition involves only one row-level predicate, it can be written in the **WHEN** clause. Otherwise, it has to be written in the PL/SQL block.

A limitation of current trigger systems is that the actions of rules has to be written in SQL language, Transact-SQL in SQL Server and PL/SQL in Oracle. Native languages like Java and C may be supported in the future.

2.2.3. An Example of Rules

Table *ACBR_TRAVEL_DATA* stores the user data of the travel agent domain. The attributes of *ACBR_TRAVEL_DATA* are one-to-one mapping to the features of the travel agent case base. The distinction between the two is that the raw data in the user table can be either symbolic or numeric, e.g., attribute *price* can be any positive real number, while the value of feature *Price* is generalized and converted to a symbolic value like *high* or *over 8000*.

Table *ACBR_TRAVEL_FEATURE* is used to keep the current values of each feature. The action part of the active rules updates the feature values in the feature table *ACBR_TRAVEL_FEATURE*.

An example rule literally like *If the new record has price over USD8000, set the Price of FEATURE table to 'high'* can be represented as an SQL Server trigger:

```
CREATE TRIGGER INSERT_TGGR
ON ACBR_TRAVEL_DATA FOR INSERT
AS
BEGIN
      /* other rules */
      ⋮
      IF (new.price > 8000)
      UPDATE ACBR_TRAVEL_FEATURE SET Price = 'high'
      ⋮
      /* other rules */
END
```

Note that multiple rules are generally stored in one trigger for the INSERT event. In Oracle, it is possible to map one rule to one trigger:

```
CREATE TRIGGER PRICE_HIGH_TGGR
AFTER INSERT ON ACBR_TRAVEL_DATA
FOR EACH ROW WHEN (new.price > 8000)
BEGIN
    UPDATE ACBR_TRAVEL_FEATURE SET Price = 'high'
END
```

2.3. Algorithms for the ActiveCBR Agent System

The algorithms in the two subsystems of the ActiveCBR systems are independent. The *Case Authoring* module in the CB subsystem and the *Rule Definition* module in the ADB subsystem primarily work on system reconfiguration for system flexibility. We will mainly, in this section, discuss the algorithms in *Case Firing* module and *Rule Execution* module.

2.3.1. Algorithm in the CB Subsystem

Suppose we have M cases in the case base $\mathscr{C} = \{c^1, ..., c^M\}$. For the N features $f_1, ..., f_N$, let f_n^m denote the value of the n-th feature of the m-th case, and f_n^I denote the current input value of the n-th feature. Now we have:

Case Firing Algorithm:

1. *For each new case added at runtime, mark it as enabled;*
2. *Retrieve current feature values* f_n^I*;*
3. *For each case* c^m *in* $\mathscr{C}$ *that is marked enabled:*
   (a) *For each feature* f_n*:*

      Calculate the similarity $sim(f_n^m, f_n^I)$*;*

   (b) *Calculate the score of case* c^m*;*
   (c) *Mark* c_m *as fired, if the score of* c^m *is greater than its threshold;*
   (d) *Update the firing history log, if necessary;*
4. *Visualize case firing monitor.*

The CBR subsystem provides a user interface to add a new case and change the enabled / disabled status of an existing case at runtime. Before the case retrieval iteration, Step 1 of the case firing algorithm examines whether a new case has been added into case base and, if so, marks it as enabled. This operation maximizes the system flexibility to perform real-time knowledge management.

In Step 2, runtime feature values are selected from the feature table or obtained from the *CBFeature* object in shared memory.

Step 3(a,b) is the case retrieval algorithm we used in the CBR subsystem, which is a variation of the k-nearest neighbor matching algorithm (Kolodner, 1993). Details of this algorithm are as follows.

To calculate the similarity $sim(f_n^m, f_n^I)$ between the input feature value and a case feature value, a set of quartet vectors $< m, n, k, \omega(m, n, k) >$ is used to represent the similarity in terms of the feature-value weights. A feature-value weight $\omega(m, n, k)$ in weight space $\mathscr{I}$, between 0 and 1, represents the contribution on case c_m if the value of the n-th feature is the k-th possible value.

If the n-th feature has K_n possible values, the importance weight w_n^m of the n-th feature for the m-th case is given by

$$w_n^m = max\{\omega(m,n,k) \mid k = 1, ..., K_n\} \tag{1}$$

The similarity in Step 3(a) can be calculated as (Kolodner, 1993)

$$sim(f_n^m, f_n^I) = \frac{\omega(m,n,f_n^I)}{w_n^m} \tag{2}$$

In Step 3(b), the score of case c^m is given by

$$score(m) = \frac{\sum_{n=1}^{N} w_n^m \times sim(f_n^m, f_n^I)}{\sum_{n=1}^{N} w_n^m} \tag{3}$$

Step 3(c) marks firing status of the case, and Step 3.d updates the history log table when the case is fired or terminates fired status.

For the iteration in Step 3, the complexity of the current case-firing algorithm is $O(M)$, where M is the size of the case base, i.e., the number of cases. In addition, the execution time of the algorithm is also determined by the total number of feature-value weights in the weight space $\mathscr{I}$. It is obvious that for the computation in Step 3(b) and Equation (2), the total number of feature-value weights is

$$|\mathscr{I}| = M * \sum_{n=1}^{N} K_n \tag{4}$$

Let κ be the average number of possible values of each feature. The total number of feature-value weights is

$$|\mathscr{I}| = M * N * \kappa \tag{5}$$

Therefore, the complexity of above case-firing algorithm is $O(MN\kappa)$; i.e., the algorithm is linear in terms of number of cases, number of features, and average number of values for each feature.

The linear algorithm can be improved if we have an appropriate approach to cluster the cases (Quinlan, 1986). If the clustering is effective, we can guarantee that no case from different clusters can be fired simultaneously. Hence, we need not traverse the whole case base in the case-firing algorithm; alternatively, we need just traverse along the clustering path until entering a cluster, and check the firing condition with the cases in this cluster only. On the best case, we can suppose the clustering is even and the search tree is balanced. If, on average, each cluster contains λ cases, we can improve the computation to

$$O(\lambda * log\left(\frac{M}{\lambda}\right) * N * \kappa) \tag{6}$$

In the last step of the algorithm, the updated case-firing information is sent to the user interface for display.

2.3.2. Algorithm in the ADB Subsystem

The algorithm in the ADB subsystem depends on the trigger property of the underlying relational database. We implement the `ActiveCBR` system based on two RDBMS: SQL Server and Oracle.

The rules in Oracle support two distinct granularities, *row-level* and *statement-level*, corresponding to the instance-oriented semantics and the set-oriented semantics, respectively. They also can be executed either *before* or *after* the triggering operation. Thus, there are four possible combinations by combining the two granularities and the two evaluation times, i.e., *before* and *after* (Owens and Adams, 1994). Besides the rules, database built-in integrity checking is also executed when a database manipulation occurs.

The rule-processing algorithm in Oracle is as follows.

Oracle Rule-Processing Algorithm:

1. *Execute the statement-level before-rules;*
2. *For each row in the target table:*
   (a) *Execute the row-level before-rules;*
   (b) *Perform the modification of the row and row-level referential integrity and assertion checking;*
   (c) *Execute the row-level after-rules;*
3. *Perform the statement-level referential integrity and assertion checking;*
4. *Execute the statement-level after-rules.*

SQL Server has some limitations, namely, each operation on a table can be monitored by at most one trigger. In addition, there is no distinction on granularity and evaluation time in SQL Server. Therefore, all the rules for *INSERT* event have to be written in one trigger, in which each rule is in charge of updating one possible value of a feature. The rule-processing algorithm in SQL Server is relatively straightforward:

SQL Server Rule-Processing Algorithm:

1. *Wait for event triggering;*
2. *Preprocess user data;*
3. *For each rule in the trigger for current event:*
   (a) *Evaluate the condition for this rule;*
   (b) *If the condition is true, update corresponding feature value;*
4. *Repeat (1) to (3).*

In Step 1, the ADB subsystem detects events on user tables. The preprocessing on user tables in Step 2 includes slicing, i.e., removing irrelevant tuples, and dicing, i.e., removing irrelevant attributes. By reducing the size of user tables, we can enhance the query efficiency and speed up the condition evaluation.

Finally, in Step 3(a,b), the ADB subsystem monitors condition in the form of SQL predicates and query and updates feature values for the similarity computation in the higher-layer CB subsystem.

This algorithm is also used in Oracle when we put multiple rules into the same trigger. As we have discussed in Section 2.2, an active rule can be represented in one Oracle trigger with rule condition in the **WHEN** clause; or we can also put several rules into one PL/SQL block of one trigger.

Table 4. User data of cable domain.

pid	uid	active	type	channel	time	location	solved
6732	263	Y	No picture	6	10/05 19:06	Burnaby	N
6733	546	Y	Reception	all	10/13 12:51	N.Van	Y
6734	649	Y	VCR	15	10/17 21:34	N.Van	N
6735	032	Y	VCR	n/a	10/19 02:25	Burnaby	N
6736	382	N	Reception	50	10/19 21:45	Burnaby	N
6737	234	Y	Reception	all	10/20 16:23	W.Van	N
6738	271	Y	Reception	9	10/20 20:42	Burnaby	N
6739	031	Y	No picture	13	10/20 22:19	Burnaby	N
6740	740	Y	VCR	11	10/20 22:43	UBC	N
6741	638	Y	VCR	28	10/20 23:19	SFU	N
6742*	957	Y	Reception	3	10/20 23:32	Burnaby	N
6743*	271	Y	Reception	6	10/20 23:57	Burnaby	N

2.3.3. An Example

In this section, we provide a comprehensive example in the cable domain to demonstrate how the `ActiveCBR` system works.

The user data is stored in data tables such as *ACBR_CABLE_DATA*. The content of *ACBR_CABLE_DATA* is modified at runtime, e.g., insertion when a new problem is reported, updating when the problem is solved later, and deletion when record is out of date. Consider that two new tuples with problem id (pid) *6742* and *6743* are inserted into *ACBR_CABLE_DATA* (marked with * in Table 4). An *INSERT* event occurs accordingly.

The preprocessing performs slicing and dicing operations on user data to reduce the size of the query table. For instance, in the slicing operation, the tuples with 'N' value of *active* attribute are excluded from further query; in the dicing operation, all the attributes that are not related to rule conditions such as *pid* and *solved* are removed as well. A temporary table #USERDATA is created in the preprocessing, and the number of tuples in #USERDATA is counted into variable *@num*.

```
SELECT type, channel, time, location
   INTO #USERDATA                    /* user data after preprocessing */
   FROM ACBR_CABLE_DATA
   WHERE active = 1
SELECT @num = COUNT(*)               /* number of tuples */
   FROM #USERDATA
```

Next, rule conditions are evaluated upon the *#USERDATA* table. In the `ActiveCBR` system, the values of features of the higher-level case base are updated dynamically by the rule action from the lower-level active database. For each feature in the feature space, there are several active rules to be evaluated, since only one of the conditions could be true, the feature is set to a corresponding value.

Consider the case in Table 2. For feature *Duration*, its value can be '*recent 12 hrs*', '*recent 3 days*', and '*not specified*'. The rule used to update feature *Duration* to '*recent 12 hrs*' can be described as

Table 5. Runtime feature values and similarity computation of case *Regional switch (LB) problem.*

	ProblemType	Channels	Duration	Location
Current Value f_n^I	reception	lower band	recent 12 hrs	particular
Importance weight w_n^m	75	80	70	85
Feature-value weight $\omega(n, m, f_n^I)$	65	80	70	85
Similarity $sim(f_n^m, f_n^I)$	0.87	1.00	1.00	1.00

on INSERT
if At least 1/3 of total tuples and at least 5 tuples are reported within the last 12 hours.
then Update feature *Duration* with value '*recent 12 hrs*'.

The rule can be represented as a block of SQL statements in the *INSERT* event trigger:

```
    ⋮
SELECT @npart = COUNT(*)
    FROM #USERDATA
    WHERE DATEDIFF(hour, time, GETDATE()) < 12
IF ((@npart * 3 > @num) AND (@npart > 5))
BEGIN
    UPDATE ACBR_CABLE_FEATURE
        SET value = 'recent 12 hrs'
        WHERE feature = 'Duration'
END
    ⋮
```

After all rules are executed, we have the runtime feature values as shown in row *Current Value* in Table 5.

For each feature, there may exist multiple positive feature-value weights. For instance, referring to Table 2 in the previous section, feature *ProblemType* has two feature-value weights: 75 for value *no picture* and 65 for value *reception*. The importance weight w_n^m for feature n (*ProblemType*) of case m (*Regional switch (LB) problem*) is the maximum of the value weights, i.e., 75 in this example.

The result of similarity computation according to Equation (2) is listed in the last row in Table 5. Accordingly, we can compute the score of this case by Equation (3). In the above example, the resulting score is 96. Since it is greater than the threshold 78, the case *Regional switch (LB) problem* is marked as '*fired*.

The system will fire and visualize every case whose score exceeds its threshold at runtime.

2.4. Agent System Properties

The resultant system qualifies to be an agent system (Jennings et al., 1998) because it has the following properties:

- *Reactivity:* An agent is capable of perceiving its environment through its sensors, and responding through its effectors to changes that occur in the environment in a timely manner. The sensors are not directly included in this architecture, but can be easily incorporated by connecting the sensors with the database relational tables. Furthermore, the cases in the case base can contain actions that are activated once a case is fired – the actions can be used to attain goals that the user wants to achieve, maintain or protect.
- *Autonomy:* When expected or limited unexpected events occur in an open, unpredictable environment, the `ActiveCBR` agent is capable of independently selecting and carrying out some set of operations without direct input at runtime from humans (or other agents). This property is clear from the design of the system.
- *Adaptiveness:* Over time, on the basis of previous experience, the `ActiveCBR` agent is able to adapt its behavior to suit the preferences and behavior of its users. This can be done by updating the weights in the case base, or adding new cases that the system learns from its past experience.

The `ActiveCBR` agent system is related to other work in the agent area. Sycara and Zeng (1996) classify agents into three groups according to with whom or what the agent generally communicates. Interface agents, such as the Amalthaea agents developed at MIT, communicate mainly with a single user (Moukas, 1996). These agents model the user and initiate tasks on behalf of the user. Task agents communicate mainly with other agents in order to perform the task at hand. The information brokering agent of Retsina system (Sycara and Zeng, 1996) communicates with other agents for the purpose of agent brokering, while researchers at SRI International have developed task agents which handle plans and planning-related activities (Wilkins and Myers, 1998). The information agent is closely coupled with one or more data sources. Similar to the job performed by a librarian, the job of these agents is to interpret an information request, obtain the required information from a data source and present the information in an understandable manner to the requestor (Arens et al., 1993; Etzioni and Weld, 1994; Hammer et al, 1997). Under this classification, we can consider the `ActiveCBR` agent as a task agent that performs useful tasks (e.g., diagnosis for cable TV) when connected to the external environment or through communication with other information-gathering agents.

3. Architecture and System Design

Traditional ‘passive’ CBR systems suffer from the difficulty of handling massive amounts of user data at runtime, since they lack the ability to respond to the alteration in external data source in an automatic manner. To address this problem, we propose an `ActiveCBR` architecture and an active CBR system implemented upon this architecture. In this section, we focus on system structure and composition. The internal knowledge representation and algorithms will be discussed in the next section.

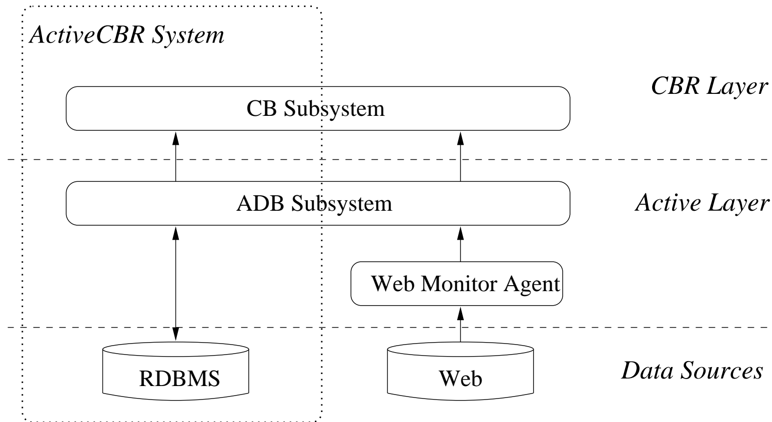

Fig. 1. The ActiveCBR architecture.

Generally, the rule execution in active database systems tends to be more difficult to understand and maintain when supporting more facilities. Even in a conservative-designed rule system like Starburst (Biliris, 1992), the semantics of rule execution are still quite complex. Rule termination and rule confluence are difficult to realize in a practical design of a large rule set. How to simplify the design and analysis of active rule sets is an important topic in active database research. With the ActiveCBR system, much of the user-level semantics are elevated to the CBR level, whereas the efficient rule-triggering mechanism is left to the database level. We will discuss this in detail in the next section.

3.1. ActiveCBR Architecture

The objective of the ActiveCBR architecture is to improve CBR systems to have the *active* property, i.e., the capability to respond to events or alterations of external data sources automatically and in a timely manner. With the active mechanism, the enhanced CBR system may not have to rely on human users to provide information of incoming problems and to activate the reasoning process manually. Instead, the system monitors the alterations of external data sources autonomously, and performs reasoning when an alteration is detected.

Figure 1 depicts a high-level view of the two-layer ActiveCBR architecture. Briefly, the procedures of CBR, such as case retrieval, case adaptation, and case maintenance, are performed in the higher *CBR Layer*, while the lower *Active Layer* encapsulates the reactive functionality to monitor the alterations in external data sources. The interaction between the two layers is carried through the feature-value pairs that are accessible to both layers. In the higher layer, feature-value pairs are used to describe the similarity between the new problem and the retained cases, while in the lower layer they reflect the result of data alterations.

– *CB subsystem* is a component in the CBR Layer that performs high-level problem solving by reusing and adapting previous knowledge retained in cases. Here, a new problem is a situation based on the real-time alteration of the data sources rather than a description provided by a customer or a user of the

system. We regard the procedure as a high-level process since the CB subsystem does not perform reasoning on the original data directly; instead, it obtains runtime feature values from the Active Layer, and retrieves cases based on the comparison between incoming features and case features stored in the case base.

We take two kinds of external data sources into consideration: relational databases and the World Wide Web. As shown in Fig. 1, each of the two components of the Active Layer, the ADB subsystem and the web monitor agent, is in charge of monitoring one of the above two data sources, respectively.

– *ADB subsystem*, as we have discussed in Section 2, is an active rule system that performs event detection, condition monitoring, and action execution on a database management system. In our architecture, the ADB subsystem uses triggers to trace the data modifications in one or more tables in the database, and updates feature values based on the rule conditions pre-compiled in the triggers.
– *Web monitor agent* is a software agent used to monitor one or multiple web sites in the `ActiveCBR` architecture. A software agent is '*a software entity which functions continuously and autonomously in a particular environment, often inhabited by other agents and processes*' (Shoham, 1997). It has several characteristics to enable the active duty:
  · *mobility:* migrating between different hosts on the Internet;
  · *reactivity:* sensing the alteration of external environment;
  · *inferential capability:* using prior knowledge (preset conditions) to infer new feature values based on current data;
  · *communication ability:* submitting the inferred results to higher layer system components.

  The structured data on web sites can be organized in data tables in the ADB subsystem. The web monitor agent is able to update the tables and trigger active rules in the ADB subsystem when an alteration occurs on the web sites. Therefore, the web monitor agent can activate the CB subsystem indirectly via the ADB subsystem.

3.1.1. Structure of `ActiveCBR` *System*

Now we concentrate our focus on how to integrate CBR and active database techniques together to construct the `ActiveCBR` system. Figure 2 depicts the structure of the `ActiveCBR` system. This structure is a concrete implementation of the `ActiveCBR` architecture described in the last section.

As shown in the figure, the system consists of the CB subsystem and the ADB subsystem, which correspond to the two layers in the `ActiveCBR` architecture. Both subsystems interact with the underlying relational database, and they communicate with each other as well.

CB Subsystem. The CB subsystem accepts inputs from a group of feature-value pairs updated dynamically by the lower-layer ADB subsystem. The feature-value pairs can be stored in a particular table in the underlying relational database, or they can be stored in shared memory that is accessible by both the CB subsystem and the ADB subsystem. The output of the CB subsystem is one or more *fired*

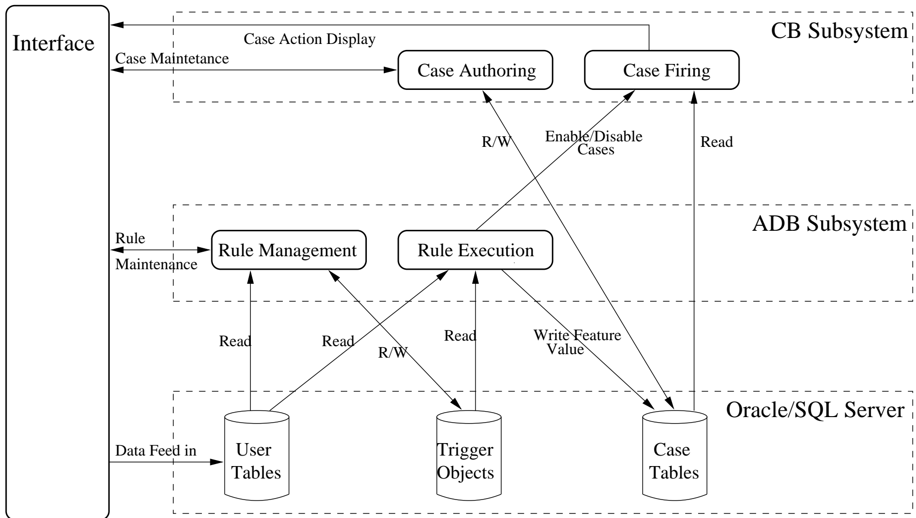

Fig. 2. Structure of the `ActiveCBR` system.

cases. A case is regarded as 'fired' when its runtime score computed by similarity comparison exceeds a pre-initialized threshold.

The CB subsystem consists of two modules:

- *Case Authoring* module manages the case base. User can add/delete cases, change the content of a case, adjust case threshold, and assign weights for different feature-value pairs. Through the *Case Authoring* module, knowledge base management can be performed in a relatively easy manner. New knowledge can be represented in a new case as long as we can define its similarity to other cases in the case base by specifying its feature-value weights. In addition, by adjusting the weights based on the previous reasoning result, the change of knowledge domain can be reflected in the case base.
- *Case Firing* module examines the real-time feature values from the feature table or the shared memory, performs similarity computation and calculates the runtime case scores; then fires a case when its score exceeds its threshold, or inhabits a case when its score reduces below the threshold. There is no interaction between the cases, since they are accessed sequentially. A user can also enable or disable a case through the *Case Firing* module at runtime. These operations further improve system flexibility. Another function of the *Case Firing* module is to record case firing history in a log file, so that users can examine the relationships among the fired cases.
  In the current implementation, the case firing information is sent to the system interface and an alert with the case solution is displayed to users. In future implementations, the output can be forwarded to a case-based planning module that can generate a series of actions towards the case solution automatically.

ADB Subsystem. The ADB subsystem has a means of utilizing the active rules for the application of CBR in the higher-layer CB subsystem. It separates the high-level knowledge that is represented in the form of cases from the raw data

in user tables. We name these active rules *feature rules*. Each feature rule in the ADB subsystem works in the following way:

- detecting one or more of the data manipulation operations in user tables. The operations can be *INSERT, UPDATE,* or *DELETE*;
- monitoring a condition in the form of a database query that makes sense to a high-level feature with a particular value; and
- once the condition is satisfied, updating the above feature to a particular value.

The ADB subsystem also has two modules:

- *Rule Management* module is in charge of the administrative operations on the active rules, e.g., adding a new rule, dropping an existing rule, and enabling or disabling an active rule. A user can examine the organization of user tables, and access database triggers to edit the active rules. Rule management is relatively more difficult than case management. This is because that the active rules generally involve very complicated database queries, and in many systems rules have to be complied into the database systems, so a rule cannot be modified at runtime.
- *Rule Execution* module is the kernel of the ADB subsystem. It accomplishes the mechanism of knowledge transformation from user data to the CBR Layer. The execution procedure is triggered when a data manipulation event is detected on specific user tables. Trigger objects in the underlying database are retrieved. All the active rules related to the event are executed sequentially, and the corresponding feature values in feature table or shared memory are updated for the use of the CB subsystem.

Subsystem Interaction. The interaction between the CB subsystem and the ADB subsystem is carried through the feature-value pairs that are visible to both layers. To perform high-level reasoning, each rule in the ADB subsystem is bound with one of the features in the CB subsystem, and the action of the rule is defined as the updates of the specific value of the bound feature.

We have two approaches to handle the interaction between the two subsystems:

- *Asynchronous mode* works in a top-down fashion. In this mode, the *Rule Execution* module and the *Case Firing* module operate independently. The interaction happens exclusively in the feature-values that are stored in public table or shared memory. In the Active Layer, the *Rule Execution* module updates the feature values based on the action of rules, while the *Case Firing* module in the CBR Layer examines these values periodically, hence performs the similarity computation and the score calculation, then fires appropriate cases.
- *Synchronous mode*, in contrast, works in a bottom-up fashion. Once the *Rule Execution* module triggers a rule, and if the condition is satisfied, a signal is created and sent upwards to the *Case Firing* module along with the result of feature-value updates after the rule execution. Subsequently, high-level case-base reasoning is invoked, and incurs case firing. However, most applications use asynchronous mode.

Primarily, the two approaches work similarly. However, the synchronous mode requires inter-process communication, thereby increasing the system complexity.

This is especially obvious when we extend the Active Layer to have multiple ADB subsystems. On the other hand, synchronous mode has a better real-time response property, since the CB subsystem need not retrieve runtime feature values periodically.

4. Analysis

In previous examples we have seen that the ActiveCBR system is truly reactive and adaptive. In this section we show that the system also has flexibility introduced through the simplification of knowledge management.

The knowledge management approach in the ActiveCBR system has two merits. The first one, as a static characteristic, is the simplicity of representing a massive amount of knowledge in terms of the number of cases and rules. The second one, involving the dynamic characteristic of knowledge management, is the simplicity of modifying the content of knowledge along with alteration of the external environment at runtime.

4.1. Two-Level Knowledge Structure

Knowledge in the ActiveCBR system is represented by both cases and active rules at different levels. We have also demonstrated how cases and rules are used to perform problem solving by an example in Section 2.3. Now we can conclude a natural strategy on how to define the knowledge base in a specific problem domain.

1. Investigate the problem domain and choose representative problems and solutions to construct initial cases.
2. Develop the case indexing method, establish feature values and set up initial weights.
3. Design a rule set in which a rule reflects an appropriate meaning of each feature-value pairs, i.e., for each value of each feature:
   (a) rule condition regulates a predicate; if the predicate is true, the specific feature should have the specific value;
   (b) rule action updates the feature with the value.

The first two steps set up a case base from scratch. The final step provides a guideline for rule design in the ADB subsystem; i.e., each active rule in the ADB subsystem should match one feature-value pair in the higher-level case base. Consider the example in Section 2.2; to design an active rule for a feature-value pair $(Price, high)$ in the travel agents case base, we can define the predicate $new.price > 8000$ as a rule condition, and the rule action is an assignment clause, UPDATE *feature_table* SET $Price =$ '*high*'.

Traditional rule-based systems have great difficulty in constructing a knowledge base. On one hand, a comprehensive understanding of the problem domain is hard; on the other hand, the designing of the rule set is also a labored procedure. Rule termination and modularization in active database systems are particularly difficult to ensure. In the ActiveCBR system, the semantics of active rules are well defined to perform feature-value updating. Hence, we have two observations to simplify the rule set design.

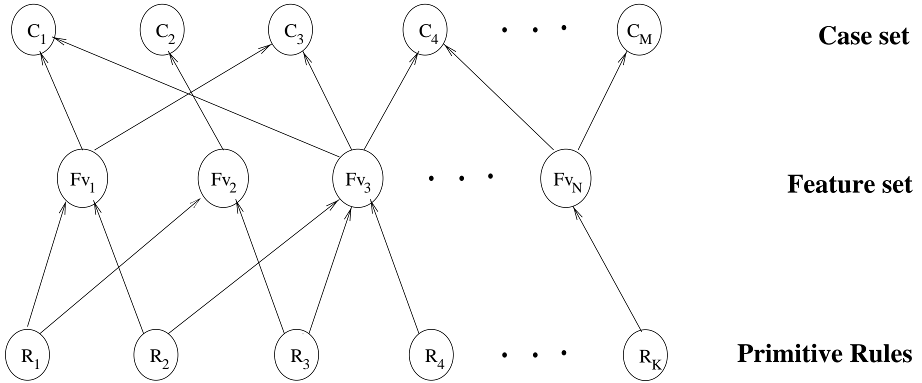

Fig. 3. Two-level knowledge representation.

- *The rule set in the ADB subsystem guarantees termination and confluence.* Termination is ensured since all active rules are triggered based on the events on user tables, while rule actions modify feature tables rather than the user tables. Confluence is ensured because whatever the sequence of rule execution activated, when an event occurs each feature rule in the conflict set is executed once and only once. Since the effect of each rule action is irrelevant, a unique database state will be obtained on the accumulating effect of rule actions.
- *The rule set in the ADB subsystem guarantees behavioral stratification for rule modularization.* After partitioning the active rules into different strata according to the problem domain, inter-domain rule conflicts are avoided in the `ActiveCBR` system. This is because the rule sets for different problem domains handle different user tables, so there is no interleaving between the rule executions from different strata.

The integrated knowledge representation method used in the `ActiveCBR` system has the advantage over active database systems when they perform the same problem-solving tasks.

Consider a problem domain described by a case base $\mathscr{C}$ with M cases, and each f_n of the N features has K_n values. Hence, in the ADB subsystem, we need a rule set $\mathscr{R}$ with $\sum_{n=1}^{N} K_n$ rules in which one rule updates one feature-value pair. Therefore, the domain knowledge can be represented in an integrated knowledge base $\mathscr{C} \cup \mathscr{R}$ with a size of

$$|\,\mathscr{C} \cup \mathscr{R}\,| = M + \sum_{n=1}^{N} K_n \tag{7}$$

The two-level knowledge representation is illustrated in Fig. 3, where K primitive rules in the ADB subsystem determine the value of N features, and N features determine the score of M cases.

In general, each case does not necessarily relate to all features. For feature f_n in Fig. 3, the number of incoming paths In_n is equal to the number of values of this feature K_n, and the number of outgoing paths Out_n is equal to the number of cases relating to this feature. Therefore, in a active rule system without the two-level knowledge representation concept, one rule is needed to represent each path from any primitive rule in the lower level to any case in the higher-level in

the figure. Hence, the total number of rules under this model in an active rule system is

$$| \mathscr{R}^* | = \left(\sum_{n=1}^{N} In_n \right) * \left(\sum_{n=1}^{N} Out_n \right) \tag{8}$$

Since each case relates to at least one feature and at most all of N features, we have the following approximate bounds for the number of paths from feature set to case set:

$$M \leqslant \sum_{n=1}^{N} Out_n \leqslant M * N \tag{9}$$

Consider $In_n = K_n$ for each feature f_n; we have the approximation

$$M * \sum_{n=1}^{N} K_n \leqslant | \mathscr{R}^* | \leqslant M * N * \sum_{n=1}^{N} K_n \tag{10}$$

Therefore, we have

$$| \mathscr{C} \cup \mathscr{R} | = M + \sum_{n=1}^{N} K_n << M * \sum_{n=1}^{N} K_n \leqslant | \mathscr{R}^* | \tag{11}$$

which implies the integrated knowledge base has a much smaller size than in the rule base.

The empirical tests in Section 5 will demonstrate the execution time on each case firing in the CB subsystem as well as the rule execution time in the ADB subsystem. We will conclude that the ActiveCBR system not only maintains a smaller knowledge base, but also has better runtime performance than the active database system.

The knowledge base of a CBR system (with size M) is smaller than the integrated knowledge base in our system. The difference is insignificant, since the relationship between the size of the case space and the size of the feature space in most case bases is given by

$$N * \kappa << M << \kappa^N \tag{12}$$

where κ^N is the total size of the problem space with complete feature-value combinations. Nevertheless, the runtime performance of the ActiveCBR system is much better than the CBR system's, when we take into account the real-time reactive property of the ActiveCBR system.

4.2. Approaches on Knowledge Management

The two-level integrated knowledge representation not only simplifies the design and generation of cases and active rules, but also provides convenient methods on knowledge management. The ActiveCBR system furnishes four approaches to modify the knowledge base in order to reflect alterations of the problem domain. Again, we use the travel agents example (Table 6) to demonstrate the process of knowledge management.

Table 6. A case of travel agents case base.

Name	TravelCase 31	
Description	#245	
Threshold	85	
Solution	Hotel Golden Coast, Attica	
Feature	**Value**	**Weight**
JourneyCode	649	0
Price	$1000–2499	80
HolidayType	Recreation	35
NumOfPerson	1–2	70
Region	Germany	75
Transportation	By plane	45
Duration	5–7 days	85
Season	Summer	65
Accommodation	Luxury	70

Create new cases. It is relatively easy to create the initial case base. Thereafter, the main task is to extend the case base by adapting old cases to create new cases and to retain them in the case base. The Case Authoring module in the CB subsystem provides the interface to create new cases at runtime, while the case firing algorithm enables the new case between the intervals of case firing iteration.

In our travel agents example, for instance, suppose in a time period the number of travelers to Germany increases. Then we need more cases to indicate the hotel destination information in the category of feature *Region* with value *Germany*. So some travel instances in this category should be added to the case base. This is not an automatic adaptation process, but it does reflect the alteration of the knowledge domain.

Other related operations for knowledge management include case deletion, enabling and disabling. All these operations can be performed at runtime.

Adjust case threshold and weights. In addition to creating new cases, the Case Authoring module has the capability to adjust case threshold and feature-value weight. These operations reflect another type of knowledge modification.

For instance, when we increase the number of cases with *German* value for the *Region* feature to a limit, we may need to increase the firing threshold for all such cases. This is because the number of firing cases in this category is likely to increase proportionally with the number of cases in the same category. For the same reason, we may reduce the weight for feature-value pair (*Region, Germany*) of all cases in this category.

Adjust active rule conditions. Condition of a rule defines the context of a specific feature to have a specific value. It is necessary to change the condition when the context of the feature-value pair changes. The Rule Management module provides the ability to change rule condition via a trigger editor.

Consider the rule representation example in Section 2.2, where a feature-value pair (*Price, high*) is bound to a rule condition '*new.price* > 8000'. In a recession period, the hotels may decrease their rent by 20% on average. In this situation, it is reasonable to change the rule condition to '*new.price* > 6400'.

Currently, the ADB subsystem does not support rule condition modification

at runtime. Whether a rule can be modified at runtime is determined by the adaptability of the underlying active database system.

Create new rules. The Rule Management module provides the same interface to create a new rule in order to change rule conditions. However, this operation requires the structure of a feature table of higher-level case base to have corresponding modification.

One possibility is that the new rule matches a new value of an existing feature. For example, we may think the feature *Season* has too large a weight, so we can partition the feature, i.e., adding values *early summer* and *late summer* to feature *season*. Therefore, we need to add two more rules as well as new feature-value weights into the weight table to reflect the modification of knowledge structure. It is possible that the added rules conflict. In this case conflict resolution methods in active databases can be applied to resolve rule conflicts (Widom and Ceri, 1996).

Another similar operation is rule dropping. Generally, these operations of knowledge management do not require runtime execution. A background modification can handle such a situation better.

4.3. Maintenance Rules

In most 'passive' CBR systems, case base maintenance is performed manually as well. For an instance in the travel agents domain, it may be a fact that few people travel to Alaska in winter. Therefore, all cases with high feature-value weight on (*Region, Alaska*) should be temporarily disabled in the winter period. Another example is that if a case has not been fired for a long time, it should be disabled as well. In contrast, when summer arrives, much new user data with destination *Alaska* needs to fire the cases related to feature-value pair (*Region, Alaska*). Hence, the disabled cases should be enabled again. Currently, these kinds of case maintenance are usually done by human users.

The ActiveCBR system is capable of enabling and disabling high-level cases via active rules in the ADB subsystem, so as to realize automatic case maintenance. We refer to this kind of active rule as *maintenance rules* to distinguish them from *feature rules*, which are used for updating feature values. A maintenance rule has the same syntax as feature rules, introduced in Section 2.2, while its semantics is more complicated. The rule confluence can be ensured by always triggering maintenance rules before feature rules:

- The event of a maintenance rule may not be a data manipulation operation on user tables; instead, it can be an internal event invoked by another maintenance rule.
- The action of a maintenance rule can be the modification on the *status* attribute of *case* table, while the CB subsystem checks the case table periodically and reloads the case status to *CBCase* object accordingly at runtime.

For instance, a maintenance rule for case disabling can have the following semantics: *If the current date is in winter, then disable the case with keyword 'Alaska'*. It can be represented as follows in a trigger:

```
CREATE TRIGGER INSERT_TGGR
ON ACBR_TRAVEL_DATA FOR INSERT
AS
BEGIN
    /* other rules */
    ⋮
    /* maintenance rule condition */
    IF (season in ("December", "January", "February"))
    /* maintenance rule action */
    UPDATE ACBR_TRAVEL_CASE
      SET status = "disabled"
      WHERE CHARINDEX("Alaska", description) > 0
    ⋮
    /* other rules */
END
```

A more complicated rule invoked by an internal event can be described as follows: *If more than 10 cases with 80 or more feature-value weights on (Region, Alaska) are disabled, then disable all such cases.* This rule will be invoked when an UPDATE event occurs on table *ACBR_TRAVEL_CASE*. In this example, the UPDATE event is internal, since it is not caused by external user data; in contrast, it is caused by the action of other maintenance rules such as the one in the previous example.

Currently, the maintenance rules in the `ActiveCBR` system are mainly used to enable and disable cases at runtime. This is useful when the size of the case base expands. By automatically disabling the cases seldom fired, the system can maintain the knowledge base in a relatively small size, in which every case has a relatively high firing frequency, so as to ensure system performance. On the other hand, when the case firing frequency is too low, other maintenance rules can enable more cases to active status, so as to enhance system effectiveness. We note here that deciding an appropriate threshold for case activation is an interesting question itself, and deserves more future work.

Other knowledge-base maintenance features such as inconsistency and redundancy detection can also be implemented in the form of maintenance rules. The `ActiveCBR` system will have more capabilities to perform automatic knowledge base maintenance in the future.

5. Empirical Tests

In this section, we will focus our study on the efficiency and effectiveness of the proposed `ActiveCBR` system. The efficiency of the `ActiveCBR` system is supported by the good system scale-up performance and good matching between the CB subsystem and the ADB subsystem under the layered architecture. The effectiveness of the system is ensured by the experiments on system stability during modification of the knowledge base at runtime.

The test data is taken from AI-CBR's online travel agents archive.[3] We use three kinds of data as knowledge base and external data source.

[3] *http://www.ai-cbr.org/cases.html.*

- The cases used in the experiments are from the travel agents case base. There are a total of 1470 cases in this case base. The solution of each case is a hotel destination, which is determined by nine features. An example case can be found in Section 2.2.
- The rules are generated according to the strategy introduced in Section 4. Each rule is designed to update one value of a specific feature. For examples of active rules, refer also to the description in Section 2.2.
- The external user data is obtained by two means: random generation based on the value ranges of each feature in the case base; or random selection from case base directly, when we need a higher case firing frequency.

Our tests are aimed at establishing the validity of the integration of an active database subsystem and a CBR subsystem. The performance of the ActiveCBR system is determined by the runtime scale-up property of both the CB subsystem and the ADB subsystem. We examine the performance of each subsystem, and hope they can match well in terms of the execution time when integrated together.

The experiments were conducted on a Pentium 133 MHz PC with 96 MB of memory running Windows NT 4.0. To present the different system performance, related modules were repeatedly run 1000 to 5000 times, as long as sufficient accuracy could be obtained. The database for case base storage and user data input is Microsoft SQL Server 6.5, running on a Pentium Pro 200 MHz PC with 150 MB of memory. The selection of host machines reflects the capacity difference in client/server applications.

5.1. Scale-Up Properties of the CB Subsystem

The *Case Firing* module is the main procedure of the CB subsystem that performs the similarity computation and fires cases. According to the Case Firing algorithm in Section 2.3, the system executes three tasks:

1. It enables new case and retrieving runtime feature values. In the experiments, the feature values are obtained from a SQL Server data table.
2. It calculates the similarity of each feature and the score of each case.
3. It visualizes updated case firing information and real-time feature values.

Figure 4 illustrates the execution time of each of the three stages with the growth of number of cases. The execution time is shown in milliseconds.

In the figure, Task 1 involves database access, and takes a constant time as long as the size of feature table is fixed. The execution time of Tasks 2 and 3 is proportional to the number of cases in the case base, and exceeds the execution time of feature table access when the size of the case base increases to more than 120 cases. Because cases represent typical knowledge in a domain, the case base with 300 cases is already quite large, since each case represents much other data. The linear-time case-retrieval algorithm provides good scalability in terms of number of cases. Its execution time is even less than the execution time for case firing visualization.

Figure 5 demonstrates the empirical results of the overall system execution time, which shows the performance as a function of the number of feature-value pairs.

In the experiments, we first fix the number of feature-value pairs to 24, and measure the execution time of the *Rule Firing* module with various numbers of

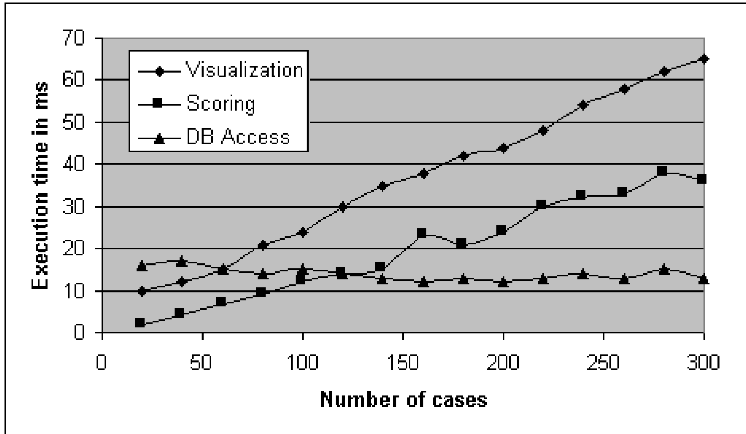

Fig. 4. The performance of the CB subsystem in steps: (a) feature values retrieval; (b) similarity and score computation; (c) visualization.

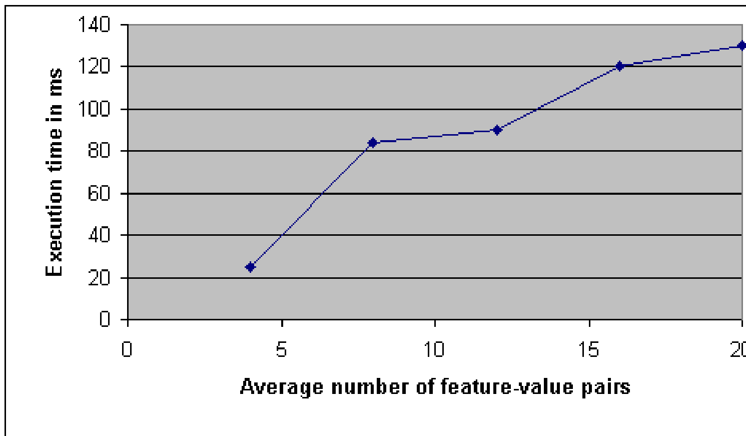

Fig. 5. The performance of the CB subsystem with growth of the number of feature-value pairs.

cases from a minimum of 20 to a maximum of 300. The execution time changes linearly from a minimum of 28 ms to a maximum 120 ms. Then we keep 150 cases in the case base and adjust the number of feature-value pairs. For each of the eight weighted features (recall the case example in Section 2.2), there are three possible values, so we get the maximum of 24 feature-value pairs. To get a smaller set of feature-value pairs, we disable one or two values of each feature on all cases or one half of the randomly selected cases. We test the system execution time on different numbers of feature-value pairs from 4 to 24 with an interval of 4. The execution time is from a minimum 27 ms to a maximum 124 ms. The scale-up property of our system with the growth of number of feature-value pairs is linear as well.

In the experiments to test the performance of the CB subsystem, the lower-layer ADB subsystem is running with 32 rules. However, since the two subsystems

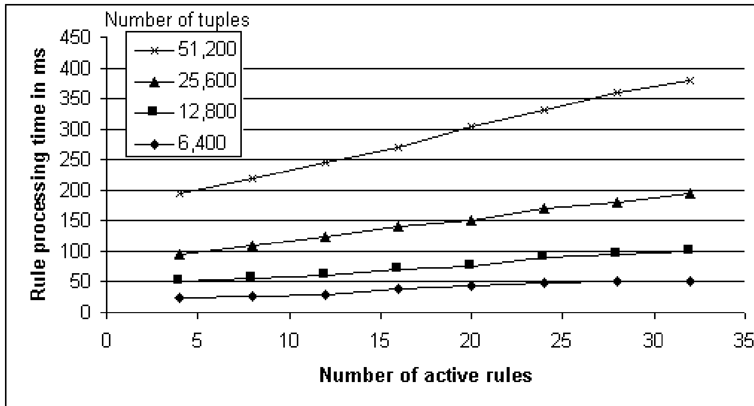

Fig. 6. The performance of the ADB subsystem with growth of number of rules and tuples.

are executed independently, the number of active rules in ADB does not affect the results of the above experiments.

5.2. Scale-Up of the ADB Subsystem

Generally, the rule execution in active database systems tends to be more difficult to understand and maintain when supporting more facilities. In the ADB subsystem, the design process of the active rule set is simplified. As we have discussed in Section 4, rule termination and rule confluence are guaranteed in our system. In addition, the rule modularization method avoids rule interactions among rule sets in different problem domains. In this section, we test the performance of the ADB subsystem on the rule set of the travel agents domain.

The performance tests verify two system design objectives: first, the system should be capable of processing a reasonable number of active rules in real time; second, the rules are operated on a large data source. We design an experiment to test the rule processing time for different sizes of rule set integrated with different numbers of tuples in the user table. Figure 6 illustrates the scale-up property with growth of number of rules and tuples in the active database.

The number of rules varies from 4 to 32, each corresponding to one feature-value pair in the higher-layer CB subsystem. Note that the rule set has to be able to reset feature values to a *null* value, 'unspecified', so that there are four rules corresponding to the three values of each feature. For each size of rule set, we test the rule processing time based on different sizes of user table, with tuples from 1600 to 12,800. From the experiments, we can conclude that the ADB subsystem has a linear scale-up property with growth of the size of the rule set.

However, the rule processing time also linearly increases when the database becomes larger. Reviewing Fig. 5 together with Fig. 6, we observe that the rule processing time for a rule set with 32 rules and a user table with 12,800 tuples is about 100 ms, which is at the same level of execution time of the *Rule Firing* module in the CB subsystem for a case base with 300 cases. The time matching is acceptable.

5.3. Summary

In this section, we introduce the initial experiments we have performed on the ActiveCBR system. System performance is verified in terms of the subsystem scalability and execution time matching between the CB subsystem and the ADB subsystem.

From the performance experiments, we observe that the rule execution time for a rule set with 32 rules and a data table with 12,800 tuples exceeds the case firing execution time for a case base with 300 cases. According to the discussion in Section 4, an active database system needs many more rules than our case-rule integrated representation to represent the same quantity of knowledge. Apparently, without the high-level case-level knowledge representation, a pure active database system will spend much more time performing the same tasks in our experiments, especially for a large database. The better scale-up property of the ActiveCBR system over active database systems is attributed to the smaller knowledge base.

From the dynamic point of view, the knowledge base is easy to modify to meet the requirements of users and the alteration of knowledge domain. In the performance experiments, we have tried to add/enable and delete/disable cases at runtime. System performance has no dramatic deviation when we modify the case base gradually. The ActiveCBR system is stable since the local modification in the knowledge base has no large impact on the global activity of the system in terms of performance.

In the future, the ActiveCBR system can be tested in different problem domains and on different underlying relational databases to further verify system performance. More experiments for flexibility validation are possible when we change other knowledge components such as feature-value weights and rule conditions at runtime.

6. Conclusions and Future Work

We have presented an ActiveCBR agent system that integrates a CBR system with an active database system. The integrated system demonstrates the properties of an agent: it is reactive, autonomous and adaptive. In addition, the layered architecture ensures that the system is flexible in knowledge management.

One interesting possibility for future work is to explore ways to make the system communicate with other agent systems in a multi-agent setting. It would be especially interesting to empower the agent for information management tasks on the Internet.

References

Aamodt A, Plaza E (1993) Foundational issues, methodological variations, and system approaches. Artificial Intelligence Communications 7(1):39–59

Arens Y, Chee CY, Hsu C, Knoblock CA (1993) Retrieving and integrating data from multiple information sources. *International Journal on Cooperative Information Systems*, 2(2):127–158, 1993.

Biliris A (1992) The performance of three database storage structures for managing large objects. In ACM SIGMOD Conference on the Management of Data, San Diego, CA

Ceri C, Fraternali P, Paraboschi S, Branca L (1996) Active rule management in Chimera. In Active database systems: triggers and rules for advanced database processing. pages 151–76. Morgan Kaufmann, Palo Alto, CA

Etzioni O, Weld D (1994) A softbot-based interface to the internet. Communications of ACM (July) Vol 37, Issue 7, pp 72–76.
Diaz O, Jaime A (1997) EXACT: an extensible approach to active object-oriented databases. VLDB Journal 6(4):282–295
Diaz O, Jaime A, Paton NW, Qaimari G (1994) Supporting dynamic displays using active rules. ACM SIGMOD Record 23(1):21–26
Gentner D (1983) Structure mapping: a theoretical framework for analogy. Cognitive Science 7:155–170
Hammer J, Garcia-Molina H, Nestorov S, Yerneni R, Breunig M, Vassalos V (1997) Template-based wrappers in the TSIMMIS system. In Proceedings of the 26th SIGMOD, Tucson, AZ, May 1997
Jennings NR, Sycara K, Wooldridge M (1998) A roadmap of agent research and development. Autonomous Agent and Multi-Agent Systems 1(1):275-306
Kolodner JL (1993) Case-based reasoning. Morgan Kaufmann, Palo Alto, CA
Leake DB, Kinley A, Wilson D (1995) Learning to improve case adaptation by introspective reasoning and CBR. In Proceedings of the first international conference on case-based reasoning. Springer, Berlin
Moukas A (1996) Amalthaea: information discovery and filtering using a multiagent evolving ecosystem. In Proceedings of the conference on practical application of intelligent agents and multi-agent technology, London
Owens K, Adams S (1994) Oracle 7 triggers: mutating tables? Database Programming and Design 7(10):31–49
Quinlan J (1986) Induction of decision trees. Machine Learning 1:81–106
Shoham Y (1997) An overview of agent-oriented programming. In Bradshaw JM (ed). Software agents. AAAI Press, pp 271–290
Sycara K, Zeng D Multi-agent integration of information gathering and decision support. In Proceedings of the 12th European conference on artificial intelligence. Wiley, Chichester, UK
Watson I, Marir F Case-based reasoning: review. Knowledge Engineering Review 9(4):355–381
Widom J, Ceri S (1996) Active database systems: triggers and rules for advanced Database Processing, Morgan Kaufmann, Palo Alto, CA
Wilkins DE, Myers KL (1998) A multiagent planning architecture. In Proceedings of the Fourth International Conference on AI Planning Systems. Pittsburgh, PA. AAAI Press. Menlo Park, CA. Pages 154–162

Author Biographies

Sheng Li graduated with a B.S. degree in mathematics from Peking University in Beijing, China, in 1992 before he began his graduate study at Simon Fraser University. After obtaining his M.Sc. degree in Computer Science at Simon Fraser University in 1999, he has been working with Seagate Software in Vancouver, BC Canada.

Qiang Yang is an associate professor of the School of Computing Science at Simon Fraser University (SFU), in British Columbia, Canada. He received his Ph.D in Computer Science at the University of Maryland, College Park, in 1989. Prior to joining Simon Fraser University, Dr Yang was an associate professor in Computer Science at the University of Waterloo. His research interests are knowledge-based systems using case-based reasoning, data mining for e-commerce and intelligent planning.

Correspondence and offprint requests to: Q. Yang, School of Computing Science, Simon Fraser University, Burnaby, BC, Canada V5A 1S6. Email: qyang@cs.sfu.ca