Data-driven Active Rules for the Maintenance of Derived Data and Integrity Constraints in User Interfaces to Databases

João Pascoal Faria (jpf@fe.up.pt)

Raul Moreira Vidal (rmvidal@fe.up.pt)

FEUP - Faculdade de Engenharia da Universidade do Porto
Rua dos Bragas – 4099 Porto Codex – Portugal

INESC - Instituto de Engenharia de Sistemas e Computadores
Praça da República, 93 R/C – Apartado 4433 - 4007 Porto Codex – Portugal

Abstract

It is presented an approach based on data-driven active rules for the maintenance of derived data (calculated data) and integrity constraints (data restrictions) in screen forms and other user interfaces to databases. The approach aims to improve the capabilities of current application development tools, by combining the power of active rules, as proposed in the context of active databases, with the simplicity and data-driven nature of spreadsheets. The proposed data-driven active rules define derived data items as functions of other data items, by explicitly reading and writing their values, conditionally or unconditionally. They are defined essentially as unordered action-only rules with implicit triggering events, and are executed essentially as ordered (prioritized) event-action (EA) rules, possibly in combination with other event-action (EA) or event-condition-action (ECA) rules originated from different sources. The specialized nature of the actions performed by these rules allows the automatic determination of the triggering events and priorities among rules, as well as the establishment of improved conditions on rule sets that guarantee termination and confluence (determinism) of rule execution. The maintenance of integrity constraints is essentially reduced to the maintenance of derived data. The approach presented is not tied to a particular rule definition language or data model.

1. Introduction

The spreadsheet paradigm is very attractive for the maintenance of derived data in user interfaces to databases, particularly screen forms and reports, due to its simplicity and data-driven nature. But spreadsheet engines and languages usually lack flexibility to support all the sorts of derived data (calculated data) and integrity constraints (data restrictions) found in complex screen forms and reports. They usually provide limited or no support for:

i) conditionally calculated data items, i.e. data items that are calculated in some circumstances and input by the user in other circumstances;
ii) calculation of multiple data items through the same formula, to avoid repeating expensive computations, like the ones that involve the access to an external database;
iii) calculation of the same data item by multiple non-contradictory formulas that naturally arise as a consequence of the support of the two previous kinds of calculations;
iv) formulas for input conversion (e.g. uppercase conversion);
v) multi-way constraints, i.e. constraints with multiple choices of derived (calculated) and primitive (input) data items (e.g. impose the constraint $c=b-a$ by allowing the user to input any two values and have the third one calculated);
vi) general integrity constraints on both primitive and derived data;

vii) procedural definition of complex derivations.

Some constraint-based graphical user interfaces use a high-level paradigm - constraint satisfaction by the perturbation model - that could be used to support nicely some of these features, specially multi-way constraints [1]. But the kinds of relationships between data that are supported are limited.

Current form managers and application development tools usually provide low-level event-driven mechanisms, such as triggers or event-action (EA) rules, that may be used to implement the features above mentioned. Triggers provide high flexibility, but place on the programmer the burden of determining the events upon which the derived data items should be recalculated or the integrity constraints be checked.

A similar situation occurs in most active database systems [2] [3] [4], where derived data and integrity constraints are maintained by means of active rules or triggers of the form event-condition-action (ECA) or event-action (EA). To overcome the disadvantages of a pure event-driven approach without compromising efficiency, several approaches have been proposed, namely the following extreme approaches:

- The definition of constraints and derivations in a high-level declarative language, and its automatic compilation into active rules (STARBUST system [5][6]). This approach has been developed mainly for the incremental maintenance of materialized views and integrity constraints. Maintenance of integrity constraints is essentially reduced to the maintenance of materialized views.
- The direct definition of derivations as executable data-driven rules (PARDES project [7][8][9]). Data-driven rules are there defined as "rules that define derived data-elements as functions of other data-elements and thus are triggered by modifications to certain data-elements". Data-driven rules are essentially unordered action-only rules, with implicit triggering events of a restricted type - modification of data items that participate in derivations, and explicit actions of a restricted type - the assignment of values to derived data items. This enables the automatic derivation of the triggering events and allows the definition of more accurate processing and analysis techniques. As in the first approach, maintenance of integrity constraints is similar to the maintenance of derived data. This approach is best suited for the maintenance of derived attributes and restrictions on the values of the attributes, and can accommodate complex derivations that are specified in a procedural language.

We think that data-driven rules, as proposed in the PARDES system, are also well suited for the maintenance of derived data and integrity constraints in user interfaces to databases. However, the concept of data-driven rules is interpreted in a limited way in the PARDES project. For example, features iii) (when different rules are involved) and iv) mentioned above are contrary to the restrictions of uniqueness and non-reflexiveness of the PARDES model. Our approach starts by removing such restrictions at the rule definition level.

2. Data model and rule definition language

2.1. Data model

For the purpose of rule processing, the data items controlled by rules are represented abstractly by *state variables*. The set of state variables is denoted by V. It's assumed that V is finite. The set of possible values of the state variables is called the *state space* and is denoted by S.

State variables may represent screen fields, dynamically varying properties of those fields (must-fill, locked, etc.), screen records, screen tables, etc., but its internal structure is ignored in the generic approach presented here.

2.2. Rule definition language

Details of a practical rule definition language are not important in the generic approach presented here, only its expressive power is important. As explained in the introduction, data-driven active rules are defined essentially as unordered action-only rules. So, only the definition of the action part is considered here.

It's assumed that rules are defined in one language with primitives to query (read) and update (write), conditionally or unconditionally, the values of the state variables. Rules that update the value of state variables are called *derivation rules*.

Rules may also conditionally call a primitive to abort rule processing and subsequently restore the system to the last quiescent state. Such rules are called *abort rules*. Without any loss of generality, abort rules are hereafter handled as derivation rules that conditionally activate a special error variable, denoted *e*.

It's assumed that rules are individually *deterministic*, in the sense that, when a rule *r* is executed in state *s*, the state *s'* reached at the end of its execution is a function only of *s*, and does not depend on past states or time. Apart from that, no additional restrictions are placed about rule definition.

It’s assumed that only the *net effect* of rules is important. That is, when a rule *r* is executed in state *s*, only the departure state *s* and the state *s'* reached at the end of *r*'s execution are important. Intermediate states are not important. Side effects (e.g. messages to the user) are allowed, but it's assumed that they play a secondary role. Rules that produce the same net effect are considered semantically equivalent. Consequently, a rule *r* may be defined abstractly as a *function* from *S* to *S*, with the same name of the rule. The state reached at the end of the execution of *r* from departure state *s* may be denoted by *r(s)*.

It's assumed that rules have *fixpoint semantics*, in the sense that a rule *r* need be executed in state *s* only if *s* is not a fixpoint for *r*. A *fixpoint* for a rule *r* is a state *s* where *r(s)=s*. The condition *r(s)=s* that gives the fixpoints for *r* is called the *underlying static constraint* of *r*, or the static constraint *enforced* by *r*.

In order to present some examples, it is used the following *abstract representation*: a rule *r* is represented by a conjunction for different *yy* of unconditional or conditional formulas of the type *y'=f(X)* or *p(Z)* Þ *y'=f(X)*, respectively. In these formulas, *y* is a state variable, *p(Z)* is an arbitrary condition on a set *Z* of state variables, and *f(X)* is an arbitrary expression on a set *X* of state variables. Primed variable references, such as *y'*, denote final values and unprimed variable references, such as *X* and *Z*, denote initial values. The symbols “Þ”and “=” have the usual logical meaning. Common conditions may be factored out. An example of a rule set is given in appendix A.

This abstract representation has the following advantages:

- To obtain the complete function mapping from *S* to *S*, it suffices to add the following *frame conditions*: for any state variable *v* without a primed reference, add *v'=v*; for any state variable *y* with a conditional primed reference *p(Z)* Þ *y'=f(X)*, add Ø*p(Z)* Þ *y'=y*. The symbol “Ø” denotes logical negation.
- To obtain the underlying static constraint imposed by a rule, it suffices to remove the primes.

- It is easily translated into a practical rule language, referencing the same state variables in corresponding modes. Primed and unprimed variable references correspond to update (write) and query (read) operations. Auxiliary variables (local to each rule) may be required to store the initial values of some variables.

A rule with a common condition factored out is called *conditional*; it may be seen as a condition-action (CA) rule. However, it is not possible to defer action execution with respect to condition evaluation, and there is no obligation to execute the action *until* the condition is false (it is sufficient to reach a fixpoint for the pair condition-action). This is different from what usually happens in active rule systems that support CA rules [10]. Hence, the proposed rules are *essentially* action-only rules.

3. Rule execution model

Data-driven active rules, defined essentially as unordered action-only rules, are best executed as ordered (prioritized) event-action (EA) rules, because of:

- the need for their integration with other EA or ECA rules originated from different sources, with similar or different purposes;
- efficiency reasons.

Data-driven active rules, defined with the purpose of maintaining derived data and integrity constraints, are most safely executed *sequentially* (without nesting or concurrency), in order to keep rules atomic with respect to each other, and, consequently, keep the net effect of each rule unaffected by the existence of other rules.

Sequential execution of prioritized EA or ECA rules (with immediate coupling mode between condition and action) may be described by the following basic algorithm:

Algorithm 1 *(sequential rule execution)*:

```
1. While there are triggered rules and the error variable $e$ has not been activated:
   1.1. Select a triggered rule $r$ such that there is no other triggered rule $r'$
        with precedence (priority) over $r$
   1.2. Detrigger rule $r$
   1.3. Execute rule $r$, i.e., evaluate the condition and, in case it is true, execute the
        action
2. If the error variable $e$ has been activated, restore the system to the last quiescent
   state
```

Sequential rule execution may be represented abstractly as function composition.

The point in time (at the end of a transaction, at the end of a user update, on the occurrence of an event, etc.) where rule execution is invoked is called a *rule processing point* (*RPP*). The precise localization of RPP's is not important here. It is only assumed that, during rule execution, the rule set R does not change and user updates (or other updates not produced by rules) are not allowed.

Triggering criteria (triggering events) and ordering criteria (rule priorities) are developed next based on a preliminary analysis of the data dependencies existing between variables and rules.

4. Data dependency analysis

The sets of variables queried and updated by each rule constitute important information for rule processing and analysis. The variables queried (read) by a rule r are called the *input* *variables* of r, denoted *i-vars(r)*. The variables updated (written) by a rule r are called the *output variables* of r, denoted *o-vars(r)*. They correspond to the variables with unprimed and primed references, respectively, in the abstract representation. In a practical language, they can be obtained by a simple syntactic analysis.

A global picture of the relationship between variables and rules is given by a *rule-variable data dependency graph* (*RVDDG* for short). Due to obvious analogies, it's used the notation of dataflow diagrams. An example of a RVDDG is given in appendix A.

Since we assume that only the net effect of rules is important, any input and output variables of a rule r that could be eliminated by rewriting the rule while preserving its net effect, should be excluded from consideration. It can be shown that the minimal sets of input and output variables that can be achieved by rewriting a rule r while preserving its net effect are unique [11]. If no input or output variables of a rule r could be eliminated by rewriting r while preserving its net effect, it is said that r is in *canonical form*. For example, all the rules in the appendix are in canonical form. Rules with the same net effect of rules r_2 and r_6 from the appendix, but not in canonical form, are:

$$r_{2a}\text{: } z \neq x + y \Rightarrow z'=x+y \qquad \text{and} \qquad r_{6a}\text{: } y' = \begin{cases} x, & \text{if } x \geq 0 \\ y, & \text{otherwise} \end{cases}.$$

In some cases, it is not convenient to write rules in canonical form, for efficiency reasons. Two approaches are possible to obtain minimal input and output variables in such cases:

- they are explicitly provided by the programmer;
- the system performs a dataflow analysis during rule compilation to detect read and write operations that could be avoided by rule rewriting (e.g. a read operation after a write operation can be avoided by using an auxiliary local variable).

A global picture of the possible interferences between rules, via the variables queried and updated by them, is given by a *rule-rule data dependency graph* (*RRDDG* for short). The vertices of this graph are the rules. A directed edge $r_i \rightarrow r_j$, with $i \neq j$, means that r_i updates some variable that is referenced (queried or updated) by r_j. A self-loop $r_i \rightarrow r_i$ means that r_i updates some variable also queried by r_i. See the example in appendix A.

5. Rule types

As it will be seen, the following types of rules are potential sources of non-termination or non-determinism of rule execution: conditional rules, recursive rules and conflicting rules. Conditional rules have already been described.

A rule with overlapping sets of input and output variables is called *self-recursive*, because it defines the new value of a variable using the old value of the same variable. Examples of useful self-recursive rules are:

- rules for input conversion, like rule r_5 in appendix A;
- rules used in iterative methods, like

  r_{10}: $y' = exp(-y)$

Sets of rules that participate in a cycle of the RVDDG with two or more rules (otherwise it's the case of a single self-recursive rule) are called *mutually recursive*, because they define the new value of a variable using the old value of the same variable. Examples of useful mutually recursive rules are:

- rules with mutually exclusive conditions, like

  r_{11}: $x=0 \Rightarrow y' = z$ and r_{12}: $x \neq 0 \Rightarrow z' = y$;

- rules that enforce the same constraint in multiple ways, like rules r_8 and r_9 in appendix A.

Pairs of rules with common output variables are called *mutually conflicting* (or simply conflicting), because of the potential for contradictory updates. Examples of admissible conflicting rules (ones that in fact do not perform contradictory updates) are:

- rules with mutually exclusive conditions, like
  r_{13}: *x=0 Þ y' = z* and r_{14}: *x¹0 Þ y' = w*;
- rules with identical actions, like
  r_{15}: *x < 0 Þ e' = true* and r_{16}: *y < 0 Þ e' = true* (abort rules);
- rules with cumulative actions, like
  r_{17}: *z' = z È x* and r_{18}: *z' = z È y*;
- rules that act on different partitions of a state variable, like
  r_{19}: *z' =z È {1}* and r_{20}: *z' = z-{2}* (assuming *z* is set-valued).

Recursive rules are more powerful, because mutually conflicting rules may be converted into mutually recursive rules [11].

6. Triggering criteria

In this section it is presented triggering criteria (triggering and detriggering events) for action-only data-driven rules, needed for their translation into event-action rules. The primary purpose of the triggering criteria is to guarantee termination of rule processing after and only after a fixpoint for all the rules has been reached. The secondary purpose is to avoid unnecessary rule executions as much as possible. It is first presented a basic triggering criterion that only requires the knowledge of the input and output variables of each rule, and several improvements are subsequently discussed.

6.1. Basic triggering criterion

Conservative triggering criterion for data-driven rules may be defined based solely on the static knowledge of their input and output variables and on the dynamic monitoring of state variable modification events. A modification event occurs when a variable is updated with a different value from the one it currently holds.

In order to cope with the assumption that only the effect of rules is important, the following *net effect requirement* has to be obeyed: either each rule is forbidden to perform mutually compensating modifications or they are detected and ignored by the event monitoring mechanism.

The basic triggering criterion is:

i) all the rules are initially triggered (at the beginning of their existence);
ii) rules are detriggered at the beginning of their execution (see algorithm 1);
iii) a rule *r* being executed is triggered when an input variable of the rule is modified (by the rule itself);
iv) a rule *r* not being executed is triggered when an input or output variable of the rule is modified (by other rules or by the user).

The variables mentioned in iii) and iv) are called the *self-triggering variables* and *external-triggering variables* of *r*, denoted s*t-vars(r)* and *et-vars(r)*, respectively.

We next give the rational behind point iii). The decision of what values are assigned to what variables when a rule *r* is executed depends only on the initial values of the input variables of *r*. If the input variables of *r* are not modified during its execution, a second execution of r would assign exactly the same values to the same variables. Hence, there is no need to execute r again, reason why it may remain detriggered.

Now we give the rational behind point iv). Assume that a rule r was executed and became in a detriggered state. As long as neither the input nor the output variables of r are modified subsequently, a new execution of r would assign the same values to the same unmodified output variables, causing no change of the state of the system. Hence, there is no need to execute r again, reason why it may remain detriggered.

6.2. Static improvements based on the elimination of redundant triggering variables

The sets of triggering variables presented above may be too conservative.

Possible ways by which less conservative sets of external-triggering variables may be obtained are:

- Take for *et-vars(r)* the variables referenced in the underlying static constraint of r. This is the optimal static choice that can be made based only on the knowledge of r [11].
- In case it is known in advance that the updates performed by others (the user or other rules) on the output variables of r do not "contradict" the ones performed by r, those variables may be excluded from *et-vars(r)*.

Possible ways by which less conservative sets of self-triggering variables may be obtained are:

- In case it is known in advance that r is *idempotent*, take *st-vars(r)={}*. A rule r is called *idempotent* when $r^2(s)=r(s)$ for any state s. Rules with disjoint sets of input and output variables are trivially idempotent. A rule such that the execution of the rule's action always falsifies the rule's condition is also idempotent. In general, a semantic analysis is required to check if a rule is idempotent. For example, rule r_5 in appendix A is idempotent (the other rules are trivially idempotent).
- In case r is conditional, restrict *st-vars(r)* to the variables that are simultaneously input and output variables of the action part of r only (i.e., ignore the condition part).
- In a practical rule language, restrict *st-vars(r)* to the variables that may be updated after being read in the action part of r.

Example: Assume the following rule

r_{21}: $y>0 \Rightarrow [(x' = x+1) \wedge (y' = 0)]$

By the basic triggering criterion, one would have $et\text{-}vars(r_{21})=st\text{-}vars(r_{21})=\{x, y\}$.

The underlying static constraint is

$y>0 \Rightarrow [(x = x+1) \wedge (y = 0)]$

which can be simplified to

$y \leq 0$.

Hence, one takes $et\text{-}vars(r_{21})=\{y\}$. Since the rule is idempotent one takes $st\text{-}vars(r_{21})=\{\}$.

6.3. Dynamic improvements based on the value of the condition

Clearly, the external triggering variables of a conditional rule may be dynamically restricted in the following way:

- if a rule r is executed and its condition does not hold, subsequently take for *et-vars(r)* the variables referenced in the condition part;
- if a rule r is executed and its condition holds, subsequently take for *et-vars(r)* the variables referenced in the action part.

Initially, the value of the condition is unknown, but that's not a problem because all the rules are initially triggered.

Example : Assume the following rule from appendix A:

r_3: y=1 Þ w' = 1

If r_3 is executed and the condition holds, one subsequently takes *et-vars(r_3)={w}*. Otherwise, one takes *et-vars(r_3)={y}*.

6.4. Dynamic improvements based on the monitoring of read and write events

In practice, an improved triggering criterion may be defined by monitoring read and write events, besides the modification events. Prior static knowledge of the input and output variables of each rule is not even required!

Dynamic input and output sets are associated with each rule and maintained as follows:

- initially, the input and output sets are empty;
- the input and output sets are emptied at the beginning of a rule execution;
- a variable x is added to the input set when it is read during the rule execution, but only if it is not already in the output set (i.e., ignore a *read* after a *write*);
- a variable y is added to the output set when it is written during the rule execution (this is done before the eventual modification event is handled);
- in the case of conditional rules, if the condition holds, the input set is emptied before the action is executed.

The basic triggering criterion still applies, but with respect to the dynamic input and output sets instead of the static input and output sets.

7. Termination analysis

Rules may trigger each other (in *cascade*) indefinitely, causing non-termination of rule processing. We next give sufficient conditions on rule sets that guarantee termination of rule processing, for any initial state and any order of rule execution (i.e., any order by which rules are selected for execution when there are multiple triggered rules). Such conditions are useful in termination analysis of particular rule sets and in the development of ordering criteria to prevent non-termination.

7.1. Conservative termination analysis based on the triggering graph

Conservative termination analysis of active rules is usually based on the construction of a *triggering graph* [12]. The triggering graph (*TG*) for a rule set R is defined as follows:

- The vertices of the TG are the rules in the rule set R.
- A directed edge $r_i \rightarrow r_j$ (possibly with $i=j$) in the TG means that r_i may generate an event that triggers r_j.

It is known that, if the TG for a rule set R is acyclic, then rule processing always terminates for any initial state and any order of rule execution [12].

The TG for a set of data-driven active rules, processed according to the triggering criteria presented in section 6, is trivially obtained from the triggering variables and the output variables of each rule, as follows:

- A rule r_i may trigger itself only if *st-vars(r_i) ¹ {}*.
- A rule r_i may trigger a distinct rule r_j only if *o-vars(r_i) Ç et-vars(r_j) ¹ {}*.

With the basic triggering criterion, the TG is identical to the RRDDG. If some triggering variables are eliminated due to further information, the TG may have fewer edges. Consequently, if the RRDDG is acyclic (i.e., if the rule set R contains no recursive or conflicting rules), rule processing always terminates for any initial state and any order of rule execution.

7.2. Improved termination analysis based on the activation graph

Let's define the *activation graph* (AG) for a rule set R in the following way:

- The vertices of the AG are the rules in R.
- A directed edge $r_i \rightarrow r_j$ (with $i ¹ j$) in the AG means that r_i may activate r_j, in the sense that the execution of r_i in a fixpoint for r_j may produce a state that's not a fixpoint for r_j.
- A self-loop $r_i \rightarrow r_i$ in the AG means that r_i is not self-disactivating, in the sense that the final state at the end of r_i' execution may be not a fixpoint for r_i. In other words, r_i is not idempotent.

See an example of an AG in appendix A.

It can be shown that, if the AG of a rule set R is acyclic and the net effect requirement is satisfied, then rule processing always terminates for any initial state and any order of rule execution [11]. The net effect requirement is important to ensure the termination of rule processing after a fixpoint for all the rules has been reached, with at most one extra execution of each rule.

Since the edges of the AG are a subset of the edges of the TG, this condition is less conservative than the one based on the TG. By contrast, the construction of the AG requires a semantic analysis that cannot be automated in general, while the construction of the TG only requires a syntactic analysis. So, both conditions are useful.

Improved termination analysis of active rules has also been proposed in [13], by combining the TG with an analysis of the effect of rules actions on the truth-value of other rules conditions. However, it can be shown that, for the specific case of data-driven active rules, the conditions obtained in [13] are more conservative than the ones we've obtained here [11]. They use an activation graph that coincides with our in case the condition of each rule is precisely the complement of its underlying static constraint.

8. Confluence analysis

The order by which rules are considered for execution when there are multiple triggered rules may affect the final state reached, at termination of rule processing. If, for a given rule set, the final state reached at termination of rule processing is independent of the order by which rules are executed, for any initial state, the rule set is called *confluent* [12]. We next give sufficient conditions that guarantee confluence. Such conditions are useful in confluence analysis of particular rule sets and for the development of ordering criteria.

8.1. Rule commutativity

Confluence analysis of active rules is usually based on the notion of rule commutativity. Two rules r_i and r_j are *commutative* (or r_i and r_j *commute*) if, for any state s, applying rule r_i and then rule r_j from state s produces the same state as applying rule r_j and then rule r_i (i.e., $r_i \circ r_j = r_j \circ r_i$, where "$\circ$" denotes function composition). Non-commutative rules are also called *order dependent* rules in this paper. For a given rule set R, order dependencies between rules are depicted by an *order dependency graph* (*ODG*) with vertex set R and undirected edges joining pairs of order dependent rules.

The following properties are useful in the construction of the ODG:

- The ODG has no self-loops, because every rule commutes with itself.
- If two rules are not adjacent in the RRDDG, then they commute [11].
- If two rules are adjacent in the AG, then they do not commute [11].

From the first two properties, a conservative ODG may be obtained from the RRDDG by replacing directed edges by undirected edges and removing parallel edges and self-loops. An example of an ODG is given in appendix A.

8.2. Confluence of unordered rules

A confluence requirement for generic partially ordered active rules has been presented in [12]. Once applied to unordered active rules, the resulting requirement is that the rule set is terminating and that every two rules in the rule set commute, which is too conservative. In fact, using classic results about confluence [14], it can be shown that, if the rule set is terminating and every two rules in the rule set are confluent, then it is confluent [11].

There is another important difference with respect to the requirement presented in [12]. The definition of rule commutativity in [12] is based on *execution states* that also include the triggered rules and their corresponding "transition tables" besides the database state (corresponding here to the state of the state variables). The specialized nature of data-driven active rules dispenses the inclusion of the triggered rules and the transition tables.

The following properties are useful to check if two rules r_i and r_j are confluent:

- If two rules are commutative and terminating, then they are obviously confluent.
- If two rules r_i and r_j have an acyclic AG, then they are confluent iff $r_i \circ r_j \circ r_i = r_j \circ r_i \circ r_j$ [11].
- In particular, if two rules r_i and r_j are not recursive and not conflicting, and the execution of any of the rules cannot *falsify* (change value from *true* to *false*) the condition upon which the value of a variable is derived by the other rule, then they are confluent [11]. For example, with respect to the rule set in the appendix, the execution of rule r_6 can falsify the condition ($y=1$) upon which the value of w is derived by rule r_3.

Consequently, if a rule set R has no recursive rules, no conflicting rules, and no rules whose execution may falsify the conditions upon which the values of variables are derived by other rules, then it is terminating and confluent.

For example, the rule set in the appendix has two pairs of non-confluent rules. In fact, it is not confluent.

9. Ordering criteria

The order by which rules are considered for execution when there are multiple triggered rules may have impact on the *semantics* (the termination of rule processing and, in case of termination, the final state reached) and the *efficiency* (the number of executions of each rule). We next present ordering criteria aimed to improve termination, efficiency, confluence and controllability of rule processing.

9.1. Basic ordering criterion

Intuitively, the principle "compute before use" should be used whenever possible. That is, the following ordering criterion should be used:

- A rule r_i takes precedence over a rule r_j if there is a path from r_i to r_j and no path from r_j to r_i in the RVDDG.

The precedence means that if both are triggered, r_j cannot be selected for execution. The absence of a path from r_j to r_i is required to avoid the generation of cyclic precedences.

This criterion has the following advantages:

- It requires only a syntactic analysis of rules.
- It usually provides the expected semantic. In particular, abort rules only check final values of their input values. In general, non-recursive rules are executed only after their input variables have reached final values [11].
- Rules that are neither recursive nor conflicting are executed at most once [11].
- In the absence of recursive rules, this criterion fixes the semantic of the rule set. That is, further ordering choices have no impact on the termination of rule processing and, in case of termination, the final state reached [11].

9.2. Ordering of recursive rules

Mutually recursive rules are not ordered by the basic criterion. Several heuristics that can be used in isolation or in combination to order such rules are:

- Either statically or dynamically eliminate a minimal set of cyclic edges from the RVDDG, and then apply the basic ordering criterion. This tends to minimize the number of rule executions.
- Use the order by which rules are defined. This gives the programmer higher control over the system.
- Give precedence to rules with input variables most recently updated by the user. This tends to preserve the most recent user updates, giving the user higher control over the system (see example below). It has the disadvantage of being dynamic.

Example: Assume that one wants to impose the static constraint $b=a+c$ in multiple ways by the following mutually recursive rules (from appendix A):

r_8: $b' = a+c$ $\qquad$ r_9: $c' = b-a$

Assume that the user updates c and then rules are processed. Rule r_8 is first selected for execution. The user update is preserved and b is recalculated. Rule r_9 is executed with no effect. A symmetric situation occurs if the user updates b. If the user updates a, the value of the variable (b or c) most recently updated by the user will be preserved.

9.3. Additional criteria

Since rules may not become totally ordered by the basic criterion, there is room for additional secondary criteria. For instance, abort rules may be given higher priority, to be executed as earlier as possible. The order by which rules are defined may also be used.

For example, two total orderings of the rules presented in appendix A that result from the criteria discussed are $r_1 r_4\ r_6 r_5 r_2 r_3 r_7 r_8 r_9$ and $r_1 r_4\ r_6 r_5 r_2 r_3 r_7 r_9 r_8$.

10. Dynamic detection of contradictory updates

Although it is unusual or may even be considered a programming error, conflicting rules may perform contradictory updates (assign different values to the same variable) that may cause non-termination of rule processing. When conflicting rules do not appear in combination with recursive rules and the "principle compute before use" is followed, once contradictory updates are performed by rules during a RPP, it can be concluded that rule processing will not terminate [11]. Hence, rule processing may be immediately aborted. In the presence of recursive rules, contradictory updates may also be performed in different executions of the same rule. Such contradictory updates may converge, so rule processing should not be aborted immediately. In practice, it may be established a bound on the number of times that each rule may be executed or the number of times that each variable may be modified.

By contrast, it is assumed that the user updates may be contradicted by rules. This is necessary for input conversion, to establish departure values for some iterative methods, and to support error repair rules (derivation rules used with the purpose of repairing input errors). Mandatory user updates that cannot be contradicted by rules are best treated as temporary rules.

11. Conclusions and further work

It was presented a data-driven rule model suited for the maintenance of integrity constraints and derived data in screen forms and other user interfaces to databases. The specialized nature of the actions performed by those rules allowed us to derive triggering events and priorities among rules for efficient rule execution, as well as improved properties on rule sets that ensure termination and confluence of rule execution. More conservative conditions require only a syntactic analysis of the rules, while less conservative conditions require also a semantic analysis. A rule system based on the approach described is being re-implemented as a core component of a software tool that automates the development of interactive database applications [15], benefiting from the experience acquired in the development of successful applications with thousands of rules in previous implementations [16].

Several assumptions related to rule definition and execution are too restrictive in many practical applications and might be relaxed.

We've assumed that rules do not reference past states. Under some conditions, rules that reference past states may be accommodated into the formalism presented, such as:

- Rules that reference the last quiescent state of the system, e.g. to check dynamic integrity constraints.
- Rules that reference the last state of the system consistent with the rule (the state at the end of its last execution, or the last quiescent state of the system, according to which is most recent) for optimization purposes (e.g. the incremental evaluation of summary fields).

We've assumed that rules do not reference time. Under some conditions, rules that reference time may be accommodated into the formalism presented, such as:

- Rules that reference the time given by the system clock, e.g. to animate user interfaces or to automatically collect data from the database (or other external data sources). The clock time may be represented by a state variable that is periodically updated by means of an alarm or a timer. Such an update is similar to a user update. The only restriction is that it should be prevented (buffered) during rule processing, as it is required with respect to the user updates.

- Rules that compare times such as the time of creation and last modification of state variables. For comparison purposes, the real time need not be used; a sequential timestamp generated by the system is sufficient. Such rules may be used to provide a more explicit support for multi-way constraints.

We've ignored the internal structure of the state variables and considered only simple modification events. But, in the case of state variables with a complex internal structure, specialized modification events should be considered for optimization purposes (such as insertion, modification and deletion of elements of a set-valued state variable).

We've assumed that, in each RPP, rules are executed until a new quiescent state is reached, before user updates are allowed again. Such an *eager* mode of evaluation is important in the presence of abort rules. But a *lazy* mode of evaluation (*on demand*) of some data items, particularly output-only data items, could be more effective.

We've assumed a net effect policy that imposes an implementation overhead. The implications of removing such net effect policy should be investigated.

We've assumed that a quiescent state of the system must be a fixpoint for *all* the rules, even in the presence of conflicting rules. Alternative approaches could be useful in practice (e.g. allow *default* and *exception* rules).

The special nature of abort rules should be taken in consideration to improve termination and confluence analysis.

We've assumed a sequential mode of execution, without nesting or concurrency. To deal with more complex user interfaces and systems, such as master-detail and hierarchical screen forms, at least a combination of sequential and nested rule execution (by means of nested RPPs) has to be considered. Such richer modes of executions are found in object-oriented databases [17]. Data-driven active rules may be integrated in such environment according to the following scenario (the details are left to [15]):

- A complex system is viewed as a collection of interacting composite objects.
- A composite object is viewed as a small-scale system, with its own *intra-object state variables* and *intra-object rules*. The intra-object state variables represent component-objects that do not interact directly with each other. Attributes of an object and references to other objects are viewed as components-objects. These non-interacting component-objects are integrated via the methods (operations) and rules attached to the composite object. Data-driven rules attached to the composite object (intra-object rules at the composite object level) may be used to maintain derivations and constraints between the component-objects. Such rules are executed in RPPs in the methods of the composite object (including methods for object creation and deletion), and are triggered by events generated inside the composite object.
- Derivations and constraints that involve objects that interact directly with each other may be managed either via intra-object data-driven rules attached to a composite object of a sufficient high level of granularity, or via partial intra-object data-driven rules attached to each of the objects involved (essentially as proposed in the Ode system [18]). In the latter case, accesses to other objects involved are viewed as *side effects* to external systems.

Important issues related to the distribution of reactive processing (by means of active rules) between the user interface layer (our main target) and the database layer, in a centralized, client-server or three-tier architecture, have not been addressed here. Some of them are analyzed in [15].

12. References

[1] M. Sannela, J. Maloney, B. Freeman-Benson and A. Borning. Multi-way versus one-way constraints in user interfaces: experience with the DeltaBlue algorithm. In *Software Practice and Experience*, Vol. 23(5), pp. 529-566, May 1993.

[2] U. Dayal. Active database management systems. In *Proc. of Third International Conference on Data and Knowledge Bases,* Jerusalem, Israel, June 1998.

[3] S. Ceri and J. Widom. Active Database Systems. Triggers and Rules For Advanced Database Processing. Morgan Kaufmann, 1996.

[4] C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass, V. Subrahmanian and R. Zicari. Advanced Database Systems. Morgan Kaufmann, 1997.

[5] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. of the 16th VLDB Conference*, pp. 566-577, Brisbane, Australia, August 1990.

[6] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of the 17th VLDB Conference*, pp. 577-589, Barcelona, Spain, September 1991.

[7] O. Eztion. PARDES - A Data-Driven Oriented Active Database Model. In *SIGMOD RECORD*, Vol. 22, No. 1, March 1993.

[8] O. Eztion, A. Gal and A. Segev. Data driven and temporal rules in PARDES. In *Proc. of the First International Workshop on Rules in Database Systems*, pp 92-108, Edinburgh, Scotland, August 1993.

[9] A. Gal and O. Eztion. Maintaining Data-driven Rules in Databases. IEEE Computer, 28(1):28-38, January 1995.

[10] E. Hanson. Rule condition testing and action execution in Ariel. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pp. 49-58, San Diego, California, June 1992.

[11] J. Faria. Data-driven rules for the maintenance of derived data and integrity constraints in user interfaces to databases. *INESC Technical Report RI 1/99*. Porto, Portugal, May 1999 (url: http://www.fe.up.pt/~jpf/research/tr3.ps.zip).

[12] A. Aiken, J. Widom and J. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 59-68, USA, June 1992.

[13] E. Baralis, S. Ceri and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In *Proc. Rules in Database Systems - Second International Workshop – RIDS'95*, pp. 165-181, Athens, Greece, September 1995.

[14] Gérard Huet. Confluent Reductions: Abstract Porperties and Applications to Term Rewriting Systems. In *Journal of the ACM*, Vol. 27, No. 4, October 1980, pp. 797-821.

[15] J. Faria. Concepção de um sistema de regras activas para a manutenção de restrições de integridade e dados derivados em aplicações interactivas de bases de dados. FEUP, Porto, Portugal, September 1999 (*Ph.D. thesis to be submitted in portuguese*).

[16] J. Faria and J. Ranito. SAGA - Uma ferramenta interactiva para o desenvolvimento expedito e fácil manutenção de aplicações de bases de dados. In *Proc. ENDIEL'91, ST2-Indústrias do Software e da Informação*, pp. 33-40, Lisboa, Portugal, June 1991 (*in portuguese*).

[17] N. Paton and O. Diaz. Active Database Systems, 1999(?). (survey to be published in ACM Computing Surveys, url: http://www.cs.man.ac.uk/~norm/papers/surveys.ps).

[18] N. Gehani and H. Jagadish. Ode as an Active Database: Constraints and Triggers. *In Proc. of the 17th International Conference on Very large Data Bases,* pp. 327-336, Barcelona, September, 1991.

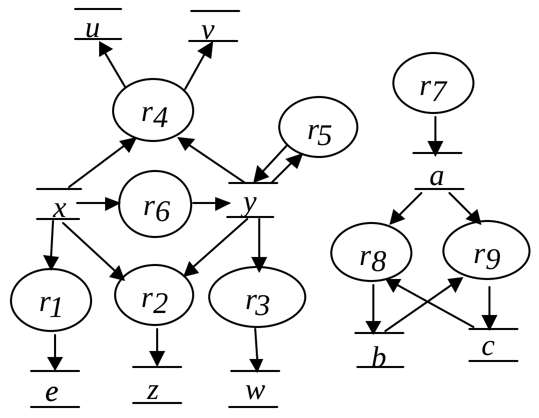
Appendix A – Example of a rule set and corresponding graphs

a) Rule set

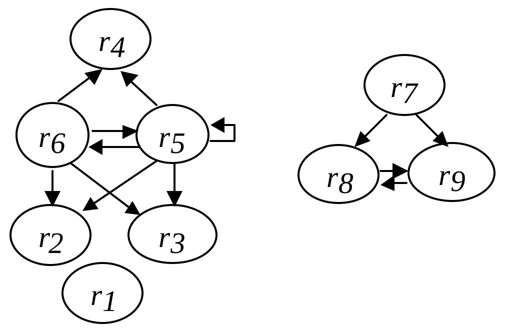
(intended to illustrate several types of rules and not a concrete application)

r_1:	*|x|>100 Þ e'=true*	*(abort rule handled as a derivation rule)*
r_2:	*z' = x+y*	*(unconditional derivation rule)*
r_3:	*y = 1 Þ w'=1*	*(conditional derivation rule)*
r_4:	*u'=max(x,y) Ù v'=min(x,y)*	
r_5:	*y' = |y|*	*(self-recursive)*
r_6:	*x ³0 Þ y'=x*	*(mutually conflicting with r_5)*
r_7:	*a' = 1*	
r_8:	*b' = a+c*	
r_9:	*c' = b-a*	*(mutually recursive with r_8)*

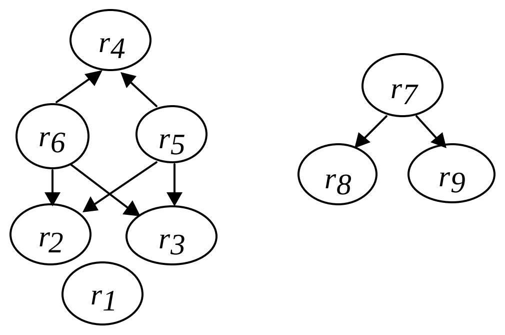
b) Rule-variable data dependency graph



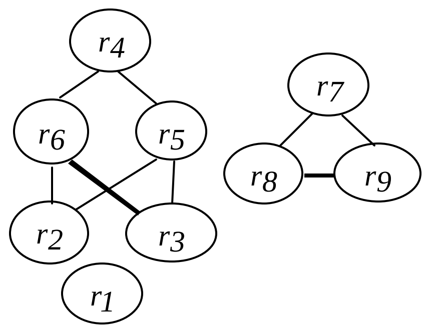
c) Rule-rule data dependency graph



d) Activation graph



e) Order dependency graph



(heavier edges link pairs of non-confluent rules)