An Abstraction-Based Analysis of Rule Systems for Active Database Management Systems

Tarek S. Ghazi and Michael Huth

Department of Computing and Information Sciences
Kansas State University, Manhattan, KS66506, USA
{ghazit,huth}@cis.ksu.edu

Abstract. An active database management system (ADBMS) augments a conventional DBMS with the capability to automatically react to stimuli occurring within and outside a database. This is usually achieved by incorporating a DBMS with a set of rules which determine the actions a DBMS should automatically execute when certain events and conditions arise. In this manner ADBMSs can be used to enforce and manage integrity constraints, provide security in databases, or act as alerters or triggers. However, one would like to be sure that the execution of some chain of rules is guaranteed to terminate; or that a set of rules interacts in a manner that is consistent with their intended semantics. We examine model checking and abstract interpretation as a possible framework for automatically analyzing and designing ADBMS rule systems. Our proposal provides the foundations for a tool that can be used to isolate properties of a given rule system's behavior. We implemented a prototype in the verification tool Spin by writing a GUI for rule system design and a Promela code generator for rule system analysis.

1 Motivation

With a conventional database management system (DBMS), insertions, deletions, and other manipulations of data are performed via user commands or application programs. This passive and potentially limiting characteristic of DBMSs is the primary motivation behind recent attempts to integrate database systems with sets of rules that define the circumstances when a DBMS may execute some actions automatically [19]. *Active database management systems* (ADBMSs) provide this additional capability by adding features allowing one to define rules that will be processed automatically when certain external or internal events, such as changes to the database state or a request from another resource, arise [12]. ADBMSs may thus automate enforcement and management of integrity constraints, database security, and alerting users of important transactions [1, 19].

As the popularity of active database management systems has grown, so has the demand for tools assisting ADBMS designers, users, and administrators who wish to analyze how a group of rules will interact [12]. Predicting the behavior of rules based on chosen simulation runs raises the same concerns which apply

www.manaraa.com

to verification of general software based on such a methodology [7]. We therefore propose to use the well established methodology of model checking [4, 18] in such a way that the design and formal verification of specified rule interaction constitutes an integrated and holistic activity. Model checking requires a description language within which one formulates a finite mathematical model, $\mathcal{M}$, of the computational situation at hand, a specification language in which one may write down the behavior, ϕ, sought to be satisfied by the system one models, and an efficiently executable semantics of the satisfaction relation $\mathcal{M} \models \phi$ between a model and its specification. The actual framework we chose is that of the description language Promela, its specification language/logic LTL [17], and the tool Spin with its property verifier [1] — this decision was obtained empirically as our implementation based on SMV [15] and CTL [8] did not perform well at all, due to the fact that SMV's case-statement, unlike Promela's if-statement, has a deterministic semantics.

Since events which trigger rules may depend on non-discrete or unbounded parameters, such as conditions on the concrete values of field entries in tables, all plausible faithful models of ADBMSs will have an infinite state space and, as such, are not analyzable with conventional formal verification tools. However, we will identify *abstractions* on the state space and its state transitions, drawing from the rich literature on model checking and abstraction (see e.g. [6, 10, 9]) and its conceptual origin in *abstract interpretation* [11]. This will allow safe approximations of vital system behavior: the termination of rule triggering, the determinacy of precedence orders between triggered rules, or the responsiveness between rule activations (e.g. Owicki's and Lamport's "leads-to" operator [16]) etc. While the research literature on model checking and abstraction mainly focuses on the mathematical foundations at the level of labeled transitions systems (e.g. the democratic Kripke structures in [9]), actual implementations, based on symbolic OBDD-based approaches [3, 15], realize abstractions directly by an abstract interpretation [11] of the symbolic execution semantics, or the operational semantics of conventional program code. This is also the approach we will take in this paper, for our main objective is a practical one of obtaining executable abstract models of ADBMSs' rule systems with the capacity of making safe judgments of rule interactions. Hence, we also need to provide a rationale for the safety enjoyed by our particular abstract interpretation. Similar practical concerns led us to animating such a framework in an existing tool for model checking, allowing for a fast prototyping and enabling us to make use of ample tool support, such as countertrace generations and other forms of diagnostics.

In Section 2 we describe the common functionality of most ADBMSs' rule systems as an intermediate language for rule specification whose syntax, presented informally, does not commit to a specific ADBMS, while at the same time allowing straightforward compilations of actual rule system designs into this language. In Section 3 we derive the formal semantics of our intermediate rule processing language by translating any rule system in a corresponding Promela program whose state transition system serves as the operational seman-

[1] `netlib.bell-labs.com/netlib/spin/whatispin.html`

www.manaraa.com

tics of the rule system. We explain this translation on our running example. This section also presents the approximating abstract interpretation of this semantics and discusses what sort of properties is will analyze safely. In Section 4 we sketch the architecture of our design and verification tool. In Section 5 we mention related work and Section 6 provides the conclusions as well as an outlook on future directions of this line of work.

2 Intermediate language for rule design

There is a firm consensus about the functional components of an ADBMS, justifying the use of an intermediate language for rule design into which actual rule systems may be compiled. We explain the necessary ADBMS functionality — as well as our overall methodology — by means of a running example presented in a Starburst-like format [22]. We will consider this format as our informal intermediate language for rule design. Our example ADBMS stores employee and salary information for a fictitious corporation. We design a number of rules reflecting the organization's bonus and salary policy and wish to analyze our rule system in order to verify the consistency of its implementation with our design. Table 1 summarizes the structure of our database. It is important to note that the underlying DBMS of a rule system can be based on any type of data model. For example, HiPAC [14] builds its rule system over an object-oriented DBMS, but the Starburst ADBMS is an extension of the Starburst relational DBMS [22]. Although HiPAC and Starburst rely on different underlying data models, both provide the standard ADBMS functionality described in this section; the data model is treated as an abstract one.

emp table	table containing employee information
empid	unique employee id number
name	name of employee
salary	current salary of employee
rank	range from 1..10 of possible ranks (10 is the highest)
bonus table	table containing salary increase information
empid	unique employee id number
raiseamnt	dollar amount of next raise

Table 1. Definition of tables and fields in our corporate database.

According to [12], an ADBMS must provide a mechanism for defining and managing event-condition-action (ECA) rules with syntax:

ON [*Event*] **IF** [*Condition*] **DO** [*Action*].

Multiple events may occur for a given event type [12] (e.g. insertions and deletions for event type "data modification"). The event associated with a rule characterizes the circumstances under which the rule is initially *signaled* or *triggered*

المنارة للاستشارات
www.manaraa.com

[12, 14]. Starburst recognizes data modification operations as possible rule triggering events [1, 5, 22, 23]. We confine our study to data modifications as event types; events such as "Fri Oct 9 15:22:13 CDT 1998" pose a separate challenge to abstraction techniques. Table 2 defines four rules and their respective triggering events, conditions, and actions. For example, whenever a user initiates an update to the rank field in the **emp** table, rule r_1 will be triggered. Observe that such events are already abstractions as they leave the employee's identity implicit. An ECA rule's condition determines whether or not a triggered rule is actually activated. Formally, a rule condition is a boolean expression whose truth value is dependent on and determined by the state of the underlying database [12]. For example, Table 2 shows that r_1 is activated whenever an employee's rank is updated, and that employee's *new* rank is less than 5. The condition for r_2, on the other hand, always evaluates to TRUE. This simply means that whenever an update on **bonus**(rank) occurs, the rule is immediately activated. Such rules are commonly referred to as event-action (EA) rules and are often used in practice [12, 20]. Since performed actions may cause events to arise or disappear, we can capture the execution semantics of ADBMSs as a state transition system. We will define this state transition system by the operational semantics of the Promela program generated by the static information of a given rule system.

An action may consist of a data modification operation, a data retrieval operation, a transaction operation (ie. COMMIT/ABORT), or a call to external procedures or methods [12]. Starburst allows all of these to be defined as possible actions [1]. However, in our examples, we limit ourselves to using only data modification and data retrieval actions; modeling events and actions which interface with modules which are external to the underlying DBMS requires additional machinery. The execution model of an ADBMS's rule system provides a semantics for rule triggering and activation [12, 21, 23] and is the guiding principle in generating a Promela model for such systems. In general, the execution model varies widely among different rule systems [21]. But even though Starburst implements its own rule definition language, it still follows the generic ECA paradigm of our intermediate language. Thus, it is possible to uniformly model and verify various execution models as long as they rest on the ECA format. In general, any ADBMS must have [12]: an underlying DBMS, a facility to define a set of rules, an event detector, a condition evaluator, and an action processor. Only the organization of these components varies over different rule systems, making our framework applicable to a whole range of actual ADBMS systems.

Since events may trigger multiple rules at the same time, the execution model of an ADBMS requires an explicit *conflict resolution* policy [12] which regulates how an ADBMS chooses a single rule from a set of triggered ones [23]. A common approach is to introduce priorities for rule triggerings. See rule 4 in Table 2 for how we present such priorities in our running example. We should stress that conflict resolution *precedes* rule condition evaluation in the ECA paradigm. The semantics of a particular ADBMS execution trace is also dependent on an initial state which is composed of a data part, responsible for condition evaluation, and an event part, which in conjunction with the conflict resolution policy determines

المنارة للاستشارات
www.manaraa.com

which rule(s) are triggered. Such an initial state determines a, possibly non-deterministic, computation which we illustrate on our running example. The initial date state is sketched in Table 3.

Rule	ECA definition
r_1	**ON** update to **emp**(rank) **IF** new **emp**(rank) < 5 **THEN** update **bonus**(raiseamnt)
r_2	**ON** update to **bonus**(raiseamnt) **IF** TRUE **THEN** update **emp**(salary)
r_3	**ON** update to **emp**(salary) **IF** **emp**(salary) $>$ \$50000 **THEN** retrieve **emp**(empid,name,salary,rank)
r_4	**ON** update to **emp**(rank) **IF** TRUE **THEN** retrieve **emp**(empid,name,salary,rank) **PRECEDES** $\{r_1\}$

Table 2. Our complete rule system.

empid	name	salary	rank	empid	raiseamnt
1	Matt Shirley	50000	3	1	1000
2	Jasmine Reick	65000	4	2	1500
3	Darren King	45000	2	3	500

Table 3. emp and **bonus** table.

Now, suppose Matt and Jasmine receive promotions and need to have their ranks in their company increased by one; this represents the initial event part. This update on **emp**(rank) produces an event triggering rules r_1 and r_4. Starburst adds these rules to the *consideration set* $\mathcal{R}_c$ which initially is empty. Thus, $\mathcal{R}_c = \{r_1, r_4\}$. If $\mathcal{R}_c$ contains *multiple* elements, the ADBMS applies its conflict resolution policy for choosing a rule from $\mathcal{R}_c$; so Starburst selects r_4 for consideration and removes it from $\mathcal{R}_c$. Since r_4's condition holds vacuously, the system proceeds to execute the action, which was defined as a data retrieval transaction to display the current values of the tuples being updated. This action does *not* trigger any new rules. Although new rules were not triggered $\mathcal{R}_c$ still contains r_1 as a sole rule. Thus, r_1 is chosen for consideration; its condition also evaluates to TRUE, but for Matt only: r_1's action increases Matt's raiseamnt (in the **bonus** table) by, say, \$500, but leaves Jasmine's raiseamnt unchanged (since her rank equals 5). Furthermore, execution of this action triggers the rule r_2 (see Table 4). In this manner r_2 causes Matt's and Jasmine's

المنارة للاستشارات
www.manaraa.com

salaries in the **emp** table to be increased by \$1,500, and also triggers r_3 which displays the composite result of our initial update to the terminal. Thus, Matt's and Jasmine's employee information will be displayed since their salaries are more than \$50,000. Thereafter, since $\mathcal{R}_c$ contains no more rules, rule processing terminates (see Table 4). The resulting values of the **emp** table are shown in Table 5. Although this particular system evolution seemed to work out just fine, our rule system does have a design flaw in it. Certainly, we would like to ensure that whenever an employee's rank is updated then that employee's salary is also updated. Using appropriate atomic propositional formula, one may write this as an LTL formula **G** (`update on emp rank` → `update on emp salary`) recalling that an LTL formula holds in a state if it holds for all computation paths beginning in that state. Our Promela program generated by this rule system specification indeed detects a countertrace to this invariant: we notice that an employee's salary only gets updated if their new rank is less than five or if their rank is updated at the same time as *another* employee's whose new rank is less than five. Thus, in our example run above Jasmine was lucky to have her rank updated at the same time as Matt's! In order to correct this problem, we need to modify our rule system by replacing r_2 with a new rule r_5 which updates an employee's salary whenever their rank is updated. Although formal verification is not essential for spotting this error in our toy example, such support is needed as soon as the number of rules and their degree of non-determinism increase to realistic sizes.

transaction	event	triggers	$\mathcal{R}_c$	considered rule	condition	action
initial	update	$\{r_1, r_4\}$	$\{r_1, r_4\}$	r_4	TRUE	retrieve
r_4's action	N/A	N/A	$\{r_1\}$	N/A	N/A	N/A
N/A	N/A	N/A	$\{r_1\}$	r_1	TRUE	update
r_1's action	update	$\{r_2\}$	$\{r_2\}$	r_2	TRUE	update
r_2's action	update	$\{r_3\}$	$\{r_3\}$	r_3	TRUE	retrieve
r_3's action	N/A	N/A	$\{\}$	N/A	N/A	N/A

Table 4. Complete summary of events following initial transaction.

empid	name	salary	rank	empid	raiseamnt
1	Matt Shirley	*51500*	*4*	1	*1500*
2	Jasmine Reick	*66500*	*5*	2	1500
3	Darren King	45000	2	3	500

Table 5. emp and **bonus** tables after completion of rule processing.

المنارة للاستشارات
www.manaraa.com

3 Abstract interpretation of possible behavior

3.1 Operational semantics of rule systems

The static information present in the rule system of Table 2 can be translated into a non-deterministic program in Promela in a completely automatic way. This translation is driven by the informal description of the execution model which is being formalized by implementing it in Promela. The declarative part of this program

```
#define N 4 /*defines the number of rules in our model*/

mtype = {update,retrieve,emp,bonus,empid,name,salary,raiseamnt,rank,all};
/*symbolic constants representing our ADBMS's events, tables and fields*/

typedef ecarule {  /*record structure for storing rule information*/
mtype triggeredby;  /*the first three fields represent the event*/
mtype triggeredtab; /*which causes this rule to be triggered*/
mtype triggeredfie;
mtype actionexec;   /*the next three fields represent the action*/
mtype actiontab;    /*that occurs if this rule is executed*/
mtype actionfie;
bool  rtype         /*rtype is set to 0 if this is an EA rule*/
};                 /*rtype is set to 1 if this is an ECA rule*/
ecarule rules[N];  /*array for keeping our rule information*/
bool c[N]; /*boolean array representing our consideration set*/

mtype i_event=update; /*our initial rule triggering transaction*/
mtype i_table=emp;
mtype i_field=rank;

chan selected = [0] of {byte};
/*communication interface between ADBMS's environment and system*/
chan action = [0] of {mtype,mtype,mtype,bool};
/*communication interface between ADBMS's system and environment*/

bool done;
/*becomes true when consideration set is empty*/
/*this represents the termination of rule processing*/
int temp;
/*loop counter variable*/
```

declares a type `mtype` of all events, tables and fields occurring in our rule system, creates a record structure `eacrule` which stores the event (e.g. `[update, emp,rank]`) followed by the action resulting from the rule's execution (e.g. `[update,bonus,raiseamnt]`) and is concluded by a flag `rtype` indicating whether this is an EA or EAC rule. The array `rules` has that record type and stores the entire static rule information at a subsequent initialization point. The boolean array `c` models our consideration set $\mathcal{R}_c$, `c[i] == 1` representing the consideration of rule r_{i+1}. The next three lines declare the event part of the initial state:

المنارة للاستشارات
www.manaraa.com

the transaction which is responsible for setting off the active rule system. We use two channels `selected` and `action` as communication interfaces between two processes: process `environment`, the ADBMS's execution environment, and process `system`, the system responsible for the evaluation of conditions. The process `environment` simply has to determine the next triggered rule, if any, which it passes along as an index through the channel `selected`. The process `system` will then determine the selected rule's associated action and decide whether this action will be executed or not. It will report back this decision as well as the action components to process `environment` along the channel `action`; the environment process then uses that information in turn to determine the next selected rule. Finally, the boolean flag `done` will implement our termination check described below and `temp` is a counter.
The next portion of the program

```
init()
 {
    rules[0].triggeredby=update; /*assign rule 1's  values*/
    rules[0].triggeredtab=emp;
    rules[0].triggeredfie=rank;
    rules[0].actionexec=update;
    rules[0].actiontab=bonus;
    rules[0].actionfie=raiseamnt;
    rules[0].rtype=1;

    rules[1].triggeredby=update; /*assign rule 2's values*/
    rules[1].triggeredtab=bonus;
    rules[1].triggeredfie=raiseamnt;
    rules[1].actionexec=update;
    rules[1].actiontab=emp;
    rules[1].actionfie=salary;
    rules[1].rtype=0;

    rules[2].triggeredby=update; /*assign rule 3's values*/
    rules[2].triggeredtab=emp;
    rules[2].triggeredfie=salary;
    rules[2].actionexec=retrieve;
    rules[2].actiontab=emp;
    rules[2].actionfie=all;
    rules[2].rtype=1;

    rules[3].triggeredby=update; /*assign rule 4's values*/
    rules[3].triggeredtab=emp;
    rules[3].triggeredfie=rank;
    rules[3].actionexec=retrieve;
    rules[3].actiontab=emp;
    rules[3].actionfie=all;
    rules[3].rtype=0;
    temp=0;
```

المنارة للاستشارات
www.manaraa.com

```
do
:: (temp<N) ->  /*while (temp<4) do*/
  if
  ::  i_event==rules[temp].triggeredby
      && i_table==rules[temp].triggeredtab
      && i_field==rules[temp].triggeredfie  -> c[temp]=1;
                                              temp=temp+1

      /*if the initial transaction triggers rule i, add rule i to*/
      /*R_c by setting c[temp] to 1... increment temp*/
      /*the if structure is exited and control passes back to the do loop*/

  ::  else -> temp=temp+1

      /*otherwise, do not set c[temp] to 1, and simply increment temp*/
      /*the if structure is exited and control passes back to the do loop*/
  fi
:: (temp>=N) -> break
od;

done=0; /*done=0 indicates that R_c is not empty*/
        /*done will be set to 1 only when R_c becomes*/
        /*empty again, i.e. when rule processing terminates*/
```

is an initialization phase which assign to the array `rules` all the static information represented in our rule system of Table 2. Then it inspects each rule to see whether its event (represented by the sub-array `[triggeredby,triggeredtab, triggeredfie]`) matches the initial one; in that case `c[i]` is set to `1`. To check for the termination of rule execution we initialize `done` to `0` modeling that the consideration set $\mathcal{R}_c$ is non-empty; `done` will be set to `1` only if no more rules are under consideration. Promela allows us to annotate the assignment `done = 1` with the label `progress` to check whether this program point is ever reached, thereby implementing a termination check.
The program body runs the processes `environment` and `system` in a synchronous parallel composition:

```
atomic{run environment(); run system()}
}
```

The Promela code for the environment process

```
proctype environment()
   {
    mtype event,table,field;
    bool fire;

    /*choose a rule for consideration and send it to system()*/
 end:do
    :: if
```

المنارة للاستشارات
www.manaraa.com

```
    /*since rule 4 has priority over rule 1,*/
    /*rule 1 should only be selected if rule 4 is*/
    /*not in R_c*/
       :: (c[0]==1) && (c[3]==0)  -> c[0]=0;
                                     selected!0
       :: c[1]==1 -> c[1]=0;
                     selected!1
       :: c[2]==1 -> c[2]=0;
                     selected!2
       :: c[3]==1 -> c[3]=0;
                  selected!3
       :: else -> progress: done=1; break
       fi;

      /*after control pass to system(), the environment waits for*/
      /*the system to respond*/

       action?event,table,field,fire;

       /*determine which rules were triggered by the system's action*/
       /*this is the same routine as in the init() process*/
       temp=0;
       do
       :: (temp<=N) ->
         if
         :: event==rules[temp].triggeredby
            && table==rules[temp].triggeredtab
            && field==rules[temp].triggeredfie && fire==1  -> c[temp]=1;
                                                             temp=temp+1
         ::  else -> temp=temp+1;
         fi
       :: (temp>N) -> break
       od
     od
}
```

has local variables which bind the information sent by process `system`. The Promela if-statement — a case-statement in disguise — has an unusual semantics. If several case test patterns evaluate to `1` it will not merely execute the top-most case — as would be the case with most functional and imperative programming languages — but exhaustively explore the program behavior of all these cases. This was the operative reason for choosing Spin over the model checker SMV as its case-statement follows the conventional semantics, resulting in an aggravated state-explosion. Note how the case test patterns simply check whether `c[i]` equals `1`, in case that there are no overwriting priorities. If a rule is preceded by other rules, we simply code this as a conjunction of literals. For example, the test pattern `(c[0]==1) && (c[3]==0)` makes sure that rule 1 won't be selected if rule 4 is triggered as well. After `environment` obtained the infor-

www.manaraa.com

mation from channel `action`, it executes a loop which determines which rules are being triggered by this event and updates the boolean vector `c` accordingly. The system process

```
proctype system()
    {
      int rulenum;
      mtype event,table,field;

  end:do
         /*receive the selected rule from the environment*/
      :: selected?rulenum;
         /*determine the selected rule's associated action*/
         event = rules[rulenum].actionexec;
         table  = rules[rulenum].actiontab;
         field  = rules[rulenum].actionfie;
         if
         /*if an EA rule is under consideration always execute its action*/
         ::  rules[rulenum].rtype==0 -> action!event,table,field,1
         ::  rules[rulenum].rtype==1 ->
         /*if an ECA rule is under consideration, non-deterministically choose*/
         /*whether or not to execute its action*/
           if
           :: action!event,table,field,0
           :: action!event,table,field,1
           fi
         fi
      od
   }
}
```

has local variables for representing the action (`event,table,field`) and the selected rule provided by process `environment` (`rulenum`). This process simply determines the action associated with the selected rule. If this rule is an EA rule, this action will be executed, resulting in the flag `1` as the fourth value to be send to `environment`. Otherwise, the rule is a general ECA rule. We abstract the actual operational semantics of our intermediate language by a Promela if-statement which exhaustively explores the program behavior for both possibilities: sending the flag `0` along with the action to `environment` means that the action is not executed and rule selection resumes; sending the flag `1`, however, means that the environment executes this action, resulting is an updated consideration set etc.

3.2 Abstract interpretation and safe LTL model checks

Spin uses LTL formulas whose propositional atoms are boolean expressions referring to the state of a given Promela model or program points reached during

www.manaraa.com

its exeuction. For example, determining whether the consideration of one rule eventually leads to the consideration of another rule is expressed by the LTL formula `[](c[i]==1 -> <>(c[j]==1))` where $\texttt{i,j} < \texttt{N}$. The GUI for our rule system design offers a variety of such LTL formulas as options to be included into the generated Promela program.

Abstract interpretation is a methodology for program analysis unifying seemingly quite different modes of analysis (e.g. invariants, strictness analysis, convex polyhedra for the analysis of hybrid systems etc) under one conceptual umbrella. At the core, one considers state transition systems (S, R, I), where $R \subseteq S \times S$ is the state transition relation, and $I \subseteq S$ is the set of initial states. One then studies hierarchies of abstractions thereof: (S', R', I') is an abstraction of (S, R, I) if there exist monotone functions $\alpha: S \to S'$ and $\gamma: S' \to S$ with $\alpha \circ \gamma = \mathrm{id}$ and $\gamma \circ \alpha \geq \mathrm{id}$ such that I is contained in $\gamma(I')$ and such that for all $s' \in S'$ we have that $\mathrm{post}(\gamma(s'))$ is a subset of $\gamma(\mathrm{post}(s'))$. Here $\mathrm{post}(X)$ is the strongest postcondition of X, i.e. the set of all states which can be reached from X within one computation step. Note that our "concrete" semantics is already a collecting semantics [11] and as such already a form of abstraction. The abstraction induced by the if-statement for ECA rules simply enlarges the transition relation: we go from (S, R, I) to (S, R', I) for some $R \subseteq R'$ and have $\alpha = \gamma = \mathrm{id}$. Thus, any computation path of the system (S, R, I) beginning in a state in I is also a computation path in the abstracted system. So if the LTL formula `[](c[i]==1 -> <>(c[j]==1))` holds in state $\alpha(s)$, then we know that it must hold in s as well. Otherwise, there would be a computation path beginning in s which fails that property. Similar reasoning applies to our termination check and other properties we deemed to be important for rule analysis. Since all of these LTL formulas are also in the universal fragment of CTL*, such safety results are hardly surprising given the results in [7] for this temporal logic for model checking with abstraction.

4 Implementation architecture

In model checking rule systems of ADBMSs defined in our intermediate language, we chose to omit any representation of database states in our model. We already explained how this can be seen as a conventional abstract interpretation and that it justifies safe model checks of important specification patterns coded in LTL. The Promela code of our running example was automatically generated from the static information specified in a GUI for rule design. This simple GUI application, written in Java, allows a user to define, modify, interactively edit rule systems, and save them for future analysis. This platform also enables designers to automatically generate the Promela code represented by the rule system defined so far, and execute that model with a number of LTL specifications beginning in a user specified initial state. Sample specifications supported by this tool are deterministic ordering and rule integrity constraints. A termination check of rule firing is enforced implicitly by our implementation. Our application generates Starburst rule processing models based on our intermediate

المنارة للاستشارات
www.manaraa.com

language for rule design, but could be customized to generate Promela models reflecting the execution semantics of another paradigm. Figure 1 illustrates a high-level architectural description of this implementation. Spin's countertrace facility, which produces a graphical counter-example whenever a property is not satisfied, proved to be most useful for analyzing rule behavior. Of course, Spin will return only one possible countertrace so the designer has to read off conclusive information from that trace in order to debug his or her system.

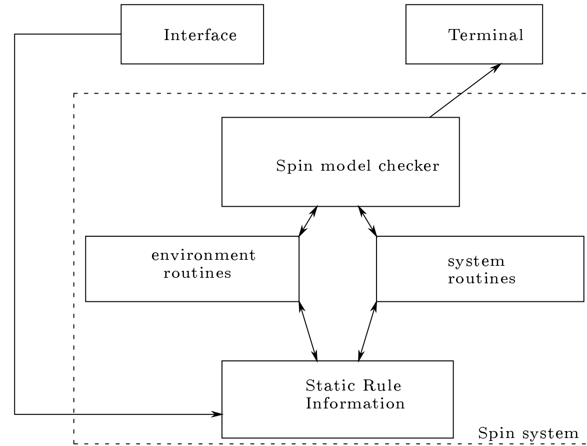

Fig. 1. Architecture of our platform for rule system design and analysis.

5 Related work

Previous efforts at analyzing rule systems can be found in [1, 5, 22, 23]. These efforts were directed specifically towards Starburst. In [2] one finds approximative algorithms for analyzing the following three properties: termination of a set of rules, confluence of a set of rules, and observably deterministic behavior of a set of rules. It is hardly surprising that attempts to analyze these properties have been somewhat inconclusive. In fact, these notions are in general undecidable [12].

6 Conclusion and outlook

We used our tool to generate models of rules processing for rule sets of various sizes and complexity. Then we analyzed properties such as termination, confluence, and rule integrity constraints. In analyzing termination, rule integrity constraints, and deterministic ordering properties for a set of twenty ECA rules,

المنارة للاستشارات
www.manaraa.com

we tested the accuracy of our model by seeding it with various violations to these properties. In all cases, Spin flagged these errors and provided a visual counter-example of where the property failed; since we compute with conditions *symbolically*, this is informative even though the data values are abstracted. The effect of additional non-determinism caused the state space to grow quite dramatically. In one case the size of the state space reached 215,000 states. This clearly indicates that our analysis is susceptible to the "state-explosion" problem [4]. Upon prioritizing rules the state space decreased considerably. In [1] it is suggested that large rule sets often can be "partitioned" into smaller, independent groups of rules. Since rules in one group could not affect the behavior of rules in another group, each partition could be analyzed separately, hence alleviating the size of the state space. We would like to add more detail to our Promela models in an effort to obtain more fine-grained rule analysis, e.g. by simulating rule *untriggering*. We are also interested in exploring techniques for state-space reduction using filter-based refinement [13], to "filter out" or eliminate unlikely or impossible computation paths. There are good reasons for having a *domain-specific language* (DSL) for formulating rule systems. The latter would allow the conventional tools of program analysis such as partial evaluation and (refined) abstract interpretation to operate on rule system specifications directly. We believe rule analysis in our framework can be effectively used in conjunction with the confluence requirement algorithm [1] by isolating certain rule pairs for which it is not even necessary to apply that algorithm, and providing a "tool box" of specifications for determining whether a pair of rules commute.

Other applications for this methodology are conceivable. For example, there are rule systems for the generation of database query optimizations using a rewrite system.

References

1. A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 59–68, San Diego, California, June 1992.
2. E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, Santiago, Chile, 1994.
3. R. R. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
4. J. R. Burch, E. M. Clarke, D. L. Dill K. L. McMillan, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. Proceedings of the Fifth Annual Symposium on Logic in Computer Science, June 1990.
5. S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 566–577, Brisbane, Australia, August 1990.
6. E. Clarke, O. Grumberg, and D. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

المنارة للاستشارات
www.manaraa.com

7. E. Clarke, J. M. Wing, and et al. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.
8. E. M. Clarke and E. M. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In D. Kozen, editor, *Proc. Logic of Programs*, volume 131 of *LNCS*. Springer Verlag, 1981.
9. R. Cleaveland, P. Iyer, and D. Yankelevich. Optimality in Abstractions of Model Checking. In A. Mycroft, editor, *Statis Analysis*, volume 983 of *Lecture Notes in Computer Science*, pages 51–63. Springer Verlag, September 1995. Glasgow, United Kingdom.
10. R. Cleaveland and J. Riely. Testing-Based Abstractions for Value-Passing Systems. In B. Jonsson and P. Parrow, editors, *Proceedings of Concur'94*, volume 836 of *Lecture Notes in Computer Science*, pages 417–432. Springer Verlag, August 1994. Uppsala, Sweden.
11. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
12. K. R. Dittrich, S. Gatzui, and A. Geppert. The active database management system manifesto: A rulebase of adbms features. In *Second International Workshop on Rules in Database Systems*, pages 3–20, Athens, Greece, September 1995.
13. M. B. Dwyer. Modular Flow Analysis for Concurrent Software. In *Proceedings of the 12th International Conference on Automated Software Engineering*, pages 264–273, November 1997.
14. D. R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215–224, Portland, Oregon, May 1989.
15. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
16. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. on Programming Languages and Systems*, 4(3):455–495, 1982.
17. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J.W. de Bakker, editor, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, 1985.
18. J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the fifth International Symposium on Programming*, 1981.
19. M. Stonebreaker. The integration of rule systems and database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):415–423, October 1992. Invited Paper.
20. M. Stonebreaker, editor. *Readings in Database Systems*, chapter 4, pages 345–349. Morgan Kaufman Publishers, 2 edition, 1994.
21. J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Data Processing*, chapter 4, pages 32–33,88–108. Morgan Kaufman Publishers, 1996.
22. J. Widom, R. J. Cochrance, and B. G. Lindsay. Implementing set-oriented production rules as an extension to starburst. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 275–285, Barcelona, Spain, September 1991.
23. J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, Atlantic City, New Jersey, May 1990.

المنارة للاستشارات
www.manaraa.com