Compiling complex database transition triggers

Donald Cohen
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
(213) 822-1511
DonC@vaxa.isi.edu

Abstract: This paper presents a language for specifying database updates, queries and rule triggers, and describes how triggers can be compiled into an efficient mechanism. The rule language allows specification of both state and transition constraints as special cases, but is more general than either. The implementation we describe compiles rules and updates independently of each other. Thus rules can be added or deleted without recompiling any update program and vice versa.

1. Introduction

People often wish to tell a database to watch out for certain conditions arising in the data, for instance, "whenever an account's balance drops below a threshhold, notify its owner(s)", or "do not allow a change that causes more than one employee to occupy the same office". Most databases provide little or no support for stating and reacting to such conditions, in large part because they can provide little or no implementation support. Rather a programmer has to figure out what updates could possibly cause a condition of interest to arise and for each such update supply code to recognize that condition.

This paper presents a temporal extension to the language of first order logic which enables convenient description of a wide variety of rule triggers by allowing reference to both the state before and the state after a transition. State and transition constraints are special cases of these more general conditions. Although the language is simple, we know of no previous attempts to express what it expresses. This includes Postgres [Stonebraker 87].

The bulk of the paper shows how such specifications can be efficiently implemented using a network similar to that of the RETE algorithm [Forgy 79]. Of course, triggering, like querying, is not free, and some triggers, like some queries, simply cannot be implemented cheaply. However, the cost of triggering with our method is generally competitive with the result of hand optimization. The implementation we describe has been in daily use for several years by about a dozen users with databases containing thousands of rules.

1.1. Related work

Previous work in this area mostly concentrates on integrity constraints, e.g., [Eswaran 75], [Hammer 75], [Stonebraker 75], [Hammer 78], [Blaustein 81], [Stonebraker 87], [Qian 86], etc. Our work is most closely related to that of Nicolas [Nicolas 82] and Forgy [Forgy 79]. Our implementation preserves the best features of both of these approaches.

Nicolas showed how to efficiently enforce "state constraints", which prevent invalid database states such as employees sharing offices. Constraints are represented as well-formed formulae (**wffs**) of first order logic. The method alters a given program that updates the database to take a given constraint into consideration. For instance, the program that assigned an office to a person would be altered to check that the office was not assigned to any other person. This method assumes that the constraint was satisfied before the update.

Forgy's RETE algorithm is commonly used for implementing expert systems. It efficiently maintains a set of "matches" for various "patterns" as a (small) database changes. These patterns may be described approximately as conjunctions with free variables, where a "match" is a tuple representing an assignment of values to the free variables that satisfies the conjunction.[1] The RETE algorithm reacts to a database transition. It detects any tuples that the transition causes to start or stop matching any pattern of interest. Patterns are compiled into a

This research was supported by the Defense Advanced Research Projects Agency under contract No. MDA903 81 C 0335. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any person or agency connected with them.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 0-89791-317-5/89/0005/0225 $1.50

[1] Actually, the RETE language is more general in some ways - it allows relations of variable arity, matching on relations, and segment variables. We will ignore these features.

directed acyclic graph called a "match network". A node reads data from incoming edges and writes (different) data on outgoing edges. The computations at the nodes gradually transform a set of database updates into a set of assignments that start or stop matching the patterns of interest.

Nicolas' algorithm is more general than Forgy's in that it allows arbitrary wffs, while Forgy's algorithm is more general in that it finds matches (as opposed to simply checking that there are none). Another advantage of Forgy's algorithm is that each pattern is compiled once, whereas Nicolas requires that every constraint be compiled for every update (so adding a new constraint requires recompiling many update programs). However, Forgy's algorithm is only appropriate for very small databases, e.g., it uses more than enough internal storage to store every fact in the database (sometimes much more!). Another problem with RETE is that it prevents "query optimization" - it compiles a given pattern into a fixed algorithm which may not be anywhere near the most efficient one. Nicolas, on the other hand simply compiles constraints into queries which can be optimized like any other queries.

Our method generalizes both of these in that it finds matches for arbitrary "transition conditions."[2] Like Forgy, we compile each trigger once. New triggers and update programs can both be added independently (at little cost) while the database is in use. Finally, our algorithm is, at worst, almost as efficient as either of the other two (for the cases they handle), and sometimes much more so.

1.2. Outline

Section 2 describes our query language and the extension with which rules are specified. Section 3 describes our implementation which compiles the rule triggers into efficient programs and runs these programs at appropriate times. Section 4 deals with efficiency issues. It discusses some limitations of our method, implementation alternatives, and the considerations that make them more or less desirable.

2. The language

Our method is implemented as part of AP5, a database oriented extension of commonlisp [Steele 84]. We will insulate the reader from lisp syntax. The query language of AP5 is an extension to first order logic. It includes relations of fixed arity, boolean connectives $\wedge$ (and), $\vee$ (or), $\sim$ (not), $\supset$ (implies), etc. and quantifiers $\forall$ (for-all) and $\exists$ (exists). AP5 adds to lisp the following database constructs which we will use freely:

- `insert` *relation* (o_1, ..., o_i)
  adds the tuple (o_1, ..., o_i) to the (i-ary) relation *relation*. It does nothing if the tuple is already present.
- `delete` *relation* (o_1, ..., o_i)
  deletes the tuple (o_1, ..., o_i) from the (i-ary) relation *relation*. It does nothing if the tuple is already absent.
- `if` *wff* ...
  is conditional execution, depending on the truth value of *wff*.
- `iterate over` var_1, ..., var_j `suchthat` *wff* ...
  where var_1, ..., var_j are the variables free in *wff*, iterates over tuples (o_1, ..., o_j) such that the result of substituting o_i for (free) occurrences of var_i ($1 \leq i \leq j$) in *wff* is true.
- `atomic` [$program_1$; ...; $program_k$]
  executes the programs in order, but delays all database updates (insert's and delete's) until the end. They are then combined into a single transition. If the updates cannot *all* be done for some reason (see below), then *none* of the updates are done.

In addition to these programming constructs, there are rules which are declared with syntax described below. Semantically there are several disjoint classes of rules. When a transition is proposed, "consistency rules" ensure consistency with any integrity constraints. These rules may cause the transition to be altered or rejected. This process is described in more detail below. For now we merely note that all accepted transitions satisfy all constraints. The rule preventing shared offices is a consistency rule. The rule triggering mechanism is also used to maintain "materialized views", but that will not be described in detail here.

When a transition is accepted, it invokes "automation rules", so called because they are used to automate such activities as sending low balance notifications. It is not inconsistent to have a low balance, or even to have a low balance and not yet have been notified. Automation rules may cause further database transitions. AP5 does not specify the order in which automation rules are executed. It only promises that every automation invoked by a transition will run before control returns to the program that requested the transition.

The atomic construct is especially relevant for rule triggering, because it defines the observable states of the database. Constraints apply only to these observable states. For instance, if every employee were required to have a unique office, and the employee Joe had Office11 then
`insert Office(Joe, Office12)` would not be allowed since that would cause him to have two offices. Likewise
`delete Office(Joe, Office11)` would not be allowed since that would cause him to have no offices. However,

```
atomic [insert Office(Joe, Office12);
        delete Office(Joe, Office11)]
```

would be allowed, unless it violated other constraints. Similarly, automation rules react to entire database transitions, not to "parts" of

[2] Transition conditions are formally defined later but are more general than either of the other two. The low balance notification is an example.

transitions. In the low balance rule, suppose each account had its own threshhold. Then, reducing the balance from 100 to 10 in the same atomic transition as reducing the threshhold from 20 to 5 would not cause a notification.

2.1. Language extension

We now introduce a simple language extension that allows us to specify triggers such as that of the low balance rule. The extension allows us to talk about two adjacent database states: the state before the transition under consideration and the (proposed) state after. In the context of a transition, queries may use two temporal operators. Either could be defined in terms of the other, but it is convenient to have both:

`Previously` *wff*
is true if *wff* was true before the update.
`Start` *wff*
is true if *wff* was false before the update and is true after.

In the context of a (proposed) transition, any wff not in the scope of a Start or Previously is evaluated in the new state. Since we only provide access to two states, we do not allow nested temporal references. This extension does *not* allow general temporal reference, for instance we cannot notify owners only the first time in any month that their accounts fall below threshold.[3]

We also introduce a syntactic construct called a **Description** of the form "*variables* `suchthat` *wff*", as illustrated above in the loop construct. *Variables* are the variables free in *wff*. A tuple **satisfies** a description if substitution of the values in the tuple for (free) occurrences of *variables* in *wff* is true. In the special case where *wff* has no free variables, the empty tuple satisfies "`nil suchthat` *wff*" if *wff* is true.
(Note: "nil" denotes the empty variable list, not a single variable named nil!) We refer to the set of such tuples as the **matches** of the description. We refer to the process of computing this set as **matching** the description.

2.1.1. Declaring automation rules

We are now in a position to introduce syntax for the rules described above. Automation rules are declared as follows:

```
AutomationRule notify-low-balance
  trigger: person, acct suchthat
       owns(person, acct) ∧
       Start [∃ bal, min
              [balance(acct, bal) ∧
               threshhold(acct, min) ∧
               balance<min]]
  Response: notify(person, acct)
```

On every database transition AP5 must find every tuple (person, acct) satisfying the trigger, in this case, every person and acct such that person owns acct and in this transition it became the case that acct's balance is less than its threshhold. For every such tuple AP5 must call the notify procedure on person and acct.

Notice that the position of the start, like the position of quantifiers, allows fine control over when the response will be executed. For instance, this trigger will not notify a person who becomes an owner of an account that already has a low balance. We could get that affect by moving the `owns` inside the start.

Automation triggers describe database *transitions* rather than states - they always refer to both the old and new states. The reason is that we want to detect matches by only reacting to updates of the relations in the trigger. This won't work for a trigger like `x suchthat P(x)` - it can be satisfied even when we make updates unrelated to P. Formally, our current implementation requires triggers to have a property we call **relevance**: they must only be satisfied during transitions that update some relation that they use. This condition is slightly stronger than simply referring to both the old and new states. For instance

`x suchthat [P(x) ∨ Start Q(x)]`

is not relevant, since any tuples in P will satisfy it even during transitions that don't update P or Q. It is possible to match such descriptions by storing the set of matches, but the need never seems to arise in languages that support the notion of atomic database transitions.

2.1.2. Declaring consistency rules

We can think of consistency rules in two parts: first, the rule must not be violated when it is declared.[4] Otherwise the data must be fixed first. Second, it must be enforced incrementally. We can think of this part as an automation that gets to run before the transition is accepted and is allowed to abort the transition in its response. We use the word ConsistencyRule instead of AutomationRule to indicate that the rule runs before the transition is accepted, and the word "abort" to mean the transition should be rejected. We incrementally enforce the constraint by aborting any transition in which it becomes violated:

```
ConsistencyRule no-office-sharing
  trigger: nil suchthat
       Start [∼∀ emp1, emp2, off
                 [[office(emp1, off) ∧
                   office(emp2, off)]
                  ⊃ emp1=emp2]]
  Response: abort
```

Although this is sufficient for implementing all consistency rules, we mention a few refinements.

First, while the straight forward translation of the constraint is to abort whenever it is violated, it is often more useful to respond with an attempt to repair the violation. The repair normally depends

[3] Of course, this can be "programmed" by introducing additional data. The point is that the trigger cannot be directly expressed in our language.

[4] In AP5 we allow users to declare that the rule need not be checked at declaration. Although usually "just an optimization", this is sometimes necessary - some constraints cannot be checked and yet *can* be incrementally enforced!

on the particular instantiation(s) at fault. AP5 provides an elaborate consistency repair mechanism which is beyond the scope of this paper. However, we offer an annotated example to convey its flavor.

First, suppose we had written the following rule instead of the one above:

```
ConsistencyRule no-office-sharing
  trigger: emp1, emp2, off suchthat
       Start [~[[office(emp1, off) ∧
                 office(emp2, off)]
                ⊃ emp1=emp2]]
  Response:
       if Previously office(emp1, off)
          delete office(emp1, off)
```

This would mean that on every proposed transition, AP5 should find all tuples (emp1, emp2, off) such that emp1 and emp2 are distinct employees (proposed to be) in the same office, off. It should then execute the response for each such tuple. If, for instance, Joe occupies office 14 and we try to assign Jim to office 14, we get two such tuples: (Jim, Joe, Office14) and (Joe, Jim, Office14). For the first of these the response has no effect, since Jim did not previously occupy Office14. The second results in removing Joe from Office14.

Of course, there is no guarantee that this results in a consistent transition. AP5 interprets consistency rules to mean that (1) no transition in which the trigger is satisfied is consistent, but also (2) that executing the repair on tuples that satisfy the trigger is a permissible way to augment attempted transitions that violate the constraint. AP5 finds all such tuples and executes the repairs. (It arranges for the repairs to run independently of one another - their order doesn't matter.) The repairs are themselves constrained, for instance they may not "undo" the original updates. If they do propose additional updates, these are added to the original set, and the resulting transition is proposed in place of the original. It is in turn checked for consistency. The process continues until either consistency is achieved or AP5 can tell that consistency cannot be achieved. In the example above, our attempt to put Jim in office 14 would, in the absence of any other rules, have the same effect as atomically putting Jim in office 14 and removing Joe.

As another refinement, we can exploit the fact that no constraints were violated in the previous state. When consistency triggers have the form
variables `suchthat Start` *wff*, we replace the Start with a new syntactic construct, Start*. This means the same thing as Start except that it tells the implementor (in this case a program) that the wff was false before the transition, which allows some optimization.[5] We are now in a position to characterize the set of triggers handled by Nicolas. They are, in our language, the descriptions of the form
`nil suchthat Start*` *wff*, where *wff* has no free variables or temporal operators.

Finally, since so many consistency triggers look like the one above (except that Start is replaced by Start*), we provide a more direct syntax which elides the "Start* ~" and allows the constraint to be stated directly:

```
AlwaysRequire no-office-sharing
  ∀ emp1, emp2, off
    [[office(emp1, off) ∧
      office(emp2, off)]
     ⊃ emp1=emp2]
  Response:
       if Previously office(emp1, off)
          delete office(emp1, off)
```

This is equivalent to the rule above - every instantiation of the leading universal variables satisfying Start* of the rest of the wff causes an invocation of the response.

3. Implementation

When presented with a rule declaration, AP5 tries to build a piece of match network for its trigger. In some cases this is impossible, e.g., the description is irrelevant or certain transitions would cause infinitely many tuples to satisfy the description. In other cases AP5 can determine that it is unnecessary, since the trigger is unsatisfiable. In other cases the desired piece of network will be built and added to the existing match network, after which the rule will be triggered by appropriate database transitions.

When database transitions are proposed, AP5 must find all tuples that satisfy all rule triggers. This process requires three inputs: the previous database, P, the set of proposed database updates, U, with no-ops removed, and a set of relevant descriptions, D, used as rule triggers. The output is, for each $d \in D$, the tuples satisfying d. Of course, by the time a transition is proposed, D has been compiled into a network of **matchnodes**, N. Each matchnode, n, has an associated description, d_n, and a program, p_n, and is "connected" to other nodes, its predecessors and successors. D is a subset of $\{d_n \mid n \in N\}$. (This set also contains some "intermediate results" which are not in D.)

Tuples satisfying d_n are computed at n and are used as the inputs to the successors of n. If n has predecessors, p_n is run on each tuple produced as output of any predecessor to compute a set of tuples that satisfy d_n. The set of all tuples satisfying d_n (the output of n) is simply the union of all sets computed by p_n. The only nodes without predecessors are those that correspond to database updates, e.g.,

```
x, y suchthat Start office(x, y).
```

Their outputs are computed directly from U.

[5] This "specification" of start* can be formalized as follows: [Previously ~P] ⊃ [Start P ≡ Start* P]. This permits many "implementations" of Start*. The one we show below appears to give the same results as Nicolas if his quantifiers are optimally ordered.

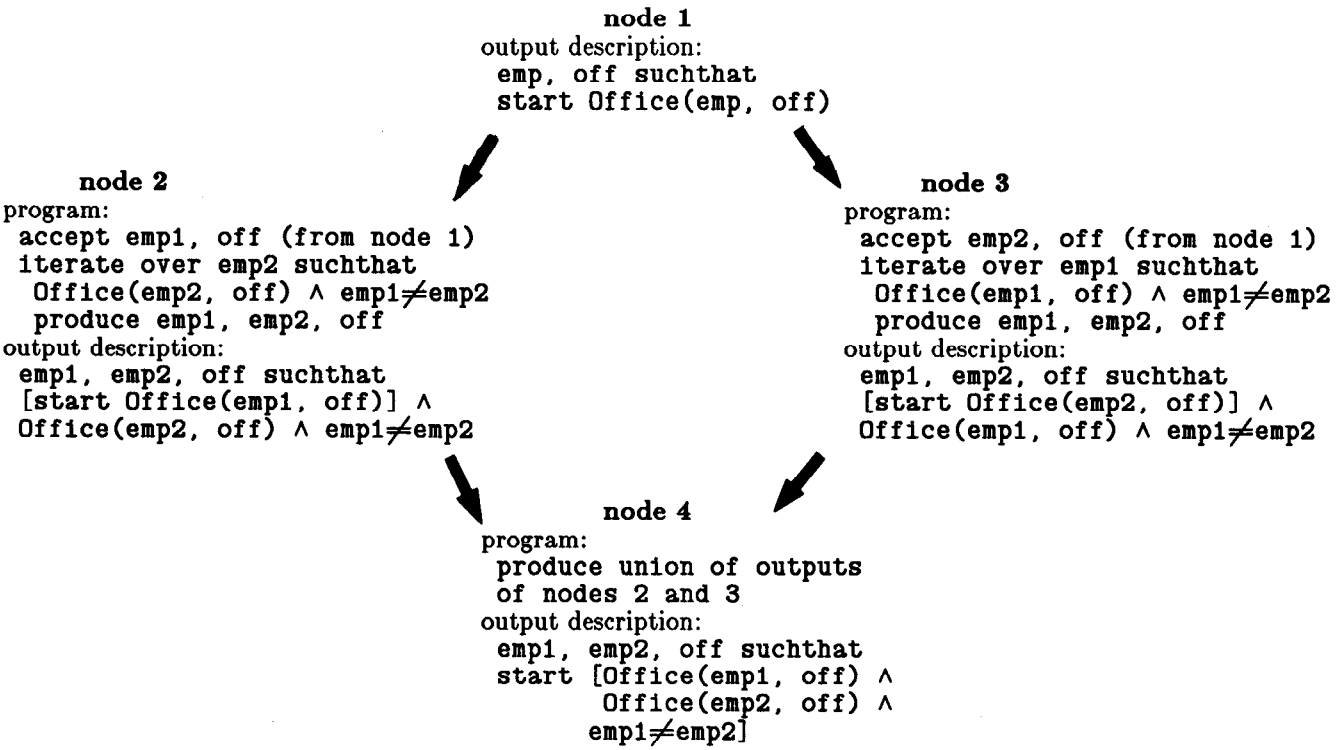

Figure 3-1: match network for office constraint

3.1. Example

To enforce the constraint:

```
∀ emp1, emp2, off
   [[office(emp1, off) ∧
     office(emp2, off)]
    ⊃ emp1=emp2]
```

(in the style of the last consistency rule, with the repair) AP5 builds a node for:

```
emp1, emp2, off suchthat
  Start* ~[[office(emp1, off) ∧
            office(emp2, off)]
           ⊃ emp1=emp2]
```

This simplifies to:

```
emp1, emp2, off suchthat
  Start* [office(emp1, off) ∧
          office(emp2, off) ∧
          emp1≠emp2]
```

The $\neq$ relation is known to be **static** (never updated), so we don't need to worry about it *becoming* true. This description compiles into four nodes as shown in the figure above.

Note that the programs access the "new" state of the database, so that atomically replacing one occupant with another will not cause a match. The implementation of this is discussed later.

3.2. Compiling the match network

The "matchnode compiler", a program for constructing nodes, takes as input a variable list and a simplified wff.[6] We will assume that all Previously's have been translated into Start's:

Previously P $\equiv$ (Start ~P ∨ (P ∧ ~Start P))

The result is either a node that computes tuples satisfying the description, an indication that the description is unsatisfiable, e.g.,
`x, y suchthat Start x=y`, or an indication that the description cannot be compiled, e.g, it's "irrelevant". Another reason a description cannot be compiled is that its program cannot be compiled (see [Cohen 86]).[7]

The matchnode compiler builds matchnodes that can be understood in four levels, described below. The matchnodes in any level have predecessors only in earlier levels.

3.2.1. Input nodes

Input nodes are the simplest. Their descriptions have two forms:

$x_1,\ldots,x_n$ `suchthat Start` $R(x_1,\ldots,x_n)$

$x_1,\ldots,x_n$ `suchthat Start` $\sim R(x_1,\ldots,x_n)$

where R is an n-ary relation. A node with the first description computes the set of tuples added to R, while a node with the second description computes the set of tuples deleted from R.

3.2.2. Starts of other literals

More general literals (non-compound wffs and their negations) differ from those above only in that their argument lists may contain constants and duplicated variables, and that the variables in the arguments and in the variable list may appear in different orders. Starts of such literals are handled by successors of

[6] The simplifier reduces all connectives to {and, or, not}, pushes negations all the way inward and does "obvious" simplifications, e.g., ~~P = P.

[7] In AP5, "optimizations" such as Start* and static relations sometimes influence whether a description can be compiled. For instance, they may "optimize out" something that is uncompilable.

input nodes, as illustrated by the nodes described below. These examples all take input from the node for

`x, y suchthat Start office(x, y)`

- *Constants:* The node for
  `y suchthat Start office(John, y)`
  has this program:
  `if x=John produce y`
- *Repeated variables:* The node for
  `x suchthat Start office(x, x)`
  has this program:
  `if x=y produce x`
- *Permutations:* The node for
  `y, x suchthat Start office(x, y)`
  has this program:
  `produce y, x`

As an example that combines all these cases, suppose we have an input node,

`x, y, z, w suchthat Start R(x, y, z, w)`

and we want a node for

`x, y suchthat Start R(2, y, x, x)`

For purposes of exposition we rename the variables to match the input node:

`z, y suchthat Start R(2, y, z, z)`

To obtain such a node we start with the constant:

`y, z, w suchthat Start R(2, y, z, w)`
`program: if x=2 produce y, z, w`

Next the repeated variable:

`y, z suchthat Start R(2, y, z, z)`
`program: if x=2 if z=w produce y, z`

Finally the permutation gives:

`z, y suchthat Start R(2, y, z, z)`
`program: if x=2 if z=w produce z, y`

3.2.3. Conjunctions containing a Start of a literal

An example is

```
emp1, emp2, off suchthat
  [[Start office(emp1, off)] ∧
   office(emp2, off) ∧
   emp1≠emp2]
```

These nodes have one predecessor which computes tuples satisfying the Start conjunct:

```
emp1, off suchthat
    Start office(emp1, off)
```

(This is really the same input node that we saw before. We rename the variables for purposes of exposition.) The program is constructed from these two descriptions. It accepts as inputs the variable list of the predecessor node's description, *inputs*, and does the following:

`iterate over` *outputs - inputs* `suchthat`
remainder-of-conjunction
`produce` *outputs*

where *outputs* is the variable list of this node's description, *outputs - inputs* is the set difference (order is immaterial) and *remainder-of-conjunction* is the conjunction to be matched with the Start conjunct removed. In this case the program is:

```
iterate over emp2 suchthat
  [office(emp2, off) ∧ emp1≠emp2]
  produce emp1,emp2,off
```

When *outputs - inputs* is empty, the loop simplifies to a test:

`if` *remainder-of-conjunction* `produce` *outputs*

If several conjuncts are Start's, a choice can be made on grounds of efficiency.

This level also handles outer existential quantifiers, e.g.,

```
emp1 suchthat
   ∃ emp2, off
    [[Start office(emp1, off)] ∧
     office(emp2, off) ∧
     emp1≠emp2]
```

The program is almost the same as if the existential variables were in *outputs* but the existential variables are no longer produced, and they remain quantified unless they're in *inputs*:

```
if ∃ emp2
    [office(emp2, off) ∧ emp1≠emp2]
  produce emp1
```

3.2.4. Other compound wffs

An example is

```
emp1, emp2, off suchthat
   Start* [office(emp1,off) ∧
           office(emp2, off) ∧
           emp1≠emp2]
```

Such descriptions, "*vars* `suchthat` *wff*", are translated into equivalent descriptions in disjunctive normal form (**DNF**), "*vars* `suchthat` $W_1 \vee \ldots \vee W_n$" where each W_i takes the form described in 3.2.3. Intuitively, W_i corresponds to one occurrence of a non-static literal, L, in *wff*, and has the meaning that *wff* becomes true, at least in part, due to L becoming true. W_i contains `Start` L as one conjunct. Informally, we call "the rest of" W_i the **residue** of *wff* with respect to L.

The matchnode for "*vars* `suchthat` *wff*" simply computes the union of the matches of its predecessors, which match the descriptions "*vars* `suchthat` W_i". Unless all disjuncts use all of *vars* there will be infinitely many matches, which will prevent compilation.

The desired form is achieved by applying distributive laws such as:

`∃x[P ∨ Q] = ∃xP ∨ ∃xQ`

In general this should be done from the outside in - wffs of the wrong form may appear as subwffs of acceptable wffs, in which case they need not be rewritten. For instance

```
y suchthat
  [Start R(y)] ∧ ∃x[P(x,y) ∨ Q(x,y)
```

matches the pattern of 3.2.3.

When a conjunction must be rewritten it must be distributed over *some* conjunct, C, with the property that *variables* `suchthat` C is relevant, where *variables* is a list of variables free in C.

`P ∧ (Q ∨ R) = (P ∧ Q) ∨ (P ∧ R)`
`P ∧ ∃xQ = ∃x(P ∧ Q)`

Usually there's no choice - only one conjunct contains Start. If no such conjunct leads to a compilable program, the description cannot be matched.

Start's are pushed inward (only as needed) by using the following identities:

```
[Start [P ∧ Q]] =
   [[Start P] ∧ Q] ∨  [[Start Q] ∧ P]
[Start [P ∨ Q]] =
   [[Start P] ∧ ~[Previously Q]] ∨
   [[Start Q] ∧ ~[Previously P]]
[Start ∀xP] = ∃x[Start P] ∧ ∀xP
[Start ∃xP] =
   ∃x[Start P] ∧ ~[Previously ∃xP]

[Start* [P ∧ Q]] =
   [[Start* P] ∧ Q] ∨ [[Start* Q] ∧ P]
[Start* [P ∨ Q]] =
   [Start* P] ∨ [Start* Q]
[Start* ∀xP] = ∃x[Start* P] ∧ ∀xP
[Start* ∃xP] = ∃x[Start* P]
```
[8]

Note that, for literals, Start* is the same as Start.

Notice that outer ∀'s cannot be matched. Such descriptions look like

```
x suchthat ∀y Start office(x, y)
```

AP5 assumes a world with infinitely many objects. Therefore this describes a transition with infinitely many updates.[9]

Note the result contains at most one disjunct per literal occurrence in the original wff.

3.3. Detailed example

We now illustrate by carrying out the procedure for the automation trigger:

```
person, acct suchthat
  owns(person, acct) ∧
  Start ∃ bal, min
         [balance(acct, bal) ∧
          threshhold(acct, min) ∧
          bal<min]
```

Since this is a conjunction but not in the desired form, it will have to be distributed over a relevant conjunct. Only one conjunct contains a start, but that start is on the outside, so it must be pushed inward:

```
owns(person, acct) ∧
∃ bal, min
   [Start [balance(acct, bal) ∧
           threshhold(acct, min) ∧
           bal<min]]
 ∧
~Previously ∃ bal, min
              [balance(acct, bal) ∧
               threshhold(acct, min) ∧
               bal<min]
```

Now we have three conjuncts, only one of which (an existential) contains start. We can push the conjunction inside the existential:

```
∃ bal, min
  [Start [balance(acct, bal) ∧
          threshhold(acct, min) ∧
          bal<min]]
  ∧ owns(person, acct)
  ∧
  ~Previously ∃ bal, min
                [balance(acct, bal) ∧
                 threshhold(acct, min) ∧
                 bal<min]
```

This still has the wrong form because the conjunction in the existential does not contain start of a literal. The conjunction must again be pushed inside a relevant conjunct. Again, only one conjunct contains start, and since that start is on the outside, it must be pushed in:

```
∃ bal, min
  [[Start balance(acct, bal) ∧
    threshhold(acct, min) ∧
    bal<min]
   ∨
   [balance(acct, bal) ∧
    Start threshhold(acct, min) ∧
    bal<min]]
  ∧ owns(person, acct)
  ∧
  ~Previously ∃ bal, min
                [balance(acct, bal) ∧
                 threshhold(acct, min) ∧
                 bal<min]
```

Note that < never changes so its Start conjunct simplifies to false.

We can now move the conjunction and quantifier inside the disjunction to achieve the desired form:

```
∃ bal, min
   [owns(person, acct) ∧
    Start balance(acct, bal) ∧
    threshhold(acct, min) ∧
    bal<min ∧
    ~Previously
        ∃ bal, min
          [balance(acct, bal) ∧
           threshhold(acct, min) ∧
           bal<min]]
 ∨
∃ bal, min
   [owns(person, acct) ∧
    balance(acct, bal) ∧
    Start threshhold(acct, min) ∧
    bal<min ∧
    ~Previously
        ∃ bal, min
          [balance(acct, bal) ∧
           threshhold(acct, min) ∧
           bal<min]]
```

[8] This definition of Start* has the properties that [Start P] ⊃ [Start* P] and that [Start* P] ⊃ P. It thus meets the requirements of Start*, since [Previously ~P] ⊃ [[Start P] ≡ P].

[9] The example can be altered to be satisfied by finite transitions, e.g., `nil suchthat ∀y [Q(y) ⊃ Start P(y)]` but this is "irrelevant" if Q is ever empty. For quantifying over fixed finite sets, ∧ seems more appropriate than ∀. In any case, outer ∀'s have not arisen in practice.

The match network is shown below. Node 3 produces (person, account) tuples that satisfy the first disjunct above while node 4 produces those that satisfy the second. The program for node 3 accepts (acct, bal) pairs that have just been added and does the following:

```
iterate over person suchthat
  ∃ min
   [owns(person, acct) ∧
    threshhold(acct, min) ∧
    bal<min ∧
    ~Previously
        ∃ bal, min
         [balance(acct, bal) ∧
          threshhold(acct, min) ∧
          bal<min]]
 produce person,acct
```

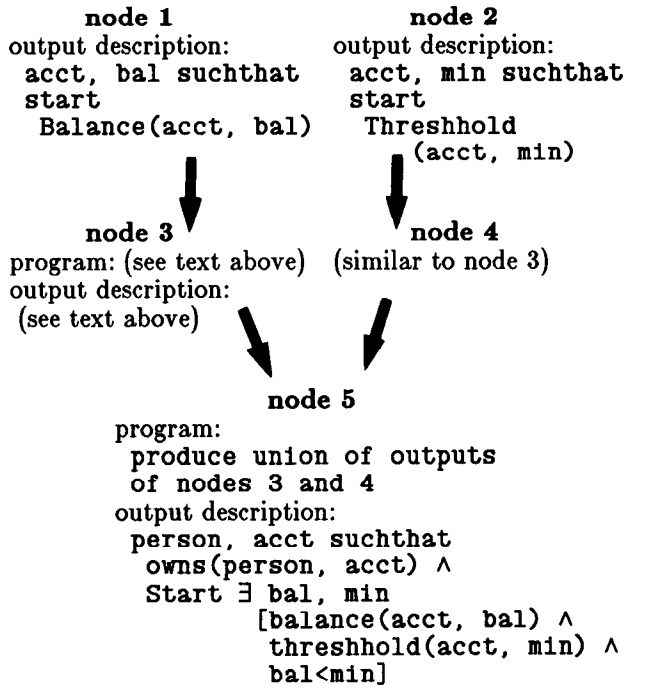

Figure 3-2: match network for low balance rule

A good query optimizer (see [Cohen 86]) can make this query reasonably cheap: if the balance is less than the threshold and this was not previously true, iterate over all owners of the account.

4. Discussion

4.1. Large atomic updates

One possible way to match atomic transitions involving several updates is to consider the updates in some order, and match each separately. This is what RETE does. Our algorithm exploits the fact that the resulting "intermediate states" are of no interest, which in some cases saves a lot of effort. For example, consider matching

`x, y suchthat Start [P(x) ∧ Q(x) ∧ R(y)]`

in response to

`atomic [insert P(1); delete Q(1)]`. This uses the matchnode for

`x, y suchthat`
`[[Start P(x)] ∧ Q(x) ∧ R(y)]`, containing the program:

```
iterate over y suchthat [Q(x) ∧ R(y)]
produce x,y
```

Normally the query optimizer will decide to test Q before generating R. In the update above, this is false, so R needn't be generated. If the atomic were treated as an addition to P followed by a deletion of Q, a tuple for every element of R would have to be created for the intermediate state and then discarded.

Notice that our algorithm accesses the database in both the previous and new states. Our implementation actually divides "the database" into a large previous database and a smaller database of updates. All code that accesses the database is therefore slightly more expensive: code to iterate over tuples of a relation first iterates through the tuples added in the current transition, and then iterates through tuples in the previous database, skipping any removed in the current transition. Classifying the updates by relation makes the extra cost nearly negligible for relations that are not being changed. Hashing the changes further reduces the cost for accessing relations that are being changed. Of course, queries inside Previously are trivial.

4.2. Sharing code

One way in which a human programmer might achieve more efficiency is to share code among several rules. One could imagine trying to transform wffs in order to obtain common subexpressions and then matching those subexpressions only once. There are several reasons not to do this. One is that it would be quite expensive. This argument is relevant to AP5 since rules are frequently added and deleted while the database is in use. Thus the expense would be paid at "run time" rather than "compile time". A more interesting reason not to factor out common subexpressions is that efficiency can suffer due to limitations this places on query optimization. In extreme cases the result may even be uncompilable.

For example, suppose we want to match these two triggers:

```
x y suchthat
    Start [P1(x, y) ∧ [Q(x) ∨ R(y)]]
x y suchthat
    Start [P2(x, y) ∧ [Q(x) ∨ R(y)]]
```

We might try to "share" the common disjunction by producing a matchnode for

`x y suchthat Start [Q(x) ∨ R(y)]`

But this is impossible - a single addition to Q would produce infinitely many matches (almost none of which will satisfy either P1 or P2). The current algorithm generates two matchnodes to react to additions to Q. They have similar programs:

```
iterate over y suchthat
  [P1(x, y) and ~Previously R(y)]
  produce x,y
```

The compiler can handle this by generating P1 and testing R. By not overspecifying we allow the query optimizer to find a good algorithm.

The RETE algorithm does try to achieve maximal sharing of common subexpressions, although it does not rewrite expressions to increase sharing. However, since it does no query optimization this sharing can

only help it.

Our present implementation does exploit a few opportunities for sharing. Rather than building two different matchnodes for the same description, it connects the same matchnode to additional outputs (thus saving the computation that would be done in the copy). This sharing occurs mainly at input nodes, but occasionally at nodes described in 3.2.2, which is the main reason for building such nodes. Sharing at other nodes is, of course, rare.

Another source of sharing is defined relations. When meaningful expressions recur, programmers tend to invent a new relation, using the expression as its definition. AP5 supports this activity. The default in AP5 is to build matchnodes to detect tuples that start or stop satisfying these definitions. The optimization decision of when to override this default is the programmer's responsibility. Fortunately, this does not seem to arise often. Other than these special cases, we have found that sharing common subexpressions seems not to help much in practice.

4.3. Alternative residues

The residues[10] computed above may seem obvious, but others are possible. For instance, Start* can be defined in many different ways. Residues of course must filter out non-matches, but they can also provide efficient, though logically redundant, filters. This presents interesting choices. Consider the example:

```
x, y suchthat
     Start [P(x, y) ∧ [~Q(x) ∨ R(y)]]
```

When R(y) becomes true we execute:

```
iterate over x suchthat
  [Previously Q(x) ∧ P(x, y)]
  produce x,y
```

If the description were known to be previously unsatisfied, the program for Start* would seem better:

```
iterate over x suchthat
  P(x, y) produce x,y
```

since it avoids testing a redundant condition. But the redundant condition does not necessarily indicate wasted effort! Suppose P were indexed on x, i.e., it's easy to generate y suchthat P(x, y) but hard to generate x suchthat P(x, y). It might be faster to generate Q and test P. AP5 allows users to choose (and even invent) representations for relations. If P(x, y) is represented by a pointer from x to y, so x suchthat P(x, y) cannot be generated at all, we've "optimized out" what was needed to match the description![11] We have tried to factor out query optimization concerns from the matchnode compiler. It therefore does not know which of these alternatives will actually be better. It seems best for Start* to not actually discard the extra conjuncts that would have been produced by Start, but mark them as "optional", so the query optimizer can use them only if they help. This facility could be put to good use in other contexts as well, e.g., optimizing queries in contexts where certain constraints can be assumed. This appears straight forward, although we have not yet done it in AP5.

The next example shows that query optimization can depend on rather esoteric information. Consider the constraint that some person has no spouse:

```
∃x [P(x) ∧ ∀y ~S(x, y)]
```

We would trigger on

```
Start* [∀x [~P(x) ∨ ∃y S(x,y)]]
```

Our algorithm indicates that whenever S(x,y) is added we should optionally check Previously P(x). If it's true (or we didn't bother to check) then recheck the original constraint. The optional check is likely to be much cheaper than rechecking the constraint (unless we maintain some additional information, such as the unmarried people or their number). The remaining question is how effective the optional filter will be, i.e., the likelihood that a person with a new spouse is also a new person. This kind of data describing update patterns is rarely available to query optimizers.

4.4. "Compiling in" more context

Another way to improve efficiency is to make use of more contextual information. For example, if we can detect that one constraint implies another, the weaker one (may) not need to be enforced.[12] Similarly, we might prove at compile time that atomically creating two new people and asserting they are married cannot violate the constraint that some person must be unmarried.

While neither Nicolas nor Forgy do the kind of reasoning required for the examples above, Nicolas uses one piece of information that AP5 does not use: constants in updates. Suppose P(2) were prohibited. For the update `insert P(x)` both Nicolas and AP5 would have to check that $x \neq 2$. However, for the update `insert P(1)`, Nicolas would detect at compile time that the constraint could not be violated, while AP5 would have to verify at run time that $1 \neq 2$. Of course, such interaction between updates and rules is unusual, and the cost of failing to notice it is small. AP5 cannot make this optimization because it does not compare particular updates to particular rules, but compiles each rule once for all updates. This reduces the cost of compiling N rules and M updates from O(NM) to O(N+M), but it prevents any optimization that relies on the interactions between particular rules and particular updates.

Similarly, the example above of one constraint implying another involves interactions among rules, and one can imagine other complicated reasoning involving updates and flow of control in the program. Such a high degree of optimization, though difficult, might be worthwhile for programs that rarely change. We have not concentrated on such optimizations because in our typical use of AP5 the programs and rules are extremely volatile.

---

[10] see 3.2.4 for the meaning of "residue"

[11] Notice that this constraint can be incrementally enforced even though it cannot be totally checked.

[12] In AP5 this is semantically equivalent only if the weaker constraint has no repair!

5. Summary

First we presented some small query language extensions, viz. descriptions, Start and Previously, which permit convenient specification of triggers involving database transitions. This language, while far short of general temporal reference, is much more expressive than first order logic without any temporal reference. Next we showed how such specifications could be compiled into a matcher that efficiently detects when the triggers are matched and by what data. In the special case of state constraints, the matcher compiles algorithms similar to those of Nicolas. However, it has some significant advantages, even in this restricted context: it produces matches that can be used to repair violations and it compiles rules independenly of any particular database transition, which makes it much easier to change the application program.

6. Acknowledgements

In addition to the RETE algorithm, AP5 has been greatly influenced by its predecessor, AP3 and the gist specification language. These in turn are the work of my colleagues at ISI, notably Neil Goldman. He also deserves credit for complaining about previous versions of this paper until an opaque algorithmic description became a set of logical manipulations whose correctness is almost obvious. I also thank Dennis McLeod for helping to give the presentation more of a database orientation.

References

[Blaustein 81] Blaustein, Barbara T., *Enforcing database assertions: techniques and applications*, Ph.D. thesis, Harvard University, August 1981.

[Cohen 86] Cohen, Donald, "Automatic compilation of logical specifications into efficient programs," in *Proceedings of the fifth national conference on artificial intelligence*, pp. 20-25, AAAI, August 1986.

[Eswaran 75] Eswaran, K.P. and D.D. Chamberlin, "Functional specifications of a subsystem for database integrity," in *Proceedings of international confernce on Very Large Data Bases*, September 1975.

[Forgy 79] Forgy, Charles L., *On the efficient implementation of production systems*, Ph.D. thesis, Carnegie-Mellon University, February 1979.

[Hammer 75] Hammer, M.M. and McLeod, D.J., "Semantic integrity in a relational data base system," in *Proceedings of international confernce on Very Large Data Bases*, pp. 25-47, September 1975.

[Hammer 78] Hammer, M.M. and Sarin, S., "Efficient monitoring of database assertions," in *Proceedings of ACM SIGMOD International conference on the management of data*, June 1978.

[Nicolas 82] Nicolas, J.-M., "Logic for improving integrity checking in relational data bases," *Acta Informatica* 18, (3), 1982, 227-253.

[Qian 86] Qian, X. and Weiderhold, G., "Knowledge-based integrity constraint validation," in *Very large data bases*, pp. 3-12, August 1986.

[Steele 84] Steele, Guy L., *Common Lisp: the language*, Digital Press, 1984.

[Stonebraker 75] Stonebraker, "Implementation of integrity constraints and views by query modification," in *Proceedings of ACM SIGMOD International conference on the management of data*, pp. 65-78, May 1975.

[Stonebraker 87] Stonebraker, M. and Rowe, L, *The postgres papers*, Electronics research laboratory, Univ. of California, Berkely, Technical Report UCB/ERL M86/85, June 1987.