Detecting Termination of Active Database Rules Using Symbolic Model Checking

Indrakshi Ray and Indrajit Ray

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873
Email: {iray, indrajit}@cs.colostate.edu

Abstract. One potential problem of active database applications is the non-termination of rules. Although algorithms have been proposed to detect non-termination, almost all provide a conservative estimate; that is, the algorithms detect all the potential cases of non-termination. These algorithms then leave it to the database programmer to analyze each case to determine if indeed the rules are non-terminating. Our work proposes the use of automated tools for software specification and verification, to analyze active database applications. In this paper we show how the database programmer can automatically detect non-termination using an existing symbolic model checker. Our approach does not require much expertise on the part of the database programmer, and can be used to detect termination cases which the conservative approaches reject as non-terminating ones. Our approach, thus, complements the conservative approaches.

1 Introduction

Active database systems [1, 7–9, 16–18] appear to be a very promising technology. However, users are reluctant to use it because of the uncertainty associated with how a set of active database rules acting on their own will interact with each other and with other transactions [15]. To ensure that the application will behave in a predictable manner, active database applications need to be formally verified. In this paper, we focus on how one important property of active database systems, namely, termination of active database rules, can be formally verified using symbolic model checking.

Detecting termination of active database rules is, in general, an undecidable problem. However, researchers have proposed conditions [2, 12, 13] which are sufficient to ensure termination. Thus, a given problem can be analyzed to check whether it satisfies these conditions; if it does then the rule-sets are guaranteed to terminate, otherwise they may or may not. One such condition is the acyclicity of the triggering graph [2]. Absence of cycles in a triggering graph indicates that the rules will eventually terminate. Cycles in the triggering graph indicate potential for non-termination. In other words, if a cycle exists in the triggering graph, further analysis must be done before one can give a more definite answer about termination. The question is how to analyze the application to find out whether the rules involved in the cycle will indeed terminate or not. This paper aims to answer the above question. In this work we show how model checking,

a software verification technique, can be used to analyze the application and give some useful results about termination.

To illustrate our approach we use the SMV model checker[1] [14]. The first step involves converting the active database application to an SMV specification. The second step is to express the termination property as a CTL (Computational Tree Logic) formulae. The third step involves using the model checker to see if the property holds. The model checker performs an exhaustive search and reports whether the property holds or not. If the property does not hold, the model checker provides a trace showing the violation of the property.

We find that our method is able to

1. predict termination of rules that are also predicted to terminate by the triggering graph approach;
2. predict termination of rules that do terminate, but are diagnosed as possibly non-terminating by the triggering graph approach and
3. predict non-termination for rules that are indeed non-terminating and, additionally, provide a sequence of rule execution that leads to the non-termination.

The last property of our approach is specially important; it provides a hint to the database designer how to modify the application to ensure correct behavior.

The rest of the paper is organized as follows: Section 2 discusses some related work. Section 3 gives an overview of our approach. Section 4 first presents an overview of the SMV model checker and then illustrates our approach using a motivating example. Section 5 presents a second example with terminating rule sets and the outcome of analyzing this example with SMV. Section 6 concludes this paper. Appendix 1 provides the SMV specification corresponding to the example described in section 4.2. Appendix 2 provides the output generated by the SMV model checker for the specification given in Appendix 1.

2 Related Work

One of the major works on rule termination detection is that by Widom et al. [2]. In this work, the authors propose a static, graph theoretic approach to detect if a set of active database rules have the termination as well as other desirable properties, such as, confluence and observable determinism. This work involves building a *triggering graph* of an active database application and then analyzing it to determine the properties of the application. The vertices of the triggering graph correspond to the active rules of the application. There is an edge from vertex *i* to another vertex *j* if there is a possibility that the rule represented by vertex *i* can trigger the rule represented by vertex *j*. The absence of cycles in the graph indicate that the given set of rules are terminating. Cycles in the execution graph identify potential scenarios of non-termination.

Some examples of termination that are not detected by the triggering graph cited by the authors are as follows: (i) The action of some rule *r* deletes tuples from a table, and

[1] The SMV model checker is available from `http://www.cs.cmu.edu/afs/cs/project/modck/pub/www/modck.html`

no other rules on the cycle insert tuples into these tables. (ii) The action of some rule r on the cycle performs a monotonic update, guaranteeing that the condition of some rule r' eventually becomes false. In both these cases the triggering graph generates a cycle indicating a possibility of non-termination. Once a cycle has been detected, the rules involved in the cycle must be analyzed. The authors do not specify how the users can perform such an analysis. In a real-world active database application, a cycle may involve a large number of rules. Manually analyzing all these rules may be tiresome and error-prone. In our work we show how this analysis can be automated by using model checking. Note that, both the cases of termination identified above can be detected using our approach.

Another work on termination detection has been proposed by Karadimce et al. [12]. The authors argue that a simple syntactic analysis may produce a graph containing cycles whereas a more detailed analysis may produce an acyclic graph. If a detailed analysis can show that a rule p can never trigger another rule q, then the edge (p, q) can be safely removed from the graph. Removing edges in this manner may at some stage yield an acyclic graph indicating that the rules do indeed terminate.

Baralis and Widom [6] introduced another kind of graph known as *activation graph* to detect termination. The vertices corresponds to the rules of the application. An edge (i, j) signifies that the action of rule i may satisfy the condition of rule j. Acyclicity of this graph means definite termination. Baralis proposed a rule-reduction method [5] that makes use of both activation and triggering graphs. Any vertex that does not have an incoming activation or triggering edge is removed together with its outgoing edges. If all the vertices are removed in the process, the rule set is indeed terminating.

Rule analysis focussing on the termination and confluence properties have also been proposed by Van der Voort et al. [13]. The authors base their work on object-oriented database and propose a design theory which they use to ascertain whether the rules are terminating and/or confluent. The focus is on deciding termination in a fixed number of steps. The rule model used is very restrictive – rule actions can only modify data selected by the rule condition, and it appears that insertions and deletions are not allowed. Decidability results for the termination of rules has also been given by Bailey et al. [4]. In a separate work Bailey et al. [3] show how collecting semantic information can be useful in termination analysis.

3 Our Approach

Our approach requires building a finite state model of the active database application which can then be verified by an existing verification technique known as *model checking*. Model checkers have been traditionally used to verify hardware devices. They require that the model to be verified be represented as a finite state machine. This requirement has historically limited the use of model checking for software verification, since software systems, in general, are infinite state machines. Only recently researchers have shown how to build a finite state abstraction of the software system and verify it using model checking [11, 19].

In our approach, we first build a finite state abstraction of an active database application and then verify the finite state model using the symbolic model checker known as

SMV. We use the input language to the SMV to represent the finite state machine. The termination properties are represented as Computational Tree Logic (CTL) (a subset of branching time temporal logic) formulae. The SMV model checker verifies the model by performing a search on the state space and comes up with the appropriate response: either it prints that the property holds or produces a counterexample illustrating the violation of the property. Since the model checker searches on the entire state space, our approach is able to correctly predict termination of rules. Moreover, the counterexample, generated by the model checker, provides the designer with a trace of rule execution that results in non-termination. We believe that such a trace gives enough information to the designer to suitably modify the active database application.

We use two examples (many others are, of course, possible) to describe our approach to rule termination analysis. The rules in the first example trigger each other indefinitely and the termination property does not hold. The result obtained in this case concurs with that obtained by applying the triggering-graph analysis of Widom et al. [2]. Next we modify the example slightly so that the rules do terminate. The corresponding triggering-graph has cycles indicating the possibility that some rules may not terminate. Our automated analysis, however, indicates that the rules do terminate.

3.1 Rule Execution Semantics

The modeling of the application in SMV is dependent on the rule execution semantics. We assume that our active database system incorporates the following rule execution semantics. Note that, our choice was arbitrary. We could have chosen a different rule execution semantics; in such a case the model in SMV would be different. How the modeling changes with different rule execution semantics will be explored in a future work.

1. Sequential Execution: We assume a sequential mode of execution. That is, only one input transaction or active rule is processed at a time.
2. Rule Processing Granularity: The rules are processed after each occurance of a transaction. We assume that the transaction consists of a single database operation.
3. Conflict Resolution: If two or more rules are triggered at any time, the rule chosen for activation depends on the priorities specified in the rule definition. The other rules are queued up for later execution. If no priorities are specified, one rule is chosen arbitrarily for activation.
4. Iterative Rule Processing: One rule is selected and processed at a time before selecting and processing the other rule.
5. Set-Oriented Execution: We assume that a rule is executed once for all database instances triggering the rule or satisfying the rules condition.
6. Coupling Mode is Dependent-Decoupled: The rule processing takes place only after the original transaction has committed.

3.2 A Note on Notation

The notations we use are as follows. We use the **bold font** to describe database variables, sans serif font to describe the SMV variables, and *italics* to describe SMV keywords.

4 Modeling the Active Database Application

At this point we briefly describe the SMV model checker [14]. This will help the user understand how we build finite state abstractions of the active database application.

4.1 The SMV Model Checker

The input to the SMV model checker is an SMV program. The program consists of four parts: (i) declarations of state variables, (ii) the initial states of the variables, (iii) the transition relations that change the state of the variables and (iv) the specification of the properties to be verified. We briefly describe the syntax of each of these four parts.

Declaration of State Variables: The variables used in an SMV specification must be declared before use. The declaration also includes the type of the variable. A state variable can be of the following types: boolean, scalar, and fixed arrays. In the example below three variables x, y and z have been declared. The variable x is a boolean variable, that is, it can take one of two values: 0 or 1. The variable, y, is of type enumerated, that is, it can take one of the three values A, B or C. The variable z can take any value in the range 1 to 5.

```
x : boolean;
y : {A, B, C};
z : 1..5;
```

State Initialization: The SMV *init* function defines the initial values of the variables. For example, suppose variable x is initialized to 0. This is specified in SMV as:

```
init(x) := 0;
```

State Transformation: The SMV *next* function defines how the next state values of the variables are computed using the current state values of the variables. Often a variable changes value depending on whether certain conditions are satisfied by the current state variables. Moreover, the value may change in different ways if different conditions are satisfied. These conditions can be specified using a *case* statement. A *case* statement returns the first expression on the right hand side of the colon (:), such that, the condition on the left hand side of the colon is true. The default case is often specified as the last expression of the *case* statement: the left hand side of the colon is a 1 and the right hand side is the default value. The example below will help illustrate the point. The next state value of the variable y is A if the current state values of x and z satisfy both the conditions x and (z = 1). The next state value of y is B if y currently equals B and z equals 2. If neither of these conditions are true, that is, in the default case, the value of y remains unchanged.

```
next(y) :=
 case
   x & (z = 1): A;
   (y = B) & ( z = 2) : B;
   1 : y;
 esac;
```

Specifications of the Properties using CTL: The properties to be verified must be specified as Computational Tree Logic (CTL) formulae. A CTL formula is a boolean expression, an existential (E) path formula, an universal (A) path formula, or the application of standard boolean operators to CTL formulae. A path formula is the application of the temporal operators next (X), eventually (F), or globally (G), to a CTL formula. For example if we want to state that it is always true that if `X` is true in a state, then in some future state `Y` will be true we specify it as follows:

```
AG(X -> EF(y))
```

4.2 Example Application with Non-Terminating Rules

The database application has a number of tables, user-defined transactions, and rules. To keep the example short, we describe only those entities that are relevant for our purpose. The two tables that are of interest are: **emp(id,rank,salary)** and **bonus(emp-id,amount)**. Table **emp** records each employee's rank and salary, table **bonus** records a bonus amount awarded to each employee.

The two transactions that we are interested in are: **updateRank** and **updateAmt**. The transaction **updateRank** increases the rank of an employee and **updateAmt** increases the bonus amount. To keep the example simple, we assume that an employee's rank is always increased by 1 and the bonus is always increased by 10.

We define two rules. The first rule, **bonusRank**, states that whenever an employee's bonus is increased by 10, the employee's rank is increased by 1:

create rule **bonusRank** on **bonus**
when updated(**amount**)
  then update **emp**
    set **rank** = **rank** + 1
      where **id** in (select **emp-id** from **new-updated**, **old-updated**
        where **new-updated.emp-id** = **old-updated.emp-id**
          and **new-updated.amount** - **old-updated.amount** = 10)

The second rule, **rankBonus**, states that whenever an employee's rank is modified, that employee's bonus is increased by 10.

create rule **rankBonus** on **emp**
when updated(**rank**)
  then update **bonus**
    set **amount** = **amount** + 10
      where **emp-id** in (select **id** from **new-updated**)

Note that these two rules trigger each other indefinitely.

4.3 Converting the Database Application to an SMV Specification

A database application often has a large number of states; it is possible for the application to have an infinite number of states. The specification that is to be executed by the

SMV model checker must have a finite a number of states; in fact, to be efficiently executed by the model checker the number of states must be minimized. Thus, the biggest challenge is in downsizing the application without changing its properties.

In this section we give an algorithm for converting a database application to an SMV specification. First, we give the general algorithm for converting the database application to an SMV specification. Then, we give some hints on how to reduce the state space.

Step 1: Creation of SMV State Variables

Step 1a: For each cell in each table of the database define two SMV variables. One of these variables will contain the current state value of the cell and the other will contain the previous state value. As a convention the previous state variables contain the prefix 'old'. Recall that each variable in SMV must be specified with a range. For now, we assume that the range of this variable equals the domain of the values that the corresponding attribute can take. Later on, we specify how to reduce this range to minimize the state explosion problem.

State variables for the tables in the example: For each row i (where i = 1,2,3, ...) in the table **emp** we define the variables id-i, rank-i, salary-i to store the current state values of the respective cells, and the variables oldid-i, oldrank-i, oldsalary-i to store the previous state values. We create similar variables for the table bonus. Assuming that each table has just one row, the following variables are created. id-1 , rank-1, salary-1, oldid-1 , oldrank-1, oldsalary-1, empid-1, amount-1, oldempid-1, oldamount-1. For the sake of convenience we elimiate the suffix -1 from the variables. The declaration of the variables and their types using a pseudo SMV notation[2] is:

```
id : 0..1000;
rank : 0..no upper limit;
salary : 0..no upper limit;
oldid : 0..1000;
oldrank : 0..no upper limit;
oldsalary : 0..no upper limit;
empid  : 0..1000;
amount : 0..no upper limit;
oldempid  : 0..1000;
oldamount : 0..no upper limit;
```

Step 1b: For each rule in the application, define an SMV variable. Each of these variables are specified as boolean. The value 1 of the variable indicates that the corresponding rule has been triggered. The value 0 indicates that the corresponding rule has not been triggered.

State variables for the rules in the example: The motivating example has two rules that generate the following state variables in SMV.

[2] In SMV we must specify a range with a lower and an upper limit

```
rankBonus : boolean;
bonusRank : boolean;
```

Step 1c: Define an SMV variable active to indicate which rule is currently active. The variable active is an enumerated type (known as scalar in SMV). Corresponding to each rule in the application there is an enumerator. Depending on which rule is active, the variable active takes on the corresponding value. There is also another enumerator, n, that is used to denote that no rules are active in a particular state.

State variable to represent current active rule: For our example the variable active can take the value bR (indicating that the rule **bonusRank** has been chosen for activation), rB (indicating that the rule **rankBonus** has been chosen for activation), or n (indicating that no rules have been chosen for activation).

```
active : {bR, rB, n};
```

Step 1d: Define an SMV variable input to indicate which input transaction is currently being processed. The variable input is also an enumerated type. For each input transaction, there is a corresponding enumerator. There is further an enumerator, n, that is used to denote that no input transaction is currently being processed.

State variable to represent current active transaction: The motivating example has two transactions **updateRank** and **updateAmt**. Thus the variable input can take in any of the three values uR (indicating that the current transaction being processed is **updateRank**), uA (indicating that the current transaction being processed is **updateAmt**) and n (indicating that no input transaction is currently being processed).

```
input : { uR, uA, n};
```

Step 2: State Initialization

Step 2a: For each SMV variable derived from the database table, specify the appropriate initial values using the *init* statement. The initial values depend on the application semantics and cannot be generalized.

Initializing state variables corresponding to the table: For our motivating example, the initial values are as follows:

```
init(id) := 0;
init(rank) := 0;
init(salary) := 0;
init(oldid) := 0;
init(oldrank) := 0;
init(oldsalary) := 0;
init(empid) := 0;
init(amount):= 0;
init(oldempid) := 0;
init(oldamount) := 0;
```

Step 2b: The variables corresponding to the rules are initialized to 0. This is because initially no rules are triggered.

Initializing state variables corresponding to the rules: For our example this is done as follows:

```
init(rankBonus) := 0;
init(bonusRank) := 0;
```

Step 2c: The variables active and input are each initialized to n – indicating that none of the rules are active and no input transaction is being processed.

Initializing the variables active *and* input*:* For our example this is implemented as follows:

```
init(active) := n;
init(input) := n;
```

Step 3: State Transformation

Step 3a: The SMV variables that contain previous states values are changed as follows: the next state values of these variables equal the current values of the corresponding variables.

State transformation of variables containing previous state values: For our example this is

```
next(oldid) := id;
next(oldrank) := rank;
next(oldsalary) := salary;
next(oldempid) := empid;
next(oldamount) := amount;
```

Step 3b: The SMV variables containing current values are modified as per application semantics. Typically, in a database application, there are transactions and/or rules that update the corresponding cells in the table. A *case* statement is used to model this. The number of expressions in the *case* statement is the number of transactions and rules that update this cell plus one for the default case.

State transformation of variables containing current state values: For our example, rank and amount are the only variables that are changed by triggers or transactions. Consider the state transformation of the variable rank given below:

```
next(rank) :=
  case
   (input = uR) : rank + 1;
   (active = bR) : rank + 1;
   1 : rank;
  esac;
```

The first expression in the *case* statement indicates that rank gets updated if the current input transaction is **updateRank**. The second expression indicates that rank gets updated if the current active rule is **bonusRank**. The last expression is the default case – if none of the above two cases are true, then rank remains unchanged.

Similarly, the variable amount's state transformation is given by the following SMV code:

```
next(amount):=
  case
    (input = uA) : amount + 10;
    (active = rB) : amount + 10;
    1 : amount;
  esac;
```

Step 3c: The rule variables are set if the corresponding rules get triggered. Note that, a rule gets triggered if some event has occured and some condition is satisfied. The event can be either an insert, update or delete operation on a table. The condition may depend on the value of some attributes. The state variables representing the rules are modified in the following way. There is *case* statement describing the different conditions in which the rule gets triggered. The first expression contains the conjunction of the events and the conditions that cause the rule to be triggered. The second expression says that if the rule is triggered, but has not been processed, then the rule remains triggered. The third expression gives the default case which means the rule variable is set to 0; this indicates that the rule will not be triggered if the above conditions are not satisfied.

State transformation of rule variables: For the rule **rankBonus** the SMV specifications is:

```
next(rankBonus) :=
 case
  !(rank = oldrank) : 1;
  (rankBonus = 1) & !(active = rB) : 1;
  1 : 0;
 esac;
```

The first expression says that the rule **rankBonus** is triggered if the value of rank changes. The second expression says that if **rankBonus** was triggered but not activated, it will be activated in the next state. Finally, the default case says that if none of the above conditions is true then the rule is not triggered.
Similarly the SMV specification for the rule **bonusRank** is:

```
next(bonusRank) :=
 case
  !(amount = oldamount) : 1;
  (bonusRank = 1) & !(active = bR) : 1;
  1 : 0;
 esac;
```

Step 3d: The variable active gets changed in the following way. A *case* statement is used to specify how the value of active changes. For each rule variable there are two expressions. The first expression indicates that if a rule variable is true but the rule has not been activated, then active gets the value of that rule. The second expression says that if the event and the condition of a rule are both satisfied, the rule must be activated.

Note the order in which the expressions corresponding to the different rule variables are specified in the *case* statement, determine the priority of rule execution. This is because of the property that if the conditions for multiple expressions in a *case* statement are satisfied, only the first expression is executed.

State transformation of variable active*:* The first expression in the *case* statement says that if the current active trigger is not **bonusRank** but it is queued up for activation, then in the next state it will become active. The second expression makes a similar argument about the trigger **rankBonus**. The third expression says that if the **amount** has been changed then in the next the rule **bonusRank** will be activated. The fourth expression says that if the **rank** has been updated, then the rule **rankBonus** will be activated. If none of the above conditions are true, then no rule will be activated in the next state.

```
next(active) :=
  case
    !(active = bR) & (bonusRank = 1) : bR;
    !(active = rB) & (rankBonus = 1) : rB;
    !(amount = oldamt) : bR;
    !(rank = oldrank) : rB;
    1 : n;
  esac;
```

Step 3e: The variable input determines which transaction is being processed. We assume that once an input transaction is being processed or there is a trigger activated, no other new transaction is accepted because the processing is not yet complete. Otherwise the input can take any value from its possible enumerators. The value that input takes in this case is specified non-deterministically.

State transformation of variable input*:* For our example, the SMV specification is:

```
next(input) :=
 case
   !(input = n) : n;
   !(active = n) : n;
   (input = n) & (active = n) : {uR, uA, n};
 esac;
```

Converting Database Application to a Verifiable SMV Specification The SMV model checker checks all possible states for the satisfaction of the property. Hence, to avoid the state explosion problem it is required that the number of states be kept to a minimum level. In this section we discuss some optimizations to reduce the number of states. But before we talk about optimization, we need to elaborate on some details left out earlier.

Specify Upper Limits: In some applications no upper bound is specified for an attribute. However, in the SMV specification both the upper and the lower bound must be specified for a particular variable. To simulate the case of a variable not

having an upper bound, we use the modulo operation and wrap around when the variable reaches the upper limit.

Specifying upper limits for the example: In our example application no upper limit has been specified for the rank attribute. However, the corresponding SMV variable is specified with an upper bound shown below.

```
rank : 0..4;
```

Consider the state transformation of the variable rank as discussed in Step 3b.

```
next(rank) :=
  case
   (input = uR) : rank + 1;
   (active = bR) : rank + 1;
   1 : rank;
  esac;
```

With the above example, the rank will soon reach an upper bound. To avoid this scenario, we use the modulo operation shown below:

```
next(rank) :=
  case
   (input = uR) : (rank + 1) mod 5;
   (active = bR) : (rank + 1) mod 5;
   1 : rank;
  esac;
```

Note that the modeling above does not distinguish between ranks 1, 6, 11 etc. So if the actual value of the rank is important for some application then some extra measures must be taken. For example, suppose the application requires that if the rank is 1 then an employee gets an extra bonus. To model this we need to introduce another variable rankIsOne which equals 1, if the rank is one, and 0 otherwise.

Eliminate Redundant Variables: In **Step 1** we mentioned that for each field we create two variables. Now some of the fields are never changed in an application and they are not important with respect to the property being verified. Such variables can be safely eliminated from the specification. For instance, in the motivating example, the variables id, empid, salary, are never used in the specification. So these can safely be eliminated. Some fields in a table are never updated. For these fields we do not require a variable to store the previous state values. For example, id, empid never get updated – so we can do without the variables oldid, oldempid.

Reduce the Range of Variables: Sometimes a variable can take a wide range of values. For example, the salary of an employee can take any value in the range 0 to 200000. However, specifying salary as an integer with the above range will lead to a state explosion problem.

One solution is to just list a few possible values that the salary can take as in

```
salary : {40000, 45000, 50000, 55000, 60000}
```

This solution is fine for applications in which the range is 40000 to 60000 and increments occur in 5000. However, if the application requires that an increment of 1000 be given to an employee then this cannot be modeled if we specify the salary as above.
A second solution may be to just scale down the salary:

```
salary : 0..20
```

This solution assumes that the unit is in thousands. If this solution is used, then care must be taken to divide all the usage of salary figures by 1000. This solution is used to scale down the value of amount variable in our motivating example. Note that other solutions may also be possible. The solution which must be used depends on the application.

The complete specification that is input to the SMV model checker is provided in Appendix 1.

4.4 Specifying Non-termination Properties using Computational Tree Logic

Once we have built the finite model, the next step is to specify the non-termination property using Computational Tree Logic (CTL) formulae. To show non-termination, we have to show that in the absence of input transactions, the triggers will all be reset, that is, no trigger will be active.

The CTL formula for the termination property is as follows:

```
SPEC AG((input = n)  -> AF((active = n)))
```

The above CTL formula states that it is always the case that in the absence of any input transactions, eventually none of the triggers are active. Note that all properties stated in CTL begin with the word `SPEC`. The formula `AG(f)` means that `f` holds in every state along every path. The formula `AF(f)` means that along every path there exists some future state in which `f` holds.

The output of the model checker is as follows:

```
-- specification AG (input = n  -> AF active = n)
-- as demonstrated by the following execution sequence
```

⋮

The CTL formulae are false; that is the rules do not terminate. The model checker also provides the scenario under which the rules do not terminate. The detailed output is given in Appendix B.

5 Example with Terminating Rules

To illustrate the advantage of our approach over [2], we create a second example by slightly modifying the rule **rankBonus** of the example in Section 4.2. The modified rule, which we call **rankBonus2**, is given below. As before the **rankBonus2** trigger is set when rank is updated. However, this trigger updates the amount for the employee whose **rank** is less than 3.

create rule **rankBonus2** on **emp**
when updated(**rank**)
  then update **bonus**
    set **amount** = **amount** + 10
      where **emp-id** in (select **id** from **new-updated**
            and **new-updated.rank** $<$ 3)

The finite state model for this application is developed in the same manner as in the previous example. The model is very similar except that we introduce a boolean state variable rankLessThan3. The value of rankLessThan3 equals 1 when rank is less than 3 and 0 otherwise. Initially rankLessThan3 is 1. Whenever the current rank is greater than or equal to 2 and the current input transaction is **updateRank** or the current active trigger is **bonusRank**, rankLessThan3 is set to 0. Otherwise, the value of rankLessThan3 is not changed.

The SMV specification for the state initialization and state transformation of rankLessThan3 are given below:

```
init(rankLessThan3) := 1;
 next(rankLessThan3) :=
  case
   (input = uR) & (rank >= 2) : 0;
   (active = bR) & (rank >= 2) : 0 ;
   1 : rankLessThan3 ;
  esac;
```

The **rankBonus2** trigger is set when rank is updated. This trigger updates the amount for the employee whose rankLessThan3 is true. The SMV specification for the state initialization and state transformation of rankBonus2 is identical to that of rankBonus of the previous example. This is expected because the event and the condition part of the triggers are the same. The action parts of the two triggers are different and this is reflected in the state transformation of variable amount.

```
init(amount) := 0;
 next(amount) :=
  case
    (input = uA) : (amount + 1) mod 5;
    (active = rB) & rankLessThan3 : (amount + 1) mod 5;
    1 : amount;
  esac;
```

The CTL formulae remains the same:

```
SPEC AG((input = n)  -> AF((active = n)))
```

For this particular example, the rules **bonusRank** and **rankBonus2** are terminating. The triggering graph analysis technique [2] detects a cycle signifying a potential source of non-termination. However, when we execute the SMV program corresponding to this example, the CTL formulae return true indicating that the triggers do indeed terminate.

```
-- specification AG (input = n -> AF active = n) is true
```

The reason why the model checker is able to answer in the positive is that it performs an exhaustive search on the entire state space for the finite state model.

6 Conclusion

Our contribution in this paper is that we illustrate how state-of-the-art formal method tools can be used to reason about active database applications. In this paper we focus on how termination of active database rules can be formally verified by the database programmer using an easy-to-use symbolic model checker. The formal analysis provides assurance that the resulting system indeed possesses the termination property. Using a very simple example, we illustrate that our approach can detect termination for a case where the triggering graph approach of Aiken et al. [2] fails.

The major difficulty in this approach is developing an accurate finite state model of the active database application. The modeling should be such that the property being verified should not be altered in the process of developing the finite state abstraction. In this paper we illustrate manually how the model can be developed. A future work remains how some automated verification methodology can be used to create a model that can be automatically checked by the model-checker. A similar idea, in the context of automated verification of communication protocols, was proposed by Havelund and Shankar [10]. In this work the authors show how theorem proving can be used to generate a finite state abstraction of a communication protocol, which, in turn, can be verified by model checking. We plan to explore, if and how, such ideas can be used to verify active database applications.

References

1. R. Agrawal and N. Gehani. Ode (Object Database and Environment): The Language and the Data Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 36–45, Portland, OR, May 1989.
2. 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(1):3–41, March 1995.
3. J. Bailey, L. Crnogorac, K. Ramamohanarao, and H. Søndergaard. Abstract Interpretation of Active Rules and its use in Termination Analysis. In *Proceedings of the 6th International Conference on Database Theory*, pages 188–202, Delphi, Greece, January 1997.
4. J. Bailey, G. Dong, and K. Ramamohanarao. Decidability and Undecidability Results for the Termination Problem of Active Database Rules . In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 264–273, Seattle, Washington, June 1998.
5. E. Baralis, S. Ceri, and S. Parboschi. Improved Rule Analysis by means of Triggering and Activation Graphs. In T. Sellis, editor, *Rules in Database Systems*, volume 985 of *Lecture Notes in Computer Science*, pages 165–181. Springer-Verlag, 1995.
6. E. Baralis and J. Widom. An Algebraic Approach to Rule Analysis in Expert Database Systems. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 475–486, Santiago, Chile, September 1994.

7. K. P. Eswaran. Specification, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System. Technical Report IBM Research Report RJ 1820, IBM San Jose Research Laboratory, August 1976.
8. N. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proceedings of the 17th International Conference on Very Large Databases*, pages 327–336, Barcelona, Spain, September 1991.
9. L. M. Haas et al. Starburst Mid-flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.
10. K. Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In *Proceedings of the 3rd International Symposium of Formal Methods Europe*, volume 1, pages 662–681, Oxford, U.K., March 1996. Springer-Verlag.
11. D. Jackson. Niptick: A Checkable Specification Language. In *Proceedings of the Workshop on Formal Methods in Software Practice*, San Diego, CA, January 1996.
12. A. P. Karadimce and S. D. Urban. Refined Triggering Graphs: A Logic-Based Approach to Termination Analysis in an Active Object-Oriented Database. In *Proceedings of the 12th International Conference on Data Engineering*, pages 384–391, New Orleans, LA, February 1996.
13. L. van der Voort and A. Siebes. Termination and Confluence of Rule Execution. In *Proceedings of the 2nd International Conference on Information and Knowledge Management*, pages 245–255, Washington, DC, November 1993.
14. K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.
15. E. Simon and A. Kotz-Dittrich. Promises and Realities of Active Database Systems. In *Proceedings of the 21st International Conference on Very Large Databases*, pages 642–653, Zürich, Switzerland, September 1995.
16. M. Stonebraker, L. A. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Transaction on Knowledge and Data Engineering*, 2(1):125–142, March 1990.
17. J. Widom and S. Ceri. *Active Database Systems. Triggers and Rules for Advanced database Processing*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.
18. J. Widom, R. Cochrane, and B. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. In *Proceedings of the 17th International Conference on Very Large Databases*, Barcelona, Spain, September 1991.
19. J. M. Wing and M. Vazari-Farahani. A Case Study in Model Checking Software Systems. *Science of Computer Programming*, 28:273–299, 1997.

Appendix 1: SMV Specifications for Example 1

```
MODULE main
VAR
rank : 0..4;
amount : 0..4;

oldrank : 0..4;
oldamount : 0..4;

bonusRank : boolean;
rankBonus : boolean;
```

```
active : {bR, rB, n};
input : {uR, uA, n};

ASSIGN

 init(oldrank) := 0;
 next(oldrank) := rank;

 init(oldamount) := 0;
 next(oldamount) := amount;

 init(rank) := 0;
 next(rank) :=
  case
   (input = uR) : (rank + 1) mod 5;
   (active = bR) : (rank + 1) mod 5;
   1 : rank ;
 esac;

 init(amount) := 0;
 next(amount) :=
  case
    (input = uA) : (amount + 1) mod 5;
    (active = rB)  : (amount + 1) mod 5;
    1 : amount;
  esac;

 init(active) := n;
 next(active) :=
  case
    !(active = bR) & (bonusRank = 1) : bR;
    !(active = rB) & (rankBonus = 1) : rB;
    !(amount = oldamount) : bR;
    !(rank = oldrank) : rB;
    1 : n;
  esac;

 init(rankBonus) := 0;
 next(rankBonus) :=
  case
    !(rank = oldrank)  : 1;
    (rankBonus = 1) & !(active = rB) : 1;
    1 : 0;
  esac;
```

```
 init(bonusRank) := 0;
 next(bonusRank) :=
  case
    !(amount = oldamount) : 1;
    (bonusRank = 1) & !(active = bR) : 1;
    1 : 0;
  esac;

 init(input) := n;
 next(input) :=
  case
    !(input = n) : n;
    !(active = n) : n;
    (input = n ) & (active = n) : {uR, uA, n};
  esac;


--Checking for Rule Termination
--Returns true if the rules will eventually terminate

 SPEC AG((input = n)  -> AF((active = n)))
```

Appendix 2: Output Produced by SMV Model Checker

```
-- specification AG (input = n -> AF active = n) is false
-- as demonstrated by the following execution sequence
```

```
state 1.1:
rank = 0
amount = 0
oldrank = 0
oldamount = 0
bonusRank = 0
rankBonus = 0
active = n
input = n

state 1.2:
input = uR

state 1.3:
rank = 1
input = n

state 1.4:
oldrank = 1
rankBonus = 1
active = rB
input = uR

state 1.5:
rank = 2
amount = 1
rankBonus = 0
active = n
input = n

state 1.6:
oldrank = 2
oldamount = 1
bonusRank = 1
rankBonus = 1
active = bR

-- loop starts here --
state 1.7:
rank = 3
bonusRank = 0
active = rB

state 1.8:
amount = 2
oldrank = 3

state 1.9:
amount = 3
oldamount = 2
bonusRank = 1
rankBonus = 0
active = bR

state 1.10:
rank = 4
oldamount = 3

state 1.11:
rank = 0
oldrank = 4
bonusRank = 0
rankBonus = 1
active = rB

state 1.12:
amount = 4
oldrank = 0

state 1.13:
amount = 0
oldamount = 4
bonusRank = 1
rankBonus = 0
active = bR

state 1.14:
rank = 1
oldamount = 0

state 1.15:
rank = 2
oldrank = 1
bonusRank = 0
rankBonus = 1
active = rB

state 1.16:
amount = 1
oldrank = 2
```

```
state 1.17:
amount = 2
oldamount = 1
bonusRank = 1
rankBonus = 0
active = bR

state 1.18:
rank = 3
oldamount = 2

state 1.19:
rank = 4
oldrank = 3
bonusRank = 0
rankBonus = 1
active = rB

state 1.20:
amount = 3
oldrank = 4

state 1.21:
amount = 4
oldamount = 3
bonusRank = 1
rankBonus = 0
active = bR

state 1.22:
rank = 0
oldamount = 4

state 1.23:
rank = 1
oldrank = 0
bonusRank = 0
rankBonus = 1
active = rB

state 1.24:
amount = 0
oldrank = 1

state 1.25:
amount = 1
oldamount = 0
bonusRank = 1
rankBonus = 0
active = bR

state 1.26:
rank = 2
oldamount = 1

state 1.27:
rank = 3
oldrank = 2
bonusRank = 0
rankBonus = 1
active = rB


resources used:
user time: 0.09 s, system time: 0.01 s
BDD nodes allocated: 10029
Bytes allocated: 1048576
BDD nodes representing transition relation: 1314 + 1
```