Maintaining Data-driven Rules in Databases using an Invariant Based Language

Avigdor Gal, Opher Etzion

September 7, 1995

Technion - Israel Institute of Technology
Faculty of Industrial and Management Engineering
Haifa, 32000, Israel

Abstract

A *data-driven rule* is a rule that is activated as a result of modifications of data items in a database. Data-driven rules become very useful in applications that use a combination of rules and DBMS technologies. This paper presents the considerations in choosing an adequate programming style for representing data-driven rules based on software engineering aspects, namely, ease of use in a high-level language and better verifiability of the rule language. We show that contemporary database programming styles fail to provide the required tools and present the *invariant* based language as an adequate tool for representing data-driven rules.

Keywords: Database systems, High-level languages, Software engineering, Active databases, Temporal databases

1 Introduction and motivation

The incorporation of rules in computerized applications has evolved in recent years as a major research area. Various research efforts deal with embedding rules in applications that rely upon DBMS systems. The integration of rules and databases is effective for applications such as expert systems, real time database systems, decision support systems and simulation systems.

A *data-driven rule* is a rule that is activated as a result of modifications of data items in a database. Data-driven rules are fundamental primitives for three types of activities:

Maintaining derived data: A *derived data item* is a data item that is derived from related data in the database. In many applications, derived data is required to be persistent, that is, to be physically stored in the database, due to either efficiency reasons (many retrieval operations vs. low update frequency) or effectiveness considerations (updates in a real time environment). The value of derived

data items depends upon the value of other data items. Therefore, a natural implementation vehicle for maintaining derived data is the use of data-driven rules that are contingent upon the modification of any data item that participates in the calculation.

Enforcing integrity constraints: Integrity constraints are logical assertions that a consistent database must satisfy. The assertions refer to values of data items, thus a modification in such values may require re-evaluation of assertions. The enforcement of integrity constraints can be implemented by data-driven rules that are contingent upon the modification of data items that participate in the assertion.

Triggering external operations: External operations are operations external to the rule system that invoke a routine whose logic is not controlled by the rule system. For example, broadcasting a message or activating an alerter. The triggering mechanism can be implemented using data-driven rules that trigger the operation when data item that participates in the rule is changed.

1.1 Software engineering considerations

High-level languages should be natural, direct, self-contained and accurate. These goals can be achieved using two properties:

1. Enriching a language with semantic primitives which model the application's domain. Semantic primitives affect the ease and rapidness of programming. The ability to combine semantic concepts at the language's primitive level, facilitates an automated updating process.
2. Using an abstract language stating what the user wants to do, rather then how it is done [7]. This is achieved using the declarative programming style, that does not require the specification of all the details of each operation.

Our goal is to integrate these two properties within an update language, so that combining data-driven rules would respond to these software engineering requirements.

1.2 Motivating example

For concreteness, consider the following example.

a newspaper distributing system in Old-Man city assigns subscribers to distributors based on the city map, the subscribers' addresses, the load each distributor can handle, etc. Some data-driven rules in this application are:

r1: A change in a distributor's load results in recalculation of the distributor's commission.

r2: A change in either a distributor's load or in a load's upper bound results in the checking of an integrity constraint, verifying that the load of a distributor does not exceed the allowed upper bound for the distributer's type.

r3: Any modification in a subscriber's address, activates a heuristic algorithm for new assignment of subscribers to distributors.

The rule **r1** maintains the values of the derived data commission. It is worth noting that a family of rules that deal with the same data item can be generalized to a higher level rule presented by rule **r1**. Some of the rules in this family are:

1. When a subscriber is added to the distributor's area, increase the distributor commission.
2. When a subscriber is removed from the distributor's area, reduce the distributor commission.
3. When the commission per subscriber is changed, recalculate the distributor commission.

The rule **r2** enforces an integrity constraint. In this case, the knowledge about the dependencies among data items is sufficient to infer all the cases in which a rule is triggered. The rule **r2** is triggered if there is an increase in a distributer load or a decrease in a load's upper bound; in the latter case, the constraint is checked for each distributor affiliated with this distributor type.

The rule **r3** triggers the invocation of an external operation (heuristic procedure). In this case, the logic of the rule cannot be inferred, however, its triggering conditions and its effect on the database may be inferred by the same inference procedure that infers triggering conditions for the other two types of rules.

This paper establishes requirements, based on software engineering considerations, for a programming style that supports data-driven rules and generates criteria to check the suitability of programming styles for this task. As shown in Section 2, the contemporary database programming styles fail to meet those criteria, thus the paper introduces an improved alternative that supports data-driven rules and satisfies most of our desired requirements (Section 3). Section 4 concludes the paper with directions for further research.

2 The inadequacy of existing programming styles

Existing database programming languages can be divided into four main programming styles, namely, the imperative style, the active style, the deductive style and the script based style. Section 2.1 briefly describes each of these styles. The requirements of programming environment for data-driven rules are defined and evaluated in Section 2.2. Section 2.3 concludes this section with a summary table.

2.1 Review of relevant programming styles

Imperative Style: Imperative languages are general purpose procedural languages that do not contain database commands. Therefor, database operations are embedded in a host language, or performed by calling auxiliary routines.

Typically, the life-span of a variable in an imperative language is limited to the life-span of the routine in which it is defined. The only persistent data type in an imperative language is a *file*. Persistent imperative languages support variables whose persistence does not depend upon their data type. The life-span of these variables are longer than the creating program's life-span, and there is no need for explicit or implicit physical data movement by the programmer [1]. In addition, persistent imperative languages employ type completeness [1], where the same database operations can be applied on each data type. An example of a fully persistent language is PS-ALGOL, a language that "tolerates" all

data types, including the primitive types of different databases. As a result, data can move from one database to another.

Some persistent languages, support only partial requirements of persistence, according to the needs of the specific language. For example, Pascal/R is a persistent language that combines general purpose language (Pascal) with a data structure. A *tuple* is a primitive in the language, similar to the *record* primitive in Pascal and the type *relation of* is presented in a manner similar to *set of* in Pascal. The type *relation of* requires two parameters: the record type (which can be also a tuple) and a subset of the record's fields that is used as a primary key. A data type named *database* has the functionality of a record, except that its fields are only relations. Since *database* can be used as a parameter for files, the same program can be executed against different databases with the same structure. The language is only partially persistent, because not every variable type can appear as the first parameter of the *relation of* type, and some of the relational operators defined in the language cannot be applied for all data types.

For example, Figure 1 presents a partial program for calculating rule **r1** written in Pascal/R. It is worth noting that the rules in persistent languages are context-dependent, written in an incremental manner.

Embedded languages combine an imperative language with a query language such as SQL. Queries are written using a host language, and a pre-compiler is used to identify queries and to invoke the query language. The query variables, which are not handled by the host language, are marked using a special symbol, to separate them from the regular variables. Variables which do not exist in an imperative language are handled using special methods. For example, relations in the relational model are handled by defining a pointer that simulates the navigation among the records in the relation. Examples of embedded languages are CODASYL interfaces to COBOL, PL/I and FORTRAN.

Active style: The programming style employed by most active database models follows the *Event-Condition-Action* (E-C-A) architecture introduced in HiPAC [12]. Under this architecture a rule is composed of three components:

Event : either a database transition (e.g., when a distributor load is modified do:), or an external event such as a clock triggered event (e.g., each morning at 7:00am do:).

Condition : a collection of queries.

Action : a user defined program.

For example, part of the functionality of rule **r1** is written in E-C-A as follows:

Event:	modify Assigned-Distributor
Condition:	true
Action:	calculate Commission

When an *event* occurs, if the *condition* is evaluated to "True," the *action* is activated. Integrity constraints, derived data and other active database features are all applied as rules.

The *action* part of a rule is written in an imperative language but the control part of the application, that is, the part that monitors the decisions about rule activations, is applied by a programming style on a higher level than imperative languages.

```
program Example
type
    String          =  packed array [1..15] of char;
    SubList         =  ↑Subscriber;
    DisList         =  ↑Distributor;
    TypeList        =  ↑Dist-Type;
    Subscriber      =  record
                               Sub-Number: Integer;
                               Name: String;
                               . . .
                               Assigned-Dist: DisList;
                               Expiration-Date: Date;
                               Next-Sub: SubList
                       end;
                               . . .
var Database:          record
                               Subscribers: SubList;
                               Distributors: DisList;
                               Dist-Types: TypeList
                       end;
Begin
Procedure Insert-Subscriber;
    var
                   sl: SubList;
                   dl: DisList;
    begin
                   . . .
                   case sl.DistList.Dist-Status of
                       'LOW':    sl.Dist-List.Commission := sl.Dist-List.Commission + Low-Comm-Rate
                       'MEDIUM'sl.Dist-List.Commission := sl.Dist-List.Commission + Med-Comm-Rate
                       'HIGH':   sl.Dist-List.Commission := sl.Dist-List.Commission + High-Comm-Rate
    End

Procedure Modify-Address;
    var
                   sl: SubList;
                   dl: DisList;
    begin
                   . . .
                   case sl.DistList.Dist-Status of
                       'LOW':    sl.Dist-List.Commission := sl.Dist-List.Commission - Low-Comm-Rate
                       'MEDIUM'sl.Dist-List.Commission := sl.Dist-List.Commission - Med-Comm-Rate
                       'HIGH':   sl.Dist-List.Commission := sl.Dist-List.Commission - High-Comm-Rate
    . . . Choose new distributor
                   case sl.DistList.Dist-Status of
                       'LOW':    sl.Dist-List.Commission := sl.Dist-List.Commission + Low-Comm-Rate
                       'MEDIUM'sl.Dist-List.Commission := sl.Dist-List.Commission + Med-Comm-Rate
                       'HIGH':   sl.Dist-List.Commission := sl.Dist-List.Commission + High-Comm-Rate
    End
```

Figure 1: Partial program for rule **r1** using Pascal/R

Rule is a database primitive, and can be inserted, modified and deleted. In addition, a rule can be *fired*, skipping the event and starting by evaluating the condition. A rule can be also *disabled*, where no event would trigger it until it is *enabled* again.

Deductive style: Deductive style programming languages are derivatives of logic programming (especially PROLOG). They are used in loose or tight coupling with database languages. The deductive style supports inferences that are phrased in a subset or an extension of first order logic using recursive database queries [11].

For example, the following PROLOG style clauses, used in DATALOG [11] capture the functionality of rule **r1** that calculates the distributor's commission:

(1) Distributor(d,n1,t,s1,n2,c):- Distributor-Type(t,l1,m1,h1,m2,h2,s2), s2="Low", c:=n2*l1

(2) Distributor(d,n1,t,s1,n2,c):- Distributor-Type(t,l1,m1,h1,m2,h2,s2), s2="Medium", c:=n2*m1

(3) Distributor(d,n1,t,s1,n2,c):- Distributor-Type(t,l1,m1,h1,m2,h2,s2), s2="High", c:=n2*h1

The last element in *Distributor*, the distributor's commission c, is dependent upon the number of subscribers for that distributor (*n2*) and the commission rate (*l1, m1* or *h1*) of the appropriate distributor's type (t). The three clauses represents three different calculations, based on the distributor status (*s2*).

In most cases, combining relational database with PROLOG decreases significantly the system abilities even for simple updates, due to high complexity evolving from the recursive operations in PROLOG and the unification and resolution mechanisms.

Script style: In this programming style, rules are implemented as long term activities, using augmented petri nets called *scripts*. A script is a set of active entities, with an internal set of states that exchange messages.

An example of such a language is Taxis [9]. Taxis was originally devised as a conceptual modeling language. It supports a framework based on entities and properties with additional semantic abstractions (e.g., generalizations and aggregations). A class in Taxis is a set of entities that is a part of a class hierarchy starting with **Any Class**. **Any Class** has predefined subclasses, such as **Data Class, Exception Class** and **Transaction Class**. Properties can be attached to classes, as well as to instances. A **MetaClass** is a class that contains the Class properties. An infinite number of meta-class levels can be devised, by assigning properties to different levels of meta-classes.

A script in Taxis is a petri net of states, connected with arcs that represent transitions, along with operating conditions and communication primitives. A transition is limited to a single source and a single destination. Each operation in the script can be a transaction, another script or an exception routine. For example, Figure 2 presents a script that follows the calculation of a distributor's commission.

2.2 Requirements of programming environment for data driven rules

Conventional programming languages moved through several generations of programming environments, starting with assembly languages, moving through third generation languages (PL/I, Pascal, C, etc.), and proceeding to high level languages. In a similar way, the programming environment for rule languages that exists in many of the current models is analogous to an assembly language, supplying all the building blocks without any further abstractions. For example, in the imperative style, activation of a rule should be *hard coded* for any condition that might trigger this rule. In this section, we employ several requirements that

```
define scriptClass calculateCommission with
states
        dReadyToCalculate: ReadyToCalculate
        dLowCommission: NonInitialState
        dMediumCommission: NonInitialState
        dHighCommission: NonInitialState
transitions
        1AddSubscriber
        from dReadyToCalculate
        to dLowCommission
        conditions
        if Number-Of-Subscribers < Medium-Limit
        ...
        actions
        calculate Commission
        ...
...
endScriptClass
```

Figure 2: Partial example of a Taxis script for calculating distributor's commission

appear in the literature for high level languages. Using the analogy above, we modify these requirements to support a high level programming environment to data-driven rules. It should be noted that data-driven rules have inherent semantic properties that make the use of high level language both feasible and attractive.

Structural clarity requires the ease of use of the language from the programmer's point of view considering the tasks of writing, understanding, and debugging.

The numerous lines of code and the low level of abstraction in third generation languages burden the user in writing, understanding, and debugging these languages. In the imperative style, these problems are inherited from the host language.

In the deductive style rules are expressed in a formal language. The language of logic is clear and concise to logicians, however it cannot be effectively comprehended by programmers.

In the active style and the script style, the control part of rules, that is, the mechanism deciding which rule should be triggered, is specified in a declarative way, thus the clarity is improved relative to imperative languages. The *action* part of the rule is specified using an imperative language, thus its clarity problems are inherited.

Uniformity requires the homogeneity of the language in its syntax and logic. A programming language should be uniform and self-contained, with minimal need for lower level external routines.

Persistent imperative languages are uniform due to the fact that the database commands are syntactically integrated. The embedded languages use at least two different languages (host language and DB-related language).

Some active models, such as HiPAC, use two separate languages, one to define the control mechanism and the other to describe the actions. Other models unify all the operations in a single homogenous language.

A tightly coupled deductive languages, such as DATALOG [11], is a uniform language, while loosely coupled deductive languages employ at least two languages with different syntaxes (inference language and DB language).

Script languages are heterogenous; they employ several syntactic structures.

Abstraction level requires high level primitives embedded in the language syntax. The abstraction level is concerned with the ability to model applications, using a data model which does not have a limited view of aspects of the real world. These primitives should include terms like "event," "trigger," "derivation" and "exception handling mode." In addition, the model should support temporal rules and keep track of the schema evolution, including history of rules.

In imperative languages, there are no high level abstractions of the desired type. In deductive languages, terms such as "event" and "trigger" cannot be captured by first order logic but can be supported by more advanced logics. In the active and script languages, abstractions that are common to all types of rules exist in most of the languages, but abstractions that are specific to data-driven rules, such as "derivation" exists only in part of these models (CACTIS [5]).

All types of languages cannot capture temporal and other dimensional knowledge.

Extended data independence [2] requires the autonomy of the program from some structural and behavioral aspects in the database. There are two extended types of data independence in addition to the known types.

Situation independence: The dependencies are not situation oriented, that is, the programmer does not need to define and handle every single case in which a dependency can be realized, but general definition of the dependency is sufficient.

Referential independence: Reference connections (such as match or join operations) are determined as much as possible by the system without involving the programmer.

Referential independence is not supported by any of the existing languages of all types. All the languages require explicit reference in each case.

Situational independence is partially supported by deductive models (if the derivation is expressible in the language) and by some active models (CACTIS). Other language types do not support situational independence.

Deterministic execution: [3] requires a deterministic interpretation of update operations. Deterministic models enable the definition of well defined semantics for the execution process, thus permitting reasoning and verification of the execution process.

The imperative languages are deterministic, since programs are sequential by nature. The semantics of the program, however, are left to the programmer's discretion.

In deductive languages determinism exists in restricted forms of first order logic that do not include disjunction or existential quantifiers, e.g., Horn clauses.

In many of the active and script models, determinism is not guaranteed, due to either non-deterministic selection of rules or non-deterministic order of execution.

Consistency enforcement mechanism: [8] requires the ability to define global constraints and enforce their satisfiability in the database.

In imperative languages, consistency enforcement mechanisms is left to the programmer's control and cannot be guaranteed by the language.

Requirement	Imperative	Active	Deductive	Script
Structural clarity	-	p	-	p
Uniformity	persistent + embedded -	s	s	-
Abstraction level	-	p	p	p
Extended data independence				
Situation independence	-	p	p	-
Referential independence	-	-	-	-
Deterministic execution	p	s	s	-
Consistency enforcement	-	p	+	-
Updating redundancy	-	s	s	-

Figure 3: Requirement satisfaction comparison

In the active languages, rules can be defined for consistency enforcement. However, two major restrictions apply. First, an integrity constraint may not be phrased as a single rule which leads to a verification problem. Second, exceptions to integrity constraints cannot be handled in a general way and exception handlers are expressed using imperative programs.

In the deductive languages, any expressible constraint can be stated as an assertion (a model axiom). In the script languages, integrity constraints should be hard coded by the programmer.

Update redundancy: [5] requires the existence of a control mechanism to eliminate redundant updates that may be created from interconnected dependencies.

In imperative and script languages, update redundancy mechanisms are left to the programmer's control and cannot be guaranteed by the language. Most of the active languages do not have a redundancy control mechanism; CACTIS is an exception to this.

In deductive languages, the update redundancy control is contingent upon the implementation of the deduction process.

2.3 Conclusion

Most of the requirements are not fully supported by any of the programming language styles described above. Figure 3 compares the programming styles with respect to the above discussed requirements. We use the following notation:

+ means that the particular style fully satisfies the requirement.

- means that it is not possible to fulfill the requirement using this style.

p means that a style partially meets the requirements.

s means that it is possible to satisfy the requirement using this programming style, but only some of the implementations do so.

An analysis of the table in Figure 3 suggests that both the imperative languages and the script languages are highly inadequate to serve as a base for data-driven rules. Imperative languages lack the abstraction

level required to define these rules. Both types of languages do not supply the proper mechanism to define and maintain rules.

It is somewhat surprising to discover that active languages and deductive languages are inadequate as well. Although both types of languages handle some sort of rule types, none of the examined languages could fully satisfy the requirements for data-driven rules.

Active languages use a strict definition of rules (E-C-A), thus all types of rules should be stated in the same manner whether it is necessary or not. This property implies that the size of the rule set is relatively big even for small systems. Big sets of rules are harder to handle, and problems may occur from internal contradictions among rules.

Deductive languages use extensions of first order logic as a basis. First order logic is an accurate language with a substantial expression power for representing complex primitives. However, there is no effective routine to decide whether a statement is valid [6]. If it is valid, it may take a long time before we know it. If it is not valid, we may never be able to show it.

3 A paradigm for data-driven rules

The inadequacy of the existing programming styles, as concluded from Section 2, initiated the need to search for a new programming paradigm. In this section we present novel features of a paradigm for data-driven rules. This paradigm is based on the PARDES model [3] and its temporal extension [4]. Sections 3.1 through 3.3 present the basic premises of this paradigm; Section 3.4 examines the features relative to the requirements discussed in Section 2.

3.1 Basic premises of the paradigm for data-driven rules

3.1.1 Programming by invariants

The programming by invariants [3] paradigm in data-driven rules stems from the observation that in data-driven rules, the dependencies among data items can be inferred implicitly by the system. Programming by invariants uses high level abstractions to support dependencies among data items. We use the term *derived data item* for a data item that is being updated by a derivation rule, and the term *deriver* for a data item that participates in a derivation or a constraint rule.

The term invariant evolved from program verification theory. *Invariant* is an assertion that must hold at all times. For example, consider the following Pascal statement.

For i:=3 to 100 do A[i]:=A[i-1]+A[i-2]

This statement describes the calculation of the hundred's component in the Fibonacci series. The loop's invariant might be described as: "on the j-th entrance to the loop, every A[i] where $i < j + 2$ holds the i-th component of the Fibonacci series".

Database programming employs two sorts of invariants, as follows.

Logical invariants denote a constraint. For example, the constraint "a distributor load cannot exceed the load upper bound of that distributor" is a simplified version of the rule **r2** in Section 1. The appropriate invariant which represents this constraint,

$$Distributor\text{-}Load \leq Upper\text{-}Bound$$

stands for the formal expression "for each x, such that x is a distributor, Distributor-Load(x) $\leq$ Upper-Bound(x)". This interpretation is inferred, since we do not want to burden the system designer in using formal logic.

Computational invariants define the derived data item as an arithmetic function of other data items.

In the relational model, this type of invariants can manipulate existing information using the relational algebra operations *select, project* and *join*, e.g., "create a view that contains distributors which have load of 1.5 and above." In the context of materialized views, this view definition may be regarded as an invariant which states an actual dependency.

An invariant is included in a language in order to support data dependencies. Dependencies can operate either unidirectionally, when the change of x can affect y but not the opposite, or bidirectionally when x and y are interdependent. Figure 4 presents the schema and invariants definition of the case study.

In the invariants definitions, the first three rules are derivation rules, the next three rules are constraint rules, and the last rule invokes a routine whose logic cannot be captured by the system. The last rule indicates that an external operation called Apply-Heuristic-Assignment is triggered whenever a change in any member of the derivers list (Zipcode, Expiration-Date) occur. These types of rules are not discussed in our analysis.

Four major types of dependencies among data items are identified:

Constraint is a restriction upon a legal system state written as a global definition [7]. For example, consider the limitation on different levels of commission rate:

$$Low\text{-}Commission\text{-}Rate \leq Medium\text{-}Commission\text{-}Rate$$

In this example *Low-Commission-Rate* and *Medium-Commission-Rate* are interdependent.

Constraints are supported by the logical invariant.

Derivation is an operation required to maintain a derived data item. For example, a simplified example of an invariant representing the commission of a distributor is defined as:

$$Commission := Number\text{-}Of\text{-}Subscribers * Commission\text{-}Rate$$

In this case *Commission* is dependent upon *Distributor-Load* and *Commission-Rate*.

A derivation is supported by a computational invariant.

Class =	Subscriber	Class = Distributor-Type	
Properties=	Subscriber-Number	Properties=	Distributor-Type-Code
	Name		Low-Commission-Rate
	Address		Medium-Commission-Rate
	Zipcode		High-Commission-Rate
	Assigned-Distributor		Medium-Lower-Bound
	Expiration-Date		High-Lower-Bound
Class=	Distributor	Subscribers-Limit	
Properties =	Distributor-Number		
	Name		
	Distributor-Type		
	Distributor-Status		
	Number-of-Subscribers		
	Commission		

Invariants definition

Number-of-Subscribers	:=	count (Subscriber)
Distributor-Status	:=	'Low' when Number-of-Subscribers < Medium-Lower-Bound 'Medium' when Number-of-Subscribers < High-Lower-Bound 'High' otherwise
Commission	:=	Number-of-Subscribers * Low-Commission-Rate when Distributor-Status='Low' Number-of-Subscribers * Medium-Commission-Rate when Distributor-Status='Medium' Number-of-Subscribers * High-Commission-Rate otherwise
Number-of-Subscribers	$\leq$	Subscribers-Limit
Low-Commission-Rate	$\leq$	Medium-Commission-Rate
Medium-Commission-Rate	$\leq$	High-Commission-Rate
Assigned-Distributor	:=	Apply-Heuristic-Assignment, derivers = (Zipcode, Expiration-Date)

Figure 4: Schema and Invariants Definition

Conditional dependency either a constraint or a derivation that is conditioned upon the database state. For example, consider the rule for deriving the commission of a distributor as appear in Figure 4 above. In this rule, the *Commission* is dependent upon the distributor status as well as upon the appropriate commissions rates and the number of subscribers.

The conditional dependencies are supported by both logical and computational invariants.

Referential dependency exists whenever there is a dependency between two data items from different data groups (relations or records in various data models) and matching data items are required to support this dependency. For example, consider the constraint on a distributor load as defined in **r2** in Section 1:

$$Number\text{-}of\text{-}Subscribers \leq Subscribers\text{-}Limit$$

While *Number-Of-Subscribers* is a property of *Distributor*, *Subscribers-Limit* is a property of *Distributor-Type*. The matching is done by the property *Distributor-Type* in *Distributor*.

A referential dependency is hidden in the structure of the invariant and thus supported by the language.

3.1.2 Automatic translation mechanism

An automatic translation mechanism allows the user to design data-driven rules freely without being concerned with low level programming considerations. The automatic translation mechanism consists of:

Intra-class matching Two properties of the same class, that participate in a rule, infer the rule's effect on properties of the same instance. For example, the rule:

$$Low\text{-}Commission\text{-}Rate \leq Medium\text{-}Commission\text{-}Rate$$

Inter-class matching Dependencies among properties of different classes require more sophisticated matching process than the previous type of matching. For example:

$$Number\text{-}of\text{-}Subscribers := count\ (Subscriber)$$

There is a need to determine which instances of *Subscriber* affect the values of a certain instance of *Distributor*, or the converse, i.e., which instances of *Distributor* are affected by a change in a given instance of *Subscriber*. This matching is done explicitly in conventional models by designating the conditions for this match (e.g., *Join* in relational algebra) or by using *path expressions*. The automatic matching protocol attempts to infer such a matching by using semantic equivalences. Properties are *semantically equivalent* if they are mapped to the same set and have the same meaning. Automatic matching simplifies the language and makes it *matching independent*, hence classes can be united or partitioned without the need to change the set of invariants.

Although the invariant language allows the user to write an invariant with minimal need for explanation, there are cases in which an invariant is ambiguous. In the previous example, suppose there are two different properties in *Subscriber* that refer to the distributor's number: *Assigned-Distributor* which is

a list of all the distributors that were ever assigned to the subscriber, and *Current-Assigned-Distributor* which is the distributor currently assigned to the subscriber. In this case, the system cannot determine the matching attribute and the system designer is asked to decide between the two possibilities. The system designer may also specify the desired matching in the invariant language as follows:

$$Number\text{-}of\text{-}Subscribers := count\ (Subscribers)\ over\ Current\text{-}Assigned\text{-}Distributor$$

The "over..." clause is added to avoid ambiguity.

Mutual excluding conditions: An Invariant may include a conditional expression such as:

$$\begin{aligned} Distributor\text{-}Status \quad := \quad & \text{`}Low\text{'}\ when\ Number\text{-}of\text{-}Subscribers < Medium\text{-}Lower\text{-}Bound \\ & \text{`}Medium\text{'}\ when\ Number\text{-}of\text{-}Subscribers < High\text{-}Lower\text{-}Bound \\ & \text{`}High\text{'}\ otherwise \end{aligned}$$

The conditions must be mutually exclusive, otherwise a problem of nondeterminism may occur. The mutual exclusiveness is enforced by assuming that the order of conditions determines the relative priority. Thus, each condition is evaluated as a conjunction (*and* relationship) of itself and the negations of all the previous conditions. For example, in the previous invariant, the second condition is interpreted as:

$$Number\text{-}of\text{-}Subscribers < High\text{-}Lower\text{-}Bound \wedge \neg\ (Number\text{-}of\text{-}Subscribers < Medium\text{-}Lower\text{-}Bound)$$

that is:

$$Medium\text{-}Lower\text{-}Bound \leq Number\text{-}of\text{-}Subscribers < High\text{-}Lower\text{-}Bound$$

3.1.3 Increasing efficiency

After resolving all the required matchings, the set of invariants is translated into a *dependency graph*, that models the *data-driven dependencies* among the various data-elements. This graph determines the transitive closure of an update operation and facilitates the reasoning about the update process including optimizations. Figure 5 presents a sketch of the dependency graph in the distributor example, based on Figure 4.

The dependency graph is used to infer the actions that should be taken as a result of a data item modification. For example, for the operation:

$$\begin{aligned} Commission \quad := \quad & Number\text{-}of\text{-}Subscribers * Low\text{-}Commission\text{-}Rate \\ & \qquad when\ Number\text{-}of\text{-}Subscribers < Medium\text{-}Lower\text{-}Bound \\ & Number\text{-}of\text{-}Subscribers * Medium\text{-}Commission\text{-}Rate \\ & \qquad when\ Number\text{-}of\text{-}Subscribers < High\text{-}Lower\text{-}Bound \\ & Number\text{-}of\text{-}Subscribers * High\text{-}Commission\text{-}Rate \\ & \qquad otherwise \end{aligned}$$

the following actions are inferred:

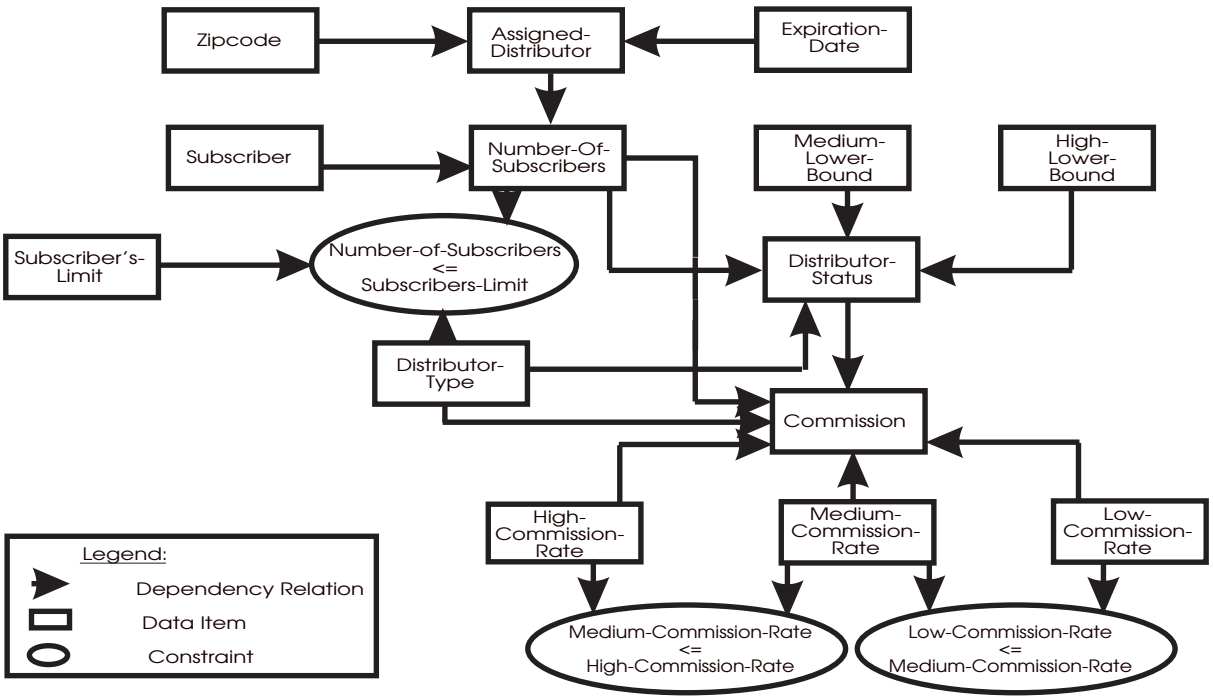

Figure 5: Dependency graph of the distributor's database

1. When a *Subscriber* is assigned to a *Distributor*: recalculate the *Commission* according to the appropriate range of *Number-of-Subscribers*.
2. When a *Subscriber* is erased from the *Distributor*'s assignment: recalculate the *Commission* according to the appropriate range of *Number-of-Subscribers*.
3. When a *Subscriber* is reassigned to another *Distributer*: recalculate the *Commission* of both *Distributors*.
4. When *Low-Commission-Rate* is modified: modify the commission of all the relevant *Distributors* of the same *Distributor-Type*. The same applies for *Medium-Commission-Rate* and *High-Commission-Rate*.
5. When a *Medium-Lower-Bound* is modified: recalculate the commission of all the relevant *Distributors* of the same *Distributor-Type*. The same applies for *High-Lower-Bound*.
6. When the *Distributor-Type* of a *Distributor* is modified: recalculate the *Commission* according to the new *Distributor-Type* parameters.

The combination of the dependency graph and the automatic matching mechanism makes the invariant definition directly executable. No extra programming is needed to maintain the invariant.

The dependency graph enables the support of situation independence. As a result, the number of rules is drastically reduced in comparison to any other alternative.

The update control mechanism is based on the dependency graph. The order of update operations, resulting from a user transaction, is determined by using topological order on the graph, to eliminate redundancy in update operations. This mechanism is similar to the one in **CACTIS**.

3.1.4 Further premises

Further premises of the invariant language and its supporting model are as follows.

Uniqueness: Each derived data-element appears on the left hand side of exactly one invariant. The uniqueness property enables the creation of a deterministic set of rules with no conflicts. It is worth noting that it is impossible either to maintain determinism or to prevent conflicts in data-driven rules, using the E-C-A approach, simply because a single rule cannot capture all the different cases when a rule should be triggered.

Non Reflexiveness: An instance of any data element may not be derived directly or indirectly from its own value. A derivation from the *old* value of a data-element (the value that existed prior to the transaction start) is allowed. For example:

not allowed:	Y := Y + 1
allowed:	Y:= old (Y) + 1

Since the value of old (Y) is not changed during the transaction, no infinite loop is generated. Because loops are forbidden, the use of the same data item both on the left hand side and on the right hand side, imply the use of the old value in the right hand side, without explicitly stating it.

3.2 Combining temporal data with data-driven rules

In recent years, the discipline of *temporal databases* has been constructed, aimed to add the time dimension to databases [10]. This discipline adds a historical dimension to the database, by maintaining different values for the same data item for different time points. This section surveys the need to augment the *data-driven rule* to support temporal features, especially proactive and retroactive processing, and discusses the types of such rules and their general logic. Since in our model we treat retroactive updates and proactive updates in the same manner, we shall use the term "non current" in place of both retroactive and proactive.

A *non current update* is an update operation that modifies past or future values of data items.

A *non current rule* is a rule whose action includes a non current update.

A *non current rule activation* is the application of a rule to past or future states.

The above definitions indicate that rules can affect data non currently in two main ways, either due to non current rules or due to non current activation of rules. The latter case can occur for two reasons:

1. a rule is introduced in the system with past or future validity time interval(s).
2. A non current update occurred, and the updated data item(s) trigger a rule which is valid at a non current time.

The temporal dimension effect is presented in the following examples.

Retroactive Update: In March 1993, it was decided that the *High-Lower-Bound* for *Distributor-Type-Code* = 1 is modified from 150 to 130, and that this change is done retroactively as of January 1993.

Proactive Update: In June 1993, it was decided that the *High-Commission-Rate* for all *Distributor-types* will be raised 10 percent starting September 1993.

Retroactive Rule: In July 1993 it was decided that the *Low-Commission-Rate* for each *Distributor-Type* is dependent upon the *Medium-Commission-Rate* and is captured by the invariant
*Low-Commission-Rate := Medium-Commission-Rate * 0.8*
This rule is retroactively applied to April 1993.

Proactive Rule: In July 1993 it was decided on a new heuristic allocation routine that will be valid starting from January 1994.

3.2.1 Temporal extension to the data model

The support of the temporal component is carried out using an extension of the basic data model. In the basic model, an instance of each property is called a *variable* and has a variable state associated with it. A *variable state* is a value bounded to the property data type. To support temporal information, each *variable state* becomes a collection of state elements. A *state element* is a pair: $<$ *value, temporal extension* $>$, where a *temporal extension* is a triple $< t_x, t_d, t_v >$ that represent the following time types.

Transaction Time (t_x) is the commit time of the transaction which updated the variable state.

Decision Time (t_d) is the decision occurrence time in the real world. For example, if a decision regarding a distributor type was decided in March 1993, and inserted to the database in April 1993, t_d would be March 1993, and t_x would be April 1993. The decision time is used to maintain the correct order of happenings in the real world, which is not always satisfied by the transaction time. This time value becomes extremely important in databases with real time constraints.

Valid Time (t_v) is the set of time points in which the decision maker believes that this value reflects the object's value in the real world. t_v is expressed by a time-point or an interval [t_s, t_e] or a collection of intervals and time-points.

All time types are not restricted to the data level. Meta data entities also have different time types which limit their effects on the database. For example, t_v values may be associated with: objects, designating the object life span, variables, rules, generalizations, existence of properties in classes and classification of objects to classes.

3.2.2 Types of Temporal Rules

The effect of the temporal dimension on rules is materialized in several ways:

The applicability of rules: A rule is a meta data object which is an instance of the "Rule" class. Properties of this class stand for assignments in the derivation case and assertions in the constraint case. If a rule is modified then the *assignment* variable of this rule has more than one state-element, applicable to different time points. For each update, a decision should be made, to select the rule or rules that are applicable for the validity interval of this update. Examples for this type of effect are the last two

examples shown above. For example, in the retroactive rule change presented above, there are two possibilities for the calculation of the *Low-Commission-Rate* in April 1993.

Temporal conditions: As discussed in Section 3.1, each rule may have several conditional assignments or assertions. The conditions may include operations that refer to any of the time perspectives, as well as to other temporal expressions. For example, consider the following rule.

Annual-Commission := *0.01 * Total-Dividend*
when Distributor-Status='Low' in [NOW(), NOW()-1year]
*0.02 * Total-Dividend*
when Distributor-Status='Medium' in [NOW(), NOW()-1year]
*0.05 * Total-Dividend*
when Distributor-Status='High' in [NOW(), NOW()-1year]

This rule calculates the annual commission, based on the lowest status of the distributor in the preceding year.

Temporal actions: The *action* part of a rule might affect time points that are different than the one in which the rule is evaluated. In this case, the t_v itself is determined by the rule and not derived as a function of other t_v values. For example:

Annual-Commission := *0.01 * Total-Dividend in [NOW()+2month, ∞]*
when Distributor-Status='Low' in [NOW(), NOW()-1year]
*0.02 * Total-Dividend in [NOW()+1month, ∞]*
when Distributor-Status='Medium' in [NOW(), NOW()-1year]
*0.05 * Total-Dividend in [NOW(), ∞]*
when Distributor-Status='High' in [NOW(), NOW()-1year]

The above rule delays the low annual commission for two month and the medium annual commission for one month.

The major problem of rules with temporal actions is a loop problem defined in the science fiction literature as "the grandfather paradox": a rule is activated in time point τ_1 and affects τ_2. As a result, a series of changes has occurred in the database, in such a manner, that the reason for activating the rule in τ_1 no longer exists. By extending the "dependency graph" technique, discussed in Section 3.1, these types of rule may be combined naturally and safely in databases.

The principles of the temporal extension are further discussed in [4]. The implementation aspects are now in an early prototype phase.

3.3 Compatibility of the described paradigm with the requirements

In this section we compare the paradigm described in the previous section with the requirements introduced in Section 2.

Structural clarity: The invariant language consists of declarative statements and contains only the minimal details needed for disambiguity. For example, the invariant
Number-of-Subscribers $\leq$ *Subscribers-Limit*
is easier to write then its formal interpretation:
$\forall$ *d* $\in$ *Distributor: [Number-of-Subscribers(d)* $\leq$ *Subscriber-Limit(t)* | *t=Distributor-Type(d)*]
or its equivalent path expression:
d.Number-of-Subscribers $\leq$ *d.Distributor-Type.Subscribers-Limit.*
We eliminate the variables *d* and *t* and the matching condition.

The language is unambiguous yet not a formal one. This increases the ease of use of the language.

The situation independence property significantly reduces the size of the rule set, thus improving the manageability of the program.

Uniformity: The invariant language is an integral part of the schema definition language. There is a uniform language that contains the static schema, the update logic, the exception handling definitions, the transaction management parameters and the retrieval operations. The same syntactic structure is kept in all these operations. An exception is an external operation, in which the calling environment is consistent with the uniform syntax. The expressive power of the model supports most of the required operations, thus the need for use of external operations is minimized.

Abstraction level: The schema and invariant language supports high level abstractions. In the schema definition it supports an extended set of semantic abstractions (classification, association, generalization, aggregation and partition). At the invariant level it supports "derivations" and "constraints." Temporal rules are used through the "state element" definition and the extensions in data-driven rules.

Extended data independence: Situational independence is fully supported by using the dependency graph and defaults based on the *filter* concept. Referential independence is supported in inferable cases by the *automatic matching* component. In cases where automatic matching is impossible (due to ambiguity or lack of information), the user is addressed with a specific question.

Deterministic execution: Deterministic execution is enforced by combination of the following premises: the **uniqueness** premise guarantees that there are no conflicting rules that update the same derived data item and are triggered by the same operation. The **mutual exclusiveness** premise guarantees that within this unique rule there is at most one condition that applies to each case. Therefore, only deterministic updates can be generated.

Consistency enforcement mechanism: The invariant definitions are used as consistency assertions. It traces all the inconsistencies generated by a change in the database relative to these assertions. The consistency is enforced without a need for any additional programming.

Update redundancy: The topological order of the dependency graph coupled with the nonreflectiveness premise eliminates loops, avoids update redundancies, and guarantees minimal number of update operations.

4 Conclusion

The major contribution of this study is the construction of a conceptual approach and supporting mechanisms to data-driven rules. Data-driven rules have an important role in many applications that maintain complex relationships between data items, or interdependencies between various parts of the database. Data-driven rules are handled by contemporary models as part of the general rule language. As shown in Section 2, all contemporary language types fail to deal with data-driven rules according to proper software engineering criteria. We proposed a model that improves the handling of data-driven rules by using the inherent semantic properties of such rules and meets the requirements that were discussed in the literature for high level languages.

The PARDES project aims at the research and the implementation of data-driven rules. In this project, we currently extend the basic PARDES model, that uses all the properties described in Section 3, as described in Section 4. In addition, there are several extensions that require further research:

Supporting event-driven rules: Event-driven rules differ from data-driven rules by being triggered by events rather then by modification of data items. Event-driven rules do not have the semantic properties that permit them to be phrased as invariants. In order to make the PARDES a complete model, event-driven rules should also be supported. To support event-driven rules in a uniform style we phrase event-driven rules in an invariant-like syntax with the addition of an event list. Examples:

*High-Commission-Rate := old(High-Commission-Rate) * 1.1,*
events=(Regular-Raise, Special-Raise)
*Medium-Commission-Rate := old(High-Commission-Rate) * 1.1,*
events=Regular-Raise

In this case, *Regular-Raise* and *Special-Raise* are events that are signaled externally.

Although this derivation rule is not an invariant, we can take advantage of both the high level properties (abstraction, situational independence, referential independence, etc.) and the use of dependency graphs to model the flow of application. Many of the contemporary active models which support event-driven rules fail to achieve this goal.

Distributed PARDES: Another extension to the PARDES model is to use the above mentioned model in order to support interdependencies among several physical fragments of the database. A distributed model with a central data dictionary, extending the situation interpreter and the matching process is described in [2]. An extension of this model to a distributed data dictionary model with negotiation protocols among the various data dictionaries is currently under construction.

Implementation issues: Within the PARDES project we have devised a prototype implementation of a database supporting data-driven rules, using the programming by invariants method. The prototype is written in C++ using btreive as an access method.

The first prototype supports the invariant component for each invariant, and includes full implementation of this component. A second prototype including the other components are now in the design phase.

Further research will extend the prototype implementation, apply some complex applications using the model and deal with issues of adding deductive capabilities, devising interfaces with existing databases, so that PARDES will be able to serve as a coordinator between existing applications (*legacy systems*) and extending the model to support several types of AI applications.

Acknowledgments

This temporal and multi-dimensional study is a continuation of work that has been done with Arie Segev.

References

[1] : M.P. Atkinson, O.P. Buneman - Types and Persistence in Data Base Programming Languages, *ACM Computing Surveys 19(2)*, June 1987.

[2] : O. Etzion - Active Interdatabase Dependencies, to appear in *Information Sciences*, 1993.

[3] O. Etzion - PARDES-A Data-Driven Oriented Active Database Model, *SIGMOD RECORD, 22(1)*,pp. 7-14, Mar 1993.

[4] A. Gal, E. Etzion, A. Segev - Representation of Highly-Complex Knowledge in a Database, *Journal of Intelligent Information Systems, 3(2)*, Mar. 1994.

[5] : S. Hudson, R. King - CACTIS: A Database System for Specification Functionally Defined Data, *proc. IEEE OOBDS Workshop*, 1986.

[6] : D. Israel - Notes on Inference: A Somewhat Skewed Survey, *in: M.L. Brodie, J. Mylopoulos (eds)-On Knowledge Base Management Systems, Springer-Verlag*, 1986.

[7] : P.E. London, M.S. Feather -Implementing Specification Freedoms, *Science of Computer Programming 2, Amsterdam: North-Holland Publishing Company*, pp. 91-131, June 1989.

[8] : M. Morgenstern - Constraint Equations: Declarative Expression of Constraints with Automatic Enforcement, *Proc. VLDB*, pp. 291-300, 1984.

[9] : B. Nixon, L. Chung, D. Lauzon, A. Borgida, J. Mylopoulos, M. Stanely - Implementation of compiler for a Semantic Data Model: Experience with Taxis, *Proc. ACM SIGMOD 1987 Annual Conf., San Francisco, CA*, pp. 118-131, May 1987.

[10] R. Snodgrass, I. Ahn - Temporal Databases, *IEEE Computer 19*, pp. 35-42, Sep 1986.

[11] : J.D. Ullman - Implementation of Logical Query Language for Databases, in: *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1985

[12] : Xerox Advanced Information Technologies - HiPAC: A research Project in Active, Time-Constrained Database Management, *Final Technical Report, XAIT-89-02*, July 1989.