Vol. 47 No. SIG 19(TOD 32) IPSJ Transactions on Databases Dec. 2006

*Regular Paper*

# Model Checking Active Database Rules under Various Rule Processing Strategies

Eun-Hye Choi,[†] Tatsuhiro Tsuchiya[††] and Tohru Kikuno[††]

An active database is a database system that can react to internal, as well as external, database events. The reactive behavior of an active database is determined by a predefined set of active database rules, along with a rule processing strategy. A common problem associated with active database systems is the possible non-termination of the active database rules. Previous work on the analysis of the conditions required for the termination of active database rules has only considered limited rule processing strategies. This paper proposes an approach for automatically detecting the non-termination of active database rules using a model checking technique. With this approach, a general framework for modeling active database systems is first proposed. This framework is useful for analyzing the behavior of rules with different rule processing strategies and for allowing the adoption of different contexts and different execution coupling modes for the active database rules. Based on the proposed modeling framework, the termination property of active database rules with various rule processing strategies is next checked using SPIN, a model checking tool. Through experimental results, we demonstrated the feasibility of using this method.

## 1. Introduction

An *active database*[17)] is a database system that has the functionality to react to internal, as well as external, database events. On the other hand, a traditional database is limited to responding to external events that can include user queries or outside applications. Since 1980, the extra functionality of an active database, which allows the integration of reactive behavior in a centralized and timely manner, has attracted the attention of researchers and has led to the development of a number of systems, including Starburst[21)], SQL-3[13)], HiPAC[7)], and Chimera[4)].

The reactive functionality of active database systems can be described by *rules*, which have three components: an *event*, a *condition*, and an *action*. The *event* defines the trigger for the given rule; the *condition* defines when the given rule is activated; and the *action* defines the action that must be taken based on the activated rule. The behavior of an active database rule system depends not only on a predefined set of rules, but also on the rule processing strategy that is adopted by the system. It is usually difficult to predict the behavior based on the interactions of the active database rules associated with a given rule processing strategy.

One of the most common problems in an active database system is the non-termination of active database rules, since a given rule may indefinitely trigger another rule. In general, detecting the termination of active database rules is known as the *undecidable problem*. Most of the previous research[1),2),6),12),14),19),20)] that analyzed active database rules has focused on static analysis and has provided the principal conditions for the termination of rules, including acyclicity in the trigger-graph of the rules[1)]. However, static analysis cannot be used to predict or specify the undesirable behavior of active database rules. In addition, previous work on the termination analysis of active database rules has only considered limited rule processing strategies.

To remedy this gap, our paper proposes an approach for automatically checking the termination of active database rules with different rule processing strategies using a *model checking* technique[5),10),15)]. Model checking is a verification technique that can exhaustively check whether or not a finite state transition system satisfies the temporal logic property. It is automatically performed using the currently available model checkers.

Specifically, we propose a modeling framework for constructing an abstract finite model of an active database system with different rule processing strategies. This model is parametric in several contexts and several coupling modes used in the existing active database sys-

† National Institute of Advanced Industrial Science and Technology
†† Osaka University

tems[4),7),13),21)]. The model is also amenable to verification by the model checker, SPIN[10)]. We determine how to automatically check the termination property of the rules in the proposed model using model checking.

Thus, the proposed method is the first termination checking method of active database rules that is not limited to specific contexts and execution coupling modes of the active database system. Moreover, the proposed termination checking of the active database rules is automatically performed using model checking and, if the termination property does not hold, one can specify the undesirable behavior of the active database rules by analyzing a counterexample output from the model checker. We believe that the proposed approach can be applied and extended for analyzing rule behaviors in various active database systems with different features of rule processing.

This paper is organized as follows. Section 2 introduces the rules, the rule processing strategies of active database systems, and model checking with SPIN. Section 3 explains the proposed modeling framework, which can deal with various contexts and execution coupling modes. Section 4 describes how, based on the proposed modeling framework, model checking can be used to determine the termination property of rules. Section 5 shows the experimental results obtained when the proposed method is applied to a sample rule set with different rule processing features. Section 6 briefly discusses the related work. Section 7 summarizes the main conclusions that can be drawn from the study.

## 2. Preliminaries

### 2.1 Active Database Rules

An active database system consists of a set of rules that describe the desired reactive behaviors. An active database rule consists of three components: an *event*, a *condition*, and an *action*, with the following syntax:

**ON**[*event*] **IF**[*condition*] **DO**[*action*].

**Example 1** **Figure 1** shows two sample data tables, 'Emp', which contains the records employee's 'id' and employee's 'rank', and 'Bonus', which contains the records employee's 'id' and employee's 'amount' of bonus. Now, consider the following two rules, $r_1$ and $r_2$, which are given in **Fig. 2**. The rules add the following reactive behaviors to an active database system[☆]: Rule $r_1$ signifies that whenever Emp(rank), the rank of an employee, is updated, Bonus(amount), the amount of the employee's bonus, is increased by 10, if Emp(rank) is even. Rule $r_2$ signifies that whenever Bonus(amount), the amount of the employee's bonus is updated, Emp(rank), the rank of the employee is increased by 1. □

Table Emp

| | |
|---|---|
| id | employee's id |
| rank | employee's rank |

Table Bonus

| | |
|---|---|
| empid | employee's id |
| amount | employee's bonus |

**Fig. 1** Sample data tables.

| | |
|---|---|
| $r_1$ | **ON** update(Emp(rank)) <br> **IF** Emp(rank) mod 2 = 0 <br> **DO** update(Bonus(amount)) <br> Bonus(amount) = Bonus(amount) + 10 |
| $r_2$ | **ON** update(Bonus(amount)) <br> **IF** TRUE <br> **DO** update(Emp(rank)) <br> Emp(rank) = Emp(rank) + 1 |

**Fig. 2** Sample active database rules.

Active database rules are characterized by two models: a knowledge model and an execution model[8),17)]. The knowledge model describes the structural characteristics of the rules, while the execution model captures the runtime characteristics of the rule processing.

The knowledge model is characterized by the event type and action type, as well as the context of the conditions and the actions. Event types include data modification (e.g., insert and update) and clock (e.g., at 13:00 every Monday). Two or more events can be combined, and this is also considered to be a single event. Action types include data retrieval and modification, transaction operations, such as commit or abort, and an external call.

The execution model is characterized by the conflict resolution policy, the scheduling policy, and the coupling mode of the rules. When mul-

☆ The definition of rules, $R_1$ and $R_2$, given in Fig. 2, is implicit in that the definitions of the rules are given independently of which entries in the table's tuples are used for the events, the conditions, and the actions. Such an abstraction is used in the definition of active database rules, since the explicit semantics of the definitions are derived based on the rule description languages.

tiple rules are triggered and activated at the same time, the conflict of rules is resolved based on the numerical or relative priorities of the rules according to the conflict resolution policy. Multiple rules are executed sequentially or in parallel based on the scheduling policy.

In this paper, for simplicity, we will assume that the event and action types are only data modifications and that the rules and the transactions are executed sequentially. These assumptions have been made in previous research papers [1),9),18)].

## 2.2 Rule Processing of Active Database Systems

**Figure 3** shows the general abstract architecture of an active database system, which is reproduced from Ref. 17). An active database system consists of several principal components (rectangles in the figure) and data stores (circles in the figure), as illustrated in Fig. 3. Once a set of active database rules has been defined in a *rule base*, an active database system performs rule processing as follows:

- The *Event Detector* detects the events of interest to the system. Data modification events are noticed from the *database*.
- The *Condition Monitor* evaluates the conditions of rules associated with the detected events and stores the rules whose conditions are evaluated to true in a *conflict set*.
- The *Scheduler* chooses and runs a rule from the conflict set according to the conflict resolution policy.
- The *Query Evaluator* executes the database queries, such as transaction queries and conditions and actions resulting from the rules. When evaluating such queries, the query evaluator accesses not only the current state of the database but also, if necessary, past states, *history*, of the database.

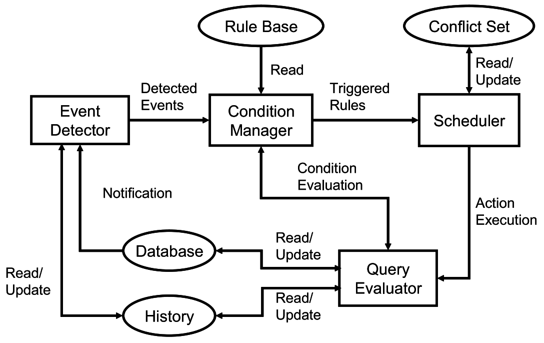

**Fig. 3** General active database architecture.

The behavior of the active database rules depends on the rule processing strategy, that is, the knowledge and execution model of the rule processing adopted by the system. Among the characteristics of the knowledge and execution model, the *contexts* and the *coupling modes* of rules are important factors in determining the termination of rules. Therefore, in this paper, for the knowledge model, we focused on the rule contexts, while for the execution model, we focused on the coupling modes of rules.

**Context** indicates, which state of a database is used in the rule processing. Let $D_T$, $D_E$, $D_C$, and $D_A$ denote the states when the current transaction starts, the event occurs, the condition is evaluated, and the action is executed. In this paper, we consider the choices in **Table 1** as the contexts for the condition, *condition-context*, and for the action, *action-context*. These contexts are, in fact, used in existing active database systems [4),7),13),21)].

**Execution coupling mode** consists of the *event-condition mode* and the *condition-action mode*. The *event-condition mode* determines when the condition is evaluated after the corresponding event has occurred. The *condition-action mode* determines when the action is executed after the corresponding condition was evaluated. The following coupling modes are supported in current active database systems [4),7),13),21)]:

- *Immediate*, where the condition (action) is evaluated (executed) immediately after the event (condition) of the rule.
- *Deferred*, where the condition (action) is evaluated (executed) anywhere within the same transaction as the event (condition) of the rule.
- *Decoupled* (or *detached*), where the condition (action) is evaluated (executed) as a different transaction.

**Table 1** Contexts.

| Choices | Condition-Context | Action-Context |
|---|---|---|
| $C_1$ | $D_C$ | $D_A$ |
| $C_2$ | $D_T$ | $D_T$ |
| $C_3$ | $D_E$ | $D_E$ |

**Table 2** Execution coupling modes.

| Modes | Event-Condition | Condition-Action |
|---|---|---|
| $M_1$ | Immediate | Immediate |
| $M_2$ | Immediate | Deferred |
| $M_3$ | Deferred | Immediate |
| $M_4$ | Deferred | Deferred |
| $M_5$ | Decoupled | Decoupled |

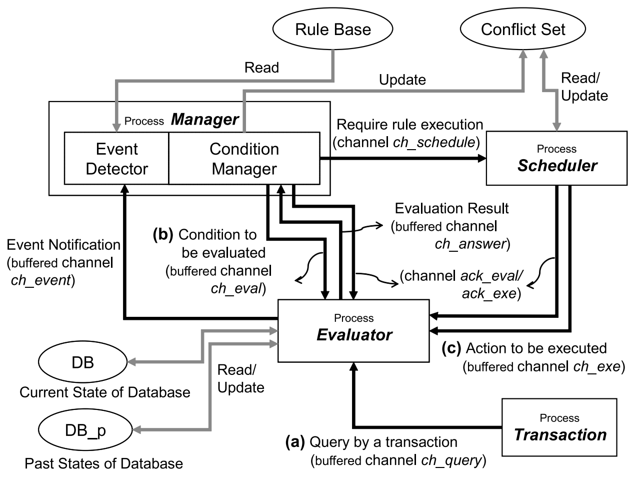
Rule Base
Conflict Set
Read
Update
Read/ Update
Process Manager
Event Detector
Condition Manager
Require rule execution (channel ch_schedule)
Process Scheduler
Evaluation Result (buffered channel ch_answer)
Event Notification (buffered channel ch_event)
(b) Condition to be evaluated (buffered channel ch_eval)
(channel ack_eval/ ack_exe)
Process Evaluator
DB
Current State of Database
Read/ Update
(c) Action to be executed (buffered channel ch_exe)
DB_p
Past States of Database
(a) Query by a transaction (buffered channel ch_query)
Process Transaction

**Fig. 4** Abstract model of an active database system.

In this paper, we will consider the five choices given in **Table 2** for the event-condition mode and the condition-action mode.

## 2.3 Model Checking and SPIN

Model checking is a verification technique that exhaustively checks whether or not a system modeled using a finite state transition system satisfies the property expressed by the temporal logic formula. Since, given a transition system and a temporal logic formula, model checking can be automatically and rapidly performed using existing tools, it has recently regained the attention of researchers. Model checking is also helpful in locating system errors, since whenever a system model does not satisfy a property formula, a counterexample that gives the system behavior that violates the property is given as the output.

SPIN[10] is known as one of the most powerful model checkers. An input model to SPIN is described in *Promela* (Process Meta-Language) which has C-like syntax. A Promela model consists of one or more asynchronous processes with data objects, non-deterministic constructs, and communication primitives. Processes can communicate via synchronous and asynchronous message passing with buffered channels or shared memory. SPIN verifies the claims specified by the Linear Temporal Logic (LTL) formulae or process invariants, which can express the basic safety and liveness properties. SPIN performs on-the-fly verification and supports several useful state search and compression strategies. With our approach, we used SPIN because of its powerful verification capability. Furthermore, SPIN was used since an active database system can easily be modeled as an asynchronous process system using Promela.

## 3. Modeling of Active Database Systems under Different Rule Processing Strategies

### 3.1 Basic Modeling Framework

**Figure 4** illustrates our abstract model of an active database system. We constructed the model based on the general active database architecture in Fig. 3, by adding the model of transactions. This transaction model is necessary for checking the behavior of active database rules. As will be explained below, the proposed model is thus slightly different from the architecture in Fig. 3. The difference lies only in the facility of modeling and abstraction and thus does not affect the generality of the proposed modeling framework.

The state of the proposed model is determined from the states of four types of data storage illustrated using circles in the figure, *DB*, *DB_p*, *RuleBase*, and *ConflictSet*, and buffered channels ch_query, ch_event, ch_cond, ch_action where

- *DB* denotes the current database state;
- *DB_p* denotes the list of the database past states;
- *RuleBase* denotes the defined set of rules,
- *ConflictSet* denotes the current conflict set of rules,
- ch_event denotes the channel buffering detected events,
- ch_cond denotes the channel buffering conditions to be evaluated, and
- ch_action denotes the channel buffering ac-

tions to be executed.

The proposed model consists of the following four processes (illustrated using rectangles in the figure): *Manager*, *Scheduler*, *Evaluator*, and *Transaction*. The Process *Manager* models the behavior of the event detector and the condition monitor. The Process *Scheduler* models the behavior of the scheduler, while the process *Evaluator* models the behavior of the query evaluator of the active database system. The Process *Transaction* models the users or the applications that send transactions to the database system. The processes read and update the data storage and buffered channels, as well as communicate with each other via synchronous or asynchronous message passing through the channels. The behavior of each process is as follows:

**Process Transaction**

This process generates and sends the transaction queries (data operations) to the buffered channel, ch_query. The Process *Evaluator* will receive the transaction queries via asynchronous message passing through the channel. In the model, we allow the transaction to consist of multiple data modification operations. However, we fixed the length of the transactions so that the state model is finite.

**Process Manager**

This process cycles through the following procedure:

(1) Detect an event occurrence via asynchronous message passing using the buffered channel ch_event.
(2) Trigger rules associated with the detected event by referring to *RuleBase*.
(3) Request the evaluation of conditions for the triggered rules to the Process *Evaluator*. The identifier for the rule to be evaluated is buffered to the channel ch_cond.
(4) The result of the evaluation is received from the Process *Evaluator* via asynchronous message passing through the channel, ch_answer. After receiving the results of the evaluation, the rules, whose conditions evaluated to true, are sent to *ConflictSet*.
(5) Request the execution of rules in *ConflictSet* to the Process *Scheduler* via synchronous message passing using the channel, ch_schedule.

**Process Scheduler**

This process non-deterministically chooses a rule among the rules having the highest priority from the *ConflictSet* and sends a request to the Process *Evaluator* to execute the action of the rule. The identifier of the rule to be activated is sent via asynchronous message passing using the buffered channel, ch_action.

**Process Evaluator**

This process evaluates the following three kinds of data operations: transaction queries requested by the Process *Transaction*, rule conditions requested by the Process *Manager*, and rule actions requested by the Process *Scheduler*.

(a) Evaluation of the transaction queries: After receiving a transaction query from the channel, ch_query, the query is evaluated and a new event corresponding to the query is sent to the channel, ch_event. This evaluation procedure is finished when an acknowledgment message is received from the synchronous channel called ack_eval.
(b) Evaluation of the rule conditions: After receiving the identifiers of the rules from the channel, ch_cond, the rule conditions are evaluated, and the results of the evaluation are sent to the channel ch_answer. The evaluation procedure is finished when an acknowledgment message is received from the channel called ack_exe.
(c) Execution of the rule actions: After receiving the rule identifier from channel ch_action, the rule action is executed, and a new event corresponding to the action is sent to channel ch_event. This evaluation procedure is finished when an acknowledgment message is received from the channel ack_eval.

To evaluate these three kinds of operations, this process accesses the data from either *DB* or *DB_p*. In the case of (a) and (c), a new event may occur after the evaluation. In these two cases, the acknowledgment is received from channel ack_eval after the Process *Manager* notices the new event and sends the additional rules triggered by the event to *ConflictSet*. In the case of (b), the acknowledgment is received from channel ack_exe after the Process *Scheduler* has sent the activated rule to the channel ch_action. Waiting for the acknowledgment prevents the evaluation procedures of (a), (b), and (c) from being intermingled with the other procedures.

## 3.2 Modeling Different Contexts and Execution Coupling Modes

The basic framework for modeling an active database system was proposed in Section 3.1, and the modeling framework is applicable to active database systems with different rule processing strategies. In this section, we explain how to model active database systems with the contexts given in Table 1 and the coupling modes given in Table 2, based on the modeling framework.

**Modeling of Contexts**

Let us examine the condition-context and the action-context. We consider the three choices, $C_1$, $C_2$, and $C_3$, shown in Table 1. In our modeling framework, as shown in Fig. 4, the Process *Evaluator* reads the current and/or the past database states when evaluating data operations. Active database systems with $C_1$, $C_2$, or $C_3$ are then straightforwardly modeled by selecting the state of the database to be read by the Process *Evaluator*.

In the case of $C_1$, $D_C$, the state when the condition is evaluated, is the condition-context state, and $D_A$, the state when the action is executed, is the action-context state. Hence, to model $C_1$, we let the Process *Evaluator* access *DB*, that is, the current state of the database when evaluating conditions and executing actions.

In the case of $C_2$, $D_T$, the state when the transaction starts, is used for the condition-context and the action-context. Hence, to model $C_2$, we store the state when the current transaction starts in *DB_p* and let the Process *Evaluator* access *DB_p* when evaluating the conditions and executing the actions.

In the case of $C_3$, $D_E$, the state when the corresponding event occurs, is used for the condition-context and the action-context. To model $C_3$, we store the state when each event occurs in *DB_p*. When evaluating a condition or a rule action, we let the Process *Evaluator* access the state of the *DB_p* corresponding to the event of the rule.

**Modeling of Coupling Modes**

Let us examine the coupling modes that consist of the event-condition mode and the condition-action mode. We consider the five choices, $M_1$, $M_2$, $M_3$, $M_4$, and $M_5$, given in Table 2. In our model framework, the Process *Evaluator* performs three types of evaluations: (a) the transaction query evaluation, (b) the condition evaluation, and (c) the action execution. Operations for (a), (b), and (c) are respectively received from the buffered channels ch_query (from the Process *Transaction*); ch_cond (from the Process *Manager*); and ch_action (from the Process *Scheduler*). As shown in Table 3, an active database system with $M_1$, $M_2$, $M_3$, $M_4$, or $M_5$ is then modeled by assigning priorities to the three channels to determine the order of receiving operations in the Process *Evaluator*.

In the case of $M_1$, the coupling modes for both the event-condition and the condition-action are *Immediate*, and thus the condition (the action) has to be evaluated (executed) immediately after event occurrence (the condition evaluation). Mode $M_1$ is modeled by giving a higher priority to channels ch_cond and ch_action than to channel ch_query.

In the case of $M_2$, the event-condition mode is *Immediate* and the condition-action mode is *Deferred*. Thus the condition has to be evaluated immediately after event occurrence, while the action has to be executed within the same transaction. Thus, mode $M_2$ is modeled as follows. Since the event-condition mode is *Immediate*, the higher priority is given to channel ch_cond than to channels ch_action and ch_query. T_end in the table denotes a Boolean variable that is true during the period after the current transaction is completed and before the next transaction starts. Since the condition-action mode is *Deferred*, the higher priority is given to channel ch_action than to channel ch_query when T_end is true. This guarantees that the action is executed within the same transaction.

In the case of $M_3$, the event-condition mode is *Deferred* and the condition-action mode is *Immediate*. Thus, mode $M_3$, like mode $M_2$, is modeled as follows. The higher priority is given to channel ch_action than to channels ch_cond and ch_query. The higher priority is given to channel ch_cond than channel ch_query when T_end is true.

In the case of $M_4$, the event-condition and the condition-action modes are both *Deferred*. Mode $M_4$ is modeled by giving a higher priority to channels ch_cond and ch_action than channel ch_query when T_end is true.

In the case of $M_5$, the event-condition and the condition-action modes are *Decoupled*. Thus the condition (action) is evaluated (executed) as a different transaction after terminating the transaction associated with the corresponding event (condition). Mode $M_5$ is modeled by giving a higher priority to channel ch_query than to channels ch_cond and ch_action if T_end is false; otherwise a higher priority is given to channels ch_cond and ch_action than to channel ch_query.

## 4. Termination Checking of Active Database Rules

In order to verify the behavior of the active database rules, we first translate the proposed model described in Section 3 into the equivalent Promela model. Then, we check the termination property of rules using the Promela model and SPIN.

### 4.1 Promela Model of Active Database Rule Systems

The proposed model for the active database systems can be easily translated to an equivalent Promela model. We briefly illustrate the translation of our active database system model with the sample rule sets, $R_1$ and $R_2$, as shown in Example 1, the context, $C_1$, and the coupling mode, $M_1$. Sample Promela code is given in Appendix A. (For more details about Promela syntax, see Ref. 10).)

**Initialization**

Lines 1 to 5 define the number of rules, the size of the conflict set, the maximum number of operations in a transaction, the maximum number of transactions, and the size of buffered channels. Note that the state space of a Promela model is required to be a minimum in order to avoid the state explosion problem and hence to be verifiable by model checking. In the example Promela code, we set the values of variables in Lines 2 to 5 to be a minimum level required for checking the termination of the example rule set $R_1$ and $R_2$. These values should be determined to be suitable for checking a given target rule set.

Lines 7 to 26 declare global variables and user-defined types. The variable, `emp_rank`, represents the current value of the record, `Emp(rank)`, while the variable, `bonus_amount`, represents the current value of the record, `Bonus(amount)`. For simplicity, in this example, we abstracted a data model in such a way that the tables `Emp` and `Bonus` contain one tuple with the same employee's `id`. (In the case of $C_2$ or $C_3$, *DB_p* is necessary, and thus we prepared a buffered channel, which stores the past record states, $D_T$ or $D_E$.) The rule set, `rules[N]`, represents *RuleBase*, while the rule buffer, `cs`, represents *ConflictSet*. User-defined types, `event_type` and `rule_type`, represent respectively the types of events and rules.

**Table 3** Priorities of channels under different coupling modes.

| Modes | Priorities of Channels |
|---|---|
| $M_1$ | ch_query < ch_cond = ch_action |
| $M_2$ | ¬T_end → ch_query = ch_action < ch_cond<br>T_end → ch_query < ch_action < ch_cond |
| $M_3$ | ¬T_end → ch_query = ch_cond < ch_action<br>T_end → ch_query < ch_cond < ch_action |
| $M_4$ | ¬T_end → ch_query = ch_cond = ch_action<br>T_end → ch_query < ch_cond = ch_action |
| $M_5$ | ¬T_end → ch_cond = ch_action < ch_query<br>T_end → ch_query < ch_cond = ch_action |

Lines 28 to 40 declare the communication channels in Fig. 4. In order to model the channel priorities in **Table 3**, channels `ch_query`, `ch_event`, `ch_cond`, `ch_answer`, and `ch_action` are declared as asynchronous communication channels with a fixed buffer size. In our model, multiple messages (data operations or rules) are needed to be buffered in channels `ch_cond`, `ch_answer`, and `ch_action` but not in `ch_query` and `ch_event`. Hence, in the example Promela program, the buffer sizes of `ch_query` and `ch_event` are set to one, while the buffer sizes of `ch_cond`, `ch_answer`, and `ch_action` are set to the value of `size_buffer`. Other channels `ch_schedule`, `ack_eval`, and `ack_exe` are declared as synchronous channels, because they need only to pass a message but not to buffer a message.

Line 42 declares the Boolean variable, `T_end`, in Fig. 4.

Lines 45 to 77 declare the initial process. The initial process declares *RuleBase* and creates the process for *Transaction*, *Manager*, *Scheduler* and *Evaluator*. Promela processes are executed concurrently and scheduled nondeterministically. Using `d_step` and `atomic` statements reduces the state space and prevents the statements that are to be executed from being interfered with by other processes.

**Process Transaction**

Lines 79 to 103 declare the Process *Transaction*. This process chooses the data operation for the transaction and sends it to channel `ch_query`. The last operation of the transaction is specially marked so that the Process *Evaluator* can know the tail of the current transaction. In Promela execution semantics, each statement is either blocked or executable. As for the `do`-statement and the `if`-statement, if more than one of these statements is to be executable, that is, the guards of the statements are true, then one of the statements is non-deterministically selected and executed. In checking the model, SPIN exhaustively explores all possible behaviors. The Process *Transaction* models the possible transactions by selecting the transaction operations non-deterministically.

**Process Manager**

Lines 105 to 174 declare the Process *Manager*. When an event is received from the channel `ch_event`, this process finds the rules in `rules` triggered by the event and sends the identifiers for the triggered rules to the channel `ch_cond`. When the evaluation results are received from the channel `ch_answer`, this process sends to `cs` the identifiers for the rules whose conditions are true. After both procedures, `T_end` is set to false if `cs`, `ch_cond`, and `ch_action` are empty, that is, there are no rules to be evaluated or executed; otherwise a scheduling request is sent to the channel `ch_schedule`.

**Process Scheduler**

Lines 176 to 197 declare the Process *Scheduler*. When a scheduling request is received from the channel `ch_schedule`, this process non-deterministically chooses a rule from `cs` and sends the identifier for this rule to the channel `ch_action`. In Promela, a non-deterministic receive operation from a buffered channel is described using the operator '`??`'.

**Process Evaluator**

Lines 199 to 262 declare the Process *Evaluator*. This process receives data operations from the three channels `ch_query, ch_cond` and `ch_action`, and evaluates the operations received. The receiving order from the three channels follows the channel priority in Table 3 and then the different execution coupling modes can be easily modeled as explained in Section 3.2.

In the case of $M_1$, the channel priority is ch_query $<$ ch_cond $=$ ch_action in Table 3. This priority is expressed by Lines 208, 230, and 246. Lines 208, 230, and 246 respectively represent the conditions of receiving operations from the channels `ch_query`, `ch_cond` and `ch_action`. Line 208 signifies that an operation is received from the channel `ch_query` if the channel `ch_query` is not empty and the both channels `ch_cond` and `ch_action` are empty. This implies that ch_query $<$ {ch_cond, ch_action}. On the other hand, Line 230 (246) signifies that an operation is received from the channel `ch_cond` (`ch_action`) only if the channel is not empty. This implies that ch_cond $\geq$ {ch_query, ch_action} (ch_action $\geq$ {ch_query, ch_cond}). Therefore, by Lines 208, 230, and 246, the channel priority for $M_1$ can be expressed. Promela models for $M_2$ to $M_5$ are also easily obtained by changing the lines in this way, according to the channel priorities in Table 3.

## 4.2 Termination Checking

In our Promela model of an active database system, we can easily express the termination property for the rules using Promela's `progress` label, as shown in Lines 140 and 164 of the Promela code in Appendix A.

```
137      if
138         :: T_end==1 && (len(ch_cond)==0)
139             && (len(ch_action)==0) ->
140   progress1:    T_end = 0
141         :: else -> skip
142      fi;
```

In SPIN, `progress` is the label name for specifying the liveness properties. A statement marked by the `progress` label is required to be visited an infinite number of times in any infinite execution sequence. In our Promela code, the statements labeled with `progress` are executable if and only if there remain no rules to be evaluated or executed. Therefore, this implies that rule processing is terminated. If there is an execution sequence where the rule processing is never terminated, the statement labeled with `progress` is not visited and SPIN detects an error, as well as reporting the counterexample execution sequence.

**Example 2** Appendix B shows the result of checking whether there are any non-progress execution cycles for the sample Promela model given in Appendix A. The result shows that, for the sample model, no errors are detected and, thus, rules, $R_1$ and $R_2$, under the Context, $C_1$, and Coupling mode, $M_1$, satisfy the termination property. □

Note that, for rules $R_1$ and $R_2$, a previous static checking method in 1) detects a cycle representing a potential non-termination. Differently from the previous method, the proposed model checking method can answer that rules $R_1$ and $R_2$ indeed terminate mutual triggering as a result of an exhaustive search on the proposed finite active database model, since condition evaluation for rules is explicitly considered in our modeling.

(a) Termination Property

| | $M_1$ | $M_2$ | $M_3$ | $M_4$ | $M_5$ |
|---|---|---|---|---|---|
| $C_1$ | True | True | False | False | False |
| $C_2$ | False | False | False | False | False |
| $C_3$ | True | True | True | True | True |

(b) Time and Memory needed for Model Checking

| | | $M_1$ | $M_2$ | $M_3$ | $M_4$ | $M_5$ |
|---|---|---|---|---|---|---|
| $C_1$ | Time (s) | 0.320 | 0.400 | 0.160 | 1.370 | 1.410 |
| | Memory (Mbyte) | 325.886 | 326.910 | 324.657 | 339.505 | 339.300 |
| $C_2$ | Time (s) | 0.030 | 0.030 | 0.050 | 0.030 | 0.070 |
| | Memory (Mbyte) | 322.302 | 322.302 | 322.302 | 322.302 | 322.302 |
| $C_3$ | Time (s) | 0.450 | 0.710 | 0.700 | 0.890 | 0.730 |
| | Memory (Mbyte) | 327.934 | 331.723 | 331.62 | 333.976 | 332.337 |

**Fig. 5** Experimental results.

Note that in our approach each data item involved in rule processing is explicitly modeled. Because of this fine grained representation and of the exhaustive state search with the model checking technique, the proposed method can carry out exact termination detection, which means that it does not allow any false negatives.

SPIN verifies the claims specified by the LTL[11] formulae in addition to the process invariants. Therefore, other desirable safety properties can also be checked using our Promela model. For example, suppose that the *Query Evaluator* must answer a request from the *Condition Manager* in active database systems. In our model, this property is expressed using LTL as follows: `[] (nempty(ch_cond) -> <> nempty(ch_answer))`, which then can be verified using SPIN.

## 5. Experiment

In the experiment, we checked the termination property of the sample rules, $R_1$ and $R_2$, in Example 1 for different contexts and coupling modes by applying the proposed method. (In previous works[1),9),18)], using almost identical rules as $R_1$ and $R_2$, termination checking had been performed assuming only a specific context and coupling mode.)

First, we constructed the Promela models for every pair of contexts, $C_1$ to $C_3$, and coupling modes, $M_1$ to $M_5$, as explained in Section 4. Next, using SPIN, we checked the termination property of the Promela models. For model checking, we used a Linux workstation with an Intel Xeon 3.0 GHz and 4 GB memory.

**Figure 5** shows the experimental results. Figure 5-(a) shows the result of model checking the termination property for each pair of contexts and coupling modes. Based on the results, we know whether or not, depending on the contexts and coupling modes, a given rule set satisfies the termination property. For the cases where an error is detected by the model checking, we can obtain the counterexample trace that results in the rule processing to violate the termination property.

When the context is $C_1$ and the coupling mode is $M_3$, $M_4$, or $M_5$, the termination of rules $r_1$ and $r_2$ does not hold in the following counterexample trace: After the execution of a transaction query that updates the value of `rank` to an even number, rule $r_1$ and rule $r_2$, triggered by $r_1$, are consecutively executed, resulting in an odd value of `rank`. Subsequently, rule $r_1$ triggered by $r_2$ is executed after another transaction query updated the value of `rank` to an even number. In this way $r_1$ and $r_2$ are executed indefinitely. This can happen when the context is $C_1$ and the event-condition mode is not *Immediate*, that is, the coupling mode is $M_3$, $M_4$, or $M_5$.

If the context is $C_2$, the termination of rules $r_1$ and $r_2$ does not hold regardless of coupling modes. The counterexample trace is as follows: The value of `rank`, $x$, is even, when a transaction starts. After a query of a transaction updated the value of `rank`, rule $r_1$ is executed and thereafter rule $r_2$ triggered by $r_1$ is executed. Subsequently, $r_1$ and $r_2$ are indefinitely executed by turns, since value $x$ of `rank` when the transaction started is used for the condition evaluation of $r_1$ and thus $r_1$ is activated repeatedly.

Figure 5-(b) shows the time and memory required for model checking. For all cases, less than 1.5 seconds was needed, with a memory usage of about 330 Mbytes.

## 6. Related Work

As mentioned in Section 1, termination

checking of active database rules is generally an undecidable problem. Most previous works on analyzing active database rules have focused on static analysis 1),2),6),12),14),19),20).

Aiken, et al. 1) first presented the concept of a trigger-graph of rules and the approach using the trigger-graph basically predicts possible non-termination of active database rules by investigating sufficient conditions for termination, such as acyclicity in the trigger-graph.

Thereafter, in order to improve the accuracy of predicting potential non-termination, several techniques like the refinement of the trigger-graph and the investigation of more accurate termination conditions for the trigger-graph approach have been proposed 2),6),12),14),19),20).

However, to the best of our knowledge, no previous trigger-graph approaches have presented a general method that can deal with various contexts and coupling modes including *immediate*, *deferred*, and *decoupled*, unlike our proposed method. For example, only the *decoupled* coupling mode was addressed in Ref. 6).

Most previous work has addressed the problem of analyzing the behaviors of single events; however, there is little work on the verification of the complex behavior of *composite events*. An exception is Ref. 19), where termination analysis for composite events is addressed. Although for simplicity we assumed single events in the paper, it is not difficult to deal with composite events in the proposed approach. Specifically, such composite events can be described in Promela using, for example, user-defined record structures.

Unlike static methods previously proposed, our approach automates rule analysis with no need to create a new program. By using an existing model checking tool, not only the verification result (whether terminating or non-terminating) but also the non-terminating rule execution can be automatically obtained.

Model checking has been applied for active database rule analysis in two previous works 9),18). However, so far, the general modeling framework applicable to active database systems with various rule processing strategies has not been considered.

In Ref. 9), Ghazi and Huth presented an abstract model framework for active database management systems and implemented a prototype of the Promela code generator. In their model, an active database system was simply modeled as two concurrent processes called *system* and *environment*. In their model, questions dealing with how to model data operations for query evaluation, condition evaluation, and action execution were not addressed.

In Ref. 18), Ray, et al. verified the termination of rules using the model checker SMV 15). Their model assumed that only specific execution semantics were used for the rules, that is, a transaction consisted of a single data operation, that rule processing was performed only after a transaction was committed, and that the contexts for the condition and the action were limited to the current state of the data.

## 7. Conclusion

This paper's main contribution is a new method to model check active database rules with different contexts and coupling modes. First, we proposed the modeling framework, which is applicable to various active database systems with different contexts and coupling modes. Next, we presented how to translate our model to Promela and how to check the rule behavior using the SPIN model checker. Finally, using sample rules, we showed that the proposed method can efficiently check the termination of the rules with different contexts and coupling modes. To the best of our knowledge, this is the first time that the rule termination has been checked for context cases, $C_2$ to $C_3$, and coupling mode cases, $M_2$ to $M_5$.

The main difficulty in model checking active database rules is how to make an appropriate finite state model that represents the behavior of the active database rules. Since general model checking tools can only handle finite state models that do not cause state explosion, it is necessary to make an efficient finite abstraction of an active database system model.

Since our active database model incorporates some degree of abstraction, the proposed termination checking is not an exact method, like the previous works on rule termination analysis. However, the proposed method is more accurate than, for example, the previous work in Refs. 1), 12), 14), in the sense that the proposed method (1) can predict the termination of rules that are decided to terminate by the previous work, (2) can predict the termination of rules that are predicted to possibly non-terminate by the previous work, and (3) can predict the non-termination of rules that are decided not to terminate by the previous work and can furthermore specify undesirable behavior of the rules.

### 7.1 Discussion and Further Work

Here we discuss some questions raised by our approach and further works.

- *How accurate is the proposed method, compared to the previous trigger-graph approach?*

Because of the very different nature of the proposed approach and the trigger-graph approach, they have different strength and weakness with respect to the accuracy.

Although the proposed method has advantages over the trigger-graph approach as described in Section 6, it has also weakness with respect to accuracy. Specifically, the accuracy of the proposed method is suffered from the size of the model, since model checking tools can only handle finite state models of limited size.

Thus we would conclude that our approach can be best utilized, by using it together with the trigger-graph approach in a complementary fashion.

- *What is the advantage of using the proposed method for actual active database systems?*

The functionality of active database rules has started attracting more attention, not only in the traditional database systems area but also in the field of various databases and applications, for example, XML[16], sensor databases[22], etc. The proposed approach is intended not to depend on particular settings and thus can be helpful for rule analysis in various environments with different rule contexts and execution semantics.

Moreover, the proposed approach can be useful for complementing the previous methods. In the cases where further analysis is needed after the conventional methods predicted the potential non-termination, the proposed method can be used for a more detailed analysis, thus improving the accuracy of detecting non-termination of active database rules.

- *How well does the proposed method scale?*

The scalability of the proposed method depends on the state space size of the active database model and the verification power of the model checking tool. The state space size of the active database model is determined from many factors such as the number of rules, the type of events, the complexity of the conditions, etc. Consequently, the quantitative evaluation of scalability of the proposed method is not easy.

Our future work will evaluate the applicability of the proposed model through case studies using actual active database rules. Many active database rules can be set in active databases. We conjecture that, even in such cases, the rules that must be checked can be separated into small-to-moderate sized groups so that the proposed method can handle each of these groups of rules. We also anticipate that model checking tools will be continuously improved become to handle more complex system models.

Applying model checking of an infinite behavior model[3] to termination analysis of active database rules is also an interesting topic for further study.

**Acknowledgments** We would like to thank the anonymous reviewers for their valuable comments which helped in improving the quality of this paper. We also would thank Prof. Jun Miyazaki, the editor in charge, for his constructive comments to revise our manuscript.

This work was supported by Grant-in-Aids from the Ministry of Education, Culture, Sports, Science and Technology of Japan (No.18049055 and 17700033).

## References

1) Aiken, A., Hellerstein, J.M. and Widom, J.: Static analysis techniques for predicting the behavior of active database rules, *ACM Transactions on Database Systems*, Vol.20, pp.3–41 (1995).

2) Baralis, E., Ceri, S. and Paraboschi, S.: Improved rule analysis by means of triggering and activation graphs, *Rules in Database Systems*, Sellis, T. (ed.), pp.165–181, Springer-Verlag (1995).

3) Bultan, T., Gerber, R. and Pugh, W.: Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results, *ACM Transaction on Programming Languages and Systems*, Vol.21, No.4, pp.747–789 (1999).

4) Ceri, S., Fraternali, P., Paraboschi, S. and Tanca, L.: Active rule management in Chimera, *Active Database Systems*, Widom, J. and Ceri, S. (eds.), pp.151–176, Morgan Kaufmann Publishers (1996).

5) Clarke, E.M., Grumberg, O. and Peled, D.: *Model Checking*, MIT Press (1999).

6) Datta, A.H. and Son, S.H.: A Study of Concurrency Control in Real-Time, Active Database Systems, *IEEE Transactions on Knowledge and Data Engineering*, Vol.14,

No.3, pp.465–484 (2002).

7) Dayal, U.: Active database management systems, *Proc. 3rd International Conference on Data and Knowledge Bases* (1988).

8) Dittrich, K.R., Gatziu, S. and Geppert, A.: The Active Database Management System Manifesto: A Rulebase of ADBMS Features, *Proc. 2nd International Workshop on Rules in Database Systems*, pp.3–20 (1995).

9) Ghazi, T. and Huth, M.: An abstraction-based analysis of rule systems for active database management systems, Technical report, KSU-CIS-98-6 (1998).

10) Holzmann, G.J.: The Model checker SPIN, *IEEE Trans. Softw. Eng.*, Vol.23, No.5, pp.279–295 (1997).

11) Huth, M.R.A. and Ryan, M.D.: *Logic in Computer Science — Modelling and reasoning about systems*, Cambridge University Press (2000).

12) Karadimce, A.P. and Urban, S.D.: Refined Trigger Graphs: A Logic-Based Approach to Termination Analysis in an Active Object-Oriented Database, *Proc. International Conference on Data Engineering* (*ICDE'96*), pp.384–391 (1996).

13) Kulkarni, K., Mattos, N. and Cochrane, R.: Active database features in SQL3, *Active Rules in Database Systems*, Paton, N. (ed.), pp.197–219, Springer-Verlag (1999).

14) Lee, S.Y. and Ling, T.W.: Unrolling Cycle to Decide Trigger Termination, *Proc. 25th International Conference on Very Large Database* (*VLDB'99*), pp.483–493 (1999).

15) McMillan, K.L.: *Symbolic Model Checking*, Kluwer Academic (1993).

16) Park, I., Hyun, S., Kang, S. and Lee, D.: An XML-based Active Rule Support for Fast Reconfiguration of Business Processes, *Proc. Information Systems and Databases*, pp.234–239 (2002).

17) Paton, N.W. and Diaz, O.: Active database systems, *ACM Computing Surveys*, Vol.31, No.1, pp.63–103 (1999).

18) Ray, I. and Ray, I.: Detecting termination of active database rules using symbolic model checking, *Proc. Fifth East-European Conference on Advances in Databases and Information Systems* (2001).

19) Vaduva, A., Gatziu, S. and Dittrich, K.: Investigating Termination in Active Database Systems with Expressive Rule Languages, *Rules in Database System '97* (*LNCS 1312*), pp.149–164 (1997).

20) Voort, L. and Siebes, A.: Termination and confluence of rule execution, *Proc. 2nd international conference on Information and knowledge management*, pp.245–255 (1993).

21) Widom, J. and Finkelstein, S.J.: Set-oriented production rules in relational database systems, *Proc. SIGMOD'90*, pp.259–270 (1990).

22) Zoumboulakis, M., Roussos, G. and Poulovassilis, A.: Active rules for sensor databases, *Proc. 1st international workshop on Data management for sensor networks*, pp.98–103 (2004).

## Appendix

### A.1 Example Promela Program

```
#define num_R 2 /*number of rules*/
#define num_CS 4 /*size of the conflict set*/
#define N 2 /*maximum number of operations in a transaction*/
#define M 2 /*maximum number of transitions*/
#define size_buffer 4 /*size of buffered channels*/

mtype = {update,emp,rank,bonus,amount};

byte emp_rank; /*current value of Emp(rank) in DB*/
byte bonus_amount; /*current value of Bonus(amount) in DB*/

typedef event_type{ /*type of events*/
  mtype operation;
  mtype table;
  mtype field;
  byte m;
};

typedef rule_type{ /*type of rules*/
  event_type event;
  bool condition;
  event_type action;
};
rule_type Rules[num_R]; /*RuleBase: Set of rules*/

chan CS = [num_CS] of {byte}; /*ConflictSet*/

/*channels*/
chan ch_query = [1] of {bool,mtype,mtype,mtype,byte};
   /*{1 iff end of transaction,operation,table,field,value}*/
chan ch_event = [1] of {mtype,mtype,mtype,byte};
                       /*{operation,table,field,value}*/
chan ch_cond = [size_buffer] of {byte}; /*{rule-id}*/
chan ch_answer = [size_buffer] of {byte,bool};
                       /*{rule-id,evaluation-result}*/
chan ch_schedule = [0] of {bool};
chan ch_action = [size_buffer] of {byte}; /*{rule-id}*/

chan ack_eval = [0] of {bool};
chan ack_exe = [0] of {bool};

bool T_end;
   /*TRUE after queries of the current transition were completed*/

init
{
  /*declare RuleBase*/
  d_step{
  Rules[0].event.operation = update;
  Rules[0].event.table = emp;
  Rules[0].event.field = rank;
  Rules[0].event.m = 1;
  Rules[0].condition = 1;
  Rules[0].action.operation = update;
  Rules[0].action.table = bonus;
  Rules[0].action.field = amount;
  Rules[0].action.m = 10;

  Rules[1].event.operation = update;
  Rules[1].event.table = bonus;
  Rules[1].event.field = amount;
  Rules[1].event.m = 1;
  Rules[1].condition = 0;
  Rules[1].action.operation = update;
  Rules[1].action.table = emp;
  Rules[1].action.field = rank;
  Rules[1].action.m = 1;
  };

  /*run processes*/
  atomic{
    run Transaction();
    run Manager();
    run Scheduler();
    run Evaluator()
  }
}

proctype Transaction(){

  byte n; /*current number of operations in a transaction*/
  byte m; /*current number of transactions*/
  bool tail; /*TRUE for the last operation of a transaction*/

end:do
      :: m<M ->
         atomic{
           n++;
           if
             :: n<N -> tail=0 /*not tail of a transaction*/
             :: n<=N -> tail=1; n=0; m++ /*tail of a transaction*/
             :: else -> skip
           fi;
           /*select a data operation nondeterministically and
           buffer it as a query of a transaction to ch_query*/
           if
             :: ch_query!tail,update,emp,rank,1
             :: ch_query!tail,update,bonus,amount,1
           fi
         }
      :: break
  od
}

proctype Manager(){

  mtype ev_operation,ev_table,ev_field;
  byte ev_m, i;
  bool trg; /*TRUE if any rule is triggered*/
  bool result; /*evaluation result*/
end:do
      :: ch_event?ev_operation,ev_table,ev_field,ev_m;
                                     /*receiving an event*/
         atomic{
           i=0; trg = 0;
           do /*find Rules triggered by the event*/
             :: (i<num_R)->
                if
                  :: ev_operation==Rules[i].event.operation
                     && ev_table==Rules[i].event.table
                     && ev_field==Rules[i].event.field ->
                     /*request condition evaluation to Evaluator*/
                     ch_cond!i; trg = 1;
                     i++
                  :: else -> i++
                fi
             :: (i>=num_R) -> break
           od;
         };
         if
           :: (trg==0) ->
              /*if ConflictSet is empty and no rules to be evaluated
                or executed remain, set T_end to FALSE,
                otherwise call Scheduler*/
              if
                :: empty(CS) ->
                   if
                     :: T_end==1 && (len(ch_cond)==0)
                        && (len(ch_action)==0) ->
progress1:              T_end = 0
                     :: else -> skip
                   fi;
                   ack_eval!1
                :: nempty(CS) -> ch_schedule!0
              fi
           :: else -> ack_eval!1
         fi
      :: nempty(ch_answer) -> /*receiving evaluation results*/
         atomic{
           do
             /*add rules evaluated to true to ConflictSet*/
             :: nempty(ch_answer) ->
                ch_answer?i,result;
                if
                  :: result -> CS!i
                  :: else -> skip
                fi
             :: empty(ch_answer) -> break
           od
         };
         /*if ConflictSet is empty and no rules to be executed
           remain, set T_end to FALSE, otherwise call Scheduler*/
         if
           :: empty(CS) ->
              if
                :: T_end==1 && (len(ch_action)==0) ->
progress2:         T_end = 0
                :: else -> skip
              fi;
              ack_exe!1
           :: nempty(CS) -> ch_schedule!1
         fi
  od
}

proctype Scheduler(){

  byte id; /*id of a rule to be executed*/
  bool b;

end:do
      /*nondeterministically choose a rule to be executed
        from ConflictSet and send a request to Evaluator*/
      :: atomic
         {
           ch_schedule?b ->
           do
             :: nempty(CS) -> CS??id; ch_action!id
             :: empty(CS) -> break
           od;
           if
             :: b==0 -> ack_eval!1
             :: b==1 -> ack_exe!1
           fi
         }
  od
}

proctype Evaluator(){

  mtype ev_operation,ev_table,ev_field;
  byte ev_m;
  byte id; /*id of a rule to be evaluated or executed*/
  bool tail;

end:do
      :: /*(a) evaluation of query*/
         nempty(ch_query) && empty(ch_cond) && empty(ch_action)
         ->
         atomic{
           /*receiving a query from Transaction*/
           ch_query?tail,ev_operation,ev_table,ev_field,ev_m;
           /*if the query is the tail of the transaction,
                  set T_end to 1*/
           if
             :: tail -> T_end = 1
             :: else -> skip
           fi;
           /*evaluate the query and notice an event to Manager*/
           if
             :: ev_table==emp -> emp_rank = emp_rank+ev_m
             :: ev_table==bonus ->
                bonus_amount = bonus_amount+ev_m
             :: else -> break
           fi;
           ch_event!ev_operation,ev_table,ev_field,ev_m;
           ack_eval?1
         }
      :: /*(b) evaluation of condition*/
         nempty(ch_cond) ->
         atomic{
           do
             /*receiving the id of a rule to be evaluated*/
             :: empty(ch_cond) -> break
             :: nempty(ch_cond) ->
                ch_cond?id;
                /*evaluate the condition and send the result*/
                if
                  :: id==0 -> ch_answer!id,(emp_rank % 2 == 0)
                  :: id==1 -> ch_answer!id,1
                fi
           od;
           ack_exe?1
         }
      :: /*(c) execution of action*/
         nempty(ch_action) ->
         atomic{
           /*receiving the id of a rule to be executed*/
           ch_action?id;
           /*execute the action and notice an event to Manager*/
           if
             :: id==0 -> bonus_amount = bonus_amount+ev_m
             :: id==1 -> emp_rank = emp_rank+ev_m
             :: else -> break
           fi;
           ch_event!Rules[id].action.operation,
           Rules[id].action.table,Rules[id].action.field,
           Rules[id].action.m;
           ack_eval?1
         }
  od
}
```

## A.2 Example Output of SPIN

```
% ./pan -l
(Spin Version 4.2.6 -- 27 October 2005)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 156 byte, depth reached 808, errors: 0
   27741 states, stored (41570 visited)
   73537 states, matched
```

```
  115107 transitions (= visited+matched)
  127857 atomic steps
hash conflicts: 1445 (resolved)

Stats on memory usage (in Megabytes):
4.550   equivalent memory usage for states (stored*(State-vector + overhead))
3.877   actual memory usage for states (compression: 85.21%)
        State-vector as stored = 132 byte + 8 byte overhead
2.097   memory used for hash table (-w19)
320.000 memory used for DFS stack (-m10000000)
319.824 other (proc and chan stacks)
0.088   memory lost to fragmentation
325.886 total actual memory usage
```

(Received June 19, 2006)
(Accepted October 4, 2006)

(Editor in Charge: *Jun Miyazaki*)

**Eun-Hye Choi** received the M.E. and Ph.D. degrees in computer engineering from Osaka University in 1999 and in 2002, respectively. She had worked in Toshiba Corp. since 2002 and had engaged in research on XML database and transaction processing. Since 2004 she has worked in Research Center for Verification and Semantics of National Institute of Advanced Industrial Science and Technology (AIST). Her current research interests are in the areas of automatic verification and dependable distributed computing.

**Tatsuhiro Tsuchiya** received the M.E. and Ph.D. degrees in computer science from Osaka University in 1995 and 1998, respectively. He is currently an associate professor in the Department of Information Systems Engineering at Osaka University. His research interests are in the areas of model checking and distributed fault-tolerant systems.

**Tohru Kikuno** received the B.E., M.Sc., and Ph.D. degrees in electrical engineering from Osaka University in 1970, 1972, and 1975, respectively. He joined Hiroshima University from 1975 to 1987. Since 1990, he has been a Professor of the Department of Information and Computer Sciences at Osaka University. His research interests include the analysis and design of fault-tolerant systems, the quantitative evaluation of software development processes, and the design of procedures for testing communication protocols. He is a senior member of IEEE, a member of ACM, and a fellow of IPSJ (Information Processing Society of Japan) and IEICE (the Institute of Electronics, Information and Communication Engineers).