, , 1–31 ()

Abstract Interpretation for Termination Analysis in Functional Active Databases

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

Editors: Peter Gray, Larry Kerschberg, Peter King

Abstract. An active database consists of a traditional database supplemented by a set of Event-Condition-Action (ECA) rules. One of the key questions for active database designers is that of termination of the ECA rules. The behaviour of the ECA rules may be obscure and their semantics is often not specified formally. Consequently, developing termination analysis algorithms and proving their correctness is a challenging task. In this paper we address this problem for functional active databases by adopting an abstract interpretation approach. By "functional active databases" we mean active databases whose transaction execution semantics have been expressed in a purely functional language. Although we demonstrate our techniques for a specific active DBMS which supports a functional database programming language interface, these techniques are directly applicable to other active DBMSs whose execution semantics have been specified using a functional or a denotational approach.

Keywords: Active rules, abstract interpretation, functional databases, functional specification, rule termination.

1. Introduction

Active databases provide the functionality of traditional databases and additionally are capable of reacting automatically to state changes, both internal and external, without user intervention. They have been intensively studied for over a decade and many prototypes have been built. Work on the formalisation of active database systems and their analysis, however, is somewhat scarcer. As a contribution in this direction, in this paper we show how *abstract interpretation* (Abramsky and Hankin, 1987) can be applied to functional specifications of transaction execution semantics in order to develop tests for active rule termination, and to formally prove the correctness of such tests.

The property of termination is particularly important for sets of ECA rules. A set of ECA rules terminates if for any initial database and transaction, each ECA rule will fire a finite number of times during the execution of the transaction. Given the undecidability of this problem even for relatively simple rule languages (Bailey et al., 1998), work on termination analysis focuses on detecting definite termination of ECA rules. Several static analysis techniques have been proposed for guaranteeing *a priori* that a given set of ECA rules definitely terminates. There has been little attempt, however, to explain their behaviour in a unifying manner. In this paper, we aim to show that abstract interpretation is a particularly appropriate formalism for understanding these analyses. Moreover, it provides a framework for developing correctness proofs for such algorithms, a non-trivial task at best for ECA rule systems.

Given a set of ECA rules, our objective is to provide the user with an executable program which terminates only if the set of ECA rules terminates for all initial databases and transactions. Such a program can be thought of as an *abstract version* of the transaction execution semantics. We present our ideas using the functional database programming language PFL (Reddi et al. 1995; Poulovassilis et al. 1996) to specify both the transaction execution semantics and their abstract version. The paper is organised as follows:

In Section 2 we give an overview of those aspects of PFL that are necessary for our purposes here, including its facilities for defining ECA rules and transactions. Note that we will be using PFL in two ways in this paper — as an example of an active DBMS for which termination analysis of ECA rules is important, and as a specification language for expressing concrete and abstract transaction execution semantics.

In Section 3 we specify the transaction execution semantics for the subset of PFL's ECA component that we will be considering (see footnote 3), using PFL itself as the meta language. In Section 4 we present a test that signals definite termination of a given set of ECA rules. This test uses a PFL function which is an abstract version of the transaction execution semantics and which, when applied to an abstract representation of a set of initial databases and transactions, terminates if and only if the ECA rules would terminate for that set of initial databases and transactions. We formally prove this property of the abstract semantics in Section 5 by using standard techniques from abstract interpretation. In Section 6 we summarise the results of the paper so far, and compare our rule termination analysis approach with related work.

In Section 7 we demonstrate the practical application of our termination analysis approach by identifying a broad class of rules for which it improves upon previous approaches. In Section 8 we show how our approach can be refined and extended: firstly, to handle more expressive rule languages by using approximate, but safe, tests that terminate only if the ECA rules would do so for any initial database and schedule; secondly, to detect definite non-termination in cases that the same database and schedule recur during transaction execution; and thirdly, to handle the non-determinism that results if the assumption that no two rules have the same priority is dropped.

We give our conclusions and directions for further work in Section 9.

2. The Language

PFL is a polymorphic, statically typed, functional database programming language. It allows users to declare new types and type synonyms (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) and define an ECA rule (Section 2.5).

In this section we give an overview of those aspects of PFL that are prerequisites for this paper. More details can be found in (Reddi et al. 1995; Poulovassilis et al. 1996): the first of these papers describes how PFL was extended with an ECA component without having to introduce any new language for defining ECA rules, and without changing PFL's computation model or type system; the second paper gives a formal semantics for the new ECA component and discusses the expressiveness of events and the optimisation of event-condition queries.

Henceforth in this paper, we use the terms "ECA rule", "active rule", and "rule" synonymously. 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. We use `teletype` font for PFL code.

2.1. Types

The built-in types are Num, Chr, and polymorphic list, set, product and function types. For any types $t, t_1, \ldots, t_n$, the type $[t]$ consists of lists of elements of type t, $\{t\}$ of sets of elements of type t, $(t_1, \ldots, t_n)$ of tuples of the form $(e_1, \ldots, e_n)$ where each $e_i :: t_i$, and $t_1 \rightarrow t_2$ of functions that map values of type t_1 to values of type t_2. The operator $\rightarrow$ is right-associative, so that $t_1 \rightarrow t_2 \rightarrow t_3$ is synonymous with $t_1 \rightarrow (t_2 \rightarrow t_3)$.

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 $e_1 : (e_2 : \ldots (e_n : [\,]))$. Similarly, the set constructors are $\{\,\}$ and $SetCons$ where, for all types t, $\{\,\} :: \{t\}$ and $SetCons\ e_1\ e_2 :: \{t\}$ if $e_1 :: t$ and $e_2 :: \{t\}$. An expression of the form $\{e_1, e_2, \ldots, e_n\}$ is synonymous with $SetCons\ e_1\ (SetCons\ e_2 \ldots (SetCons\ e_n\ \{\,\}))$.

New types and type synonyms can be declared using the symbols `::=` and `==`. For example, the boolean and string types can be declared as follows:

```
Bool  ::= True | False
Str    ==  [Chr]
```

We assume the usual syntax for strings in that, for example, the string "*hi*" is syntactically equivalent to the list of characters $['h', 'i']$. Our specification of PFL's rule execution semantics requires a type *SMode* (scheduling mode for active rules) consisting of two constants, *Imm* (immediate) and *Def* (deferred):

```
SMode ::= Imm | Def
```

2.2. Expressions and functions

An *expression* can be a variable, a constructor, a relation, a built-in function, an application of one expression to another, a lambda abstraction $\lambda p.e$ 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, a list comprehension or a set comprehension. Variables and relational names start with a lower-case letter and constructors with an upper-case letter. Application is left-associative and has higher precedence than any operator.

The built-in functions include the usual arithmetic and relational operators, and two set union operators, a lazy one, $\cup$, and a strict one, $\cup S$. $\cup$ and $\cup S$ differ in that the former outputs partially defined sets as they are computed whereas the latter outputs either a fully computed set or no information at all ($\perp$). Also built-in is a function $setmap :: (a \rightarrow \{b\}) \rightarrow \{a\} \rightarrow \{b\}$ which distributes a function of type $(a \rightarrow \{b\})$ over a set of type $\{a\}$ and returns the (lazy) union of the results. *setmap* has the following semantics:

$$
\begin{array}{lcl}
setmap\ f\ \{\,\} & = & \{\,\} \\
setmap\ f\ \{x\} & = & if\ (x = \perp)\ then\ \perp\ else\ (f\ x) \\
setmap\ f\ (s_1 \cup s_2) & = & (setmap\ f\ s_1) \cup (setmap\ f\ s_2)
\end{array}
$$

Its strict counterpart, *setmapS*, has $\cup S$ on the right-hand-side of the last equation rather than $\cup$. Set intersection, *intersect*, and set difference, *minus*, can be defined using *setmap* — see (Poulovassilis and Small, 1994; Poulovassilis and Small, 1996) for details. Another useful function that can be defined using *setmap* is $map :: (a \rightarrow b) \rightarrow \{a\} \rightarrow \{b\}$ which applies a function to each element of a set. Its strict counterpart, *mapS*, is defined similarly, using *setmapS* rather than *setmap*:

$$
\begin{array}{lcl}
map\ f\ s & = & setmap\ (\lambda x.\{f\ x\})\ s \\
mapS\ f\ s & = & setmapS\ (\lambda x.\{f\ x\})\ s
\end{array}
$$

The user can define new functions by specifying equations for them. These equational definitions translate into a single λ abstraction — see (Peyton Jones, 1987) for details of the translation. Below, we define some functions that will be needed later on in the paper. The symbol `==` is used for equation definitions and `=` 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;

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

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, and we sometimes use the syntactic sugar $if\ e_1\ then\ e_2\ else\ e_3$ for $if\ e_1\ e_2\ e_3$. Given a list of expressions $[e_1, \ldots, e_n]$, a binary operator *op* and an expression e_0, $fold\ [e_1, \ldots, e_n]\ op\ e_0$ is semantically equivalent to $(\ldots((e_0 \; op \; e_1) \; op \; e_2) \; op \ldots) \; op \; e_n$, writing *op* as an infix operator.

let expressions introduce local definitions for a set of variables $x_1, \ldots, x_n$. They take the form $let \; x_1 \;==\; e_1; \;\ldots;\; x_n \;==\; e_n \; in \; e$ and for the purposes of this paper can be regarded as being semantically equivalent to $(\lambda x_1.\ldots(\lambda x_n.e)\; e_n \ldots)\; e_1$ (however see (Peyton Jones, 1987) for a discussion of let-polymorphism in functional languages).

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". List comprehensions add no expressiveness to a functional language and are syntactically equivalent to successive applications of $flatmap$ and if:

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

Set comprehensions similarly take the form $\{e \mid q_1; \;\ldots;\; q_n\}$, where each q_i is either a filter or a generator, and are syntactically equivalent to successive applications of *setmap* and if:

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

2.3. Bulk data

Bulk data is stored in relations. The extent of a relation is a set of values of the same type. [1] 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 empty and can be updated by the built-in functions *inc* and *exc*. These take as arguments a relation name and a set of values, s. *inc* includes s into the relation's extent, by side-effect, and returns the relation name. *exc* similarly excludes s from the relation's extent and returns the relation name. [2]

2.4. Transactions

A transaction is a sequence of expressions

$$e_1; \; e_2; \; \ldots; \; e_n$$

which are evaluated in turn. The evaluation of an expression may cause the occurrence of one or more *primitive events*. The invocation of *inc* or *exc* is a primitive event. [3]

For each relation `r::{t}`, the "history" relations `rIncH::{t}` and `rExcH::{t}` respectively record the values that have been included into, and excluded from, the extent of `r` during the execution of the transaction. The "delta" relations `rIncD::{t}` and `rExcD::{t}` similarly record the inclusions into, and exclusions from, the extent of `r` during the evaluation of the current expression.

The ECA rules are processed after the evaluation of each expression of a transaction. No implicit event consumption occurs and all primitive events that have occurred during the transaction are accessible to the user via the history relations.

2.5. ECA Rules

ECA rules are defined by a command of the form:

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

where i is a unique number which determines the priority of the ECA rule. The occurrence of events is detected by the event-condition query, *ec*, which is of type $\{t\}$ for some t. *ec* can query the history and delta relations but must not update the database i.e. must not call *inc* or *exc*. Also, it must evaluate to empty if all the delta relations are empty i.e. if no events have occurred in the evaluation of the previous expression.

The evaluation of *ec* will yield a set of values, $evs = \{ev_1, \ldots, ev_n\}$, where each ev_j can be thought of representing the occurrence of an event during the evaluation of the current expression. The rule is triggered when *evs* is non-empty.

Each $action_i$ is a function to be applied to *evs*, and the resulting reaction transaction is

$$[action_1\ evs, \ldots, action_m\ evs]$$

sm is an expression of type *SMode* (defined in Section 2.1). The reaction transaction is run immediately before the current transaction is resumed if $sm = Imm$, and is deferred until the completion of the current transaction if $sm = Def$.

To illustrate, suppose that `accSite::{(Acc,GridRef)}` records road traffic accidents and the grid references at which they occurred, `roadClass::{(GridRef, Class)}` the class of road at each grid reference, `fatalities::{(Acc,Num)}` the number of fatalities of each accident, `fatalSpots::{GridRef}` grid references at which a fatality has occurred, and `mWayFatalSpots::{GridRef}` grid references that are on motorways and at which a fatality has occurred.

Then we can define the following ECA rule which materialises `fatalSpots` and `mWayFatalSpots` from `accSite` and `fatalities`:

```
_ecaRule 0 ==
   ({g | (a,g)<-accSiteIncD; (a,f)<-fatalities; f > 0},
    [lambda gs.inc fatalSpots gs,
     lambda gs.inc mWayFatalSpots
               {g | g<-gs; (g,c)<-roadClass; c = MWay}],
    Def);
```

For ease of reading, we have extended the comprehension notation here with a little more syntactic sugar, in that later references to the same variable within generator patterns inherit the bindings of earlier references. Thus, for example, the qualifiers `(a,g)<-accSiteIncD; (a,f)<-fatalities` above are equivalent to `(a,g)<-accSiteIncD; (a',f)<-fatalities; a=a'`.

The event-condition query of this rule returns a set consisting of the grid references of accidents that have been inserted into `accSite` during the last expression evaluation and at which fatalities have occurred. When applied to this set of grid references, the rule's first action will include them into `fatalSpots`. When applied to the same set of grid references, the rule's second action will include those that are on motorways into `mWayFatalSpots`. The `Def` scheduling mode causes re-materialisation of `fatalSpots` and `mWayFatalSpots` at the end of transactions which update `accSite`.

We note that the event-condition query of this rule is expressible in the relational algebra as

$$\pi_{\$2}\ (\sigma_{(accSiteIncD.\$1=fatalities.\$1)\wedge(fatalities.\$2>0)}(accSiteIncD \times fatalities))$$

This query also defines the set passed to `inc` in the rule's first action. The set passed to `inc` in the second action is also expressible in the relational algebra as follows, where *ec* is the above expression for the event-condition query:

$$\pi_{\$1}\ (\sigma_{(ec.\$1=roadClass.\$1)\wedge(roadClass.\$2=MWay)}(ec \times roadClass))$$

ECA rules such as this one whose event-condition queries and updates are expressible using relational algebra operators $\pi, \sigma, \times$ and $\cup$ are significant because we can develop a termination test for them which is *precise*, in the sense that it terminates if and only if the ECA rules terminate for all initial databases and transactions — we give this test in Section 4 below, and we discuss its practical application in Section 7.

For more expressive event-condition queries and updates it is still possible to develop practical termination tests that are *safe* but lose precision i.e. they terminate only if the ECA rules terminate for all initial databases and transactions, but may fail to terminate for some terminating ECA rule sets. We discuss such tests and their practical application in Section 8.1 below.

3. The Transaction Execution Semantics

In (Poulovassilis et al., 1996) PFL's transaction execution semantics were specified using PFL itself as the meta language. Three advantages of this were that (i) these formal semantics formed a robust basis for implementing PFL's ECA component, (ii) the semantics were executable and so allowed experimentation with the design of the ECA component before effort was invested in implementation, and (iii) the semantics were easily understood by PFL users since they were expressed in PFL itself. Here we aim to illustrate a further significant advantage of such functional specifications, namely that they can be used to develop tests for rule termination, and moreover for formally proving the correctness of such tests.

In this section we specify the transaction execution semantics for the subset of PFL's ECA component that we will be considering in this paper (see footnote 3). We use `teletype` font to specify these semantics in order to emphasize that they are expressed in PFL itself and thus form an executable specification. We first define the meta types necessary for writing the specification in Section 3.1. We then give the specification itself in Section 3.2.

3.1. The meta types

We first define a meta type `Expr` for expressing PFL expressions:

```
Expr ::= C Chr | N Num | Con Str | Var Str | RelName Str |
         Rel Str Expr  | Op Str [Expr] | ConFun Str [Expr] |
         Lambda Str Expr | Apply Expr Expr
```

The argument of the constructors `C`, `N`, `Con`, `Var` and `RelName` represents a character, number, constant, variable or relation name, respectively (we need the last of these because *inc* and *exc* take a relation name as an argument and return it as their result). The arguments of `Rel` are a relation name and its extent. The arguments of `Op` and `ConFun` are the name of a built-in function or constructor function and the list of arguments to which they are being applied. The constructors `Lambda` and `Apply` represent lambda abstractions and applications, respectively.

Consider for example the expression `R U S`, where the current extents of `R` and `S` in the database are {} and {1}, respectively. Recall from Section 2.1 that set enumeration is just syntactic sugar for applications of the constructor `SetCons` and the constant {}. Thus {1} is synonymous with `(SetCons 1 {})`. So the expression `R U S`, where `R` and `S` have current extents {} and {1}, is represented by the following value of the meta type `Expr`:

```
Op "U" [Rel "R" (Con "{}"),
        Rel "S" (ConFun "SetCons" [N 1,Con "{}"])]
```

We require some further meta type synonyms for specifying the execution semantics:

```
DBState   ==  {Relation}
Relation  ==  (Expr, Expr, Expr, Expr, Expr)
Schedule  ==  [Expr]
```

Thus, a database consists of a set of relations. A relation is represented by a 5-tuple of expressions, respectively denoting the relation's extent, and the extents of its inclusion history relation, exclusion history relation, inclusion delta relation and exclusion delta relation. In the database at the start of a transaction, the 5-tuple associated with a relation with name `"r"` and initial extent `e` is:

```
(Rel "r" e, Rel "rIncH" (Con "{}"), Rel "rExcH" (Con "{}"),
            Rel "rIncD" (Con "{}"), Rel "rExcD" (Con "{}"))
```

A schedule is a list of expressions, and the initial schedule consists of the expressions of the initial top-level transaction. We assume that each of these expressions is an inclusion into, or an exclusion from, a relation, and we make the same assumption for rule actions — we focus on such transactions and rule actions for the purposes of analysing rule termination since only *inc* and *exc* can generate events. Without loss of generality, we also assume that in the initial transaction each expression being included into or excluded from a relation is itself a relation. In other words, the initial transaction is a sequence of expressions of the form $inc/exc\ r\ \ s$ where both r and s are relations.

Expressions of the top-level transaction and expressions that result from the scheduling of reaction transactions are indistinguishable as far as the execution semantics are concerned. So we assume a canonical syntactic form for all such expressions, which is

$$inc/exc\ r\ \ (f\ evs) \tag{1}$$

for some function f and set-valued expression evs. We note that for expressions in the top-level transaction, f is the identity function, $id = \lambda x.x$, and evs is a relation. We assume that all functions f satisfy $f\ \{\} = \{\}$ i.e. the value of r is not changed if evs is empty.

3.2. The specification

We specify the transaction execution semantics as a function, `execSched`. This takes a database state and a schedule, and (i) executes the update at the head of the schedule, returning an updated database and schedule, (ii) processes each ECA rule in turn with respect to this new database, returning an updated schedule, and (iii) calls itself recursively with the new database and schedule. `execSched` terminates when passed an empty schedule:

```
execSched :: (DBState,Schedule) -> (DBState,Schedule)
execSched (db,s)  ==
    if s = []
    then (db,s)
    else execSched (makeSched o (fold RULES schedRule) o
                    addPreSuf o execUpdate (db,s))
```

Here, `RULES` is a global list holding the ECA rules in order of their priority, where each rule is represented by a triple `(ec,actions,sm)::(Expr,[Expr],SMode)`. We next define each of the functions called by `execSched`.

`execUpdate` pops the first expression, `e`, from the front of the schedule, and applies the function `update e` to each 5-tuple in the database state:

```
execUpdate (db,e:s) == (map (update e) db,s)
```

Due to our canonical representation of all expressions in a schedule ((1) above), `e` will be of the form `Op "inc" [RelName rel,tuples]` or `Op "exc" [RelName` rel,tuples] for some relation name rel, where tuples is of the form Apply f evs for some function f and set-valued expression evs expressed as values of type Expr. update e has no effect on any but the relation being updated. For that relation, its current extent and its history and delta extents are updated to reflect the inclusion or exclusion:

```
update (Op "inc" [RelName rel,tuples])
       (Rel r e,Rel rIncH e2,Rel rExcH e3,Rel rIncD e4,Rel rExcD e5)
== if (rel = r)
   then (Rel r (Op "U" [e,tuples]),
         Rel rIncH (Op "U" [e2, Op "minus" [tuples,e]]),
         Rel rExcH e3,
         Rel rIncD (Op "minus" [tuples,e]),
         Rel rExcD (Con "{}"))
   else (Rel r e,Rel rIncH e2,Rel rExcH e3,Rel rIncD e4,Rel rExcD e5)

update (Op "exc" [RelName rel,tuples])
      (Rel r e,Rel rIncH e2,Rel rExcH e3,Rel rIncD e4,Rel rExcD e5)
== if (rel = r)
   then (Rel r (Op "minus" [e,tuples]),
         Rel rIncH e2,
         Rel rExcH (Op "U" [e3, Op "intersect" [e,tuples]]),
         Rel rIncD (Con "{}"),
         Rel rExcD (Op "intersect" [e,tuples]))
   else (Rel r e,Rel rIncH e2,Rel rExcH e3,Rel rIncD e4,Rel rExcD e5)
```

addPreSuf prepares for rule processing by creating an empty prefix schedule and suffix schedule. The reaction transaction of each rule that fires will subsequently be appended to the prefix schedule if the rule has immediate scheduling mode, or to the suffix schedule if the rule has deferred scheduling mode (see schedRule below). After rule processing, makeSched creates the new schedule to be passed to the next call of execSched:

```
addPreSuf (db,s)         == (db,[],s,[])
makeSched (db,pre,s,suf) == (db,pre ++ s ++ suf)
```

schedRule considers a particular ECA rule, and if its event-condition query evaluates to a non-empty set, appends the rule's reaction transaction to the schedule prefix or suffix, depending on the rule's scheduling mode:

```
schedRule (db,pre,s,suf) (ecQuery,actions,schedMode) ==
    let (db,pre,s,suf,evs) == addEvents ecQuery (db,pre,s,suf) in
        if evalEvents (db,pre,s,suf,evs) = (Con "{}")
        then (db,pre,s,suf)
        else if schedMode = Imm
             then addToPre actions (db,pre,s,suf,evs)
             else addToSuf actions (db,pre,s,suf,evs)
```

The functions addEvents, evalEvents, addToPre and addToSuf are defined as follows:

```
evalEvents (db,pre,s,suf,evs)       == eval evs
addEvents ecQuery (db,pre,s,suf )   ==
    (db,pre,s,suf,substitute ecQuery db)
addToPre actions (db,pre,s,suf,evs) ==
    (db,pre++[betaReduce (Apply a evs) | a <- actions],s,suf)
addToSuf actions (db,pre,s,suf,evs) ==
    (db,pre,s,suf++[betaReduce (Apply a evs) | a <- actions])
```

In evalEvents, the function eval::Expr->Expr is a normal-order reduction evaluator that successively applies β-reduction and δ-reduction in order to convert its argument to normal form (β-reduction rewrites an application of a lambda abstraction to an argument by substituting occurrences of the bound variable in the body of the lambda abstraction by the argument; δ-reduction rewrites an application of a built-in function by the result of executing the predefined code for that function for the given arguments; see (Peyton Jones, 1987) for more details). In addEvents, the function substitute replaces each occurrence of a relation name, history relation name or delta relation name in an event-condition query by its current extent in the database. Finally, in addToPre and addToSuf, the function betaReduce carries out a β reduction on the application of a rule action, a, to a set, evs, in order to convert it to the canonical form in (1) above.

4. Testing for Rule Termination

Given a set of ECA rules, we want to know if execSched will terminate for all initial databases and schedules. In this section we develop a test that signals definite termination of a given rule set. This test uses a function execSched* which is applied to an abstract representation of a set of database states and schedules. execSched* terminates if and only if execSched terminates for those database states and schedules — we will formally prove this in Section 5 by drawing on standard techniques from abstract interpretation (Cousot and Cousot, 1992) . Here, we first define execSched* in 4.1 and then give the overall rule termination test in 4.2.

4.1. The Abstract Semantics

We require the following types and type synonyms in order to define execSched*:

```
Schedule*  ==  [Expr*]
DBState*   ==  {Relation*}
Relation*  ==  (Expr*,Expr*,Expr*,Expr*,Expr*)
Expr*      ::= C Chr | N Num | Con Str | Var Str | RelName Str |
               Rel Str Expr* | Op Str [Expr*] | ConFun Str [Expr*] |
               Lambda Str Expr* | Apply Expr* Expr* | Null
```

These "abstract" types are identical to their "concrete" counterparts, except that `Null` is now a valid expression. It is used as an abstraction for the initial extent of all relations. Thus, in the abstract database state at the start of a transaction, the 5-tuple associated with a relation name `"r"` is:

```
(Rel "r" Null, Rel "rIncH" (Con "{}"), Rel "rExcH" (Con "{}"),
               Rel "rIncD" (Con "{}"), Rel "rExcD" (Con "{}"))
```

Similarly, the initial abstract schedule is a sequence of expressions of the form *inc/exc r s*, where there are tuples with first component *Rel r Null* and *Rel s Null* in the initial abstract database state. `execSched*` is defined as follows:

```
execSched* :: (DBState*,Schedule*) -> (DBState*,Schedule*)
execSched* (db*,s*) ==
    if s* = []
    then (db*,s*)
    else execSched* (makeSched o (fold RULES schedRule*) o
                     addPreSuf o execUpdate (db*,s*))

schedRule* (db*,pre*,s*,suf*) (ecQuery,actions,schedMode) ==
    let (db*,pre*,s*,suf*,evs*) ==
        addEvents ecQuery (db*,pre*,s*,suf*) in
        if equivEvents (db*,pre*,s*,suf*,evs*) (Con "{}")
        then (db*,pre*,s*,suf*)
        else if schedMode = Imm
             then addToPre actions (db*,pre*,s*,suf*,evs*)
             else addToSuf actions (db*,pre*,s*,suf*,evs*)
```

We have again used `teletype` font in order to emphasize that `execSched*` is expressed in PFL itself and is thus directly executable. The functions `makeSched`, `addPreSuf`, `execUpdate`, `addEvents`, `addToPre` and `addToSuf` are as defined in Section 3.2. The function `equivEvents` in `schedRule*` is defined as follows:

```
    equivEvents (db*,pre*,s*,suf*,evs*) (Con "{}") ==
        equiv evs* (Con "{}")
```

where `equiv` satisfies the following equivalence:

$$equiv\ evs*\ \ (Con\ "\{\}") \Leftrightarrow map\ eval\ (conc\ (evs*))\ =\ \{Con\ "\{\}"\}$$

Here, *conc* expands out an abstract expression into a set of all its possible concrete counterparts by substituting for each occurrence of *Rel r Null* an actual concrete extent in place of *Null* — we discuss *conc* in more detail in Section 5.2. In other words, the expression *evs∗* is equivalent to the empty set if and only if all possible instantiations of concrete extents for relations would result in it evaluating to the empty set.

Clearly, it is not practically feasible to implement *equiv* like this. In the case that the event-condition queries and updates of the ECA rules are expressible in the relational algebra, and there are no subexpressions of the form e_1 *minus* e_2 occurring within the scope of a projection operation in these expressions, we can employ the algorithm described in (Sagiv and Yannakakis, 1980) to implement *equiv*. If the ECA rules do not satisfy this restriction, it is still possible to develop practical, safe approximations of *equiv*, safe in the sense that the following implication holds:

$$equiv_{approx}\ evs*\ \ (Con\ "\{\}") \Rightarrow map\ eval\ (conc\ (evs*))\ \ = \{Con\ "\{\}"\} \quad (2)$$

We discuss one example of such an approximation in Section 8.1 below.

4.2. The Rule Termination Test

Let $r_1, \ldots, r_n$ be the set of relations called by the event-condition queries of the rules in `RULES`. The rule termination test consists of invoking *execSched∗* $2n$ times, once for each top-level transaction of the form $[inc\ r_i\ s]$ or $[exc\ r_i\ s]$ for $i = 1 \ldots n$, where s is a new relation added to *db∗* as an abstraction for an arbitrary set of tuples being included into or excluded from r_i. It is sufficient to consider these $2n$ transactions since only these actions may trigger a rule. `RULES` is a terminating rule set if all $2n$ invocations of *execSched∗* terminate. We will give some examples of applying this termination test in Sections 7 and 8.

5. Correctness of the Abstract Semantics

In this section we formally prove the correctness of *execSched∗* by setting up a Galois connection (Cousot and Cousot, 1992). For readers not familiar with the fundamentals of abstract interpretation and Galois connections, an overview is given in Appendix A.

The information ordering on sets that we use is that of the Smyth powerdomain (Schmidt, 1986; Søndergaard and Sestoft, 1992) since we are interested in showing the correctness of *execSched∗* in cases of definite termination. The Smyth powerdomain over a set consists of all finite, non-empty subsets of the set, together with a least element, $\perp$. The information ordering is defined as

$$s_1 \sqsubseteq_{Smyth} s_2 \ \text{ if } \ s_2 \subseteq s_1$$

The Smyth powerdomain is the correct one to use for our purposes since functions over sets have worse termination properties when applied to larger sets than when applied to smaller sets.

We begin the section by defining a function *execSchedC* that captures the collective termination behaviour of *execSched* for a set of input database states and schedules (Section 5.1). We then define an “abstraction” and a “concretisation” function for converting between our concrete and abstract domains and show how standard theory from abstract interpretation automatically yields an approximation of *execSchedC* over the abstract domain (Section 5.2). We then show that the *execSched∗* function defined in Section 4 is indeed this approximation (Section 5.3).

5.1. The Collecting Semantics

The first thing we need to do is to develop a "collecting" semantics that captures the collective termination behaviour of *execSched* for a set of database state and schedule inputs, *dbs*. The semantics we want is given by

$$mapS\ \ execSched\ \ dbs$$

The use of *mapS* here rather than *map* means that the resulting set of database and schedule pairs will be output only when every application of *execSched* to a pair $(db, sched) \in dbs$ has terminated (with an empty schedule). Thus, the above expression terminates if and only if *execSched* terminates for all $(db, sched) \in dbs$.

We now want to rewrite this expression so that the database and schedule pairs in *dbs* are processed collectively rather than individually. This is achieved by the following function `execSchedC`:

```
execSchedC :: {(DBState,Schedule)} -> {(DBState,Schedule)}
execSchedC dbs ==
    if {s | (db,s) <- dbs} = {[]}
    then dbs
    else execSchedC ((map makeSched) o (fold RULES schedRuleC) o
                     (map addPreSuf) o (map execUpdate) dbs)

schedRuleC dbPSS (ecQuery,actions,schedMode) ==
    let dbPSSE == map (addEvents ecQuery) dbPSS in
        if (map evalEvents dbPSSE) = {Con "{}"}
        then dbPSS
        else if schedMode = Imm
             then map (addToPre actions) dbPSSE
             else map (addToSuf actions) dbPSSE
```

We claim that

$$execSchedC\ \ dbs\ =_{eval}\ mapS\ \ execSched\ \ dbs$$

Here, $=_{eval}$ is a semantic equality test i.e. it applies *eval* to the relation extents in each database state on the left-hand side and right-hand side and then undertakes an ordinary equality test, $=$.

To see why this equivalence holds, we first we observe that *execSchedC dbs* applies precisely the same functions to each $(db, sched) \in dbs$ as *mapS execSched dbs* would, except that *execSchedC* continues scheduling a rule's reaction transaction until the rule's event-condition query evaluates to the empty set for all $(db, sched)$ pairs (i.e. until $map\ \ evalEvents\ \ dbPSSE\ =\ \{Con\ "\{\}"\}$ is true in *schedRuleC*) whereas *mapS execSched dbs* would stop scheduling a rule's reaction transaction for any individual $(db, sched)$ pair for which the rule's event-condition query evaluated to the empty set (i.e. when $evalEvents\ \ (db, pre, s, suf, evs)\ =\ (Con\ "\{\}")$ is true in *schedRule*).

However, in the case of rules whose event-condition query evaluates to the empty set, the updates scheduled by $execSchedC$ are of the form $inc\ r\ e$ or $exc\ r\ e$ where e evaluates to the empty set. These are "null" updates in the sense that they do not change the value of the relation extents or history extents, and they also set all the delta extents to empty. Since they set the deltas to empty, such updates can only result in further "null" updates being scheduled by $execSchedC$. We finally note that, like $mapS\ \ execSched\ \ dbs$, $execSchedC$ terminates when all the schedules have become empty. Thus, $execSchedC\ \ dbs$ and $mapS\ \ execSched\ \ dbs$ both either output an equivalent set of database state and schedule pairs or both fail to terminate and output no information.

5.2. Defining abs and conc

Without loss of generality, we can assume that the possible input databases all have the same schema. We first need an abstraction function

$$absOne :: (DBState, Schedule) \rightarrow (DBState{*}, Schedule{*})$$

which replaces every concrete extent of a relation r appearing in the database and schedule by $Null$. $absOne$ can be extended to abs which operates over sets of pairs of database states and schedules

$$abs :: \{(DBState, Schedule)\} \rightarrow (DBState{*}, Schedule{*})$$

provided abs is applied to a set of pairs whose database components all have the same schema and whose schedule components are identical apart from the extents of relations. All applications of abs below do satisfy this proviso and we assume it henceforth.

Conversely, we can write a concretisation function

$$conc :: (DBState{*}, Schedule{*}) \rightarrow \{(DBState, Schedule)\}$$

which replaces every abstract (i.e. $Null$) extent of a relation appearing in the abstract database state and schedule, by a possible concrete extent. Assuming that all relations have finite domains, a finite set of possible database states and schedules is returned. Again, we also make this assumption henceforth.

It is easy to see that for any set $S :: \{(DBState, Schedule)\}$ whose database components all have the same schema and whose schedule components are identical apart from the extents of relations,

$$S \ \subseteq\ conc \circ abs\ S$$

and so

$$conc \circ abs\ S \sqsubseteq_{Smyth} S \tag{3}$$

It is also easy to see that for any abstract database and schedule pair, $(db{*}, sched{*})$,

$$(db{*}, sched{*}) \ = \ abs \circ conc\ (db{*}, sched{*}) \tag{4}$$

Now, $execSched$ will terminate for any initial database and schedule in $conc$ ($db*$, $sched*$) if

$$execSchedC\ (conc\ (db*, sched*))\ \neq \perp$$

so we require a test function, $test$, such that

$$test\ (db*, sched*) \neq \perp\ \Rightarrow execSchedC\ (conc\ (db*, sched*)) \neq \perp \tag{5}$$

By (3) and (4), abs and $conc$ satisfy the definition of a Galois connection and so we can show that

$$test\ =\ abs \circ execSchedC \circ conc \tag{6}$$

approximates $execSchedC$ i.e. that for all sets $S :: \{(DBState, Schedule)\}$ whose database components all have the same schema and whose schedule components are identical apart from the extents of relations,

$$conc \circ test \circ abs\ S\ \sqsubseteq_{Smyth}\ execSchedC\ S \tag{7}$$

(the general proof of this for any Galois connection is given in Appendix A).

Now, if $test\ (abs\ S) \neq \perp$, then $conc\ (test\ abs\ S) \neq \perp$ and so, by (7), $execSchedC\ S \neq \perp$. In other words,

$$test\ (abs\ S) \neq \perp\ \Rightarrow execSchedC\ S \neq \perp$$

Substituting for S by $conc\ (db*, sched*)$ and using (4) to simplify the left-hand-side gives us the result we need i.e. gives (5).

5.3. *Correctness of execSched**

It remains to show that $execSched*$ is indeed the function $test$ in (6) above i.e. that for all $(db*, s*)$

$$execSched*\ (db*, s*)\ =\ abs \circ execSchedC \circ conc\ (db*, s*) \tag{8}$$

We first observe that $execSched*$ recursively applies itself to the current abstract database and schedule pair until the schedule is empty, while $execSchedC$ recursively applies itself to the current set of database and schedule pairs until all the schedules are empty. Let us denote by $(db*_0, s*_0), (db*_1, s*_1), \ldots$ the successive abstract database and schedule pairs passed to $execSched*$ and by $ds_0, ds_1, \ldots$ the successive sets of concrete database and schedule pairs passed to $execSchedC$. Each $(db*_i, s*_i)$ (for $i > 0$) is obtained by applying

$$makeSched \circ (fold\ RULES\ schedRule*) \circ addPreSuf \circ execUpdate$$

to $(db*_{i-1}, s*_{i-1})$, while each ds_i (for $i > 0$) is obtained by applying

$$(map\ makeSched) \circ (fold\ RULES\ schedRuleC) \circ (map\ addPreSuf) \circ \\ (map\ execUpdate)$$

to ds_{i-1}. We can show by induction on i that

$$conc\ (db*_i, s*_i) \ = \ ds_i \tag{9}$$

and the proof of this is given in Appendix B.

Therefore $s*_i = [\,]$ iff $s_i = [\,]$ for all schedules s_i in ds_i. Thus $execSched*$ and $execSchedC$ either both terminate on the same iteration, say k, or both do not terminate and return $\perp$. In the latter case, (8) holds trivially. In the former case, we have from (9) that

$$conc\ (db*_k, s*_k) \ = \ ds_k$$

Since $(db*_k, s*_k) = execSched*\ (db*, s*)$ and $ds_k = execSchedC\ (conc\ (db*, s*))$, this gives that

$$conc \circ execSched*\ (db*, s*) \ = \ execSchedC \circ conc\ (db*, s*)$$

Applying abs to both sides of this equality and using (4) to simplify the left-hand side gives the result we need i.e. gives (8).

6. Discussion

In the foregoing sections, we have defined both a concrete and an abstract semantics for ECA rule execution. The formal nature of these definitions has facilitated a formal proof of correctness of the abstract semantics using Galois connections.

We observe here the syntactic similarity between $execSched$, $execSchedC$ and $execSched*$. In fact, we developed $execSchedC$ before developing $execSched*$. We then derived the definition of $execSched*$ by applying the abstraction function abs to the leaves of the computation of $execSchedC$, where sets of database state and schedule pairs are manipulated. The design of the abstract semantics was thus done in a systematic, rather than an ad hoc, fashion. This has the advantage that the definitions of $execSched$, $execSchedC$ and $execSched*$ are easily evolved if we wish to capture different execution semantics or to use different abstract domains. We consider two evolutions of them in Sections 8.2 and 8.3.

Our approach is rather different from most previous work on rule analysis, for example that of (Aiken et al., 1995; Baralis et al., 1995). Such work typically provides an informal description of a rule execution semantics, describes a rule analysis algorithm and (perhaps) argues its correctness. There are drawbacks to this kind of approach, however: the looseness of the execution specification raises doubts about the validity of any demonstration of correctness; furthermore, there is no general methodology for developing such proofs and so their formulation tends to be case-specific.

For example, consider the work in (Baralis et al., 1995), which describes how to perform termination analysis based on a method called *rule reduction*. This involves building a graph, with rules represented by vertices and two types of directed arcs: a *triggering arc* from a rule r_1 to a rule r_2 indicates that the execution of r_1's action may cause r_2's event query to become true; an *activation arc* from a rule r_1 to a rule r_2 indicates that the execution of r_1's action may cause r_2's condition query to become true if $r_2 \neq r_1$, or to remain true if $r_2 = r_1$. Vertices which do not have both an incoming triggering arc and an incoming activation arc can be recursively removed from the graph. Definite termination can be inferred if one can reduce the graph to an empty graph. An extension for handling rule priorities is also given. Although a proof of correctness is given, it is faulty — see (Zimmer et al., 1996) for a counter-example to the behaviour of the algorithm. This highlights the important point that such proofs are difficult, particularly without the assistance of established formal machinery.

Regarding the precision of $execSched*$, our abstract execution is essentially a generalisation of graph-based techniques such as rule reduction. Such techniques focus on relationships between pairs of rules. In contrast, our satisfiability testing is global in nature. Indeed our abstract execution *automatically* includes triggering and activation relationships between rules, and for a simple rule cycle of length k we obtain equal or better precision than the rule reduction method after $O(n \times k)$ iterations of $execSched*$, where n is the number of relations invoked in the rules' event-condition queries. We give some examples in Section 7 of rule sets for which rule reduction gives no information but for which our algorithm concludes definite termination after $O(n \times k)$ iterations of $execSched*$.

Regarding the efficiency of our termination test, the complexity of execution of $execSched*$ is clearly dependent upon the implementation of $equiv$ in $schedRule*$. The algorithm in (Sagiv and Yannakakis, 1980) can be used if the event-condition queries and updates of the ECA rules are expressible in the relational algebra, and there are no subexpressions of the form $e_1 - e_2$ occurring within the scope of a projection operation. This algorithm is NP-hard in the size of the expression. As the rule execution progresses, the size of the expressions passed to $equiv$ increases. It is therefore likely that after some number of iterations the test will become computationally intractable. It is an important area of research to determine where such a practical cut-off point lies for "typical" ECA rule sets. As a step in this direction, in Section 7 we identify a broad class of terminating ECA rule sets for which previous termination analysis approaches give no information and for which our algorithm quickly concludes definite termination, so that the size of the expressions passed to $equiv$ is still small.

In practice, $execSched*$ can be modified to take two extra parameters which record the maximum number of times that $execSched*$ should be called and the current iteration. If $execSched*$ terminates with a non-empty schedule after the maximum number of iterations then we can infer no information about the termination properties of the ECA rules. Again, it is an area of further research to determine a practically useful maximum number of iterations of $execSched*$ for "typical" terminating ECA rule sets. The example rule sets in Sections 7 and 8 give a preliminary indication that useful results can be obtained after a small number of iterations.

We conclude this section with a brief comparison between our approach and related work. We have already observed the difference between our approach and the graph-based methods for rule termination analysis. In (Zimmer et al., 1996), the semantics of rule execution are captured via Petri Nets and reachability analysis is then used. However, this analysis is still tied to the notions of activation and triggering. On the semantics side, work in (Widom, 1992; Coupaye and Collet, 1995) has developed denotational semantics for rule execution. These are presented "as is" however, without any direct applications to rule analysis being given.

The work presented in this paper is partly based on work in (Bailey et al., 1997; Bailey and Poulovassilis, 1998). In (Bailey et al., 1997) an initial approach to using abstract interpretation for termination analysis was presented, where it was applied to a simple variable language, not to a practical database language as we have done here. In (Bailey and Poulovassilis, 1998) methods for increasing the precision of termination analysis are described. The techniques in both those papers are described informally though, in an imperative language context, and without a full proofs of correctness.

Abstract interpretation has of course been used previously for analysing functional programs, with the first application being for strictness analysis (Mycroft, 1981). There, a simple $\{0,1\}$ domain is used, with 0 corresponding to the fact that a function argument definitely fails to terminate and 1 corresponding to the fact that it may terminate. Such analysis would seem, at least on the surface, to be suitable for examining termination properties of active rules. Unfortunately though, such descriptions are too coarse to provide interesting information. That is why we have needed to develop more sophisticated abstractions for database states and schedules, namely the types `DBState*` and `Schedule*`.

7. Practical Application

In this section we demonstrate the practical applicability of our termination analysis approach by first giving an example rule set for which previous methods would be inconclusive but our termination test concludes definite termination. We then generalise this result by identifying a broad class of terminating ECA rule sets for which previous termination analysis approaches give no information but our test concludes definite termination after a small number of iterations.

For our example we extend the schema used in for ECA rule 0 in Section 2.5 to include the relations `safe::{gridRef}` and `priority::{gridRef}`. The former contains grid references at which no accidents have occurred and the latter grid references that have top priority (e.g. are vital for reasons of national security). The following two rules are used for maintaining the `safe` relation. Rule 1 maintains the integrity of `safe` upon insertions into it, provided `safe` contains at least one priority grid reference. Rule 2 prevents any deletions from `safe` that would eliminate grid references at which no accidents have occurred:

```
_ecaRule 1 == ({g | g <- safeIncD; (a,g) <- accSite;
                    (safe intersect priority) != {}},
                [lambda gs.exc safe gs],
                Imm)
```

```
_ecaRule 2 == (safeExcD,
               [lambda gs.inc safe (gs minus {g | (a,g) <- accSite})],
               Imm)
```

To analyse the termination properties of these rules, we need to (abstractly) execute two top-level transactions, `[inc safe s]` and `[exc safe s]`. With the former, rule 1 is triggered. Substituting for `safeIncD` by `s`, its event-condition query is

```
q = {g | g <- s; (a,g) <- accSite; (safe intersect priority) != {}}
```

which is satisfiable. The action `exc safe q` is therefore executed and rule 2 is triggered. Substituting `safeExcD` by `q`, its event-condition query is just `q`, which is satisfiable. The action `inc safe (q minus {g | (a,g) <- accSite})` is therefore executed and rule 1 is again triggered. Substituting `safeIncD` by `(q minus {g | (a,g) <- accSite})`, its event-condition query is now

```
{g | g <- q minus {g | (a,g) <- accSite};
     (a,g) <- accSite;
     (safe intersect priority) != {}}
```

and this is not satisfiable because any bindings generated for `g` in the first qualifier cannot appear within `accSite`. Hence the execution halts. The situation is similar starting with the other top-level transaction, `[exc safe s]`.

The rule reduction method would make no conclusions with this example because rules 1 and 2 mutually trigger each other, rule 2 activates rule 1 and rule 2 activates itself (trivially, because its condition query is just `True`). Thus, neither rule can be eliminated from the triggering/activation graph. However, what is actually happening is that execution of rule 1 makes false one conjunct of its own condition, namely that `safe` and the grid references in `accSite` intersect, while execution of rule 2 may activate only the *other* conjunct, namely that the intersection of `safe` and `priority` is non-empty. Thus, rule 1 can never be executed more than once.

We can generalise the above example to any pair of active rules of the form

$$\begin{aligned} r_1 &= (e_1 \wedge c_1 \wedge c_1', a_1, sm_1) \\ r_2 &= (e_2 \wedge c_2, a_2, sm_2) \end{aligned}$$

where execution of a_1 may make e_2 true and makes c_1' false, and execution of a_2 may make e_1 and c_1 true and may leave c_2 true (e_1 and e_2 being the event parts and $c_1 \wedge c_1'$ and c_2 the condition parts of the two event-condition queries). The rule reduction method tells us that such a pair of rules may fail to terminate, because r_1 and r_2 both have incoming activation and triggering arcs. But in fact r_1 will never execute more than once because it makes conjunct c_1' of its own condition query false and r_2 may only make true the other conjunct of r_1's condition query, c_1.

This kind of fine-grained activation dependency between rule actions and *sub-expressions* of conjunctive condition queries is not captured by graph-based methods such as rule reduction, but it is easily detected using our approach. Rules with disjunctive conditions, i.e. with event-condition queries of the form $e_1 \wedge (c_1 \vee c_2)$, are also amenable to analysis using our method. In this case the rule action may make false only one of the disjuncts c_1, c_2 so the rule will have an activation arc to itself in the triggering/activation graph, thus lowering the precision of the rule reduction method if some other rule makes false the other disjunct. Our approach does not rely on rules being "self deactivating" in this way and so disjunctive conditions do not cause a problem.

One scenario in which rules with these kinds of fine-grained activation dependencies are likely to arise is when *statecharts* are used for the conceptual modelling of an application's behavioural semantics and are then translated into active rules, as happens in the IDEA methodology for example (Ceri and Fraternalli, 1997). This is because transitions from states that are superstates of a conjunction or disjunction of substates give rise to rules with conjunctive or disjunctive condition queries.

8. Extensions and Refinements

8.1. Extending the Rule Language

The *equiv* algorithm of (Sagiv and Yannakakis, 1980) is only able to test satisfiability of relational algebra expressions not containing subexpressions of the form $e_1 - e_2$ occurring within the scope of a projection operation. For ECA rules that do not satisfy this restriction, we need to develop practical, safe approximations, $equiv_{approx}$, that satisfy the criterion (2) in Section 4. For any such safe equivalence test, if $execSched{*}_{approx}$ is defined as $execSched{*}$ but calling $equiv_{approx}$ rather than *equiv*, it is easy to see that $execSched{*}_{approx}$ will not terminate before *execSchedC* does, making it a safe test for the definite termination of *execSchedC*.

For example, to handle arbitrary relational algebra expressions, *interval approximation* of the contents of relations can be employed. A subexpression of the form $\pi\,(e_1 - e_2)$ can be approximated by the interval $[\emptyset, \pi\, e_1]$. This treatment extends to the contents of any relation r being approximated by an interval $[a, b]$ such that $a \subseteq r \subseteq b$. Such an interval expression is satisfiable if and only if its upper limit, b, is satisfiable. The relational algebra operations are defined as follows over such interval expressions, where a, b, c, d are relational algebra expressions, $a \subseteq b$, $c \subseteq d$, $e_1 = [a, b]$ and $e_2 = [c, d]$:

$$
\begin{array}{lcl}
e_1 \cup e_2 & = & [a \cup c, b \cup d] \\
e_1 - e_2 & = & [a - d, b - c] \\
\sigma_\theta(e_1) & = & [\sigma_\theta(a), \sigma_\theta(b)] \\
\pi_A(e_1) & = & [\pi_A(a), \pi_A(b)] \\
e_1 \times e_2 & = & [a \times c, b \times d] \\
[[a, b], [c, d]] & = & [a, d]
\end{array}
$$

To illustrate, consider the following rule which, upon any insertion into `fatalSpots`, also inserts all tuples in `blackList` into `fatalSpots` if `blackList` contains grid references that don't appear in `fatalSpots`:

```
_ecaRule 3 == ({g | s <- fatalSpotsIncD;
                    g <- blackList;
                    blackList minus fatalSpots != {}},
                [lambda gs.inc fatalSpots gs],
                Imm)
```

The above event-condition query is a projection of the variable `g` from the cartesian product of `fatalSpotsIncD`, `blackList` and `blackList minus fatalSpots` i.e. it is

$$\pi_{\$2}(fatalSpotsIncD \times blackList \times (blackList - fatalSpots))$$

This expression contains a projection over a difference, and we approximate it by the interval

$$[\emptyset, \pi_{\$2}(fatalSpotsIncD \times blackList \times blackList)]$$

or, simplifying, by

$$q = [\emptyset, \pi_{\$2}(fatalSpotsIncD \times blackList)]$$

There is only one abstract transaction we have to test, `[inc fatalSpots s]`. Rule 3 is triggered, and substituting $fatalSpotsIncD$ by s in q gives a satisfiable expression

$$e_1 = [\emptyset, \pi_{\$2}(s \times blackList)]$$

The rule action executes. The abstract extent of $fatalSpots$ becomes $fatalSpots \cup e_1$ and that of $fatalSpotsIncD$ becomes $e_1 - fatalSpots$. Substituting the latter in q gives

$$e_2 = [\emptyset, \pi_{\$2}((e_1 - fatalSpots) \times blackList)]$$

Using interval approximation to again remove the projection over difference, this simplifies to e_1 above, which is satisfiable. The rule action is executed a second time. The abstract extent of $fatalSpots$ becomes $fatalSpots \cup e_1 \cup e_1$ and that of $fatalSpotsIncD$ becomes $e_1 - (fatalSpots \cup e_1)$. This is not satisfiable and the execution halts. The main point here is that employing the interval approximation does not prevent us from deducing definite termination, since the sub-expressions being approximated are not subject to modification.

For languages more expressive than the relational algebra, including, for example, recursion and aggregation functions as found in the full PFL rule language, we conjecture that techniques from incremental view materialisation and approximate queries (Chaudhuri and Kolaitis, 1997; Libkin and Wong, 1997; Qian and Wiederhold, 1991) can be used to develop safe approximate equivalence tests, and this is an important area of future work.

8.2. Detecting definite non-termination

A given set of ECA rules will fail to terminate if the same database state and schedule recurs during the transaction execution. We could therefore equip *execSched* with an extra parameter that records the history of all database states and schedules so far, and test for membership of the current database state and schedule in this history. The modified *execSched* function is:

```
execSched :: [(DBState,Schedule)] -> (DBState,Schedule) ->
              (DBState,Schedule)
execSched h (db,s)  ==
    if (s = []) or (member (db,s) h)
    then (db,s)
    else execSched ((db,s):h) (makeSched o (fold RULES schedRule) o
                               addPreSuf o execUpdate (db,s))
```

Here, *member* (db, s) h undertakes a semantic equality test of (db, s) with every element (db', s') of h i.e. it calls *eval* on the relation extents in (db, s) and (db', s'). *execSched* is initially called with the empty list and the initial database and schedule pair (db_0, s_0). If *execSched* terminates with a non-empty schedule, then the rules are definitely non-terminating for (db_0, s_0).

Due to the size of real databases, this is clearly not a practical extension for *execSched*. However, the abstract counterpart of this extension *is* potentially a useful practical extension for *execSched*∗ because abstract databases are generally much smaller than real databases (since they just contain a constant in place of each actual initial relation extent). The modified *execSchedC* and *execSched*∗ functions are:

```
execSchedC :: [{(DBState,Schedule)}] -> {(DBState,Schedule)} ->
               {(DBState,Schedule)}
execSchedC hs dbs ==
    if ({s | (db,s) <- dbs} = {[]}) or (member dbs hs)
    then dbs
    else execSchedC (dbs:hs)
                    ((map makeSched) o (fold RULES schedRuleC) o
                     (map addPreSuf) o (map execUpdate) dbs)


execSched* :: [(DBState*,Schedule*)] -> (DBState*,Schedule*) ->
               (DBState*,Schedule*)
execSched* h* (db*,s*) ==
    if (s* = []) or (member* (db*,s*) h*)
    then (db*,s*)
    else execSched* ((db*,s*):h*)
                    (makeSched o (fold RULES schedRule*) o
                     addPreSuf o execUpdate (db*,s*))
```

Here *member* ∗ $(db*, s*)$ $h*$ undertakes an equivalence test of $(db*, s*)$ with every element $(db*', s*')$ of $h*$ i.e. it calls *equiv* on the abstract relation extents in $(db*, s*)$ and $(db*', s*')$.

execSched∗ is initially called with the empty list and some initial abstract database state and schedule pair (db_0*, s_0*). It is easy to show that if *equiv* is precise and *execSched*$*$ terminates with a non-empty schedule, then so would *execSched* if invoked with any $(db_0, s_0) \in conc\ (db_0*, s_0*)$ i.e. the rules are definitely non-terminating for all database states and schedules $(db_0, s_0) \in conc\ (db_0*, s_0*)$. If *equiv* is not precise, but safe, and *execSched*$*$ terminates with a non-empty schedule, then we might as well halt the analysis since we can find out no information from it regarding the termination of any $(db_0, s_0) \in conc\ (db_0*, s_0*)$.

To illustrate, suppose we have a rule set consisting of a single rule:

```
_ecaRule 4 == (safeIncD,
               [lambda gs.exc safe blackList,
                lambda gs.inc safe recent],
               Imm);
```

This rule makes sure that `safe` doesn't contain any grid references that have been blacklisted, and also inserts into it any grid references that have recently been added to the database. Starting off with a transaction `[inc safe s]`, the abstract extent of `safe` during successive iterations of *execSched*$*$ are

iteration 1	$safe \cup s$
iteration 2	$((safe \cup s) - blackList) \cup recent$
iteration 3	$((((safe \cup s) - blackList) \cup recent) - blackList) \cup recent$

The expressions at iterations 2 and 3 are equivalent according to our *equiv* test (as are those for $safeIncD$ and $safeExcD$, not given). So we can deduce definite non-termination of this rule i.e. that there exist initial database states and schedules for which the rule execution will not terminate — consider for example the case that the same grid reference initially exists in all three relations $safe$, $blackList$ and $recent$.

8.3. Handling non-determinism

We give one more extension to the concrete and abstract semantics of Section 3, in this case to handle the non-determinism that results by dropping the assumption that no two rules have the same priority.

$RULES$ is now of type $[\{[RULE]\}]$ where $RULE == (Expr, [Expr], SMode)$. $RULES$ is ordered by the priority of the equivalence classes of rules. Each set within $RULES$ consists of the set of permutations of the rules in an equivalence class, a permutation being a list of rules. The clause $fold\ RULES\ schedRule$ in *execSched* is replaced by a function *schedRules* $RULES$ which generates a set of possible schedules for each of the possible current database states. *execUpdate* is similarly replaced by *map execUpdate* in order to apply the first expression of each schedule to its corresponding database state. *execSched* is initially invoked with a singleton set $\{(db_0, s_0)\}$ and returns the set of all possible final database states that might result:

```
execSched :: {(DBState,Schedule)} -> {(DBState,Schedule)}
execSched dbs ==
```

```
    if {s | (db,s) <- dbs} = {[]}
    then dbs
    else execSched (makeScheds o (schedRules RULES) o
                    addPreSufs o execUpdates dbs)

makeScheds            == map makeSched
addPreSufs            == map addPreSuf
execUpdates           == map execUpdate
schedRules rules dbPSS ==
    setmap (lambda dbpss.fold rules schedSet {dbpss}) dbPSS
schedSet dbPSS set    ==
    setmap (lambda list.map (fold list SchedRule) dbPSS) set
```

execSchedC and *execSched∗* are modified similarly:

```
execSchedC :: {{(DBState,Schedule)}} -> {{(DBState,Schedule)}}
execSchedC dbss =
    if {s | dbs <- dbss; (db,s) <- dbs} = {[]}
    then dbss
    else execSchedC ((map makeScheds) o (map (schedRules RULES)) o
                     (map addPreSufs) o (map execUpdates) dbss)

execSched* :: {(DBState*,Schedule*)} -> {(DBState*,Schedule*)}
execSched* dbs* ==
    if {s* | (db*,s*) <- dbs*} = {[]}
    then dbs*
    else execSched* (makeScheds o (schedRules* RULES) o
                     addPreSufs o execUpdates dbs*)

schedRules* rules dbPSS* ==
    setmap (lambda dbpss*.fold rules schedSet* {dbpss*}) dbPSS*
schedSet* dbPSS* set     ==
    setmap (lambda list.map (fold list SchedRule*) dbPSS*) set
```

To illustrate, suppose we have a rule set containing rules 1 and 2 from Section 7 above and another rule 5 whose priority is the same as that of rule 1:

```
_ecaRule 5 == (safeIncD, [lambda gs.inc log gs], Imm)
```

where `log` records which grid references have been safe at some time. If `[inc safe s]` is the initial abstract transaction, one execution thread of *execSched∗* is

$$(db*_0, [r_1, r_5]) \to (db*_1, [r_2, r_5]) \to (db*_2, [r_5]) \to (db*_3, [\,])$$

and the other is

$$(db*_0, [r_5, r_1]) \to (db*'_1, [r_1]) \to (db*'_2, [r_2]) \to (db*'_3, [\,])$$

where r_i denotes the action of rule i bound to the current abstract database state. In both cases execution terminates. The satisfiability tests that are performed are similar to the example in Section 7 for the rule set containing just rules 1 and 2. The situation is similar starting with the other initial abstract transaction, `[exc safe s]`.

We can also deduce that this rule set is *confluent* by showing that $db{*}_3$ is equivalent to $db{*}'_3$, using the *equiv* test. This result can be stated more generally: *we deduce rule confluence iff the abstract expressions representing all possible final database states are equivalent.*

9. Conclusions

In this paper we have shown how abstract interpretation can be used as a tool for rule termination analysis in active databases. We have done this by defining a concrete semantics, a concrete collecting semantics, and an abstract semantics for transaction execution. We have found the use of a functional formalism particularly appropriate for defining the concrete execution semantics because it emphasizes the modularity of the various components. This in turn assists in constructing the abstract version of the semantics and in formally proving the required correspondence between the abstract and concrete versions. In particular, we have established the correctness of the abstract semantics by using the well-understood framework of abstract interpretation and Galois connections.

A key contribution of the paper is our demonstration of how to formulate an appropriate abstract interpretation for active rule termination behaviour. This is a non-trivial application of abstract interpretation theory, due to the complexity of the domains involved, especially when compared with previous applications such as strictness analysis.

We have demonstrated the practical usefulness of our termination analysis approach by identifying a broad class of terminating rule sets for which previous graph-based methods give no information but for which our abstract interpretation algorithm concludes definite termination after a small number of iterations. These are rule sets in which one part of a conjunctive condition is turned off by the rule's action and only part of the conjunction is turned on again by other rules, or in which only one part of a disjunctive condition is turned off by the rule's action and other rules turn off the rest of the disjuncts.

We have also shown how more sophisticated approximations, such as intervals, can be used for more expressive rule languages. Incorporating such approximations into our framework is simply a matter of proving the safety of the approximations (criterion (2) in Section 4). Correctness of the abstract interpretation framework as a whole does not need to be redone from scratch.

Although the thrust of this paper has been towards termination analysis, our techniques are easily applicable to other properties of active rules such as reachability and confluence, and we gave an example of the latter in Section 8.3. This is because we are essentially concerned with developing characterisations of program state, an important building block for analysing program properties.

We are currently implementing a prototype rule analysis tool that incorporates the *execSched∗* algorithm, extended with the interval approximation discussed in Section 8.1. We will be using this tool to analyse the termination properties of real-life ECA rule sets from the area of Transport Studies.

For future work we plan to focus on:

(i) Incorporating monotonicity information into *execSched∗* in order to speed up the abstract execution (this is known as *widening*): we give some methods for identifying monotonic sub-executions of rules in (Bailey and Poulovassilis, 1998) and we plan to incorporate these ideas into the functional framework described here.

(ii) Reformulating other algorithms for rule termination analysis: our framework can provide a unifying basis for formally defining, verifying and comparing other algorithms for rule termination analysis.

(iii) Applying abstract interpretation to other aspects of active rules, such as condition optimisation, since evaluating conditions in an abstract domain may be cheaper than on the actual database.

Acknowledgments

This work has been supported by EPSRC grant no. GR/L 26872. We are grateful to the reviewers for their useful comments on an earlier version of this paper.

Appendix A

Abstract interpretation is a well-understood, semantics-based method for static analysis and has proven a useful tool in the analysis of imperative, functional and logic programs (Abramsky and Hankin, 1987; Cousot and Cousot, 1992; Marriott et al., 1994). The basic idea is to infer and collect information about a program's run-time properties and its operational aspects. Once the underlying semantics has been captured formally, a variety of analyses can be derived, as *approximations* to the semantics. A classic example is that of the "rule of signs". For instance, it is possible to conclude that the result of evaluating the expression -112×592 is negative, *without performing the multiplication*. This is done by applying the rule that a positive number multiplied by a negative one yields a negative result. More formally, each element in the "concrete" domain of integers $\mathcal{Z}$ is approximated by a description in the abstract domain $\{pos, neg, zero, any\}$. The multiplication operation $\times$ is then approximated by an abstract operation $\times^*$ using the rules $pos \times^* pos = pos$, $pos \times^* neg = neg$, $zero \times^* pos = zero$ and so forth.

To use the abstract interpretation framework then, we require two domains, a concrete one and an abstract one. The relationship between the two domains is formalised using a Galois connection. This consists of two functions

$$abs \; : \; P[D] \rightarrow D*$$

$$conc \; : \; D* \rightarrow P[D]$$

such that

(i) $\forall S \in P[D] \; . \; (conc \circ abs) \; S \sqsubseteq S$
(ii) $\forall d* \in D* \; . \; d* = (abs \circ conc) \; d*$

where $P[.]$ is a powerdomain construction over D [Sch86].

abs is called the abstraction function and *conc* the concretisation function. *abs* operates on the domain $P[D]$ rather than D, which ensures that the result of applying *conc* has the same type as the argument to *abs*. This elevation to powerdomains is tied to the concept of *collecting semantics* (Cousot and Cousot, 1992; Marriott et al., 1994). This is essentially an extension of the concrete semantics to collect the total information needed for the analysis task being undertaken.

Using *abs* and *conc*, we can define abstract counterparts for functions operating on the concrete domain. In particular, for any function $f : P[D] \rightarrow P[D]$, the function

$$f' = abs \circ f \circ conc$$

is an approximation function for f i.e.

$$conc \; (f' \; (abs \; S)) \sqsubseteq f \; S$$

This is easily shown to be so:

$$\begin{array}{lcll} conc \; (f' \; (abs \; S)) & = & conc \; (abs \; (f \; (conc \; (abs \; S)))) & \text{by the definition of } f' \\ & \sqsubseteq & conc \; (abs \; (f \; S)) & \text{by property (i) above} \\ & \sqsubseteq & f \; S & \text{by the same property} \end{array}$$

Moreover, this is the "best" approximation since, if $f = id$, then

$$\begin{array}{lcll} f' & = & abs \circ conc & \\ & = & id & \text{by property (ii) above} \end{array}$$

The relationship between the four functions $abs, conc, f$ and f' is illustrated in the commuting diagram in Figure 1.

For the rule of signs example mentioned above, $D = \mathcal{Z}$ and $D* = \{pos, neg, zero, any\}$ The abstraction function is

$$\begin{array}{lcl} abs \; S & = & pos, \; if \; \forall d \in S, \; d > 0 \\ & & neg, \; if \; \forall d \in S. \; d < 0 \\ & & zero, \; if \; \forall d \in S. \; d = 0 \\ & & any, \; otherwise \end{array}$$

and the concretisation function is

$$\begin{array}{lcl} conc \; pos & = & \{d \in \mathcal{Z} \mid d > 0\} \\ conc \; neg & = & \{d \in \mathcal{Z} \mid d < 0\} \\ conc \; any & = & \mathcal{Z} \\ conc \; zero & = & \{0\} \end{array}$$

$$\begin{tikzcd}
P[D] \arrow[rr, "abs"] & & D* \\
P[D] \arrow[u, "f"] & & D* \arrow[ll, "conc"] \arrow[u, "f'"']
\end{tikzcd}$$

Figure A.1. $f' = abs \circ f \circ conc$

Appendix B

We show by induction on i that

$$ds_i \;=\; conc\ (db*_i, s*_i)$$

where $(db*_i, s*_i)$ and ds_i are as defined in Section 5.3.

For the base case of $i = 0$, by definition we have that

$$ds_0 \;=\; conc\ (db*_0, s*_0)$$

Assume now that

$$ds_{i-1} \;=\; conc\ (db*_{i-1}, s*_{i-1})$$

Let

$$\begin{aligned}
f &= makeSched \circ (fold\ RULES\ schedRule*) \circ addPreSuf \circ execUpdate \\
g &= (map\ makeSched) \circ (fold\ RULES\ schedRuleC) \circ \\
&\quad (map\ addPreSuf) \circ (map\ execUpdate)
\end{aligned}$$

In order to prove the inductive case, we need

$$g \circ conc = conc \circ f$$

to hold. This is easily seen to be so since g collectively applies the same operations to each concrete database and schedule pair that f applies to the single abstract database and schedule pair. Thus

$$\begin{aligned}
ds_i &= g\ ds_{i-1} \\
&= g \circ conc\ (db*_{i-1}, s*_{i-1}) \\
&= conc \circ f\ (db*_{i-1}, s*_{i-1}) \\
&= conc\ (db*_i, s*_i)
\end{aligned}$$

as required.

Notes

1. We have somewhat simplified PFL's storage and update of bulk data here, to make for a clearer exposition of our rule termination analysis approach. The expressiveness of the language is not changed by these simplifications.
2. The need to distinguish between a relation's extent and a relation's name arises from the way that PFL handles updates by assigning a linear type to relation names but retaining an ordinary, non-linear, type for relation extents — see (Sutton and Small, 1995) for details.
3. In this paper we consider a subset of PFL's ECA component as described in (Reddi et al., 1995; Poulovassilis et al., 1996) in order to more easily illustrate the power of our termination analysis methods. The simplifications are that: (a) we combine the event and condition queries of ECA rules into a single event-condition query, (b) we do not consider user-defined events, aborts and abort-handling, (c) we do not record the transaction number and event number within the history and delta relations, and (d) we assume that all ECA rule actions are either inclusions into or exclusions from relations.

References

Abramsky, S. and Hankin, C. (eds.), Abstract Interpretation of Declarative Languages, Ellis Horwood, 1987.

Aiken, A., Widom, J. and Hellerstein, J. M., Static Analysis Techniques for Predicting the Behavior of Active Database Rules, ACM Transactions on Database Systems, 20, 1, pp 3–41, 1995.

Bailey, J., Crnogorac, L., Ramamohanarao, K. and Søndergaard, H., Abstract Interpretation of Active Rules and its use in Termination Analysis, Proc. 6th International Conference on Database Theory, Delphi, Greece, pp 188–202, Lecture Notes in Computer Science 1186, Springer-Verlag, 1997.

Bailey, J., Dong, G. and Ramamohanarao, K., Decidability and undecidability results for the termination problem of active database rules, Proc. 17th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Seattle, Washington, pp 264–273, 1998.

Bailey, J. and Poulovassilis, A., Termination Analysis for Active Database Rules - Techniques for Increased Precision, Technical report 98-06, Department of Computer Science, King's College London, 1998.

Baralis, E., Ceri, S. and Paraboschi, S., Improved Rule Analysis by Means of Triggering and Activation Graphs, Proc. 2nd Int. Workshop on Rules in Database Systems, pp 165–181, Lecture Notes in Computer Science 985, Springer-Verlag, 1995.

Ceri, S. and Fraternalli, P., Designing Database Applications with Objects and Rules: The IDEA Methodology, Addison-Wesley, 1997.

Chaudhuri, S. and Kolaitis, P., Can Datalog be approximated?, Journal of Computer and System Sciences, 55, 2, pp 355–369, 1997.

Coupaye, T. and Collet, C., Denotational Semantics for an Active Rule Execution Model, Proc. 2nd Int. Workshop on Rules in Database Systems, pp 36–50, Lecture Notes in Computer Science 985, Springer-Verlag, 1995.

Cousot, P. and Cousot, R., Abstract Interpretation Frameworks, Journal of Logic Programming, 13, 2&3, pp 103–179, 1992.

Libkin, L. and Wong, L., Incremental recomputation of recursive queries with nested sets and aggregate functions, Proc. 6th International Workshop on Database Programming Languages, Estes Park, Colorado, pp 222–238, Springer-Verlag LNCS 1369, 1997.

Marriott, K., Søndergaard, H. and N. D. Jones, Denotational Abstract Interpretation of Logic Programs, ACM Transactions on Programming Languages and Systems, 16, 3, pp 607–648, 1994.

Mycroft, A., Abstract Interpretation and Optimising Transformations for Applicative Programs, PhD Thesis, Department of Computer Science, University of Edinburgh, December 1981.

Peyton Jones, S., The Implementation of Functional Programming Languages, Prentice-Hall, 1987.

Poulovassilis, A., Reddi, S. and Small, C., A formal semantics for an active functional DBPL, Journal of Intelligent Information Systems, 7, 2, pp 151–172, 1996.

Poulovassilis, A. and Small, C., Investigation of algebraic query optimisation in database programming languages, Proc. 20th VLDB Conference, Santiago, pp 415–426, 1994.

Poulovassilis, A. and Small, C., Algebraic query optimisation in database programming languages, The VLDB Journal, 5, 2, pp 119–132, 1996.

Qian, X. and Wiederhold, G., Incremental recomputation of active relational expressions, IEEE Trans. on Knowledge and Data Engineering, 3, 3, pp 337–341, 1991.

Reddi, S., Poulovassilis, A. and Small, C., Extending a functional DBPL with ECA-rules, Proc. 2nd Int. Workshop on Rules in Database Systems, pp 101–115, Lecture Notes in Computer Science 985, Springer-Verlag, 1995.

Sagiv, Y. and Yannakakis, M., Equivalences among relational expressions with the union and difference operators, Journal of the ACM, 27, 4, pp 633–655, 1980.

Schmidt, D.A. , Denotational Semantics, Allyn and Bacon, 1986.

Søndergaard, H. and Sestoft, P. , Non-determinism in functional languages, The Computer Journal, 35, 5, 1992.

Sutton, D. and Small, C., Extending functional database languages to update completeness, Proc. 13th British National Conference on Databases, Lecture Notes in Computer Science 940, Springer-Verlag, pp 47–63, 1995.

Widom, J., A Denotational Semantics for the Starburst Production Rule Language, ACM SIGMOD Record, 21, 3, pp 4–9, 1992.

Zimmer, D., Meckenstock, A. and Unland, R., Using Petri Nets for Rule Termination Analysis, Proc. Workshop on Databases: Active and Real-Time 1996 (CIKM'96), Rockville, Maryland, 1996.

Contributing Authors

James Bailey received Bachelor of Engineering, Bachelor of Science and Ph.D. degrees in Computer Science from the University of Melbourne in 1993, 1994 and 1998 respectively. His doctoral research was in expressiveness and analysis of active database rules. He joined King's College London as a Research Associate in October 1997 to work on EPSRC project GR/L 26872 — "Formalisation and Optimization of Active Databases". He was promoted to Research Fellow in October 1998. His research interests are deductive and active databases, logic, and theoretical foundations of database languages.

Alexandra Poulovassilis received a B.A. in Mathematics from Cambridge University in 1982. Between 1983 and 1985 she was a Programmer/Analyst at IBM Greece. She obtained M.Sc. and Ph.D. degrees in Computer Science from Birkbeck College, University of London, in 1986 and 1990 respectively. Her doctoral research was in functional database languages. In 1989 she obtained a SERC Postdoctoral Fellowship to continue this work. She held this post at University College London until September 1991 when she joined King's College London as a Lecturer. She was appointed Senior Lecturer in September 1997. She has held a number of SERC/EPSRC research grants, most recently grant no. GR/L 26872 on "Formalisation and Optimization of Active Databases". Her research interests are database programming languages, query optimization, active databases, heterogeneous databases, and graph-based data models and languages. She is member of the BCS, ACM and IEEE Computer Society and became a Chartered Engineer (CEng) in 1992.