Data Driven and Temporal Rules in PARDES

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

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

Arie Segev
Haas School of Business, University of California and Information
& Computing Sciences Division, Lawrence Berkeley Laboratory,
Berkeley, CA 94720, USA

Abstract

Data driven rules is one of the important rule types that are used by database applications. This paper analyzes requirements for a programming paradigm appropriate for the support of data-driven rules, states the linguistic paradigm, discusses its supporting architecture and shows an extension of the model to support temporal functionalities, especially retroactive and proactive processing. The focus in this paper is on the software engineering aspects of the proposed model: ease of use through a high-level language and improving the verifiability of the rule language.

1 Introduction and motivation

The incorporation of rules in computerized applications evolved in the recent years into a major research area. Various research efforts deal with embedding of many types of rules in applications that interact with DBMS systems. The integration of rules and databases is beneficial to applications that involve constraints enforcement, expert systems, real time database systems, etc.

A *data driven rule* is a rule that should be activated as a result of modifications of certain data elements in a database.

A *temporal data driven rule* is a *data driven rule* supporting the use of the temporal dimension in rules, example: a retroactive activation of such a rule.

Throughout this paper we shall use the following example:

A newspaper distributing system in Old-Man city assigns subscribers to distributors based on the city map, the subscribers' addresses 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 a distributor's load results in an integrity constraint's check verifying that the load does not exceed the upper bound for the distributor type.

r3: Any modification in a subscriber's address, activates a heuristic algorithm for a new distribution.

Data driven rules are useful for three types of activities:

Maintaining derived data: Derived data elements are generated from other data elements. The technological advances which made secondary storage relatively inexpensive, increased the utilization of derived data. 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 (such as: real time constraints). Since the value of a derived data element is dependent on the values of the data elements used to derive it, rules whose action is contingent on the modification of data elements, are a natural implementation vehicle. The rule **r1** of the above example maintains a derived data element (commission).

Enforcing integrity constraints: Integrity constraints are logical assertions that a consistent database should satisfy. The assertions refer to values of data elements, thus a modification of 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 elements that participate in the assertion. The rule **r2** is a rule enforcing an integrity constraint.

Triggering external operations: External operations are operations that are external to the rule system and invoke a routine whose logic is not captured by the rule system. The rule **r3** is a rule triggering the invocation of a heuristic procedure. Other examples are: broadcasting a message, activating an alerter [9] etc.

There are two useful observations about the *first two types* of data driven rules:

1. The knowledge about dependencies among data elements is sufficient to infer all the cases in which a rule has to be triggered. For example, the rule **r2** is triggered if either there is an increase in a distributor's load or a decrease in a load's upper bound; in the latter case, the constraint has to be checked for each distributor affiliated with this distributor type.

2. A family of rules that deal with the same data element can be generalized to a higher level rule using the knowledge about dependencies. For example, the rule **r1** stands for a group of rules such as:

   (a) When a subscriber is added to the distributor's area, increase the distributor commission.

   (b) When a subscriber is removed from the distributor's area, reduce the distributor commission.

   (c) When the commission per subscriber is changed, recalculate the distributor commission.

In the third type of rule, the logic of the rule cannot be inferred. However, its triggering conditions may be inferred by the same inference procedure that infers triggering conditions for the other two types of rules.

In this paper we devise a data driven rule model that exploits these observations to create a simple and concise language and supporting mechanism for expressing these rules. In the search for a programming language style we explored several contemporary styles and found that none of them meets the requirements of our desired functionality; the requirements and the comparison are shown in Section 2. Section 3 described the PARDES model for maintaining data driven rules. Section 4 discusses the temporal component, Section 5 concludes this paper.

2 The inadequacy of existing programming styles

2.1 Review of relevant programming styles

Existing database programming languages can be classified into four main programming styles: the imperative style, the active style, the deductive style and the script based style.

Imperative Style: Imperative languages are general purpose procedural languages. Originally, they did not contain database commands; thus database operations are embedded in a host language, or performed by calling auxiliary routines.

Persistent imperative language is a programming language that supports variables whose life-span are longer then the creating program's life-span, without a need for explicit or implicit physical data movement by the programmer [5]. Example of persistent languages are PS-ALGOL [3] and AMBER [10].

Embedded languages combine an imperative language with a query language such as SQL. The queries are combined with a host language and a pre-compiler is used to identify a query and to make an external call

for the query language; examples of embedded languages are CODASYL interfaces to Cobol, PL/I and Fortran [4].

Active style: The programming style employed by most active database models (HiPAC [11], Ode [19], Starburst [24] etc.) follows the *E-C-A* (*Event-Condition-Action*) [25] architecture. Under this architecture a rule is composed of three components.

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

Condition : a collection of queries.

Action : a user defined program.

Active database languages apply programming of a higher level relative to imperative languages in the control part of the application, that is, monitoring the decisions about rule activations. The "actions" are still written in an imperative language.

Deductive style: Deductive style programming languages are derivatives of logic programming. 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. Examples of languages associated with this programming style are DATALOG [34] and KB2 [35].

Script style: In this programming style, rules are implemented as long term activities, using augmented petri nets [28] called *scripts*. A Script is a set of active entities, with an internal set of states that exchange messages. Examples of such languages are TAXIS [27] and Galileo [1].

2.2 Requirements of programming environment for data driven rules

Conventional programming languages migrated through several generations of programming environments: starting with assembly languages, moving through third generation languages (PL/I, Pascal, C etc.) and continuing to high level languages. In a similar way, the programming environment for rule languages that exists in many of the current models is analogues to an assembly language, supplying all the building blocks without any further abstractions. Example: in the imperative style, activation of a rule must be *hard coded* for any condition that might trigger this rule. In this Section, we employ several requirements that appear in the literature for high level languages. Using the analogy above, we modify these requirements to support a high level programming environment for

data driven rules. Note that data driven rules have inherent semantic properties that make the use of high level languages both feasible and attractive.

Structural clarity: requires the ease of use of the language with respect to the tasks of writing, understanding and debugging [6].

The problems of using third generation languages are well documented [21]; the massive code and the low level of abstraction makes it difficult to write, understand and debug. In the **imperative style**, these problems are inherited from the host language.

In the **deductive style** rules are expressed in a formal language. Empirical studies [8] determined that the language of logic is clear and concise to logicians, however it cannot be effectively comprehended by an ordinary programmer.

In the **active style** and the **script style** there is a declarative component: the control model, deciding about the rule triggering. This feature improves the clarity relative to imperative languages. The "action" part of the rule is still specified using an imperative language, and thus inherits its clarity problems.

Uniformity: requires the homogeneity of the language in its syntax and logic [2]. 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. However, the embedded languages use at least two different languages (host and DB-related).

Some active models, such as Hipac [25], use two separate languages, one to define the control mechanism and the other to describe the actions. Other models, such as Postgres [32], unify all the operations in a single homogenous language.

A tightly coupled deductive languages, such as DATALOG [34], is a uniform language, while loosely coupled deductive languages employ at least two languages with different syntax (inference language and DB language).

Script languages are heterogenous; they employ several syntactic structures.

Abstraction level: the requirement is to support high level primitives embedded in the language syntax [20]. These primitives should include terms like "event", "trigger", "derivation", "exception handling mode" etc.

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, such as situational calculus [23]. 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 [22]).

Extended data independence: requires the autonomy of the programmer from physical aspects in his programming. In addition to the traditional types of data independence [12] There are two extended types of required data independence [15]:

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 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: requires a deterministic interpretation of update operations [16]. Deterministic models enable 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. Due to the low level programming, the semantics of the program is 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, example: Horn clauses.

In many of the active and script models, determinism is not guaranteed, due to either non deterministic selection of rules (example: Postgres) or non deterministic order of execution (example: HiPac).

Consistency maintenance mechanism: [26] requires the ability to define global constraints and enforce them in the database.

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

In the active languages, rules can be defined for consistency enforcement. However, two major restrictions apply: an integrity constraint may not be

phrased as a single rule which leads to a verification problem; exceptions to integrity constraints cannot be handled in a general way, exception handlers are expressed in imperative programs.

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

Update redundancy: [22] 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 these programming language styles. Figure 1 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.

3 The PARDES Model

The conclusion of Section 2, namely, the inadequacy of the existing programming styles, motivated the search for a new programming paradigm. In this section we present the PARDES paradigm using an **invariant language**. Section 3.1 presents the basic premises of this paradigm; Section 3.2 describes the PARDES architecture and components; Section 3.3 examines the PARDES model relative to the requirements presented in Section 2.

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 maintenance	-	p	+	-
Updating redundancy	-	s	s	-

Figure 1: Requirement satisfaction comparison

3.1 The PARDES Model - Basic Premises

The **programming-by-invariants** paradigm in data driven rules stems from the observation that in all *data-driven* rules there are explicit or implicit dependencies among the data elements. We use the term **PDI** (Persistent Derived Information) for a data element that is being updated by a derivation rule, and the term **deriver** for a data element that participates in a derivation or a constraint rule.

In the derivation case, a modification of any of the derivers' instances triggers (conditionally or unconditionally) an update operation to the relevant instances of the derived data-element. We refer to such a dependency as a *computational invariant*. In the integrity constraint case, a modification of any of the participants might trigger[1] the re-evaluation of the constraint. We refer to this type of invariants as *logical invariants*. The *data-driven* dependencies in an application are defined as **executable entities** using these types of invariants only, without involving any extra programming.

The premises of the invariant language and its supporting model are as follows:

Executability : The Invariant definition is directly executable. No extra programming is needed to maintain the invariant. The execution process guarantees **soundness** (all the invariants are maintained) and **completeness** (Each consistent state is reachable).

Mutual Exclusiveness : An Invariant may include a conditional expression such as:
X:= Y, when Z $>$ 0 ; 2*Y when Z $\leq$ 0.
The conditions must be mutually exclusive. The mutual exclusiveness is

[1] An optimization mechanism may be applied to eliminate redundant triggerings.

enforced by assuming that the order of a condition determines the relative priority. Thus, each condition is conjuncted with the negations of all the previous conditions. Example:
X:= Y, when Z > 0 ; 2*Y when W > 0
the second condition is interpreted as:
$W > 0 \wedge \neg (Z > 0)$

Uniqueness : Each derived data-element appears on the left hand side of exactly one invariant.

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. Example:
Y := Y + 1 is not a valid assignment, while Y:= old (Y) + 1 is.
Since the value of old (Y) is not changed during the transaction, no infinite loop is generated.

Figure 2 is an example of schema and invariants definitions
Comments:

1. Each PDI has an associated unique invariant.

2. Reference matching among different classes is inferred where possible. Example: in the first derivation rule, there is a matching between Subscriber and Distributor based on the condition
Subscriber.Assigned-Distributor = Distributor. Distributor-Name.
See Section 3.2 for discussion.

3. The first two rules are derivation rules, the third rule is a constraint rule, the last rule applies an external rule whose logic cannot be captured by the system (this rules uses an external GIS system).

4. The last rule indicates that an external operation called Apply-Heuristic-Assignment is triggered whenever a change in any member of the derivers list (Zip-code, Expiration-Date) occurs. These types of rules will not be discussed in our analysis.

5. The Commission rule is conditioned. The conditions are mutually exclusive and priority is assumed according to the order.

3.2 The PARDES architecture

The main components of this architecture are:

1. An extended schema that includes a set of invariants.

```
Class =          Subscriber
Properties=      Subscriber-Number
                 Name
                 Address
                 Zip-code
                 Assigned-Distributor [PDI]
                 Expiration-Date

Class =          Distributor
Properties =     Distributor-Number
                 Name
                 Distributor-Type
                 Number-of-Subscribers [PDI]
                 Commission [PDI]

Class=           Distributor-Type
Properties =     Low-Commission-Rate
                 Medium-Commission-Rate
                 High-Commission-Rate
                 Medium-Lower-Bound
                 High-Lower-Bound
                 Subscribers-Limit


Number-of-Subscribers :=count (Subscriber)

Commission            :=Number-of-Subscribers * Low-Commission-Rate
                          when Number-of-Subscribers < Medium-Lower-Bound
                        Number-of-Subscribers * Medium-Commission-Rate
                          when Number-of-Subscribers < High-Lower-Bound
                        Number-of-Subscribers * High-Commission-Rate
                          otherwise

Number-of-Subscribers ≤ Subscribers-Limit

Assigned-Distributor  :=Apply-Heuristic-Assignment,
                        derivers = (Zip-code, Expiration-Date)
```

Figure 2: Schema and Invariants Definition

2. A **Translator** that includes an **automatic matching** and creates a **dependency graph**.

3. A **Situation interpreter** that interprets the **dependency graph** to determine which action should be taken when a given situation occurs.

4. A **Run-Time Controller** that controls the order of execution and eliminates redundant updates.

5. An **Exception-Handling Component** that determines the actions to be taken for any consistency violation.

6. A **Transaction Model** that determines the atomicity and synchronization of update operations.

A short description of these components follows:

Translator: A program that translates the schema and invariant definitions to a **dependency graph**. A novel feature of the **translator** is the **automatic matching** component.

Automatic Matching: Dependencies among properties of different entities may occur when a PDI belongs to a class, while a deriver of the same PDI belongs to another class. example:
Number-of-Subscribers := count (Subscribers)
There is a need to determine which instances of Subscriber affect the values of a certain instance of Distributor, or conversely: which instances of Distributor are affected by a change in a given instance of Subscriber. In conventional models this matching is done explicitly by designating the conditions for this match[2] 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*. Classes can be united or partitioned without the need to change the set of invariants.

Dependency Graph: The set of invariants is compiled (after resolving all the required matchings) to a dependency-graph, which 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. A similar graph has been proposed in [31] in the context of transaction control.

Situation Interpreter: The **dependency graph** is used to infer the action that should be taken. Example: for the operation:

[2] *Join* in relational algebra is a kind of "matching".

Commission:= Number-of-Subscribers * Low-Commission-Rate
when Number-of-Subscribers < Medium-Lower-Bound
Number-of-Subscribers * Medium-Commission-Rate
when Number-of-Subscribers < High-Lower-Bound
Number-of-Subscribers * High-Commission-Rate
otherwise

the following actions are inferred:

1. when a *Subscriber* is assigned to a *Distributor*: recalculate the *Commission* according to the appropriate value of *Number-of-Subscribers*.
2. when a *Subscriber* is removed from the *Distributor*'s assignment: recalculate the *Commission* according to the appropriate value of *Number-of-Subscribers*.
3. When a *Subscriber* is re-assigned to another *Distributor*: 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 situation interpreter enables the PARDES model to support **situation independence**. In this case, the number of rules is drastically reduced in comparison to any other alternative.

Run-Time Controller: The Update Control mechanism is similar to the one in **Cactis** by the fact that the order of update operations is determined by using topological order on the dependency graph, to eliminate redundancy in update operations.

Exception Handling Component: All the models discussed above allow the specification of Exception-Handlers *only* in an *imperative* way, at the system designer's discretion. The PARDES model provides Exception-Handling abstractions as a part of the invariant language[13], as well as support of different types of null and defaults. The abstractions stand for generic strategies for handling exceptions in cases of: syntactic (range) violations, referential integrity violations, constraints violations and direct modifications of derived updates (bypassing the derivations).

Flexible Transaction Mechanism: There are two decision problems in the transaction model of an active database:

1. Should an operation to update a PDI instance be triggered by modifications of any of its derivers?
2. If the answer to the first decision is positive then:
   (a) Should the PDI be updated synchronously?
   (b) Is the system responsible to ensure consistency at all times with respect to the deriver's update operation?

The results of these two decisions determine the materialization strategy [30]. The topic and associated algorithms are discussed in [17]. The decision about assigning a consistency mode to a rule may be recommended by an optimization model that improves its own results by getting feedback based upon the application's behavior. This issue is discussed in [7]. This flexible mechanism becomes possible only due to existence of the **dependency graph** and the **executability** properties of the PARDES model.

3.3 Discussion - compatibility of the PARDES model with the requirements

In this section we analyze the PARDES model with respect to 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 (ambiguities are resolved by prompting the system designer to choose among well-defined alternatives), 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 improves 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, exception handling definitions, transaction management parameters and retrieval operations. The same syntactic structure is kept in all these operations. An exception to that are external operations, in

which the calling environment is consistent with the uniform syntax. The expressive power of the PARDES model supports most of the required operations, thus the need for use of external operations is limited.

Abstraction level: The PARDES language supports high level abstractions. In the schema definition it supports an extended set of semantic abstractions (classification, association, generalization, aggregation and partition). In the invariant level it supports "events", "derivations" and "constraints". In the exception handling component, the exception handling modes, which are high level abstractions, are supported. In the transaction management component, the materialization strategies supported are also high level abstractions.

Extended data independence: Situational independence is fully supported by the *situation interpreter*. Referential independence is supported in inferable cases by the *automatic matching* component of PARDES. In cases where automatic matching is impossible (due to ambiguity or lack of information) the user is addressed with a specific question. This mechanism helps naive users who have little knowledge about the programming environment.

Deterministic execution: Deterministic execution is enforced by combination of the following premises of PARDES: The **uniqueness** premise guarantees that there are no conflicting rules that update the same PDI 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 maintenance mechanism: The PARDES model uses the invariant definitions as consistency assertions. It finds all the inconsistencies generated by a change in the database relative to these assertions. The consistency is enforced to the level required by the materialization strategy and the exception handling definitions without a need of any additional programming.

Update redundancy: The **run time controller** of PARDES coupled with the **non reflectiveness** premise that eliminates loops avoid update redundancies and guarantees minimal

number of update operations.

4 Temporal Data Driven Rules

In recent years the discipline of *temporal databases* [33] has been growing in scope. The objective of that research is to add support for the time dimension

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

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

Retroactive Rule: On 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: On July 1993 it is decided on a new heuristic allocation routine that will be valid starting January 1994.

Figure 3: Temporal Operations

in the DBMS. Among other things, temporal databases view a database as a history and maintain 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.

4.1 Proactive and Retroactive Processing

A Retroactive (Proactive) Update is an update operation that modifies past (future) values of data elements.
A Retroactive (Proactive) Rule is a rule whose action includes a retroactive (proactive) update.
Retroactive (Proactive) Rule Activation is the application of a rule to past (future) states.

The above definitions indicate that rules can effect data *retroactively* in two main ways: due to retroactive rules, or due to retroactive activation of rules. The latter case can occur for two reasons:

1. a rule is introduced in the system with a valid time that includes past time interval(s).

2. A retroactive update occurred, and the updated data element(s) trigger a rule which was valid at that past time. The same is true for *proactive* effects.

Retroactive processing has been mentioned in some works, e.g., [29], but no details of the execution model nor support for rules were given.

Figure 3 demonstrates examples of the temporal dimension effect:

4.2 Extensions 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. To support temporal information, each *variable state* becomes a collection of *state elements*. A *state element* is a pair:
$<$ *value, temporal extension* $>$.
A *temporal extension* is a triple $< t_x, t_d, t_v >$,where:

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

Decision Time (t_d) - The decision occurrence time in the real world.

Valid Time (t_v) - The 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 *temporal element* [18] which is a time-point or an interval [t_s, t_e] or a collection of intervals and time-points. If t_v is an interval, it is believed that the object value is constant in this interval. Usually the starting point of the valid time interval (t_s) is well defined, however, the ending point of the interval (t_e) may be ∞ (unless it will be changed in the future) or frozen (the value cannot be changed). The valid time is not restricted to the data level. Meta-data entities also have valid time which limits their effect on the database. t_v values may be associated with: objects, designating the object life-span and variables, designating the life-span of the existence of this variable within the object. t_v may also be applied to rules, generalizations, existence of properties in classes and classification of objects to classes.

4.3 Types of Temporal Rules

The effect of the temporal dimension on rules is manifested 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-elements, applicable in different time-points. For each update, a decision should be made, to select the rule or rules are applicable for the validity interval of this update.

Temporal Conditions: As discussed in Section 3, 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.

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. Direct update of the t_v may create conflict resolution problems, such as a state element with a t_v which exceeds the life span of its variable; two state elements created by the same transaction with overlapping t_v's, etc.

The principles of the temporal extension of PARDES are discussed in [14]. The implementation aspects are now in an early prototype phase.

5 Conclusion

The major contribution of this study is the construction of a high level language and a supporting mechanism to cope with data driven rules and temporal rules. Data driven rules have an important role in many applications that have to maintain complex relationships between data elements or inter dependencies between various parts of the database; yet, data driven rules are handled in contemporary models as part of the general rule language. The PARDES model improves the handling of data driven rules by using their inherent semantic properties, and it meets the requirements that were discussed in the literature for high level languages. The data driven rules are extended to support temporal processing of retroactive and proactive updates and rules, adding another novel feature to the PARDES model.

References:

[1]: Albano A., Luccam C., Rezno O. Galileo: A Strongly Typed Interactive Conceptual Language. *ACM TODS 10(2)*, 1985.

[2]: ARCADIA - Issues Encountered in Building a Flexible Software Environment; Lessons from the Arcadia Project - Proc. *ACM Sigsoft 1992*, pp 169-180

[3]: Atkinson M.P., Chisholm K.J., Cockshott W.P. PS-algol: An Algol with a persistent heap, *SIGPLAN Not. (ACM), 17(7)*, pp. 24-31, July 1981.

[4]: Atkinson M.P., Bailey P., Cockshott W.P., Chisholm K.J., Morrison R. Progress with Persistent Programming, *Cambridge University Press, Cambridge, England,* 1984.

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

[6]: Balzer R., Goodman N., Wile D. Informality in Program Specifications, *IEEE tran. on soft. Eng., 4(2)*, pp. 94-102, March 1978.

[7]: Botzer D. Optimization of Knowledge and Data Representation in Active Databases, *M.sc., Thesis, Technion- Israel Institute of Technology*, Sep 1992.

[8]: Braine M. On the Relation Between Natural Logic of Reasoning and Standard Logic, *Psychological Review*, 1978.

[9]: Buneman O.P., Clemons E. Efficiently Monitoring Relational Data Base, *ACM TODS 4(3)*, pp. 368-382, Sep 1979.
[10]: Cardelli L. Amber, *Technical Report AT&T Bell Labs, Murray Hill, N.J.*, 1984.
[11]: Chakravarthy S. et al. HiPAC: A research Project in Active, Time-Constrained Database Management, *Final Technical Report, XAIT-89-02*, July 1989.
[12]: Date C.J. An Introduction to Database Systems, *Addison Wesley*, 1986.
[13]: Etzion O. Active Handling of Incomplete or Exceptional Information Database Systems, *Proc WITS 91*, pp. 46-60.
[14]: Etzion O., Gal A., Segev A. Temporal Support in Active Databases, *Proc. WITS*, pp. 245-254, Dec 1992.
[15]: Etzion O. Active Interdatabase Dependencies, to appear in *Information Sciences*, 1993.
[16]: Etzion O. PARDES- A Data-Driven Oriented Active Database Model, *Sigmod Record*, March 1993.
[17]: Etzion O. Flexible Consistency Modes For Active Database Applications, to appear in *Information Systems*, 1993.
[18] : Gadia S.K., Yeung C.S. A Generalized Model for a Relational Temporal Databases, *Proc. of ACM SIGMOD 88*, pp. 251-259, June 1988.
[19]: Gehani N.H., Jagadish H.V. Ode as an Active Database: Constraints and Triggers, *Proc. VLDB 91*, pp. 226-327, 1991.
[20]: Hammer M., McLeod D. Data Base Description with SDM: a semantic Data Base Model, *ACM TODS, 6(3)*, 1981.
[21]: Horowitz E., Kamper A. Application Generators, *IEEE software 2(1)*, 1985.
[22]: Hudson S., King R. CACTIS: A Database System for Specification Functionally Defined Data, *proc. IEEE OOBDS Workshop*, 1986.
[23]: Kowalski R. Logic for Problem Solving. *North-Holland*, 1979.
[24]: Lohman G.M., Lindsay B., Pirahesh H., Schiefer K.B. Extensions to Starburst: Objects, Types, Functions and Rules, *cacm 34(10)*, pp. 94-109, Oct 1991.
[25]: McCarthy D., Dayal U. The Architecture of an Active Data Base Management System, *Proc. 1989 ACM SIGMOD International Conference*, pp. 215-224, June 1989.
[26]: Morgenstern M. Constraint Equations: Declarative Expression of Constraints with Automatic Enforcement, *Proc. VLDB*, pp. 291-300, 1984.
[27]: Mylopoulos J., Berenstein P., Wong H.K.T. A Language Facility for Designing Database Intensive Applications, *ACM TODS 5(2)*, Jun 1980.
[28]: Peterson J.L. Petri Net Theory and The Modeling of Systems, *Prentice-Hall, Englewood Cliffs, N.J. 07632*, 1981.
[29]: Sarda N.L. HSQL: Historical Query Language. *Chapter 5 in [33], pp 110-140.*

[30]: Segev A., Fang, W., Optimal Update Policies for Distributed Materialized Views. *Management Science*, Vol. 37, No. 7, July 1991.
[31]: Sheth A., Rusinkiewicz M., Karabaits G. Using Polytransactions to Manage Interdependent Data, Chapter 14 in: A. Elmagarmid (ed)- Transactions Models for Advanced Database Applications, *Morgan-Kaufmann*, 1992.
[32]: Stonebraker M., Hanson E., Potamianos S. The POSTGRES rules manager, *IEEE Tran. on soft. Eng*, pp. 897-907, July 1988.
[33]: Tansel A.U., Clifford J., Gadia S., Jajodia S., Segev A., Snodgrass R. Temporal Databases: Theory Design and Implementation. *The Benjamin/Cummings Publishing Co. Inc, Redwood City, Ca*, 1993.
[34]: Ullman J. Implementation of Logical Query Language for Data Bases, in: *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1985
[35]: Wallace M. Kb2: A Knowledge Based System Embedded in Prolog, *Technical Report TR-KB-12, European Computer Industry Research Centre, Munich*, Aug. 1986.