ELSEVIER

Information and Software Technology 41 (1999) 627–638

INFORMATION AND SOFTWARE TECHNOLOGY

Using KBS verification techniques to demonstrate the existence of rule anomalies in ADBs

A.V. Pai[a], R.F. Gamble[a,*], R.T. Plant[b]

[a]*Department of Mathematical and Computer Sciences, University of Tulsa, 600 South College Avenue, Tulsa, OK 74104, USA*
[b]*Department of Computer Information Systems, School of Business Administration, University of Miami, Coral Gables, FL 33124, USA*

Received 6 November 1996; received in revised form 25 February 1999; accepted 4 March 1999

Abstract

As the field of verification and validation for knowledge-based systems (KBSs) has matured, much information, technology, and theory has become available. Though not all of the problems with respect to KBSs have been solved, many have been identified with solutions that can be used in an analogous manner in situations where the application is not necessarily a traditional KBS. As one example, the ''active'' component in an active database (ADB) consists of rules that execute as a result of database accesses and updates. In this paper, we demonstrate that anomalies found to impact the correctness of a KBS can also exist in ADBs. We first compare the rule structure of a KBS with the rule structures of various ADBs. To show their existence, we convert the rule syntax of the ADBs into a consistent format for analysis and anomaly detection. Once converted, we apply KBS verification techniques to isolate these anomalies. Due to the more increasing use of triggered rules in ADBs, this work illustrates the danger these anomalies can pose and the ever increasing need for ADB verification techniques to exist.

Keywords: Knowledge-based systems; Rule verification; Active databases

1. Introduction

An active database (ADB) is a database that dynamically executes specified actions when certain conditions arise, without any user interaction. ADBs have grown in importance because they can be used to enforce integrity constraints, trigger actions based on occurrences of a particular database state or transition between states, and perform knowledge based processing within the database system [1]. Oracle 8 is one such widely used, commercial database. The ''active'' component in an ADB consists of rules that execute as a result of database accesses and updates. The incorporation of an ''active'' component or a rule system to a passive database increases the overall power of the database by promoting inference with the data available. This additional power provides the flexibility of interpreting information from data that would otherwise have little meaning.

Verification involves demonstrating that the system behaves as expected. We use the term ''anomaly'' for an execution or structural attribute of the system that causes unexpected behavior. Not all anomalies lead to errors. Knowledge-based system (KBS) verification includes examining the structure and semantics of a knowledge base, which mainly consists of sets of rules. Primary verification tools generally analyze the knowledge base for anomalies, which may be logical, epistemological, or semantic, and often fall into the following categories [2,3]: *redundancy*, *inconsistency*, and *incompleteness*. Redundancy is the presence of rules that add no new knowledge to the database. Inconsistency refers to conflicts among rules when more than one rule can execute but with contradictory consequences. Incompleteness occurs when the rule system cannot achieve a goal state from a legal initial state. These categories of anomalies and methods of their detection have already been widely researched for various types of KBSs, including rule-based and hybrid systems that may include objects, message passing, and procedures e.g. [2–10]. We use these definitions to determine whether reliability can be compromised with respect to the rule activation and execution in ADBs.

In this paper, we demonstrate via KBS verification techniques that anomalies exist in rule execution of an ADB that can go undetected. The paper is organized as follows: Section 2 provides brief background material on ADBs. Section 3 discusses verification as it is applied to the rules

* Corresponding author. Tel.: + 1-918-631-2988; fax: + 1-918-631-3077.
E-mail address: gamble@utulsa.edu; (R.F. Gamble)

0950-5849/99/$ - see front matter
PII: S0950-5849(99)00017-8

in KBSs. This section also includes an examination of the verification anomalies: redundancy, incompleteness, and inconsistency. Section 4 compares the rule structures of KBS rules to ADB rules, showing that the overall rule formats between these systems are the same. In addition, this section presents our method for demonstrating these anomalies exist, which includes placing the ADB rule in an intermediate representation and using KBS verification analysis over that representation. Section 5 demonstrates the existence of rule anomalies in ADBs and discusses their impact on its integrity. Section 6 concludes the paper with discussion.

2. Active databases

The dynamic properties of an ADB, that cause specified actions to occur automatically when certain conditions in the database arise, can be extremely useful to programmers. For example, consider an inventory database with a large number of instances of products, containing the quantity on hand, threshold, and reorder flag for each product. In a traditional database, as products are sold, the quantity on hand is continually updated. An ADB goes one step further by monitoring the quantity on hand such that if it falls below the threshold, the reorder flag is set. A *trigger* or *rule*, containing an event, condition, and action is placed on the quantity on hand field. When its associated event (an update to the quantity on hand) occurs, the condition of the trigger is evaluated. If the condition evaluates to true, the action is executed, and the reorder flag is set accordingly. The ability to detect events and evaluate/execute rules over a large amount of data make active databases more flexible than traditional database systems [11]. The ability to manipulate data with rules indicates the need for a sound rule processing component to ensure the overall integrity.

In another inventory example, assume that a trigger is put in place by one programmer to reorder ten widgets when the quantity is low. More than ten widgets would require unnecessary additional space and cost, since ten widgets usually last in stock for several months. Unknowingly, another programmer places a trigger to reorder ten widgets whenever the quantity is low and five widgets have been purchased during a single week. If no restrictions are placed on this redundancy by the database system, then when the latter rule is triggered, the former rule is also triggered. Together they each cause the reordering of ten widgets, for a total of twenty widgets. If the error is not caught through some manual intervention, additional costs will incur—either to accept or return the extra widgets—or more space will need to be allocated to hold them.

The development of active databases began with ON conditions in the CODASYL COBOL database systems [12] and in SYSTEM R which utilized triggers in integrity constraints [13]. The refinement of active databases through prototype and experimental systems has continued through the 1980s and 1990s, with systems such as:

- HiPAC (High Performance Active DBMS) [14,15] subsequently extended into an Object form [16–19]
- POSTGRES Rule System [20,21]
- Ariel [1,22]
- The STARBURST Rule System [23].

Detailed examination and discussion of the rule processing components of each of these prototypes can be found in Refs. [14,24–27].

These prototypes and experimental systems have resulted in several commercial systems and are available from Oracle [28], Sybase, Commercial INGRES, and Interbase, [11].

Researchers have examined the language [1], semantics [20], and execution [14] of the rule processing component of various ADBs, such as Ariel, Postgres, and HiPAC. Research shows how the outcome from the activation of a rule set is dependent upon the characteristics of the ‘‘active’’ component of the ADB in question. While current research examines how the execution and outcome of a rule-set is highly dependent on characteristics of the rule processing component, there is limited research available that focuses on whether the outcome resulting is indeed reliable.

Research in the area of active database integrity and correctness has been performed and documented in several areas:

- Formal semantics and specification of active database systems [29–33].
- Active database language design [26,34].
- Tools for the validation, visualization and debugging of active database behavior [35–40].
- An area of significance to the validation of ADB systems has been the research upon termination behavior. Baralis defines this as ‘‘A rule set is guaranteed to terminate if, for any database state and initial modification, rule processing cannot continue forever’’ (i.e. rules cannot activate each other indefinitely), [41]. Several approaches to the analysis of rule termination have been suggested. Selection of an appropriate technique depends upon the underlying rule model. In a syntactic approach Bararis et al. [42] build upon this method by not only analyzing the information from the triggering graph but also the activation graph. This decreases the number of false cycles detected. Bararis also examines the effect of defining the rule prioritization and sequences showing that without prioritization it is not possible to make any assumptions on the execution ordering of rules and that activation will be non deterministic, as chosen by the system. An alternative approach to the termination problem has been suggested by Kovadimce and Urban [33] who reduce every event condition action rule to term re-writing systems to which known techniques for termination are applied. A model theoretic approach to ADB

```
Triggering Statement:          AFTER UPDATE OF column name ON table name
Trigger Restriction (Optional): WHEN (new.column nameA < new.column nameB)
Trigger Action:                FOR EACH ROW
                                    DECLARE variable and variable data type
                                    BEGIN
                                        IF...THEN..END  IF
                                        :
                                    END.
```

Fig. 1. Abstracted Oracle Trigger [28].

termination, conflict and non determinism has been considered by Bidot and Maabout, [43] whom define a well founded semantics in relation to static and transition integrity constraint enforcement.

This is in addition to the extensive literature on the construction, use and application of active database systems [44–54].

2.1. Rules and Triggers

The experimental platform used in this research to investigate the presence of rule anomalies in active databases is the widely used Oracle 8 ADB.

The Oracle 8 architecture is similar to that of traditional relational database. However, as an active database Oracle 8 represents triggers, which embody condition and action statements, like rules. A trigger is viewed as a procedure similar to a stored procedure and is processed by Oracle's RDBMS. As a result, when a trigger is created, a parsed procedural representation is stored in the database. When the trigger is activated the procedure is loaded into the System Global Area (SGA). The PL/SQL Engine and SQL Statement Executor then combine to process the statements within it.

A database trigger in Oracle is defined as a rule and an associated event–condition–action (E–C–A) procedure that is associated with a table, written in PL/SQL, stored as a PL/SQL block. It is implicitly executed when an INSERT, DELETE, or UPDATE event is issued against the table for which the trigger is defined. A trigger can only fire when it is enabled. It can also fire other enabled triggers, producing a cascading effect. The basic format of a database trigger consists of a triggering event, a trigger restriction, and a trigger action and is depicted in Fig. 1.

A triggering event is the SQL statement that causes the trigger to fire. A trigger restriction specifies a Boolean expression that must be TRUE for the trigger to fire. A trigger action is the procedure that contains the code to be executed when the trigger restriction evaluates to true.

There are two types of triggers:

Row triggers
A row trigger is fired each time a table is affected by the triggering statement. For example, if an UPDATE statement is issued for a table, the row trigger is fired for each row affected by the UPDATE statement.

Statement triggers
A statement trigger is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects, even if no rows are affected. For example, if a DELETE statement deletes several rows from a table, a statement-level DELETE trigger is fired once, regardless of how many rows are deleted.

When a trigger is defined, a coupling mode is specified to indicate when the triggering action is to be executed in relation to the triggering statement:

Before triggers (immediate mode)
BEFORE triggers execute the trigger action before the triggering statement.
After triggers (deferred mode)
AFTER triggers execute the triggering action after the triggering statement is executed.

As a result four types of triggers can be created: BEFORE statement triggers, BEFORE row triggers, AFTER statement triggers, and AFTER row triggers.

Multiple triggers can be associated with a single table and can be fired either concurrently or as part of a cascade of triggers. For example, a BEFORE statement trigger and a BEFORE row trigger can be defined for an identical command. Conflict resolution strategies have been defined to manage the interrelationships among the four trigger types allowed. Our paper illustrates that there are remaining anomalies from concurrent firing or cascading firing that are not detected.

3. Verification of knowledge based systems

Assuming a correct valid specification for a KBS has been achieved, establishing the correctness of that systems implementation, against the specification, is termed the *verification* task and includes examining the structure and semantics of its knowledge base. Primary verification tools generally analyze the knowledge base for anomalies, which may be logical, epistemological, or semantic, and often fall into the following categories [2,3]: *redundancy*, *inconsistency*, and *incompleteness* (which includes *circularity* and *errors of omission*).

Below we provide brief definitions for the potential anomalies that can be found among KBS rules. We use the term potential because there may be cases where the

developers caused the anomalies purposely. For instance, redundancy may be used to address distribution of performance issues, inconsistency may be used to allow for nondeterminism, and errors of omission may not be a concern for KBSs that evaluate partial information. For each category we discuss why these anomalies can impact negatively the reliability of a KBS.

- *Redundant rules*: This anomaly occurs if a rule is present and executable but does not contribute to the knowledge. Redundantly executing rules can cause problems in two ways. The first is through unexpected behavior such as a value is double what is expected. The second concern is when the knowledge is no longer considered appropriate or useful resulting in deletion of the rule. Hidden redundant rules that are not deleted will now execute based on inappropriate knowledge. Types of redundancy include:

  —*Duplication*: Two rules are completely identical.
  —*Subsumption*:One rule is a generalization of another.
  —*Reducible*: Two rules can be reduced to a single rule accomplishing the same task.
  —*Indirect*: Two deductive paths lead to the same result.

- *Inconsistent rules*: This anomaly occurs when simultaneously firable rules produce inconsistent or conflicting results. One type of inconsistency may be that the same fact is added and deleted in a single execution. Another type of inconsistency may be that two incompatible results are present but do not directly conflict. For instance, there may a constraint that only one ball is allowed to be stored, no matter what its color. If two rules are allowed to execute from the same starting state such that one asserts a blue ball and one asserts a red ball, then there is an inconsistency in the rules. Resolution, in this instance, would be that a check for an existing ball be part of the rule condition and the removal of the existing ball be an action performed prior to asserting a new ball. Hence, in many cases, detection of conflicting rules requires certain meta-rules that define system constraints.
- *Incomplete rules and information*: This anomaly category encompasses common errors that originate from structuring knowledge base or configuring the rules and data. Some of the anomalies will not directly affect reliability, but may affect understandability and efficiency. The category is divided into two major parts.

  —*Circularity*: This anomaly is present when there is a circular chain of executing rules. Most conflict resolution strategies preclude circularity at execution time. However, if the design is circular, then the KBS will not reach a conclusion. It will either be thwarted from infinitely looping by conflict resolution or it will loop indefinitely and not converge.
  —*Errors of omission*: This anomaly is a subcategory that covers instance when missing information or knowledge prevents the system from reaching a goal state. Within this category are the following detailed anomalies.

  Missing rules: The set of rules does not cover all possible inputs.
  Unused inputs and outputs: If rules are not missing, then these unnecessarily clutter the storage area.
  Unfirable rules: A rule's condition can never be satisfied.
  Impossible combinations: Input conditions are stored that cannot coexist. This is similar to the inconsistency problem, except that rules have not fired to produce these conditions, they are there at the start of execution.
  Dead end rules: Rules exist that do not lead to any conclusions.

Though there are many execution models of KBSs, the structural configuration and interaction of the rules can often be abstracted to show where anomalies exist [2]. In this paper, we focus on the above categories of redundancy and inconsistency. Circularity is internally prevented by Oracle. Errors of omission in ADBs require distinctive tests because it must be assumed that all the information available is present at the time of execution or has a definite point of interaction. We discuss how the understanding of rule structure can serve as the bases for determining the potential existence of anomalies in the rules of an active database.

4. Obtaining a workable ADB rule representation

The logical form of a typical KBS rule and an active database rule is similar, however the rule processing within an ADB differs from the rule processing in KBSs. A KBS typically uses a match–select–execute cycle in which the knowledge base holds the rules, the working memory holds the available facts, and the inference engine deduces new information by comparing the contents of working memory to the rules. If the conditions in the left-hand side (LHS) of the rule hold, then the actions in the right-hand side (RHS) of the rule are performed against the contents of working memory. An ADB uses coupling mode information that indicates when the conditions and actions of rules are to be evaluated and executed relative to the event.

In this section, we examine the rule processing of an ADB and compare this to the processing of KBSs. We illustrate that a subset of the above anomalies can go undetected in ADBs, resulting in the possibility of the ADB not satisfying its verification criteria.

4.1. Knowledge representation in ADBs and KBSs

The rule structure of KBSs consists of

IF Condition THEN Action

statements (C–A). If the condition is satisfied then the

Table 1
A process for anomaly identification

Phase 1	Identify events, relations, and attributes relative to the ADB and rule set
Phase 2	For all rules in the rule processing component Generate RuleID statement Generate Event Trigger statement Generate Rule Condition statement Generate Rule Action statement
Phase 3	Determine using KBS verification techniques the existence of potential anomalies and their categories
Phase 4	Test for anomalous behavior original rules within Oracle 8
Phase 5	Report actual existence and impact

action is executed. The condition is a Boolean expression that is composed of predicates and logical operators that must evaluate to *true* prior to action execution.

The rules in ADBs consist of:

IF Event–Condition THEN Action

statements (E–C–A). The E–C–A statement is similar to the C–A format of KBSs for verification purposes. The event can be translated into a KBS rule condition using a glossary to maintain the mappings. The distinction lies in the time of rule firing. Rules in KBSs are fired whenever the state of working memory matches the conditions set by the rules. Rules in ADBs fire whenever the event specified by the rule occurs. However, this distinction does not impact the analysis we present in this paper. Thus, we can transform an Oracle rule into a KBS format. For example, starting with an Oracle rule:

```
Rule 5
CREATE TRIGGER rchain 1
AFTER UPDATE OF quantity
ON inv
FOR EACH ROW
WHEN (((new.QUANTITY - old.DEFECTED_QTY) <
= old.THRESHOLD) AND (old.status = 'UNPRE-
                              PARED'))
BEGIN
        UPDATE stock
        SET status = 'MIN'
        WHERE itemNo = :old.itemNo;
```

The Event, Condition and Action parts are:

```
Event:      UPDATE OF quantity ON inv
Condition:      ((new.QUANTITY-
                old.DEFECTED_QTY) < = old.
                THRESHOLD) AND (old.status =
                'UNPREPARED')
Action:     UPDATE stock SET status = 'MIN'
            WHERE itemNo = :old.itemNo;
```

These can then be converted into predicates as:

```
Event:      E1
Condition:      P(x, y, z) AND Q(w)
Action:     E7 AND T(v)
```

which can be stated in KBS rule format

```
Condition:      E1 AND P(x, y, z) AND Q(w)
Action:     E7 AND T(v)
```

In this third step, the event of an ADB rule is translated to part of the condition statement of a KBS rule normalizing its final format.

4.2. Demonstrating anomalies in active databases

Having understood the structural similarities between typical KBS and ADB rules, in order to determine if the anomalies are carried over to ADBs and if there are any safeguards against them, we adhere to the process in Table 1. In phase 1, we identify the entities to abstract in an ADB to create a KBS rule format for analysis. This leads to phase 2, where the individual rule-processing components are transformed to the KBS rule format. We use established KBS definitions for the existence of a potential anomaly along with a sample ADB rule set to identify according to KBS standards, where potential anomalies occur in phase 3. The anomalies are then tested using the original rules in Oracle 8 in phase 4. Whether or not the anomalies exist in the commercial ADB and their impact on its behavior is reported in phase 5.

4.2.1. Intermediate representation generation

The second phase in Table 1 requires conversion to a KBS rule format. However, due to the nature of programming language syntax and semantics, it is difficult to examine programs directly for anomalies, and as in other aspects of computer science, it is necessary to develop and utilize an abstracted level of the system in which to reason about the program. Thus we need to develop an intermediate representation prior to full conversion.

The intermediate representation serves as a generic language in which to convert different ADB representations of rules into a consistent form for analysis. By abstracting away the particular syntax of an ADB rule set we can efficiently isolate potential anomalies.

Analyzing the conjunctive normal form of the rules, the intermediate representation consists of the major components:

1. *Rule Id*: which refers to a unique rule name
2. The *Event trigger* which consists of:
   - the event id—unique for each event,
   - event list—at most one legal event that triggers the rule,
   - relation—the database relation affected by the event, and

Table 2
A sample mapping of an oracle rule to the intermediate representation

Oracle rule	Intermediate representation
RULE 1	RuleId :: = R1
CREATE TRIGGER rule 1	Event Trigger statement :: = E1 : UPDATE INV (quantity)
AFTER UPDATE OF quantity	Rule Condition statement :: = P(a, b, c) AND Q (d)
ON inv	Rule Action statement :: E2 : UPDATE ON_REORDER (?q1,q2*2,q3,q4)
FOR EACH ROW	
WHEN (((new.QUANTITY-old.DEFECTED_QTY) < = old.THRESHOLD) AND (old.status = 'UNPREPARED'))	
BEGIN	
reorder(:old.itemNo);	
END;	
CREATE PROCEDURE reorder (itemNum in VARCHAR2)	
AS BEGIN	
UPDATE on_reorder	
SET QTY_RECORDER = qty_reorder *2	
WHERE itemNo = itemNum;	
END reorder;	

Table 3
A second sample mapping of an Oracle rule

Oracle rule	Intermediate representation
RULE 2	RuleId :: = R2
CREATE TRIGGER rule2	Event Trigger statement :: = E1 : UPDATE INV (quantity)
AFTER UPDATE OF quantity	Rule Condition statement :: = P(a,b,c) AND Q(d) AND R(e)
ON inv	Rule Action statement :: = E2 : UPDATE ON_REORDER (?q1, q2*2, q3, q4)
FOR EACH ROW	
WHEN (((new.QUANTITY-old.DEFECTED_QTY) < = old.THRESHOLD) AND (old.status = 'UNPREPARED') AND (old.ON_RECALL_LIST = 'TRUE'))	
BEGIN	
reorder(:old.itemNo);	
END;	
CREATE PROCEDURE reorder (itemNum in VARCHAR2)	
AS BEGIN	
UPDATE on_reorder	
SET QTY_REORDER = qty_reorder *2	
WHERE itemNo = itemNum;	
ENDreorder;	

- attribute list—the attributes within the relation that the event affects. All the components of the *event trigger* reflect the tuple(s) that have been instantiated as a result of evaluating the triggering event.

3. The *Rule Condition* specifies all the predicates that must evaluate to true in order for rule execution
4. The *Action* of the rule executes on the tuple set.

The intermediate representation allows for both predicates and functions. A predicate in Oracle can have an embedded function, such as $abs(p2.time - p1.time) < 30$. For simplicity, we assign to an embedded function identifiers whose variables receive the return value of the function. In the intermediate form, this identifier is assigned an entire predicate. Any function calls, variables, and attributes evaluated within the predicate are considered parameters. Thus, the intermediate form for the predicate,

$$abs(p2.time - p1.time) < 30$$

would be

$$P(x)$$

where P represents the identifier for the entire predicate and x represents the return value of the embedded function call.

The *rule action* contains an action list that contains other events that may be triggered upon the evaluation and execution of the current rule. The Backus Naur Form (BNF) for the intermediate representation is detailed in [25].

4.2.2. The transformation process

This section considers the transformation of the Oracle rule set into the intermediate form. This is illustrated through Rule 1 as described in Table 2.

Each rule as stated earlier is assigned:

1. A Rule Id,
2. An Event Trigger Statement,
3. A Rule Condition Statement,
4. A Rule Action Statement.

Consider Rule 1 in Table 2, the following procedure is required for the transformation.

Rule Id: We assign 'R1' as the identifier of Rule 1.

In the *Event Trigger Statement*, a unique event id is assigned to each of the events that cause the trigger to fire. We assign E1 since the event list consists of only one event that causes the rule to fire (UPDATE, in this case). The relation being updated is INV, and the attribute updated in the INV relation is quantity.

The *Rule Condition Statement* consists of two predicates, (new.QUANTITY - old.DEFECTED_QTY) <= old.THRESHOLD) and (old.status = 'UNPREPARED').

Table 4
An example rule set

Oracle ADB rule set	KBS rule set
RULE 1 CREATE TRIGGER rule 1 AFTER UPDATE OF quantity ON inv FOR EACH ROW WHEN (((new.QUANTITY-old.DEFECTED_QTY) < = old.THRESHOLD) AND (old.status = 'UNPREPARED')) BEGIN reorder(:old.itemNo); END;	R1: E1, P(x,y,z), Q(w) → E2
RULE 2 CREATE TRIGGER subsumption AFTER UPDATE OF quantity ON inv FOR EACH ROW WHEN (((new.QUANTITY-old.DEFECTED_QTY) < = old.THRESHOLD) AND (old.status = 'UNPREPARED') AND (old.ON_RECALL_LIST = 'TRUE')) BEGIN reorder(:old.itemNo); END;	R2: E1,P(x,y,z),Q(w), R(v) → E2
RULE 3 CREATE TRIGGER duplication AFTER UPDATE OF quantity ON inv FOR EACH ROW WHEN (((new.QUANTITY-old.DEFECTED_QTY) < = old.THRESHOLD) AND (old.status = 'UNPREPARED')) BEGIN reorder(:old.itemNo); END;	R3: E1,P(x,y,z), Q(w) → E2
RULE 4 CREATE TRIGGER unnec_if AFTER UPDATE OF quantity ON inv FOR EACH ROW WHEN (((new.QUANTITY-old.DEFECTED_QTY) < = old.THRESHOLD) AND (old.status = 'PREPARED')) BEGIN reorder (:old.itemNo); END;	R4: E1,P(x,y,z), ¬ Q(w) → E2
RULE 5 CREATE TRIGGER rchain 1 AFTER UPDATE OF quantity ON inv FOR EACH ROW WHEN (((new.QUANTITY-old.DEFECTED_QTY) < = old.THRESHOLD) AND (old.status = 'UNPREPARED')) BEGIN UPDATE stock SET status = 'MIN' WHERE itemNo = :old.itemNo; END;	R5: E1,P(x,y,z), Q(w) → E7, T(v)
RULE 6 CREATE TRIGGER rchain2 AFTER UPDATE OF status ON stock FOR EACH ROW WHEN (new.status = 'MIN') BEGIN reorder(:old.itemNo); END;	R6: E7,T(x) → E2

Each predicate is assigned an identifier. We assign P to the first predicate and Q to the second predicate. In addition, each attribute involved in the predicate is assigned a parameter.

Finally, all the events executed in the action are stated in the *Rule Action Statement*. Note that the action of the rule may contain function calls that execute INSERT, DELETE, or UPDATE statements. As a result, these functions are examined to assure that all events executed in the action of this rule have been identified in the *rule action statement*. The rule action statement consists of an event id for each new event executed, the event, the relation affected by the event, and the tuple instantiated in the relation. The instantiated tuple is represented by identifiers for each attribute in the tuple. The attributes that represent the key are preceded by '?'. If particular values are being assigned by the (INSERT, DELETE, UPDATE) event, these values can directly replace the attribute identifier. For example, the attribute q2 in Table 2 is updated by this event to q2*2.

As an additional example, we show the intermediate representation for Rule 2 in Table 3.

5. Anomaly occurrence in ADBs: an example rule set

To determine if the KBS verification anomalies can occur in an ADB, we developed a sample rule set for execution in Oracle 8. The KBS abstraction of the rules is used to convey the existence of a potential anomaly. The original rule set purposely contained anomalies for redundancy, inconsistency, and incompleteness [25], though we restrict our presentation here to the first two categories. Rules 1 and 2 from Tables 2 and 3, respectively are part of the rule set.

5.1. When it is an anomaly

We present only the pertinent subset of rules that exhibit particular redundancies and inconsistencies and discuss their behavior when executed in Oracle 8. The rules and corresponding KBS representation are presented in Table 4.

Duplication: R1 and R3 are duplicate rules because they have identical LHSs and RHSs. These rules fire when an UPDATE to the quantity field of the INV table occurs. Both rules state that if the quantity in the inventory table (INV) of a product falls below the threshold and preparations to reorder have not been made then to reorder the item. The reorder function assigns the quantity to be reordered in the ON_REORDER table. Upon enabling these triggers (named *rule1* and *duplication*), the event, *event1*, is executed. *Event1* performs an UPDATE to the INV table causing both triggers to activate. Since the inventory is

evaluated to be low for the items in the INV table, both of the trigger actions are executed. Duplicate rules are an anomaly because they succeed in the same situation and give the same results. Further, both rules affect the integrity of the database because these redundant rules cause the quantity of an item to be reordered twice.

Subsumption: R2 subsumes R1. R2 is the more restrictive of the subsumed rules because it has the additional conditions, (old.ON_RECALL_LIST = 'TRUE'), which must be satisfied for its activation. Upon enabling these triggers (*rule1* and *subsumption*), the event, *event1*, is executed similar to the duplication example above. When the restrictive rule, R2, succeeds, the less restrictive rule succeeds also, causing a redundancy. The execution of both rules causes the quantity of an item to be reordered twice, once again compromising the reliability of the database.

Reducible: R1 and R4 portray reducible rules because of unnecessary conditions. R1's condition (old.status = 'UNPREPARED') and R4's condition (old.status = 'PREPARED') show that one rule contains the positive form of status and the other rule contains its negative form. R4 executes on an UPDATE to the INV table when the inventory of an item falls below the threshold and the status of the reorder is 'PREPARED'. Since the status of an item can either be 'PREPARED' or 'UNPREPARED', these conditions on the status are unnecessary and ineffective in determining which rule should fire. The implication of these rules is that a reorder of the product can be made regardless of their status.

Indirect: R1, R5, and R6 represent a potential chained redundancy since R1 and R5 have identical LHSs, and the consequent of R1 is reached through a series of deductions from rules R5 and R6. R5 is triggered when an UPDATE to the quantity field of the INV table occurs. The only difference is that R5 updates the stock table to indicate that the status of the item is low (status = 'MIN'). This UPDATE of the status field in the Stock table triggers R6 to execute and reorder this item. When R5, R6, and R1 are enabled, and the UPDATE event on the INV table is executed, a reorder to the items satisfying the conditions of these rules are made twice, thus displaying incorrect information in the database.

Table 5 summarizes the redundant configuration by grouping the rules according to the type of redundancy that occurs.

Contradicting actions in ADBs can take the following command forms: UPDATE-DELETE, UPDATE-INSERT, INSERT-INSERT, and UPDATE-UPDATE, falling into two types of conflict. The first type of conflict, *direct conflict*, occurs when one of the two rules (who have identical conditions and events) attempts to perform an UPDATE to an attribute in a relation and the other rule attempts to perform a DELETE action on the same attribute and relation being updated by the first rule.

Rules R7 and R8 below are directly conflicting. The LHSs of these rules are identical, but the RHSs require contradictory actions be taken. For example, when an UPDATE to the qty_reorder attribute of the On_Reorder table occurs, these triggers are evaluated. When executed the trigger actions of R7 performs an UPDATE to the Def_Stock table and Items_To_Return table for a designated tuple. The trigger actions of R8 delete the same tuple updated by R7 in the Items_To_Return table. An update action and a delete action of the same tuple(s) can result in unexpected behavior from the conflict.

Actual results in Oracle show that R8 succeeds in its deletion. The indication that R7 also was executed is shown by the UPDATE to the Def_Stock table. R7 also performed an Update to the Items_To_Return table, but this update was overridden by R8's actions. The data that results, in this case, are from the trigger action of R8 since it was executed last. Such conflict can result in the unintentional modification or deletion of information and questions the reliability of the database (Table 6).

The second type of conflict, *semantic or complex conflict*, can occur in multiple situations. The first situation occurs when two rules with identical event triggers and event condition statements attempt to update the same tuple of the same relation with different values (UPDATE-UPDATE). The second situation occurs when two rules with identical event trigger and event condition statements attempt to insert different tuple values into the same relation (INSERT-INSERT). The third complex conflict situation occurs when one of two rules with identical event trigger and event condition statements attempts to update attributes of a relation to a certain value, and the other rule inserts a tuple in the same relation whose attribute values differ from what is being reassigned by the first rule (UPDATE-INSERT). We give an example of this conflict using the following two rules (Table 7).

Rules R9 and R10 depict a complex conflict. The LHSs of these rules can succeed when their conditions evaluate true, however their RHS cannot be both true simultaneously. A priori meta-knowledge is used over the potential values of the variables that states that the dual update is unacceptable. For example, when an UPDATE to the PrepareToMail attribute for the Items_To_Return event occurs, these rules are triggered. Given that the conditions evaluate to true, both trigger actions are executed. The trigger action of R9 assigns the shipment type attribute with the value 'LOCAL' while the trigger action of R10 assigns the same attribute the value 'US'. These assignments are semantically conflicting.

Actual results in Oracle show that R9 succeeds in assigning 'LOCAL' to the shipment type of the tuples meeting the condition of the trigger. In order to verify that R10 was also executed, additional tests were needed. In these tests, additional update statements were placed in the trigger actions of R9 and R10. These update statements modified attributes of different tables. By examining whether these statements were executed in the appropriate tables, it was determined that in the case of conflicting rules, both rules are executed. However, the data that results, in this case, is from the trigger action of R9 since it was executed last. The reliability

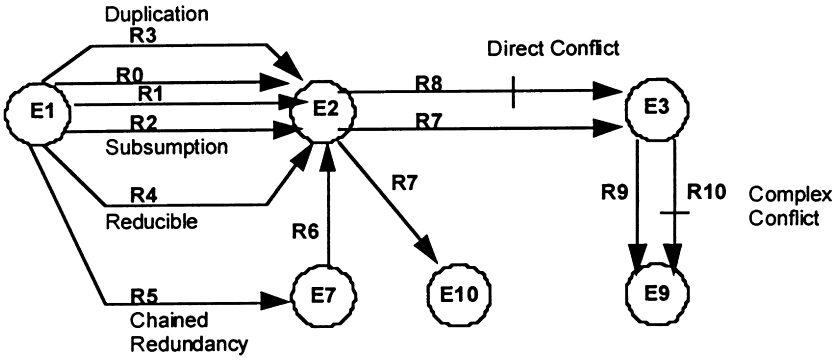

Fig. 2. A sample event dependency graph.

of the database is at stake when semantically contradicting values are being assigned by internal mechanisms of the system since it is unclear which assignment will take precedence.

When there are only two values for an attribute, meta-knowledge can be applied easily to detect and resolve conflicts. However, in the cases in which a variable has more than two values, the analysis becomes more complex. Meta-knowledge would be used to examine seemingly inconsistent rules to determine if the actions of the rules assign distinct values to the same variables.

5.2. When it is not an anomaly

Analysis in the KBS format alerts the programmer as to which rules need to be examined to assess the exact cause behind the anomaly. Upon taking a closer look at the execution of the suspect rules in the original rule set, the anomaly's existence is determined. For instance, circular rules may be placed in the ADB and statically analyzed using KBS verification techniques. However, during execution of these rules, it was seen that Oracle has a built-in mechanism to disallow circular triggering.

As stated earlier, this static analysis indicates the potential for an anomaly. Deeper analysis and testing determines the actual extent and impact. For example, one type of static representation may be an event dependency graph that shows how rules link together in their execution. Such a graph may be used by automated KBS anomaly detection tools such as Cover [8] the KB-Reducer System [55], and the Expert Validation Associate [56]. Each of these tools has the capability of identifying all of the forms of redundancy, inconsistency, and incompleteness discussed. Below is the summary of events used by the subset of rules discussed that form their connection. Fig. 2 depicts the graph for the rules we have discussed in this paper, with the addition of a new rule R0 discussed below. The crossbar on the arcs indicates the ''negative'' event action, which may be direct or guided by meta-rules.

EVENT SUMMARY LISTING

E1: UPDATE INV (?v1, quantity - 100, v3, v4, v5, v6)

E2: UPDATE ON_REORDER (?q1, q2*2, q3, q4)

E3: UPDATE ITEMS_TO_RETURN (?i1, i2, i3, i4, TRUE, i6,. i7, i8, i9, i10)
DELETE ITEMS_TO_RETURN (?i1, i2, i3, i4, i5, i6, i7, i8, i9, i10)

E7: UPDATE STOCK (?s1, MIN, s3)

E9: UPDATE DEF_STOCK (?d1, d2, d3, LOCAL, d5, d6, d7)
UPDATE DEF_STOCK (?d1, d2, d3, US, d5, d6, d7)

E10: UPDATE DEF_STOCK (?d1, ret, d3, d4, d5, d6, d7)

For example, if the rule R0, shown in Fig. 3, were a part of this rule set, it would be seen in the diagram as an arc going from E1 to E2. However, upon comparing this rule with the other rules that are triggered by E1 and reach E2, it is concluded that this rule is not redundant with the others since the LHS of R0 and the other rules leading to event E2 are not problematic.

6. Summary of implementation and impact

Once we found that these anomalies can exist in ADBs, we determined what effect they had, if any, on the reliable execution of ADBs. The rule set designed reflects actions to be taken for inventory management. For example, when the quantity for a particular product falls below the desired threshold, then a reorder flag is set. If a product is defective or has been recalled,

RULE 0

```
CREATE TRIGGER notRedundant
AFTER UPDATE OF quantity
ON inv
FOR EACH ROW
WHEN (new.ItemNo = '10A')
BEGIN reorder(:old.itemNo); END;
```

Fig. 3. Rule illustrating additional analysis after anomaly detection.

Table 5
Anomaly rule groups

Redundancy type	Rule groups	Result
Duplication	R1: E1,P(x,y,z), Q(w) → E2	Both rules execute
	R3: E1,P(x,y,z), Q(w) → E2	The qty_reorder attribute of the On_Reorder table updated for all attributes regardless of the restricting conditions on the status of the Inv table
Subsumption	R1: E1,P(x,y,z), Q(w) → E2	Both rules execute
	R2: E1,P(x,y,z), Q(w), R(v) → E2	The qty_reorder attribute of the On_Reorder table updated twice
Reducible	R1: E1,P(x,y,z), Q(w) → E2	Either rule executes independent of the value of Q(w)
	R4: E1,P(x,y,z), ¬ Q(w) → E2	The qty_reorder attribute of the On_Reorder table is updated in eachcase
Indirect	R1: E1,P(x,y,z), Q(w) → E2	All three rules execute
	R5: E1,P(x,y,z), Q(w) → E7, T(v)	The qty_reorder attribute of the On_Reorder table updated twice as a result of R1 and R6
	R6: E7, T(x) → E2	

proper measures are taken to ship these items back to the supplier. The rule set also incorporates, as discussed earlier, anomalies prevalent to KBSs.

Each rule is assigned the deferred mode of execution, referred to as AFTER, indicating that the rules' triggering actions are executed after the triggering event has been executed. The mode limitation allows us to determine more clearly that the anomalies are a result of the rule contents and structure rather than their execution modes. The trigger type used to define the rules in the rule set are row triggers.

The basic steps for testing include: identifying the anomaly under test, isolating the rules or triggers causing the anomaly, executing the events that cause these triggers to fire, and examining how the anomaly effects the data residing in the database as a result of trigger execution [25].

The results indicate that each of the anomalous rules execute when the appropriate event occurs. The execution of these rules affect the final data values assigned to the attributes in the database. For example, redundant rules have the potential of updating an attribute twice. Inconsistent rules have the potential of assigning different values to an attribute depending on which rule executes last, which may occur in random order. Thus, the consequences of these anomalies have a negative impact on the integrity of the database.

7. Discussion and conclusion

Database management systems are becoming more sophisticated with the incorporation of rules and a rule processing component. The advantage of this new capability results from the implicit activation of the rule inference process, thus making the system an ''active'' database. This implicit processing forestalls users from observing the changes being made to relevant data, thus heightening the need for implementing correct rules. As a result, the necessity to verify that the rules maintain the integrity of the database.

We show that the verification anomalies once applicable to KBSs are also relevant to ADBs. The rule syntax of an ADB is converted to an intermediate form. We structurally examine the rules according to KBS verification anomaly definitions. We then confirm through an Oracle implementation that these anomalies question the integrity of the ADB. For example, our test results show that redundant rules can perform the same action on the specified attribute twice, possibly inserting incorrect information in the database. By executing these rule sets in Oracle and seeing that in fact these anomalies can go undetected, there should be some concern among the ADBs that the more complex the systems become and the more developers there are to encode applications, the reliability of the ADB results can be impacted.

Table 6
Rule groups 7 and 8

```
RULE 7
CREATE TRIGGER direct_conflict1
AFTER
UPDATE OF qty_Reorder
ON on_Reorder
FOR EACH ROW
WHEN (new.qty_reorder > 0)
BEGIN
    PrepareForReturn(:old.ItemNo);
END;
```

```
RULE 8
CREATE TRIGGER direct_conflict2
AFTER
UPDATE OF qty_Reorder
ON on_Reorder
FOR EACH ROW
WHEN (new.qty_reorder > 0)
BEGIN
    PrepareForRestock(:old.ItemNo);
END;
```

Table 7
Rule groups 9 and 10

RULE 9	RULE 10
CREATE TRIGGER com_conf1	CREATE TRIGGER com_conf2
AFTER	AFTER
UPDATE OF PrepareToMail	UPDATE OF PrepareToMail
ON Items_to_Return	ON Items_to_Return
FOR EACH ROW	FOR EACH ROW
WHEN (new.PrepareToMail = 'TRUE')	WHEN (new.PrepareToMail = 'TRUE')
BEGIN	BEGIN
UPDATE def_stock	UPDATE def_stock
SET shipmtype = 'LOCAL'	SET shipmtype = 'US'
WHERE itemNo = :new.itemNo;	WHERE itemNo = :new.itemNo;
END;	END;

Of the existing automated KBS verification tools, an analysis of the direct applicability of these tools to ADBs is required. Further, an automated tool to compile the syntax of our intermediate representation and subsequently generates an event dependency diagram requires further development.

The intermediate representation we define captures potential anomalies, such as those prevalent in KBSs, independent of coupling modes. Though the intermediate representation can be applied to rule sets using varying modes, it does not directly capture anomalies that are a result of coupling modes. Further analysis can be performed to determine the existence of these anomalies [25].

The importance of this paper is not *how* to detect rule anomalies in ADBs, but rather the demonstration that these anomalies can exist and can cause unexpected and unreliable behavior. Once an understanding of the abstraction needed for translation to a KBS rule format is achieved, then ADB researchers can examine KBS automated detection and resolution techniques. These techniques can then be extended to overlay particular ADB rule execution models.

Acknowledgements

This research was supported in part by the U.S. Department of Energy, contract #DEAC22-93BC14894 and DARPA CAETI program, contract #N66001-95-C-8628.

References

[1] E.N. Hanson, The design and implementation of the Ariel active database rule system, Technical Report UF-CIS-018-92, Department of Computer and Information Sciences, University of Florida, 1991.
[2] R.F. Gamble, T.M. Shaft, Eliminating redundancy, inconsistency, and incompleteness in rule-based systems, International Journal of Software Engineering and Knowledge Engineering 7 (4) (1996) 673–697.
[3] S. Murrell, R.T. Plant, A survey of tools for the verification and validation of KBS—1985–95, Decision Support Systems 21 (1997) 4.
[4] R.T. Plant, Validation and verification of knowledge-based systems, Workshop Notes AAAI. 1994, Seattle, WA, 1994.
[5] R.F. Gamble, C. Landauer, Validation and verification of knowledge-based systems, Workshop Notes IJCAI-95, Montreal, Que. Canada, 1995.
[6] R. Mukherjee, R.F. Gamble, J.A. Parkinson, Classifying and detecting anomalies in hybrid knowledge based systems, Decision Support Systems 21 (1997) 231–251.
[7] D.E. O'Leary, Verification of frame and semantic network knowledge bases, AAAI-89 Workshop on Knowledge Acquisition for KBSs, 1989.
[8] A.D. Preece, R. Shinghal, A. Batarek, Verifying expert systems: a logical framework and practical tool, Expert Systems with Applications 5 (1992) 421–436.
[9] R. O'Keefe, D.E. O'Leary, Expert system verification and validation: a survey and tutorial, Artificial Intelligence Review 7 (1993) 3–42.
[10] J. Schmolze, A. Vermesan, Validation and verification of knowledge-based systems, Workshop Notes AAAI-96, Portland, OR, 1996.
[11] S. Chakravarthy, A comparative evaluation of active relational databases, Technical Report UF-CIS-TR-93-002, Department of Computer and Information Sciences, University of Florida, 1993.
[12] CODASYL Data Definition Language Journal of Development National Bureau of Standards Handbook 113 US Government Printing Office, (SD Catalog No c13.6/2:113), Washington, DC, 1973.
[13] K.P. Eswaran, D.D. Chamberlain, Functional specifications of a subsystem for database integrity, in: Proceedings of the DM SIGMOD Conference 1992, pp. 81–91.
[14] U. Dayal, Active database management systems, in: Proceedings of the Conference of Data and Knowledge Bases, Jerusalem, 1988.
[15] U. Dayal, F. Manola, The HiPAC project: combining active databases and timing constraints, SIGMOD Rec. 17 (1988) 51–70.
[16] N.H. Gehani, H.V. Jagadish, O. Shmueli, Event specification in an active object-oriented data base, ACM SIGMOD, 1992, pp. 81–90.
[17] S. Gatziu, S. Dittrick, Events in an object-oriented database, in: N.W. Paton, M.H. Williams (Eds.), Proceedings of the 1st International Workshop on Rules in Database Systems, Springer, Berlin, 1994, pp. 23–39.
[18] S. Chakravarthy, V. Krishnaprasad, E. Anwar, S.K. Kim, Composite events for active databases: semantics, contexts and detaction, in: J. Bocca, M. Jarke, C. Zaniolo (Eds.), Proceedings of the 20th International Conference on Very Large Data Bases, Morgan Kaufmann, Los Altos, CA, 1994, pp. 606–617.
[19] S. Ceri, P. Fraternali, S. Paraboschi, L. Tanca, Active rule management in chimera, in: J. Wisdom, S. Ceri (Eds.), Active Database Systems: Triggers and Rules for Active Database Processing, Morgan Kaufmann, Los Altos, CA, 1996, pp. 151–175.
[20] M. Stonebraker, E. Hanson, S. Potaminos, The POSTGRES rule manager, IEEE Transactions on Software Engineering 14 (7) (1988) 897–907.
[21] M. Stonebraker, L. Rowe, M. Hirohama, The implementation of POSTGRES, IEEE Transactions on Knowledge Data Engineering, March, 1990.
[22] E.N. Hanson, Ariel, in: N.W. Paton (Ed.), Active Rules in Database Systems, Monographs in Computer Science, Springer, Berlin, 1999, pp. 221–232.
[23] J. Widom, S. Finkelstein, Set-oriented production rules in relational database systems, Association for Computing Machinery, 1990, pp. 259–270.
[24] E.N. Hanson, J. Widom, Rule processing in active database systems, International Journal of Expert Systems 6 (1) (1993) 83–119.
[25] A.V. Pai, Verifying the rule processing components of active databases, MS Thesis, Department of Mathematical and Computer Sciences, University of Tulsa, 1995.
[26] S.D. Urban, A.M. Wang, The design of a constraint/rule language for an object-oriented data model, Journal of Systems and Software 28 (1995) 203–224.

[27] R. Elmasri, S. Navathe, Fundamentals of Database Systems, Benjamin/Cummings, Redwood City, CA, 1989.
[28] Oracle, Oracle Corporation, Redwood Shores, CA 94065. Oracle 7 Server Manuals, Release 7.1, 1992.
[29] T. Coupaye, C. Collet, Denotational semantics for an active rule execution model, in: Proceedings of the Second International Workshop Rules in Database Systems '95, Glyfada, Athens, Greece, September 25–27, 1995.
[30] N.W. Paton, J. Campin, A.A.A. Fernandes, M.H. Williams, Formal specification of active database functionality: a survey, in: Proceedings of the Second International Workshop Rules in Database Systems '95 Glyfada, Athens, Greece, September 25–27, 1995.
[31] M.L. Brodie, Specification and verification of database semantic integrity, PhD Thesis, Department of Computer Science, University of Toronto, Toronto, 1978.
[32] S.D. Urban, B.B. Lim, An intelligent framework for active support of database semantics, Advances in Databases and Artificial Intelligence, 1, JAI Press, 1995 pp. 167–208.
[33] A. Karadimce, S.D. Urban, Conditional term rewriting as a formal basis for analysis of active database rules, in: J. Widom, S. Chakravarthy (Eds.), Proceedings of the Fourth International Workshop on Research Issues in Data Engineering: Active Database Systems, Houston, TX, February 1994, IEEE Computer Society Press, Los Almitos, CA, 1994.
[34] R. Sturm, J.A. Mulle, P.C. Lockemann, Temporized and localized rule sets, in: Proceedings of the Second International Workshop Rules in Database Systems '95 Glyfada, Athens, Greece, September 25–27, 1995.
[35] S. Chakravarthy, A visualisation and explanation tool for debugging ECA rules in active databases, in: Proceedings of the Second International Workshop Rules in Database Systems '95 Glyfada, Athens, Greece, September 25–27, 1995.
[36] E. Benazet, H. Guehl, M. Bouzeghoub, VITAL: a visual tool for analysis of rules behaviour in active databases, in: Proceedings of the Second International Workshop Rules in Database Systems '95 Glyfada, Athens, Greece, September 25–27, 1995.
[37] O. Diaz, A. Jamie, N. Paton, Dear: a debugger for active rules in an object oriented context, in: Proceedings of the 1st International Workshop on Rules in Database Systems, 1993, pp. 180–193.
[38] O. Diaz, Tool support, in: N.W. Paton (Ed.), Active Rules in Database Systems, Monographs in Computer Science, Springer, Berlin, 1999, pp. 127–146.
[39] T. Coupaye, C.L. Roncancio, C. Bruley, J. Larramona, 3D visualization of rule processing in active databases, in: NPIV '97, Proceedings of the Workshop on New Paradigms in Information Visualization and Manipulation, November 13–14, Las Vegas, NV, 1997, pp. 39–42.
[40] T. Rische, M. Skold, Monitoring complex rule conditions, in: N.W. Paton (Ed.), Active Rules in Database Systems, Monographs in Computer Science, Springer, Berlin, 1999, pp. 81–102.
[41] E. Baralis, Rule analysis, in: N.W. Paton (Ed.), Active Rules in Database Systems, Monographs in Computer Science, Springer, Berlin, 1999, pp. 51–67.
[42] E. Baralis, S. Ceri, S. Paraboschi, Improved rules analysis by means of triggering and activation graphs, in: Proceedings of the Second International Workshop Rules in Database Systems '95 Glyfada, Athens, Greece, September 25–27, 1995.
[43] N. Bidot, S. Maabout, A model theoretic approach to update rule programs, in: F. Afrati, P. Kolaitism, (Eds.), Database Theory—ICDT '97, 6th International Conference, Delphi, Greece, January 1977, pp. 172–202.
[44] S. Ceri, J. Widom, Deriving production rules for constraint maintenance, in: Proceedings of the 16th VLDB Conference, 1990, pp. 566–577.
[45] S.K. Das, M.H. Williams, Integrity checking methods in deductive databases: a comparative evaluation, Proceedings of the Seventh British National Conference on Databases, Cambridge University Press, Cambridge, 1989.
[46] L.M.L. Delcambre, K.C. Davis, Automatic validation of object-oriented database structures, in: Proceedings of the Fifth International Conference on Data Engineering, Los Angeles, CA, 1989, pp. 2–9.
[47] A. Karadimce, S.D. Urban, Diagnosing anomalous rule behaviour with integrity maintenance production rules, in:J. Goers, A. Heuer, G. Saake (Eds.), Proceedings of the Third Workshop on Foundations of Models and Languages for Data and Objects, Aigen, Austria, 1991, pp. 77–102.
[48] B.B.L. Lim, A formal framework for the specification and enforcement of object centered constraints and triggers, PhD Dissertation, The Center for Advanced Computer Studies, University of Southwestern Louisiana, 1992.
[49] B. Martens, M. Bruynooghe, Integrity constraint checking in deductive databases using rule/goal graph, in: Proceedings of the 2nd International Conference on Expert Database Systems, Tysons Corner, VA, 1988.
[50] J. Qian, G. Wiederhold, Knowledge-based integrity constraint validation, in: Proceedings of the Twelfth International Very Large Database Conference, Kyoto, Japan, 1986.
[51] J. Qian, G. Wiederhold, Integrity constraint reformulation for efficient validation, in: Proceedings of the Thirteenth International Very Large Database Conference, Brighton, England, 1987.
[52] S.D. Urban, L.M.L. Delcambre, Constraint analysis: a tool for explaining the semantics of complex objects, in: K. Dittrich (Ed.), Proceedings of the 2nd International Workshop on Object Oriented Database Systems, Germany, September 1988, Lecture Notes in Computer Science, 334, Springer, Berlin, 1988, pp. 156–161.
[53] S.D. Urban, M. Desiderio, Translating constraints to rules in CONTEXT: a CONstrainT Explanation Tool, Proceedings of the IFIP Working Conference on Database Semantics. Object-Oriented Databases: Analysis, Design and Construction, Windermere, UK, July 1990, North Holland, Amsterdam, 1991, pp. 373–392.
[54] Active rules in database systems, in: N.W. Paton (Ed.), Monographs in Computer Science, Springer, Berlin, 1999.
[55] A. Ginsberg, Knowledge-base reduction: a new approach to checking knowledge bases for inconsistency and redundancy, in: Proceedings of the 7th National Conference on Aritificial Intelligence: AAAI 88, vol. 2, 1988, pp. 585–589.
[56] C.L. Chang, J.B. Combs, R.A. Stachowitz, A report on the expert systems validation associate (EVA), Expert Systems with Applications (US) 1 (3) (1990) 217–230.