Extending a functional DBPL with ECA-rules

Swarup Reddi*, Alex Poulovassilis+, Carol Small*

*Dept. of Computer Science, Birkbeck College,
University of London, Malet St. London WC1 7HX.
{carol,swarup}@dcs.bbk.ac.uk

+Dept. of Computer Science, King's College London,
Strand, London WC2R 2LS.
alex@dcs.kcl.ac.uk

Abstract. We describe how a functional database programming language can be extended with transactions and ECA-rules. Fundamental requirements of such an extension are that the declarative semantics of the language not be compromised, that the syntax of the language not be disrupted unduly, and that impedance mismatch problems be avoided. We define what we mean by events, conditions and actions and describe the semantics of ECA-rules with reference to event detection, parameter-binding, transaction scheduling and abort handling. We define the execution semantics for the ECA System in the language itself. These semantics form the basis of our implementation of the ECA System.
Keywords. ECA-rules, deductive databases, functional database languages, PFL.

1 Introduction

This paper describes how ECA-rules can be added to PFL, a functional database programming language. PFL is "declarative" in the sense that expression evaluation is confluent i.e. the order in which the sub-expressions of an expression are evaluated does not affect the value of the overall expression. This remains true of expressions which invoke functions that update the database since PFL's type system guarantees that only confluent expressions are accepted for evaluation (see [Sut95] for details). The advantages of confluence of expression evaluation are many, including the possibility of formal proofs of correctness and support for coarse grain optimisations.

PFL is also "deductive" in the sense that it supports extensionally defined relations, and also functions that play the role of intentional relations [Pou91, Sma91]. Recently there has been considerable interest in the integration of active and deductive database technologies - see, for example, [Bay93, Har93, Sim92, Zan93]. There is clearly a degree of overlap in the functionality active and deductive technologies provide, and indeed some researchers have argued that the two paradigms should be kept separate [Wid94]. Nevertheless, we view the functionalities provided by the two paradigms as complementary. For example, to provide the semantics of ECA-rules in a deductive system either the deductive rules must be modified (with the result that application semantics are "embedded" within these rules, and so are hard to manage and modify), or the database must be polled periodically (with the result that if the polling period is too short then system resources are wasted, whereas if it is too long then there is a loss of "currency"). Conversely, considerable investment has been made into query optimisation and theoretical foundations for deductive languages, and it is clearly desirable to use this work in the context of active systems.

This paper describes how ECA-rules can be added to PFL, and indeed any functional database language of similar expressiveness. The challenges involved in this task are two-fold: (i) At the syntactic level, how can ECA-rules be specified in a style that is consistent with the rest of the language? (ii) At the semantic level, how can ECA-rules be supported whilst retaining the declarative semantics of the language?

With respect to the first of these, we first extend PFL to support *transactions*. These are sequences of the form $e_1;\ \ldots;\ e_n$ where each e_i is an expression. An ECA-rule then has three components: an expression *events* (the "event query"), an expression *condition* (the "condition query"), and a sequence of expressions, $a_1;\ \ldots;\ a_n$ (the "actions"). The firing of an ECA-rule comprises the following steps: the evaluation of *events*, yielding a (possibly empty) list of expressions, *evs*, corresponding to event occurrences; the removal of each element of *evs* which does not satisfy the *condition*, giving a new list of expressions evs' (the "parameter list"); the scheduling of a transaction $a_1\ evs'; \ldots ; a_n\ evs'$.

With respect to the second of the above points, transactions and ECA-rules are not "first class objects" in our language: functions cannot take transactions or ECA-rules as arguments nor can they return them as results and so transactions and ECA-rules cannot be "embedded" within expressions. Also, when processing a transaction $e_1;\ \ldots;\ e_n$, each e_i is evaluated in turn and the ECA-rules are fired only when the evaluation of e_i terminates (so our ECA system uses a production system model [Han92]). The evaluation of each expression e_i within a transaction is thus confluent. However, events - such as clock and external events - which need to be detected and acted upon *during* expression evaluation are not supported.

The structure of the remainder of this paper is as follows. In Section 2 we give a brief description of PFL, including its type system and its support for relations, transactions and ECA-rules. In Section 3 we specify the semantics of transaction execution using PFL itself as the meta language. In Section 4 we summarise our contribution and give directions of further work.

2 Preliminaries

PFL supports the following commands each of which is described in this section, apart from command (i) which is discussed in Section 3.3:

(a) the declaration of a new type,
(b) the definition of a type synonym,

(c) the declaration of the type of a new function,
(d) the specification of an equation for a function,
(e) the declaration of a new relation,
(f) the definition of an ECA-rule,
(g) the evaluation of a transaction,
(h) the evaluation of an expression (which is treated as a transaction consisting of a single expression),
(i) the casting of an arbitrary function as a primitive event.

The new features of the language beyond those described in previous papers [Pou91, Sma91] are (f), (g) and (i). In the sequel, we use `teletype` font for PFL code and *italic* font for syntactic variables. We use $e, e_1, \ldots, e_n$ to denote expressions, $t, t_1, \ldots, t_n$ to denote types, and $e_1, \ldots, e_n :: t$ to indicate that the expressions $e_1, \ldots, e_n$ all have type t.

2.1 Types

PFL is a polymorphic, statically typed, functional database programming language. It is *higher-order* in the sense that functions may take functions as arguments and return functions as results. It is computationally complete, and also update-complete in the sense that any computable transformation over relations can be defined [Sut95].

PFL provides a set of built-in types and facilities for the user to declare new types and their constructors. The built-in types include `Num`, `Str` and `Chr` which are respectively populated by numbers, strings and characters. Also built-in are list, product and function types: for any types $t, t_1, \ldots, t_n$, the type $[t]$ consists of lists of elements of type t; the type $(t_1, \ldots, t_n)$ consists of n-tuples of the form $(e_1, \ldots, e_n)$ where $n > 1$ and each $e_i :: t_i$; and the type $t_1 \rightarrow t_2$ consists of functions that map values of type t_1 to values of type t_2. The list constructors are $[\ ]$ and $(:)$ where, for all types t, $[\ ] :: [t]$ and $(:)\, e_1\, e_2 :: [t]$ if $e_1 :: t$ and $e_2 :: [t]$. The constructor $(:)$ can be written infix, in which case it is unbracketed. We support some syntactic sugar for enumerated lists in that an expression $[e_1, e_2, \ldots, e_n]$ is synonymous with $e_1 : [e_2, \ldots, e_n]$. Also, $\rightarrow$ is right-associative so that $t_1 \rightarrow t_2 \rightarrow t_3$ and $t_1 \rightarrow (t_2 \rightarrow t_3)$ are synonymous.

New types and type synonyms can be declared at any time. User-declared type names and constructors must start with an upper-case letter. For example, our definition of the ECA system in Section 3 below assumes two types, `SMode` ("scheduling mode") and `AMode` ("abort mode"), and two synonyms, `TranId` ("transaction identifier") and `EventId` ("event identifier") for the type `Num`:

```
SMode   ::= Imm | Def;
AMode   ::= ParentChild | ChildParent | Mutual | Independent;
TranId  == Num;
EventId == Num;
```

Note that `::=` and `==` introduce new types and type synonyms respectively.

2.2 Expressions and Functions

An *expression* in PFL is one of: a variable, a function name, a constructor, or the application of one expression to another. Application is left-associative and has higher precedence than any operator. Variables and function names must start with a lower-case letter and constructors must start with an upper-case letter. The keyword `let` introduces a local definition for a variable. For example, `let x==2; in x*x` is equivalent to `2*2` while `let a==1; b==2; in (a,b)` is equivalent to `(1,2)`.

Also supported are *list comprehensions* [Pey87]. They take the form $[e \mid q_1;\ \ldots;\ q_n]$, and are read as "the list of e such that q_1 and ... and q_n". Each q_i is either a boolean-valued expression which must be satisfied, or a generator of the form $p \leftarrow l$ which is read as "the pattern p takes each value in the list l in turn" (a *pattern* being an expression that contains no functions). For example, the expression

```
[(x,y) | x <- xs; y <- ys; pred (x,y)]
```

returns the subset of the cartesian product of the lists `xs` and `ys` which satisfies the predicate `pred`.

The built-in functions include the usual arithmetic and relational operators, and also the operator `++` which is used both for concatenating two strings and for appending one list to another. The user may define new functions at any time. Functions are defined by means of equations and we give three examples below. The symbol `==` is used for equation definitions while the symbol `=` is the equality operator. The type of a function is automatically inferred from its defining equations although the user may optionally pre-declare its expected type. As an aid to readability, we include the expected types in our examples:

```
if :: Bool->a->a->a;
if True x y          == x;
if False x y         == y;

case :: [(Bool,a)]->a;
case ((bool,val):rest) == if bool val (case rest);

fold :: (a->b->a)->a->[b]->a;
fold op end []         == end;
fold op end (x:xs)     == fold op (op end x) xs;
```

`case`, when applied to a list of pairs `[(b1,v1),...,(bn,vn)]`, returns the first value `vi`, such that `bi` evaluates to `True`. Given a binary operator `op`, an expression `end` and a list of expressions `[e1,..,en]`, `fold op end [e1,...,en]` is equivalent to

```
e1 op (e2 op (... op (en op end) ... )).
```

2.3 Bulk Data

Bulk data is stored in a class of 0-ary, list-valued functions we term *relations*. The extent of a relation is initially the empty list, and can be updated by means of two built-in functions, `include` and `exclude`. These both take as arguments the name of a relation, r, and a value, e. `include` adds e to the head of the extent of r, provided e is not already present there, and returns r. `exclude` removes e from the extent of r and returns r.

Although `include` and `exclude` update relations by side-effect, PFL's type checker will reject any query or function definition which would compromise the confluence of the language (see [Sut95] for details). For example, if `r` is an initially empty relation of type `[Num]` and the function `ext` returns the extent of its argument, then the expression `(ext r, include 1 r)` evaluates to either `([],r)` or `([1],r)`, depending on whether the first or second component is evaluated first. Thus, the expression is not confluent and would be rejected by the type checker.

2.4 Transactions

A transaction is specified by a command of the form

$$tranStart;\ e_1;\ e_2;\ \ldots;\ e_n;\ tranEnd;$$

where $e_1, e_2, \ldots, e_n$ are arbitrary expressions and *tranStart* and *tranEnd* are built-in functions indicating the start and end of a transaction.

The $n + 2$ expressions comprising a transaction are evaluated in turn. The evaluation of the current expression may cause the occurrence of one or more *primitive events* which are recorded by the evaluator in a system-reserved relation, the `delta` relation. The invocation of any of the functions `include`, `exclude`, `transStart` and `transEnd` constitutes a primitive event, as also does the invocation of any other function so declared by the user (see Section 3.3). A further system-reserved relation, the `history` relation, contains a cumulative record of all primitive events (including those in the `delta` relation). The `history` relation is cleared whenever the database is committed while the `delta` relation is cleared immediately before the evaluation of each e_i.

The `delta` and `history` relations cannot be accessed directly because these relations are type-incorrect. Instead, for each class of primitive event an individual history and delta relation are provided which are derived from the overall `history` and `delta` relations. In particular, for each relation `r::[t]`, the relations `rIncH, rIncD, rExcH, rExcD :: [(EventId,TranId,t)]` respectively record the cumulative inclusions, the delta inclusions, the cumulative exclusions, and the delta exclusions with respect to `r`. Similarly, `tranStartH, tranStartD, tranEndH, tranEndD::[(EventId,TranId)]` respectively record the cumulative and delta transaction start events, and the cumulative and delta transaction end events.

Thus, from the user's perspective, each class of primitive event is associated with two relations: a cumulative history that contains a tuple for each event occurrence since the database was last committed and a delta relation that contains a tuple for each event occurrence during the evaluation of the transaction's current expression. These relations are type-correct views of the system-reserved `history` and `delta` relations, which, for the sake of simplicity, we have materialised in our current implementation.

The ECA-rules are fired after the evaluation of each expression of the transaction has completed and the occurrence of events (both primitive and composite) is detected by the event queries of the ECA-rules (see Section 2.5), which interrogate the cumulative and delta relations associated with each primitive event.

2.5 ECA-Rules

The user may define new ECA-rules by commands of the form

$\_ecaRule\ i \quad == \quad (events, condition, [action_1, \ldots, action_m], sm, am);$

where:

- i is a unique number which defines the priority of the ECA-rule.
- $events$ is an expression of type $[t]$ for some type t. It must not update the database. Its evaluation yields a list of expressions, $[ev_1, \ldots, ev_n]$, where each ev_j represents the occurrence of an event during the evaluation of the current expression.
- $condition$ is an expression of type $t \rightarrow Bool$ and must also not update the database. $condition$ is applied to each element of the list $[ev_1, \ldots, ev_n]$ and only those elements for which $condition\ ev_j$ evaluates to $True$ are retained.
- Each $action_i$ is an expression (which can, and usually will, update the database) to be applied to the list of events which satisfy the $condition$. Thus overall the user can regard the ECA-rule as specifying the following sequence of "reactions":

  $let\ evs\ ==\ [e \mid e\ \leftarrow\ events;\ condition\ e];$
  $in\qquad tranStart;\ action_1\ evs;\ \ldots;\ action_m\ evs;\ tranEnd;$
- sm is an expression of type `SMode`. If the value of sm is `Imm` ("immediate") then a new transaction comprising the reactions will be run prior to resuming the current transaction. If the value of sm is `Def` ("deferred") then a new transaction comprising the reactions will be run after the completion of the top-level transaction.
- am is an expression of type `AMode` representing the abort dependency between the original transaction, $tran_o$, and the reaction transaction, $tran_r$:
  - `ParentChild` - if $tran_o$ aborts then $tran_r$ is to abort;
  - `ChildParent` - if $tran_r$ aborts then $tran_o$ is to abort;
  - `Mutual` - if either $tran_o$ or $tran_r$ aborts then so must the other; and
  - `Independent` - there is no abort dependency between $tran_o$ and $tran_r$.

To illustrate, suppose that we have three relations: `accSite` which relates road traffic accidents (`acc` below) to the grid references (`gr` below) at which they occurred, and `blackSpots` and `mWayBlackSpots` which record the grid references of all accident black spots and motorway black spots respectively (here, a black spot is a grid reference at which six or more accidents have occurred). We can define ECA-rules which (i) enforce the integrity constraint that accidents must occur on roads, and (ii) materialise `blackSpots` and `mWayBlackSpots` from `accLocation` as follows, where `$` returns the length of a list, `inclist` adds a list of tuples to a relation, and `error` prints its argument and causes the evaluation of the current expression to abort:

```
_ecaRule 1  == (e1,c1,[a1],Imm,ChildParent);
_ecaRule 2  == (e2,c2,[a2,a3],Def,Mutual);

e1          == [tuple | (eId,tId,tuple) <- accSiteIncD ];
c1 (acc,gr) == roadClassAt gr = None;
a1 x        == error "accident not located on road";

e2          == [tuple | (eId,tId) <- tranEndD;
                   (eId2,tId2,tuple) <- accSiteIncH; tId=tId2];
c2 (acc,gr) == $ [gr | (a2,gr2) <- accSite; gr = gr2] > 5;
a2 x        == inclist [gr | (acc,gr)<- x] blackSpots;
a3 x        == inclist [gr | (acc,gr)<- x; roadClassAt gr = MWay]
                   mWayBlackSpots;
```

The `ChildParent` abort mode of the first rule causes any transaction violating the integrity constraint to be aborted, whilst the `Def` scheduling mode of the second rule causes re-materialisation of `blackSpots` and `mWayBlackSpots` only at the end of a transaction which updates `accSite`.

3 Transaction Execution Semantics

In this section we specify the semantics of transaction execution using PFL itself as the meta language. We thus obtain a formal declarative semantics for transaction execution which not only forms the basis of our implementation of the ECA System but also is immediately accessible to the PFL user.

For the purposes of manipulation by the ECA System (and also the evaluator) PFL expressions are represented as values of type Expr:

```
Expr ::= C Str | S Str | N Str | Con Str | Rel Str |
         Var Str | Ap Expr Expr | Error
```

Here the constructors `C`, `N`, `S`, `Con` and `Rel` are understood to state that the associated string represents a character, number, string, constructor, or relation name respectively. The constructor `Error` is discussed in Section 3.2 below. The ECA system does not need to distinguish between variables and function names (except relation names) so the constructor `Var` is used for either of these. For example, the expression `head [1,2]` is represented as:

```
Ap (Var "head") (Ap (Ap (Con "(:)") (N "1"))
                    (Ap (Ap (Con "(:)") (N "2")) (Con "[]")))
```

3.1 Data Structures Used by the ECA System

The ECA System uses the following type synonyms and functions:

```
Tran       ==  (TranId,[Expr]);
Schedule   ==  [Tran];
```

```
DBState     ==  [(Str,[Expr])];
ECAState    ==  (EventId,TranId,AbortGraph,ECARules,History);
AbortGraph  ==  [(AMode,TranId,TranId)];
ECARules    ==  [ECARule];
ECARule     ==  (Expr,Expr,[Expr],AMode,SMode);
History     ==  (Str,[(EventId,TranId,Expr)]);
nextEv      ::  EventId->EventId;
nextTran    ::  TranId->TranId;
getRel      ::  DBState->Str->[Expr];
incRel      ::  DBState->Str->Expr->DBState;
clearDeltas ::  DBState->DBState;
```

`EventId`, `TranId` are synonyms for numbers that uniquely identify events and transactions (see Section 2.1). `EventId`s and `TranId`s are totally ordered and the function `nextEv` (`nextTran`), when applied to an `EventId` (`TranId`), returns the next `EventId` (`TranId`) according to the total ordering. `EventId`s are assigned in the order in which events occur. In contrast, `TranId`s are assigned in the order in which transactions are scheduled.

`Tran` is a synonym for a pair comprising a transaction identifier and a list of the expressions comprising the transaction.

`Schedule` is a synonym for a list of transactions to be executed in sequence.

`DBState` is a synonym representing the database state. It consists of a list of pairs, the first component being the name of a function and the second being its definition, of type `[Expr]`. In the case of a non-relational function, the definition consists of a single `Expr`. In the case of a relation, the definition consists of one `Expr` for each element of the relation. For example, a relation seen by the user as having name `r` and extent `[1,2]` is held in a `DBState` as the pair `("r",[N "1", N "2"])`.

The functions `getRel`, `incRel` and `clearDeltas` interrogate and alter a `DBState`. For a database state `db`, relation name `r`, and expression `e`, `getRel db r` returns the extent of `r`, `incRel db r e` adds `e` to the head of the extent of `r` provided it is not already present and returns `db`, and `clearDeltas db` sets to the empty list the extent of every delta relation i.e. every relation whose name ends in "D", together with the system-reserved `delta` relation.

`ECAState` is a synonym representing the current state of the ECA System. It comprises the "next" identifier to be assigned to an event; the "next" identifier to be assigned to a transaction; an abort graph which, in the case of a transaction aborting, is used to identify the other transactions to be aborted; the list of ECA-rules; and the system-reserved `history` relation.

An `AbortGraph` is a list of triples each of which contains a pair of transaction identifiers and the abort dependency between them.

`ECARules` is a synonym for a list of 5-tuples representing the ECA-rules of the system. The order of the rules within this list is according to their priority and consequently the user-specified rule identifier (see Section 2.5) is not needed. Each ECA-rule comprises an event and condition (both of type

Expr), an action (of type [Expr]), an abort-dependency (of type AMode) and a scheduling-mode (of type SMode).

3.2 Specification of the ECA System

Our specification of the ECA System assumes a function `eval::Expr->DBState ->(Expr,DBState)` which takes an expression and a database state, and fully evaluates the expression while logging all primitive event occurrences in the `delta` relation. `eval` returns a pair comprising the reduced expression and updated database state. If an error arises during the evaluation of an expression (e.g. an attempt to divide by zero, or an invocation of the `error` function) then `eval` reduces the expression to `Error` and returns the database unchanged. Thus `eval` is in effect the PFL evaluator which, apart from supporting retrieval from, and update of, relations and logging of primitive events, is much like the evaluator for a conventional functional programming language (for which implementation details may be found in [Pey87]).

At the top level of our specification is `execSched`, which takes as its argument a triple comprising a `DBState`, an `ECAState` and a `Schedule`. It repeatedly calls `execExp` (see below) to process the transaction at the head of the schedule (i.e. the *current transaction*) until the schedule is empty. It returns the updated database and ECA states:

```
execSched :: (DBState,ECAState,Schedule)->(DBState,ECAState);
execSched (db,eca,[])           == (db,eca);
execSched (db,eca,(tran:trans)) ==
    execSched (execExp tran (db,eca,trans));
```

`execExp`, defined below, processes each expression of the current transaction in turn. If the transaction is non-empty, `execExp` removes its first expression, `exp1`, and puts the remainder of the transaction back at the head of the schedule. It then clears all the delta relations within the current database state and calls `eval` to evaluate `exp1` with respect to the updated database state. Thus, the `delta` relation in `db2` contains only those primitive events resulting from the evaluation of `exp1`. If the evaluation of `exp1` terminates normally, `ecaControl` (see below) is called to update the history relation, to do event-detection and to schedule reactions. If the evaluation terminates abnormally, `abortHandle` (discussed in Section 3.4) is called to abort the current transaction:

```
execExp :: Tran->(DBState,ECAState,Schedule)
          ->(DBState,ECAState,Schedule);
execExp (tId,[]) (db,eca,sched)          ==  (db,eca,sched);
execExp (tId,exp1:exps) (db1,eca,sched1) ==
    let  sched2     == (tId,exps):sched1;
         (exp2,db2) == eval exp1 (clearDeltas db1);
    in   if (exp2 = Error)
             (abortHandle tId (db2,eca,sched2))
             (ecaControl  tId (db2,eca,sched2));
```

The arguments to `ecaControl` are the identifier of the current transaction, the current database state, the current ECA state and the current schedule. `ecaControl` returns the updated database state, ECA state and schedule. The tasks required of `ecaControl` are twofold. Firstly, the overall `history` relation and the individual history and delta relations associated with each primitive event need to be updated: this is effected by `logEvent` whose definition we defer until we have specified event detection and scheduling. Secondly, event detection must be undertaken and any reactions arising must be scheduled (i.e. inserted into the appropriate location in the `Schedule`):

```
ecaControl :: TranId->(DBState,ECAState,Schedule)
              ->(DBState,ECAState,Schedule);
ecaControl tId (db1,eca1,sched1) ==
    let  newEvents     == getRel db1 "delta";
         (db2,eca2)    == fold (logEvent tId) (db1,eca1) newEvents;
         (eca3,sched2) == detectAndSched tId (db2,eca2,sched1);
    in   (db2,eca3,sched2);
```

In practice, only those ECA-rules whose event expressions reference updated history and/or delta relations need be considered during event detection. However, for the purposes of specifying the event detection semantics we ignore this optimisation. Thus, `detectAndSched` calls `schedRule` to process each ECA-rule in turn, and then returns the updated `ECAState` and `Schedule`:

```
detectAndSched :: TranId->(DBState,ECAState,Schedule)
                  ->(ECAState,Schedule);
detectAndSched tId (db,(e,t1,g1,rules,h),sched)  ==
    let  (imm,def,t2,g2) ==
             fold (schedRule tId db) ([],sched,t1,g1) rules;
    in   ((e,t2,g2,r,h),imm++def);
```

`schedRule` lies at the heart of the ECA System. It considers a particular ECA-rule, evaluates the parameter list, and then schedules the reactions and updates the abort graph as necessary. In addition to the current transaction identifier, database state and ECA-rule, `schedRule` takes as its argument a 4-tuple. The first two components of this are the immediate and deferred reactions so far accumulated (i.e. the reactions arising from the ECA-rules already processed). The third component is the next unassigned transaction identifier and the fourth is the abort graph. `schedRule` returns an updated version of this 4-tuple. That is, if the ECA-rule under consideration generates a reaction, then this reaction is added to the appropriate list of reactions (according to the rule's scheduling mode), the abort graph is updated and the unassigned transaction identifier is incremented. When processing a rule the evaluation of the parameter list may terminate abnormally. In this case we invoke the abort handler as soon as possible by scheduling as the earliest possible transaction one whose list of expressions is simply `[Error]`:

```
schedRule :: TranId->DBState->([Tran],[Tran],TranId,AbortGraph)
```

```
             ->ECARule->([Tran],[Tran],TranId,AbortGraph);
schedRule tId db (imm,def,tid2,graph) (e,c,a,mode,abDep) ==
    let (pl,db) == eval (Ap (Ap (Var "parmList") e) c) db;
        reacts  == [Ap ai pl | ai <- a];
        tran    == (tid2,[tranStart]++reacts++[tranEnd]);
        g2      == (abDep,tId,tid2):graph;
        nextId  == nextTran tid2;
    in  case [(pl = Error,    ((tid2,[Error]):imm,def,nextId,g2)),
              (pl = Con "[]", (imm,def,tid2,graph)),
              (mode = Imm,    (imm++[tran],def,nextId,g2)),
              (mode = Def,    (imm,def++[tran],nextId,g2))];
```

Here `parmList` is a function defined as
`parmList events cond == [e | e <- events; cond e];`
and stored within the database, `db`, in the appropriate form.

There are two points to note regarding the event-detection and scheduling. The first is that, since each rule is processed in turn, the order in which reactions are scheduled is the same as the order of precedence of the rules. The second is that, since the event and condition expressions of ECA-rules only query the `DBState`, all the parameter lists are evaluated against the same `DBState`.

We now return to the logging of events. To recap, `eval` evaluates an expression and records in `delta` the primitive events that occur. In order to undertake event detection, the cumulative `history` relation and the individual history and delta relations need to be updated to reflect the events recorded in `delta`. For the `history` relation this is accomplished by prefixing to it a triple consisting of the current event, the current `TranId` and the "next" `EventId` (the `EventId` in the `ECAState` is incremented accordingly). The updating of individual history and delta relations is accomplished by `incHistAndDelta`, defined below. `logEvent` returns the updated `DBState` and `ECAState`:

```
logEvent : TranId->(DBState,ECAState)->Expr->(DBState,ECAState);
logEvent tId (db1,(e,t,graph,rules,history)) event  ==
    let eca2   ==  (nextEv e,t,graph,rules,(e,tId,event):history);
        db2    ==  incHistAndDelta db1 (e,tId,event);
    in  (db2,eca2);
```

As described in Section 2.4, the individual history and delta relations are views of the system-reserved `history` relation and we materialise these views for ease of both explanation and implementation. `incHistAndDelta` first calls the `unwind` function to unpack the event expression into the name of the function whose evaluation constituted the primitive event (for example `include`, `exclude`, `tranStart` or `tranEnd`) and the list of that function's arguments. This information is then used to construct the name of the appropriate history and delta relations. The expression to be added to the extents of these relations represents a tuple comprising the `EventId`, the `TranId`, and the arguments to the function. The function `makeTuple :: [Expr] -> Expr`, which we have not specified here, is used to construct this expression: it takes a list of n expressions and returns a single expression representing an n-tuple. Also used is a built-in function `numToExpr :: Num->Expr` which converts a number into its representation as an `Expr`. Overall, the functions required for event logging are:

```
incHistAndDelta :: DBState->(EventId,TranId,Expr)->DBState;
unwind          :: (Expr,[Expr])->(Str,[Expr]);
relEvent        :: DBState->[Expr]->Str->EventId->TranId->DBState;
funEvent        :: DBState->Str->(EventId,TranId,[Expr])->DBState;

incHistAndDelta db (eid,tid,exp) ==
    let  (fun,args) == unwind (exp,[]);
    in   case [ (fun = "include", relEvent db args "Inc" eid tid),
                (fun = "exclude", relEvent db args "Exc" eid tid),
                (True,  funEvent db fun (eid,tid,args))];

unwind ((Ap exp1 exp2),args) == unwind (exp1, exp2:args);
unwind (Var fun,args)        == (fun,args);

relEvent db [(Rel r),val] incOrExc eid tid ==
    funEvent db (r++incOrExc) (eid,tid,[val]);

funEvent db fun (eid,tid,args) ==
    let  tuple  == makeTuple (numToExpr eid:(numToExpr tid:args));
    in   incRel (incRel db (fun++"D") tuple) (fun++"H") tuple;
```

3.3 Casting a Function as a Primitive Event

This is analogous to designating the invocation of a method as an event in an active object-oriented system. There are two restrictions on functions that can be cast as primitive events. The first is that the function cannot be called by the event or condition queries of any ECA-rule since these may not alter the `DBState`. The second is that the function must have a monomorphic type so that the relation in which the event occurrences are recorded only contains tuples of a single type.

For any function $f :: t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t$ cast as a primitive event, the names of its history and delta relations are the function name suffixed by "H" and "D", respectively, and their types are both $[(EventId, TranId, t_1, \ldots, t_n)]$.

When declaring the function to be a primitive event, the user is required to specify a sequence of inverse events to be executed in the case that an abort occurs (see below). Any primitive events that occur during the evaluation of a function will automatically be compensated. Consequently, the evaluation of the user-specified inverse events can be regarded as a "do instead" activity.

3.4 Abort Handling

As we saw in the definition of `execExp` above, if an expression evaluates to `Error` the transaction containing this expression is aborted by the invocation of abortHandle. For reasons of space, we can only briefly describe the functionality of abortHandle here and refer the reader to [Red95] for its definition.

First, the other transactions that must be aborted are identified using the AbortGraph component of the ECAState. Then, rather than being rolled-back, these transactions are *compensated* by compensating each of their constituent primitive events in the reverse order of their occurrence. In particular, inclusions and exclusions of values in relations are compensated by exclusions and inclusions respectively, while user-defined primitive events are compensated by their user-defined sequence of inverse events. A single compensating transaction compensates all the aborted transactions. The compensating events are not logged but instead the events that they compensate are removed from the overall and individual history relations.

4 Conclusions

We have described how the functional database language PFL can be extended with ECA-rules and have given an execution semantics for the extended system by using PFL itself as the meta language. Our approach to ECA-rules can be summarised as follows:

- *No impedance mismatch*: PFL's computational model, type system, and data structures are unaffected by the implementation of the ECA System. No new language need be learnt for specifying events, conditions or actions.
- *Simplicity*: a primitive event occurrence is simply a value in the appropriate history and delta relations. Thus the specification and detection of both composite and primitive events simply comprises the definition and evaluation of queries over these relations.
- *Expressiveness*: events and conditions are specified in a computationally complete language and actions in an update-complete one.
- *Extensibility*: a function can be cast as a primitive event at any time.
- *Generality*: the user has access to both individual history and delta relations and can therefore make explicit choices regarding event consumption for each event instead of having them implicitly assumed and fixed for all events.

Figure 1 summarises our approach to supporting ECA-rules according to the criteria proposed by Paton *et al.* [Pat93]. For reasons of space we can only discuss a few of these criteria here and refer the reader to [Red95] for a fuller discussion.

Condition evaluation occurs when the event is detected, although it can be deferred until, say, the end of the transaction by defining an ECA-rule whose event query detects a tranEnd event. Since both the condition and the action are expressions, a condition can also be moved to the action part of a rule and can hence be evaluated when the action is scheduled to occur.

PFL allows an action to occur either immediately as a sub-transaction, or as a separate transaction. In the latter case the action modes "Detached Dependent" and "Detached Independent" are obtained by using the AModes ParentChild and Independent respectively and the SMode Def. The remaining AModes supported

Events	Type	$\subset$ { Primitive, Composite }
	Source	$\subset$ {Structure Operation, Transaction, Behaviour Invocation, Error }
	Level	$\subset$ { Instance, Collection }
	Role	$\in$ { Mandatory }
Condition	Mode	$\subset$ { Immediate, Deferred, Detached }
	Role	$\in$ { Optional }
Action	Mode	$\subset$ { Immediate, Detached Dependent, Detached Independent }
	Options	$\in$ { Update-Db, Abort, Do Instead }
Execution Model	Transition Granularity	$\in$ { Set }
	Binding Model	$\in$ { Set }
	Constraints	$\in$ { }
	Scheduling	$\in$ { All Fired }
	Priorities	$\in$ { Numerical }
Management	Operations	$\in$ { }
	Description	$\in$ { Programming Language }
	Adaptability	$\in$ { Run Time }
	Data Model	$\in$ { Functional }

Fig. 1. Categorisation of PFL's support for ECA-rules

are not covered by the criteria of Paton *et al.* Any sequence of expressions can be specified as the action part of an ECA rule, thus allowing database updates to be performed and transactions to be aborted (by evaluation of an expression leading to `Error`). The action cannot, however, cause the firing of other ECA-rules to be over-ridden.

At present we do not allow rules to activate or deactivate other rules - although such a facility can be simulated by explicitly checking the value of a switch as part of an event query and turning the switch on or off as part of an action. Rules are not "first class" values: they can be changed only at the command level and cannot be added or deleted *dynamically* by evaluating an expression. However, since rules can be added and deleted once the database is populated we have classified PFL's rule adaptability as "run time". Although we believe that our language could be extended to offer dynamic adaptability of rules, we are uncertain of the desirability of this since it would make the semantics considerably more complicated.

There are a number of areas in which further work is required. Firstly, our current method for assigning priorities to ECA-rules is rudimentary and we plan to provide the user with tools to (re)prioritise ECA-rules. Secondly, we need to investigate optimisation strategies for event detection and condition evaluation. In [Pou91, Sma91] we introduced functions that play the role of intentional relations and in [Red93] developed techniques for the efficient enforcement of integrity constraints over them. We believe that these techniques can be generalised to optimise event detection for event queries that are intentional relations. Thirdly, we need to support the database commands of Section 2 as primitive events and to provide their inverse events. Finally, the database state resulting from the evaluation of an expression e_i followed by the execution of the ECA-rules that it triggers is in general dependent on the order in which sub-expressions of e_i are evaluated. For example, given two relations `r1` and `r2` of type `[Num]` the evaluation of the tuple `(include 1 r1, include 1 r2)` is confluent. However, the detection of a composite event consisting of an update of `r1` followed by an update of `r2` depends on the order of evaluation of the components of the tuple. Thus we need to explore syntactic criteria which guarantee that a set of ECA-rules gives rise to a unique database state.

Acknowledgements

We are grateful to David Sutton for discussions on PFL's linear type system. This work was supported by EPSRC grant no. GR/J 98134.

References

[Bay93] Bayer, P., and Jonker, W. A framework for supporting triggers in deductive databases, *Proc. 1st Int. Workshop on Rules in Database Systems*, Edinburgh, 1993.

[Han92] Hanson, E.N. Rule condition testing and action execution in Ariel, *Proc. ACM SIGMOD Conference*, San Diego, 1992.

[Har93] Harrison, J.V. and Dietrich, S.W. Integrating active and deductive rules, *Proc. 1st Int. Workshop on Rules in Database Systems*, Edinburgh, 1993.

[Pat93] Paton, N.W. Díaz, O. Williams, M.H. Campin, J. Dinn, A. and Jaime, A. Dimensions of active behaviour, *Proc. 1st Int. Workshop on Rules in Database Systems*, Edinburgh, 1993.

[Pey87] Peyton-Jones, S.L. *The implementation of functional programming languages*, Prentice-Hall, 1987.

[Pou91] Poulovassilis, A. and Small, C. A functional approach to deductive databases, *Proc. 17th VLDB Conference*, Barcelona, 1991. Santiago, 1994.

[Red93] Reddi, S. Integrity constraint enforcement in the functional database language PFL, *Proc. 11th British National Conference on Databases*, Keele, July 1993.

[Red95] Reddi, S. Poulovassilis, A. and Small, C. Extending PFL with ECA-rules, Technical Report, Dept, of Computer Science, Birkbeck College. July 1995.

[Sim92] Simon, E., Kiernan, J. and de Maindreville, C. Implementing high level active rules on top of a relational DBMS, *Proc. 18th VLDB Conference*, Vancouver, 1992.

[Sma91] Small, C. and Poulovassilis, A. An overview of PFL, *Proc. DBPL-3*, Nauplion, 1991.

[Sut95] Sutton, D. and Small, C. Extending functional database languages to update completeness. To appear in *Proc. 13th British National Conference on Databases*, Manchester, July 1995.

[Wid94] Widom, J. Research issues in active database systems, *ACM SIGMOD Record*, 23(3), 1994.

[Zan93] Zaniolo, C. A unified semantics for active and deductive databases, *Proc. 1st Int. Workshop on Rules in Database Systems*, Edinburgh, 1993.