Proceedings

DEXA 2003

Fourteenth International Workshop on

Database and Expert Systems Applications

1-5 September 2003
Prague, Czech Republic

Proc. of 14th International Workshop on Database and Expert Systems Applications (DEXA 2003), pp. 936--941 (Sep. 2003).

Dynamic Construction Mechanism of a Trigger Graph on Active Databases in Mobile Computing Environments

Tsutomu TERADA
Cybermedia Center
Osaka University
tsutomu@cmc.osaka-u.ac.jp

Masahiko TSUKAMOTO
Grad. School of Information Science and Technology
Osaka University
tuka@ist.osaka-u.ac.jp

Shojiro NISHIO
Grad. School of Information Science and Technology
Osaka University
nishio@ist.osaka-u.ac.jp

Abstract

As a result of the rapid development of wireless communications and computer hardware technologies, we can now access various information anywhere using handy terminals. To support the integrative use of data held by mobile hosts in this environment, we proposed the AMDS (Active Mobile Database System) as the kernel system for data management. The behavior definition language of this system, ECA(Event-Condition-Action) rules, has a high description capability that enables users to define complicated behavior. However, the execution of the ECA rules may fall into a chain of unexpected behaviors. In general, a directed graph, called a trigger graph, is used for detecting chains. Since the trigger graph highly depends on network topology, it is difficult to employ the trigger graph in a mobile computing environment. In this paper, we propose a method that reconstructs the trigger graph dynamically to adapt to changes in network topology. Using this mechanism, mobile applications with ECA rules can be used more safely.

1 Introduction

Now, we can access information anywhere using handy terminals with wireless communication capabilities. We call such an environment as the *mobile computing environment*. In this environment, the following services are required:

- Company employees have a handy terminal which includes their schedule and work progress. A manager integrates the schedule data on handy terminals under his/her command, and allocates new assignments to the appropriate employee.
- When a visitor enters a museum, he/she walks around with a handy terminal that automatically displays information about the nearest showpiece.

Such services require a function that collects data from mobile hosts, and this function must be available on any mobile host. Therefore, it needs a common platform to manage the connection to/disconnection from a network and data collection from other hosts automatically. Highly motivated by such requirements, we proposed the *AMDS (Active Mobile Database System)*[6]. AMDS is an enhanced active database that can handle various events in mobile computing environments easily. In AMDS, system behaviors are described in the *ECA rule (Event-Condition-Action rule)*. While a chain-execution of ECA rules achieves complicated behaviors, the execution of ECA rules may fall into unexpected behaviors, such as an infinite loop. Therefore, it is necessary to observe the execution of the ECA rules to avoid abnormal behavior. Consequently, we propose an infinite loop detection mechanism in the mobile computing environments. Our method can detect infinite loops of the ECA rules by a little additional network traffic because it exchanges only necessary information among hosts.

The remainder of this paper is organized as follows. Section 2 outlines AMDS, and Section 3 explains the infinite loop of an active database and methods for solving this problem. Section 4 presents our method for detecting the infinite loop. Section 5 illustrates the implementation of our method, and Section 6 explains considerations of our work. Section 7 presents conclusions and our plans for future study.

2 AMDS

AMDS is an enhanced active database system. An active database is a database system that processes prescribed actions in response to the occurrence of an event generated inside/outside of the database[5]. The behaviors of an active database are described by ECA rules. Each ECA rule consists of three parts: the event, the condition, and the action. The event part describes an event occuring in the system. The condition part describes the conditions for executing the following action. The action part describes the operations to be carried out when the event occurs and the condition part is satisfied.

In conventional active databases, only database operations are considered as events. Further, active databases can carry out actions only concerning database operations. In AMDS, the description capability of the ECA rule is enhanced to fulfill various requirements in mobile computing environments.

2.1 ECA Rule

Figure 1 shows the syntax of the ECA rule in AMDS. In the figure, '*Definition of variables*' defines

```
CREATE RULE  Rule name ON  Event name
     [  Definition of variables ]
     [ WHERE  Conditions ]
     THEN DO  Actions
```

Figure 1: Syntax of ECA rules

Table 1: Events provided by AMDS

Name	Content
CONNECT	MH (Mobile Host) connection to a cell
DISCONNECT	MH disconnection from a cell
SELECT	Data retrieve
INSERT	Data insert
DELETE	Data delete
UPDATE	Data update
RECEIVE	Receiving a data packet
TIMER	Firing a timer

Table 2: Actions provided by AMDS

Name	Content
QUERY([*expression*])	Database operation
SEND([*opponent*], [*expression*])	Send a data packet
INSERT_ECA([*description*])	Store an ECA rule
DELETE_ECA([*identification*])	Delete ECA rules
ENABLE_ECA([*identification*])	Activate ECA rules
DISABLE_ECA([*identification*])	Deactivate ECA rules
SET_TIMER([*condition*])	Set a new timer
KILL_TIMER([*identification*])	Remove a timer

Table 3: Contents of NEW data and OLD data

Event	NEW	OLD
CONNECT	MH information	–
DISCONNECT	–	MH information
SELECT	Retrieved data	–
INSERT	Inserted data	–
DELETE	–	Deleted data
UPDATE	Data after update	Data before update
TIMER	Timer ID	–
RECEIVE	Received data	–

the local variables used in this rule. '*Conditions*' specifies the condition for executing the following actions. In '*Action*', the operations to be carried out are described. Tables 1 and 2 show the events and the actions provided by the AMDS. Moreover, the AMDS provides two system parameters, *NEW data* and *OLD data*, for each event. The system sets the information shown in Table 3 to these parameters when the event occurs. These parameters can be used anywhere in an ECA rule.

Figure 2 shows an example of the ECA rules for collecting schedule information from mobile hosts. *RULE1* for a mobile host server sends a request of schedule data to mobile hosts when the hosts connect to the cell of the mobile host server. *RULE2* for the mobile hosts replies with the schedule data on the local database to the mobile host server. When a mobile host connects to a mobile host server, RULE1 is activated. Then, the action of RULE1 triggers RULE2. In this way, it is possible to describe complex behaviors by chaining the execution of the ECA rules.

```
CREATE RULE RULE1 on CONNECT
  THEN DO
   SEND( new.from, "Request'" );

CREATE RULE RULE2 on RECEIVE
  WHERE new.header = 'Request''
  THEN DO data = QUERY("select s.* from Schedule s");
   SEND( new.from, "result'", data );
```

Figure 2: An example of ECA rules

3 Detection of Nontermination

In this section, we show an example of an infinite loop by a chain of rules execution. Then, we explain the detection techniques for infinite loops, the problem of these techniques, and our approach to solve these problems.

3.1 Infinite Loop of ECA Rules Execution

Since an event can be triggered by executing the action of another ECA rule, two or more rules can be executed in a chain. While this characteristic enables complex behaviors, it also brings about unexpected behaviors such as an infinite loop.

Figure 3 and Table 4 show an example of such unexpected behavior. $R1$ and $R2$ are rules stored in a mobile host server, and $R3$ is a rule stored in a mobile host. When the mobile host connects to the cell of the mobile host server, $R1$ is triggered and sends a query to the mobile host. Since the query triggers $R3$ on the mobile host, $R3$ sends a query to the mobile host server. This query triggers $R2$ and $R2$ triggers $R3$ again. As a result, an infinite loop is generated between $R2$ and $R3$. It is difficult to detect an infinite loop by checking the ECA rules individually. Moreover, infinite loops can be created by various factors such as the migration of hosts and a change of the rule set in a host. Therefore, the system must provide a mechanism to investigate the possibility of an infinite loop beforehand, and to detect an infinite loop when it actually occurs.

3.2 Detection Methods

Generally, the following two types of methods are used for detecting an infinite loop:

- **Run-time detection:** The system detects an infinite loop while the system is running.
- **Static detection:** The system detects an infinite loop logically using a digraph called a *trigger graph*.

In run-time detection, the system detects an infinite loop using a chain counter and timestamps of rule execution. Therefore, the system cannot use this method

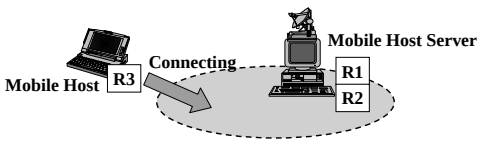

Figure 3: An example of an infinite loop

Table 4: Contents of R1-R3

Rule	E	C	A
R1	MH Connection	–	ID Request
R2	Receive	ID Request?	ID Reply
R3	Receive	–	ID Request

when we do not want to stop the system or when we want to check the safety beforehand.

In a trigger graph used for static detection, the node of the graph represents one rule. Two rules, $r1$ and $r2$, are connected by a directed edge from $r1$ to $r2$ if the action of $r1$ triggers the event of $r2$. The absence of cycles in the graph guarantees the termination of the set of rules[1]. The trigger graph is used widely as a dependable technique for detecting an infinite loop.

However, in mobile computing environments, the network topology changes dynamically with the migration of mobile hosts. Moreover, since mobile hosts exchange ECA rules frequently in the AMDS, the system must consider the correlation of the ECA rules between multiple hosts. Thus, the system must know all the ECA rules on every host in order to construct the trigger graph in mobile computing environments. In addition, since changes of the network topology cause changes of the trigger graph topology, the system must construct trigger graphs in all cases that each mobile host connects to every mobile host server. In the real world, it is difficult to predict what rules the connecting mobile host has. Additionally, it is not realistic to examine all topologies of the trigger graph from the viewpoint of computational complexity.

4 Dynamic Detection

In this section, we describe the *dynamic method* we propose. In the dynamic method, the system exchanges a part of the trigger-graph information in response to a change in the network topology. This method detects an infinite loop before the loop actually occurs. Moreover, this method can react to the connection of a new mobile host with a little additional network traffic, because hosts exchange only the necessary information concerning the change in the network topology.

Hereinafter, we explain the construction algorithm of the trigger graph in our method. Next, we describe how the system detects an infinite loop, and show an example of this procedure.

4.1 Construction of Trigger Graph

A trigger graph is constructed by the following steps:

1. Creation of the trigger graph and detection of infinite loops on the local host
2. *RS-path* extraction
3. *RS-path* degeneracy
4. Transmission of *RS-path*
5. Processing *RS-path* on other hosts

First, each host creates the trigger graph of the ECA rules and detects infinite loops on the local host. If no infinite loop is detected in the first step, the host sends to other hosts a part of the trigger graph that consists of the rules with the possibility of causing chain-execution between hosts. We call this sub-graph the *RS*(*Receive-Send*)*-path*. When another host receives the RS-path, the above procedure is performed again including the received RS-path. This trigger-graph creation algorithm is performed in the case of the initiation of the system, the change of network topology, and the chage of rule set.

Each step is explained in detail in the following section.

4.1.1 Creation of Local Trigger Graph

The trigger graph of the local host is constructed by the following steps:

1. **Extracting a set of triggering paths**
   For all rules in the local host, the system extracts triggering paths from one rule to another rule. Then, the system extracts chaining paths, which are the linked paths triggered in a chain. If there is no loop in the triggering-path set, the system ends the algorithm.
2. **Evaluating the condition of the loop path**
   If there are loops in the rule set, the system evaluates the condition of the rules in each loop for judging whether they become an infinite loop in reality.

2-1. **Rewriting values of NEW/OLD data**
The system rewrites NEW data and OLD data in each condition of the rules by converting 'NEW' or 'OLD' to an actual table name or value. This operation gives the system some information for making a decision. For example, the system can know whether multiple conditions access the same table.

2-2. **Merge of conditions**
The system links up the conditions of the rules in a loop path with AND operator. When linking up a condition, the system checks the action of the rule. If the action accesses a table, conditions concerning the table are removed from the linked condition. This is because these conditions may be invalidated by the action.

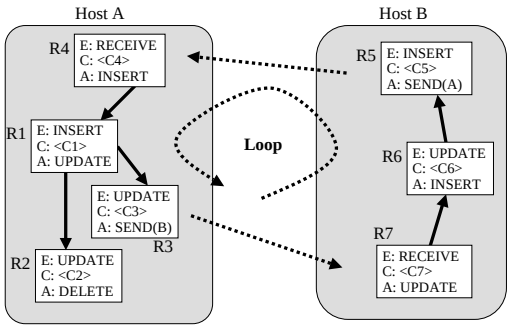

Figure 4: An example of chains between different hosts

3-3. **Evaluation of the merged condition**
If there is no contradiction on the merged condition, the system regards the loop as an infinite loop. Otherwise, the loop is disregarded.

4.1.2 RS-path Extraction

As illustrated in Figure 4, the chain execution of rules between multiple hosts is caused only by pairs of the SEND action on a host and the RECEIVE event in another host. Additionally, the chaining path from the rule that has the RECEIVE event to the rule that has the SEND action in the same host is required for composing an infinite loop between multiple hosts. We name the chaining path the *RS*(*Receive-Send*)*-path*. Since a host with RS-paths may be a part of infinite loops between multiple hosts, the host sends RS-paths to other concerned hosts for the detection of such infinite loops.

The RS-path extracting procedure is as follows. Firstly, the system extracts a path that starts the RECEIVE event. Next, from among the chaining-paths started from the path, the system selects paths ended by the SEND action as the RS-paths.

4.1.3 RS-path Degeneracy

Since the RS-path may be still retransmitted to other hosts, the network traffic may increase seriously when transmitting the RS-path without any compression. Thus, unnecessary information is reduced from the RS-path according to the following procedures:

1. **Condition degeneracy in the RS-path**
Around the rules in the RS-path, the system uses only the part of the SEND action of the last rule and the RECEIVE event of the first rule. Thus, the RS-path can be reduced into one rule. The reduced rule has the event of the first rule and the action of the last rule in the RS-path. The condition of the reduced rule is the merged condition made with a similar procedure to the one described in Section 4.1.2. Moreover, the conditions concerning the local database are removed because such information has no relation to other hosts.

2. **Merge of the multiple RS-path**
The system selects a merged-RS-path that has the same opponent address of the SEND action. Then, it integrates these RS-paths by linking the conditions with the OR operator.

4.1.4 Transmission of the RS-path

The generated RS-paths are transmitted to either of the following destination hosts:

- **The SEND action of the RS-path specifies the destination address**

  When the destination address is fixed, the system sends the RS-path only to destination hosts because there is no possibility that the chain execution with the RS-path is propagated except at these destinations.

- **The SEND action has no destination**

  If the SEND action does not specify the destination address, this action sends the RS-paths to all hosts within the cell of the sender.

4.2 Recovery from Infinite Loop

If our method does not detect any infinite loops, the system guarantees the safety of the rule set. On the other hand, even if our method detects an infinite loop, the system does not always guarantee the abnormal behaviors. Therefore, if our method detects infinite loops, the system should provide flexible service methods. sFrom the above observation, our method provides the following three methods to deal with the detection of an infinite loop:

- **Notification to users**

  This method prompts users to select the stop/continuance of the application when the system detects infinite loops. If the user selects continuance, the system regards the triggering paths that consist of infinite loops as the dangerous paths. When the ECA rules in the dangerous paths are triggered, the system continues logging triggered events and rules.

- **Removing the cause of the infinite loop**

  This method avoids the infinite loop by separating the rule or the host that causes the infinite loop. For example, if the connection of a mobile host causes an infinite loop, the mobile host server refuses the connection of the mobile host.

- **Generation of the ERROR event**

  When the infinite loop is detected, the system generates the ERROR event that has the NEW data concerning the status of the infinite loop. Flexible error handling is achieved by describing ECA rules that handle the ERROR event.

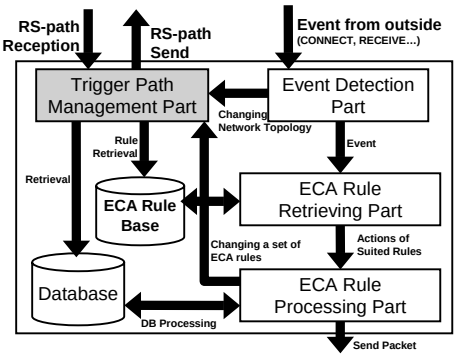

Figure 5: The system structure of AMDS

5 Implementation

We implemented the proposed method on the AMDS. The system structure of the AMDS including the mechanism is illustrated in Figure 5. We implemented the $TriggerPathManagementPart$ in addition to the modules of the previous AMDS. The $TriggerPathManagementPart$ creates trigger graphs and RS-paths based on the contents of the local $Database$ and $ECARuleBase$.

6 Consideration

6.1 Verification of Detectivity

Here, we define that the system is $Safe$ if there is no infinite loop of the ECA rules. Then, the following theorem is self-evident.

Theorem 1. For any set of ECA rules, if there is no loop that can be reached from a given event, the set of rules is $Safe$.

Next, we show the following four lemmas.

Lemma 1. A loop across multiple hosts is a combination of two or more RS-paths.

Proof. Referring to the syntax of the ECA rule, only the SEND action can trigger an event on other hosts. Similarly, only the RECEIVE event can be triggered by other hosts. If the triggering-paths make a loop, the sub-path of each host starts by the rule with the RECEIVE event and ends by the rule with the SEND action. Moreover, a single RS-path cannot be a loop. □

Lemma 2. If our algorithm starts based on either of the rules in a certain RS-path, the loop including the RS-path is detected.

Proof. The end of the RS-path is the SEND action. Since the RS-path information is transmitted to the opponent of the SEND action, the information is transmitted as long as the triggering-path exists between hosts. For the above reasons and lemma 1, lemma 2 is proved. □

Lemma 3. Every RS-path is evaluated by our algorithm when the RS-path can be organized.

Proof. A set of triggering-path changes when a set of rules changes in the local host or the network topology changes by the migration of mobile hosts. The detection algorithm starts in each case. □

Lemma 4. The degeneracy of the RS-path does not cause mistakes in loop detection.

Proof. Because the degeneracy of the RS-path does not change the connection status of the rules in the RS-path, it does affect the loop detection. □

Then, the following theorem is proved.

Theorem 2. If the proposed algorithm does not detect the loop, the set of rules is $Safe$.

Proof. From Theorem 1 and the four lemmas, infinite loops within the local host are detected by the trigger graph, and infinite loops across multiple hosts are detected by the RS-path. □

From Theorem 2, the system guarantees the safety of rules when our algorithm does not detect the infinite loop. However, we must remember, even if our method detects an infinite loop, the system does not always guarantee against abnormal behaviors of the rule set.

6.2 Evaluation

In this section, we evaluate the additional network traffic caused by using our method. The evaluation uses an application for an amusement park on the AMDS simulator. This application automatically transmits information about the waiting time of each facility and suggests routes when the users approach each facility. Moreover, users can generate a query to get the information voluntarily. The number of ECA rules for this application is eight in a mobile host and one dozen in a mobile host server. There are six mobile host servers as facilities. We evaluate the amount of network traffic by changing the number of mobile hosts that existed at the same time. We evaluate our method by comparing it with two other methods, the $Conventional$ method and the $Non\text{-}Merge$ method. The former method transmits the information of the trigger-graph to all hosts. The latter transmits the RS-path the same way as the proposed method. However, the latter method does not use the degeneracy of the RS-path.

Mobile hosts are allocated randomly on the 500*500 square field, and each host can move to the neighbor area with each step. Each mobile host aims to reach a certain mobile host server. Then, if the host reaches the opponent server, the host aims to reach another server after some rest. Under such conditions, we performed the simulation in 100,000 steps.

Figure 6 shows the total amount of traffic by changing the number of mobile hosts. The total amount of traffic is the sum of the traffic that the application actually sends/receives and the path information.

The figure shows that the traffic of the Conventional method increases exponentially. This is because the increase of hosts increases both the frequency of the triggering-graph updating and the number of opponent hosts receiving the triggering-path information. In our method, since the system sends information only to hosts that are opponent hosts of the SEND action, the traffic increases linearly.

Figure 7 shows the ratio of the triggering-path traffic in the total amount of traffic. In the Conventional method, the trigger information makes up the majority of the traffic even where the number of hosts is considerably few. The Non-Merge method greatly im-

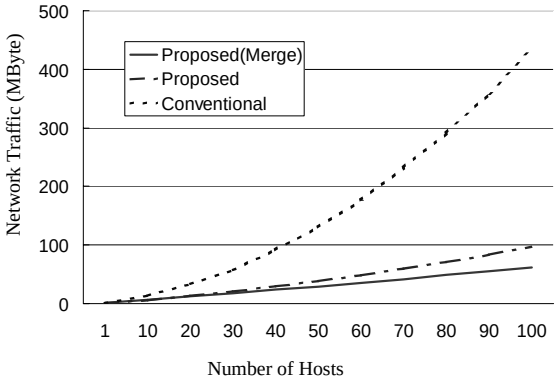

Figure 6: Network traffic in response to the number of mobile hosts

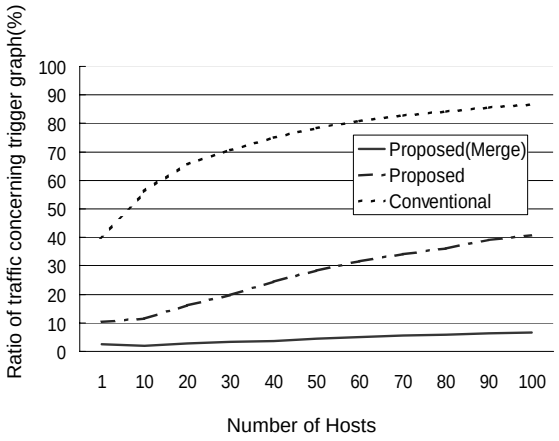

Figure 7: Ratio of path traffic in total traffic

proves the efficiency of the traffic compared with the Conventional method. However, this method may not be efficient where there are a large number of hosts, because about 40 percent of the traffic is consumed by trigger information with a hundred hosts. In the proposed method, the trigger information traffic is always under 10 percent of the total traffic

6.3 Related Work

Much research has been conducted seeking to improve trigger graph detectability[2][3][4]. In the early research of the trigger graph, the system detected infinite loops using only the event and the action. Therefore, these studies improved the detectability by considering the condition part of the rule. The approach of Baralis and Widom[2] constructs the trigger graph considering the condition with an algebraic approach. The approaches of Karadimce[3] remove the edges of the trigger graph that are not executed in a chain by considering the condition of each edge. [4] determines whether the detected loop is the infinite loop or the finite loop within two or more cycles by unrolling the detected loop.

These studies assume that the system knows all the contents of the ECA rules beforehand. Therefore, they can coexist with our method, which aims to detect the infinite loop between multiple hosts efficiently. Applying these techniques to the construction of a trigger graph on the local host in our method may decrease the mis-detection of infinite loops.

7 Conclusion

In this paper, we proposed a dynamic method for detecting infinite loops. Using our method, we can operate the AMDS more safely in mobile computing environments. Moreover, we showed that our method can reduce network traffic, compared with the conventional method, by simulation studies. Further study is desired to evaluate the efficiency and the network traffic in real machines. We are also planning to propose the mechanism for detecting infinite loops by integrating the dynamic method, the run-time method, and the static method.

Acknowledgments

This research was supported in part by "The 21st Century Center of Excellence Program", Grant-in-Aids for Scientific Research number 13780331 from the Japan Society for the Promotion of Science and Special Coordination Funds for promoting Science and Technology of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[1] Aiken, A., Widom, J. and Hellerstein, J. M.: Behavior of Database Production Rules: Termination Confluence, and Observable Determinism, *ACM SIGMOD International Conf on the Management of Data*, pp. 59–68 (1992).

[2] Baralis, E. and Widom, J.: An Algebraic Approach to Rule Analysis in Expert Database Systems, *20th VLDB Conf*, pp. 475–486 (1994).

[3] Karadimce, A. P. and Urban, S. D.: Refined Trigger Graphs: A Logic-Based Approach to Termination Analysis in an Active Object-Oriented Database, *ICDE'96*, pp. 384–391 (1996).

[4] Lee, S. Y. and Ling, T. W.: Unrolling Cycle to Decide Trigger Termination, *25th VLDB Conf*, pp. 483–493 (1999).

[5] Lohman, G., et al. "Extentions to Starburst: Object, Types, Functions, and Rules," *Communications of the ACM*, Vol. 34, No. 10, pp. 94–109, 1991.

[6] Murase, T., Tsukamoto, M. and Nishio, S.: Active Mobile Database Systems for Mobile Computing Environments, *IEICE Trans. Info. and Syst.*, Vol. E81-D, No. 5, pp. 427–433 (1998).