, , 1–23 ()

A Formal Semantics for an Active Functional DBPL

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

SWARUP REDDI AND CAROL SMALL {swarup,carol}@dcs.bbk.ac.uk
Dept. of Computer Science, Birkbeck College, Malet St. London WC1E 7HX. UK.

Abstract. We describe how the functional database programming language PFL is extended with an active component without compromising either its declarative semantics or its syntax. We give a formal specification of the active component using PFL itself, including event specification and detection, parameter-binding, reaction scheduling and abort handling. We describe how a user-specified function can be cast as a primitive event, and discuss the expressiveness of events and the optimisation of event detection.

Keywords: ECA-rules, deductive databases, functional database languages, PFL.

1. Introduction

Recently there has been considerable interest in the integration of active and deductive database technologies - see, for example, [25, 13]. Deductive databases provide mechanisms for defining intentional relations in terms of extensional relations, enforcing constraints over these relations, and updating relations. Active databases support Event-Condition-Action (ECA) rules which respond to events by executing actions if conditions are satisfied. Thus active database technology can be used for constraint enforcement, materialising views, providing audit trails, and so forth.

Although there is clearly a degree of overlap in the functionality active and deductive technologies provide, we view them as being complementary. To provide the functionality of ECA-rules in a deductive DBMS 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 resources are wasted, whereas if it is too long then there is a loss of currency). Conversely, considerable investment has been made into such areas as query optimisation and theoretical foundations for deductive languages, and we exploit some of this work in the context of our active system.

PFL is a functional database programming language (DBPL) that is deductive in that it supports extensionally defined relations and functions that play the role of intentional relations [19, 26]. A key feature of functional languages is that expression evaluation is *confluent*: ie. the value of an expression is independent of the order of evaluation of its sub-expressions. In the case of PFL this is also true of expressions which update the database since the type checker guarantees that only confluent expressions are accepted for evaluation. (This usage of the term "confluent" is distinct from, but related to, the normal usage in the area of active databases in which it denotes a rule set which is deterministic.)

In contrast to logic DBPLs, functional ones support higher-order functions which abstract out recursion patterns. Moreover, the deterministic semantics of function evaluation can be used to define default rules, whereas logic DBPLs require extra-logical constructs such as negation-as-failure. In contrast to imperative DBPLs, such as those based on C++ and Smalltalk, functional DBPLs have simple semantics and are confluent. These features allow properties of expressions, such as termination and equivalence with other expressions, to be formally proven. Thus whereas imperative DBPLs apply different optimisations to 'program' and 'DML' statements, and support only fine-grained optimisations for program statements, functional DBPLs support powerful, coarse-grained optimisations (such as those discussed in [20]) for all sub-expressions of a query since they do not distinguish between 'program' and 'DML' statements.

PFL is being used for the analysis of road accident data in a long-term project at the Centre for Transport Studies, University College London. The data is of poor quality, and requires extensive validation and default assumptions to be made about missing data. The items of primary interest to the users are not explicitly recorded but must be inferred from the raw data. This inference process is computationally expensive and so it is often necessary to materialise views. Extending PFL to support ECA rules was an obvious solution to the problems faced by this application, and the extended PFL now allows the flexible handling of constraint violations, default rules and view materialisation.

To our knowledge, this is the first time that an active component has been added to a functional DBPL, and it has been achieved without compromising either the confluence of expression evaluation or the integrity of the type system. Provision of the active component required three extensions to PFL: transactions, ECA-rules, and the casting of a user-defined function as a primitive event.

This paper provides a formal specification of the active component of PFL using PFL itself. It extends our work of [23] by: considering the expressiveness and optimisation of event specification; addressing abort handling; giving a detailed description of how a function can be cast as an event; discussing implementation issues; and comparing our approach with other systems. The paper is structured as follows. In Section 2 we describe PFL, including its support for transactions and ECA-rules, we compare the expressiveness of event specification with that of SAMOS [9, 10], and we describe how event detection can be optimised. In Section 3 we specify the semantics of transaction execution. Abort handling complicates these semantics so we postpone this issue to Section 4. The invocation of any function can be designated as an event as we show in Section 5. In Section 6 we discuss our implementation of PFL's active component. In Section 7 we compare our approach with related work. Finally, in Section 8 we give our conclusions, summarise our contributions, and identify directions of further work.

2. Preliminaries

PFL is a polymorphic, statically typed, functional DBPL. It is *lazy* in that arguments to functions are evaluated only if needed to return a result. It is *higher-order* in that functions can take functions as arguments and return them as results. It is computationally complete and also update-complete, in that any computable transformation over relations can be defined [27].

PFL supports commands to declare new types and type synonyms (as described in Section 2.1), define an equation for a function (Section 2.2), declare a new relation (Section 2.3), evaluate a transaction (Section 2.4), define an ECA-rule (Section 2.5), evaluate a query, and cast an function as a primitive event (Section 5). The new features required to support ECA-rules are those described in Sections 2.4, 2.5, and 5. 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 $e_1, \ldots, e_n$ all have type t.

2.1. Types

PFL provides a set of built-in types and facilities to declare new types and their constructors. The built-in types include `Num` and `Chr`, respectively populated by numbers and characters, and 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, $(t_1, \ldots, t_n)$ consists of n-tuples of the form $(e_1, \ldots, e_n)$ where each $e_i :: t_i$, and $t_1 \to t_2$ 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 $e_2 :: [t]$. Syntactic sugar is provided for lists in that $(:)$ can be written infix, in which case it is unbracketed, and $[e_1, e_2, \ldots, e_n]$ is synonymous with $e_1 : [e_2, \ldots, e_n]$. Also, $\to$ is right-associative so that $t_1 \to t_2 \to t_3$ and $t_1 \to (t_2 \to t_3)$ are synonymous.

New types and type synonyms can be declared. For example, the ECA system defined in Section 3 requires two types, `SMode` (scheduling mode) and `AMode` (abort mode), and two synonyms, `TranId` (transaction identifier) and `EventId` (event identifier) for the type `Num`. The symbols `::=` and `==` introduce new types and type synonyms respectively:

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

We assume one more type synonym in that `Str == [Chr]`. We also support the usual syntax for strings in that, for example, `"hi"` is equivalent to `['h','i']`.

2.2. Expressions and functions

An *expression* can be a variable, an operator, a constructor, an application of one expression to another, a lambda abstraction of the form $\lambda p.e$ where e is an expression and p is a pattern (a *pattern* is a variable or an n-ary constructor applied to n patterns), a "let" expression, or a list comprehension. Variables start with a lower-case letter and constructors with an upper-case letter. Application is left-associative and has higher precedence than any operator. An occurrence of a variable x in an expression e is *bound* if it occurs in a sub-expression of e of the form $\lambda p.e'$ and x also occurs in p; otherwise it is *free*. $FV(e)$ denotes the set of variables with at least one free occurrence in e.

Given variables $x_1, \ldots, x_n$ and expressions $e_1, \ldots, e_n$, a "let" expression introduces local definitions for x_1 to x_n:

$$let\ x_1\ ==\ e_1;\ \ldots;\ x_n\ ==\ e_n\ in\ e$$

and is semantically equivalent to $(\lambda x_1. \ldots (\lambda x_n.e)\ e_n \ldots)\ e_1$.

The built-in functions include the usual arithmetic and relational operators. New functions can be defined by means of equations and we give some 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 definition although the user may optionally declare it:

```
append [] ys        == ys;
append (x:xs) ys    == x:(append xs ys)

flatmap f []        == []
flatmap f (x:xs)    == (f x) ++ (flatmap f xs)

if True x y         == x;
if False x y        == y;

case ((b,v):xs)     == if b v (case xs);

fold f x []         == x;
fold f x (y:ys)     == fold f (f x y) ys;
```

Thus, `append` concatenates two lists, and for convenience can be written infix as `++`. `flatmap` applies a list-valued function `f` to each element of a list and concatenates the resulting lists. `if` is the usual conditional operator. `case` when given a list `[(b1,v1),...,(bn,vn)]` returns the first `vi` for which `bi` is `True`. Given a binary operator `op`, expression `end` and list of expressions `[e1,...,en]`,

```
fold op end [e1,...,en]
```

is (treating `op` as infix) equivalent to

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

Multi-equation definitions are translated into a single expression for storage in the database and evaluation — [18] gives full details of how this can be done.

List comprehensions take the form $[e \mid q_1;\ \ldots;\ q_n]$. Each q_i is either a *filter* or a *generator*. A filter is a predicate which must be satisfied and a generator takes the form $p \leftarrow l$, which can be read as "the pattern p takes each value in the list l in turn". For example:

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

returns the elements of the cartesian product of `xs` and `ys` for which `pred` is true. List comprehensions add no expressiveness to our language and are simply syntactic sugar for successive applications of the functions `flatmap` and `if`:

$$\begin{array}{lcl} [e \mid p \leftarrow xs; Q] & \equiv & flatmap\ (\lambda p.[e \mid Q])\ xs \\ [e \mid pred; Q] & \equiv & if\ pred\ [e \mid Q]\ [\,] \\ [e \mid\ ] & \equiv & [e] \end{array}$$

where $\equiv$ means "is syntactically equivalent to". Thus the above example is equivalent to `flatmap (\x.flatmap (\y.if pred [(x,y)] []) ys) xs`, where `\x` is PFL's concrete syntax for λx.

2.3. Bulk data

Bulk data are stored in relations. The extent of a relation is a list of values of the same type. If a relation contains tuples of type t then its extent is of type $[t]$. A relation r which contains elements of type t is declared by a command of the form

$$\_rel\ r\ ::\ [t];$$

The extent of r is initially the empty list and can be updated by the built-in functions `inc` and `exc`. These take as arguments the name of a relation, r, and a value, e. `inc` adds e to the head of r, provided e is not already present there, and returns r. `exc` removes e from the extent of r and returns r. The name of a relation and its extent are distinguished by prefixing the name with "`&`". In the sequel we use the term "relation" to mean the relation's extent.

Although `inc` and `exc` update relations by side-effect, PFL's type checker will reject any query which would compromise the confluence of the language ([27] gives details of how this is achieved). For example if `r::[Num]` is initially empty, then `(r, inc 1 &r)` evaluates either to `([1],&r)` or to `([],&r)` depending on which component of the tuple is evaluated first. Thus the expression is not confluent and would be rejected by the type checker.

2.4. Transactions

A transaction is a sequence of expressions:

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

where *tranStart* and *tranEnd* are built-in functions and $e_1, \ldots, e_n$ are arbitrary expressions. These $n+2$ expressions 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. This relation is cleared immediately before the evaluation of each expression. The invocation of `inc`, `exc`, `tranStart` and `tranEnd` constitute primitive events, as also does the invocation of any function so declared by the user (see Section 5). 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 (a database commit in PFL is an explicit top-level command).

An ECA-rule cannot refer to the `delta` and `history` relations directly because they would be perceived as being type-incorrect. Instead, for each class of primitive event an individual history and delta relation are provided which are type-correct views of the overall `history` and `delta` relations, as described in Section 5 below. For the moment we need only know that for each relation `r::[t]`, the relations `rIncH`, `rIncD`, `rExcH`, `rExcD::[(EventId,TranId,t)]` respectively record the cumulative and delta inclusions and the cumulative and delta exclusions with respect to `r`. Similarly, `tranStartH`, `tranStartD`, `tranEndH`, `tranEndD ::[(EventId,TranId)]` record the cumulative and delta transaction start and end events.

Thus, from the user's perspective, each class of primitive event is associated with two relations: one containing a tuple for each event occurrence since the database was last committed and one containing a tuple for each event occurrence during the evaluation of the current expression. Since the transaction identifier is an attribute of each history relation, the tuples pertaining to a particular transaction are readily accessible.

2.5. ECA-rules

The ECA-rules are fired after the evaluation of each transaction expression. They are defined by a command of the form:

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

where i is a unique number which determines the priority of the ECA-rule.

The occurrence of events, both primitive and composite, is detected by the event query component of an ECA-rule, $events :: [t]$ for some t. $events$ must not update the database. It can interrogate the history and delta relations associated with each primitive event and its evaluation will yield 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. It is applied to each ev_j and only those for which $condition\ ev_j$ is $True$ are retained. Thus, the ECA-rule specifies the following sequence of events: $[e \mid e \leftarrow events;\ condition\ e]$. We term this expression the *parameter list* and a rule is triggered when its parameter list is non-empty.

Each $action_i$ is a function to be applied to the parameter list; it can, and usually will, update the database. Thus, overall, the user can regard the ECA-rule as specifying the following sequence of reactions provided the parameter list *evs* is non-empty (otherwise the rule has no effect):

$$\begin{aligned} let\ evs\ &== [e \mid e \leftarrow events;\ condition\ e]\ \ in \\ &\quad tranStart;\ action_1\ evs;\ \ldots;\ action_m\ evs;\ tranEnd; \end{aligned}$$

sm is an expression of type `SMode`. The above reaction transaction is run immediately before the current transaction is resumed if *sm* is `Imm`, and is deferred until the completion of the current transaction if *sm* is `Def`.

am is an expression of type `AMode` representing the abort dependency between the original transaction, $tran_o$, and the reaction transaction, $tran_r$, as follows:

`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::[(Acc,GridRef)]` which relates road traffic accidents to the grid references at which they occurred, and `blackSpots, mWayBlackSpots::[GridRef]` which record the grid references of all accident black spots and motorway black spots respectively (a "black spot" is a grid reference at which more than 5 accidents have occurred). We can define ECA-rules which (i) enforce the constraint that accidents must occur on roads, and (ii) materialise `blackSpots` and `mWayBlackSpots` from `accSite` as follows, where `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          == [(acc,gr) | (eId,tId,(acc,gr)) <- accSiteIncD];
c1 (acc,gr) == roadClassAt gr = None;
a1 x        == error "accident not located on road";

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

The `ChildParent` abort mode of the first ECA-rule causes any transaction violating the constraint to abort, whilst the `Def` scheduling mode of the second ECA-rule causes re-materialisation of `blackSpots` and `mWayBlackSpots` at the end of transactions which update `accSite`. We note that in our approach no implicit event consumption occurs; instead, all primitive events that have occurred since the last commit are accessible via the history relations.

2.6. Expressiveness of event specification

In this section we compare the expressiveness of events in PFL with those of SAMOS [9], which we take as representative of ECA systems with a rich set of event specification constructs.

Primitive events in SAMOS are recorded with four *event parameters*: the transaction id, the object triggering the event, the time of the event occurrence and the originating user. The last two of these are not recorded in PFL but could easily be added. SAMOS has four classes of primitive event [10]:

Time events. We support neither time events nor time intervals, although a rudimentary form of time interval could be provided by recording a time stamp for an event occurrence in addition to the event identifier.

Method events. The execution of a method defines two events in SAMOS, which occur immediately before and immediately after the method is executed. Method events correspond to functions cast as primitive events in our approach and occur after the function has been evaluated. Extending our approach to support "before" events would give no extra expressiveness since ECA rules are executed only after the evaluation of the current expression has terminated.

Transaction events. SAMOS recognises transaction start, end and abort as events. The first two are also events in PFL, but not the last. However, it would be simple to extend the abort handler to log the abort of a transaction.

Abstract events. SAMOS allows events to be raised by user programs. In our approach this functionality is obtained by casting a function as a primitive event.

In common with SAMOS, we support sequencing, disjunction and conjunction of events. Given n events, logged within history relations e_1H, e_2H, ..., e_nH, a sequence of these n events is detected by the following event query, assuming that the sequence is regarded as being concurrent with the last of its constituent events:

$$\begin{aligned}[(eId_n, tId_n, expr) \mid\ & (eId_1, tId_1, tuple_1) \leftarrow e_1H;\ filter_1;\\ & (eId_2, tId_2, tuple_2) \leftarrow e_2H;\ filter_2; eId_1 < eId_2;\\ & \ldots\\ & (eId_n, tId_n, tuple_n) \leftarrow e_nH;\ filter_n; eId_{n-1} < eId_n];\end{aligned}$$

Here each $filter_i$ is a predicate that can be used to match the components of $tuple_1$, ..., $tuple_n$, analogously to the use in SAMOS of `same` to match event parameters. The disjunction of n events, specified by the (list-valued) event queries eq_1, eq_2, ... eq_n is specified by their concatenation:

$$eq_1\ {+}{+}\ eq_2\ {+}{+}\ \ldots\ {+}{+}\ eq_n$$

The conjunction of events can be expressed as a disjunction of sequences e.g the conjunction of events e_1 and e_2 is equivalent to the disjunction of the sequences

$e_1; e_2$ and $e_2; e_1$. Finally, event queries representing sequences, disjunctions and conjunctions can be nested to arbitrary levels of nesting.

2.7. Optimisation of event queries

Event queries can be optimised by (i) reducing duplicate work by replacing history relations by delta relations; (ii) optimising the event query using well-known query transformation techniques such as splitting up compound selections, performing selections as early as possible, and combining cascades of selections; and (iii) optimising the event query and the condition query in tandem, using similar transformation techniques.

Since there is no explicit event consumption in our system, optimisation (i) cannot in general be performed automatically and is left to the programmer. In contrast, (ii) and (iii) can be automated,and before describing them in Section 2.7.2 we first give some necessary information on the optimisation of functional DBPLs.

2.7.1. Optimisation of functional DBPLs

We investigated many useful equivalences for optimising functional DBPLs in [20]. A representative sample of the equivalences over comprehensions which we require here govern splitting and merging compound filters, commuting two filters, commuting a filter and a generator, commuting two generators, and moving a filter into a nested comprehension, respectively (where $eq/3$ requires that $FV(p) \cap FV(f) = \emptyset$ and $eq/4$ that $FV(p) \cap FV(l') = FV(p') \cap FV(l) = \emptyset$):

$$\begin{array}{lll}
eq/1 & [e \mid Q;\ f_1;\ f_2;\ Q'] & = [e \mid Q;\ f_1 \ \&\ f_2;\ Q'] \\
eq/2 & [e \mid Q;\ f_1;\ f_2;\ Q'] & = [e \mid Q;\ f_2;\ f_1;\ Q'] \\
eq/3 & [e \mid Q;\ f;\ p \leftarrow l;\ Q'] & = [e \mid Q;\ p \leftarrow l;\ f;\ Q'] \\
eq/4 & [e \mid Q;\ p \leftarrow l;\ p' \leftarrow l';\ Q'] & = [e \mid Q;\ p' \leftarrow l';\ p \leftarrow l;\ Q'] \\
eq/5 & [e \mid p \leftarrow [p \mid Q];\ f;\ Q'] & = [e \mid p \leftarrow [p \mid Q;\ f];\ Q']
\end{array}$$

In [21] we also discuss a class of functions called *selectors* that optimise retrieval from relations by taking advantage of any physical-level indexes available. In particular, associated with each relation `r` is a set of system-defined selectors $\mathtt{r}p$ where p is a variable-free pattern all of whose constants are either `f` or `b` (denoting "free" and "bound" respectively). A selector $\mathtt{r}p$ takes a number of boolean-valued functions as arguments, depending on the number of occurrences of `b` in p, and these functions represent conditions to be satisfied by the values retrieved from the relation. To illustrate, the possible selectors for the relation `accSite::[(Acc,GridRef)]` are:

```
accSitef     :: [(Acc,GridRef)]
accSiteb     :: ((Acc,GridRef) -> Bool) -> [(Acc,GridRef)]
accSite(f,f) :: [(Acc,GridRef)]
```

```
accSite(f,b) :: (GridRef -> Bool) -> [(Acc,GridRef)]
accSite(b,f) :: (Acc -> Bool) -> [(Acc,GridRef)]
accSite(b,b) :: (Acc -> Bool) -> (GridRef -> Bool) -> [(Acc,GridRef)]
```

For example, the query `[y | (x,y) <- accSite; x = Acc1]` is equivalent to `[y | (x,y) <- accSite(b,f) (\x.x = Acc1)]`.

2.7.2. Optimisation of event queries

To illustrate the optimisation of event queries, consider the event query which detects the insertion of a road condition `Wet` for an accident with weather severity `Fine`:

```
[acc1 | (acc1,sev)  <- accWeather; sev = Fine;
        (e,t,(acc2,road)) <- accRoadIncD; road = Wet; acc1 = acc2];
```

Equivalences $eq/2$, $eq/3$ and $eq/4$ can be used to move the generator and filter involving `accRoadIncD` in front of the generator and filter involving `accWeather` (this is desirable since `accRoadIncD` is likely to contain fewer tuples than `accWeather`):

```
[acc1 | (e,t,(acc2,road)) <- accRoadIncD; road = Wet;
        (acc1,sev)  <- accWeather; sev = Fine; acc1 = acc2];
```

Information in the filter expressions can be used to restrict the tuples retrieved from these relations. Thus, using selectors, we obtain

```
[acc1 | (e,t,(acc2,road)) <- accRoadIncD(f,f,(f,b)) (\x.x = Wet);
        (acc1,sev) <- accWeather(b,b) (\x.x = acc2) (\x.x = Fine)];
```

To illustrate the optimisation of an event and condition query in tandem, we recall from Section 2.5 that an ECA-rule specifies the list of events $[e \mid e \leftarrow events;\ condition\ e]$. Since *events* and *condition* do not update the database, splitting the event and condition into two expressions is really just a syntactic convenience and this list comprehension can be optimised using the same techniques as for optimising *events* on its own. For example, consider ECA-rule 1 of Section 2.5 which specifies the following list of events:

```
[(acc,gr) | (acc,gr) <- [(acc,gr) | (eId,tId,(acc,gr))<-accSiteIncD];
            roadClassAt gr = None]
```

Using $eq/5$ to move the filter inside the nested comprehension gives

```
[(acc,gr) | (acc,gr) <- [(acc,gr) | (eId,tId,(acc,gr))<-accSiteIncD;
                                    roadClassAt gr = None]]
```

Using the selector optimisation, the final list of events is given by

```
[(acc,gr) | (acc,gr) <- [(acc,gr) | (eId,tId,(acc,gr)) <-
            accSiteIncD(f,f,(f,b)) (\gr.roadClassAt gr = None]]
```

3. Transaction Execution Semantics

We specify the semantics of transaction execution using PFL itself as the meta language, obtaining a formal declarative semantics which not only forms the basis of our implementation of the ECA system but is also immediately accessible to the PFL user. In this semantics, expressions are represented as values of type `Expr`:

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

The argument associated with the constructors `C`, `N`, `Con` and `Rel` represents a character, number, constructor, or relation name, respectively. The constructor `Error` is discussed in Section 3.2 below. The ECA system does not need to distinguish between function names and other variables (except for relation names) so `Var` is used for both of these. For example, the expression `length []` is represented as

```
Ap (Var "length") (Con "[ ]")
```

3.1. Data structures used by the ECA system

The semantics requires the following type synonyms and functions:

```
Tran        ==  (TranId,[Expr]);
Schedule    ==  [Tran];
DBState     ==  [(Str,[Expr])];
ECAState    ==  (EventId,TranId,AbortGraph,[ECARule],History);
AbortGraph  ==  [(AMode,TranId,TranId)];
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` and `TranId` were defined in Section 2.1. Event ids and transaction ids are totally ordered and `nextEv` (`nextTran`), when applied to an event id (transaction id) returns the next id according to the total ordering.

`Tran` is a pair consisting of a transaction id and the list of expressions comprising the transaction.

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

`DBState` represents the database state. It consists of a list of pairs, the first component of each pair being the name of a function and the second being its definition. The definition of an equationally-defined function contains a single `Expr` while that of a relation contains 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])`.

`getRel`, `incRel` and `clearDeltas` interrogate and update a `DBState`: `getRel db r` returns the extent of `r` in `db`, `incRel db r e` adds `e` to the head of the extent of `r` provided it is not already present and returns the new database state, and `clearDeltas db` sets to the empty list the extent of the individual delta relations and the `delta` relation, and returns the new database state.

`ECAState` represents the current state of the ECA system. It consists of the identifier to be assigned to the "next" event, the identifier to be assigned to the "next" transaction, the abort graph (a list of triples comprising two transaction identifiers and the abort dependency between them), the list of ECA-rules, and the `history` relation. The ECA-rules are ordered according to their user-specified rule identifier. There is no fundamental reason for holding the `history` relation within the ECA state rather than the database state; it merely makes for a simpler specification of the semantics.

`ECARule` represents an ECA-rule, consisting of 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 requires a function `eval:: Expr->DBState ->(Expr,DBState)` which takes an expression `e` and a database state `db`, and evaluates `e` while logging all primitive event occurrences in the `delta` relation within `db`. If a run-time error arises (e.g. an attempt to divide by zero, or an invocation of the `error` function) `eval` returns the pair `(Error,db)`. Otherwise, it returns the value of `e` and the updated database state. Apart from supporting retrieval from, and update of, relations and logging of primitive events, `eval` is much like the evaluator of a conventional functional programming language [18].

The top-level function of our specification is `execSched`, which takes a database state, ECA state and schedule, repeatedly calls `execExp` (see below) to process the transaction at the head of the schedule until the schedule is empty, and returns the updated database and ECA states:

```
execSched (db,eca,[])            == (db,eca);
execSched (db,eca,(tran:trans)) ==
    execSched (execExp tran (db,eca,trans));
```

`execExp` processes each expression of the current transaction in turn. For each expression, it clears the delta relations within the current database state and then calls `eval` to evaluate the expression with respect to the updated database state. If the evaluation terminates normally, `ecaControl` (see below) updates the history relation, undertakes event detection and schedules reactions; otherwise, `abortHandle` (see Section 4) aborts the current transaction:

```
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));
```

`ecaControl` takes the current transaction id, database state, ECA state and schedule. It repeatedly calls `logEvent` (see below) to update the `history` relation and the individual history and delta relations associated with each primitive event in `delta`. It then calls `detectAndSched` (see below) to undertake event detection and schedule any reactions, and returns the updated database state, ECA state and 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);
```

`detectAndSched` calls `schedRule` to process each ECA-rule in turn, and returns the updated ECA state and 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,rules,h),imm++def);
```

`schedRule` lies at the heart of the ECA system. It considers a particular ECA-rule, evaluates its parameter list, schedules the reactions and updates the abort graph. In addition to the current transaction id, database state, and ECA-rule, `schedRule` takes a 4-tuple comprising the lists of immediate and deferred reactions so far accumulated (`imm` and `def` below), the next unassigned transaction id and the abort graph. If the ECA-rule's parameter list evaluates to empty, `schedRule` returns this 4-tuple unchanged. Otherwise, the next unassigned transaction id is incremented, the abort graph is updated and the reaction transaction is appended to `imm` or `def` according to the rule's scheduling mode:

```
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 = Con "[]", (imm,def,tId2,graph)),
              (mode = Imm,    (imm++[tran],def,nextId,g2)),
              (mode = Def,    (imm,def++[tran],nextId,g2))];
```

In the above definition, `parmList` is a function defined as

```
parmList events cond == [e | e <- events; cond e];
```

and stored within the database, `db`, in the appropriate form.

We now return to `logEvent`. This prefixes to the `history` relation a triple consisting of the "next" event id, the current transaction id and the current event, updating the "next" event id in the ECA state accordingly. It then updates the individual history and delta relations by calling `incHistAndDelta`, defined below, and returns the updated database and ECA states.

```
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);
```

`incHistAndDelta` calls `unpack` (defined below) to unpack the event expression into the name of the function whose evaluation constituted the primitive event and the list of the function's arguments. This information is then used to construct the name of the corresponding history and delta relations. The expression to be added to the extents of these relations represents a tuple consisting of the event id, the transaction id, and the function's arguments. The function `makeTuple :: [Expr] -> Expr` (not specified here) constructs this expression, taking a list of n expressions and returning a single expression that represents an n-tuple:

```
incHistAndDelta db (eid,tid,exp) ==
    let  (fun,args) == unpack (exp,[])
    in   case [ (fun = "inc", relEvent db args "Inc" eid tid),
                (fun = "exc", relEvent db args "Exc" eid tid),
                (True,  funEvent db fun (eid,tid,args))];

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

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

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

3.3. Summary and illustration

Our execution semantics depend primarily on our notions of schedules and transactions. A transaction is a sequence of expressions and a schedule is a sequence of transactions. Both sequences contain elements in the order that they are to be executed. A schedule may have partially executed transactions and these contain only the expressions still to be evaluated. We have a simple nested transaction model where the schedule initially contains a single user-specified transaction. All other transactions are reactions to events, either within the top-most user-defined transaction or within some nested reaction transaction. When an ECA-rule fires, the associated reaction transaction is added to the schedule in a location defined by the rule's scheduling mode and priority.

Event detection merely comprises evaluating parameter lists and firing each ECA-rule whose parameter list is non-empty. Since events and conditions do not update the database, all parameter lists are evaluated against the same database state.

An ECA-rule can fire only once in a detection/scheduling phase. However, it can fire in *any* detection/scheduling phase and event detection and scheduling take place after the evaluation of every expression in every transaction. Thus, at a given instant, the schedule can contain several reaction transactions from the same ECA-rule. These reaction transactions need not be identical since the parameter-lists their execution depends on reflect the database state at the different times the event was detected.

To illustrate, we describe the effect on the schedule of the execution of a particular transaction. Consider the relations `r1,r2::[Num]` and the ECA-rule defined by

```
e3          == [t | (eId,tId,t) <- r1IncD];
a3 xs       == incList xs &r2;
_ecaRule 3  == (e3,(\x.True),[a3],Def,Mutual);
```

This rule adds to `r2` any tuple added to `r1`. We assume it to be the only ECA-rule and we trace the schedule through the execution of the transaction

`tranStart; incList [1,2,3] &r1; inc 4 &r1; tranEnd;`

The schedule initially contains this single transaction. Assuming the transaction's identifier is 1, the schedule first passed as an argument to `execSched` is

`[ (1, [Var "tranStart",` e_1`,` e_2`, Var "tranEnd"]) ]`

where e_1 and e_2 are, respectively, the expressions `(incList [1,2,3] &r1)` and `(inc 4 &r1)` represented as `Exprs`.

`execSched` calls `execExp` to evaluate the first expression, `Var "tranStart"`. Since ECA-rule 3 is not triggered by this evaluation, no reaction transactions are scheduled and `execSched` calls itself recursively with the schedule

`[ (1, [`e_1`,` e_2`, Var "tranEnd"]) ]`

This time, when invoked, `execExp` evaluates e_1. The rule's parameter list now contains the three numbers just added to `r1`, and the rule is triggered. The rule's `SMode` is `Def`, so `schedRule` adds the reaction transaction to the end of the schedule. The schedule returned by `execExp` and with which `execSched` again calls itself is

`[ (1, [`e_2`, Var "tranEnd"]), (2, [Var "tranStart",`e_3`,Var "tranEnd"]) ]`

where e_3 is `(a3 [1,2,3])` represented as an `Expr`.

The third time `execExp` is called, it evaluates e_2 and the parameter list contains the single number added to `r1`. The rule fires again and another reaction transaction is appended to the schedule for the next invocation of `execSched`:

```
[ (1, [Var "tranEnd"]), (2, [Var "tranStart",e3,Var "tranEnd"]),
            (3, [Var "tranStart",e4,Var "tranEnd"]) ]
```

where e_4 is `(a3 [4])` represented as an `Expr`.

We note that transactions 2 and 3 both arise from the same rule, but differ in that e_3 and e_4 are responses to the evaluation of e_1 and e_2, respectively. This difference is embodied in the arguments to which the action `a3` is applied, `[1,2,3]` and `[4]`, which are the values of the parameter list at the times at which the rule fired.

4. Abort Handling

When an expression evaluates to `Error` the transaction containing it is aborted. This may necessitate other transactions to be aborted as well. These transactions may be interspersed with transactions, and contain nested transactions, that should not be aborted. Due to this interleaving and nesting of normal and aborted transactions and the consequent interleaving of wanted and unwanted primitive event-occurrences, an aborted transaction is not rolled-back. Instead, it is *compensated* by compensating each of its constituent primitive events in the reverse order of their occurrence. A single transaction compensates all the aborted transactions. Rather than log this compensating transaction, we remove the events being compensated from the overall and individual history relations.

`abortHandle`, defined below, is invoked when an expression reduces to `Error` (see `execExp` in Section 3.2) and processes the abort handling in five stages. Firstly, it calls `transClosure`, not given below, to calculate the transitive closure of the abort dependencies, starting from the original transaction id. Secondly, it determines the events of aborted transactions which have already occurred, and which must therefore be compensated: these primitive events are retrieved from the `history` relation and for each a compensating sequence of "inverse" events is determined by `invEvent`. Thirdly, the aborted transactions which have either not started or not completed execution are removed from the schedule. Fourthly, a new history and database are constructed using `rollbackHist`, not given below, which removes from the appropriate individual history relations and the overall `history` relation any primitive event that occurred during an aborted transaction. Finally, `execCompEvs` is called to execute the compensating transaction:

```
abortHandle t (db1,eca1,sched1) ==
   let (eid,tid,graph,rules,history)  ==  eca1;
       aborted == transClosure t graph;
       events  == [exp | (e,t,exp) <- history; member aborted t];
       compEvs == [y | x <- events; y <- invEvent x];
       sched2  == [(t,es) | (t,es)<-sched1; not (member aborted t)];
       (h2,db2)== rollbackHist history db1 aborted;
       db3     == execCompEvs compEvs db2;
       eca2    == (eid,tid,graph,rules,h2)
   in  (db3,eca2,sched2);
```

```
invEvent (Ap (Ap (Var "inc") r) val) == [Ap (Ap (Var "exc") r) val];
invEvent (Ap (Ap (Var "exc") r) val) == [Ap (Ap (Var "inc") r) val];
invEvent ev                          == [];

execCompEvs [] db    == db;
execCompEvs (h:t) db == let (e,d2) == eval h db in execCompEvs t d2;
```

5. Turning a Function into a Primitive Event

Casting a function as a primitive event is analogous to designating method invocation as an event in an active object-oriented system. Functions cast as primitive events cannot be called by the event or condition queries of any ECA-rule since these may not alter the database. The evaluator (`eval` in Section 3.2) has access to the list of the functions cast as primitive events and logs their invocation.

From the user's perspective, each time that such a function is evaluated a tuple is added to the associated history and delta relations. This tuple contains the event and transaction identifiers and the arguments to which the function was applied. The history and delta relations are denoted by the function name appended with "H" and "D", respectively. Thus, when a function $f :: t_1 \to t_2 \to \ldots \to t_n \to t$ is cast as a primitive event, the following relations are created:

$$fH,\ fD\ \ ::\ [(EventId,\ TranId,\ t_1, \ldots, t_n)]$$

To cast f as a primitive event the following command is used:

$$\_makeEvent\ f\ v_1\ \ldots\ v_n\ \ ==\ [action_1, \ldots, action_m];$$

where $v_1, \ldots, v_n$ are variables and $action_1$, ..., $action_m$ are expressions to be executed to compensate an aborted invocation of f. For example, the declaration of `foo::Num->Num` as a primitive event and `goo` as its compensating event is given by

```
_makeEvent foo x == [goo x];
```

and has three effects: an equation is added to the function `invEvent` of Section 4:

```
invEvent (Ap (Var "foo") x) == [Ap (Var "goo") x];
```

the relations `fooH,fooD::[(EventId,TranId,Num)]` are declared; and `eval` logs all invocations of `foo` thereafter.

Logging a function's invocation requires its arguments to be evaluated, and hence casting a function as a primitive event makes it strict in all its arguments. (A function is *strict* in an argument if that argument needs to be evaluated in order for the function to return its result.) We note that any primitive events that occur during the evaluation of a function will automatically be compensated in the event of an abort. Consequently, the evaluation of the inverse function (`goo` in the above example) is a "do instead" activity.

6. Implementation

We have implemented the semantics described above, with one minor departure: primitive events are logged directly on the individual history and delta relations. Thus, the overall `delta` is not needed, and the overall `history`, which is needed only by the abort handler, can be derived from the individual history relations.

PFL's active component incurs performance penalties in two places, logging and detection of events. Although our implementation logs all events and evaluates all event queries, both logging and detection are amenable to optimisation.

In order to discuss the optimisation of event logging, we define the database's *call graph*: every function in the database (including relations) is represented by a node in the graph and if a function f_2 occurs on the right-hand side of an equation that defines a function f_1 then there is an arc in the call graph from f_1 to f_2.

For each function f that has been cast as a primitive event, the relations fD and fH are maintained. It is clear that if both fD and fH are isolated nodes in the call graph then they can never be evaluated, and thus need not be maintained. Similarly, inclusions into (exclusions from) a relation r need not be logged if $rIncD$ and $rIncH$ ($rExcD$ and $rExcH$) are isolated nodes in the call graph.

Restricting the ECA-rules to be checked during event detection is not so straightforward in general. However, for event queries expressed as list comprehensions (as all our examples have been) we observe that a list comprehension will return a non-empty list only if all its generators iterate over non-empty lists. Incremental view maintenance techniques can be used to statically determine event queries that will evaluate to an empty list because they depend on an empty delta relation. (A function f_1 *depends on* a function f_2 if f_2 is reachable from f_1 in the call graph.) For reasons of space, we refer the reader to [22] for details.

A further optimisation would be to derive delta relations from history relations rather than to log each primitive event twice. To achieve this, we need simply to number the expressions within a transaction; then adding this number to the tuples stored in a history relation would afford efficient retrieval of a delta.

7. Comparison with Related Work

In this section we first consider the impact that extending PFL with an active component has on confluence. We then discuss related work, addressing in particular three issues: scheduling, including event consumption and coupling modes; event specification and detection; and specification of the semantics of ECA-systems.

7.1. Confluence

Transactions and ECA-rules are not "first-class objects" in PFL: functions can neither take them as arguments nor return them as results, and so transactions and ECA-rules cannot be embedded within expressions. Transactions are sequences of expressions and processing a transaction entails evaluating each expression in turn. ECA-rules are fired only upon the termination of the evaluation of each expression, and so the evaluation of each expression is confluent.

The database state resulting from the evaluation of an expression followed by the execution of the ECA-rules that it triggers *is* in general dependent on the order in which sub-expressions of the expression are evaluated. For example, given two relations `r1,r2::[Num]`, the evaluation of the tuple `(inc 1 &r1, inc 1 &r2)` is confluent. However, an 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. In such cases of interleaved updates on more than one relation, PFL provides a view construct which allows the programmer to group relations into a single, higher-level identifier. Update requests on the individual relations are then invoked via the view and the type system (statically) guarantees a single possible execution order for updates of the view. In this way the final database state is uniquely determined.

7.2. Scheduling

A corollary to our scheduling strategy (summarised in Section 3.3) is that event consumption is left for the user to specify within event and condition queries and, hence, can be tailored to each ECA-rule. Sentinel provides four event consumption behaviours or *Parameter Contexts* [5], but these are specified at the application level. Ode [11] allows two rule-specific options: either the rule fires every time the event is raised or fires once and then is deactivated until it is explicitly reactivated. In Starburst [28], to avoid executing an action twice with the same parameters, the database state against which the condition is tested depends on the database state when the action last executed, rather than when the event was detected or the current database state. Duplication of event processing can also be avoided by turning rules on and off dynamically. We do not provide this facility directly, but can simulate it by checking the value of a switch in an event query and by turning the switch on or off in an action. A-RDL [25] adopts a similar approach to deactivation by attaching global variables to rules. In PFL the rule is still active until the reaction transaction executes, and deactivating a deferred rule requires an extra immediate rule, with identical event and condition, to turn the switch off.

In general, it is the coupling mode that encapsulates the relationship between a parent and reaction transaction. We decompose this relationship into scheduling mode and abort dependency. While the combinations of our `SMode` and `AMode` span the coupling modes listed in [14], (*Immediate*, *Deferred*, *Detached Dependent* and *Detached Independent*,) our `ChildParent` abort mode is not reflected by any coupling mode, although in practice it may be desirable. The distinction between scheduling and abort handling can also simplify implementation in that extra scheduling modes (for example one that defers a reaction to the end of the current, as opposed to the top-level, transaction) can be added without affecting the abort handler.

Coupling modes assume concurrent transactions and are often defined for parallel and distributed systems. Applying them to a single-user sequential system such as PFL can render them misleading or irrelevant. For example, we can obtain the *Detached Dependent* and *Detached Independent* coupling modes by using the `AModes` `ParentChild` and `Independent` respectively and the `SMode` `Def`. However, a detached transaction in a multi-user system also determines the visibility of updates, whereas all updates are immediately visible in PFL.

7.3. Event specification and detection

In PFL, a primitive event occurrence is simply a value in the appropriate history and delta relations. The specification and detection of both primitive and composite events merely consist of defining and evaluating queries over these relations.

A history of primitive events can be used to define the semantics of event operators [12, 5]. However, such histories are not necessary for event detection. Alternatives are the Petri nets, finite automata and event graphs of SAMOS, Ode and Sentinel, respectively. Our events are specified in PFL which is computationally complete. In contrast, event expressions in Ode are reducible to finite automata and consequently are computationally incomplete. Similarly, events in SAMOS are implemented as Petri nets which are also not computationally complete, although adding inhibitor arcs or prioritisation of transitions make them so [17]. Clearly, the expressiveness of an event language depends on the primitive events supported as well as on the power of operations over them. The restrictions on the events we support arise from our execution model, which was chosen to retain confluence of expression evaluation. Thus, we support neither a "before" event operator nor primitive events, such as clock and external events, which need to be detected and acted upon during expression evaluation. The richest event specification is found in object-oriented systems such as HiPAC [7], REACH [2] and SAMOS. The goal of such systems seems to be a new programming paradigm, requiring a sophisticated transaction model [3], and akin to reactive programming with persistence and access to real time as a variable. Our aim, in contrast, has been to supply a general-purpose tool for flexible handling of constraint violations, default rules and view materialisation that complements, rather than redefines, the functional paradigm.

Starburst and A-RDL both use delta relations to provide conditions and actions with access to values from earlier database states and hence to details of an event's effect. These systems attach a rule to a particular set of relations (a singleton set in the case of Starburst) and the only possible events are accesses or updates to these relations. A rule has access only to the deltas of its associated relations. In A-RDL deltas summarise the changes over a transaction and in Starburst they reflect the changes since last time the rule fired. PFL actions are bound to values from a previous database state through the parameter lists they are applied to, and which were evaluated at the time of event detection. Since an action is expressed in a language that is both computationally and update complete, it has access to any values obtainable from the current database state as well as to those of its parameter list. Although in PFL deltas are subsets of history relations, past deltas cannot be retrieved. Should this facility be required, we could provide it by numbering the expressions within a transaction and adding this expression number to the tuples stored in the history and delta relations. The tuples of a given delta would then be those with a particular transaction identifier and expression number.

7.4. Specifying the semantics of ECA systems

The formal specification of an active system is highly desirable [8]: the execution semantics are made explicit and the system is amenable to formal analysis and comparison with other systems. The choice of such a specification language, which must span different active systems, is not a simple one [15]. We have used PFL to specify its own active semantics. As a functional language, it is similar to the denotational semantics approach and has similar advantages and disadvantages. Both combine mathematical rigour with notational elegance. They provide high-level, modular, declarative specifications that are succinct and amenable to formal analysis [4, 6, 16]. However, this formalism may not be immediately accessible to every reader and may not be the clearest way to formally specify a given semantics. In this respect, an advantage of our approach is that PFL is immediately accessible to our users and it is well-suited to specifying the semantics of the new active component.

8. Conclusions

We have extended the functional DBPL PFL with an active component and have achieved this without introducing impedance mismatch since no new language need be learned for the specification of events, conditions and actions, and since PFL's computation model and type system remain unchanged. Events and conditions are expressed in a computationally complete language and actions in an update-complete one. The approach is extensible in that the user can specify the invocation of any function to be a primitive event, and also has complete control over event consumption. The active component does not affect the confluence of expression evaluation, permitting ECA rules to be optimised as illustrated in Section 2.7. We have given a formal specification of the active component using PFL itself as the metalanguage. The extended PFL is currently in use for road accident analysis at the Centre for Transport Studies at University College London. The novelty of this paper and of our approach can be summarised as follows:

Firstly, to our knowledge, this is the first time that an active component has been added to a functional DBPL, and this has been achieved without introducing impedance mismatch or compromising the confluence of expression evaluation.

Secondly, the semantics of the active component were specified before it was implemented, giving a sound basis for implementing the system and for explaining its functionality to end-users.

Thirdly, since the semantics were executable, we were able to experiment with the active component and identify the practical implications of design choices before investing effort in implementation. In fact, having finalised the semantics, implementation and testing took only two man months.

Fourthly, defining the specification of the active component in the DBPL itself means that the specification is immediately accessible to those already familiar with the DBPL. This approach is not confined to functional DBPLs — any database language with a sufficiently rich computational model and set of types could be used to specify the semantics of the active component of its own DBMS.

However, the high-level, modular nature of functional languages makes them particularly natural for specification. Moreover, the fact that our active system is specified in a functional language means that it is amenable to a wealth of formal analysis techniques developed for functional programming [4, 6, 16]. Thus, we conjecture that when used in conjunction with a set of ECA-rules, properties of the rule-set, such as *triggering* and *activation dependencies* [1] between rules and the termination of the rule-set, can be formally proved and this is an area of research that we intend to pursue. Other directions of further work are to provide the user with a more sophisticated scheme for prioritising ECA rules and to implement a query optimiser that will undertake the optimisations discussed in Section 2.7.

Acknowledgments

We are grateful to David Sutton for discussions on PFL's linear type system and his implementation of grouping relations into views. We are also grateful to Peter Gray and to the anonymous referees for their useful remarks on this work. The work was supported by EPSRC grant no. GR/J 98134.

References

1. Baralis, E., Ceri, S. and Paraboschi, S. "Improved Rule Analysis by Means of Triggering and Activation Graphs", in [24]. pp 165-181.
2. Branding, H., Buchmann, A., Kudrass, T. and Zimmerman, J. "Rules in an open system: the REACH rule system", in [13]. pp 111-126.
3. Buchmann, A., Özsu, M.T., Hornick, M., Georgakopoulos, D. and Manola, F.A. "A transaction model for active distributed object systems", in Elmagarmid, A.K. (Editor), *Database Transaction Models for Advanced Applications*, Morgan Kaufman, 1992. pp 123-158.
4. Burstall, R. "Proving properties of programs by structural induction", *The Computer Journal*, 12, 1969. pp 41-48.
5. Chakravarthy, S., Krishnaprasad, V., Anwar E. and Kim, S.-.K. "Composite Events for Active Databases: Semantics, Contexts and Detection", *Proc. 20th VLDB Conference*, Santiago, 1994. pp 606-617.
6. Clack, C. and Peyton-Jones, S.L. "Strictness Analysis - A Practical Approach", in Jouannaud, J.P. (Editor) *Functional Programming Languages and Computer Architecture.* Springer-Verlag LNCS 201, 1985. pp 35-49.
7. Dayal, U., Blaustein, B., Buchmann, A.P., Chakravarthy, S. Goldhirsch, D., Hsu, M., Ladin, R., McCarthy, D. and Rosenthal, A. "The HiPAC project: combining active databases and timing constraints", *SIGMOD RECORD*, 17 (1), 1988. pp 51-70.

8. Dittrich, K.R., Gatziu, S. and Geppert, A. "The Active Database Management System Manifesto: A Rulebase of ADBMS Features", in [24]. pp 3-17.
9. Gatziu, S. and Dittrich, K.R. "SAMOS: an Active Object-Oriented Database System", *IEEE Bulletin of the TC on Data Engineering: Special Issue on Active Databases*, 15:(1-4), 1992.
10. Gatziu, S. and Dittrich K.R. "Events in an Active Object-Oriented Database System", in [13]. pp 23-39.
11. Gehani, N.H. and Jagadish, H.V. "Ode as an Active Database: Constraints and Triggers", *Proc. 17th VLDB Conference*, Barcelona, 1991. pp 327-336.
12. Gehani, N.H., Jagadish, H.V. and Schmueli, O. "Composite Event Specification in Active Databases: Model & Implementation", *Proc. 18th VLDB Conference*, Vancouver, 1992. pp 327-338.
13. Paton, N.W. and Williams, M.H. (Editors) *Proc. 1st Int. Workshop on Rules in Database Systems*. Springer-Verlag, 1993.
14. Paton, N.W., Diaz, O., Williams, M.H., Campin, J., Dinn, A. and Jaime, A. "Dimensions of active behaviour", in [13]. pp 40-57.
15. Paton, N.W., Campin, J., Fernandes, A.A.A. and Williams, M.H. "Formal Specification of Active Database Functionality: A Survey", in [24]. pp 21-35.
16. Paulson, L.C. *Logic and computation: Interactive proof with Cambridge LCF*, Cambridge University Press, 1987.
17. Peterson, J.L. Petri Nets, *ACM Computing Surveys*, 9(3), 1977. pp 223-252.
18. Peyton-Jones, S.L. *The implementation of functional programming languages*, Prentice-Hall, 1987.
19. Poulovassilis, A. and Small, C. "A functional approach to deductive databases", *Proc. 17th VLDB Conference*, Barcelona, 1991. pp 491-500.
20. Poulovassilis, A. and Small, C. "Investigation of algebraic query optimisation in database programming languages", *Proc. 20th VLDB Conference*, Santiago, 1994. pp 415-426.
21. Poulovassilis, A. and Small, C. "Algebraic query optimisation in database programming languages", To appear in the *VLDB Journal*, April 1996.
22. Reddi, S. "Integrity enforcement in the functional database language PFL", *Proc. 11th British National Conference on Databases*, Keele, Springer-Verlag LNCS 696, 1993. pp. 238-257.
23. Reddi, S. Poulovassilis, A. and Small, C. "Extending a Functional DBPL with ECA-rules", in [24]. pp 101-115.
24. Sellis, T. (Editor) *Proc. 2nd Int. Workshop on Rules in Database Systems*. Springer-Verlag LNCS 985, 1995.
25. 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. pp 315-326.
26. Small, C. and Poulovassilis, A. "An overview of PFL", *Proc. DBPL-3*, Nauplion, 1991. pp 96-110.
27. Sutton, D. and Small, C. "Extending functional database languages to update completeness", *Proc. 13th British National Conference on Databases*, Manchester, July 1995. Springer Verlag LNCS 940. pp 47-63.
28. Widom, J., Cochrane, R.J. and Lindsay, B.G. "Implementing set-oriented production rules as an extension to Starburst", *Proc. 17th VLDB Conference*, Barcelona, 1991. pp 275-285.

Received Date
Accepted Date
Final Manuscript Date