Language Constructs for Programming Active Databases*

Richard Hull and Dean Jacobs
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782 USA
{hull,jacobs}@pollux.usc.edu

"You cannot step twice into the same river; for fresh waters are ever flowing in upon you."
Heraclitus, circa. 500 B.C.

Abstract:

This paper presents database programming language constructs that can be used to realize a variety of different semantics for rule application in active database systems. The primary novel feature introduced is the "delayed update", or *delta*, which is a first-class value representing a set of proposed modifications to the underlying persistent store. Deltas can be created, inspected, and combined without committing to the given modifications. The utility of these concepts for expressing the semantics of active databases is demonstrated through a series of examples, including the presentation of the essential features of rule application in the AP5 system of USC/Information Sciences Institute and the Starburst Rule System being developed at IBM Almaden. Technical results concerning the simulatability of certain fundamental constructs by other fundamental constructs are also presented. The discussion is based on Heraclitus[Rel], an imperative language containing a relational calculus sublanguage and deltas.

1 Introduction

"Active" databases generally support the automatic triggering of updates as a response to user-requested or system-generated updates [M83]. Many active database systems, e.g., [CC$^+$90, Coh86, Coh89, MD89, H89, SdM88, SIG89, SJ$^+$90, WF90, ZH90], use a paradigm of *rules* to generate these automatic updates, in a manner reminiscent of expert systems. As discussed in [HJ90] and elsewhere, each of the systems described in the literature uses a different semantics for rule application. Some of these differences stem from the choice of underlying data model (e.g., relational or object-oriented), but the most crucial differences stem from choices concerning when rules should be fired (e.g., at transaction boundaries or within transactions), how they should be fired (e.g., in parallel or sequentially in some order), and how their effects should be combined (e.g., aborting on conflict or giving priority to insertions). This highlights the fact that the "knowledge" represented in both active and deductive databases stems from two distinct components: (a) the rule base and (b) the semantics for rule application. It appears that different rule-application semantics will sometimes be appropriate, even within a single database. This perspective is supported by the LOGRES system [CC$^+$90], in which users can choose, for each requested update, from a palate of six rule application semantics. It seems unlikely that a fixed collection of choices will suffice however, especially as active databases become increasingly sophisticated. For example, there has been recent interest in developing techniques for modularizing rules, e.g., by clustering them with classes in an object-oriented system (cf. [MP90]) or with different kinds of database transactions. It seems natural that designers will require different semantics for different kinds of clusters.

This paper presents database programming language constructs that can be used to directly realize a variety of different semantics for rule application. The primary novel feature introduced is the "delayed update", or *delta*, which is a first-class value representing a set of proposed modifications to the underlying persistent store. Deltas can be created, inspected, and combined without committing to the given modifications. The utility of these concepts for expressing the semantics of active databases is demonstrated through a series of examples, including the presentation of the essential features of rule application in the AP5 system [Coh87, Coh86, Coh89] of USC/Information Sciences Institute and the Starburst Rule System described in [WF90, CW90], currently under development at IBM Almaden. This is a contribution in and of itself because for both of these systems, rule application is specified only by informal, natural-language descriptions (see [Coh87, CW90]). We also show how these constructs can be

*This research was supported in part by a grant from AT&T. The first author was supported in part by NSF grant IRI-8719875, and by the Defense Advanced Research Projects Agency under DARPA grant MDA903-81-C-0335. The second author supported in part by the German Academic Exchange Service. This research was in part performed while both authors were visiting the Technische Universität Berlin.

used to express the semantics of some deductive database systems.

Our discussions are based on the database programming language Heraclitus[Rel] [JH91], currently in the initial stages of implementation at USC and USC/ISI. The basic Heraclitus feature of deltas can be realized for a variety of underlying programming paradigms and data models. Heraclitus[Rel] is a simple imperative language using the relational model and a calculus sublanguage, much along the lines of PASCAL/R [Sch77] and AP5. The short term goal of the current implementation effort is to provide a testbed for experimenting with different rule application semantics. In the long term, we hope to develop an implementation that is efficient enough to be directly used in practical applications.

Heraclitus also serves as a basis for the theoretical analysis of alternative rule application semantics and their interaction with, e.g., integrity constraints, parallelism, heterogeneity, etc. In this paper, we give a small taste of this kind of analysis by examining the relative simulatability of two approaches to accessing deltas. The first approach, called "peeking", permits the programmer to directly inspect the proposed modifications contained in a delta; this approach is used in the Starburst system, among others. The second approach, called "hypothesizing", permits the programmer to query the hypothetical database that would be obtained by applying a delta to the actual current database; this approach is used in the AP5 system, among others. In the general context of Heraclitus, peeking and hypothesizing can simulate each other. However, we exhibit a set of restrictions on Heraclitus programs under which hypothesizing cannot simulate peeking.

Although not considered here, Heraclitus also appears useful in the context of hypothetical reasoning, truth maintenance systems, and in connection with specifying implementation strategies for database transaction processing.

In Section 2 we present an overview of the kernel of Heraclitus[Rel]. In Section 3 we show how Heraclitus can be used to express rule application semantics, for both active and deductive databases. In Section 4 we consider the AP5 semantics, and in Section 5 we consider the semantics of the Starburst system. Section 6 considers peeking and hypothesizing. Brief conclusions are offered in Section 7.

2 Overview of Heraclitus[Rel]

Heraclitus[Rel], hereafter referred to more simply as Heraclitus, is a simple imperative language with the following features.

- It is statically-typed,[1] i.e. all possible type errors are detected at compile-time, modulo the issue of interfacing with the persistent store. The types of program variables must be explicitly declared by the user and the types of quantified variables are automatically inferred by the system.
- It supports relation types and relation variables. Relation variables may persist or be declared locally by the programmer.
- It supports a type `delta`, whose values represent proposed insertions into, and deletions from, the values of (persistent and transient) relation variables. Delta variables may be declared by the programmer.
- It has a relational calculus sublanguage that can be used to construct relation and delta values. This sublanguage supports (a) quantified variables which are not range-restricted, (b) variables and function symbols which are bound "outside" of calculus formulas, and (c) the use of deltas within formulas. This appears to be richer than the calculus sublanguages supported in other relational database programming languages (e.g., PASCAL/R [Sch77]), and we have had to extend the usual notions of "safety" and the tranformations used to translate safe formulas into relational algebra expressions (see, e.g., [GT89]). (Heraclitus also includes explicit algebraic operators for union, intersection and difference.)

Delta values are generated by evaluating delta expressions. The atomic delta expression <+R($e_1, \ldots, e_n$)> produces a delta which calls for the value of tuple expression $(e_1, \ldots, e_n)$ to be inserted into the value of relation variable R. (Relations are viewed as sets; if $(e_1, \ldots, e_n)$ is in the current value for R then applying <+R($e_1, \ldots, e_n$)> causes no change to R.) Similarly, the atomic delta expression <-R($e_1, \ldots, e_n$)> produces a delta which calls for the value of tuple expression $(e_1, \ldots, e_n)$ to be deleted from the value of relation variable R. Two binary operators for combining deltas are provided: *Merge*, denoted &, forms the "union" of two deltas, but produces the special delta *fail* if conflicting updates are proposed. *Smash*, denoted !, resolves conflicting updates in favor of the second delta. For example,

- <+R(1)> & <+R(2)> produces a delta that calls for (1) and (2) to be inserted into R.
- <+R(1)> & <-R(1)> produces *fail*.
- <+R(1)> ! <+R(2)> produces a delta that calls for (1) and (2) to be inserted into R.
- <+R(1)> ! <-R(1)> produces a delta that calls for (1) to be deleted from R.

In the context of combining the output of rule applications, merge implements a semantics based on "accumulation" of requested updates, while smash implements a semantics based on "overwriting" (see [HJ90]). In general, a delta value may refer to more than one relation variable.

Let DB be the database state corresponding to the binding of values to all relation variables in the scope at some point in a Heraclitus program. If Δ is a delta value, then *apply*(Δ, DB) is the database state corresponding to the application of the modifications requested by Δ to DB. For a delta expression δ, *eval*(δ, DB) is the value of δ under the bindings specified by DB (and the bindings for delta

[1] In this article we sometimes relax this discipline, permitting for succinctness the type *tuple* of arbitrary signature.

and simple-type variables, which we suppress here). Finally, the command apply δ has the effect of reassigning all relation variables according to $apply(eval(\delta, DB), DB)$.

Heraclitus provides two ways of accessing deltas without committing to the proposed modifications, as in the command apply introduced above. The first way, called "peeking", permits the programmer to directly inspect the proposed modifications. The boolean expression δ_1 in δ_2 produces true iff the value of delta expression δ_1 is a "subset" of the value of delta expression δ_2. For example, `<-R(1)> in current` tests whether the value of delta variable `current` calls for the tuple (1) to be deleted from the value of relation variable R. The second way, called "hypothesizing", permits the programmer to query the hypothetical database obtained by applying a delta to the actual current database. In particular, the expression E when δ evaluates expression E in the database obtained by applying the value of delta expression δ to the current state.

There are several different quantified expressions in Heraclitus[Rel], all of which have the basic form

$$Q\{x_1, \ldots, x_n \mid \Phi \mid E\}$$

where Q is the name of the quantifier (this includes at present **forall**, **exists**, union, the, **merge**, and also arithmetic aggregates such as **sum**); $x_1, \ldots, x_n$ introduces quantified variables, Φ is a formula analogous to the ones in the relational calculus, and E is an expression. It is useful to view such "three-pronged" expressions in terms of operations on multi-sets, although this type does not appear formally in the language. Conceptually, the body of a quantified expression represents the multi-set containing the value of E for each set of bindings for $x_1, \ldots, x_n$ that satisfies Φ. Each quantifier corresponds to a way of collapsing a multi-set of values of a particular type. As an example using the aggregate sum quantifier,

```
sum{x | 0<x<6 | 2*x}
```

evaluates to 30. Note that **sum** is the extension of the binary operation + on integers to multi-sets of integers. The quantifiers in Heraclitus[Rel] generally have this property: **forall** and **exists** are the extensions of and and or on booleans, union is the extension of binary union + on relations, and **merge** is the extension of binary merge & on deltas.

Most of the quantifiers have special abbreviated forms which correspond to their common usage. In particular, union$\{ x_1,\ldots,x_n \mid \Phi \}$ abbreviates union$\{ x_1,\ldots,x_n \mid \Phi \mid \langle x_1,\ldots,x_n\rangle\}$; the body of the latter expression produces a set of singleton relations, which are then unioned. Also, (**forall** $x_1,\ldots,x_n$. Φ) abbreviates **forall**$\{ x_1,\ldots,x_n \mid$ **true** $\mid \Phi \}$ and (**exists** $x_1,\ldots,x_n$. Φ) abbreviates **exists**$\{ x_1,\ldots,x_n \mid$ **true** $\mid \Phi \}$. Note that these abbreviated forms cannot always be used: as a rather contrived example

```
forall{x | 0<x<6 | (1/x)<y}
```

is not equivalent to

```
(forall x . not(0<x<6) or (1/x)<y)
```

because of the possibility of division by zero. Thus, these quantifiers incorporate the conventional programming language notions of "conditional and" cand and "conditional or" cor.

We now define the semantics of smash more precisely. Given two delta values Δ_1 and Δ_2, the *smash* of Δ_1 and Δ_2 is that delta value Δ such that for all states DB, $apply(\Delta, DB) = apply(\Delta_2, apply(\Delta_1, DB))$. More generally, given delta expressions δ_1 and δ_2 and state DB, then $\Delta = eval(\delta_1 ! \delta_2, DB)$ has the property $apply(\Delta, DB) = apply(\Delta_2, apply(\Delta_1, DB))$, where $\Delta_i = eval(\delta_i, DB)$ for $i \in \{1,2\}$. An expression equivalent to $\delta_1 ! \delta_2$ can be obtained using merge and peeking; specifically, $\delta_1 ! \delta_2$ is equivalent to the merge over all relation variables R (occuring in the relevant scope) of **merge**$\{ t \mid$ (<+R(t)> in δ_1 and not <-R(t)> in δ_2) or <+R(t)> in δ_2 $\mid$ <+R(t)> $\}$ merged with **merge**$\{ t \mid$ (<-R(t)> in δ_1 and not <+R(t)> in δ_2) or <-R(t)> in δ_2 $\mid$ <-R(t)> $\}$. From this example it is clear that other operators for combining deltas, e.g., to give precedence for insertions as in LOGRES [CC+90], can be defined in Heraclitus.

The interaction of **when** and ! is interesting: for all expressions E and delta expressions δ_1 and δ_2, (E **when** δ_1) **when** δ_2 is equivalent to E **when** (δ_2 ! (δ_1 **when** δ_2)).

3 Expressing Rule Application Semantics

In this section we illustrate by simple examples the spirit of how the constructs of Heraclitus can be used to specify the semantics of rule application in active and deductive database systems.

In most active database systems, a rule consists of a *trigger* (or condition) that controls when the rule should be fired and a *body* which specifies the modifications that are contributed when firing occurs. The trigger and the body will generally be able to access the original (most recently committed) state of the database, as well as various intermediate states proposed by the user and other rules. In Heraclitus, a rule can be represented as a function that takes deltas, representing intermediate states, as input and produces deltas, representing contributed modifications, as results. For example, the rule

```
function rule(curr:delta):delta
 return merge{ x | R(x) and (not R(x) when curr)
              | <-S(x)> }
```

can be applied during the processing of a transaction to propagate deletions from R to S. This same rule can be written in (set-oriented [WF90, CW90]) trigger/body form as two functions.

```
function trigger(curr:delta):rel(int)
 return union{ x | R(x) and (not R(x) when curr) }
```

```
function body(T:rel(int)):delta
 return merge{ x | T(x) | <-S(x)> }
```

A particular semantics for rules can be expressed in Heraclitus as a procedure, referred to as a (*rule application*) *template*, which controls how functions such as the ones above are called. In order to facilitate the manipulation of rules, we introduce the notion of *indexed-families of functions*, as the first example below shows. This can be viewed as a shorthand for a function taking as input an integer, and containing a case statement which maps the input integer to the appropriate code fragment. An alternative would be to permit explicit arrays of rules, but this would entail elevating procedures to being first-class citizens, which would distract us from the main focus of this article.

We begin with some simple examples involving graphs, represented using two unary relations, root(string) and part(string), and one binary relation, PS(string, string). Intuitively, part holds (names of) parts, PS holds part-subpart relationships, and root holds those parts which serve as roots for the part-subpart graph.

We consider four constraints on instances of this schema:

(a) All strings occurring in root occur in part

(b) All strings occurring in PS occur in part

(c) Each string in part is reachable from a string in root via a path in PS.

(d) PS is a directed acyclic graph (i.e., has no directed cycles).

The first series of examples focus on sets of rules which maintain these constraints in the presence of deletions from one or more of the three relations. (Here rules 1,2 and 3 maintain constraints a,b and c, respectively.) Under the precedence rules of Heraclitus, a when connective is grouped with the smallest complete subformula preceding it.

```
function rule@1(curr:delta):delta
 return merge{x | root(x) and
                 (not part(x) when curr)
                | <-root(x)> }

function rule@2(curr:delta):delta
 return merge{x,y | PS(x,y) and ((not part(x) or
                        not part(y)) when curr)
                | <-PS(x,y)> }

function rule@3(curr:delta):delta
 return merge{x | part(x) and ((not root(x) and
                 forall y.not PS(y,x)) when curr)
                | <-part(x)> }
```

The following template, which is similar to the template for applying consistency rules in AP5 (see Section 3), may be used with these rules. We assume for the following template that the user-proposed database update is passed to the rule system by the parameter prop (which consists entirely of deletions). Execution consists in repeated parallel application of the rules, with a merging of intermediate results, until a fixpoint is reached, i.e., no further changes occur. Finally, this fixpoint is applied to the database. (We use dec-in-enddec to specify a set of declarations and their scope. Constant, function and procedure declarations are identified by keywords; unspecified declarations declare variables.)

```
procedure maintain_constraints(prop:delta)
dec next:int,
    prev,curr:delta in
 curr := prop;
 repeat
  prev := curr;
  curr := curr & merge{i | 1<=i<=3 | rule@i(curr)}
 until curr = prev endrepeat;
 apply curr
enddec
```

During each execution of the loop, the current value of curr is merged with the outputs of rule@i(curr) for i in $\{1,2,3\}$. The loop is executed until a fixpoint is reached. It can be shown that for any input instance satisfying the four constraints listed above and delta prop consisting exclusively of deletions, that execution of maintain_constraints will yield the unique maximal instance contained in the initial instance such that the tuples "deleted" by prop are absent, and such that the constraints are satisfied.

Note that the command assigning curr in the above loop has the same semantics as

```
dec temp:delta in
 temp := empty_delta;
 for i := 1 to 3 do
  temp := temp & rule@i(curr) endfor;
 curr := curr & temp
enddec
```

In this case, the rules are computed sequentially, all in the context of curr. The output of the rules is held in temp, which is merged with curr only after all rules have been used.

We now present a variation of rule@3, which has the same impact but which does not produce redundant deltas. This uses two deltas as input, one corresponding to the "current" delta, and the other corresponding to the delta computed most recently before that one during rule application. It also uses peeking, i.e., explicit tests of membership in deltas using the connective in.

```
function peekrule@3(prev,curr:delta):delta
 return merge{ x | part(x) when curr and
                  <-root(x)> in  curr and
                  forall y.(PS(y,x) when prev ->
                          <-PS(y,x)> in curr)
                | <-part(x)> }
```

Assuming that analogs of rule@1 and rule@2 using prev and curr are also specified, the following rule-application template will have the same effect as maintain_constraints.

```
procedure peek_maintain_constraints(prop:delta)
```

```
dec next:int,
    prev,curr:delta in
 curr := prop;
 repeat
  (prev,curr) := (curr,curr &
    merge{ i | 1<=i<=3 | peekrule@i(prev,curr)})
 until curr = prev endrepeat;
 apply curr
enddec
```

We now turn to a family of rules and a rule application template which will accomplish a general form of garbage collection. We assume two unary relations root and node and the binary relation link (all over type string). We also assume that the initial database satisfies the following constraints:

(a) All strings occurring in root occur in node

(b) All strings occurring in link occur in node

(c) Each string in node is reachable from a string in root via a directed path in link.

In this example, the rule template gc_template we exhibit shall be a function from deltas to deltas, such that if Δ is an arbitrary set of insertions and deletions on a database instance DB, then *apply*(gc_template(Δ), DB) will be the result of garbage collection on *apply*(Δ, DB). gc_template will not have side-effects on the database, so the entire computation can be rolled back if desired.

The variable OK is declared in the template to range over unary relations of strings, and is initialized to be empty. The template for rule application enforces a prioritization of the rules – the first group of rules (numbers 1 and 2) "determines" the set of nodes that are still connected to a root after the user input delta prop has been applied to the database, and places them into the temporary relation OK. The second group of rules (3, 4 and 5) uses this information to do the garbage collection. We first present the rules:

```
function gc_rule@1(curr:delta):delta
 return merge{ x | (root(x) and node(x)) when curr
               | <+OK(x)> }

function gc_rule@2(curr:delta):delta
 return merge{ y | exist x.(OK(x) and node(y) and
                            link(x,y)) when curr
               | <+OK(y)> }

function gc_rule@3(curr:delta):delta
 return merge{ y | root(y) and not OK(y) when curr
               | <-root(y)> }

function gc_rule@4(curr:delta):delta
 return merge{ y | node(y) and not OK(y) when curr
               | <-node(y)> }

function gc_rule@5(curr:delta):delta
 return merge{ x,y | link(x,y) and (not OK(x) or
                            not OK(y)) when curr
                 | <-link(x,y)> }
```

The following function is used to compute the impact of a cluster of rules, using accumulation. This function is analogous to maintain_constraints, but restricting its attention to rules with indices in X.

```
function gc_no_change(curr:delta,X:rel(int)):delta
dec next:int,
    prev:delta in
 repeat
  (prev,curr) := (curr,curr &
         merge{ i:int | X(i) | gc_rule@i(curr) })
 until curr = prev endrepeat;
 return curr
enddec
```

The full template for garbage collection is now given. Note that the functions given above are viewed as part of the declaration part of this procedure.

```
function gc_template(prop:delta):delta
dec temp:delta,
    OK:rel(string),
    % ... declarations for gc_rules, gc_no_change
    in
 OK := empty;
 temp := gc_no_change(prop, <1>+<2>);
 temp := temp!gc_no_change(temp, <3>+<4>+<5>);
 return temp
enddec
```

In the procedure gc_template we use notation for explicitly building relations (e.g., <1>+<2>). In Heraclitus, $<x_1, \ldots, x_n>$ denotes an n-ary relation holding the single tuple $(x_1, \ldots, x_n)$, and + denotes relational union.

Let DB denote the initial database instance. When gc_template is called on prop, the first action is to compute the set of OK nodes using gc_no_change(prop, <1>+<2>). Note that this delta contains no modifications to the persistent database variables. The next step is to compute the modifications which correspond to garbage collection that should be made to *apply*(*eval*(prop, DB), DB). This is accomplished by calling gc_no_change(temp, <3>+<4>+<5>).

The second assignment of gc_template computes the smash of the input delta prop and the output of gc_no_change(temp). When the value of temp is returned, all atomic deltas involving OK are removed, because OK is declared locally within the procedure.

This example is reminiscent of the LOGRES system [CC+90], which permits the sequential application of different rule "modules". The example also embodies some of the spirit of the theoretical language DATALOG$^{*\neg}$ [AV88, AS90], a variant of DATALOG in which rule heads can be positive or negative. Unlike DATALOG$^{*\neg}$, which supports either inflationary semantics and a semantics based on nondeterministic application of rules, this example uses a semantics reminiscent of stratified logic programming.

In the next example, we present a template that would be useful if rule actions had associated costs.[2] For example, suppose that in a business, based on certain conditions, rules will be fired in order to remedy problems, (e.g., if sales volume is too low then increase advertising; or lower prices by 5%). Suppose further that the rules are clustered, with the remedies proposed by some clusters being more "costly" than others; the more costly ones should be invoked only if the cheaper clusters are unable to remedy the problem. The following template assumes that there are c rule clusters, ordered by increasing cost, and attempts to find the cheapest solution to the current status of the database:

```
procedure apply_cheapest_solution
dec attempt:delta,
    constant c:int,
    % ... declarations for functions
    in
 for i := 1 to c do
  attempt := apply_rules(i);
  if satisfies_constraints(attempt)
   then apply attempt; return endif
 endfor
 print_message('no cluster offers a solution')
enddec
```

We conclude this section by indicating how Heraclitus can be used to express a popular semantics for rule application in deductive databases, namely stratified DATALOG$^\neg$ (e.g., see [Min88]). Under this approach, a set of "exensional" database relations is assumed, and a set of rules is used to populate a disjoint set of "intentional" database relations, which are initially assumed to be empty). In the example, we follow the usual convention, and do not materialize the contents of the intentional relations.

DATALOG$^\neg$ rules can be represented in Heraclitus by functions of the form

```
rule@i(curr:delta):delta
 return merge{ x_1,...,x_n | Φ when curr
                           | <+R(x_1,...,x_n)> }
```

where Φ is an existentially quantified conjunction of positive and negative literals.

The following function produces a delta corresponding to the effect of applying the set of rules whose indexes occur in the input relation:

```
function apply_rule_set(X:rel(int),
                              prop:delta):delta
dec prev,curr:delta in
 curr := prop;
 repeat
  (prev,curr) :=
     (curr,curr & merge{i | X(i) | rule@i(curr)})
  until prev = curr
 endrepeat;
 return curr
enddec
```

[2]The authors thank Serge Abiteboul for suggesting this example and the following one.

Suppose now that the rule base is stratified with n levels, and let LEVEL_i hold a unary relation containing the indices of the rules at level i. We now define:

```
function apply_all_levels:delta
dec curr:delta,
    % ... declaration for apply_rule_set
    in
 curr := empty-delta;
 for i := 1 to n do
  curr := apply_rule_set(LEVEL_i,curr) endfor;
 return curr
enddec
```

The expression `Q when apply_all_levels` evaluates to the value of query `Q` in the deductive database.

4 The semantics of AP5

In this section we specify the core of the semantics of rule application in the AP5 system. AP5 [Coh87, Coh86, Coh89] is a database programming language which extends LISP, and supports virtual memory databases and a transaction facility. It was implemented over five years ago at the USC/Information Sciences Institute, and has been in continuous use since then. AP5 was initially developed in connection with software specification and transformation, and has also been used to support office management functions and research on heterogeneous databases.

AP5 distinguishes two kinds of rules: *Consistency* rules are intended to be used to perform repairs of constraint violation. Speaking informally, the semantics of consistency rule application is based on accumulation, and the set of all consistency rules are fired until further applications yield no change. (If inconsistent updates arise, or a rule with no specified repair is triggered, then a rollback to the last commit is performed.) *Automation* rules may call for more substantial actions, including overwriting of previously requested modifications, and side-effects outside of the database. There are different semantics for applying the two kinds of rules: the application of automation rules forms an outer loop which calls for application of consistency rules as an inner loop.

Both kinds of rules are triggered on the basis of an "old" state and a "new" state; there is no peeking in AP5. At the beginning, the "old" state is the initial state of the database, and the "new" state is the result of applying the user proposed delta prop to that state. As computation progresses, the underlying database state is modified, and both "old" and "new" state may take on new values. In particular, the "old" always refers to the value that the database actually has, and "new" refers to the result of applying the delta currently being considered (typically denoted 'curr') to that state.

In the examples below, we assume that each consistency rule takes as input a delta corresponding to the "new" state, and returns a delta (which will ultimately be merged with the proposed one). The rule's and gc_rule's given above have the correct form to be used here as consistency rules. (AP5 provides different conventions for specifying consistency rules: conditions there permit the keyword 'start', where **start** φ holds if φ is true in the new state and false in the old one. Also, unqualified formulas are interpreted over the new state, as opposed to the old one as done here.)

The template for applying consistency rules, on input delta prop, is based on repeated parallel application of the rules, with a merging of intermediate results, until a fixpoint or inconsistency[3] is reached. Assuming that there are r consistency rules with names `consist_rule@i` for $i \in [1..r]$, this is captured by the following function. (This function differs from `maintain_constraints` in two ways: first, it returns the final delta, rather than applying it; and second, it may return the special delta `fail`, denoting inconsistency of the user requested update with the effect of the consistency rules.)

```
function consist(prop:delta):delta
dec curr,prev:delta,
    constant r:integer in
 curr := prop;
 repeat
  (prev,curr) := (curr,curr &
       merge{i | 1<=i<=r | consist_rule@i(curr)})
  until curr = prev or curr = fail
  endrepeat;
 return curr
enddec
```

If this function returns a delta not equal to *fail*, then the delta is eventually applied to the database. If this procedure returns the delta *fail* the system does not modify the database. (At present we view *fail* as carrying no additional information; however, a semantics can be developed in which *fail* carries with it information, e.g., about why the fail occurred.)

The actions of *automation* rules in AP5 can be arbitrary programs, possibly with side-effects outside of the database. Automation rules are triggered on the basis of the output of the function **consist**. The rule actions are applied in a nondeterministic order. Since these are arbitrary actions, they may themselves modify the database, thus invoking the rule-application module of AP5 recursively. Importantly, automation rules can do database modifications which consistency rules alone cannot do. For example, automation rules can alone simulate the garbage collection template of the previous section, whereas consistency rules alone cannot. This is because the application of consistency rules cannot "undo" anything which the user has requested.

We view automation rules as having two components, a "trigger" and an "action". Following the spirit of AP5 automation rules, the trigger returns a relation, but the action is specified in terms of single tuples. To provide an example, we use relations: `stud(string,string)`, holding student names and majors; `GPA(string,real)`, which give student names and their GPAs; `Dlist(string)` holding the names of students on the Dean's list; and `Dcount(string,int)`, which will give for each major the number of students having that major on the Dean's list.

[3]We simulate each rule with no specified repair by a rule which, when triggered, introduces `<+R(t)> & <-R(t)>` for some R and t.

The rules 1 and 2 below implement the policy that a student is placed on the Dean's list if s/he obtains a GPA of at least 3.8, but is removed from the Dean's list if the GPA falls below a 3.6. These rules also send a message to the (un)fortunate student. Rules 3 and 4 maintain the relation Dcount. The procedure **apply_rules**, invoked by the rule-actions here, has the effect of applying the AP5 rule system, and will be specified shortly. (The "de-setting" quantifier the in **action@3** returns the element of a singleton set, and is undefined otherwise.)

```
function trigger@1(curr:delta):rel(string,real)
 return union{x,y | not Dlist(x) and y>=3.8 and
                          (GPA(x,y) when curr)}

procedure action@1(t:(string,real))
dec output:delta in
 send_to(t.1, 'We are happy to inform
                              you ...',t.2,'...');
 output := <+Dlist(t.1)>;
 apply_rules(output)
enddec

function trigger@2(curr:delta):rel(string,real)
 return union{x,y | Dlist(x) and y < 3.6 and
                          (GPA(x,y) when curr)}

procedure action@2(t:(string,real))
dec output:delta in
 send_to(t.1, 'We regret informing
                              you ...',t.2,'...');
 output := <-Dlist(t.1)>
 apply_rules(output)
enddec

function trigger@3(curr:delta):rel(string,string)
 return union{ x,m | not Dlist(x) and
           (stud(x,m) and Dlist(x)) when curr}

procedure action@3(t:(string,string))
dec output:delta,
    i:int in
 i := the{ i | Dcount(t.2,i)}
 output := <-Dcount(t.2,i)> & <+Dcount(t.2,i+1)>;
 apply-rules(output)
enddec

function trigger@4(curr:delta):rel(string,string)
 return union{ x,m | Dlist(x) and stud(x,m) and
                        not Dlist(x) when curr}

procedure action@4(t:(string,string))
dec output:delta
    i:int in
```

```
 i := the{ i | Dcount(t.2,i)}
 output := <-Dcount(t.2,i)> & <+Dcount(t.2,i-1)>;
 apply-rules(output)
enddec
```

Note that if the user requests the update <+Dlist('Joe')> and Joe has a GPA of 2.1, then only rule 3 will be be fired. In particular, then, the presence of rule 2 does not enforce an integrity constraint that no student can have a GPA below 3.6 and be on the Dean's list.

We can now state (the central core[4]) of the semantics of AP5 rules application. We assume that the automation rules are numbered from 1 to r. Also, in this code we step outside of the current capabilities of Heraclitus by using a relational variable which ranges over sets of tuples whose second coordinates which are themselves tuples, and have unspecified arities and signatures. In AP5 the order of firing of applicable automation rules is nondeterministically selected; we assume that a function select(rel(int,tuple)):(int,tuple) is defined to accomplish this.

```
procedure apply_rules(prop:delta)
dec fix:delta;
    X:rel(int,tuple),
    constant r:int in
 fix := consist(prop);
 if fix = fail then return;
    % abort this procedure call if the proposed
    % update cannot be ''fixed'' by the
    % consistency rules
 X := { i,t | 1<=i<=r | t in trigger@i(fix) };
    % compute the ''trigger set'' before applying
    % fix to the database; X now holds
    % rule-identifier/witness-tuple pairs, giving
    % a list of all triggered rules and
    % (intuitively) all tuples responsible for
    % triggering the rule.
 apply fix;
 while X != empty_reln do
  (i,t) = select(X);
  X =: X - <i,t>;
  action@i(t)
    % recursion results because actions can have
    % calls to apply_rules
 endwhile
enddec
```

The cycle of rule application begins with a user requested delta prop. The consistency rules are applied to obtain fix = consist(prop). If fix = fail, then the database is left unchanged. Otherwise, the automation rules are considered with the "new" database equal to the result of applying fix to the initial database. A relation X is created and populated with pairs of (labels of) commands corresponding to the actions of triggered automation rules, and "witness" tuples that triggered them. Then fix is applied to the database. Processing now continues by considering each separate command called for in X as if it where a user generated request. Execution terminates when the initially created set is empty and the processing of its last pair has terminated.

[4]The AP5 system incorporates a number of special kinds of rules, each with their own semantics for application [Coh87].

5 The Starburst Semantics

In this section we use Heraclitus to specify the (core of the) semantics of the Starburst Rule System, an active database system being developed at IBM Almaden [WF90, CW90]. We present here the semantics as described in [CW90].

Some of the philosophical foundations underlying the Starburst semantics for rule applications are significantly different than those for AP5. These include

(a) partitioning the test for whether a rule should be applied into two parts: a "trigger", which is expressed in a restricted language and testable deep inside the database implementation; and a "condition" (expressed in an SQL extension), which is used subsequently to determine if a rule should really be applied. It turns out that the trigger is based on properties of a specific delta, while the condition and action are determined by that delta and the "current" state.

(b) the use of peeking into a delta, instead of hypothesizing.

(c) maintaining a sequence of states, corresponding to the sequence of successful rule applications, and triggering different rules according to different choices of "old" and "current" states. (In our simulation, we think in terms of a sequence of deltas rather than states.) The intuition of [CW90] is that when a rule action is executed, then the rule has completely resolved the problem which lead to its action execution. Thus, future consideration of the rule should be based entirely on modifications to the database which occurred *after* the rule firing. As a result, building rules with the intension of having them fire recursively might be more cumbersome in the Starburst semantics.

(d) in the Starburst semantics each tuple has an "object identifier" (OID); and insertion, deletion and modification are considered. In this presentation, tuples do not have OIDs, and tuple modification is not explicitly supported. (These features can be simulated in Heraclitus.)

Following the spirit of the Starburst framework, we assume that rules for this semantics have three components, a *trigger*, an *condition*, and an *action*.

As a simple example of this, we rewrite one of rules of [CW90] (the optimized version of rule 1c). The example of that paper concerns setting up electrical networks between power stations and users; one aspect of the problem focuses on the tubes used to house the wires. In their example, they assume a 4-ary relation tube which gives a tube-id, the source and destination of the tube, and its (tube) type; and also a 3-ary relation tube_type, whose tuples hold a (tube) type, a boolean indicating whether this type of tube is "protected" or not, and the diameter of a cross-section of this type of tube. The following rule corrects the situation where a tuple is inserted into tube but the type value of the tuple does not appear in tube_type. (In the formulas below, the symbol '_' is used in coordinate positions to denote a distinct variable which is existentially quantified immediately outside of the atom in which the symbol occurs.)

```
function trig@1c(change:delta):bool
 return exists tid.<+tube(tid,_,_,_)> in change

function cond@1c(change,curr:delta):bool
 return exists tid,type.
          (<+tube(tid,_,_,type)> in change and
               not tube_type(type,_,_) when curr)

function action@1c(change,curr:delta):delta
 return merge{ tid,fr,to,type
   | <-tube(tid,fr,to,type)> in change
            and not tube_type(type,_,_) when curr
   | <-tube(tid,fr,to,type)> &
            <+tube(tid,fr,to,default_tube_type> }
```

Here default_tube_type is a variable which is defined external to the functions given here.

Because of the overlap of computation between the condition and action here (which seems typical of consistency rules arising in the context of the [CW90] framework), it is convenient to view the condition as a function mapping to relations, and to obtain the boolean information by testing whether the relation is empty. We therefore rewrite the above as:

```
function cond@1c(change,curr:delta):
                              rel(string,string)
 return union{ tid,type
       | <+tube(tid,_,_,type)> in change and
              not tube_type(type,_,_) when curr}

function action@1c(curr:delta,
             witnesses:rel(string,string)):delta
 return merge{ tid,fr,to,type
       | (witnesses(tid,type) and
                 tube(tid,fr,to,type)) when curr
       | <-tube(tid,fr,to,type)> &
            <+tube(tid,fr,to,default_tube_type>}
```

Let us assume now that there are r rules in the rulebase specified by groups of indexed functions. For simplicity of exposition, we again step outside of the strong typing of Heraclitus, and specify the rule components in terms of relations of unspecified arities and signatures.

```
function trig@i(change:delta):bool
function cond@i(change,curr:delta):rel
function action@i(change,curr:delta,
                             witnesses:rel):delta
```

In the Starburst semantics, rules are fired and "applied" to the underlying database in sequence. As noted above,

Δ_0	=	the empty delta
Δ_1	=	user proposal
Δ_2	=	...
$\vdots$	$\vdots$	$\vdots$
Δ_i		suppose rules 3,5 and 8 triggered at ith step; the condition of rule 3 was evaluated and false; the condition of rule 5 was evaluated and true; the condition of rule 8 was not evaluated.
Δ_{i+1}	=	output of action of rule 5
$\vdots$	$\vdots$	$\vdots$
Δ_n		Suppose rules 3,5,8 are not triggered between steps i and n

At this point, then, we have

MR[3]	=	i
MR[5]	=	$i+1$
MR[8]	=	value of MR[8] at step i
PREV[3]	=	$\Delta_0 ! \Delta_1 ! \cdots ! \Delta_i$
PREV[5]	=	$\Delta_0 ! \Delta_1 ! \cdots ! \Delta_i ! \Delta_{i+1}$
PREV[8]	=	value of PREV[8] at step i
CHANGE[3]	=	actual(PREV[3], $\Delta_{i+1} ! \Delta_{i+2} ! \cdots ! \Delta_n$)
CHANGE[5]	=	actual(PREV[5], $\Delta_{i+2} ! \Delta_{i+3} ! \cdots ! \Delta_n$)
CHANGE[8]	=	actual(PREV[8], $\Delta_{\mathtt{MR[8]}+1} ! \cdots ! \Delta_n$)
curr	=	actual(empty_delta, $\Delta_0 ! \Delta_1 ! \cdots ! \Delta_n$)

Figure 1: Illustration of sequence of deltas used for Starburst semantics

we specify this in Heraclitus by constructing a sequence of deltas (see Figure 1). In our specification, we use the variable curr to hold the cumulative effect of all of the deltas created so far. In addition to computing the current cumulative effect of the rule-created deltas, the Starburst semantics keeps track of the most recent time which a rule is known to have been "satisfied". In a series of comments, we maintain an array MR[1..r] of integers which essentially indicate, for each i, the most recent step after which rule i was known to be satisfied.

A rule i is triggered on the basis of the net (requested) change of the database which occurred since the most recent time when it was satisfied, i.e., from the MR[i]-th step to the current step. In our specification, we maintain these net changes in the array CHANGE[1..r] of deltas. For computational purposes, we also maintain an array PREV[1..r] of deltas, which hold for each i the delta corresponding to the "old" state against which rule i will be considered.

To describe the semantics more precisely, we need some technical terminology. A rule is *triggered* at step k if trig@i(CHANGE[i]) is true at that time. A rule is *evaluated* at step k if cond@i is evaluated (i.e., tested) during this step. (This can occur only if this rule was triggered here.)

During step k, the set of all triggered rules is computed, and then a loop is executed in which the triggered rules are evaluated in sequence until one of them yields cond@i(CHANGE[k],curr) = true. If rule i is triggered at step k and its condition is evaluated with value false, then MR[i] is set to k. If rule i is triggered and is evaluated with value true, then its effect is added to curr to form the $(k+1)$-st delta of the sequence, and MR[i] is set to $k+1$.

Suppose now that DB is the initial database, that we have performed n steps of the computation, and that MR[i] $= j$. Then PREV[i] will hold $\Delta_0!\Delta_1!\ldots!\Delta_j$, and rule i is going to act as if the underlying database is $DB' = apply($PREV[i]$, DB)$, and that the requested update is $\Delta' = \Delta_{j+1}! \cdots !\Delta_n$. It is assumed that Δ' is in *reduced* form relative to DB'. Intuitively, this means that Δ' is replaced by the minimal set Δ'' of atomic ground deltas such that $apply(\Delta'', DB') = apply(\Delta', DB')$. More formally, for database DB and deltas Δ_1, Δ_2 we define actual(Δ_1, Δ_2) to be the merge over all relations R in DB of merge{t | not R(t) when Δ_1 and R(t) when $\Delta_1!\Delta_2$ | <+R(t)>} merged with merge{t | R(t) when Δ_1 and not R(t) when $\Delta_1!\Delta_2$ | <-R(t)>}. We assume that actual is defined as a function.

In order to compute the values of CHANGE[i] incrementally, we also maintain an array PREV[1..r] of deltas, where PREV[i] holds actual(empty_delta,$\Delta_0! \cdots !\Delta_{\texttt{MR[i]}}$). We now have the Starburst semantics:

```
procedure starburst(prop:delta):delta
dec position:int,
    flag:boolean,
    total,increment:delta,
    constant r: int,
    PREV,CHANGE:array([1..r]) of delta,
    function actual(prev,curr:delta):delta  % ...
    function select_next(rel(int)):int  % ...
    in
 curr := actual(empty_delta,prop);
     % curr always holds the full effect of all
     % deltas computed so far
 for i := 1 to r do    %initialization
  % MR[i] := 1;
  PREV[i] := empty_delta
  CHANGE[i] := curr
 endfor;
 position := 1;
     % position holds the number of current step
 repeat  % begin main loop
  TRIG := empty_reln;
  for i := 1 to r do if trig@i(CHANGE[i])
                  then TRIG := TRIG + <i> endif;
     % TRIG now holds indices of triggered rules
  flag := false;
  while TRIG != empty_reln and not flag do
     % we assume select_next() is a procedure
     % for element selection; Starburst
     % semantics suggests rule priorities
   i := select_next(TRIG);
   TRIG := TRIG - <i>;
     % rule i is evaluated in next step
   if cond@i(CHANGE[i],curr) != empty_reln
    then flag := true
   else
      % we now have bookkeeping because
      % cond@i has been evaluated
    % MR[i] := position;
    PREV[i] := curr;
    CHANGE[i] := empty_delta;
   endif
  endwhile;
 if flag then
  increment := action@i(CHANGE[i],curr);
  if increment = rollback
   then return empty_delta endif;
     % exit from procedure if the called-for
     % update is ''rollback'' otherwise, add
     % the delta to the sequence
  position := position + 1;
  curr := actual(empty_delta,curr!increment);
        % now do bookkeeping for all rules
  % MR[i] := position;
  PREV[i] := curr;
  CHANGE[i] := empty_delta;
  for j := 1 to r do
   if j != i then CHANGE[j] :=
      actual(PREV[j],CHANGE[j]!increment) endif;
   endfor
  endif % of test on flag
 until TRIG = empty_reln
 endrepeat;  % end main loop
     % if one of the rules was applied, then
     % TRIG did not become empty, and so the
     % loop will be repeated.  If TRIG did
     % become empty, then the condition of
     % each triggered rule failed, i.e., no
     % rule can be applied
 return curr
enddec
```

In the above procedure, two arrays of deltas are maintained. An alternative procedure can be specified in Heraclitus in which only one array (of size r) of deltas is maintained, but with considerable computational overhead, thus providing a kind of space-time trade-off.

6 "Hypothesizing" vs. "Peeking"

Some active databases support hypothesizing (i.e., using when) in formulas as the means for accessing deltas, while others support peeking (i.e., using in). In this section we use Heraclitus to explore the ability of each of these to simulate the other. In general the simulations go in both directions, but we exhibit a family of restrictions under which peeking cannot be simulated by hypothesizing. Due to space limitations, we omit many of the formal arguments, and provide only a sketch of the main result. Also, we focus primarily on the use of in and when in calculus formulas.

Simulation of when by in in formulas can be accomplished by a transformation of the formulas. Suppose that φ is a formula involving when but not in. In the first step, φ is placed into prenex conjunctive normal form (in particular, so that each negation symbol immediately precedes an atom) and the when's are "moved inwards" so that they range exclusively over atoms and negated atoms (or atoms which are already qualified by when's). Nested when's are resolved by replacing (φ when δ_1) when δ_2 with φ when (δ_2 ! (δ_1 when δ_2)). It can be shown that these transformations preserve equivalence. Now perform the following replacements: Replace R(t) when δ by

<+R(t)> in δ or (R(t) and not <-R(t)> in δ)

Replace not R(t) when δ by

<-R(t)> in δ or (not R(t) and not <+R(t)> in δ)

Proposition 6.1: The transformation from a formula with hypothesizing to peeking described above yields an equivalent formula.

The transformation may extend the length of the formula as much as exponentially, because of the transformation to conjunctive normal form. The problem of finding a less expensive transformation from hypothesizing to peeking remains open at this time, as does an analysis of the expressive complexity [Var82] (intuitively, the succintness) of programs using hypothesizing vs. peeking.

The simulation of peeking by hypothesizing cannot be accomplished using a transformation on formulas analogous to the one just given for the opposite direction, as shown in Theorem 6.2 below. After presenting this result, we give a less direct simulation of peeking which uses neither peeking nor hypothesizing.

Theorem 6.2 focuses on two classes $\mathcal{F}_{in}$ and $\mathcal{F}_{when}$ of Heraclitus functions. It should be noted that the the computation of deltas for the semantics of the AP5 consistency rules can be formulated as an instance of $\mathcal{F}_{when}$, and the related semantics of rule application studied in [ZH90] can be formulated as an instance of $\mathcal{F}_{in}$. In particular, then, this result shows that the template of [ZH90] is strictly stronger than the consistency rule portion of AP5 considered in isolation. (This statement must of course be taken with a grain of salt, because the theorem restricts attention to the case where additional relational variables cannot be used, but it is easy in AP5 to create additional relations in the database, which can serve as relational variables.)

Theorem 6.2: Consider the class of Heraclitus functions $\mathcal{F}$ with the following properties:

(a) the input variable for the program is curr;

(b) the only variable used with the return command is curr;

(c) no relational variables are introduced;

(d) only curr is qualified by when or in;

(e) the only boolean tests on delta variables are = and != (in particular, there are no tests for = fail);

(f) for each assigment statement of the form curr := expr, expr has the form curr & ..., (i.e., curr is modified only through augmentation); and

(g) there are no function or procedure calls, and in particular no use of arithmetic or string manipulation functions.

Furthermore, let

(i) $\mathcal{F}_{in}$ denote elements of $\mathcal{F}$ which do not use when; and

(ii) $\mathcal{F}_{when}$ denote elements of $\mathcal{F}$ which do not use in.

there is a function peek_can_do in $\mathcal{F}_{in}$ which is not equivalent to any function in $\mathcal{F}_{when}$.

The intuition of the proof of this theorem stems from the fact that a delta variable Delta might hold an element <-R(t)> where R(t) is false in the database. If peeking and relational variables are not used, it turns out that it is impossible to detect the presence of such elements of Delta. (Note that in the Starburst semantics, the deltas which are used in peeking are reduced by the actual function, and so these no-op tuples cannot play a role there.)

Sketch of proof of Theorem 6.2: We begin by describing the function peek_can_do in $\mathcal{F}_{in}$. It uses an underlying database with three binary relations R, S and T, all ranging over strings. peek_can_do will have the property that on an input instance $[I, J, \emptyset]$ (where I is the relation assigned to R, J to S and $\emptyset$ to T) and input empty_delta we will have[5] *apply*(peek_can_do(empty_delta), $[I, J, \emptyset]$) = $[I, J,$ *trans_closure*$(I) \cap J]$. The function is given by:[6]

```
function peek_can_do(curr:delta):delta
dec prev:delta in
 repeat
  (prev,curr) := (curr,curr &
      merge{ x,z | exists y.(R(x,y) and
                   (R(y,z) or <-R(y,z)> in curr)
                   and not R(x,z))
                 | <-R(x,z)> }
  until curr = prev endwhile;
 return merge{ x,y | S(x,y) and
                     (R(x,y) or <-R(x,y)> in curr)
                   | <+T(x,y)> }
enddec
```

Intuitively, after the repeat-loop has executed, we have *trans_closure*(R) = union{ x,y | R(x,y) or <-R(x,y)> in curr }.

This is used in the return statement of peek_can_do, which uses curr to identify the tuples that should be inserted into T.

The proof that this cannot be simulated by an element of $\mathcal{P}_{when}$ relies on (a generalization of) the fact that the relational calculus cannot compute transitive closure [AU79].

[5] *trans_closure*(K) denotes the transitive closure of a binary relation K.

[6] This program can be expressed using the language of [ZH90], with rules that satisfy the conditions stated there for order-independent rule application.

In particular, suppose that the function `will_not_work` in $\mathcal{P}_{when}$ does simulate `peek_can_do`. Let I_1 consist of two long non-intersecting "chains", one from \$1 to \$2 and the other from %1 to %2. Let I_2 be similar, but with chains from \$1 to %2 and from \$1 to %2. In particular, the chains should be chosen to be so long relative to `will_not_work` that no formula occurring in `will_not_work` is able to distinquish between I_1 and I_2. Also, let J = <\$1,\$2>. Note that

$$\begin{array}{lcl} \texttt{peek\_can\_do}[I_1, J, \emptyset] & = & [I_1, J, J] \\ \texttt{peek\_can\_do}[I_2, J, \emptyset] & = & [I_1, J, \emptyset] \end{array}$$

We now induct on the execution of `will_not_work` on inputs $\mathtt{I}_1 = [I_1, J, \emptyset]$ and $\mathtt{I}_2 = [I_2, J, \emptyset]$, showing that on input $\mathtt{I}_k$, at each step of the execution,

$$\begin{array}{l} \texttt{union\{ x,y | R(x,y) when curr \}} = I_k \\ \texttt{union\{ x,y | S(x,y) when curr \}} = J \end{array}$$

and that at each step, `union{ x,y | T(x,y) when curr }` is $\emptyset$ or <\$1,\$2 >. (This last observation follows in part because the final output of `will_not_work` must be computed by augmenting the delta value held in curr; at most the element <\$1,\$2> can be added to T.)

In the proof we also handle the case of computations occurring with delta variables other than curr. □

Finally, we state

Proposition 6.3: Heraclitus programs using in can be simulated by Heraclitus programs using neither in nor when.

One way to achieve this simulation is to maintain "new" relation variables $\mathtt{R}_{\mathtt{U},add}$ and $\mathtt{R}_{\mathtt{U},sub}$ for each relation variable R and each delta variable D occurring in the program. Commands are added to the initial program so that at each point of the computation these relation variables hold, respectively, `union{ t | <+R(t)> in D }` and `union{ t | <-R(t)> in D }`. It is now trivial to replace all occurrences of in by tests to these relational variables.

7 Conclusions

In this paper we have shown how the constructs of the Heraclitus language can be used to specify the rule application semantics of prominent active database systems found in the literature, and also used it to study a particular technical issue concerning hypothesizing vs. peeking. In addition to the specific contributions of providing the first specifications for two active database systems in a formal langauge (as opposed to English), this paper demonstrates that Heraclitus can be used as a common language for specifying a wide variety of alternative semantics for active databases.

Heraclitus can provide part of the foundation for the study of a wide range of topics. We are currently in the initial phases of implementing Heraclitus, in order to provide a test bed for experimentation with active database semantics, and also to understand implementation issues in the context of active databases and more generally, delayed updates. Other directions to be pursued include: the development of compile-time tools for certifying that rule bases and rule application templates will enforce various integrity constraints; extending the Heraclitus constructs to incorporate features from semantic and object-oriented databases; studying the impact of rules in the context of heterogeneous databases; and to better understand the interplay of concurrent database usage and rule application.

Acknowledgements

The authors are grateful to members of the Software Systems group at USC/Information Sciences Institute, including in particular Dennis Allard, Don Cohen, Neil Goldman, and Dave Wile, for numerous informative discussions concerning active databases and AP5; and also Junhui Luo, for clarifying the subtleties of nested when's.

References

[AS90] S. Abiteboul and E. Simon. Fundamental properties of deterministic and nondeterministic extensions of Datalog. Technical report, INRIA, July 1990. to appear in *Theoretical Computer Science.*

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 110–120, 1979.

[AV88] Serge Abiteboul and Victor Vianu. Datalog extensions for database queries and updates. Technical Report 900, INRIA, September 1988. to appear in *Journal of Computer and System Sciences*; extended abstract appears in *Proc. ACM Symp. on Principles of Database Systems, 1988.*

[CC+90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 225–236, 1990.

[Coh86] Don Cohen. Programming by specification and annotation. In *Proc. of AAAI*, 1986.

[Coh87] Don Cohen. AP5 reference manual. Technical report, USC/Information Sciences Institute, 1987.

[Coh89] Don Cohen. Compiling complex database transition triggers. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 225–234, 1989.

[CW90] Stefano Ceri and Jennifer Widom. Deriving production rules for constraint maintenance. Technical Report RJ 7348 (68829), IBM Alemaden Research Center, March 1990. Abstract appears in *Proc. of Intl. Conf. on Very Large Data Bases*, 1990.

[GT89] A. van Gelder and R.W. Topor. Safety and translation of relational calculus queries. Technical Report UCSC-CRL-89-40, Baskin Center for Computer Engineering and Information Sciences, University of California, Santa Cruz, December 1989. to appear, *ACM Transactions on Database Systems.*

[H89] E.N. Hanson. An initial report on the design of Ariel: A DBMS with an integrated production rule system. *SIGMOD Record* 18(3), September 1989, 12-19

[HJ90] R. Hull and D. Jacobs. On the semantics of rules in database programming languages. In *Next Generation Information System Technology: Proc. of the First International East/West Workshop, Kiev, USSR, October 1990*, ed. by J. Schmidt and A. Stogny, Springer-Verlag LNCS, volume 504, 1991, to appear.

[JH91] D. Jacobs and R. Hull. Database programming with delayed updates. Technical report, Computer Science Department, University of Southern California, 1991. In preparation, to be submitted to DBPL 91.

[MD89] Dennis R. McCarthy and Umeshwar Dayal. The architecture of an active data base management system. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 215–224, 1989.

[Min88] Jack Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan-Kaufmann, Inc., San Mateo, CA, 1988.

[MP90] C.B. Medeiros and P. Pfeffer. A mechanism for managing rules in an object-oriented database. Technical report, Altair, 1990.

[M83] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. *Proc. of Intl. Conf. on Very Large Data Bases*, pages 34–42, 1983.

[Sch77] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Trans. on Database Systems*, 2(3):247–261, September 1977.

[SdM88] E. Simon and C. de Maindreville. Deciding whether a production rule is relational computable. In *Proc. of Intl. Conf. on Database Theory*, pages 205–222, 1988.

[SIG89] SIGMOD Record 18:3, "Special Issue on Rule Management and Processing in Expert Database Systems", September 1989.

[SJ+90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 281–290, 1990.

[Var82] Moshe Y. Vardi. The complexity of relational query languages. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 137–146, 1982.

[WF90] Jennifer Widom and Sheldon J. Finkelstein. Set-oriented production rules in relational database systems. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 259–264, 1990.

[ZH90] Y. Zhou and M. Hsu. A theory for rule triggering systems. In *Intl. Conf. on Extending Data Base Technology*, pages 407–421, 1990.