AIST-PS-2006-001

Model Checking Active Database Rules

Eun-Hye CHOI [1]
Tatsuhiro TSUCHIYA [2]
Tohru KIKUNO [2]

1: AIST CVS,　2: Osaka University

Model Checking Active Database Rules

Eun-Hye CHOI[†], Tatsuhiro TSUCHIYA[‡], Tohru KIKUNO[‡]
[†]Research Center for Verification and Semantics,
National Institute of Advanced Industrial Science and Technology (AIST),
5th Floor, 1–2–14 Shin-Senri Nishi, Osaka, 560–0083 Japan
[‡]Graduate School of Information Science and Technology, Osaka University
1–3 Machikaneyama, Toyonaka-shi, Osaka, 560–8531 Japan
[†] e.choi@aist.go.jp [‡]{t-tutiya,kikuno}@ist.osaka-u.ac.jp

Abstract

An active database is an autonomical database system that can react to events occurring inside and outside of the database. A set of active database rules defines a reactive behavior of the active database. One of the most potential problems with active database systems is non-termination of active database rules. This paper proposes an approach for automatically checking the termination of active database rules by a model checking technique. In our approach, we give a general framework for modeling active database systems and rules which can be useful for analyzing rule behaviors under various execution semantics and contexts of active database rules. Based on the proposed modeling framework, we model check the termination property of active database rules using a model checking tool, SPIN. Through experimental results, we show the feasibility of the proposed method.

1 Introduction

An *active database* [11] is a database system that has a functionality to react to events occurring inside and outside of the database, while a traditional database responds only to queries from users or applications outside of the database. Since this extra functionality of an active database is helpful to integrate reactive behavior of applications in a centralized and timely manner, active database systems have received attention from 1980s and a number of systems, e.g. Starburst [14], SQL-3 [9], HiPAC [5], and Chimera [3], have been developed.

The reactive functionality of active database systems is described by *rules* which have three components: an *event*, a *condition*, and an *action*. The *event* part defines an event that the rule is triggered by, the *condition* part defines a condition that the rule is activated, and the *action* part defines an action executed by the rule. Executing a rule can trigger other rules and thus the non-termination of active database rules is one of the most potential problems in active database systems.

The behavior of an active database rule system depends on not only a set of rules given to the system but also a strategy of rule processing adopted by the system. For example, which point of data is used as a context of rule processing and how soon a rule is evaluated and executed in or out of transactions depend on the rule processing strategy of an active database system. Consequently, it is difficult to predict the behavior and interactions of active database rules associated with a given rule processing strategy. Therefore, an automatic mechanism to analyze active database rules and verify desired properties such as the termination of rules is indispensable.

In this paper, we propose an approach for automatically verifying the termination and other safety properties of active database rules by *model checking* [4, 7, 10]. Model checking is an automated verification technique that can exhaustively check whether a finite state transition system satisfies a temporal logic formula or not. Given a system modeled by a finite state transition system and a property expressed by a temporal logic formula, verification of the property for the system is automatically performed by using existing powerful model checkers.

In our approach, we first give a modeling framework for constructing an abstract model of an active database rule system, which can be formally described and analyzed using model checker *SPIN* [7]. We next explain how to verify the termination and other safety properties of rules using the constructed model by model checking. Since our modeling framework is general in the sense that it is not limited to a specific active database setting, the proposed approach can be commonly applied for analyzing rule behaviors in various active database systems with different features of rule processing.

The rest of this paper is organized as follows: In Section 2, we introduce active database rule systems and model checking with SPIN. The proposed modeling framework which can deal with various execution semantics of rules is explained in Section 3. In Section 4, we describe how to verify the rule behavior using the proposed modeling framework. In Section 5, we show the experimental results of applying the proposed method for verifying the termination property of an example rule set under different features of rule processing. Finally, related works and the conclusion are respectively given in Sections 6 and 7.

r_1	**ON** update(emp(rank)) **IF** emp(rank) mod 2 = 0 **DO** update(bonus(amount)) bonus(amount) = bonus(amount) + 10
r_2	**ON** update(bonus(amount)) **IF** TRUE **DO** update(emp(rank)) emp(rank) = emp(rank) + 1

Figure 1: Example Active Database Rules

2 Preliminaries

2.1 Active Database Rule System

An active database rule system contains a set of rules that describe desired reactive behaviors. An active database rule consists of three components: an *event*, a *condition*, and an *action*, with syntax **ON**[*event*] **IF**[*condition*] **DO**[*action*].

Example 1 Assume a database containing two data tables: emp(id,rank) and bonus(empid,amount), where components id, rank, empid, and amount are records of tables emp and bonus. Consider the two rules r_1 and r_2 in Figure 1. Rule r_1 signifies that whenever emp(rank) is updated, bonus(amount) is increased by 10 if emp(rank) is even. Rule r_2 signifies that whenever bonus(amount) is updated, emp(rank) is increased by 1. □

An active database rule system is characterized by two models: a knowledge model and an execution model. The knowledge model describes the structural characteristics of rules, while the execution model captures the runtime characteristics of rule processing.

Characteristics of a knowledge model consist of the types of events and actions, the contexts of conditions and actions, etc. Types of an event include data modification (e.g. insert and update), clock (e.g. at 13:00 on every Monday), etc. A composition of events can also be regarded as an event. Types of an action include data retrieval and modification, transaction operation (e.g. commit or abort), an external call, etc.

Characteristics of an execution model consist of conflict resolution policy, scheduling policy, and execution coupling mode of rules, etc. When multiple rules are triggered and activated at the same time, a conflict of rules is often resolved by priorities of rules, which are defined numerically or relatively. Multiple rules are executed sequentially or in parallel.

In this paper, for the simplicity of modeling an active database rule system, we assume that the type of events and actions of rules is only data modification

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

Coupling Modes	Event-Condition	Condition-Action
M_1	Immediate	Immediate
M_2	Immediate	Deferred
M_3	Deferred	Immediate
M_4	Deferred	Deferred

and that rules and transactions are executed sequentially.

The behavior of active database rules depends on the knowledge and execution model of rule processing adopted by the system. Among the characteristics of the knowledge and execution model, *contexts* and *execution coupling modes* of rules are particularly important factors to determine the termination of rules. Therefore, in this paper, we especially focus on the contexts of rules in the knowledge model and the execution coupling modes of rules in the execution model.

Context indicates which state of a database is used in the rule processing. Here 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, respectively. In this paper, we consider the choices in Table 1 as contexts for the condition and for the action (denoted as the *condition-context* and the *action-context*, respectively). These contexts are actually used in existing active database systems [3, 5, 9, 14].

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 occurred. The *condition-action mode* determines when the action is executed after the corresponding condition was evaluated. The following coupling modes are supported in actual active database systems [3, 5, 9, 14]:

- *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.

Here we consider the four choices in Table 2 for the event-condition mode and the condition-action mode.

In this paper, we propose a general modeling framework which can be easily applied to active database rule systems with any of the contexts and execution coupling modes in Tables 1 and 2.

2.2 Model Checking and SPIN

Model checking is a verification technique that exhaustively checks whether or not a system modeled by a finite state transition system satisfies a property expressed by a temporal logic formula. Recently, model checking has become more and more attractive since, given a transition system and a temporal logic formula, model checking can be automatically and rapidly performed by using existing powerful tools. Model checking is also helpful to locate errors of systems because, when a system model does not satisfy a property formula, a counterexample that traces a system behavior violating the property is output.

SPIN [7] 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 claims specified by Linear Temporal Logic (LTL) formulas or process invariants which can express basic safety and liveness properties. SPIN performs on-the-fly verification and supports several useful state search and compression strategies. In our approach, we adopt SPIN because of its powerful verification capability and because an active database rule system can be naturally modeled as an asynchronous process system in Promela.

3 Modeling of Active Database Rule Systems

3.1 Basic Modeling Framework

Generally, active database rule systems have the following principal facilities:

- The *Event Detector* detects events occurring inside or outside of the database and triggers the rules associated with the events.
- The *Condition Manager* evaluates the conditions of the rules triggered by the event detector. Rules whose conditions are evaluated to true are stored in a set called a *conflict set*.
- The *Scheduler* chooses a rule from the conflict set updated by the condition manager according to the conflict resolution policy and fires the rule.
- The *Query Evaluator* executes queries from transactions and the condition manager and actions of rules fired by the scheduler. When evaluating

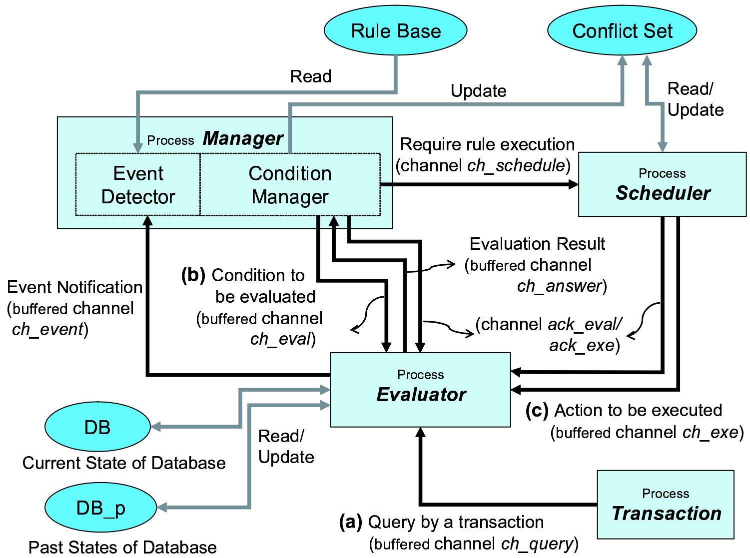

Figure 2: Abstract Model of an Active Database Rule System

queries, the query evaluator accesses not only the current state of the database but also past states of the database such as D_T and D_E if necessary.

We give a modeling framework that describes an active database system, which consists of the above facilities, a database, and a set of rules, and transactions to the database system, as will be explained in the following. The proposed modeling is general and is thus applicable to various active database rule systems.

Figure 2 shows our modeling of an active database rule system. The proposed model consists of the following four processes (illustrated using rectangles in the figure) : *Manager*, *Scheduler*, *Evaluator*, and *Transaction*. Process *Manager* models the event detector and the condition manager, Process *Scheduler* models the scheduler, and Process *Evaluator* models the query evaluator of the active database rule system. Process *Transaction* models the transactions from users or applications to the database system.

The proposed model also includes the following four kinds of data stores (illustrated using circles in the figure): *DB*, *DB_p*, *RuleBase*, and *ConflictSet*. *DB* and *DB_p* respectively store the current state and past states of the database. *RuleBase* maintains a set of rules defined in the system, and *ConflictSet* maintains the current conflict set of rules.

The processes read and update the data stores and communicate with each other via asynchronous message passing or asynchronous message passing using buffered channels. The behavior of each process is as follows:

- Process *Transaction*

  This process sends queries of transactions to Process *Evaluator* via asynchronous message passing using a buffered channel denoted by ch_query. We consider that a transaction consists of single or multiple data modification operations. Each operation of the transaction is buffered to ch_query.

- Process *Manager*

  This process iterates the following procedure:

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

- Process *Scheduler*

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

- Process *Evaluator*

  This process evaluates the following three kinds of queries: queries requested from Process *Transaction*, conditions of rules requested by Process *Manager*, and actions of rules requested by Process *Scheduler*.

  a) Evaluation of queries of transactions: After receiving a query from channel ch_query, evaluate the query and send a new event corresponding to the query to channel ch_event. This evaluation procedure is finished when receiving an acknowledgment message from a synchronous channel called ack_eval.

b) Evaluation of conditions of rules: After receiving identifiers of rules from channel ch_eval, evaluate conditions of the rules and send the evaluation result to channel ch_answer. This evaluation procedure is finished when receiving an acknowledgment message from a channel denoted by ack_exe.

c) Execution of actions of rules: After receiving an identifier of a rule from channel ch_exe, execute the action of the rule and send a new event corresponding to the action to channel ch_event. This evaluation procedure is finished when receiving an acknowledgment message from channel ack_eval.

To evaluate the above three kinds of queries, this process accesses to data of *DB* and/or *DB_p*. In the cases of (a) and (c), a new event may occur after the evaluation. In those cases, the acknowledgment is received from channel ack_eval after Process *Manager* noticed the new event and added rules triggered by the event to *ConflictSet*. In the case of (b), the acknowledgment is received from channel ack_exe after Process *Scheduler* sent an activated rule to channel ack_exe. Waiting for the acknowledgment prevents each evaluation procedure of (a), (b), and (c) from being interleaved with other procedures.

3.2 Modeling Different Execution Semantics

The basic framework for modeling an active database rule system was proposed in Section 3.1. The modeling framework is applicable to various active database rule systems with different knowledge and execution models of rules. Here, based on the modeling framework, we explain how to model active database rule systems with the contexts in Table 1 and the execution coupling modes in Table 2.

First, as to the condition-context and the action-context, we consider three choices C_1, C_2, and C_3 in Table 1. In our modeling framework, as shown in Figure 2, Process *Evaluator* reads current and/or past states of the database when evaluating queries. Active database rule systems with C_1, C_2, or C_3 are then straightforwardly modeled by selecting the state of the database to be read by Process *Evaluator* in our modeling framework as follows.

In the case of C_1, D_C, the state when the condition is evaluated, and D_A, the state when the action is executed, are respectively used for the condition-context and the action-context. Hence, to model C_1, we let 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 Process *Evaluator* access *DB_p* when evaluating conditions and executing actions.

Table 3: Priorities of Channels under Different Execution Coupling Modes

Modes	Priorities of Channels
M_1	ch_query $<$ ch_exe $=$ ch_eval
M_2	$\neg$T_end $\rightarrow$ ch_query $=$ ch_exe $<$ ch_eval
	T_end $\rightarrow$ ch_query $<$ ch_exe $<$ ch_eval
M_3	$\neg$T_end $\rightarrow$ ch_query $=$ ch_eval $<$ ch_exe
	T_end $\rightarrow$ ch_query $<$ ch_eval $<$ ch_exe
M_4	$\neg$T_end $\rightarrow$ ch_query $=$ ch_exe $=$ ch_eval
	T_end $\rightarrow$ ch_query $<$ ch_exe $=$ ch_eval

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 an action of a rule, we let Process *Evaluator* access the state of *DB_p* corresponding to the event of the rule.

Next, as to execution coupling modes, consisting of event-condition and condition-action modes, we consider four choices M_1, M_2, M_3, and M_4 in Table 2. In our modeling framework, Process *Evaluator* performs the three kinds of query evaluations: (a) the evaluation of queries from transactions, (b) the condition evaluation, and (c) the action execution. Queries for (a), (b), and (c) are respectively received from buffered channels ch_query (from Process *Transaction*), ch_eval (from Process *Manager*), and ch_exe (from Process *Scheduler*). Active database rule systems with M_1, M_2, M_3, or M_4 are then modeled by assigning priorities to the three channels to determine the order of receiving queries in Process *Evaluator* as shown in Table 3.

In the case of M_1, coupling modes for the event-condition and the condition-action are *Immediate*, and thus the condition (the action) has to be evaluated (executed) immediately after the event occurrence (the condition evaluation). Mode M_1 is modeled by giving a higher priority to channels ch_exe and ch_eval 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 the event occurrence while the action has to be executed within the same transaction. Mode M_2 is modeled as follows: Since the event-condition mode is *Immediate*, give a higher priority to channel ch_eval than to channels ch_exe and ch_query. T_end in the table denotes a boolean variable that is true during the period after queries of the current transaction were completed and before the next transaction starts. Since the condition-action mode is *Deferred*, give a higher priority to channel ch_exe than 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*. Mode M_3 is thus modeled similarly to the case of M_2

in the following way: Give a higher priority to channel ch_exe than to channels ch_eval and ch_query. Give a higher priority to channel ch_eval than channel ch_query when T_end is true.

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

4 Verification of Active Database Rules

In order to verify the behavior of active database rules, we first translate the proposed model described in Section 3 to a Promela model and next model check termination and safety properties of the Promela model using SPIN.

4.1 Promela model of Active Database Rule Systems

The proposed model for active database rule systems can be easily translated to a Promela model. Here we briefly illustrate the translation of our model of an active database rule system with example rule sets, R_1 and R_2, in Example 1 and context C_1 and coupling mode M_1. The example Promela code is given in Appendix A. (For details of the Promela syntax, see [7].)

Lines 1–42 are used to declare global variables.

- Variables `emp_rank` and `bonus_amount` respectively represent the current values of records `emp(rank)` and `bonus(amount)`. (In the cases of C_2 or C_3, *DB_p* is necessary, and thus we prepare a buffered channel which stores past states D_T or D_E of records.)

- Set of rules `rules[N]` and buffer of rules `cs` respectively represent *RuleBase* and *ConflictSet*. User-defined types `event_type` and `rule_type` respectively represent the types of events and rules.

- Communication channels in Figure 2 and boolean variable `T_end` are prepared. `ch_query, ch_event, ch_eval`, `ch_answer`, and `ch_exe` are asynchronous communication channels, while `ch_schedule`, `ack_eval`, and `ack_exe` are synchronous ones.

Lines 44–76 declares an initial process. The initial process declares *RuleBase* `rules` and creates processes of *Transaction*, *Manager*, *Scheduler* and *Evaluator*. Promela processes are executed concurrently and scheduled nondeterministically. Using `d_step` and `atomic` statements reduces a state space by making statements to be executed without being interleaved by other processes.

Lines 78–102 declares Process *Transaction*. This process chooses a data operation as a query of a transaction and sends it to channel `ch_query`. The last operation of a transaction is specially marked so that Process *Evaluator* can know the tail of the current transaction. In the Promela execution semantics, each statement is either blocked or executable. As for `do`-statement and `if`-statement, if more than one statements in them are executable, that is, guards of the statements are true, one of the statements is nondeterministically selected and executed. SPIN exhaustively explores all possible behaviors in checking the model. Process *Transaction* models possible transactions by selecting operations of a transaction nondeterministically.

Lines 104–171 declares Process *Manager*. When an event is received from channel `ch_event`, this process finds rules in `rules` triggered by the event and sends identifiers of triggered rules to channel `ch_eval`. When evaluation results are received from channel `ch_answer`, this process sends to `cs` the identifiers of rules whose conditions are true. After both procedures, `T_end` is set to false if `cs`, `ch_eval`, and `ch_exe` are empty, that is, there are no rules to be evaluated or executed; otherwise a request of scheduling is sent to channel `ch_schedule`.

Lines 173–257 declares Process *Scheduler*. When a request of scheduling is received from channel `ch_schedule`, this process nondeterministically chooses a rule from `cs` and sends the identifier of the rule to channel `ch_exe`. Nondeterministic receive operation from a buffered channel is described using operator '`??`' in Promela.

Lines 195–257 declares Process *Evaluator*. This process receives queries from channels `ch_query,` `ch_eval` and `ch_exe`, and evaluates those queries received. Receiving order from the three channels follows the channel priority as explained in Section 3.2. In the case of M_1, the channel priority is $\mathsf{ch\_query} < \mathsf{ch\_exe} = \mathsf{ch\_eval}$ and this is described as Lines 205, 225, and 241. (Promela models for M_2–M_4 are easily obtained by changing the lines according to the channel priorities in Table 3.)

4.2 Termination Checking

In our Promela model of an active database rule system, we can easily express the termination property of rules using a `progress` label of Promela, as described in Lines 137 and 164 of Promela code in Appendix A.

```
135    if
136     :: T_end==1 && (len(ch_eval)==0) && (len(ch_exe)==0) ->
137  progress1: T_end = 0
138     :: else -> skip
139    fi;
```

In SPIN, `progress` is a label name for specifying liveness properties; a statement marked by the `progress` label is required to be visited infinitely 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, and this means that the rule processing is terminated. If there is an execution sequence where the rule processing is never terminated, the statement

(a) Termination Property

	M_1	M_2	M_3	M_4
C_1	True	True	False	False
C_2	False	False	False	False
C_3	True	True	True	True

(b) Time and Memory needed for Model Checking

		M_1	M_2	M_3	M_4
C_1	Time(s)	0.320	0.400	0.160	1.370
	Memory(Mbyte)	325.886	326.910	324.657	339.505
C_2	Time(s)	0.030	0.030	0.050	0.030
	Memory(Mbyte)	322.302	322.302	322.302	322.302
C_3	Time(s)	0.450	0.710	0.700	0.890
	Memory(Mbyte)	327.934	331.723	331.62	333.976

Figure 3: Experimental Results

labeled with `progress` is not visited and then SPIN detects an error and reports the counterexample execution sequence.

Example 2 Appendix B shows the result of model checking whether there are non-progress execution cycles for the example Promela model in Appendix A. The result shows that no error is detected and thus one can know that example rules R_1 and R_2 under Context C_1 and Coupling mode M_1 satisfy the termination property. □

SPIN verifies claims specified by LTL [8] formulas in addition to process invariants. Therefore, other desirable safety properties can also be checked using our Promela model. For example, suppose that *Query Evaluator* must answer requests from *Condition Manager* in active database systems. In our model, this property is expressed using LTL as follows: `[] (nempty(ch_eval) -> <> nempty(ch_answer))`. The property can then be verified using SPIN.

5 Experiment

In the experiment, we checked the termination property of the example rule set, R_1 and R_2, in Example 1 under different contexts and execution coupling modes by applying the proposed method. First, we generated Promela models for every pairs of contexts C_1–C_3 and execution coupling modes M_1–M_4 as explained in Section 4. Next, we checked the Promela models using SPIN. For model checking, we used a Linux workstation with Intel Xeon 3.0GHz and 4GB memory.

Figure 3 shows the experimental results. Figure 3-(a) shows the result of model checking the termination property for each pair of contexts and execution coupling modes. From the result, one can know that whether the rule set satisfies the termination property or not depends on the contexts and the coupling modes considered.

Figure 3-(b) shows the time and memory required for the model checking. For all cases, the needed time was less than 1.5 seconds while the memory size actually used was around 330 Mbytes.

6 Related Work

In general, detecting termination of active database rules is an undecidable problem, and most previous works on analyzing active database rules have been focused on static analysis [1, 2, 13]. Such static works have provided principal conditions for termination of rules (e.g. acyclicity of a triggered graph of rules) but are not convenient to predict and specify undesirable behavior of rules under actual database systems.

Model checking has been applied for active database rule analysis in two previous works [6, 12]. However, the general modeling framework applicable to active database rule systems with different execution semantics has not considered so far.

In [6], T. S. Ghazi and M. Huth presented an abstract modeling framework for active database management systems and implemented a prototype of a Promela code generator. In their modeling, an active database system was simply modeled as two concurrent processes called *system* and *environment*. In their model, how to model data and data operations for the query evaluation, the condition evaluation, and the action execution were not addressed.

In [12], I. Ray et al. presented the verification of the termination of rules using model checker SMV [10]. Their modeling assumed only specific execution semantics for rules where a transaction consists of a single data operation, the rule processing is performed only after a transaction committed, and contexts for the condition and the action are limited to the current state of data.

7 Conclusion

In this paper, we proposed a new method to automatically check active database rules by model checking. We first proposed the modeling framework which is applicable to various active database systems with different execution semantics and contexts of rules. We next presented how to translate our model to Promela and check the rule behavior using the model by SPIN model checking. Finally, through the experiment using example rules, we showed that the proposed method can efficiently check the termination of the rules with different execution semantics and contexts. To the best of our knowledge, this is the first time that the termination of rules has been checked for the cases of contexts C_2–C_3 and execution coupling modes M_2–M_4. Our future work includes the evaluation of the applicability of the proposed method to actual active database rules with practical size.

References

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

[2] E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In T. Sellis, editor, *Rules in Database Systems*, pages 165–181. Springer-Verlag, 1995.

[3] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Active rule management in chimera. In J. Widom and S. Ceri, editors, *Active Database Systems*, pages 151–176. Morgan Kaufmann Publishers, 1996.

[4] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[5] U. Dayal. Active database management systems. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, 1988.

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

[7] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[8] M. R. A. Huth and M. D. Ryan. *Logic in Computer Science - Modelling and reasoning about systems.* Cambridge University Press, 2000.

[9] K. Kulkarni, N. Mattos, and R. Cochrane. Active database features in sql3. In N. Paton, editor, *Active Rules in Database Systems*, pages 197–219. Springer-Verlag, 1999.

[10] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic, 1993.

[11] N. W. Paton and O. Diaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

[12] I. Ray and I. Ray. Detecting termination of active database rules using symbolic model checking. In *Proceedings of the Fifth East-European Conference on Advances in Databases and Information Systems*, 2001.

[13] L. van der Voort and A. Siebes. Termination and confluence of rule execution. In *Proceedings of the second international conference on Information and knowledge management*, pages 245–255, 1993.

[14] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of SIGMOD'90*, pages 259–270, 1990.

Appendix

A. 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}; /*Conflict Set*/

/*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_eval = [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_exe = [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 queries in a transaction*/
  byte m; /*current number of transactions*/
  bool tail; /*TRUE for the last query 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_eval!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_eval)==0) && (len(ch_exe)==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_exe)==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_exe!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
    :: if
           :: /*(a) evaluation of query*/
              nempty(ch_query) && empty(ch_eval) && empty(ch_exe) ->
              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 current 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_eval) ->
              atomic{
                do
                  /*receiving the id of a rule to be evaluated*/
                  :: empty(ch_eval) -> break
                  :: nempty(ch_eval) ->
                     ch_eval?id;
                     /*evaluate the condition and send the result to Manager*/
```

```
                     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_exe) ->
              atomic{
                /*receiving the id of a rule to be executed*/
                ch_exe?id;
                /*execute the action and notice an event to Manager*/
                if
                  :: id==0 -> bonus_amount = bonus_amount+10
                  :: id==1 -> emp_rank = emp_rank+1
                  :: else -> break
                fi;
                ch_event!Rules[id].action.operation,Rules[id].action.table,
                Rules[id].action.field,Rules[id].action.m;
                ack_eval?1
              }
         fi
  od
}
```

B. 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
```

モデル検査によるアクティブデータベースルールの自動検証
(in English)
(算譜科学研究速報)
発行日：2006 年 2 月 10 日
編集・発行：独立行政法人産業技術総合研究所関西センター尼崎事業所
システム検証研究センター
同連絡先：〒661-0974 兵庫県尼崎市若王寺 3-11-46
e-mail：informatics-inquiry@m.aist.go.jp

Model Checking Active Database Rules
(Programming Science Technical Report)
Jan. 20, 2006
Research Center for Verification and Semantics (CVS)
AIST Kansai, Amagasaki Site
National Institute of Advanced Industrial Science and Technology (AIST)
3-11-46 Nakouji, Amagasaki, Hyogo, 661-0974, Japan
e-mail: informatics- inquiry@m.aist.go.jp