A Framework for Model Checking 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

April 20, 1998

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. Intuitively, this requires a DBMS to possess some degree of "knowledge". Such knowledge can be provided by incorporating a DBMS with a set of rules which determine the actions a DBMS should automatically execute when certain events and conditions arise. The advantages of ADBMSs have been well documented. ADBMSs can be used to enforce and manage integrity constraints, provide security in databases, and act as alerters or triggers. However, it has also been noted that determining the effect of interaction between groups of rules is difficult at best. For example, 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 the intended semantics conceived by their designer. The purpose of this paper is to examine Model Checking as a possible framework for automatically analyzing ADBMSs. Our framework provides the basis for development of a tool that can be used to isolate properties of a given rule set's behavior. Such a verification tool is critical for supporting the pre-development analysis and design of realistic ADBMSs. We also implemented significant parts of our framework directly in the verification tool *Spin* by writing a GUI for rule design and a *Promela* code generator.

Contents

1 Introduction

1.1 Active Database Management Systems

With a conventional DBMS, insertions, deletions, and other manipulations of data are performed via user commands or application programs that modify a database. From this perspective, a DBMS is *passive*, since it only responds to commands that it receives explicitly [14]. This 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 [18]. An *active database management system* (ADBMS) provides this additional capability over a conventional, "passive" database by adding features allowing one to define rules that will be processed automatically when certain events, such as changes to the database state, arise [12]. Thus, issues such as enforcement and management of integrity constraints, database security, and alerting users when important events occur may be automated [1, 18].

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 behave [12]. Predicting how a group of rules will interact with one another in all situations, however, is difficult at best [20]. In spite of these facts, the supply of such tools has lagged far behind demand. In light of this, methods for analyzing behavior of rules under the Starburst rule system represent a major step in the right direction and provide a foundation for developing interactive tools that would aid ADBMS designers in predicting rule behavior [1, 8, 21, 22]. This makes Starburst an important benchmark against which new efforts at analyzing rule systems should be compared.

1.2 Model Checking

Model Checking is an approach to validating hardware and software, which has become popular within the past decade [7]. Under this approach, verification is not performed on the actual system. Instead, one builds an *abstract model* of the system that excludes many real features of the actual physical system, allowing one's focus to be directed towards only those features necessary to verify a particular property at hand [7]. Properties of such a system are written in some type of temporal logic [11, 15, 16, 17]. Ultimately, one is interested in verifying the correctness of these properties with respect to the abstract model, and then extrapolate these results to the actual system.

Model Checking offers some unique advantages over other popular approaches of verification. The most distinct advantage is automation [7, 11]. Model Checking relies on efficient algorithms that serve as the foundation for *model checkers*, which implement these algorithms [7, 11, 17]. Given an abstract model of a system and a property specified in some temporal or modal logic, a model checker automatically determines whether the property holds in that particular model [11]. Perhaps even more importantly, model checkers can be used to construct an execution trace that shows why a property *did not* hold with respect to a given model, facilitating future debugging of the system [10, 4].

1.3 Motivation

According to [16], computerized systems can be broken down into two distinct categories: transformational and reactive. In the transformational view, a

system receives input data, performs some operation on that data, and produces some output upon termination. This view implies that the abstract model of a transformational system consists of an initial state, some intermediate states, and a final state [16].

The computational paradigm of reactive systems, on the other hand, is not so intuitive. Reactive systems do not have principal beginning and ending states. As their name implies, reactive systems may respond to a variety of events at any time. Because of this, Pnueli suggests that modeling a reactive system requires enhancing the model by incorporating information about the system's interaction with its environment [16]. This is necessary if we wish to accurately reason about the system's behavior in response to a particular event. Thus, an event can be seen as a transition from one state to the next, and a series of events can be viewed as a possible execution path of the system. In order to reason about such sequences of events, Pnueli proposes the use of temporal logic to code specifications of a system.

Application of temporal logics to specify system properties is nothing new. Efficient model checkers for linear-time temporal logics (LTL) and branching-time temporal logics, such as CTL, already exist [7, 11, 15]. These model checkers make use of special algorithms and data structures that allow representations of abstract models to be stored compactly, and specification formulas to be analyzed automatically [11]. Unfortunately, both CTL and LTL can only be naturally applied to *closed systems*, which depend only on states of the system. *Open systems*, on the other hand, must be characterized by their behavior in a particular environment [2]. Any ADBMS is *reactive* since its functionality requires that it be able to respond automatically to certain events [12]. It may also be characterized as *open* since its behavior may change according to decisions made by its environment [20]. Thus, it would be difficult to *directly* apply CTL or LTL as a means of describing an ADBMS's behavior without explicitly including assumptions about an ADBMS's environment.

Therefore, the aim of this paper is to effectively develop a Model Checking methodology for the problem of verifying properties of active database management systems. We present a framework for constructing an abstract model of an ADBMS that captures the behavior of the ADBMS in the context of any decisions made by its environment. In order to empirically evaluate our methodology, we apply our framework toward analyzing a sample rule base, which will be constructed using an ADBMS similar to the Starburst rule system [1, 8, 21, 22]. However, one should realize that the true advan-

tage and power of our approach is its *flexibility*. Unlike current efforts at analyzing rule systems, our application of Model Checking can be used as a generic framework for *any* ADBMS regardless of its unique operational semantics. Furthermore, we offer Event-Action Temporal Logic (**EATL**), a customized branching-time temporal logic that can be used to specify properties we wish to verify under this model. Finally, we demonstrate how existing model checkers can be creatively applied towards analyzing ADBMSs. In particular, we formally analyze rule processing for the Starburst rule system using *Spin* [4], an LTL model checker, and its modeling language, *Promela*. We show how, given explicit information about Starburst's rule processing environment, *Promela* can be used to model different rule processing scenarios. Moreover, we have developed a simple, interactive GUI environment for generating *Promela* models of these rule processing scenarios. We believe the development of *different* "libraries", each generating models of unique ADBMSs could be consolidated under a single, user-friendly interface. Indeed, such a tool would make it feasible for any ADBMS designer, user, or administrator to effectively analyze rule behavior.

2 ADBMS Functionality

Similar to conventional DBMSs, there is a general consensus about what functional components an ADBMS should possess. These additional features of an ADBMS are considered the minimum essentials necessary to give a conventional DBMS an "active" functionality [12]. Starburst is no exception, and clearly reflects this consensus. Therefore, our approach to explaining necessary ADBMS functionality is by means of Starburst as an example. We begin with an ordinary DBMS and incrementally construct an ADBMS in the spirit of the Starburst rule system [1, 8, 21, 22]. At the end of this section, we present a simple functioning rule system, which offers a subset of Starburst's features. This simple system will serve as our running example in unfolding our proposed ADBMS Model Checking methodology.

The ADBMS we design is simple, but practical; it keeps a database storing employee and salary information for a fictitious corporation. We design a number of rules reflecting the organization's bonus and salary policy. Ultimately, we wish to modelcheck our rule system in order to verify the consistency of its implementation with our design. In Section 4, we discover that, although our sample ADBMS is small, subtle mistakes in rule design can still

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.

occur quite easily.

2.1 The underlying DBMS

As their name implies, an ADBMS must subsume a DBMS; it must provide all the functionality of a DBMS and can in fact be used as a conventional system by simply ignoring its active components [12]. 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 [21]. Although HiPAC and Starburst rely on different underlying data models, both provide the standard ADBMS functionality described in this section. The following example defines a simple relational DBMS scheme we use as part of our running example throughout this paper.

Example 1 Consider the following relational DBMS scheme, consisting of two tables.

emp(empid, name, salary, rank)
bonus(empid, raiseamnt).

Assume these tables make up the database of our rule system which maintains employee and salary information for some corporation. Table 1 summarizes the contents of the database (the primary key for each table is underlined). As suggested earlier, we wish to define some rules over this scheme.

2.2 Adding Rules to the DBMS

Given the relational database scheme in the previous example, we need a mechanism for defining rules over it. We intend to design a set of rules that automatically enforces our corporate's policy regarding employee salaries and bonuses. According to [12], an ADBMS must provide a mechanism for defining and managing *Event-Condition-Action* (ECA) rules. The ECA rule paradigm follows the syntax:

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

By examining each syntactic category (i.e. event, condition, action) of this statement in detail we naturally gain the required insights into its semantics.

2.2.1 Events

It is important to distinguish between *events* and *event types*. For instance, multiple events may occur for a given event type [12]. Table 2 depicts this idea. It is necessary to define an *event* for each rule in our rule base. After all, this attribute determines the circumstances under which the rule is initially *signaled* or *triggered* [12, 14]. The events we may choose from depend on the *event types* our rule system is able to recognize. Starburst recognizes data modification operations as possible rule triggering events [1, 8, 21, 22]. In the case of the three data modification operations seen in Table 2, most ADBMSs, including Starburst, allow us to augment the name of the database table and even the field(s) within that table to which the event (i.e. insert, delete, update) applies [1, 14]. The next example demonstrates this convention. We initially only define the triggering events for our rules.

Example 2 Assume that our ADBMS recognizes only *data modification* event types. Then our rule base may consist of rules triggered by an insertion, deletion, or update to the tables in the database. Table 3 summarizes our initial design sketch of three rules r_1, r_2, and r_3 and their respective triggering event.

In Table 3 we observe that whenever a user initiates an update to the rank field in the **emp** table, rule r_1 will be triggered. In a similar fashion, we may determine what triggers the other two rules. At this point, we have only determined what triggers each of our rules. In order to complete our design, we need to consider the conditions and actions of these ECA rules.

EVENT TYPE	EVENT
Data modification	insert, delete, update
Data retrieval	SQL select statement
Absolute time	12:00AM
Relative time	"4 hours after an update"
Periodic time	"each Tuesday at 8:00AM"

Table 2: Events and their event type category.

Rule	triggering event
r_1	**ON** update to **emp**(rank) ...
r_2	**ON** update to **bonus**(raiseamnt) ...
r_3	**ON** update to **emp**(salary) ...

Table 3: Three rules and the events that signal each.

2.2.2 Conditions

The event of a rule determines under which circumstances it is triggered. The next relevant portion of an ECA rule is its *condition*, which determines whether or not the rule is actually *activated*. Formally, a rule condition defines the state that a database must have in order for a rule to execute [12]. The condition may be expressed in two ways:

- as a predicate written in the DBMS query language, such as SQL, that returns either TRUE or FALSE, or
- as a database query that returns either EMPTY or NOT EMPTY.

Starburst follows the first approach [1, 8, 21, 22]. We continue the design of our rule base by specifying the conditions for the rules of Example 2.

Example 3 Table 4 defines what state our simple database must have in order for each rule to be activated (i.e. it defines r_i's condition).

Examination of Table 3 in conjunction with Table 4 shows that r_1 is activated whenever an employee's rank is updated, and that employee's *new* rank is less than 5. We may reason about rule r_3 in a similar manner. 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 [12, 22]. Such rules, where the rule's condition always evaluates to

Rule	activating condition
r_1	... **IF** new **emp**(rank)< 5 ...
r_2	... **IF** TRUE ...
r_3	... **IF** **emp**(salary)$> \$50,000$ ...

Table 4: Three rules and the conditions required for their activation.

TRUE, are commonly referred to as *event-action* (EA) rules which are often used in practice [12, 19]. Therefore, many ADBMSs, including Starburst, allow a rule designer to optionally omit a rule's condition [21, 22].

2.2.3 Actions

The *action* of a rule is executed as a *reaction* to any triggering of a rule's event, when a rule's condition holds [1, 8, 12, 14, 21, 22]. Such actions essentially characterize the "reactive" behavior of an ADBMS discussed in Subsection 1.3 [12]. Naturally, this is also what causes ADBMS behavior to be so difficult to predict [1, 21]. 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.

Recall from Example 2 that the first action type (data modification) also constitutes an event type in Starburst. This means that the action of one rule may automatically cause an event that triggers more rules. This *cascaded triggering* of rules makes the prediction of rule behavior difficult as soon as the number of rules in the system grows large and raises serious challenges when attempting to design a sensible set of rules [1].

Example 4 We complete our set of ECA rules by specifying what action should be taken when each rule is activated. Table 5 lists our rules and their associated actions.

Table 6 allows a cascaded triggering of rules $r_1 \rightarrow r_2 \rightarrow r_3$ as a chain reaction initiated by r_1's action triggering r_2, and r_2's action, in turn, triggering r_3. In our example, the chain of events terminates at r_3, since we do *not* consider r_3's action (i.e. data retrieval) to be a triggering event. However, such a small number of rules is uncommon and hides the unpredictability of

rule	action performed
r_1	... **THEN** update **bonus**(raiseamnt)
r_2	... **THEN** update **emp**(salary)
r_3	... **THEN** retrieve **emp**(empid,name,salary,rank)

Table 5: Three rules and the actions they perform if activated.

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) $> \$50,000$ **THEN** retrieve **emp**(empid,name,salary,rank)

Table 6: Our complete set, $\mathcal{R}$, of rules.

such rule systems present in realistic examples. For more substantial cases it will be next to impossible for a designer to construct an unambiguous set of rules [1] without any proper tool support. The next example demonstrates one possible instance of what could go wrong if we are not careful when designing rules.

Example 5 Suppose that we temporarily add two more rules to our current set in Table 7.

Clearly, an event triggering r_4 or r_5 would result in a cyclic execution of

Rule	ECA definition
r_4	**ON** update to **emp**(salary) **IF** TRUE **THEN** update **emp**(rank)
r_5	**ON** update to **emp**(rank) **IF** TRUE **THEN** update **emp**(salary)

Table 7: A non-terminating set of rules.

these two rules that would never terminate. In fact, these two rules would cause an employee's salary to increase indefinitely! We can easily predict this adverse behavior in our simple examples, but such design flaws may remain undetected in a larger rule set with longer cycles.

2.3 Definition of ECA rules

The syntax of ECA rules provides a generic framework for *logically* designing a rule base and reasoning about the behavior of the rules therein. Table 6, for instance, reflects our corporation's policy to automatically give increased raises to promoted employees until they achieve a rank of five. From that point on, an employee should receive the same raise upon being promoted. The definition of new ECA rules must be possible either through the ADBMS's data definition language (DDL) or by adopting a new rule definition language (RDL) [12]. HiPAC and Starburst adopt such RDLs, and, although the syntax for defining new rules depends on the particular system being used, both languages provide constructs for defining various events, conditions, and actions [1, 14, 21, 22]. We adopt the notation of Starburst's RDL [1, 21, 22]. The following is a fragment of the syntax for rule definition in Starburst:

create rule *name*
when *event*
[**if** *condition*]
then *action*

Rule definition in Starburst closely matches the general format of ECA rules. However, note that a rule's condition is optional. This optional facility allows Starburst to define EA rules. In our example, r_2 has no condition (i.e. it is an EA rule), and, therefore, the condition is coded as "IF TRUE". Figures 1, 2, and 3 show how to express our rules using Starburst's syntax. Reader's familiar with SQL certainly notice the SQL-like characteristics of this statement. In fact, the conditions themselves are simply SQL predicates [22]. For those unfamiliar with SQL, Table 8 describes in detail the meaning of each statement involved in the insertion of r_1 (from Figure 1). Commands to insert the other two rules are similar and are given in Figures 2 and 3. An informal explanation of these insertions is not provided, but may be obtained by comparison with rule_1's explanation. Please observe that rule_2 has no

```
create rule rule_1 on emp
when updated(rank)
if select empid e1 from new updated emp.rank
    where

          (select rank from new updated
              emp.rank where
          new updated emp.rank > old up-
              dated emp.rank
          and new updated emp.rank < 5)

    then update bonus

          set raiseamnt = raiseamnt + 500
          where empid = e1.empid
```

Figure 1: Command to insert rule_1 into Starburst rule base.

condition since it is an EA rule. Also note that, in rule_3, the SQL query, "select * ...," retrieves *all* fields from the specified table.

2.4 Execution Models

The final necessary ingredient of any active database management system is its execution model [12]. For any ADBMS, its execution model describes the manner in which rules are processed [12, 20, 22]. In general, this model varies widely among different rule systems [20]. On the surface, this may seem problematic for our proposed methodology of model checking ABDMSs. Yet, recall that, even though Starburst implements its own language for defining rules (its RDL), it still follows the generic ECA paradigm. Thus, it is possible to isolate similarities among various execution models as long as they rest on the ECA paradigm. We use Starburst as our illustratory case study of an actual execution model.

2.4.1 Physical ADBMS Components

The most obvious resemblance among different execution models are the physical components required for rule processing. Any ADBMS must have

STATEMENT	MEANING
create ...	says rule_1 effects **emp** table
when ...	says this rule is signaled on an update of the rank field
new updated emp.rank	refers to the **emp** table *after* the update
old updated emp.rank	refers to the **emp** table *before* the update
if select ...	instruction to find all the *updated* tuples whose *new* rank is less than 5 and, if the result is *not empty*, the condition holds
then ...	says that if previous condition holds, then update the **bonus** table
set ... where	says that an update adds 500 to the raiseamnt of those tuples *for which the previous condition* holds

Table 8: Informal explanation of rule_1 definition.

```
create rule rule_2 on bonus
when updated(raiseamnt)
then update emp

        set salary = salary + raiseamnt
        where empid in (select empid from

                new updated emp.rank, old updated emp.rank
                where new updated emp.rank >
                old updated emp.rank)
```

Figure 2: Command to insert rule_2 into Starburst rule base.

```
create rule rule_3 on emp
when updated(salary)
if exists (select * from new updated emp.salary

        where new updated emp.salary > old updated emp.salary and
        new updated emp.salary > 50000 )

  then

        select * from new updated emp.salary where
        new updated emp.salary > old updated emp.salary and
        new updated emp.salary > 50000
```

Figure 3: Command to insert rule_3 into Starburst rule base.

[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.

2.4.2 Conflict Resolution

In addition to physical components, any ADBMS's execution model requires an explicit *conflict resolution* policy [12]. Since the occurrence of an event may trigger multiple rules, this policy describes how an ADBMS chooses which rule should be processed first [22]. There are three predominant strategies for conflict resolution [22]: no ordering (i.e. non-deterministic selection), partial ordering, and total ordering (i.e. deterministic selection). Partial and total ordering can be enforced by assigning priorities to rules at the time of rule creation. The full syntax for the creation of ECA rules in Starburst allows such enforcements and is given by:

create rule *name*
when *event*
[**if** *condition*]
then *action*
[**precedes** *rule-list*]
[**follows** *rule-list*]

```
create rule rule_1 on emp
when updated(rank)
if select empid e1 from new updated emp.rank
    where

          (select rank from new updated
              emp.rank where
          new updated emp.rank > old up-
              dated emp.rank
          and new updated emp.rank < 5)

    then update bonus

          set raiseamnt = raiseamnt + 500
          where empid = e1.empid

    precedes {rule_2}
follows {rule_3}
```

Figure 4: Command to insert rule_1 with priority info into a Starburst rule base.

The optional syntactic categories **precedes** *rule-list* and **follows** *rule-list* can be used to reflect one rule's priority over another.

Example 6 Recall our insertion of rule_1 from Figure 1. Suppose, instead, that we had inserted what is now shown in Figure 4. For the sake of discussion, assume that some set of events triggered r_1, r_2, and r_3 at the same time. *Before* Starburst begins evaluating any rule, it must resolve this conflict by choosing one of the rules in a fashion consistent with its conflict resolution policy. Then r_3 would be evaluated first, for all priority information is specified in r_1's definition (r_2 and r_3 remain as they appear in Figures 2 and 3). Thus, r_1 precedes (i.e. has priority over) r_2, but not r_3. Hence, r_3 is selected and evaluated first. We should stress that conflict resolution *precedes* rule condition evaluation.

As mentioned throughout this subsection, we wish to identify the similarities between different rule systems. Although different rule systems enforce

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

Table 9: **emp** table containing three tuples.

different policies for resolving conflicts, we make note of the fact that such policies can be classified to be either unordered, partially ordered, or totally ordered.

2.4.3 Execution Semantics of Starburst

We have almost completed the construction of our concrete ADBMS. So far, we have created a small database containing employee information, designed three ECA rules, translated those rules into Starburst syntax, and determined Starburst's policy of resolving conflicts. The only thing we have not yet defined is Starburst's *execution semantics*, which describes how all these components interact in order to process information [20]. Unlike the previous issues we have discussed, the execution semantics of a particular ADBMS is *unique*. This is what creates the real challenge inherent in analyzing rule systems [20]. In Section 3, we tackle the issue of providing a generic framework modeling each ADBMS's unique execution semantics. Because of the uniqueness of each ADBMS's execution semantics, we rely on our running example for presenting a simulation of Starburst's execution semantics. We begin with an initial transaction which triggers rules in our rule base. We continue to follow the sequence of events that occur subsequent to this initial transaction, pausing to discuss each step along the way.

Example 7 The next few pages provide a detailed example of the executional semantics of Starburst.

We assume that some tuples already exist in our two database tables as shown in Tables 9 and 10. Also, we make this example more interesting by adding one more rule r_4 to our rule base. The resulting set of rules is summarized in Table 11. Notice that r_1 and r_4 are signaled by the same event, but r_4 has a higher priority as specified in the **precedes** clause.

Now, suppose Matt and Jasmine receive promotions and need to have their ranks in their company increased by one. We could perform this update

empid	raiseamnt
1	1000
2	1500
3	500

Table 10: **bonus** table containing three tuples.

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 11: Rule Set used for example 7.

using the following SQL command:

update emp

set rank = rank + 1
where empid = 1 or empid = 2.

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, $\mathcal{R}_c = \{r_1, r_4\}$. At any point during rule processing, the consideration set contains all rules that have been triggered, but have not yet been evaluated [1]. If $\mathcal{R}_c$ is *not empty* the ADBMS "knows" that there are some rules it must process (i.e. evaluate the condition of the chosen rule and, if true, perform its action). Moreover, if $\mathcal{R}_c$ contains *multiple* elements, the ADBMS applies its conflict resolution policy for choosing a rule from $\mathcal{R}_c$.

At this point, Starburst selects r_4 for *consideration* and removes it from $\mathcal{R}_c$. *Consideration* is simply Starburst's process of choosing a rule from $\mathcal{R}_c$, evaluating its condition and, if true, executing its action [1]. 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. Table 12 summarizes what has occurred up to this point. Although new rules were not triggered, note that $\mathcal{R}_c$ still contains r_1. Starburst realizes $\mathcal{R}_c$ is not empty and that it contains a sole rule. Thus, r_1 is chosen for consideration; its condition also evaluates to TRUE, *but for Matt only*! To see this, refer back to Figure 1. The condition read as follows:

(select rank from *new updated emp.rank* where
new updated emp.rank > old updated emp.rank
and *new updated emp.rank* < *5*)

where *old updated emp.rank* contains our initial values (Table 9) but *new updated emp.rank* contains what is shown in table 13. The implications of r_1's action are to increase Matt's raiseamnt (in the **bonus** table) by \$500, but leave Jasmine's raiseamnt unchanged (since her rank *equals* 5). Furthermore, execution of this action triggers the rule r_2 (see Table 14).

Starburst proceeds to automatically process newly triggered rules in the same fashion. In this manner r_2 causes Matt's and Jasmine's salaries in the **emp** table to be increased by \$1,500, and also triggers r_3 which displays

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

Table 12: Summary of events following initial transaction and r_4's action.

.

empid	name	salary	rank
1	Matt Shirley	50000	*4*
2	Jasmine Reick	65000	*5*
3	Darren King	45000	2

Table 13: new updated emp.rank table.

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	$\{r_2\}$	$\{r_2\}$	N/A	N/A	N/A

Table 14: Summary of events following initial transaction and r_1's consideration.

.

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 15: Complete summary of events following initial transaction.

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

Table 16: **emp** after completion of rule processing.

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. Table 15 summarizes this entire process from start to finish. The resulting values of the **emp** table are shown in Tables 16 and 17.

2.5 Previous Efforts to analyze ADBMS Behavior

Having a clear description of this ADBMS and its operation, we can now address the issues of modeling and Model Checking rule systems. Before we do that, we briefly discuss previous efforts at analyzing rule systems and assess the results that were obtained.

Previous efforts at predicting rule system behavior have been addressed in [1, 8, 21, 22]. These efforts were directed specifically towards Starburst, but according to [1] can be modified and applied to any rule system. In [3], a

empid	raiseamnt
1	*1500*
2	1500
3	500

Table 17: **bonus** table after completion of rule processing.

more general relational and algebraic approach was used. The benefit of such an approach is that it is not specific to Starburst; its major disadvantage, however, is that it can only be applied to rule systems built over DBMSs using the relational model. In [3] one finds 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.

Termination is a property which ensures that rule processing is *guaranteed* to terminate. In particular, one would like to ensure that rule execution does not continue in a cyclic manner [1] as noted in Example 5.

A set of triggered rules is *confluent* if, the order in which rules under consideration are being processed is immaterial to the final outcome [1]. Assigning priorities to rules is the easiest means of achieving a confluent set of rules.

A set of triggered rules is *observably deterministic* if, regardless of the order in which *non-prioritized* rules are processed, any final *observable* outcome (caused, for instance, by a rule that displays information to the user's terminal) will be the same [1].

It is hardly surprising that attempts to analyze these properties have been somewhat inconclusive. In [1, 8, 12], it is suggested that it would not be possible to construct an algorithm which can prove, in general, whether or not these properties hold for a set of rules. In fact, these notions are in general undecidable. Therefore, our best hope is to provide *sufficient* conditions under which our model checking methodology which successfully verify the properties above. If such model checks fail, we can only infer that the given ADBMS *may or may not* enjoy the respective property. Surprisingly, no previous efforts seem to have been directed towards analyzing more specific properties of rule sets; for instance, checking if certain rule integrity constraints, specified in the pre-developmental design specification of an ADBMS, hold.

Example 8 Reconsider our set of rules from previous examples. Our primary goal was to design these rules to reflect our corporation's policy of giving raises whenever an employee is promoted in rank. Thus, we certainly expect that our rules behave in a way such that an increase in an employee's rank *always* results in an increase to that employee's salary. We usually take for granted that such properties holds. However, human error and unpredictable rule interaction can foil even the most meticulous design specification. In fact, it turns out that this property *is not satisfied* in our simple

rule set! Such subtle errors can occur quite frequently in rule design. Thus, the problems inherent in designing a set of unambiguous rules are reminiscent to problems occurring in the design and implementation of concurrent hardware and software. This makes model checking a natural candidate for a methodology of analyzing ADBMSs [16].

3 Our Model of an ADBMS

Model Checking ADBSMs means that our analysis of ADBMSs won't be applied to actual ADBMSs, but to an *abstract model* of such systems. Abstraction simplifies our verification efforts by allowing us to omit details of a real ADBMS that do not effect the analysis of rule behavior [7]. In Subsection 1.3 we pointed out that it is beneficial to view an ADBMS as an *open system*, since decisions made by the environment, in which the ADBMS operates, could affect its behavior. Although we have not yet given any explicit examples of such decisions, Section 2 suggests what some of these might be. The Example 9 below depicts a situation where decisions made by Starburst's environment affect its behavior.

Example 9 Consider Example 7. The initial update triggered two rules r_1 and r_4. Rule r_4 was selected for consideration first, since it had a higher priority. This scheduling decision made by Starburst's *environment* and based on its conflict resolution policy. An ADBMS operating under a *different* environment, e.g. one which enforced a *different* conflict resolution policy (see Subsection 2.4.2), would, in general, behave differently.

In this section, we present our model of an ADBMS. A motivating factor behind the design of this model is its ability to represent an ADBMS in the context of different environments. Thus, although we apply our efforts toward modeling the Starburst rule system, it should become clear that any rule system could be modeled under this framework. We begin with a formal definition of the model and proceed by examining each piece in greater detail.

3.1 The Model

Definition 10 Formally, a model of ADBMS rule processing is a tuple,

$$\mathcal{M} = (\mathcal{R},\ \mathcal{C},\ \mathcal{S},\ \stackrel{r}{\rightarrow},\ \stackrel{\mathtt{tt}}{\rightarrow},\ \stackrel{\mathtt{ff}}{\rightarrow},\ L_1,\ L_2)$$

consisting of:

- a set of rules $\mathcal{R} = \mathbf{R_1} + \mathbf{R_2}$, where $\mathbf{R_1}$ is a set of ECA rules, and $\mathbf{R_2}$ is a set of EA rules.
- a set of environment (controller) states $\mathcal{C}$,
- a set of system states $\mathcal{S}$,
- for each $r \in \mathcal{R}$, a binary relation, $\xrightarrow{r} \subseteq \mathcal{C} \times \mathcal{S}$
- two binary relations $\xrightarrow{\texttt{tt}}$ and $\xrightarrow{\texttt{ff}}$, on $\mathcal{S} \times \mathcal{C}$, and
- two labeling functions $L_1 : \texttt{Atoms} \rightarrow \mathcal{P}(\mathcal{C})$ and $L_2 : \texttt{Atoms} \rightarrow \mathcal{P}(\mathcal{S})$ which map each propositional atom to the set of *environment*, respectively *system* states where that proposition is true.

In addition, we demand that for each $c \in \mathcal{C}$, each $s \in \mathcal{S}$, and each $r \in \mathbf{R_1}$, if $c \xrightarrow{r} s$, then s has exactly one `tt` and one `ff` successor. Furthermore, we demand that for each $c \in \mathcal{C}$, each $s \in \mathcal{S}$, and each $r \in \mathbf{R_2}$, if $c \xrightarrow{r} s$, then s has exactly one `tt` successor and *no* `ff` successor.

Figure 5 shows our model of Example 7. Note that the names inside nodes are state names and not propositional atoms true at such states. The latter will be addressed in a later section. The initial state, c_0, reflects the state of Starburst immediately after the initial update to Matt's and Jasmine's salaries. The collection of paths, beginning at c_0, reflect all the computations that could *possibly* occur during rule processing. The execution path that Starburst *actually* selected in Example 7 is denoted by dashed lines. Also note that s_0 and s_2 only have a `tt` successor since r_2 and r_4 are EA rules. This model satisfies the requirement given in Definition 10. We refer back to this model throughout the remainder of this section. The reader familiar with ordinary models for CTL will recognize that our models are based on CTL models which strictly alternate between system and environment capacities.

As customary, the states of the model reflect possible states of the ADBMS we are modeling. For instance, an ADBMS may be in a state of computing what rules have been triggered by an event, or it may be in a state of evaluating some rule's condition. Our binary relations describe the ADBMS's capacity to move from one state to the next; e.g. such a transition could result when an ADBMS executes a rule's action. Finally, the labeling functions (omitted in our example) determine what atomic propositions are true

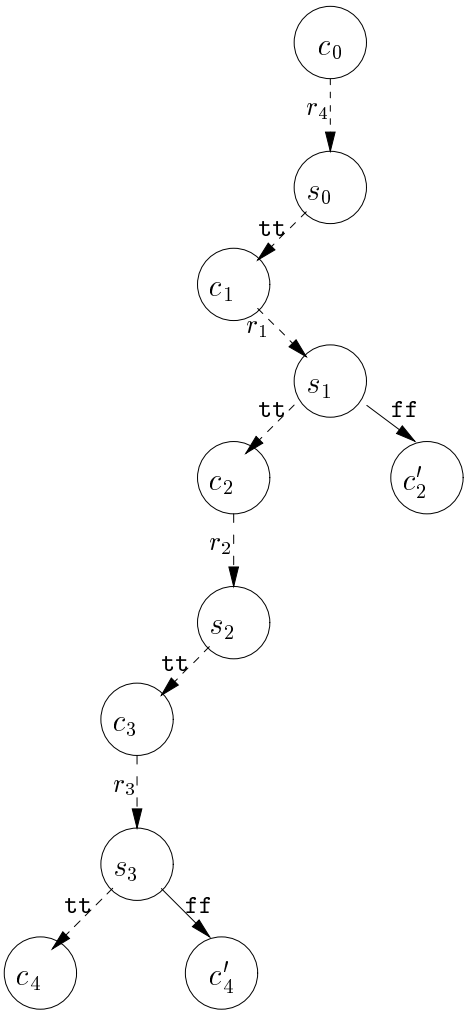

Figure 5: A model of Example 7 showing all the possible behavior of our ADBMS.

state	event	table	field(s)	$\mathcal{R}_c$	priority
c_0	update	**emp**	rank	$\{r_1, r_4\}$	r_4 precedes r_1

Table 18: Configuration of the environment in state c_0.

at each state. This allows one to represent states as bit vectors using e.g. the description language of SMV [15]. The logic **EATL** associated with such models is presented in Section 4.

Our notion of model allows us to distinguish between those decisions that depend on the particular ADBMS we are modeling (i.e. its environment), and those which do not (i.e. its underlying system).

3.2 The Interaction between Environment and System

Section 2 provided us with a general idea of how interaction occurs between an ADBMS's environment and its system. For instance, at any point *following the initial event* and *preceding the termination of processing*, the *next transition* of the ADBMS is determined as follows:

- If the ADBMS is in some state $c_j \in C$ and $\mathcal{R}_c$ is not empty at c_j, then the ADBMS's *environment* must choose a rule r_i for consideration based on c_j's current configuration. The ADBMS then makes a transition to some system state $s_j \in S$.

- If the ADBMS is in some state $s_j \in S$ then the *system* chooses whether it executes r_i's action or not (recall that r_i was chosen above in state c_j). This choice depends on how the system evaluates r_i's condition. If r_i is an EA rule this choice is deterministic and the system always executes its action. Regardless of what the system chooses, the ADBMS then makes a transition to some environment state $c_{j+1} \in C$.

Example 11 Consider Example 7. Initially, we executed an update on Matt's and Jasmine's ranks in the **emp** table. In terms of our model, we can imagine Starburst making a transition into environment state c_0 when the initial event occurs. At this point, Starburst's environment must decide which rules are triggered by an update to **emp**(rank), which rules should be added to $\mathcal{R}_c$, and which rule should be selected for consideration.

state	action	table	field(s)
s_0	retrieve	**emp**	empid,name,salary,rank

Table 19: Configuration of the system in state s_0.

state	event	table	field(s)	$\mathcal{R}_c$	priority
c_0	update	**emp**	rank	$\{r_1, r_4\}$	r_4 precedes r_1
c_1	retrieve	**emp**	empid,name,salary,rank	$\{r_1\}$	N/A

Table 20: Configuration of the environment in state c_1.

Table 18 shows the configuration of Starburst's *environment* in state c_0. Based on c_0's configuration, the environment determines that $\mathcal{R}_c$ is not empty and chooses r_4 for consideration (since r_4 precedes r_1). Upon consideration of r_4, Starburst makes a transition into system state s_0 (see Figure 5). The *system* must now determine if r_4's condition is TRUE, and if so, what action should be executed.

Table 19 shows the configuration of Starburst's *system* in state s_0. Because r_4 is an EA rule, the only possible transition out of s_0 is a `tt` transition. Thus, the system performs a retrieve on the **emp** table. Starburst makes a transition into state c_1 and control reverts back to its environment. Table 20 shows the configuration of Starburst's environment in state c_1. Notice that although s_0's action did not trigger any new rules, the environment still has knowledge of its configuration *prior* to entering c_1 (i.e. c_0's configuration). This concurs with what would occur in the actual ADBMS.

In state c_1, the environment has to select r_1 for consideration, and the ADBMS enters system state s_1. At this point, the system needs to determine whether r_1's condition is TRUE, and if so, what action should be executed. We chose to omit any representation of the database state in our model. This omission is the basis for an abstract interpretation of the actual ABDMS and makes it possible to model rule processing in a *finite* number of states. In our example this means that the system has absolutely no information about the concrete values in tables **emp** and **bonus**. So how can the system determine if it should execute r_1's action? We address this critical problem of modeling rule condition evaluation in Subsection 3.4. For now, we pretend that the system can obtain information from Starburst's database and evaluate each rule's condition according to Example 7. Then, since r_1's condition was TRUE, the system executes an update on **bonus**(raiseamnt).

state	event	table	field(s)	$\mathcal{R}_c$	priority
c_0	update	**emp**	rank	$\{r_1, r_4\}$	r_4 precedes r_1
c_1	retrieve	**emp**	empid,name,salary,rank	$\{r_1\}$	N/A
c_2	update	**bonus**	raiseamnt	$\{r_2\}$	N/A
c_3	update	**emp**	salary	$\{r_3\}$	N/A
c_4	retrieve	**emp**	empid,name,salary,rank	$\{\ \}$	N/A

Table 21: Configurations of all environment states reached during rule processing.

state	action	table	field(s)
s_0	retrieve	**emp**	empid,name,salary,rank
s_1	update	**bonus**	raiseamnt
s_2	update	**emp**	salary
s_3	retrieve	**emp**	empid,name,salary,rank

Table 22: Configurations of all system states reached during rule processing.

The interaction between the environment making decisions and the system reacting to those decisions continues until Starburst hopefully reaches a state where rule processing terminates (i.e. $\mathcal{R}_c$ becomes empty). Tables 21 and 22 summarize this entire interactive process.

A salient feature of this interaction is that it strictly alternates between choices made by Starburst's environment and its system. In [2], Alur and others suggest that this alternation can be thought of as a "game" of environment versus system, where each "agent" makes a move on its respective turn. Indeed, when Starburst's environment chooses a rule for consideration, it is up to the system to determine if that rule's action should be executed. Analyzing all the *possible* outcomes of this game helps us determine if the environment and system could interact in a manner inconsistent with the designer's intentions [2].

3.3 The States of our Model

In general, an ADBMS begins processing ECA rules in response to some triggering event which is initiated by the user. From that point, up to the point where rule processing terminates, either the ADBMS's environment or its underlying system determine how rule processing is to proceed. Thus, at any time during rule processing, an ADBMS is either in some environment

state, $c \in \mathcal{C}$, or some system state, $s \in \mathcal{S}$. Initially, the ADBMS is in environment state c_0.

From Subsection 2.4, we know that the manner in which ECA rules are processed under Starburst is, in part, determined by its execution model. Furthermore, we discovered that different ADBMSs often have different execution models. Therefore, we can conclude that *it is the execution model which defines how a particular ADBMS behaves.* The next example illustrates how Starburst's execution model defines its environment.

Example 12 Recall Examples 7 and 11. Our initial command updated Matt's and Jasmine's salaries. In terms of Figure 5, we can imagine Starburst entering state c_0. At this point, Starburst's environment determined that r_1 and r_4 should be triggered, r_1 and r_4 should be added to $\mathcal{R}_c$, and r_4 should be considered before r_1.

All of these decisions reflect some aspect of Starburst's execution model. For instance, 1) reflects the event types which Starburst's event detector recognizes; 2) reflects Starburst's execution semantics; and 3) reflects Starburst's conflict resolution policy. Hence, during rule processing, we define an ADBMS to be in an environment state, $c \in \mathcal{C}$, whenever it is computing the rules that are triggered by an event, the set of rules eligible for evaluation and possible execution (i.e. $\mathcal{R}_c$), and the rule to be selected for consideration. Similarly, we define an ADBMS to be in a system state, $s \in \mathcal{S}$, whenever a rule is already under consideration, and an action is ready for execution.

Of course, the ability to compute these results presumes that each state in our model is enriched with information about rules in the ADBMS's rule base. Section 5 gives some examples of how we might encode this information into states. For now, we assume that, at each state, the ADBMS may obtain any rule information necessary to perform its computations. Clearly, whenever the system must evaluate a considered ECA rule, it considers *two* possibilities: executing the rule and not executing the rule. Later on we will see that our verification framework accounts for that by doing a conservative flow analysis.

3.4 The Transition Relations of our Model

Subsection 3.2 informally described the interaction between an ADBMS's environment and its underlying system. In this subsection we wish to formalize and generalize these notions. The binary relation $\xrightarrow{r}$ is defined for

each $r \in \mathcal{R}$. We write $c \xrightarrow{r_i} s$ to express that it is possible for the ADBMS to reach state s from state c by considering rule r_i. The following example illustrates such state transitions.

Example 13 Consider the first transition, c_0 to s_0, described in Example 11 and illustrated in Figure 5. The transition was made upon the environment choosing r_4 for consideration at c_0. Formally, we would express this transition in our model as: $c_0 \xrightarrow{r_4} s_0$.

Example 11 is somewhat unrealistic because at every state, $c_i \in \mathcal{C}$, only one rule is eligible for consideration. This does not allow us to demonstrate the true power of our model. Our next example presents a more complicated situation.

Example 14 Consider Example 11 again, but this time, let's assume that r_1 and r_4 have the *same* priority. Then, at state c_0, the environment must *non-deterministically* select which rule should be considered first. Hence, there are two possible transitions out of state c_0: $c_0 \xrightarrow{r_4} s_0$ and $c_0 \xrightarrow{r_1} s_0'$. Figure 6 shows how we would represent this graphically. The model is quite large and demonstrates how non-determinism can drastically increase the state space. We should point out, however, that, as in Figure 5, the paths in this model all terminate eventually.

The power to express this non-deterministic choice in our model is essential to realizing reliable results during verification. We may also apply non-determinism in order to abstract certain details of the actual system [10]. It is this application of non-determinism that allows us to simulate the evaluation of an ECA rule's condition at *system* states; a topic which was previously ignored in Subsection 3.2.

The two binary relations, $\xrightarrow{\texttt{tt}}$ and $\xrightarrow{\texttt{ff}}$, allow us to effectively abstract all details about the ADBMS's database. We know from Section 3.2 that when our ADBMS is in some state $s \in \mathcal{S}$, the system may choose either to execute or not execute an ECA rule's action. Therefore, we write $s \xrightarrow{\texttt{tt}} c$ to express that it is possible for an ADBMS to reach state c from state s by *executing* rule r_i's action. We write $s \xrightarrow{\texttt{ff}} c$ to express that it is possible for an ADBMS to reach state c from state s when r_i's action is *not executed* (i.e. if r_i is an ECA rule).

Example 15 Consider again the transition from s_1 to c_2 given in Example 11 and illustrated in Figure 5, but this time suppose we do not know that r_1's

state	event	table	field(s)	$\mathcal{R}_c$	priority
c_0	update	**emp**	rank	$\{r_1, r_4\}$	r_4 precedes r_1
c_1	retrieve	**emp**	empid,name,salary,rank	$\{r_1\}$	N/A
c_2	update	**bonus**	raiseamnt	$\{r_2\}$	N/A
c_2'	N/A	N/A	N/A	$\{\ \}$	N/A
c_3	update	**emp**	salary	$\{r_3\}$	N/A
c_4	retrieve	**emp**	empid,name,salary,rank	$\{\ \}$	N/A
c_4'	N/A	N/A	N/A	$\{\ \}$	N/A

Table 23: Configurations of all the environment states for Figure 5.

condition must evaluate to true. Then there are two transitions out of state s_1: $s_1 \stackrel{\texttt{tt}}{\rightarrow} c_2$ and $s_1 \stackrel{\texttt{ff}}{\rightarrow} c_2'$. On the other hand, for any EA rule $r \in \mathbf{R_2}$ under consideration at state $s \in \mathcal{S}$, we have exactly one `tt` successor and *no* `ff` successor. For example, at state s_0, the EA rule r_4 is under consideration.

Notice that in Figures 5 and 6 there are various paths along which rule processing may proceed. Each of these alternatives represents a unique *computation path* of the ADBMS. Formally, a computation path of $\mathcal{M}$ is a sequence of states $\eta = c_0, s_0, c_1, s_1, \ldots, c_j, s_j, \ldots$, such that $c_j \stackrel{r_i}{\rightarrow} s_j$, and $(s_j \stackrel{\texttt{tt}}{\rightarrow} c_{j+1}$ or $s_j \stackrel{\texttt{ff}}{\rightarrow} c_{j+1})$.

Example 16 Consider Examples 14 and 15. One possible computation path is $\eta = c_0, s_0, c_1, s_1, c_2 \ldots$. Another, different possibility is $\eta' = c_0, s_0, c_1, s_1, c_2'$.

In order to understand a model in its entirety, it is useful to account for information computed and stored at each state. Table 23 shows the configuration of each environment state depicted in Figure 5. Technically, this information could be provided by labeling states with appropriate atomic propositions. Notice that at any environment state where $\mathcal{R}_c$ is empty (such as c_4), there are no edges out of that state's node. This indicates that rule processing terminates in such states. The configuration of the system states is the same as in Table 22.

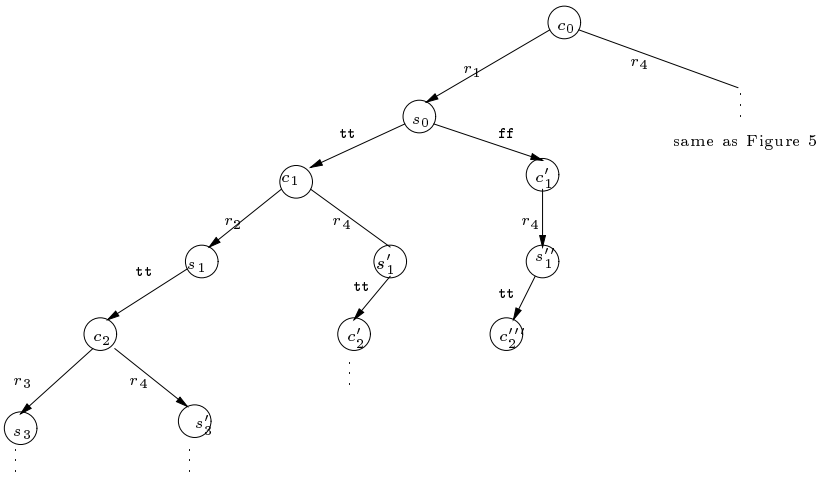

Figure 6: A complete model of example 14.

3.5 Labeling Models with Propositions

As rule processing proceeds in a rule system, we expect certain atomic propositions to be true when the ADBMS is in a certain state. For instance, in Figure 5, it may be useful to know that state c_0 "is triggered by an update action", or state s_0 "can execute a retrieve action". We can apply these atomic propositions towards constructing formulas which specify properties we desire to hold in our rule system. Indeed, the primary objective of Model Checking is to verify automatically that what we desire to be true about a model actually be true.

In our model, we view an ADBMS as a cooperative of environment and system. Thus, we expect certain propositions to hold at each environment state, and certain, *different* propositions to hold for the system. This view is facilitated by the existence of two labeling functions in our model: L_1 and L_2. It is not problematic that atoms for both the environment and system come from the same set, since our labeling functions effectively handle such a distinction. In fact, we could check if the proposition "can execute a retrieve action" is TRUE at some environment state, but such a check always returns a FALSE answer. Such consistencies will be implicitly guaranteed by a program which describes the model (e.g. a program written in SMV or *Spin*).

4 Event-Action Temporal Logic

Certainly, as ADBMS designers, we would write a formal specification defining the requirements we expect our rule system to meet. Example 8, for instance, suggests a rule behavior constraint we want to enforce in our rule system. This constraint specified that, if an employee's rank is updated, their salary should ultimately be updated automatically. Thus, our specification should require that, no matter how rules are processed, this constraint holds. Model Checking, or any other formal verification method, should then ensure our rule system's correctness with respect to this specification [11, 17]. Since we wish to reason about ADBMS rule processing, we offer a specification language that reflects the interaction that occurs as a result of the sequence of decisions, made in an alternating fashion, by the ADBMS's environment and its system. We present the syntax and semantics of Event-Action Temporal Logic (**EATL**), a branching-time temporal logic based on CTL (which models *closed* systems) [2]. We can use **EATL** to describe the behavior we expect an ADBMS to exhibit during rule processing. As before, we apply **EATL** towards checking properties pertaining to our Starburst rule system (and gain some surprising insights!).

4.1 Syntax of EATL

In Section 3, we modeled rule processing as a series of decisions, made in an alternating fashion, between the environment and system of an ADBMS. More precisely, $\bigcup_{r \in \mathcal{R}} \xrightarrow{r}$ captures all possible decisions made by the ADBMS's environment which take the ADBMS from an environment state to a system state. Likewise, $\xrightarrow{\mathtt{tt}} \cup \xrightarrow{\mathtt{ff}}$ represents all possible actions taken by the ADBMS's underlying system which move the ADBMS back to an environment state. For this reason, the logic **EATL** (an action-based, alternating CTL) is comprised of two *distinct* but *interdependent* sub-logics: $\mathbf{EATL}_C$ and $\mathbf{EATL}_S$. Their interdependence allows us reason about the *interaction* between environment and system.

Definition 17 Given a set of propositional atoms $\mathtt{Atoms}$, two sets of propositional variables, $\mathtt{Var_1}$ and $\mathtt{Var_2}$, a set of ECA rules $\mathcal{R} = \mathbf{R_1} + \mathbf{R_2}$, we let K range over subsets of $\mathcal{R}$ and define $\mathbf{EATL}_C$ formulas ψ and $\mathbf{EATL}_S$ formulas ϕ as:

$$\begin{array}{rcl} \psi & ::= & x \mid \texttt{ff} \mid p \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid [K]\phi \mid \langle K\rangle\phi \mid \mu x.\psi \\ \phi & ::= & y \mid \texttt{ff} \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid (\texttt{tt})\psi \mid (\texttt{ff})\psi \mid \mu x.\phi. \end{array}$$

where $x \in \texttt{Var}_1$ and $y \in \texttt{Var}_2$.

As usual, we may derive the logical operators $\vee$ and $\Rightarrow$ for both $\mathbf{EATL}_C$ and $\mathbf{EATL}_S$. Moreover, we define $\nu x.\psi$ by $\neg\mu x.\neg\psi[\neg x/x]$ and $\nu x.\phi$ by $\neg\mu x.\neg\phi[\neg x/x]$, where $[\neg x/x]$ is the substitution of $\neg x$ for free occurrences of x in its argument. The meanings of the modalities $\langle K\rangle$ and $[K]$ are based on the standard semantics for the modal mu-calculus; we use $\langle -\rangle$ as a shorthand for expressing $\langle \mathcal{R}\rangle$ as in [5].

Example 18 Let us reconsider the model in Figure 6. Suppose the $\mathbf{EATL}_C$ formula $\psi = [-]\phi$ is true at environment state c_0. Then this formula states that for *all* possible transitions out of c_0, which take place following the consideration of a rule in $\mathcal{R}_c$ ($c_0 \xrightarrow{r_1} s_0$ and $c_0 \xrightarrow{r_4} s_0'$ in this case), the $\mathbf{EATL}_S$ formula ϕ is true at *all* the next system states (s_0 and s_0'). Suppose, instead, the formula $\psi = [\mathbf{R_2}]\phi$ were true at state c_0. Then this formula states that for all possible transitions out of c_0, which take place following consideration of an EA rule in $\mathbf{R_2}$ ($c_0 \xrightarrow{r_4} s_0'$), the $\mathbf{EATL}_S$ formula ψ is true at all the next system states (at s_0'). The $\mathbf{EATL}_C$ formula $\psi = \langle -\rangle\phi$ has a similar meaning, but is less restrictive. The formula $\psi = \langle -\rangle\phi$ is true at state c_0 if consideration of *at least one* rule in $\mathcal{R}_c$ causes the ADBMS to enter a system state where ϕ is true (s_0 or s_0').

Example 18 shows how the modalities of $\mathbf{EATL}_C$ can be used to express the usual branching-time temporal properties of CTL imposed on an environment state. In a similar fashion, the modal operators defined for $\mathbf{EATL}_S$ are used to express properties that, depending on how a Starburst system evaluates a rule's condition, should hold at the subsequent environment state.

Example 19 Consider again the model in Figure 6 and suppose the $\mathbf{EATL}_S$ formula $\phi = (\texttt{tt})\psi$ is true at state s_0 Then, this formula states that, if Starburst's system executes r_1's action, ψ is true at state c_1. Similarly, if the formula $\phi = (\texttt{ff})\psi$ were true at state s_0, then ψ would be true at state c_1'.

Of course, one can now write specifications which *hide* intermediate system or environment states. We will return to this point later.

4.2 Semantics of EATL

The semantics of **EATL** formulas over our notion of models is the obvious adaptation of the standard semantics for the corresponding fragment of the modal mu-calculus [5].

Definition 20 Given a model $\mathcal{M} = (\mathcal{R},\ \mathcal{C},\ \mathcal{S},\ \xrightarrow{r},\ \xrightarrow{\texttt{tt}},\ \xrightarrow{\texttt{ff}},\ L_1,\ L_2)$ of ADBMS rule processing, a valuation $\rho =< \rho_1, \rho_2 >:< \texttt{Var}_1 \mapsto 2^{\mathcal{C}}, \texttt{Var}_2 \mapsto 2^{\mathcal{S}} >$, and some $\psi \in \mathbf{EATL}_C$, we denote the set of states where ψ is true with respect to ρ by $[\![\psi]\!]^c_\rho$. We define $[\![\psi]\!]^c_\rho$ inductively as follows:

1. $[\![x]\!]^c_\rho = \rho_1(x)$
2. $[\![p]\!]^c_\rho = \{c \in \mathcal{C} \mid c \in \mathcal{L}_1(p)\}$, for each $p \in \texttt{Atoms}$
3. $[\![\ \texttt{ff}\ ]\!]^c_\rho = \{\}$
4. $[\![\neg\psi]\!]^c_\rho = \mathcal{C} - [\![\psi]\!]^c_\rho$
5. $[\![\psi_1 \wedge \psi_2]\!]^c_\rho = [\![\psi_1]\!]^c_\rho \cap [\![\psi_2]\!]^c_\rho$
6. $[\![[K]\phi]\!]^c_\rho = \{c \in \mathcal{C} \mid \forall s \in \mathcal{S}, \forall r \in K\ .\ c \xrightarrow{r}\ s \text{ implies } s \in [\![\phi]\!]^s_\rho\}$
7. $[\![\langle K\rangle\phi]\!]^c_\rho = \{c \in \mathcal{C} \mid \exists s \in \mathcal{S}, \exists r \in K\ .\ c \xrightarrow{r}\ s \text{ and } s \in [\![\phi]\!]^s_\rho\}$
8. $[\![\mu x.\psi]\!]^c_\rho = \bigcap\{\mathcal{A} \subseteq \mathcal{C} \mid [\![\psi]\!]^c_{\rho[x\mapsto\mathcal{A}]} \subseteq \mathcal{A}\}$, where $\rho[x \mapsto \mathcal{A}]$ is the valuation ρ' which agrees with ρ on all y except that $\rho'(x) = \mathcal{A}$.

For any $\phi \in \mathbf{EATL}_S$, we denote the set of states where ϕ is true with respect to ρ by $[\![\phi]\!]^s_\rho$. We define $[\![\phi]\!]^s_\rho$ inductively as follows:

1. $[\![x]\!]^s_\rho = \rho_2(x)$
2. $[\![p]\!]^s_\rho = \{s \in \mathcal{C} \mid s \in \mathcal{L}_2(p)\}$, for each $p \in \texttt{Atoms}$
3. $[\![\ \texttt{ff}\ ]\!]^s_\rho = \{\}$
4. $[\![\neg\phi]\!]^s_\rho = \mathcal{S} - [\![\phi]\!]^s_\rho$
5. $[\![\phi_1 \wedge \phi_2]\!]^s_\rho = [\![\phi_1]\!]^s_\rho \cap [\![\phi_2]\!]^s_\rho$
6. $[\![(\texttt{tt})\psi]\!]^s_\rho = \{s \in \mathcal{S} \mid \exists c \in \mathcal{C}\ \ s \xrightarrow{\texttt{tt}}\ c \text{ and } c \in [\![\psi]\!]^c_\rho\}$

7. $[\![(\texttt{ff})\psi]\!]^s_\rho = \{s \in \mathcal{S} \mid \forall c \in \mathcal{C} \quad s \xrightarrow{\texttt{ff}} c \text{ implies } c \in [\![\psi]\!]^c_\rho\}$

8. $[\![\mu x.\phi]\!]^s_\rho = \bigcap\{\mathcal{A} \subseteq \mathcal{S} | [\![\phi]\!]^s_{\rho[x \mapsto \mathcal{A}]} \subseteq \mathcal{A}\}$, where $\rho[x \mapsto \mathcal{A}]$ is the valuation ρ' which agrees with ρ at all y except that $\rho'(x) = \mathcal{A}$.

Theorem 1 The semantics of **EATL** is well-defined since all operators have a monotone meaning except $\neg$, which is anti-tone.

The proof of this in a mere adaptation of the corresponding fact for the modal mu-calculus. It should be clear that this semantics gives rise to a generalization of the model checking algorithms which are based on labeling states with subformulas of a given specification [15].

Convention 21 Given a model $\mathcal{M} = (\mathcal{R},\ \mathcal{C},\ \mathcal{S},\ \xrightarrow{r},\ \xrightarrow{\texttt{tt}},\ \xrightarrow{\texttt{ff}},\ L_1,\ L_2)$ of ADBMS rule processing, a valuation ρ, and an $\mathbf{EATL}_C$ or $\mathbf{EATL}_S$ formula, we define $\mathcal{M}, c \models_\rho \psi$ (read "c satisfies ψ in $\mathcal{M}$ w.r.t. ρ") by $c \in [\![\psi]\!]^c_\rho$ and $\mathcal{M}, s \models_\rho \phi$ (read "s satisfies ϕ in $\mathcal{M}$ w.r.t. ρ") by $s \in [\![\phi]\!]^s_\rho$.

In the next example, we demonstrate the usefulness of **EATL** by checking a few simple properties for the model depicted in Figure 5.

Example 22 Refer back to Tables 21 and 22 which show the configurations of the states for the model in Figure 5. We can use this information to define a number of atomic propositions. Suppose `Atoms` includes the following four atomic propositions:

- p_1=`is triggered by an update event`,
- p_2=`is triggered by a retrieve event`,
- p_3=`can execute an update action`, and
- p_4=`can execute a retrieve action`.

Of course, Tables 21 and 22 provides much more information, and each atomic description is a suitable candidate for an additional atomic proposition. Some of the checks we can make are listed below.

- $c_0 \models p_1$ holds since, according to Table 21, c_0 is in L_1 (`is triggered by an update event`).

- $s_0 \models p_4$ holds since, according to Table 22, s_0 is in L_2 (`can execute a retrieve action`).
- $c_0 \models p_1 \wedge [r_4]p_4$ holds since $c_0 \models p_1$, $s_0 \models p_4$, and since $c_0 \stackrel{r_4}{\rightarrow} s_0$ is the only possible r_4 transition out of c_0.
- $s_0 \models p_4 \wedge (\mathtt{tt})\neg p_2$ holds since $s_0 \models p_4$, $c_1 \not\models p_2$, and there is a transition $s_0 \stackrel{\mathtt{tt}}{\rightarrow} c_1$.

4.3 Branching-Time Operations in EATL

In Subsection 1.3, we mentioned that one limitation of using Computation Tree Logic (CTL) for analyzing ADBMS rule processing was its natural restriction to closed systems [2]. Nevertheless, the ability to write branching-time properties gives CTL much expressive power [10] which can be put to actual use since several software tools for model checking CTL, such as the Symbolic Model Verifier (SMV) [15], have been developed. SMV is a Model Checking tool which uses efficient data structures and algorithms [6] for checking CTL formulas over models of finite-state systems [10, 15] and a description language for models and specifications [10] . Alas, we will shortly see why SMV cannot be practically applied towards modeling ADBMS rule processing.

4.3.1 Expressing Branching-Time Properties

We examine in detail only the branching-time operations for $\mathbf{EATL}_C$which are suitable abstractions of those for CTL. We adopt a CTL-like notation:

- $c_0 \models \mathrm{AG}^c\,\psi$ iff for all computation paths $\eta = c_0, s_0, c_1, s_1, \ldots$ we have $c_i \models \psi$ for all c_i along that path.
- $c_0 \models \mathrm{EG}^c\,\psi$ iff there exists a computation path $\eta = c_0, s_0, c_1, s_1, \ldots$ such that we have $c_i \models \psi$ for all c_i along that path
- $c_0 \models \mathrm{AF}^c\,\psi$ iff for all computation paths $\eta = c_0, s_0, c_1, s_1, \ldots$ we have $c_i \models \psi$ for some c_i along that path.
- $c_0 \models \mathrm{EF}^c\,\psi$ iff there exists a computation path $\eta = c_0, s_0, c_1, s_1, \ldots$ such that $c_i \models \psi$ for some c_i along that path.

- $c_0 \models \mathrm{A}^c(\psi_1 \mathrm{U} \psi_2)$ iff for all computation paths $\eta = c_0, s_0, c_1, s_1, \ldots$ we have $c_i \models \psi_2$ for some c_i along that path, and for each $j < i$, $c_j \models \psi_1$.
- $c_0 \models \mathrm{E}^c(\psi_1 \mathrm{U} \psi_2)$ iff there exists a computation path $\eta = c_0, s_0, c_1, s_1, \ldots$ such that $c_i \models \psi_2$ for some c_i along that path, and for each $j < i$, $c_j \models \psi_1$.

4.3.2 Expressing AG^c and EG^c

In [7], it is shown that $\mathrm{EG}\,\psi$ in CTL can be characterized as the greatest fixed point of the equation: $\mathrm{EG}\,\psi = \nu x.\psi \wedge \langle - \rangle x$. This results in a labeling algorithm computing the set of states which satisfy $\mathrm{EG}\,\psi$ [15]: 1) Assume that each state satisfying ψ has been assigned the label ψ; 2) Assign *all* states which are labeled with ψ the label $\mathrm{EG}\,\psi$; 3) Repeat until no change: Remove the label $\mathrm{EG}\,\psi$ from a state if it has *no* successors labeled $\mathrm{EG}\,\psi$.

We can express $\mathrm{EG}^c\,\psi$ in $\mathbf{EATL}_C$ in the same fashion. The only interesting question is what are suitable notions of *next states* in our model. Since an ADBMS alternates between environment and system states, the next state of some environment state c is actually a *system* state s. However, from the environments point of view this system state s will determine the next environment state c'. Our adapted CTL operators think of the sequence $c \to s \to c'$ as a single transition $c \to c'$ by hiding the interaction with the system. This opens up several interpretations of hiding. We chose the most sensible, conservative one, which treats the hidden activity as demonic non-determinism (if some interaction with the system causes the specification to fail, then our model check will fail).

Demonic non-determinism is only one possible and extreme interpretation of "possible next environment states". Complete abstraction of data of the underlying DBMS is the corresponding extreme on the date side. Obviously, there is a range of abstract interpretations of these notion and their actual choice would depend on the concrete ADBMS and the abstract interpretations performed on data in the underlying DBMS. One could begin testing with this extreme case and if verification fails one would make the interpretations more concrete until the verification succeeds or an actual design flaw has been found.

Example 23 Recall that $\mathrm{EG}^c\,\psi$ requires ψ to hold at every state on *at least one* computation path. Now suppose $c \models \mathrm{EG}^c\,\psi$ holds. Then by definition we *require* that ψ holds at c and $\mathrm{EG}^c\,\psi$ hold at one of its possible *environment*

successors. Clearly, this requirement would be satisfied if for some system state s, where $c \xrightarrow{r} s$ for some $r \in \mathcal{R}$, the $\mathbf{EATL}_S$ formula:

- $\phi = (\mathtt{tt})\psi$ holds at state s, if $r \in \mathbf{R_2}$, *or*
- $\phi = (\mathtt{tt})\psi \vee (\mathtt{ff})\psi$ holds at state s, if $r \in \mathbf{R_1}$.

Thus, using what we know about $\mathrm{EG}\,\psi$ in CTL, we arrive at the following equation:

$$\mathrm{EG}^c\,\psi = \nu x.\psi \wedge [\langle \mathbf{R_1} \rangle((\mathtt{tt})x \vee (\mathtt{ff})x) \vee \langle \mathbf{R_2} \rangle((\mathtt{tt})x)].$$

We can iteratively compute this set in a fashion similar to $\mathrm{EG}\,\psi$ in CTL: 1) Assume that each *environment* state c, satisfying ψ, has been assigned the label ψ; 2) Assign *all* environment states in $\mathcal{C}$ which satisfy ψ the label $\mathrm{EG}^c\,\psi$; 3) Repeat until no change: Remove the label $\mathrm{EG}^c\,\psi$ from an environment state if none of its system successors' successors are labeled ψ.

For all practical purposes, this algorithm works just like the associated algorithm for computing $\mathrm{EG}\,\psi$ in CTL. However, since our model of rule processing alternates between environment and system states, we must reference an environment state's system state successor in order to examine (and possibly remove) the label at the next *environment* state. Additionally, the branching time mode at system states depends on whether it stems from an EA or an ECA rule. It should now be apparent that the time complexity of such algorithms will be quadratic in the number of environment states.

Next, we examine $\mathrm{AG}^c\,\psi$, which is similar to $\mathrm{EG}^c\,\psi$, except that it requires ψ to hold at every state along *all* possible computation paths. We express this operation with the following equation:

$$\mathrm{AG}^c\,\psi = \nu x.\psi \wedge [[\mathbf{R_1}]((\mathtt{tt})x \wedge (\mathtt{ff})x) \wedge [\mathbf{R_2}]((\mathtt{tt})x)].$$

Notice how this equation is much more restrictive that $\mathrm{EG}^c\,\psi$, but similar in structure; we only negated the notion of "there exists a next environment state". Figure 8 illustrates a model whose initial environment state satisfies $\mathrm{AG}^c\,\psi$. Certainly, this operation is quite important for checking properties we demand to hold throughout rule processing.

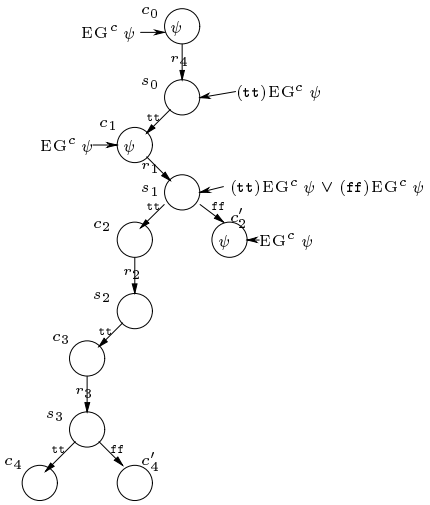

Figure 7: Model where $c_0 \models \mathrm{EG}^c\,\psi$.

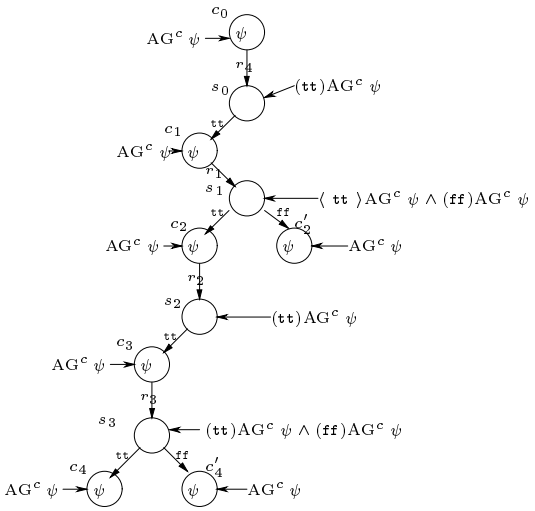

Figure 8: Model where $c_0 \models \mathrm{AG}^c\,\psi$.

4.3.3 Expressing AF^c and EF^c

Whereas AG^c and EG^c are used to check "invariant" properties that hold at every environment state along all or some paths, the branching-time operators AF^c and EF^c are used to check properties that we don't expect to hold for all states along a computation path, but do expect to hold eventually; perhaps only after a certain situation arises. Unlike the previous two branching-time operations, AF^c and EF^c are characterized as least fixed points [15]. Again, $\mathrm{AF}^c\,\psi$ expresses a more stringent property than $\mathrm{EF}^c\,\psi$. Consider the following fixed point characterization of $\mathrm{EF}^c\,\psi$:

$$\mathrm{EF}^c\,\psi = \mu x.\psi \vee [\langle \mathbf{R_1} \rangle(\mathtt{tt})x \vee (\mathtt{ff})x) \vee \langle \mathbf{R_2} \rangle((\mathtt{tt})x)].$$

Like $\mathrm{EF}\,\psi$ in CTL, the set of states satisfying $\mathrm{EF}^c\,\psi$ can be determined by the following algorithm [15]: 1) Assume that each *environment* state, c, satisfying ψ, has been assigned the label ψ; 2) For *all* states labeled ψ in step 1, assign these states the label $\mathrm{EF}^c\,\psi$; 3) Repeat until no change: *Assign* (i.e. add) the label $\mathrm{EF}^c\,\psi$ to any environment state if at least one of its *system* successors' successors are labeled ψ.

This labeling activity is characterized as a least fixed point: we iterate the labeling process beginning with no initial labels. The formula $\mathrm{AF}^c\,\psi$ is shown in the following equation:

$$\mathrm{AF}^c\,\psi = \mu x.\psi \vee [[\mathbf{R_1}]((\mathtt{tt})x \wedge (\mathtt{ff})x) \wedge [\mathbf{R_2}]((\mathtt{tt})x)].$$

By modifying step 3 of the algorithm for computing $\mathrm{EF}^c\,\psi$, we can compute the set of states satisfying $\mathrm{AF}^c\,\psi$. Step 3 would be changed so that we add the label ψ to an environment state only if *all* system successors' successors are labeled ψ. Figures 9 and 10 illustrate models whose initial environment states satisfy $\mathrm{EF}^c\,\psi$ and $\mathrm{AF}^c\,\psi$ respectively.

4.3.4 Expressing AU^c and EU^c

Often we need to express that some property ψ_1 should hold *until* rule processing reaches a point where another property ψ_2 is satisfied. The branching-time operation $\mathrm{E}^c(\psi_1\ \mathrm{U}\ \psi_2)$ specifies that, for at least one computation path, ψ_1 holds at the current state, and continues to hold, until rule processing reaches a state where ψ_2 holds. We characterize this formula with the following equation:

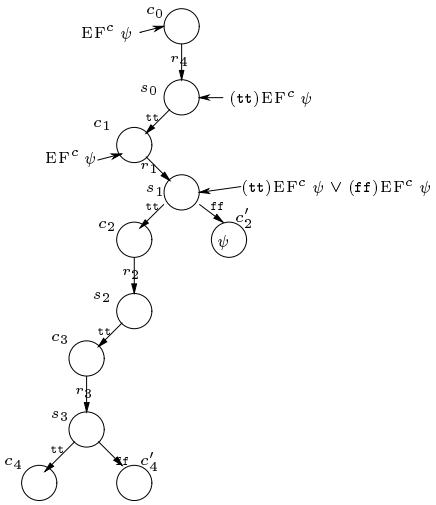

Figure 9: Model where $c_0 \models \mathrm{EF}^c\,\psi$.

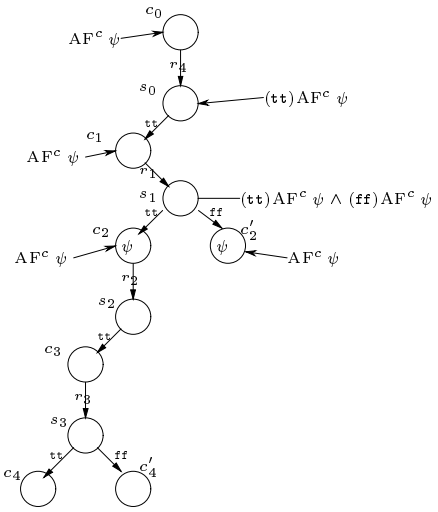

Figure 10: Model where $c_0 \models \mathrm{AF}^c\,\psi$.

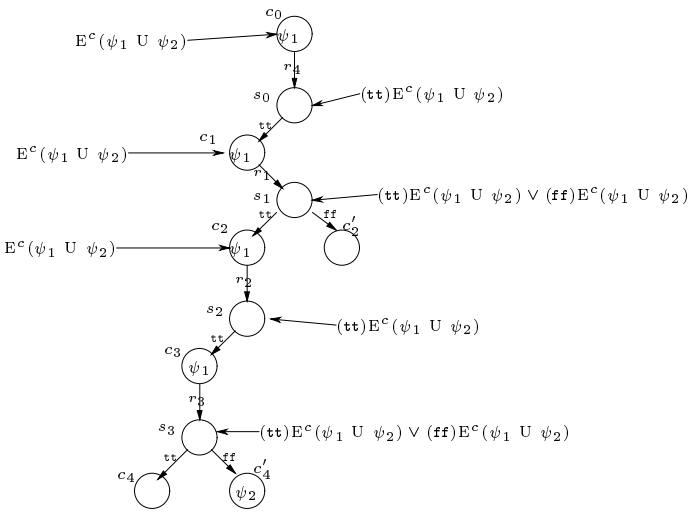

Figure 11: Model where $c_0 \models \mathrm{E}^c(\psi_1\ \mathrm{U}\ \psi_2)$.

$$\mathrm{E}^c(\psi_1\ \mathrm{U}\ \psi_2) = \mu x.(\psi_2 \vee (\psi_1 \wedge [\langle \mathbf{R_1} \rangle((\mathtt{tt})x \vee (\mathtt{ff})x) \vee \langle \mathbf{R_2} \rangle((\mathtt{tt})x)])).$$

Figure 11 shows a model whose initial environment state satisfies the formula $\mathrm{E}^c(\psi_1\ \mathrm{U}\ \psi_2)$. As before, the characterization of $\mathrm{A}^c(\psi_1\ \mathrm{U}\ \psi_2)$ is very similar to its sibling:

$$\mathrm{A}^c(\psi_1\ \mathrm{U}\ \psi_2) = \mu x.(\psi_2 \vee (\psi_1 \wedge [[\mathbf{R_1}]((\mathtt{tt})x \wedge (\mathtt{ff})x) \wedge [\mathbf{R_2}]((\mathtt{tt})x)])).$$

Figure 12 shows a model whose initial environment state satisfies the formula $\mathrm{A}^c(\psi_1\ \mathrm{U}\ \psi_2)$. Algorithms for computing the set of states satisfying these formulas could be handled iteratively, in the same fashion as previous branching-time operators. Table 24 and 25 summarizes the branching-time operations for both $\mathbf{EATL}_C$ and $\mathbf{EATL}_S$.

4.4 Practical Properties for our Starburst Model

We already pointed out that previous efforts at analyzing rule systems focused on determining whether properties such as termination, confluence and observable determinism are satisfied by a rule set. It was shown that these

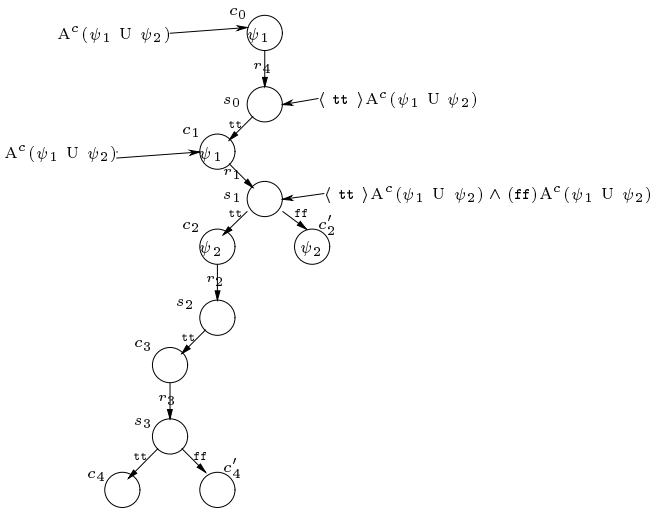

Figure 12: Model where $c_0 \models A^c(\psi_1\ U\ \psi_2)$.

$EG^c\,\psi = \nu x.\psi \wedge [\langle \mathbf{R_1} \rangle((\mathtt{tt})x \vee (\mathtt{ff})x) \vee \langle \mathbf{R_2} \rangle((\mathtt{tt})x)]$
$AG^c\,\psi = \nu x.\psi \wedge [[\mathbf{R_1}]((\mathtt{tt})x \wedge (\mathtt{ff})x) \wedge [\mathbf{R_2}]((\mathtt{tt})x)]$
$EF^c\,\psi = \mu x.\psi \vee [\langle \mathbf{R_1} \rangle(\mathtt{tt})x \vee (\mathtt{ff})x) \vee \langle \mathbf{R_2} \rangle((\mathtt{tt})x)]$
$AF^c\,\psi = \mu x.\psi \vee [[\mathbf{R_1}]((\mathtt{tt})x \wedge (\mathtt{ff})x) \wedge [\mathbf{R_2}]((\mathtt{tt})x)]$
$E^c(\psi_1\ U\ \psi_2) = \mu x.(\psi_2 \vee (\psi_1 \wedge [\langle \mathbf{R_1} \rangle((\mathtt{tt})x \vee (\mathtt{ff})x) \vee \langle \mathbf{R_2} \rangle((\mathtt{tt})x)]))$
$A^c(\psi_1\ U\ \psi_2) = \mu x.(\psi_2 \vee (\psi_1 \wedge [[\mathbf{R_1}]((\mathtt{tt})x \wedge (\mathtt{ff})x) \wedge [\mathbf{R_2}]((\mathtt{tt})x)]))$

Table 24: Branching time operators for $\mathbf{EATL}_C$.

$AG^s\,\phi = \nu x.\phi \wedge [(\mathtt{tt})[\mathbf{R_1}]x \wedge (\mathtt{ff})[\mathbf{R_1}]x) \wedge (\mathtt{tt})([\mathbf{R_2}]x)]$
$EG^s\,\phi = \nu x.\phi \wedge [(\mathtt{tt})\langle \mathbf{R_1} \rangle x \vee (\mathtt{ff})\langle \mathbf{R_1} \rangle x) \vee (\mathtt{tt})(\langle \mathbf{R_2} \rangle x)]$
$AF^s\,\phi = \mu x.\phi \vee [(\mathtt{tt})[\mathbf{R_1}]x \wedge (\mathtt{ff})[\mathbf{R_1}]x) \wedge (\mathtt{tt})([\mathbf{R_2}]x)]$
$EF^s\,\phi = \mu x.\phi \vee [(\mathtt{tt})\langle \mathbf{R_1} \rangle x \vee (\mathtt{ff})\langle \mathbf{R_1} \rangle x) \vee (\mathtt{tt})(\langle \mathbf{R_2} \rangle x)]$
$E^s\ (\phi_1 U\ \phi_2) = \mu x.(\phi_2 \vee (\phi_1 \wedge [((\mathtt{tt})\langle \mathbf{R_1} \rangle x \vee (\mathtt{ff})\langle \mathbf{R_1} \rangle x) \vee (\mathtt{tt})(\langle \mathbf{R_2} \rangle x)]))$
$A^s\ (\phi_1 U\ \phi_2) = \mu x.(\phi_2 \vee (\phi_1 \wedge [((\mathtt{tt})[\mathbf{R_1}]x \wedge (\mathtt{ff})[\mathbf{R_1}]x) \wedge (\mathtt{tt})([\mathbf{R_2}]x)]))$

Table 25: Branching time operators for $\mathbf{EATL}_S$.

properties could not be decided in the general case (i.e. without abstractions) [1, 8, 12]. In the next example, we demonstrate how we can specify termination in $\mathbf{EATL}_C$.

Example 24 Recall from Table 21 that, at each environment state, the configuration at that state includes the current status of the set $\mathcal{R}_c$. Moreover, we explained that whenever $\mathcal{R}_c$ is empty at an environment state, it means rule processing has terminated. Therefore, the $\mathbf{EATL}_C$ formula

$$\psi_{\text{term}} \equiv \text{AF}^c\,(\texttt{consideration set is empty})$$

specifies that rule processing is *guaranteed* to terminate, for it expresses that "*for* all *computation paths beginning in some environment state c there is some future environment state, where the consideration set is empty.*".

In our model from Figure 5 we have $c_0 \models \psi_{\text{term}}$ since all paths eventually terminate.

As suggested in [1] the properties of confluence and observable determinism are quite difficult to establish. In fact, analyzing these properties requires a number of steps. In the next section, we examine a possible model checking scheme for handling these properties. For now we focus on specifying *deterministic ordering* of actions. The following example demonstrates how we may apply $\mathbf{EATL}_S$ to determine if it is possible for two different actions to execute in different orders.

Example 25 Given atomic propositions

$$\begin{aligned} p_1 &= \texttt{can execute update on emp(salary)} \\ p_2 &= \texttt{can execute a retrieve on emp(salary)} \end{aligned}$$

we consider the $\mathbf{EATL}_S$ formula

$$\phi_{\text{ord}} \equiv \neg(\text{EF}^s\,(p_1 \wedge \text{EF}^s\,(p_2)) \wedge (\text{EF}^s\,(p_2 \wedge \text{EF}^s\,(p_1))).$$

If there exists any state $s \in \mathcal{S}$ such that $s \not\models \phi_{\text{ord}}$ our rule set may not have deterministic ordering with respect to the actions specified by p_1 and p_2. Otherwise, if $s \models \phi_{\text{ord}}$ then there cannot be two execution paths $\eta_1 = \ldots, s_1, \ldots, s_2, \ldots$ and $\eta_2 = \ldots, s_1', \ldots, s_2', \ldots$ where $s_1 \models p_1$ and $s_2 \models p_2$, *and* $s_1' \models p_2$ and $s_2' \models p_1$. If there did exist two such execution paths, then, depending on how rules are processed, we may either see 1) the value

of **emp**(salary) *before* it is updated, or, 2) the value of **emp**(salary) *after* it is updated. Certainly, the result of each computation path could be different. However, it may also be the case that one of these executions is impossible or unlikely to occur during rule processing. This determination must be made interactively by the rule designer. In any case, the validity of $s \models \phi_{\text{ord}}$ *ensures* deterministic ordering with respect to propositions p_1 and p_2.

Example 25 also suggests an interesting aspect of **EATL**. Although we chose to analyze deterministic ordering in the context of $\mathbf{EATL}_S$, the mutual recursiveness of **EATL** allows us to express the same property written in $\mathbf{EATL}_C$. The formula $\psi \equiv [-]\phi$ could also be used to check for deterministic orderings but at the level of the environment; in addition it provides more flexibility. For instance, if we modify the formula above slightly, we could write $\psi \equiv [K]\phi$, where $K \subseteq \mathcal{R}$, in order to check this property for a subset of our rule set. This allows for more fine-grained and calculated analyzes.

Finally, we show how we might use **EATL** for checking certain rule behavior constraints we wish to enforce, such as the one given in Example 8.

Example 26 In Example 8, we stated that we were interested in showing that our rules behave in a manner such that, whenever an employee's rank is updated, then that employee's salary is also updated. Recall Figure 5, which depicts a model of rule processing that begins with an update to two employees' ranks. Then the $\mathbf{EATL}_C$ formula

$$\begin{aligned} \mathit{leadsto}(r_1, r_3) \equiv \mathrm{AG}^c\,(&\texttt{rule 1 is selected for consideration} \\ &\Rightarrow \mathrm{AF}^c\,(\texttt{rule 3 is selected for consideration}))\end{aligned}$$

mnemonically reads as "*At any environment state, if rule 1 is selected for consideration, then eventually rule* r_3 *is chosen for consideration.*".

Unfortunately, the check $c_0 \models \mathit{leadsto}(r_1, r_3)$ fails in the model of Figure 5. As a counterexample consider the path c_0, s_0, c_1, s_1, c_2'. Notice that r_1 is selected for consideration by Starburst's environment in state c_0. However, processing may terminate in state c_2' without ever considering r_3. Indeed, we have uncovered a subtle error in our rule design. If we refer back to the original rule definitions given in Section 2, 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 Example 7, 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	ECA definition
r_1	**ON** update to **emp**(rank) **IF** new **emp**(rank) < 5 **THEN** update **bonus**(raiseamnt) **PRECEDES** $\{r_5\}$
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** $\{r_1, r_5\}$
r_5	**ON** update to **emp**(rank) **IF** TRUE **THEN** update **emp**(salary)

Table 26: Corrected rule set used for our corporation's rule system.

rule set. A correct set of rules is given in Table 26. Notice that r_2 has been removed and replaced by r_5. Now an employee's salary is updated whenever their rank is updated. As with Figure 5, Figure 13 shows the model of rule processing for this set of rules (again, the initial event is an update to **emp**(rank)). Now, c_0 satisfies the rule behavior constraint given in Example 26. Although we discovered and corrected this error without automatic verification, it should be clear that, generally, this becomes impossible for more realistic examples.

5 ADBMS Model Checking

So far we have outlined all the necessary components for a verification tool capable of checking **EATL** formulas in our framework. However, growth in the popularity of Model Checking over the past decade has already led to the development of a number of formal verification tools. Thus, rather than constructing a new model checker, it seems advisable to implement our framework in an existing verification tool. For example, it should be relatively straightforward to implement the verification of **EATL** formulas in MOCHA [2] as soon as that tool has been built. Since SMV did not perform well in our setting, we turned to *Spin*, a verification tool developed

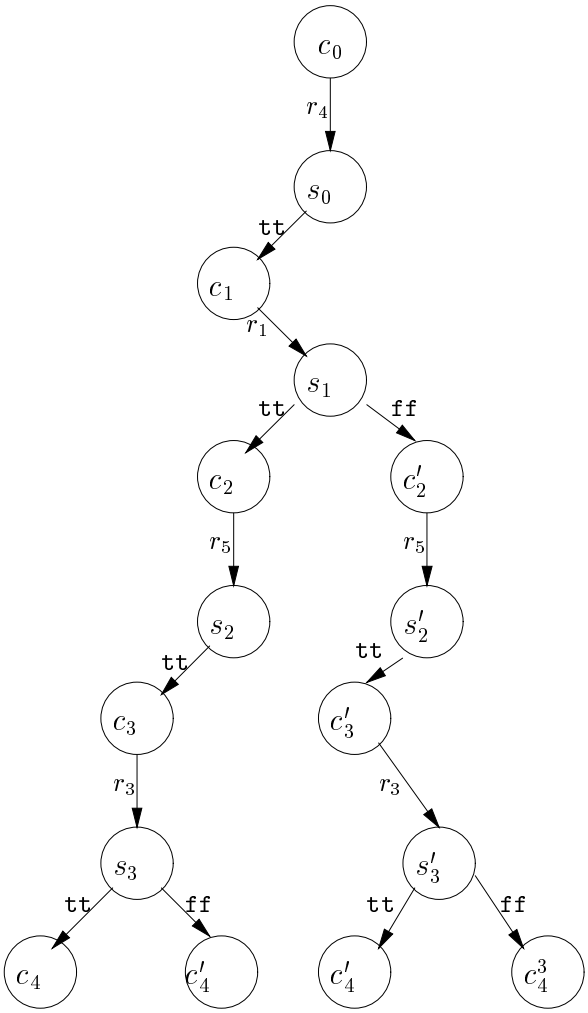

Figure 13: A complete model of our corrected rule set.

at Bell Laboratories [4]. The reader familiar with *Spin* surely realizes that its specification language LTL is a linear-time temporal logic and is therefore not suited for implementing **EATL**. Luckily, a variety of key properties, like termination and responsiveness, lie in the intersection of branching- and linear-time logics. Thus, we can employ *Spin* for those properties.

Spin provides its own description language, *Promela*, which specifies abstract models of systems. In this section, we show how the model of rule processing described in Section 3 can be written in *Promela* and analyzed by *Spin*. Specifically, we write a *Promela* specification for our running example from Section 2. Then, we show how *Spin* can be applied towards analyzing those properties of rule behavior important to ADBMS designers (see Subsection 2.5). In addition, we have developed a small, interactive environment for automatically generating *Promela* models of rule processing for different rule sets. This simple GUI application, written in Java, allows a user to define a set of rules and verify properties such as termination, deterministic ordering, and rule integrity constraints. Moreover, the user may modify the rule set by interactively adding and deleting rules, and save this rule set for future analysis. Our application generates Starburst rule processing models, but could be customized to generate *Promela* models reflecting the execution semantics of another rule system.

Furthermore, the interface itself can be used as a front end for generating rule processing models implemented with another Model Checking tool. For example, we tried to model and analyze rule processing using SMV [15]. SMV represents the state space of a model symbolically using *boolean decision diagrams* (BDDs) [6, 15] allowing for extremely compact representations of systems. In spite of all this, SMV did not perform well when applied towards modeling and analyzing of rule behavior. One reason for this was SMV's description language (see [15]), which did not allow us to easily model certain aspects of rule behavior, such as non-deterministic selection of rules for consideration. More specifically, SMV's description language, unlike *Promela*, has only a deterministic `case`-statement. Another factor might be the fact that SMV uses a state-based framework. Such a framework made it difficult to naturally model the interaction between an ADBMS's environment and system. In the end, even our simple four rule example could not be efficiently modeled and analyzed with SMV. Taking efficiency aside, we were able to automatically generate SMV models of Starburst rule processing under our interface. This suggests that it is possible to construct a single interface under which a variety of ADBMSs can be modeled using a variety of different

modeling languages. The implementation details of our application are beyond the scope of this paper. Our motivation for building this tool was to, hopefully, corroborate the practicality of Model Checking in the context of ADBMSs.

5.1 ADBMS Model Checking Environment

We begin with an overview of our proposed interactive ADBMS Model Checking environment. Figure 14 illustrates a high-level architectural description of our approach. The boxes in this figure represent functional components of the Model Checking environment, and arrows represent interaction between components. Components within the dashed box belong to or are implemented by the Model Checking application itself. In our case, this will be *Spin*, but it could easily be a different Model Checking tool (for instance, a tool custom built for Model Checking **EATL** formulas) capable of providing the functionality necessary to implement our framework. The user interface is not part of the Model Checking system, but offers its users a convenient way of interacting with it. For example, our Java GUI takes rule information and specifications provided by the user and produces a Starburst rule processing model specified in *Promela*. Finally, the Model Checking system must convey meaningful results back to the user. *Spin*'s countertrace facility, which produces a graphical counterexample whenever a property is not satisfied, is an excellent example of this type of information useful for analyzing rule behavior. Of course, *Spin* will return only *one* possible countertrace so the designer has to express the "essence" of that failure as a filter (a formula of the specification logic) which weeds out unrealistic computation paths and required to verify the model anew. Thus, debugging can be seen as an incremental testing scheme.

At this point, let us focus on the functional components within the Model Checking system, which include: rule information, environment routines (simulating the environment states of our ADBMS), system routines (to simulate the system states of our concrete ADBMS), and the *Spin* tool. The first three items are implemented by our *Promela* programs. We discuss this encoding as well as the purpose of each component by means of our running example (the complete *Promela* specification for Example 7, produced by our *Promela* code generator, can be found in Appendix A). Then, we must supply *Spin* with information about the properties we wish to analyze; those can be expressed directly in our *Promela* program. Ultimately, Spin takes

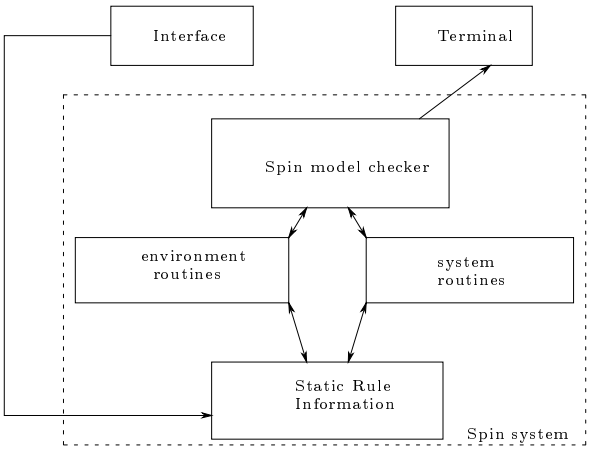

Figure 14: High-level view of our ADBMS Model Checking environment.

the *Promela* specification of our rule processing model and produces a C program which, when executed, verifies whether the properties we desire to check hold in this model. In many respects, *Promela*'s syntax is somewhat reminiscent of C. Consequently, this section assumes the reader has some familiarity or intuitive understanding of C, *Spin*, and *Promela*. A more detailed explanation of *Spin* and *Promela* can be found at the *Spin* website [4].

5.2 Representation of Rules, States, and Transitions

In many respects, *Promela* provides much of the flexibility and power of any ordinary programming language. For instance, *Promela* provides syntactic constructs for declaring structures (such as records and arrays), global and local variables (with the usual scoping rules), and *processes* (which can examine and modify variables). In addition, *Promela* provides a mechanism for defining *message passing channels*, which enables synchronous or asynchronous communication between two processes. Each of these constructs plays an integral part in implementing our rule processing framework. In particular, we use these constructs to define representations for our model's ECA rules (including the events, tables, and fields which make up these rules), environment and system states, environment and system state transitions, consideration

set, and initial rule triggering transactions. The global variable declarations for the *Promela* specification of Example 7 are illustrated in Figure 15. We examine the purpose of these variables in the following subsections.

5.3 Rule Information

As indicated in Figure 15, we can use the `typedef` construct to define a record `ecarule` which stores a variety of information about each of our four ECA rules. The first three fields of this record store the event (plus the associated table and field) that cause the rule to be triggered. The fourth, fifth, and sixth fields keep the rule's action (including its associated table and field) that takes place if Starburst's system chooses to execute the rule. Of course, an EA rule should always be executed. Hence we keep a variable `rtype` in our `ecarule` structure which indicates whether a particular rule is an EA or ECA rule. Finally, we define an array of these records, `rules`, which enables us to directly access each instance of our four rules whenever necessary. The `rules` array is set to size `N`, where `N` is a constant indicating the number of rules we are modeling.

Next, we must assign `rules` values representing the ECA rule definitions given in Table 11. For that, it is necessary to explicitly indicate which events and actions are allowable in our model. In *Promela* the `mtype` construct can be used to define symbolic values which, in our case, are simply enumerations representing the range of different events and actions our four rules might consist of. The first six fields in `ecarule` are of type `mtype`, and, thus, must be assigned one of these values. We do this by initializing each rule in `rules` as shown in Figure 16. Notice that, after initialization, `rules` essentially becomes a representation of Table 11. Note that the range of `rules` is `0..3`. Therefore, each rule r_i in Table 11 is stored in `rules[i-1]`.

5.4 The Consideration Set

Recall from Section 2 that rule processing begins after a user transaction stimulates some rule triggering event. In Example 7, this initial transaction was an update to Matt's and Jasmine's ranks in the **emp** table. In our *Promela* model, we represent this initial transaction by initializing the variables `i_event`, `i_table`, and `i_field` (each having type `mtype`) to the values `update`, `emp`, and `rank` respectively, as shown in Figure 15. Also recall that this initial transaction triggered two rules r_1 and r_4 for consideration.

```
#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*/
```

Figure 15: Global declarations in our *Promela* 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;
⋮
```

Figure 16: Initialization of our four ECA rules.

Consequently, they were added to $\mathcal{R}_c$, the *consideration set*. Our global declarations from Figure 15 indicate that $\mathcal{R}_c$ is represented as the boolean array `c` in our *Promela* model. When a rule r_i is triggered, we represent its insertion into $\mathcal{R}_c$ by setting the corresponding bit in `c` (i.e. `c[i-1]`) to `1`. Likewise, when a rule is selected by Starburst's environment and passed to its system for evaluation and (possible) execution, it is removed from $\mathcal{R}_c$ by setting `c[i-1]` back to `0`.

Every legal *Promela* program must consist of at least one main process, `init()`. This process can initialize any global variables and instantiate other processes. This fact is particularly important since Starburst's environment and system are modeled as two synchronous processes that will, ultimately, be instantiated by `init()`. We examine this protocol in more detail later on. For now, let us consider the initialization of our consideration set. Figure 17 illustrates the rest of the `init()` process which indicates how this initialization can be handled in *Promela*.

Figure 17 introduces two important control flow statements that are an integral part of any *Promela* model. Because the semantics of these structures in *Promela* is inherently different from their semantics in traditional programming languages, they deserve to be explained in more detail. The `do` structure allows us to simulate repetition in our model. Within the structure itself we can have multiple execution sequences with each sequence being preceded by a "`::`". In the case of Figure 17 our `do` structure contains two *guarded* sequences. The guard is the expression appearing on the left-hand side of the `->`. When a guard is evaluated, the statement(s) on the right-hand side of the `->` is only executed if the guard evaluates to true. Notice that the guarded statements for the `do` loop in Figure 17 can be selected deterministically, since `temp` must either be `< N` or `>= N`. In general, if more than one guarded sequence is true (and, thus, executable at the same time), *Spin executes each sequence as a separate, possible computation path.* Consequently, *Spin* performs an exhaustive search by checking *all* possible computation paths.

Figure 17 also shows that we are able to nest structures in *Promela*. For instance, the first guarded sequence in the `do` loop contains an `if` (i.e. selection) structure. The semantics of this selection structure is similar to that of the repetition structure. As before, a guarded statement can be executed only if its guard evaluates to true. When a sequence is executed, the selection structure is exited (and, in our case, control reverts back to the `do` loop); but when *none* of the statements in a selection structure are executable the pro-

```
:
temp=0;

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*/

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

Figure 17: Initialization of the consideration set.

temp	associated rule	action
0	r_1	c[0]=1; temp=temp+1
1	r_2	temp=temp+1
2	r_3	temp=temp+1
3	r_4	c[3]=1; temp=temp+1
4	N/A	break (exit do loop)

Table 27: Initialization of the consideration set.

cess *blocks* until one of the guarded sequences within the structure becomes true. Since this may *never* happen, it is possible for a process to block forever. There are two solutions to this potential problem. First, we can write our guarded sequences in a manner that ensures deterministic behavior, as we did in our `do` loop. Another option is to introduce the special guarded sequence, `else`. The `else` option becomes true (i.e. executable) if and only if *all* of the other guarded sequences in the same structure cannot be executed. Thus, in Figure 17, when the first (and only) guarded sequence in the `if` structure is false, `else` becomes true and `temp` is incremented.

Returning to the initialization of our consideration set, we begin by setting a loop counter variable, `temp`, to `0`. Next, the `do` loop is entered. Then, for *each* of our four rules, the code checks `rules` to see if the initial event "matches" the event which triggers that rule. If it does, we set that rule's corresponding bit in `c` to `1` *and increment the loop counter*. Otherwise, the corresponding bit is left set to `0`. When `temp` becomes `4` the second guarded sequence in the `do` loop becomes true and control exits the loop. The following example demonstrates this process.

Example 27 In Example 7, the initial user transaction triggered two rules, r_1 and r_4. As a result, we stated that these rules should be added to $\mathcal{R}_c$. In terms of our *Promela* model $\mathcal{R}_c$ is represented by the boolean array `c`. The value of `c` is determined by iteratively checking whether or not r_i's associated bit should be set to `1`. This iterative process is summarized in Table 27. Pictorially, after initialization, `c` would appear as shown in Figure 18.

Our code initializes the boolean variable `done` to `0`. Only when rule processing terminates (i.e. $\mathcal{R}_c$ becomes empty), will `done` be reset to `1`. This fact is particularly important since the problem of checking for termination of rule processing has now been reduced to simply checking if `done` eventually becomes `1`. After these initializations the ADBMS has entered envi-

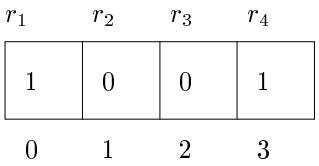

Figure 18: Representation of initial $\mathcal{R}_c$ for Example 7 in *Promela*.

ronment state c_0 where rule processing initially begins. The `init()` process next instantiates two new processes: `environment()` and `system()`. The `atomic` statement indicates that these two processes are to be executed in a non-interleaved fashion. This notion accurately reflects the semantics of our model since we can think of rule processing as synchronized communication between an ADBMS's environment and its system.

5.5 States and State Transitions

In our *Promela* specification, we model Starburst's environment and system as two synchronous processes: `environment()` and `system()`. State transitions are modeled as single buffer communication channels. Our global declarations in Figure 15 indicate two such channels: `selected` and `action`. When the environment process selects a rule for consideration, it sends the number of the rule chosen to the system via the channel `selected`. This represents a transition $c \xrightarrow{r_i} s$ from an environment state to a system state. Similarly, the system may or may not choose to execute a selected rule's action. It communicates this information (along with information about the rule's action in the case that it is executed) by way of the `action` channel. Depending on how the system chooses, this communication represents an $s \xrightarrow{\texttt{tt}} c$ or $s \xrightarrow{\texttt{ff}} c'$ transition from a system state back to an environment state. We can ensure that rule processing proceeds in a strictly alternating fashion by imposing the following restrictions in our *Promela* model:

- When the environment process selects a rule for consideration and sends it to the system process, it must *wait* for the system process to communicate its corresponding action.
- When the system process communicates its action (with respect to a selected rule) back to the system, it must *wait* for the environment

process to choose the *next* rule for consideration and send it to the system.

Figures 19 and 20 show how our environment and system processes are implemented in *Promela*. Let us see how these processes implement a strict alternation of environment and system moves. The system process is essentially in an infinite `do` loop which must wait for a message to be sent by the environment. The first statement in this `do` loop, `selected?rulenum`, *receives* the number of the rule selected for consideration by the environment from channel `selected` and stores it in `rulenum`. In *Promela* receive statements are only executable if a message *actually exists* in the associated channel. If there is no message in the channel, the process must wait for a message to arrive. Therefore, `system()` will never be allowed to proceed until *after* `environment()` selects the first rule for consideration and sends the rule number, via `selected`, to the system. This fact is quite important because it ensures that the `environment` *always proceeds first* when rule processing begins (i.e. when `init()` initially instantiates `system()` and `environment()`.

5.5.1 The Environment Process

The environment process also executes a `do` loop which will be exited only after rule processing terminates (more on this later). The first selection structure in `environment()` is executed immediately after this process is instantiated and models how Starburst selects a rule for consideration. We demonstrate this protocol in the next example.

Example 28 Recall that initially r_1 and r_4 are in $\mathcal{R}_c$. Starburst's environment must select one of these rules for consideration and pass it to the system for possible execution. When the environment process begins its execution, the first `if` structure is evaluated, and one of its guarded statements executed. Table 28 summarizes how *Spin* determines which statement to execute.

As indicated in this table, only the sequence guarded by `c[3]==1` is allowed to execute. This sequence removes r_4 from the consideration set (i.e. `c[3]=0`) and sends the number of the rule selected to the system process via channel `selected` (i.e. `selected!3`). Hence, we enforce the fact that r_4 has a higher priority than r_1 by specifying that r_1 can only be selected if `c[3]==0` (i.e. r_1 can only be selected if r_4 is *not* in the consideration set). This demonstrates how easy it is to model Starburst's conflict resolution

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

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

Figure 19: Environment process in our *Promela* model.

```
  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
 }
```

Figure 20: System process in our *Promela* model.

guard	value	action
`c[0]==1 && c[3]==0`	`0` (false)	N/A
`c[1]==1`	`0` (false)	N/A
`c[2]==1`	`0` (false)	N/A
`c[3]==1`	`1` (true)	`c[3]=0; selected!3`
`else`	`0` (false)	N/A

Table 28: Summary of rule selection from $\mathcal{R}_c$ (partial prioritization) .

guard	value	action
`c[0]==1`	`1` (true)	`c[1]=0; selected!1`
`c[1]==1`	`0` (false)	N/A
`c[2]==1`	`0` (false)	N/A
`c[3]==1`	`1` (true)	`c[3]=0; selected!3`
`else`	`0` (false)	N/A

Table 29: Summary of rule selection from $\mathcal{R}_c$ (no prioritization).

policy in *Promela*. Table 29 summarizes the effects of r_1 and r_4 having the *same* priority on *Spin*'s determination of which statement to execute. Now *two* guarded sequences are executable. As mentioned in Subsection 5.4, *Spin verifies both possibilities*. In this fashion, *Spin* checks *all possible execution paths* when performing verification. This could not be done in SMV since its `case`-statement is deterministic.

Once a rule is selected and sent to the system process, the `if` structure is exited and control passes to the statement following this structure. According to Figure 19 this would be the statement `action?event,table,field,fire` implying that it is the system's turn to execute since `environment()` must wait for `system()` to make a decision regarding the rule currently under consideration (r_4) and send it back to the environment via channel `action`.

5.5.2 The System Process

In Example 28 we indicated that the environment chose r_4 and sent its corresponding rule number (i.e. `3`) to the system via channel `selected`. At this point the statement `selected?rulenum` becomes executable. Thus, the system process receives the value `3` from this channel and store it in the variable `rulenum`. Referring to Figure 20, `system()` accesses `rules` to determine what action is associated with this rule.

Finally, the system must decide whether or not it should execute the rule's action. This determination is handled by a nested selection structure. If an EA rule is under consideration (i.e. `rtype==0`), then the system executes the statement `action!event,table,field,1` where `1` indicates that the rule's action was executed. If an ECA rule is under consideration (i.e. `rtype==1`), `system()` non-deterministically chooses to execute either:

- `action!event,table,field,0` (do not execute this rule's action), or

guard	value	action
N/A	N/A	`event=rules[3].actexec`
N/A	N/A	`table=rules[3].acttable`
N/A	N/A	`field=rules[3].actfield`
`rules[3].rtype==0`	`1` (true)	`action!retreive,emp,all,1`

Table 30: Summary of system process upon receiving r_4.

- `action!event,table,field,1` (execute this rule's action).

As before, when performing verification, *Spin* actually analyzes *both* situations. The following example illustrates this entire process.

Example 29 Let us go back to the point where `system()` receives the rule selected by `environment()` for consideration. Table 30 summarizes which statements are subsequently executed by `system()`. As shown in this table, the system process sends a message back to the environment process indicating r_4's action, and the fact that this action was executed.

Upon completing this process, `system()` returns back to the top of the `do` loop. There it waits for the next message to be sent by `environment()`. Realize, however, that if rule processing terminates, `system()` is left waiting forever. We handle this problem by preceding the `do` loop with an `end:` label. This label alerts the *Spin* verifier that we expected this situation to arise and that it should not report an error.

5.5.3 The Environment Process revisited

Recall that `environment()` was awaiting `system()`'s decision regarding rule r_4. The statement `action?event,table,field,fire` takes the message sent by `system()` and stores the actual values (i.e. `retrieve`, `emp`, `all`, `1`) in the corresponding variables. Next, the environment must determine which new rules (if any) r_4's action triggered. As shown in Figure 19 this is handled in almost exactly the same manner as demonstrated in Example 27. The only difference is that we must make an additional check to see if the system actually executed the previous rule's action. If it did not, then clearly no new rules will be triggered. In our case, r_4's action was executed. However, recall from Example 7, that its action did not trigger any new rules. Therefore, the next rule which is chosen for consideration is r_1. The process described

in the three preceding subsections is iterative and continues until all the bits in `c` become `0`. When this happens the environment won't have any rules to select for consideration. Thus, the `else` statement becomes executable. As we alluded to earlier, `done=1` indicates rule processing terminates and `break` forces the environment process to exit its `do` loop.

5.6 ADBMS Verification using *Spin*

The *Promela* program presented above describes a model of rule processing for Example 7. In order to model a different set of rules, we would need to write another *Promela* program for that particular rule set. Clearly, this can become quite inconvenient, especially since it is often the case that we wish to examine the effect that simple changes to our rule set have on rule behavior (such as the effect of adding a particular rule to a rule set on rule behavior). Fortunately, we can easily isolate patterns in our *Promela* program. This makes it feasible to automatically generate *Promela* models of rule processing. For this reason, we have developed an interactive tool for generating *Promela* rule processing models for the Starburst rule system. This tool was primarily an experimental effort; however, we hope that our endeavors motivate future, more dedicated efforts along these lines. 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.

5.6.1 Termination

Checking that a given set of rules is guaranteed to terminate is surprisingly simple with *Spin*. We check for termination by including the `progress:` label in the guarded statement `else -> progress: done==1; break`. Recall that this statement is only executable if all of the bits in the boolean array `c` are set to `0`; but this means that $\mathcal{R}_c$ is empty and rule processing has terminated. If rule processing *never* terminates, `c` always has at least one bit set to `1`. Consequently, if rule processing never terminates, the `else` statement is never executed and the `progress:` label is never reached! We can direct *Spin* to report an error whenever this situation arises. Such an error would indicate that rule processing *may not* terminate. On the other hand, if *Spin* reports no errors, then rule processing is guaranteed to terminate.

It is important to keep in mind that no matter which rule behavior properties we analyze only *positive* results obtained during verification are conclusive. For instance, we can be assured of a terminating rule set provided that *Spin* reports no errors. An error report, however, merely indicates that rule processing *may not* terminate. If this occurs it is up to the rule designer to interactively evaluate the results returned by *Spin* (such as the counterexample produced) and draw his or her own conclusions. As suggested in [1], it may very well be the case that errors reported by verification are cases that are unlikely or never occur in the actual rule system.

5.6.2 Rule Integrity Constraints

Determining whether the consideration of one rule eventually leads to consideration of another rule is also quite simple in *Spin*. We handle this using the LTL formula `[](c[i]==1 -> <>(c[j]==1))` where `i,j` $<$ `N`. Mnemonically, this formula reads "*For all paths, it is always the case that whenever* r_i *is under consideration, then* r_j *is eventually under consideration.*". We can save this formula in a file and import it directly into our *Promela* model using the `#include` command, as shown in the model of Example 7 given in Appendix A.

5.6.3 Confluence

In [1] it is suggested that checking confluence for a rule set is a difficult problem at best. Nevertheless, [1] provides a number of useful theorems and algorithms which, ultimately, can aid rule designers in their efforts to achieve a confluent rule set. Specifically, they propose a *confluence requirement* which, if satisfied, guarantees that a given rule set is confluent. Additionally, they provide an algorithm for determining if a set of rules satisfies that confluence requirement. This algorithm takes as input two rules, r_i and r_j, such that both rules have the same priority. The output of this algorithm is two *sets* of rules, $\mathcal{R}_1$ and $\mathcal{R}_2$. Then, in order to meet the confluence requirement, it is necessary to show that for *each* pair of rules $r \in \mathcal{R}_1$ and $r' \in \mathcal{R}_2$, r and r' *commute* (we discuss how to determine if two rules commute in a moment). The difficulty in applying this algorithm comes from the fact that it must be performed for *all* rule pairs, r_i and r_j, such that both rules have the same priority. As pointed out in [1], this is an extremely conservative approach and could be made more efficient by eliminating "unreasonable" cases with

additional analysis. Consequently, we believe rule analysis in our framework can be effectively used in conjunction with the confluence requirement algorithm by isolating certain rule pairs for which it is not even necessary to apply the confluence requirement algorithm, and providing a "tool box" of specifications for determining whether a pair of rules commute.

5.6.4 A Tool Box of Properties

As in [1], confluence analysis under our framework is a step-by-step, user interactive verification process. We supply the rule designer with a "tool box" of useful specifications which aid him or her in this process. These specifications can then be used to draw conclusions regarding a rule set's satisfaction of the confluence requirement (or lack thereof).

We first address the problem of eliminating certain rule pairs for which it is not necessary to apply the confluence requirement algorithm. In [1], they observe that in actuality it is *not* necessary to apply this algorithm to all pairs. As mentioned earlier, their algorithm constitutes an extremely conservative approach. In fact, the algorithm need only be applied to all rule pairs, r_i and r_j, such that both rules have the same priority *and both rules are under consideration at the same time*. In terms of our framework, this would mean that we must only consider pairs of rules that are under consideration together at some environment state, $c \in \mathcal{C}$. We can easily isolate such rules in our *Promela* models by checking our boolean array `c` each time we enter the environment process. It would then be up to the user to determine if the two rules have the same priority. For instance, in Example 7, r_1 and r_4 were both under consideration at the same time. However, since r_4 has a higher priority than r_1, we would simply ignore this situation. Moreover, this property is expressible in **EATL**. Eliminating rule pairs in such a systematic fashion could make confluence analysis yield more accurate results and make confluence analysis efforts more effective.

In [1] a number of sufficient conditions for the *non-commutativity* of rules are given. Specifically, rules r_i and r_j *may* be non-commutative if *any* of the following hold (if *none* of these hold, then r_i and r_j do commute):

1. r_i's consideration can eventually lead to r_j's consideration,
2. r_i's actions can affect what r_j's reads (i.e. r_j's data retrieval),
3. r_i's insertion actions can affect r_j's updates or deletions,

4. r_i's updates can affect r_j's updates, and

5. any of the above when we reverse r_i and r_j.

We are currently working on developing a tool box of specifications for analyzing such properties. We believe that these properties can be conveniently expressed in LTL and effectively analyzed by *Spin*.

5.6.5 Results of our ADBMS Verification

We analyzed termination, rule integrity constraints, and deterministic ordering properties for a set of twenty ECA rules. In order to test the accuracy of our model we seeded it with various violations to these properties. In all cases, *Spin* flagged these errors and provided a visual counterexample of where the property failed. As suggested previously, *Spin*'s countertrace facility is extremely informative. For instance, in the case of non-termination, *Spin* flagged the violation and showed exactly which rule caused it. In addition, along with the countertrace, *Spin* provides verbose details of the *Promela* model's execution. Since this feature displays the values of each variable during the run of the program, it can be used for interactive debugging.

As expected, the amount of time and memory needed to perform verification appears to be directly dependent on the amount of non-determinism we introduced in our model. Initially we designed our twenty rules to trigger in a "chain-like" fashion: $r_1 \rightarrow r_2 \rightarrow \ldots \rightarrow r_{20}$. When checking for termination, our model required about 5,000 states. We then changed the first ten rules to EA rules. Of course this makes our model more *deterministic* since for these ten rules, there is only one possible execution of their actions. As expected, the size of the state space decreased to about 9,000 states.

As a final test, we introduced *more* non-determinism by decomposing our chain structure. This meant that at any given time during rule processing it was likely that two or more (non-prioritized) rules would be under consideration at the same time. The effect of this non-determinism caused the state space to grow quite dramatically. In one case the size of the state space reached 215,000 states. This effect clearly indicates that, as with many systems, ours is susceptible to the "state-explosion" problem [10]. Upon prioritizing these rules (i.e. reducing the amount of non-determinism) the state space decreased considerably.

Our current analysis efforts are focused on studying the scalability of our Model Checking scheme. In particular, we wish to determine how large of a rule set we can reasonably analyze. We speculate the answer to this question is not entirely conclusive. Clearly, our current results show that the number of rules we can analyze is a function of those rule's interdependency. Still, there are a variety of ways to overcome the problem of Model Checking larger rule sets. For example, [1] suggests that it often the case that large rule sets 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. Such an approach would also be incremental. Thus, if a rule is added to one group, analysis would need to be repeated only for that group. Ultimately, we feel benefits such as these make Model Checking a suitable and effective approach to analyzing rule behavior.

6 Conclusions

The purpose of this paper was to propose a framework for analyzing rule behavior in active database management systems. As our examples have shown, analysis of rule behavior can be quite difficult, even in seemingly simple situations. To make matters worse, many of the properties that rule system designers are interested in are undecidable in the general case. Nevertheless, our work indicates that it is possible to develop methods and tools which can, at the very least, aid designers in their efforts. Specifically, we suggested a Model Checking methodology as a feasible, practical, and effective approach for analyzing rule behavior. Historically, this approach has been directed towards verifying concurrent, reactive systems. We have modeled ADBMSs as such reactive systems and their subtle behaviors are reminiscent of those which occur in the design and implementation of concurrent systems. In order to apply Model Checking to our problem, it was first necessary to design a framework that allowed us to model an ADBMS in a finite number of states. We believe our model of rule processing presented in Section 3 is an effective solution to this problem. In particular, we were able to abstract details about data stored in an ADBMS's database at an adequate level and focus our efforts towards analyzing the behavior of the rules themselves. By separating decisions made by an ADBMS's environment and its underlying system our methodology provides a generic framework for modeling any

ADBMS, regardless of its operational semantics. By considering all possible rule processing scenarios, our model accurately reflects anything that may occur in the actual rule system.

Moreoever, we have examined the possibility of applying existing Model Checking tools. In addition to applying *Spin* towards this effort, we have examined the performance of other model checkers under our general architecture. Specifically, we attempted to use the Symbolic Model Verifier (SMV) [15]. We found that SMV's input language did not easily lend itself towards modeling such things as rule priorities. Thus, we were unable to effectively model Starburst's conflict resolution policy in SMV. Still, we examined SMV's performance at verifying models of non-prioritized rule sets. In this case, SMV's performance degraded noticeably as the number of rules increased.

In the future, we plan to examine a number of possible improvements to our framework. In addition to our current work mentioned at the end of Section 5 we would like to add more detail to our *Promela* models in an effort to obtain more fine-grained rule analysis. For instance, by adding an additional selection structure in our environment process, we can easily simulate rule *untriggering*. Such a feature would indicate that it is possible for one rule's action to cause another rule, previously in $\mathcal{R}_c$ to be removed from $\mathcal{R}_c$ before it is ever selected for consideration.

We are also interested in exploring techniques for combating the observed state-explosion, caused by increased rule interdependency and nondeterminism. Along these lines, we examine the possibility of filter-based refinement [13], to "filter out" or eliminate unlikely or impossible computation paths.

While GUIs are fashionable at the time, there are several good reasons for having a *domain-specific language* (DSL) for formulating rule systems. For example, if a rule system is represented as a program then one can use conventional tools of program analysis such as partial evaluation and abstract interpretation. Ideally, such a DSL would be based on several actual RDLs in order to allow simple translations of real rule systems into such a DSL.

Finally, we wish to provide additional enhancements to our user interface. Such enhancements would include, among other things, more efficient code generation, additional options which allow users to easily define their own, customized specifications, and a standard API for creating "libraries" of different ADBMS environments.

A Promela Specification of Example 7

```
#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*/

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;

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*/

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

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

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

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
    }
}
```

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] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *COMPOS'97 International Symposium: CompositionalityThe Significant Difference*, Germany, September 1997. DRAFT.

[3] 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.

[4] Bell Labs. *Basic* Spin *manual*, 1997. URL: http://netlib.bell-labs.com/netlib/spin/whatispin.html.

[5] J. C. Bradfield. *Verifying Temporal Properties Of Systems.* Birkhaeuser, Boston, Mass., 1991.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[7] J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computing*, 98(2):142–170, June 1992.

[8] 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.

[9] E. M. Clarke and E. M. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In D. Kozen, editor, *Proc. Logic of Programs*, number 131 in LNCS. Springer Verlag, 1981.

[10] E. M. Clarke et al. Verification of the futurebus+ cache coherence protocol. In *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications*, North-Holland, April 1993.

[11] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 343–354, Albuquerque, New Mexico, January 1992.

[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 and D.A. Schmidt. Limiting state explosion with filter-based refinement. In *Proceedings of the 1st International Workshop on Verification, Abstract Interpretation and Model Checking*, October 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] 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.

[17] C. Sterling. *Handbook of Logic in Computer Science*, volume 2, pages 478–551. Clarendon Press, Oxford, 1992.

[18] 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.

[19] M. Stonebreaker, editor. *Readings in Database Systems*, chapter 4, pages 345–349. Morgan Kaufman Publishers, 2 edition, 1994.

[20] 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.

[21] 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.

[22] 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.