Heraclitus: Elevating Deltas to be First-Class Citizens in a Database Programming Language

SHAHRAM GHANDEHARIZADEH, RICHARD HULL, and DEAN JACOBS
University of Southern California

Traditional database systems provide a user with the ability to query and manipulate one database state, namely the current database state. However, in several emerging applications, the ability to analyze "what-if" scenarios in order to reason about the impact of an update (before committing that update) is of paramount importance. Example applications include hypothetical database access, active database management systems, and version management, to name a few. The central thesis of the Heraclitus paradigm is to provide flexible support for applications such as these by elevating *deltas*, which represent updates proposed against the current database state, to be first-class citizens. Heraclitus[Alg, C] is a database programming language that extends C to incorporate the relational algebra and deltas. Operators are provided that enable the programmer to explicitly construct, combine, and access deltas. Most interesting is the when operator, that supports hypothetical access to a delta: the expression E when δ yields the value that side effect free expression E would have *if* the value of delta expression δ were applied to the current database state. This article presents a broad overview of the philosophy underlying the Heraclitus paradigm, and describes the design and prototype implementation of Heraclitus[Alg, C]. A model-independent formalism for the Heraclitus paradigm is also presented. To illustrate the utility of Heraclitus, the article presents an in-depth discussion of how Heraclitus[Alg, C] can be used to specify, and thereby implement, a wide range of execution models for rule application in active databases; this includes both prominent execution models presented in the literature, and more recent "customized" execution models with novel features.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Feature; H.2.3 [**Database Management**]: Languages—*database (persistent) programming languages*; H.2.4 [**Database Management**]: Systems—*query processing*; H.2.M [**Database Management**]: Miscellaneous—*active databases*

The research was supported in part by the NSF under grants IRI-9318326, IRI-9222926, and CDA-9216321. This is Technical Report USC-CS-94-581 of the Computer Science Department at USC, and is available by anonymous ftp on perspolis.usc.edu in the file HERACLITUS/TR-581. Research by S. Ghandeharizadeh was also supported in part by NSF grant IRI-9258362. Research by R. Hull was also supported in part by NSF grant IRI-9107055 and ARPA grant NAG2-862, administered through NASA-AMES.
Authors' addresses: S. Ghandeharizadeh, Department of Computer Science, University of Southern California, Los Angeles, CA 90089; R. Hull, Computer Science Department, University of Colorado, Boulder; D. Jacobs, Sonic Solutions, 1891 East Francisco Boulevard, San Rafael, CA 94901.

General Terms: Design, Languages

Additional Key Words and Phrases: Active databases, deltas, execution model for rule application, hypothetical access, hypothetical database state

1. INTRODUCTION

The primary focus of a database management system is to maintain and provide access to the current *database state*, which holds a representation of some portion of reality. In many applications it is also common to maintain more than one version of the state or parts of it, either at a semantic or a physical level. Speaking at a semantic level, the most prominent examples are versioning systems such as those that arise in computer-aided engineering, temporal databases, and systems for supporting hypothetical data analysis. The emerging area of active databases also makes use of multiple versions of the database state, and "deltas" between them. At a more physical level, multiple states and the deltas between them arise in support of concurrency control, recovery, and archiving. In most cases, the semantics and implementation of multiple states and deltas have been developed on an ad hoc basis for each of the different application areas.

The goal of the Heraclitus project is to develop language constructs and implementation techniques for representing multiple states, using the approach of making deltas "first-class citizens" in database programming languages (DBPLs). These language constructs can be used directly in support of "what-if" scenario analysis. More generally, by supporting deltas as first-class citizens in a DBPL, Heraclitus provides a testbed both for experimenting with a wide variety of approaches to existing applications such as version control and active databases, and also opens the possibility of developing new applications, such as systems for detecting and resolving conflicts between proposed updates. This article gives a comprehensive description of the first phase of the Heraclitus project, which focuses on the design, implementation, and application of Heraclitus[Alg, C], a relational DBPL that extends C with relations and constructs for creating and accessing deltas, and hence, multiple virtual database states. It also provides an abstract and model-independent perspective on the Heraclitus paradigm, and illustrates how the Heraclitus paradigm can be used to unify and generalize current research on active database systems.

Speaking intuitively, in the Heraclitus paradigm a *delta* corresponds to a proposed update to the database that is not necessarily applied. Perhaps the most interesting operator for accessing deltas is when. This is a binary operator, written infix. If E is a side effect free expression (more precisely, an expression with no side effects on the database state) and δ is an expression yielding a delta, then "E when δ" evaluates to the value that E would have, *if* the value of δ were applied to the database. (Here E might be, e.g., a query, an expression evaluating to a delta, or a function call that returns an integer based on a complex analysis of the database state.) This is called *hypothetical access*, because it views the database state under the hypothesis that the delta had been applied. To illustrate, one can express "compute total payroll" when "give everybody in the sales department a 10% raise." Different deltas can be used to represent different hypothetical states or versions of the world. Operators are also provided to combine deltas; for example, the composition operator applied to δ_1 and δ_2 forms the delta that corresponds to applying δ_1 followed by applying δ_2.

As a richer illustration of using deltas and the when operator in support of hypothetical database access, consider a stock market database and a financial analyst managing several portfolios. The analyst is allowed to change each portfolio's investment by either selling or buying shares. However, most of the time the analyst speculates on the trends in the market, the change in price of different shares, and his or her possible response to these changes. Using the Heraclitus paradigm, the analyst may represent each speculation as a delta. Using the delta operators, the analyst may query and reason about the impact of one or more deltas on the final value of a portfolio, combine several deltas into one, and detect and resolve conflicts among deltas. If a decision is made to use a delta reflecting a given management strategy, then the delta can be *applied* to the database to achieve the result. An unusual feature of the Heraclitus paradigm is that several deltas corresponding to several different market trends and management strategies can be maintained simultaneously, and the hypothetical states that they represent can be compared against each other. Unlike most version management systems, Heraclitus provides support for rapidly switching contexts between hypothetical states, and for comparing them.

Heraclitus[Alg, C] is a full-fledged DBPL; that is, it blends the full power of a programming language (C) with concurrent access to persistent bulk data (relations and deltas). The primary challenge of Heraclitus is the identification of appropriate general-purpose language constructs for creating, accessing, and combining deltas. The main contribution of the language Heraclitus[Alg, C] is the design and implementation of such constructs for the pure relational model (i.e., no duplicate tuples and no tuple identifiers). In this context, a delta is viewed as a set of tuples to be inserted (possibly into more than one relation) and a set of tuples to be deleted (possibly involving more than one relation). We feel that the language constructs chosen for Heraclitus[Alg, C] are sufficiently rich to support in a flexible manner a variety of applications. Heraclitus[Alg, C] has already been used for investigations concerning active databases and their applications, including the development of new execution models for active databases [Chen et al. 1994; Dalrymple 1995], and the use of active technology to support software and database interoperation [Dalrymple 1995; Zhou et al. 1995].

The design and implementation of Heraclitus[Alg, C] provided us with significant experience and insight into the notion of deltas and how to support them in terms of both language design (Section 3) and implementation (Section 4). In connection with the language design, two issues are notable. First, interesting algebraic equivalences are developed for the when operator. Subtleties arise, because nested function calls involving whens preclude the possibility of compile-time optimization. The second issue involves the "intended meaning" of deltas. As detailed in Section 3.3, the intended meaning of an update such as "give everyone in the sales department a 10% raise" is not captured directly by a delta, but rather by a function that returns possibly different deltas for different database states.

A primary focus of Heraclitus[Alg, C] has been on the context of using deltas within a single transaction (as arises in many execution models for active databases), and/or the use of deltas in a single-user environment (as found in many personal-computer database systems). For completeness, Heraclitus[Alg, C] does incorporate some basic operators for concurrency control. It is possible in Heraclitus[Alg, C] to construct and use deltas that persist across transaction boundaries in a multi-user environment, but little support is currently provided to ensure that the intended meaning of deltas is preserved as the underlying database state evolves. Providing such support is an open research problem currently under investigation [Doherty and Hull 1995; Doherty et al. 1995a].

At the physical level, the when operator is implemented in combination with algebraic operators. For example, when evaluating the expression (R join S) when δ, the system will simultaneously (a) form the join of R and S and (b) "filter" the values of R and S through δ. This approach avoids the computation of the full values of R when δ and S when δ. We have experimented with two approaches for accessing deltas, one based on hashing and the other using sort-merge techniques. Benchmarking experiments verified (Section 4.3) that the cost of hypothetical access with a relational operator is a linear function of the cost to execute the operator and the size of the delta.

The Heraclitus project was initiated in 1991, with the development of an abstract Heraclitus language based on the relational calculus [Jacobs and Hull 1991] and a study of its utility in connection with specifying execution models for active databases [Hull and Jacobs 1991]. A prototype implementation of the algebraic operators for deltas in the relational model was developed in Ghandeharizadeh et al. [1992], and subsequently expanded into Heraclitus[Alg, C] [Ghandeharizadeh et al. 1993]. The present article provides a complete, unified, and refined description of this work, and includes many details that were omitted from the conference publications. It also introduces an abstract model of deltas and their operators (Section 5) that is independent of the relational model, thus providing a starting point for extending the Heraclitus paradigm to other data models, for example, functional data model, object-oriented data model, relations with duplicates, and the like.

The rest of this article is organized as follows. Section 2 surveys related research. Section 3 describes the language level of Heraclitus[Alg, C] in considerable detail in a tutorial form. Section 4 describes the implementation of Heraclitus[Alg, C], and briefly outlines the results of initial benchmarking experiments with the prototype. Section 5 presents the model-independent perspective of deltas and their operators. Section 6 considers how the Heraclitus paradigm can be used to specify, and thereby implement, a wide variety of execution models for active databases. Finally, Section 7 offers brief conclusions and describes some current and future research efforts in connection with the Heraclitus paradigm.

2. RELATED WORK

The work on Heraclitus was inspired and influenced by many related research efforts; in the following we list some of the most important ones.

One of the prime motivations for developing the Heraclitus paradigm is the importance of deltas in connection with the *execution model*, or semantics of rule application, in active database systems [Widom and Ceri 1995; Hanson and Widom 1992; Stonebraker 1992]. Indeed, as first indicated in Hull and Jacobs [1991] and detailed in Section 6, conceptually viewing deltas as first-class citizens provides a useful basis for specifying and clarifying a variety of different execution models for active databases including, for example, the Starburst Active Database Rule System [Widom and Finkelstein 1990; Widom 1993; Widom and Ceri 1995], AP5 [Cohen 1986, 1989; Zhou and Hsu 1990], POSTGRES [Stonebraker et al. 1990; Widom and Ceri 1995], ARIEL [Hanson 1989; Widom and Ceri 1995], and LOGRES [Cacace et al. 1990]. Several recent articles on active database systems, including [Widom 1993; Simon and Kiernan 1995; Collet et al. 1994], describe the semantics of their systems in terms of deltas. These systems construct and manipulate deltas solely in the context of applying rules within a single transaction. Unlike Heraclitus, these systems do not support persistent deltas, or constructs to create, manipulate, and compare completely unrelated deltas. Furthermore, as detailed in Section 6.4, Heraclitus can be used directly to specify, and thereby implement, novel execution models for specialized purposes.

Differential files [Severance and Lohman 1976] are a low-level implementation technique that supports efficient representation of multiple versions of a database. Unlike differential files, deltas in the Heraclitus framework are manipulated directly by constructs supported by the user-level programming language. Furthermore, we support a family of operators for explicitly constructing and combining deltas in addition to those for explicitly and hypothetically accessing them.

A version of hypothetical relations is introduced in Woodfill and Stonebraker [1983]. Although the work there describes carefully crafted implementation strategies for such relations, it cannot be extended easily to provide the full generality of delta usage supported in the Heraclitus framework.

Gabbay [1985] and Bonner [1990, 1995] consider extensions of Datalog that incorporate hypothetical reasoning. That work is focused primarily on the development of a proof theory and model theory that includes hypothetical reasoning. Similar to Heraclitus[Alg, C], hypothetical changes to a database state are represented as sets of atoms to be added to or deleted from the state.

It has been suggested that a reasonable approach to support the basic functionality of the when operator would be to augment existing concurrency control mechanisms, using the following steps: (a) evaluate E when δ by applying δ to the database (but do not commit), (b) evaluate E in the context of the new database, and (c) roll back the transaction in order to undo δ. Although this rollback technique will be useful in some contexts, it is just one of several feasible implementation strategies that warrant investigation. In the case of complex algebraic expressions involving several not necessarily nested deltas, it may be more efficient to incorporate optimization of when into the conventional optimization of the other algebraic operators, rather than relegating it to the orthogonal rollback mechanism. Also, the use of rollbacks to support hypothetical database access may cause unacceptable delays in the concurrency system, complicate the transaction protocols, and degrade the performance of the system.

The development of Heraclitus[Alg, C] was greatly influenced by other research on DBPLs [Atkinson and Buneman 1987], and in particular by DBPLs that combine the relational model with an imperative programming language, such as PASCAL/R [Schmidt 1977].

3. THE LANGUAGE OF HERACLITUS[ALG, C]

The foundation of the Heraclitus paradigm is the notion of *delta values*, often simply termed *deltas*; each of these is a function that maps the family of database states to the family of database states. Intuitively, a delta can be thought of as a "delayed update," that is, a command that can be used to update an arbitrary database state, but is not necessarily applied. Three operations are fundamental to deltas: applying them to a given database state to obtain a new one, function composition, and hypothetical access using the when operator.

This section presents a tutorial introduction to Heraclitus[Alg, C], a particular realization of the Heraclitus paradigm for the pure relational model (i.e., no duplicates and no tuple-ids). An algebraic perspective on how deltas are supported for the relational model is presented in Section 3.1. The embedding of deltas and their operators into C is presented in Section 3.2. Figure 3 of this section summarizes the algebraic and language-level operators of Heraclitus[Alg, C]. Some unexpected subtleties arise; these are discussed in Section 3.3. We conclude in Section 3.4 with a brief note on support for concurrency in Heraclitus[Alg, C]. A running example is used in this section both to illustrate the basic components of the language and also to illustrate the use of the Heraclitus paradigm in connection with hypothetical database access.

Because Heraclitus combines several familiar concepts in a novel manner, it may be difficult to understand some of the subtleties of Heraclitus[Alg, C] during a first reading of this section. The following three sections of the article provide different perspectives on the Heraclitus paradigm and Heraclitus[Alg, C]. Specifically, Section 4 describes the implementation of Heraclitus[Alg, C], with special emphasis on implementation of the various language-level operators. Section 5 gives a more abstract perspective on the

Supplier	Part
Trek	frame
Campy	brakes
Campy	pedals

Suppliers

Part	Quantity	Supplier	Expected
frame	400	Trek	8/31/93
brakes	150	Campy	9/1/93

Orders

Fig. 1. Relations for inventory control example: database state S_a.

Heraclitus paradigm that is essentially model independent. And Section 6 shows how Heraclitus[Alg, C] can be applied to specify a variety of execution models for active databases. That section provides rich examples of pseudocode written in Heraclitus[Alg, C].

3.1 The Algebraic Perspective

This section describes the *algebraic* level of the Heraclitus paradigm for the pure relational model. The focus is on delta *values* and the algebraic operators that manipulate them. The following sections describe how this algebra is embedded into the imperative programming language C; the focus there is on delta *expressions*.

We begin with a few philosophical remarks. Several factors affect the design of a specific realization of the Heraclitus paradigm. Obviously we expect that all deltas considered are computable. Furthermore, the family of deltas should be closed under composition. Even in this case, there is a tradeoff between the expressive power of the family of deltas incorporated, and the efficiency with which they can be manipulated and accessed. In Heraclitus[Alg, C] we use a restricted family of deltas that permits efficient access and manipulation while retaining enough expressiveness to support a variety of applications. (The tradeoff between different representations of delta values is considered in Section 5, in connection with Requirement 3 there.)

We now describe the representation for deltas used in Heraclitus[Alg, C], and the function on database states that each represents. We sometimes refer to the representation of deltas as "tabular." This is because each delta is represented using what amounts to a table that lists the tuples to be inserted or deleted. (Other representations might be based on, e.g., a functional notation.) We also describe the binary algebraic operators *smash* and *merge* on deltas.

Delta values are defined in the context of a fixed relational database schema. As a running example, we use a simple database concerning inventory control in manufacturing. Figure 1 shows part of state S_a for a database used by a hypothetical bicycle manufacturer. The `Suppliers` relation holds suppliers and the parts they supply, and the `Orders` relation shows currently unfilled orders for parts. Other relations, not shown in Figure 1, might hold information about the parts usage of different bicycle models, and the expected demand for these parts based on the production schedule of the company.

A *signed atom* is an expression of the form + ⟨*reln-name*⟩ ⟨*tuple*⟩ or − ⟨*reln-name*⟩ ⟨*tuple*⟩; intuitively these correspond to "insertions" and "deletions," respectively. In the context of Heraclitus[Alg, C], a *delta (value)* is represented as a finite set of signed atoms (referring to relations in the current database schema) that is *consistent*; it does not include both positive and negative versions of the same atom. An example is:

$$\Delta_1 = \left\{ \begin{array}{l} +\,Suppliers(Shimano, brakes), \\ +\,Suppliers(Trek, frame), \\ -\,Orders(brakes, 150, Campy, 9/1/93), \\ +\,Orders(brakes, 300, Shimano, 9/6/93) \end{array} \right\}.$$

Note that signed atoms involving more than one relation can be elements of a single delta. We also include a special delta value *fail*, that corresponds to inconsistency.

For non-*fail* delta Δ, we set

$$\Delta^+ = \{A \mid +A \in \Delta\}$$
$$\Delta^- = \{A \mid -A \in \Delta\}.$$

The consistency requirement on deltas is that $\Delta^+ \cap \Delta^- = \emptyset$. Also, we say that a pair of signed atoms *conflict* if they violate the consistency requirement, that is, if they have the form $+t$ and $-t$. (In general, the notion of delta value used may depend on the underlying data model and the underlying application. For example, if key dependencies were considered to be part of the data model, and if relation $R[A, B]$ has key A, the consistency requirement might be expanded to say that there cannot be two signed atoms of the form $+R(a, b)$ and $+R(a, c)$, where $b \neq c$.)

The delta value Δ represents the function that maps a database state[1] S to $(S \cup \Delta^+) - \Delta^-$, which, due to the consistency requirement, is equal to $(S - \Delta^-) \cup \Delta^+$. The result of applying a delta Δ to state S is denoted *apply*(S, Δ). Speaking informally, applying Δ has the effect of adding tuples of Δ preceded by a "+", and deleting tuples preceded by a "−". The insertion and deletion of these tuples can be viewed as occurring simultaneously, or in an arbitrary sequence; the consistency requirement ensures that any of these will yield the same net result.

The result of applying Δ_1 to state S_a of Figure 1 is shown as state S_b in Figure 2. Because we are working with the pure relational model, the signed tuple +*Suppliers*(*Trek, frame*) can be viewed as a "no-op" in this context; it has no impact when apply is used on state S_a. Deletes are "no-ops" if the associated tuple is not present in the underlying instance. A mechanism to express "modifies" is also incorporated; see Section 3.2.

At the algebraic level, the composition operator for deltas is called *smash*, denoted "!". The smash of two delta values is basically their union, with

[1]In this context, we view the database state to be a set of atoms, for example, {*Suppliers*(*Trek, frame*), *Suppliers*(*Campy, brakes*), . . . , *Orders*(*frame*, 400, *Trek*, 8/31/93), . . .}.

Supplier	Part
Trek	frame
Campy	brakes
Campy	pedals
Shimano	brakes

Suppliers

Part	Quantity	Supplier	Expected
frame	400	Trek	8/31/93
brakes	300	Shimano	9/6/93

Orders

Fig. 2. Result of applying Δ_1: database state S_b.

conflicts resolved in favor of the second argument. For example, given

$$\Delta_2 = \left\{ \begin{array}{l} +\,Suppliers(Cat\ Paw,\ light), \\ -\,Suppliers(Campy, pedals), \\ -\,Orders(brakes,\ 300,\ Shimano,\ 9/6/93), \\ +\,Orders(brakes,\ 500,\ Shimano,\ 9/20/93) \end{array} \right\},$$

then $\Delta_1 ! \Delta_2$ equals

$$\left\{ \begin{array}{l} +\,Suppliers(Shimano,\ brakes), \\ +\,Suppliers(Trek, frame), \\ +\,Suppliers(Cat\ Paw,\ light), \\ -\,Suppliers(Campy, pedals), \\ -\,Orders(brakes,\ 150,\ Campy,\ 9/1/93), \\ -\,Orders(brakes,\ 300,\ Shimano,\ 9/6/93), \\ +\,Orders(brakes,\ 500,\ Shimano,\ 9/20/93) \end{array} \right\}.$$

Formally, for arbitrary non-*fail* delta values Δ_1 and Δ_2, their smash is defined by

$$(\Delta_1 \ ! \ \Delta_2)^+ = \Delta_2^+ \cup (\Delta_1^+ - \Delta_2^-)$$
$$(\Delta_1 \ ! \ \Delta_2)^- = \Delta_2^- \cup (\Delta_1^- - \Delta_2^+)^{\cdot}$$

It is easily verified that smash realizes function composition for the family of deltas, that is, for state S and deltas Δ_1, Δ_2,

$$apply(S, \Delta_1 \ ! \ \Delta_2) = apply(apply(S, \Delta_1), \Delta_2).$$

Most active database systems use smash when combining the impact of different rule firings. In contrast, the AP5 active database system uses a special "merge" operator. The *merge*, denoted "&", of two non-*fail* deltas Δ_1 and Δ_2 is given by:

$$(\Delta_1 \ \& \ \Delta_2) = \begin{cases} \Delta_1 \cup \Delta_2 & \text{if this is consistent,} \\ fail & \text{otherwise.} \end{cases}$$

The merge of the two deltas of the previous example is *fail*, because of the presence of the conflicting signed atoms $+$*Orders*(*brakes*, 300, *Shimano*, 9/6/93) and $-$*Orders*(*brakes*, 300, *Shimano*, 9/6/93).

	Operator	Algebra	Heraclitus[Alg,C]
create	atomic	$+R(a_1,\ldots,a_n)$	`[ins` $R(\tau_1,\ldots,\tau_n)$`]`
		$-R(a_1,\ldots,a_n)$	`[del` $R(\tau_1,\ldots,\tau_n)$`]`
			`[mod` $R(\tau_1,\ldots,\tau_n\ ;\ \tau'_1,\ldots,\tau'_n)$`]`
	bulk		`bulk(ins` $R(\tau_1,\ldots,\tau_n)$, ρ`)`
			`bulk(del` $R(\tau_1,\ldots,\tau_n)$, ρ`)`
access	application	$apply(DB,\Delta)$	`apply` δ;
	peeking		`peekins(`R,δ`)`, `peekdel(`R,δ`)`
	hypothetical		ϵ `when` δ
combine	smash	$\Delta_1\ !\ \Delta_2$	δ_1 `!` δ_2
	compose		δ_1 `#` δ_2
	merge	$\Delta_1\ \&\ \Delta_2$	δ_1 `&` δ_2
	weak-merge	$\Delta_1\ w\&\ \Delta_2$	δ_1 `w&` δ_2

Fig. 3. Algebraic and language-level operators of Heraclitus[Alg, C].

As noted before, smash captures a form of composition, or sequential application, between deltas. In contrast, merge provides a commutative mechanism for combining pairs of deltas. Speaking intuitively, merge has a more declarative flavor than smash, because a merge of Δ_1 and Δ_2 is non-*fail* if the intended modifications represented by the two deltas do not conflict with one another. Merge is used in AP5 to combine the effects of rule firings; a transaction is aborted if the actions proposed by different rule firings conflict with each other. As detailed in Zhou and Hsu [1990], this semantics for rule application has similarities to the least-fixpoint semantics used in logic programming [Lloyd 1987], and can be used to obtain sufficient conditions on rule-bases that ensure consistent termination and order-independence of rule firing sequences.

Several other binary operators for combining deltas can be defined, for example, *weak-merge*, that is, union but delete all conflicting pairs of signed atoms (cf. Simon and de Maindreville [1988] and Cacace et al. [1990]), or union giving priority to inserts in the case of conflict. At present Heraclitus[Alg, C] provides explicit constructs for smash, merge, and weak-merge; other binary operators can be built up from more primitive Heraclitus[Alg, C] constructs.

3.2 Embedding Into C

In this and in Section 3.3 we describe how relational deltas and the algebraic operators previously described are embedded into C. The primary focus is on Heraclitus[Alg, C] expressions for (a) creating deltas, (b) combining deltas, and (c) accessing deltas. The algebraic and language-level operators of Heraclitus are summarized in Figure 3.

Heraclitus[Alg, C] supports the manipulation of both persistent and transient relations and deltas. Suppose that `Suppliers` and `Orders` are persistent relations as defined in the previous section. The following declares two variables that can hold pointers to these, and a variable that points at a transient relation Big:

```
relation *Supp, *Ord, *Big;
Supp = access_relation("Suppliers");
Ord = access_relation("Orders");
Big=empty_relation(part:char[30],qty:int,sup:char[30],exp:int);
```

Signatures for variables Supp and Ord are taken from the corresponding persistent relations. The signature[2] for transient relation variable Big is specified explicitly upon initialization. Here empty_relation() is a function that builds a new empty relation having the specified signature and returns a pointer to it. Although names may be associated with the individual coordinates (i.e., attributes) of relation types as indicated here, at present the algebra is based on coordinate positions. However, most of our examples use coordinate names to simplify the exposition. (We assume that Ord has the same field names as Big, and that Supp has field names sup and part.) Variables can also hold relations directly, without a level of indirection.

The algebra used is essentially the standard relational algebra, except that system- and user-defined scalar functions can be used in projection target lists, and in selection and join conditions. Transient relations can be made persistent using the make_persistent() function. To illustrate, suppose that foo is a user-defined scalar function in the following.

```
relation temp;
temp = project([part, qty*2], select({foo(sup)>qty}, Ord));
make_persistent("New_Temp_Reln", temp);
```

In the select expression, dereferencing of the value of Ord is automatic. Deltas are supported in Heraclitus[Alg, C] by the type delta. Deltas can be created using *atomic* commands, such as

```
delta D1, D2;
D1 = [del Supp("Campy", "pedals")];
D2 = [ins Big("brakes", 500, "Shimano", "9/20/93")];
```

After execution D1 has {−*Suppliers* (*Campy*, *pedals*)} and D2 has {+*_HERA14_*(*brakes*, 500, *Shimano*, 9/20/93)}, where *_HERA14_* is the relation identifier chosen during program execution for the transient relation Big. Variables can also hold pointers to deltas. Analogous to relations, deltas may be transient or persistent.

The *bulk* operator can be used to construct a "large" delta from data currently in the database. For example,[3]

```
bulk(ins Big(part, qty, sup, exp), select({qty > 300}, Ord))
```

[2]We assume that date values are stored as integers.

[3]As noted previously, in the current Heraclitus[Alg, C] prototype, coordinate positions rather than names are used. The coordinate positions are indicated using the "@" symbol. Typing information is also included here to simplify the task of preprocessing into C, given the fact that the signature of a relation variable is permitted to change over the lifetime of a program execution. The actual syntax of this command is bulk(ins Big(@c1, @i2, @c3, @i4),

evaluates in the context of state S_a (Figure 1) to

$$\{+\_HERA14\_(frame, 400, Trek, 8/31/93)\}.$$

More generally, the first argument to bulk must be what amounts to an atomic delta expression containing scalar expressions built up from column names and scalar values. These names are assigned possible values by the second argument to `bulk`, which must be a relation expression. Thus a `bulk` operator can be viewed as a composition of relational projection followed by parallel creation of atomic delta expressions.

Heraclitus[Alg, C] also supports atomic *modify* expressions, such as `[mod Ord("brakes", 150, "Campy", "9/1/93"; "brakes", 300, "Shimano", "9/6/93")]`. Evaluation of this expression depends on the current state: if (brakes, 150, Campy, 9/1/93) is present in `Orders` (as it is in state S_a) this expression evaluates to

$$\left\{\begin{matrix} -Orders(brakes, 150, Campy, 9/1/93), \\ +Orders(brakes, 300, Shimano, 9/6/93) \end{matrix}\right\}.$$

On the other hand, if (*brakes*, 150, *Campy*, 9/1/93) is not present in `Orders` (as in state S_b) then the expression evaluates to the empty delta. We have experimented with permitting explicit modifies inside delta values on an equal footing with deletes and inserts. However, as reported in Ghandeharizadeh et al. [1992], in the context of the pure relational model the semantics for consistency and for smash become quite cumbersome if modify atoms are supported at the algebraic level. (If tuple identifiers are supported in the underlying model, as in Widom and Finkelstein [1990], then modify atoms are easily supported.) This has led us to the compromise that modifies can be written at the language level explicitly, but their value depends on the state. Regardless of this decision, the presence of modify expressions in a program may give the compiler opportunities for optimization (e.g., by avoiding two traversals of an index).

Heraclitus[Alg, C] also permits "wildcards" in delete and modify commands. When used in atomic deletes and the left-hand part of atomic modifies, wildcards, denoted by "*," match any value. Evaluation of expressions with wildcards again depends on the current database state. As a simple example, `[del Supp("Campy", *)]` evaluates to

$$\left\{\begin{matrix} -Suppliers(Campy, brakes), \\ -Suppliers(Campy, pedals) \end{matrix}\right\},$$

when applied in the context of the state S_a. When a wildcard is used in a given coordinate in the right-hand part of a modify command, it takes on the value on the same coordinate of the tuple being replaced. For example,

`select({@i2 > 300}, Ord))`, where "c" indicates the type character string, and "i" indicates integer.

in state S_a the expression `[mod Ord("brakes", *, "Campy", "9/1/93"; "brakes", 300, "Shimano", *)]` evaluates to

$$\begin{Bmatrix} -Orders(brakes, 150, Campy, 9/1/93), \\ +Orders(brakes, 300, Shimano, 9/1/93) \end{Bmatrix}.$$

Complex algebraic expressions can be created by using the binary operators smash (`!`), merge (`&`), and weak-merge explicitly. A fourth operator, *compose* (#), is also supported; this is described in Section 3.3.

We now turn to the four operators for accessing deltas. The first is *apply*: the command `apply` δ first evaluates δ and then applies the resulting delta value to the current state. (If δ evaluates to *fail*, then `apply` δ yields a run-time error.) Although the delta expression δ may evaluate to a delta Δ containing many signed atoms, the command `apply` δ is viewed as a nondecomposable or "atomic" command, that is, all of Δ is applied, or an abort occurs.

Hypothetical expression evaluation is supported by the when operator. As a simple example,

```
Big = select({qty > 300}, Ord) when
    ([mod  Ord("brakes",  *,  "Shimano",  *;  "brakes",  500,
      "Shimano", "9/20/93")]
    & [ins Ord("light", 300, "Cat Paw", "9/3/93")]);
```

when evaluated in state S_b yields {(*frame*, 400, *Trek*, 8/31/93), (*brakes*, 500, *Shimano*, 9/20/93)}. Note that side effect free functions (more precisely, functions that do not modify the database state) can be called within the context of a when.

Nesting of whens is also permitted, and it is easily verified that

$$(E \textit{ when } \delta_1) \textit{ when } \delta_2 \equiv E \textit{ when } (\delta_2 \;!\; (\delta_1 \textit{ when } \delta_2)).$$

This plays a key role in the implementation of delta expressions consisting of nested whens (see Section 4).

The final operators for accessing deltas involve "peeking"; these permit the programmer to directly inspect a delta. The expression peekins(R, δ) evaluates to the relation containing all tuples that are to be inserted into R according to the value of δ, and the expression peekdel(R, δ) evaluates analogously. For example, `peekdel(Supp, [del ("Campy", *)])` evaluates on state S_b to {(Campy, brakes), (Campy, pedals)}.

3.3 Loss of Intended Meaning

This section illustrates a subtlety concerning the intended meaning of delta values, and introduces the compose operator for delta expressions.

Suppose in the running example that all orders are to be increased by 10%. Consider the following program fragment.[4]

```
delta order_inc;
order_inc = bulk(mod Ord(part, qty, sup, exp; part, qty*1.1,
  sup, exp), Ord)
```

If this fragment were executed in the context of state S_a, then d1 would hold the delta

$$\Delta_3 = \begin{Bmatrix} -Orders(frame, 400, Trek, 8/31/93), \\ +Orders(frame, 440, Trek, 8/31/93), \\ -Orders(brakes, 150, Campy, 9/1/93), \\ +Orders(brakes, 165, Campy, 9/1/93), \end{Bmatrix}.$$

If this delta were applied to S_a, then all orders would be increased by 10%. As an alternative, the delta might be used hypothetically. For example, if total_brakes_on_order( ) is a side effect free function that returns the total number of brakes on order, then the expression

```
total_brakes_on_order( ) when order_inc
```

evaluated for state S_a would return the value 165.

Suppose now that an update is made to the database state, yielding the state S_b (Figure 2). In this case, the delta order_inc no longer corresponds to increasing each order by 10%. For example, the expression

```
total_brakes_on_order( ) when order_inc
```

will now return the value 165 + 300 = 465. What went wrong? The problem is that the variable order_inc holds the result of evaluating the delta expression with reference to state S_a, and does not carry the same "intended meaning" in connection with the new database state S_b. This is reminiscent of the "phantom problem" [Eswaran 1976] that can arise when executing database transactions. As a simple example of the phantom problem, imagine a transaction that is intended to decrease by 20% all orders that have quantity ≥ 100. Imagine further that it is written naively to examine each tuple (p, q, s, e), to delete that tuple, and to append $(p, q * .8, s, e)$ to the end of the relation. If care is not taken, the newly inserted tuples will also be visited by the transaction, and the net result will be that all orders will be reduced to have a quantity lower than 100 (see Gray and Reuter [1993]).

Because the "intended meaning" of a delta can be lost, care must be taken when using a delta that was created with reference to a previous database state. Heraclitus[Alg, C] currently provides no support for testing

[4]In a C program, the order_inc assignment should type cast the expression qty*1.1 to an integer, for example, using (int)(qty*1.1). In the interest of readability, we generally do not include type casting in this article.

whether a delta value is "out of date," nor for preventing the application of "out of date" deltas. Thus in its present form Heraclitus[Alg, C] is best suited for environments where deltas do not lose their "intended meaning" between the time of creation and the times of usage. This is easily ensured if deltas are created and used within a single transaction (as arises in active database execution models), and it is relatively easy for a user to ensure in the context of a single-user database environment (as found in many personal-computer DBMSs). Richer environments might also satisfy this requirement, for example, a concurrent environment where the database schema is partitioned, with each user creating deltas that range only over a privately controlled component of the partition. More generally, the issue of detecting conflict between deltas, and whether a delta has become "out of date" is a topic of current research [Doherty and Hull 1995; Doherty et al. 1995a].

As an aside, we note that delta functions, that is, functions that return deltas, can be used to capture the intended meaning of hypothetical updates in a manner that is independent of the underlying state. For example, consider the function

```
delta increase_orders_by_10_percent()
{
      return bulk(mod Ord(part, qty, sup, exp; part, qty*1.1,
       sup, exp), Ord);
}
```

Now the expression `increase_order_by_10_percent()` will return a delta value that corresponds to increasing all orders by 10%, relative to the *current* database state. In particular, the expression

```
total_brakes_on_order() when increase_orders_by_10_percent()
```

when evaluated in state S_a will yield 165, and when evaluated in state S_b will yield 330. A more general perspective on the "intended meaning" of delta values is provided in Section 5.

It was noted earlier that at the algebraic level, the smash operator acts as a composition operator; that is, given a state S and delta *values* Δ_1 and Δ_2,

$$apply(S, \Delta_1 ! \Delta_2) = apply(apply(S, \Delta_1), \Delta_2).$$

Because of the potential loss of intended meaning resulting from state changes, this equivalence does not hold for delta *expressions*. For example, the equivalence does not hold for

$\delta_1 \equiv$ `[ins Ord("light", 400, "Cat Paw", "9/18/93")]`

$\delta_2 \equiv$ `[mod Ord(*, *, 400, *; *, *, 450, *)]`.

In particular, in the context of state S_a,

$$\delta_1 \quad \text{evaluates to } \{+\mathit{Order}(\mathit{light}, 400, \mathit{Cat\ Paw}, 9/18/93)\};$$

$$\delta_2 \quad \text{evaluates to } \left\{\begin{matrix} -\mathit{Order}(\mathit{frame}, 400, \mathit{Trek}, 8/31/93), \\ +\mathit{Order}(\mathit{frame}, 450, \mathit{Trek}, 8/31/93) \end{matrix}\right\};$$

$$\delta_1 \ !\ \delta_2 \text{ evaluates to } \left\{\begin{matrix} +\mathit{Order}(\mathit{light}, 400, \mathit{Cat\ Paw}, 9/18/93), \\ -\mathit{Order}(\mathit{frame}, 400, \mathit{Trek}, 8/31/93), \\ +\mathit{Order}(\mathit{frame}, 450, \mathit{Trek}, 8/31/93) \end{matrix}\right\}.$$

If δ_1 ! δ_2 is applied, then Order will include the tuple (light, 400, Cat Paw, 9/18/93). However, in the state resulting from `apply` δ_1; `apply` δ_2, no order will have quantity equal to 400.

The *compose* operator, denoted "#", has the property that the command `apply` (δ_1 # δ_2) is equivalent to `apply` δ_1; `apply` δ_2. In Heraclitus[Alg, C], compose is defined in terms of smash and when, by δ_1 # δ_2 = δ_1 ! (δ_2 `when` δ_1). With δ_1 and δ_2 as before,

$$\delta_1 \ \#\ \delta_2 \text{ evaluates to } \left\{\begin{matrix} -\mathit{Order}(\mathit{light}, 400, \mathit{Cat\ Paw}, 9/18/93), \\ +\mathit{Order}(\mathit{light}, 450, \mathit{Cat\ Paw}, 9/18/93), \\ -\mathit{Order}(\mathit{frame}, 400, \mathit{Trek}, 8/31/93), \\ +\mathit{Order}(\mathit{frame}, 450, \mathit{Trek}, 8/31/93) \end{matrix}\right\}.$$

This definition illustrates the difference between smash and compose. In `apply` (δ_1 ! δ_2), both δ_1 and δ_2 are evaluated with respect to the current state, then smashed, and then applied to the current state. In other words, `apply` (δ_1 ! δ_2) is equivalent to `d1` = δ_1; `d2` = δ_2; `apply d1; apply d2`. In contrast, δ_1 # δ_2 is equivalent to `d1` = δ_1; `apply d1; d2` = δ_2; `apply d2`. It is straightforward to verify that compose is associative. (In other data models, the compose operator may not be definable in terms of smash and when; see Section 5.)

Compose is especially useful in the context of hypothetical database access. We present an example involving two functions. The first function builds a delta that has the effect of canceling all October orders:

```
delta cancel_Oct_orders()
{
        return bulk(del Ord(*, *, *, Exp),
           select({in_Oct(Exp)},Ord));
}
```

The second one builds a delta that delays the expected date by two weeks of all orders with `Qty` > 500:

```
delta delay_big_orders()
{
      return bulk(mod Ord(part, qty, sup, exp; part, qty,
       sup, add_two_weeks(exp)),
                      select({qty > 500}, Ord));
}
```

Suppose as before that the function `total_brakes_on_order` computes the total number of brakes on order. Then the expression

```
total_brakes_on_order() when
 cancel_Oct_orders() # delay_big_orders()
```

performs a hypothetical evaluation of `total_brakes_on_order`, assuming that first the October orders where canceled, and then the big orders were delayed. Note the value resulting from the call to `delay_big_orders` takes into account the updates proposed by the value of `cancel_Oct_orders`. The following performs the hypothetical evaluation, but with the application of the two delta functions reversed.

```
total_brakes_on_order() when
  delay_big_orders() # cancel_Oct_orders()
```

In general these two expressions will evaluate to different values.

3.4 A Note on Concurrency

We conclude this section with a brief discussion of concurrency control in connection with Heraclitus[Alg, C]. At an abstract level, the goals of the Heraclitus paradigm are largely orthogonal to the goals of traditional concurrency control, although there is some interaction between them. In its present form, the Heraclitus[Alg, C] prototype provides minimal support for concurrent access to the persistent store. Specifically, portions of a Heraclitus[Alg, C] program may be surrounded by the commands `begin_transaction` and `end_transaction`. Nested transactions are not supported because we have not as yet extended the underlying storage manager, Exodus version 3.1 [Carey et al. 1986], to support this concept. As noted previously, tools are not currently provided to detect or prevent the use of "out of date" deltas.

4. THE IMPLEMENTATION OF HERACLITUS[ALG, C]

The implementation of Heraclitus[Alg, C] has two components: HERALD, a library of functions supporting the relational and delta operators, and a preprocessor that maps Heraclitus[Alg, C] programs into C programs with calls to HERALD. Developing and experimenting with this implementation has given us additional insight into the semantics of the Heraclitus paradigm and its operators, as well as providing a testbed for implementation strategies.

File services, persistence, and concurrency control in HERALD are provided by the Exodus Storage Manager, version 3.1 [Carey et al. 1986]. We have implemented relational algebra and delta operators using Exodus.

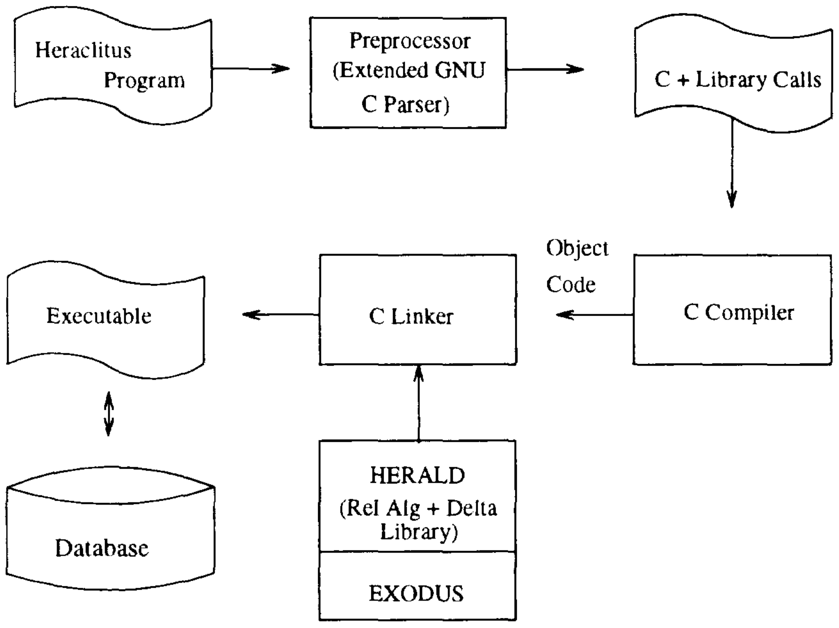


Fig. 4. Building components of Heraclitus[Alg, C].

HERALD is quite independent of Exodus and could be ported to other file systems with minimal difficulty. (Indeed, our first Heraclitus prototype [Ghandeharizadeh et al. 1992] was developed on top of WiSS [Chou et al. 1985].)

Figure 4 presents the overall architecture of the system. Given a Heraclitus[Alg, C] program, we use a preprocessor to transform this into a C program with calls to the HERALD library. This C program is compiled and linked with HERALD to generate an executable program that accesses and manipulates the database.

The preprocessor and HERALD are now described in more detail. The section closes with a description of benchmarking experiments performed with the prototype.

4.1 The Preprocessor

The preprocessor for Heraclitus[Alg, C] was constructed by modifying a GNU C compiler (available from the public domain). The primary challenge in developing the preprocessor was to develop a systematic understanding of the semantics of complex Heraclitus expressions involving the when operator. In this section we first describe this semantics, which is of independent interest and relevant to any implementation of the Heraclitus paradigm. We then describe how the semantics is supported in the current Heraclitus[Alg, C] preprocessor.

Similar to relational algebra queries, Heraclitus expressions can be translated into a tree format, where each node represents an operator that

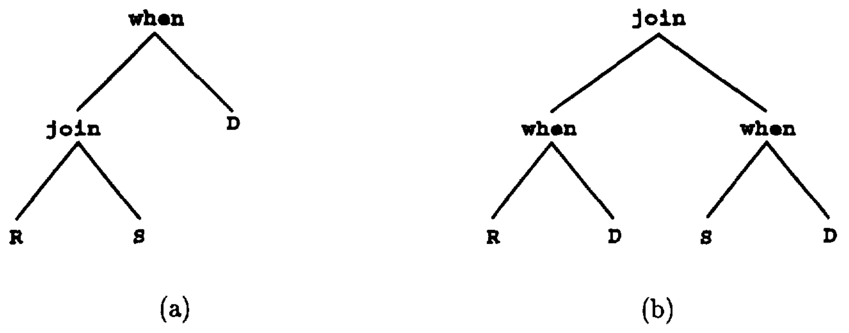


Fig. 5. Syntax tree and corresponding executable tree.

manipulates one or two inputs. The result is called a *syntax tree*. By analogy with the relational algebra, we would expect that syntax trees could be evaluated in a bottom-up fashion, using a unary or binary operator for each internal node. However, the presence of `when` operators requires that a transformation be made to syntax trees before they can be evaluated. The results of this transformation are called *executable trees*. (These correspond to the notion of "query trees" processed by query interpreters of conventional relational DBMSs; we use the more general term "executable" because not all expressions considered here are queries.) The transformation from syntax trees to executable trees is now illustrated through several examples.

We begin with a very simple example. Consider the expression `join(R, S) when D` where `R` and `S` are relation variables and `D` is a delta variable. Figure 5(a) shows the syntax tree of this expression. Suppose that this is to be evaluated using separate, independent operators for `join` and `when`. Under a bottom-up evaluation, first the values of `R` and `S` would be joined, and then the `when` operator applied. This evaluation would give the wrong result, because the contents of `D` refer to changes that are to be made (hypothetically) to `R` and `S`; but the impact of those changes cannot be determined by analyzing the output of `join(R, S)` independent of `R` and `S`.

The executable tree for the expression is shown in Figure 5(b). This was obtained by "pushing" the `when` through the `join` operator down to individual relation variables. Assuming again that each internal node represents an independent operator, this tree can be evaluated directly. Indeed, executable trees provide a naive but failsafe approach to reducing arbitrary Heraclitus expressions to a sequence of unary and binary operations.

Because expressions creating deltas may refer to relations, their evaluation can be influenced by the presence of a `when`. Consider the expression (referring to the example of Section 3)

```
bulk(del Supp(sup, part), select({qty<100}, Ord)) when D
```

In the executable tree corresponding to this expression the `when D` is "pushed" to the relation variable `Ord`, yielding the equivalent expression

```
bulk(del Supp(sup, part), select({qty<100}, (Ord when D)))
```

As a general rule, if nested whens are not present, then the executable tree corresponding to a syntax tree is obtained by "pushing" all whens to the leaves of the tree that hold relation variables.

Of course, direct evaluation of the executable tree of Figure 5(b) will be very inefficient. In a direct evaluation, first a copy of R would be constructed that reflects the changes called for in D; then a copy of S would be made with changes from D; and finally the join would be computed. As detailed in Section 4.2, HERALD provides library routines that can combine conventional relational algebra operators with when to provide more efficient implementation (in much the same way that project, select, and join are combined in conventional implementations of the relational algebra).

In the case of nested whens the transformation from syntax trees to executable trees is a bit more interesting. Consider first the following three expressions, where E is an arbitrary side effect free expression, and δ_1 and δ_2 are delta expressions.

$$e_1 \equiv E \texttt{ when} (\delta_1 \texttt{ when } \delta_2)$$

$$e_2 \equiv E \texttt{ when } (\delta_1 \; \# \; \delta_2)$$

$$e_3 \equiv (E \texttt{ when } \delta_1) \texttt{ when } \delta_2.$$

The following are equivalent expressions that correspond to the top level of the executable trees of these expressions. (Further transformation would probably be required to "push" whens into the expressions δ_1 and/or δ_2.)

$$e_1' \equiv E \texttt{ when } (\delta_1 \texttt{ when } \delta_2)$$

$$e_2' \equiv E \texttt{ when } (\delta_1 \;!\; (\delta_2 \texttt{ when } \delta_1))$$

$$e_3' \equiv E \texttt{ when } (\delta_2 \;!\; (\delta_1 \texttt{ when } \delta_2)).$$

The first expression is not transformed. The second expression is transformed directly according to the syntactic definition of compose. The transformation of the third expression corresponds to an algebraic identity of the when operator (see Section 3). As an aside, it follows from this identity that

$$(E \texttt{ when } \delta_1) \texttt{ when } \delta_2 \equiv \texttt{E when } (\delta_2 \; \# \; \delta_1).$$

In order to create the executable tree for an expression with multiple nested when operators, we can repeatedly perform the translation step used previously for e_3 in a top-down manner. For example, consider the syntax tree representation of the algebraic expression $((E$ when $\delta_1)$ when $\delta_2)$ when δ_3 shown in Figure 6(a). In the first iteration, we perform the translation on the when node at the root to construct the tree in Figure 6(b). We apply the

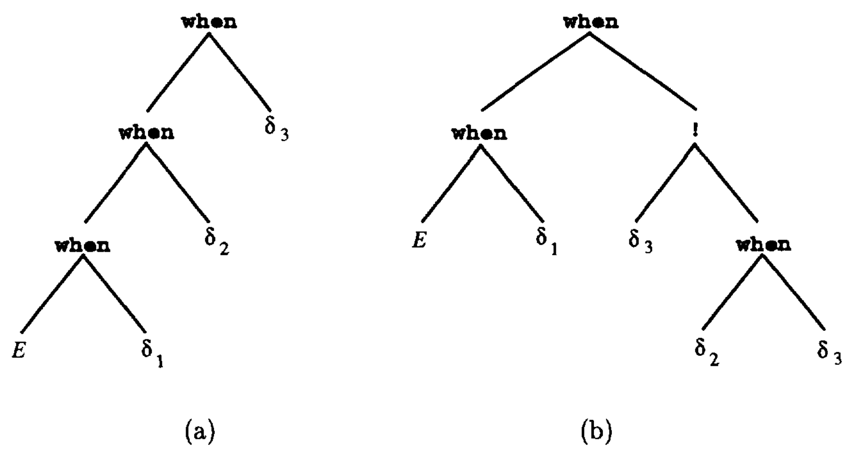


Fig. 6. First step in translation involving nested whens.

translation step once again to obtain the tree in Figure 7, which forms the top portion of the executable tree corresponding to the original expression. (Rewriting the when modifying δ_2 before rewriting the when at the root yields a different but equivalent executable tree.) Note that the executable tree of Figure 7 has two common subtrees (circled with dotted lines). These could be detected at compile time and evaluated only once.

In order to understand the current implementation of the Heraclitus[Alg, C] preprocessor, we must compare conventional query languages with DBPLs. In conventional query languages, even when embedded into an imperative language, the trees corresponding to the queries can be constructed at compile-time, and a considerable amount of optimization can also be performed then. In contrast, in full-fledged DBPLs, expressions might involve function calls and so the full expression may not be available until run-time. As a simple example in the context of Heraclitus, consider the expression `foo(u,v)` when δ_1, where `foo()` contains a conditional, and one of the return clauses of `foo()` includes the subexpression `R` when δ_2. The expression `((R` when δ_2`)` when δ_1`)` might be produced at run-time, but it is impossible to predict at compile-time all possible expressions that might arise. The fact that expressions are not known at compile-time is especially important in the context of Heraclitus, where syntax trees cannot be interpreted directly.

The focus of the first phase of the Heraclitus project has been to create a working prototype. As a result we adopted a somewhat utilitarian approach, which leaves considerable room for optimization. The central idea of the current implementation is to maintain a "run-time when stack." During the execution of a program the top of the stack holds (conceptually, at least) a delta that reflects the full effect of all deltas relevant to the evaluation of the expression currently under consideration. The delta at the top of the run-time when stack is used to "filter" all relations that are accessed.

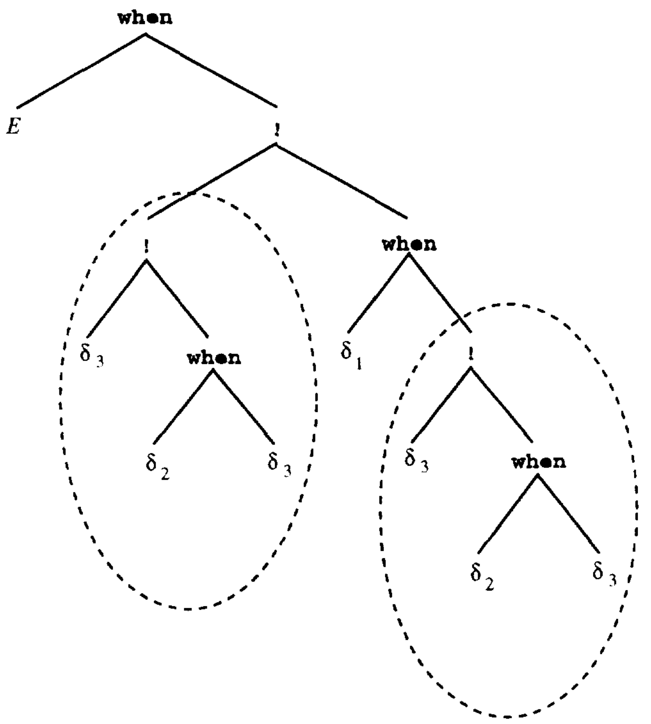


Fig. 7. Top portion of corresponding executable tree.

As a simple example, suppose that `foo(u, v)` when δ_1 is to be executed, and that the run-time when stack is currently empty. During execution, the value Δ_1 of δ_1 will be pushed onto the run-time when stack, and `foo( )` will be invoked. Suppose that inside `foo( )` an expression of the form E when δ_2 is encountered. First, δ_2 will be evaluated with reference to the current top of the run-time when stack. This will yield Δ_2, which has the value of δ_2 when δ_1. Second, Δ_1 ! Δ_2, or equivalently, the value of δ_1 ! (δ_2 when δ_1) = δ_1 # δ_2, is pushed onto the run-time when stack. Now E is evaluated with reference to the top of the stack, that is, in the hypothetical state corresponding to the result of applying the top of the stack. This yields the value of E when (δ_1 # δ_2) = (E when δ_2) when δ_1. Speaking intuitively, using the run-time when stack provides a dynamic realization of the transformation of syntax trees into executable trees.

The current implementation of the run-time when stack is eager, in the sense that each delta placed on the stack is fully evaluated. We are currently developing a lazy implementation, where the stack holds delta expressions that are evaluated only as necessary.

4.2 HERALD

The HERALD system is a library of routines that supports the full relational algebra, deltas, and the delta operators described in Section 3.

The implementation of the relational operators is very generic. A primary motivation for developing HERALD was to develop a basic understanding of how deltas and their operators might be implemented.

A central aspect of the HERALD system is to combine the evaluation of whens with evaluation of the algebraic operators. For example, suppose that R and S are relation variables, and δ a delta expression. HERALD provides a *hypothetical join* function `join_when`, that evaluates the expression `join(`$\langle$*cond*$\rangle$`, R, S) when` δ (or equivalently, `join(` $\langle$*cond*$\rangle$ `,R when` δ, `S when` δ)), without materializing `R when` δ or `S when` δ. HERALD provides hypothetical versions of all the relational operators.

HERALD currently supports two strategies for obtaining access to deltas in connection with the hypothetical algebraic operators and other delta operators, one based on hashing and the other on a sort-merge paradigm. A third approach to accessing deltas, based on B^+-trees, is currently under development.

Conceptually, HERALD represents a delta as a collection of pairs (R_Δ^+, R_Δ^-), specifying the proposed inserts and deletes for each relation variable R in the program. Here, R_Δ^+ and R_Δ^- are called *subdeltas*, and are stored as relations (actually, files) in Exodus. Hash-based access is best suited for the situation where a subdelta pair (R_Δ^+, R_Δ^-) fits into (the page buffer supported in) main memory, and sort-based access is more appropriate when a subdelta pair is larger than main memory (see Section 4.3). With large subdeltas, the current hash-based implementation may cause the buffer pool to exhibit a thrashing behavior. With both the hash-based and sort-based access, HERALD updates the system catalogue to maintain either the existence of a hash index or a sorted relation order. This persistent information enables HERALD to eliminate redundant work (e.g., sorting an already sorted relation) both during the execution of a program and across repeated execution of programs that employ the same set of relations and deltas. These two approaches to delta access are discussed in detail shortly.

The focus of the prototype implementation of Heraclitus[Alg, C] was to provide strong evidence that deltas can be incorporated into a relational DBMS with nominal loss of efficiency. Conceptually, it is straightforward to extend our hash-based implementation to form variations of the Grace [Kitsuregawa et al. 1983] and Hybrid hash join[5] techniques, so that thrashing behavior is eliminated for deltas that are considerably larger than the main memory buffer. The design, implementation, and evaluation of these algorithms as compared to sort-merge has been studied extensively [Schneider and DeWitt 1989; DeWitt et al. 1990]. We expect that the extensions of these algorithms to incorporate deltas and hypothetical operators would behave similarly.

An alternative implementation for deltas and their operators would be to build directly on top of an existing relational DBMS, where each subdelta is represented as a relation. In general this would be less efficient than if the

[5]See Bratbergsengen [1984], DeWitt et al. [1984], DeWitt and Gerber [1985], and Shapiro [1986].

techniques of HERALD were used. To see why, consider an expression `project([⟨att-list⟩], R) when D`, where the subdeltas for `R` in `D` are R^+ and R^-. This expression would be rewritten as `project([<att-list>], (R` $\cup R^+) - R^-$`)`, and then passed to the optimizer of the relational DBMS for evaluation. Most commercial optimizers for SQL or the relational algebra provide little optimization for expressions involving difference. Typically the full value of `(R` $\cup R^+) - R^-$ would be computed before evaluation of the projection could begin. In contrast, the HERALD system provides clever algorithms for evaluating expressions involving the special kinds of difference that arise in connection with the delta operators.

In the remainder of this section we discuss hash-based and sort-based access to deltas. We develop analytical models to estimate the number of disk I/O operations performed by each approach. These models can be extended to incorporate the CPU cost associated with comparing keys or hashing keys in main memory [Shapiro 1986].

4.2.1 *Hash-Based Access to Deltas.* When subdeltas are small enough to fit in main memory, HERALD maintains a hash index on each subdelta. The hash index key value to address this hash table is composite and computed based on the values of all fields (or attributes) of a record. In the following we describe the low-level algorithms for three representative delta operators, namely, `apply`, `select_when`, and `join_when`. The algorithms described here assume that no indexing is provided for base relations. We have developed and implemented analogous algorithms to support the composition of the other algebraic operators with when. We omit consideration of the merge and smash operators, as their implementation closely follows that of their respective logical implementations described in Section 3.

Apply. The inputs to this operator are a relation R and a delta Δ. It removes from R the tuples in R_Δ^- and inserts into R the tuples in R_Δ^+ (without introducing any duplicates). Our implementation of this operator is as follows:

```
(1) Open a scan on relation $R$.
(2) For each tuple $t$ in $R$,
    (a) Probe the hash index of $R_\Delta^-$ with $t$.
    (b) If a match is found, then delete $t$. Otherwise, probe the hash index
        of $R_\Delta^+$ with $t$; if a match is found then mark the tuple in $R_\Delta^+$.
(3) Open a scan on $R_\Delta^+$.
(4) For each tuple $t$ in $R_\Delta^+$,
    (a) if $t$ is not marked, insert $t$ into $R$ and clear its marked bit;
    (b) if $t$ is marked, clear its marked bit.
```

The "marking" of tuples in $R_\Delta^+ \cap R$ prevents the introduction of duplicates into the output. The marking and clearing of tuples in R^+ does not incur additional page I/Os in cases where R_Δ^+ and R_Δ^- fit into main memory.

Marking a tuple of R_Δ^+ and clearing it can be an expensive operation as it causes a disk page to become dirty, which may result in unnecessary disk writes to reflect the setting of a bit that ultimately will be cleared. This suggests a further optimization to control the buffer page replacement policy to eliminate the possibility of redundant disk writes.

Select_when. The input arguments of this operator are: a relation R, a selection condition, a delta Δ, and an output relation. Logically, this operator selects tuples of R that satisfy the selection condition in the hypothetical state proposed by Δ and stores the resulting tuples in the output relation. Its implementation is as follows:

```
(1) Open a scan on $R$.
(2) Initialize $t$ to hold the first tuple of $R$.
(3) While not $eof(R)$,
    (a) Evaluate the selection condition for $t$. If the tuple does not
        qualify go to step e.
    (b) Probe the hash index of $R_\Delta^+$ with $t$ for a matching tuple, if found
        go to step e.
    (c) Probe the hash index of $R_\Delta^-$ with $t$ for a matching tuple, if found
        go to step e.
    (d) Insert $t$ into the output relation.
    (e) Get the next tuple $t$ in $R$.
(4) For each tuple $s$ of $R_\Delta^+$ do
    (a) Evaluate the selection condition for $s$. If s satisfies this condi-
        tion, then insert $s$ into the output relation.
```

Note that we probe the hash index only if the tuple satisfies the selection condition. This minimizes the number of disk accesses because probing the hash index may result in a disk read operation.

We briefly analyze the expected I/O costs of this implementation of `select_when`. Suppose that R_Δ^+, R_Δ^- are small enough to fit into main memory, and that $s\%$ of the tuples in R satisfy the selection condition. Assuming that $s > 0$, the algorithm will call for the following I/Os:

(a) Scan R.
(b) Probe R_Δ^- for $s\%$ of R.
(c) Probe R_Δ^+ for $s\%$ of R.
(d) Scan R_Δ^+.
(e) Write output relation.

Thus the expected overhead in I/O (as compared to the standard select operator) is roughly equal to the number of pages of the hash tables for R_Δ^+ and R_Δ^+, and the number of pages of R_Δ^+ and R_Δ^- that are read during parts (b), (c), and (d). (An additional scan of all of R_Δ^+ and R_Δ^- is needed if hash tables for these have not yet been created.)

Our benchmarking experiments confirm this analysis. Specifically, they show that if hash tables already exist for the subdeltas, then the overhead (in terms of page I/Os) incurred by the select_when operator is about the size of the delta as compared to the standard implementation of the conventional select operator. The overhead is about twice the size of the delta if the hash tables do not exist.

Join_when. In the current implementation, the binary relational operators use sort-based implementations. In the case of hash-based delta access, a key subroutine for all of them is sort_when. Suppose that R is unsorted. The conventional approach to sorting R is to use heap-sort on short (e.g., 100 page) segments of R, and then to perform n-way merges of these segments. In sort_when, the impact of a delta is incorporated into the heap-sort. For example, on relation R, as portions of R are read in for heap-sorting, a hash-table for R_{Δ} is probed, and the matching tuples are not placed into the heap. Also redundant tuples in $R_{\Delta}^{'}$ are marked, to prevent later duplication. After R is completely read, the remainder of $R_{\Delta}^{'}$ is also processed by the heap sort to provide additional sorted segments. Then one or more merges is invoked to create a sorted file. In the current implementation for join with hash-based delta access, sort_when is used to sort R (as influenced by $R_{\Delta}^{'}$, R_{Δ}) and S (as influenced by $S_{\Delta}^{'}$, S_{Δ}), and then a binary merge is used to create the join. In reality, the implementation is slightly more efficient: the merges for sorting R and S are combined with the binary merge for creating the join.

When using hash-based delta access for these operators, there is an important interaction between the amount of buffer space used by the heap versus the hash tables. To illustrate, suppose in the abstract that the total available buffer pool consists of 100 pages (and so the heap-sort can perform 100-way merges). Moreover, assume that R consists of 1000 pages, R has about 90 pages that will be probed during a pass of R (termed "hitting" pages), and $R^{'}$ is empty. In this case a 10-page heap could be established, and $R - R$ would be broken into roughly 100 (or fewer) sorted segments. Now a single 100-way merge will yield a sorted version of *apply*(R, R); total cost is $2|R| + |R \;|$ page I/Os. Suppose now that R has 2000 pages, R^{-} has about 80 "hitting" pages, and $R^{'}$ is empty. It is now optimal to devote 20 pages to the heap-sort and the other 80 to hash probing. (Fewer pages for the heap-sort result in more merge passes; and fewer pages for the hash probing may result in thrashing.) Thus providing optimal support for hash-based delta access requires the ability to dynamically partition the buffer pool between these two tasks. This capability is supported by Exodus, and we plan to investigate these tradeoffs in our future research.

4.2.2 *Sort-Based Access to Deltas.* A delta may be so large that it does not fit in main memory, causing the simple hash-based implementation previously described to thrash at the buffer pool level. To remedy this, we have designed and implemented algorithms that access deltas using a sort and merge technique. We now briefly describe the low level algorithms for apply, select_when, and join_when; the implementation of other operators is analogous. Heraclitus[Alg, C] maintains information on whether relations and subdeltas are sorted, so that one or more of the sorting steps of these sort-based algorithms can be eliminated.

Apply. The inputs to this operator are a relation R and a delta Δ with relevant subdeltas consisting of R_Δ^- and R_Δ^+.

```
(1) Sort $R$ in place.
(2) Sort $R_\Delta^-$ in place.
(3) Sort $R_\Delta^+$ in place.
(4) Open a scan on relation $R$, $R_\Delta^-$, and $R_\Delta^+$.
(5) Initialize r as the first tuple of $R$, $d-$ as first tuple of $R_\Delta^-$, and $d+$ as
    first tuple of $R_\Delta^+$.
(6) While not eof($R$) OR not eof($R_\Delta^-$) OR not eof($R_\Delta^+$),
    (a) Assign t to be the tuple with minimum value among $r$, $d+$, and $d-$.
    (b) If $t = r$ and $t = d-$, then delete $r$ from $R$.
    (c) If $t \neq r$ and $t = d+$, then insert $t$ to relation $R$.
    (d) If $t = r$, then get the next tuple $r$ from $R$.
    (e) If $t = d+$, then get the next tuple $d+$ from $R_\Delta^+$.
    (f) If $t = d-$, then get the next tuple $d-$ from $R_\Delta^-$.
```

This operator sorts each relation R, R_Δ^+, and R_Δ^-. Next it performs a three-way merge between these relations in order (1) to eliminate the tuples in R that are proposed to be deleted by R_Δ^-, and (2) to insert the tuples proposed by R_Δ^+ while maintaining a duplicate-free relation.

Select_when. The input arguments of this operator are: a relation R, a selection condition, a delta Δ, and an output relation. The algorithm presented here assumes that all the inputs are unsorted; it is easy to modify the algorithm if some of the inputs are sorted. A key function used here is select_sort which takes as input a relation and a selection condition. As with sort_when, this implements a two-phase sort, but in the heap-sort phase it deletes all tuples violating the selection condition.

In the following algorithm, if no tuples of R satisfy the selection condition (i.e., *Temp* is empty), then R_Δ^+ is scanned for the qualifying tuples and returns. Otherwise, it sorts the qualifying tuples found in each of R_Δ^- and R_Δ^+ into two different temporary relations. Next it performs a three-way merge on these relations, inserting one occurrence of entries of R that match with R_Δ^+ (prevent duplicates) and eliminating those that match with R_Δ^- (tuples proposed to be deleted).

```
(1) select_sort ($R$, selection condition) into a temporary relation $Temp$.
(2) If $Temp$ is empty, then
    (a) for each tuple $t$ of $R_\Delta^+$, evaluate the selection condition for $t$. If $t$
        satisfies this condition, then insert $t$ into the output relation.
    (b) Exit.
(3) select_sort ($R_\Delta^-$, selection condition) into a temporary relation $Temp^-$.
(4) select_sort ($R_\Delta^+$, selection condition) into a temporary relation $Temp^+$.
(5) Initialize $r$ as the first tuple of $Temp$, $d-$ as first tuple of $Temp^-$, and $d+$ as first tuple of $Temp^+$.
(6) While not eof($Temp$) OR not eof($Temp^-$) OR not eof($Temp^+$),
    (a) Assign $t$ to be the tuple with minimum value among $r$, $d+$, and $d-$.
    (b) If $t \neq d-$, then insert $t$ into the output relation.
    (c) If $t = r$, then get the next tuple $r$ from $Temp$.
    (d) If $t = d-$, then get the next tuple $d-$ from $Temp^-$.
    (e) If $t = d+$, then get the next tuple $d+$ from $Temp^+$.
```

We now analyze the expected I/O cost of this implementation of select-_when, under the assumption that the inputs are not maintained in sorted order. Let $P(R)$ represent the number of disk pages for relation R; $SP(R)$ represents the number of disk pages that satisfy the selection condition, and analogously for R_Δ^+ and R_Δ^-. We assume that $SP(R) \leq$ the square of the number of available pages in the buffer pool (i.e., that only one n-way merge is required to sort the qualifying tuples of R), and similarly for R_Δ^+ and R_Δ^-. The total number of I/Os incurred by the preceding algorithm can be estimated as the sum of:

$$select\_sort(R): \quad P(R) + 3*SP(R)$$

$$select\text{-}sort(R_\Delta^+): \quad P(R_\Delta^+) + 3*SP(R_\Delta^+)$$

$$select\text{-}sort(R_\Delta^-): \quad P(R_\Delta^-) + 3*SP(R_\Delta^-)$$

$$merge: 2*SP(R) + 2*SP(R_\Delta^+) + SP(R_\Delta^-).$$

This cost function is a worst-case estimate because it assumes: (a) none of the inputs are ordered, (b) $SP(R)$ is not zero, (c) the tuples of $SP(R)$ are not redundant with those in $SP(R_\Delta^+)$, causing all their entries to be written to the output relation, and (d) the tuples of $SP(R)$ do not match with the tuples found in R_Δ^-. If the total number of segments of qualifying tuples in R, R_Δ^+, and R_Δ^- after the heapsort phase in Steps (1), (3), and (4) is sufficiently small, then the merging phases of those steps can be combined with Step (6) for a savings of $2 * (SP(R) + SP(R_\Delta^+) + SP(R_\Delta^-))$ I/Os.

The preceding algorithm also handles the case where the input relation and delta are sorted. In this case, only Steps (5) and (6) of the algorithm are executed, and the selection condition is incorporated into Step (6).

Join_when. The algorithm for join (and the other binary operators) in this case is relatively straightforward. To compute the join of R and S

under Δ, first sort each of R, R_Δ^+, R_Δ^-, S, S_Δ^+, and S_Δ^- using attribute orderings stemming from the join condition. This step can be optimized if some of the inputs are already sorted. Then perform a 6-way merge, retaining all appropriate tuples. Actually, as with the hash-based implementation, if the inputs are sufficiently small, then HERALD combines the merging phase of the join with the separate merging phases of the sorting of the six relations, thereby eliminating one reading and writing of all these relations.

Another optimization, not implemented in the current version, is to provide hash-based access to one of the subdeltas and sort-based access to the other one.

4.3 An Evaluation of HERALD

Heraclitus[Alg, C] and the underlying library HERALD are currently operational on a Sun SPARCstation 2 using the UNIX operating system. We have used this implementation to characterize the performance of HERALD for executing the delta operators and hypothetical relational operators (some of these performance results appeared in Ghandeharizadeh [1993]; see Zhou et al. [1994] for detailed descriptions of the experiments conducted and the results obtained). The goals of this evaluation were to characterize the tradeoffs associated with the alternative techniques employed by HERALD, and quantify the different factors that have an impact on the performance of the implementation. We analyzed the alternative implementation of deltas as a function of alternative sizes for the buffer pool, page size,[6] relations, and deltas. Several other factors were also considered including conflicts between deltas, and the percentage of redundant tuples between two deltas.

The experiments confirmed the following hypotheses:

(1) The hash-based implementation is efficient for small deltas that fit in main memory. It degrades due to thrashing of the buffer pool as the delta size grows larger than main memory. In this case, the sort-based implementation is more appropriate. (Another approach in this case would be to employ either Grace or Hybrid hash-join algorithms to eliminate the thrashing behavior.)

(2) Given a fixed amount of memory, the number of I/Os performed by the sort-based implementation decreases as a function of larger disk page sizes, because each page-read brings in more tuples. The hash-based implementation also benefits from the larger disk page size to a certain point. However, very large page sizes lead to thrashing behavior with deltas that are only slightly larger than the memory size. This is because a larger page size results in partially empty hash buckets that reduce the probability of a memory hit. This is best illustrated with an

[6]It is difficult to change the page size in Exodus 3.1; the experiments involving page sizes were performed with the earlier Heraclitus[Alg, C] prototype [Ghandeharizadeh 1992] that was constructed using WiSS [Chou et al. 1985].

example. In the extreme case, memory may consist of one frame, and a hash table consist of two buckets, each half full of tuples. The two buckets compete for the available frame, resulting in a 50% hit ratio.

(3) When executing a program, if the order of records in both the referenced delta and relation are maintained, the sort-based access can provide performance identical to hash-based access for small deltas.

(4) Both the sort-based and hash-based implementations benefit from a "lazy" application of deltas as long as the deltas are smaller than the referenced relations. In particular, in the current HERALD prototype executing `apply(D1!D2)` is generally faster than executing `apply D1; apply D2`. This is because in the absence of indices on the base relations, execution of `apply D` involves a pass through each base relation for which `D` is nonempty. We expect that lazy application will remain cheaper than eager application even in the presence of B^+-tree indexes. This is because eager evaluation requires two sequences of access using a large B^+-tree (i.e., of some base relation) whereas lazy evaluation requires a sequence of accesses using small B^+-tree (i.e., to compute smash of deltas) followed by one sequence of accesses using a large B^+ tree (of some base relations).

5. A MODEL-INDEPENDENT PERSPECTIVE

The notion of deltas and their basic operators provides a powerful paradigm for supporting a variety of database applications across a wide spectrum of database models. In the preceding discussion we have focused on the development of a comprehensive realization of this paradigm and its application for the pure relational model. This section briefly presents an abstract, model-independent framework for the Heraclitus paradigm. This is currently being used as a starting point for the development of an object-oriented version of Heraclitus [Boucelma et al. 1995; Doherty et al. 1995a].

In principle, deltas can be adapted to any database model/programming language combination. For this discussion we assume a conventional imperative host language. In the context of DBPLs, variables may range over conventional data types, such as integer and string, and over bulk data types, such as relations, complex objects, class extensions, and so on. In a broad sense, relation names in a relational DBMS can be viewed as special global variables holding relations. In Heraclitus[Alg, C] a rather sharp distinction has been made between the bulk data type relation and other data types. In Heraclitus[Alg, C] deltas range exclusively over relations, and only relations and deltas can be persistent. In the model-independent perspective developed here, this distinction is relaxed: deltas may range over all data types except for deltas, and persistence is viewed as orthogonal to the type system and ignored. To simplify the exposition, many of the examples presented in this section focus on the type integer rather than bulk data types.

To introduce the model-independent perspective on deltas, we make use of the notations and techniques of formal semantic theory, albeit in a somewhat informal manner. The semantics of a programming language can be formally defined by giving "meaning functions" that map each syntactic domain to an appropriate semantic domain. To set the stage, we first present the formalism for the minimal elements assumed of an imperative host language.

Syntactic Domains

Expression: ranged over by E.

Command: ranged over by C. This typically includes both atomic commands and combinations of them, for example, through compose (";").

Semantic Domains

Value: This typically contains values such as integers and strings, and possibly more complex values such as arrays of integers, and bulk values such as relations. We assume a special element $error_{Value}$.

State: ranged over by σ and containing the special element $error_{State}$. Each non-error state is typically viewed as a function from program variables to values.

$FSV = State \rightarrow_s Value$ ("Functions from *State* to *Value*").

$FSS = State \rightarrow_s State$ ("Functions from *State* to *State*"), where $\perp \in FSS$ denotes the function that maps every state to $error_{State}$.

Meaning Functions

$\mathscr{E}$: *Expression* $\rightarrow FSV$.

$\mathscr{C}$: *Command* $\rightarrow FSS$.

The meaning $\mathscr{E}[\![E]\!] \in FSV$ of an expression E is a function that maps each state to a value. Specifically, $\mathscr{E}[\![E]\!](\sigma)$ is the value that the expression E takes in the context of state σ. The special value $error_{Value}$ represents the result of evaluating an expression containing an error. Similarly, the meaning $\mathscr{C}[\![C]\!] \in FSS$ of a command C is a function mapping states to states. Specifically, an initial state σ is mapped by $\mathscr{C}[\![C]\!]$ to the final state σ' that results from executing the command C starting in σ. The special state $error_{State}$ represents the result of executing a command containing an error. Thus the function $\perp \in FSS$ maps every initial state to the error final state. In the preceding, $A \rightarrow_s B$ denotes the set of functions from A to B that are *strict* with respect to errors, that is, for all $f \in FSV$, $f(error_{State}) = error_{Value}$ and for all $f \in FSS$, $f(error_{State}) = error_{State}$. The *compose* (";") operator on commands has the semantics

$$\mathscr{C}[\![C_1;\ C_2]\!] = \mathscr{C}[\![C_2]\!] \circ \mathscr{C}[\![C_1]\!],$$

where $\circ$ denotes function composition (and $f \circ g$ means apply g and then apply f).

We briefly illustrate the preceding notation. For example, $\mathscr{E}[\![\texttt{x + 1}]\!](\sigma) = \sigma(\texttt{x}) + 1$ says that the meaning of the expression x + 1 applied to the state σ is equal to the value of variable x in σ plus one. Similarly, $\mathscr{C}[\![\texttt{x := x+1}]\!](\sigma) = \sigma[\texttt{x}/\mathscr{E}[\![\texttt{x+1}]\!]\sigma]$ says that the meaning of the command x := x+1 applied to the state σ is equal to a state that is everywhere the same as σ except at x where it has the value $\mathscr{E}[\![\texttt{x+1}]\!](\sigma)$.

The model-independent perspective on the Heraclitus paradigm is presented in the form of a series of requirements that a language extension involving deltas should satisfy. After that, some algebraic properties that are implied by these requirements is presented.

A delta is an element of *Value* that represents a mapping from *State* to *State*. Deltas are generated by evaluating delta expressions, ranged over by δ. The most basic delta expression has the form [*C*] where *C* is a command: it is useful to think of this as a form of *delayed command*. For example, we might turn the command `x := 37` into the delta expression `[x := 37]`, which would evaluate to a delta proposing that the variable x be changed to 37. In Heraclitus[Alg, C] a very restricted class of commands can be used in expressions of the form `[C]` (namely, atomic inserts, deletes, and modifies), but this can be extended to include more general commands. Also, other mechanisms can be developed for specifying delta values (such as the `bulk` commands of Heraclitus[Alg, C]).

At the semantic level, a delta should have the same form as the meaning of a command. In other words, a delta should be a function from states to states.

Requirement 1 $\mathscr{E}[\![\delta]\!](\sigma)$: *FSS*. Speaking intuitively, Heraclitus[Alg, C] uses a natural tabular representation of deltas. This tabular representation was used in Section 3 to illustrate deltas conceptually, and also served as the basis of the implementation described in Section 4. It is important to bear in mind that the choice of how deltas are represented is an implementation detail; conceptually the family of supported deltas should be viewed as a possibly restricted family of functions from states to states.

Note that, in general, Requirement 1 introduces *recursive* domain equations: values include functions over states and states are functions mapping to values. Domain theory provides standard solutions to such equations [Stoy 1977]. Alternatively, if deltas cannot themselves range over deltas, then it is possible to avoid such elaborate constructions. This was the approach taken in Heraclitus[Alg, C], and the approach we take in the remainder of this discussion.

The most basic operation involving deltas is *application*: the command `apply(`δ`)` evaluates the delta expression δ and applies the resulting delta to the current state.

Requirement 2. $\mathscr{C}[\![\texttt{apply}(\delta)]\!](\sigma) = (\mathscr{E}[\![\delta]\!](\sigma))(\sigma)$. For example, execution of the command `apply([x := x+1])` begun in an initial state in which x is

36 results in a final state in which x is 37. In general, we require the following.

Requirement 3. $\mathscr{C}[\![\text{apply}([C])]\!] = \mathscr{C}[\![C]\!]$. A primary motivation for using Requirement 3 instead of a stronger condition concerns a tradeoff between the expressive power of the family of deltas incorporated and the efficiency with which they can be manipulated and accessed. To see this, first note that Requirement 3 is considerably weaker than requiring that the value of delta expression $[C]$ be *equal* to the meaning of command C; that is, $\mathscr{E}[\![[C]]\!](\sigma) = \mathscr{C}[\![C]\!]$. In the latter caes, the value of $\mathscr{E}[\![[C]]\!](\sigma)$ would be independent of the state σ, and it would evaluate to the very function specified by command C. In this case, deltas of the form `[C]` would most likely be stored as blocks of code, and delta application would consist of interpreting these code blocks. We refer to this semantics as the *pure-interpretation* semantics for deltas.

In contrast, Heraclitus[Alg, C] satisfies Requirement 3, but upon encountering an expression such as `[C] = [mod Ord("brakes", *, "Campy", "9/1/93"; "brakes", 300, "Shimano", *)]` (see Section 3) it essentially performs a "partial evaluation" or "boiling down" of the command `C`, representing it in a tabular format (involving atomic update commands) that depends on the underlying state. In particular, a delta represented in this tabular format does not contain any variables or nested operations, and applying a delta involves only very simple computational actions. We refer to this and analogous design choices as *partial-evaluation* semantics for deltas. As indicated in Section 3, even if a given realization of the Heraclitus paradigm uses partial-evaluation semantics, the full-interpretation semantics can always be simulated by wrapping expressions of the form `[C]` in delta functions.

Behaviorally, the difference between the partial-evaluation semantics and the pure-interpretation semantics is that, in the former case, if the state of the database *changes* between the time a delta is created and the time it is applied or hypothetically accessed, then the delta may lose its "intended meaning". As a simple example, suppose that deltas of the form `[x := . . .]` are partially evaluated into the form $\{x = c\}$, where c is a constant. After executing

```
var d:delta;
    x:integer;
x := 36;
d := [x:=x+1];
x := 0;
apply(d);
```

x has the value 37, even though x has the value 0 immediately before application of d. This was illustrated in the context of Heraclitus[Alg, C] in Section 3.3.

Speaking intuitively, in a partial-evaluation semantics the reduction of expressions [C] into a tabular format can be viewed as part of the work in executing C. If the resulting delta value is to be applied or used hypothetically several times (as illustrated for active database execution models in Section 6), then the overall cost of accessing [C] is reduced because the first part of the evaluation occurs only once. This is particularly useful in the context of bulk data types. Another motivation for the partial-evaluation semantics is that a tabular representation of deltas can allow for efficient implementations of operators to access and manipulate deltas.

There are a range of possibilities with respect to the choice of semantics for deltas, with pure-interpretation at one end, and including several possible forms of partial-evaluation. We say that one kind of delta is *more expressive* than another if the set of functions represented by the first kind properly contains that set of functions represented by the second kind. In this article, in connection with the relational model we have focused on a tabular representation of delta values involving atomic inserts and deletes. Ghandeharizadeh et al. [1992] consider a more expressive kind of delta value, in which atomic modifies may also be present. Doherty et al. [1995a] consider several choices of partial-evaluation semantics deltas in an object-oriented database model.

If a partial-evaluation semantics is used in developing an instance of the Heraclitus paradigm, primary issues are:

(a) which commands C can be used in delta expressions $[C]$; and
(b) to what extent should expressions of the form $[C]$ be partially evaluated when forming the corresponding delta values; that is, what is the form of the tabular representation of deltas, if any.

The second point is pivotal in connection with the tradeoff between expressive power and efficiency of implementation. Speaking intuitively, if one kind of delta has more expressive power than a second, then the chance of losing "intended meaning" is lower for the first kind of delta. On the other hand, more expressive power generally implies a higher conceptual complexity of delta operators, and a lower efficiency with regard to manipulation and access.

We now return to requirements of the Heraclitus paradigm. *Hypothetical access* is accomplished using the when operator.

Requirement 4 $\mathcal{E}[\![E \text{ when } \delta]\!](\sigma) = \mathcal{E}[\![E]\!]((\mathcal{E}[\![\delta]\!](\sigma))(\sigma))$. It is assumed here that expression E has no side effects (or more precisely, no side effects on the portion of store over which deltas range). For example, evaluation of the expression x when [x := x+1] in a state in which x is 36 results in the value 37 but leaves x unchanged.

Delta expressions are built up from atomic delta expressions using a collection of operators. In addition to the data-model dependent atomic delta expressions $[C]$, there are two data-model independent atomic delta expressions: empty and fail.

Requirement 5

$\mathscr{E}[\![\texttt{empty}]\!](\sigma)$ = the identity function on states;
$\mathscr{E}[\![\texttt{fail}]\!](\sigma) = \bot$.

From this and Requirement 2, it follows that $\mathscr{E}[\![\texttt{apply (empty)}]\!](\sigma) = \sigma$ and $\mathscr{E}[\![\texttt{apply(fail)}]\!](\sigma) = error_{State}$. This requirement 4 imply that $\mathscr{E}[\![E$ `when empty`$]\!](\sigma) = \mathscr{E}[\![E]\!](\sigma)$ and $\mathscr{E}[\![E$ `when fail`$]\!](\sigma) = error_{Value}$. The delta associated with the expression `fail` is denoted by *fail*.

The operator *smash* on deltas, denoted "!," corresponds to function composition.

Requirement 6. $\mathscr{E}[\![\delta_1 ! \delta_2]\!](\sigma) = \mathscr{E}[\![\delta_2]\!](\sigma) \circ \mathscr{E}[\![\delta_1]\!](\sigma)$. For example, evaluation of the expression `[x:=37] ! [y:=38]` results in a delta proposing that the variable x be changed to 37 and the variable y be changed to 38. More interestingly, evaluation of the expression `[x:=37] ! [x:=38]` results in a delta proposing that the variable x be changed to 38, that is, the second update smashes the first one. With the simple tabular representation of deltas in Heraclitus[Alg, C], the smash operator has a very straightforward implementation, namely, union with preference to the second operand in case of conflict. We expect that smash will also have a conceptually simple implementation in other data models whenever a tabular representation for deltas is used.

The operator *compose* on deltas, denoted "#," is like smash except that the second delta expression is evaluated in the state that would result if the value of the first delta expression were applied to the current state.

Requirement 7. $\mathscr{E}[\![\delta_1 \# \delta_2]\!](\sigma) = \mathscr{E}[\![\delta_2]\!](\Delta(\sigma)) \circ \Delta$ where $\Delta = \mathscr{E}[\![\delta_1]\!](\sigma)$. For example, evaluation of the expression `[x:=37] # [y:=x+1]` results in a delta proposing that the variable x be changed to 37 and the variable y be changed to 38: the changes proposed in the first delta are taken into account in creating the second delta. As discussed in Section 3, in the context of Heraclitus[Alg, C] compose for delta expressions can be defined in terms of smash and when; that is, $\delta_1 \# \delta_2$ abbreviates δ_1`!(`δ_2 `when` δ_1`)`. For the model-independent perspective we choose to define compose from first principles because, in a context where expression evaluation can have side effects, we want to make it clear that the first delta expression should be evaluated only once. For example, in an object-oriented database model, the delta expression δ_1 might propose the use of a "new" object identifier. In the formal definition of compose given by Requirement 7, this "new" identifier will be created/allocated only once, whereas in the expression δ_1`!(`δ_2 `when` δ_1`)` two identifiers will be created/allocated, one each for the two occurrences of the expression "δ_1." Similar problems arise with the expression δ_1`!(`δ_2 `when` δ_1`)` in the context of tuple identifiers, and in other models where delta evaluation has side effects on the overall program state.

Note that the compose operator is of interest only in the context of partial-evaluation semantics. In the full-interpretation semantics, where the value of a delta expression does not depend on the current state, compose is the same as smash. Even in the context of partial-evaluation semantics there are times when smash and compose are equivalent. For example, suppose that variables D1 and D2 hold delta values. Then[7] D1 # D2 ≡ D1 ! D2, because the value held by D2 is independent of the current state, and so $\mathcal{E}[\![\mathrm{D2}]\!](\sigma) = \mathcal{E}[\![\mathrm{D2}]\!]((\mathcal{E}[\![\mathrm{D1}]\!](\sigma))(\sigma))$.

The operators when, smash, and compose form the basis of the Heraclitus paradigm. In Heraclitus[Alg, C] the bulk operators are also provided for creating deltas, the merge and weak-merge operators are provided for combining them, and the peeking operators are provided for accessing them. These operators depend explicitly on the specific tabular representation chosen for deltas, and are thus model dependent.

We now mention several simple algebraic properties implied by the requirements given previously for the operators when, smash, and compose. It is easy to show the following relationship between compose on deltas and compose ";" on commands.

Property 1 *(Composition)* apply(δ_1#δ_2) ≡ apply(δ_1) ; apply(δ_2). As indicated in Section 3.3, this property does not hold for smash; that is, apply(δ_1!δ_2) ≡ apply(δ_1) ; apply(δ_2) does not necessarily hold.

Property 2 *(Zero and One Laws)*

$$\begin{aligned}&\texttt{emtpy}\ !\ \delta \equiv \delta\ !\ \texttt{empty} \equiv \delta\\&\texttt{empty}\ \#\ \delta \equiv \delta\ \#\ \texttt{empty} \equiv \delta\\&\texttt{fail}\ !\ \delta \equiv \delta\ !\ \texttt{fail} \equiv \mathit{fail}\\&\texttt{fail}\ \#\ \delta \equiv \delta\ \#\ \texttt{fail} \equiv \mathit{fail}.\end{aligned}$$

Property 3 *(Associativity)*

$$\begin{aligned}&\delta_1\ !(\delta_2 ! \delta_3) \equiv (\delta_1 ! \delta_2) ! \delta_3\\&\delta_1 \#(\delta_2 \# \delta_3) \equiv (\delta_1 \# \delta_2) \# \delta_3.\end{aligned}$$

Property 4 *(Nested When)* (E when δ_1) when δ_2 ≡ E when (δ_2#δ_1). The following apply only if evaluation of the delta expression to the right of the when has no side effects.

Property 5 *(When Distributivity (Assuming Evaluation of* δ *Has no Side Effects)).*

$$\begin{aligned}&(\delta_1 ! \delta_2)\ \texttt{when}\ \delta \equiv (\delta_1\ \texttt{when}\ \delta)!(\delta_2\ \texttt{when}\ \delta)\\&f(E_1, \ldots, E_n)\ \texttt{when}\ \delta \equiv f(E_1\ \texttt{when}\ \delta, \ldots, E_n\ \texttt{when}\ \delta),\end{aligned}$$

where f is a function of the host language. The Nested When and When Distributivity laws are of particular interest because they allow an implementation where whens are "pushed inwards" towards individual variables.

As noted before, Heraclitus[Alg, C] supports the merge operator, denoted by "&". This operator is model dependent, although one could give it a fairly general characterization by refining the semantics of merge presented in Section 3. We present here the intuition behind a model-independent version of this operator, and a collection of properties it is expected to

[7]In the following, $E_1 \equiv E_2$ means that $\mathcal{E}[\![E_1]\!] = \mathcal{E}[\![E_2]\!]$, and similarly for $C_1 \equiv C_2$.

satisfy. Conceptually, merge has a declarative flavor in that it attempts to simultaneously satisfy the update demands of both of its arguments. If *conflicting* updates are proposed, then the error delta value $\perp$ is generated. For example, evaluation of the expression `[x:=37] & [y:=38]` results in a delta proposing that the variable x be changed to 37 and the variable y be changed to 38. In contrast, evaluation of the expression `[x:=37] & [x:=38]` results in the error delta, because both of these updates cannot be performed simultaneously. Merge is expected to satisfy the following properties.

Property 6 *(Zero and One Laws)*

$$\texttt{emtpy} \;\&\; \delta \equiv \delta \;\&\; \texttt{empty} \equiv \delta$$
$$\texttt{fail} \;\&\; \delta \equiv \delta \;\&\; \texttt{fail} \equiv \textit{fail}.$$

Property 7 *(Idempotence)* $\delta \;\&\; \delta \equiv \delta$.

Property 8 *(Commutativity)* $\delta_1 \;\&\; \delta_2 \equiv \delta_2 \;\&\; \delta_1$.

Property 9 *(Associativity)* $\delta_1 \;\&\; (\delta_2 \;\&\; \delta_3) \equiv (\delta_1 \;\&\; \delta_2) \;\&\; \delta_3$.

Property 10 *(When Distributivity (Assuming Evaluation of* δ *Has no Side Effects))*

$$(\delta_1 \;\&\; \delta_2) \texttt{ when } \delta \equiv (\delta_1 \texttt{ when } \delta) \;\&\; (\delta_2 \texttt{ when } \delta).$$

None of these properties captures the intuition that the merge of two deltas should correspond to a "union" of the proposed updates of these two deltas; capturing this intuition would require a property that is model-dependent.

6. ACTIVE DATABASE EXECUTION MODELS

Active database technology offers a relatively declarative style of specifying database behavior. Triggers are the most primitive form of activeness; these were enriched in Morgenstern [1983] by considering rules with more complex conditions, possibly referring to more than one (virtual) state of the database. Since then, a host of active database systems have been proposed in the literature, along with considerable research demonstrating the promise of activeness in connection with several application areas, including constraint maintenance, heterogeneous database interoperation, and incremental update for materialized views. The book by Widom and Ceri [1995] presents detailed descriptions of several active database systems and some of their applications, and the articles by Hanson and Widom [1992] and Stonebraker [1992] provide useful surveys of the field.

A major difficulty facing the field of active databases today concerns the choice of *execution model*, or semantics, for rule application. Each of the active database systems proposed in the literature has a different execution model, and some [Cacace et al. 1990; Dalrymple 1995] permit the use of multiple execution models within the same application. This variety of alternatives highlights the fact that the "knowledge" represented in an active database stems from two distinct components: the rule base and the execution model [Abiteboul 1988]. The execution models in the literature are generally described in an informal fashion, and so it is difficult to compare them, or to fully understand their semantics. Furthermore, it appears that a fixed collection of choices is unlikely to suffice for the myriad present and future applications.

In this section we show how the Heraclitus paradigm can be used to help provide a unified perspective on active database execution models. The section begins with a brief review of execution models, including the important notions of *deferred* versus *immediate* firing of rules. (This discussion is included for completeness; readers familiar with the prominent active database models may choose to skip it.) In the subsequent sections we illustrate how Heraclitus[Alg, C] can be used to specify a wide variety of execution models based on deferred rule firing, immediate rule firing, and hybrids of these. The section closes by briefly mentioning ways that Heraclitus[Alg, C] has been used to generalize previous research on activeness.

Because Heraclitus[Alg, C] is an implemented DBPL, specifications of execution models written in Heraclitus[Alg, C] can be executed directly, and we have done this for several execution models. As indicated in Section 4, in its current form Heraclitus[Alg, C] does not provide extensive optimization, and so the current Heraclitus implementation is not especially realistic for practical application. Nevertheless, the Heraclitus implementation is useful in the context of rapid prototyping of, and experimentation with, execution models. Once an execution model is finalized, a more efficient implementation can be developed from scratch, or by developing appropriate optimizations for Heraclitus. If the community of practitioners finds the Heraclitus paradigm to be useful for constructing customized execution models, we expect that more optimized implementations of the Heraclitus operators will become available.

Also, although Heraclitus[Alg, C] can specify a wide variety of execution models, the language constructs provided are admittedly somewhat low-level for this purpose. As we gain experience with Heraclitus and alternative execution models we shall develop higher-level language constructs to facilitate the specification of execution models.

6.1 An Overview of Execution Models

The space of most execution models described in the literature can be characterized in terms of two dimensions: the coupling modes of the "ECA" model, and the use of "multi-state logic."

Briefly, the ECA or "event-condition-action" model introduced by the HiPAC project [Dayal et al. 1988; Hsu et al. 1988; McCarthy and Dayal 1989; Widom and Ceri 1995] permits an active database rule to have the form

$$\texttt{on}\ \langle event \rangle\ \texttt{if}\ \langle condition \rangle\ \texttt{then}\ \langle action \rangle,$$

where *event* and *condition* are Boolean-valued expressions and *action* is a database command (possibly consisting of several atomic commands). For example, a rule might express that

> on any deletion from *Suppliers*
> if there are orders in *Order* from any deleted suppliers
> then delete those orders from *Order*.

Typically the *event* refers to either an event external to the database, or a change to the database state. In some models (e.g., POSTGRES) internal database *events* are rather elementary and can be monitored deep within the physical implementation of the database, whereas in other models (e.g., Gehani and Jagadish [1991]; Gatziv and Dittrich [1994]) quite complex "composite" events can be specified. The *condition* is typically specified using the query language of the underlying DBMS, for example, SQL. The *action* might be an explicit update to the database, as suggested here, or might involve calling a procedure written in the host language.

An important contribution of HiPAC was the identification of various *coupling modes* that specify, for a given rule for which the *event* has become true, the relative timing of when the *condition* is tested, and if that is true, when the *action* is executed. Briefly, HiPAC assumes that the underlying DBMS supports concurrent nested transactions. Three coupling modes are identified by HiPAC, these being *immediate*, *deferred*, and *separate* (or *decoupled*). The coupling modes can be used in connection with the *event-condition* pair and with the *condition-action* pair.[8] Immediate coupling means that the second activity (be it testing the condition or executing the action) follows the first "immediately," that is, before any other atomic command of the transaction triggering the event is executed. Deferred coupling means that the second activity is postponed until some later time, but still falls within the same transaction. Separate coupling means that a concurrent process is spawned to perform the second activity. (See Widom and Ceri [1995] for more details.)

In HiPAC, if two rule conditions are to be tested at the same time, then concurrent nested transactions are spawned for them. The same is true for rule actions. This means that the execution model need not perform conflict resolution between rules: all applicable rules are fired "simultaneously." On the other hand, because the concurrency manager ensures serializability, the rule conditions and actions are in effect executed in a serial and nondeterministic order. Furthermore, in many cases the condition testing and firing of rules will be independent of each other because they are considered in separate subtransactions. Thus the condition of one rule will not be able to inspect the impact of other concurrently fired rules. An extension of the HiPAC approach is presented in Beeri and Milo [1991]. This provides language constructs so that the user can specify more

[8]A total of seven combinations are considered, because deferred-immediate and deferred-deferred are considered to be equivalent, and separate-immediate and separate-deferred are considered to be equivalent. (See Hsu et al. [1988].)

explicitly the timing of action executions. (Beeri and Milo [1991] do not support conditions.)

The ECA model and the coupling modes of immediate and deferred are relevant even in the context of serial execution without nested transactions. For example, the version of POSTGRES described in Stonebraker et al. [1990] focuses on immediate coupling for both *event-condition* and *condition-action* pairs, and does not use nested transactions. In contrast, AP5, the Starburst Rule System, and ARIEL support serial execution models that use deferred testing of conditions with immediate execution of rule actions. POSTGRES, Starburst, and ARIEL use priority schemes to select the order of rule firing when two or more *event*s become true simultaneously. The execution model of AP5 is based on cycles of rule firing; with each cycle all eligible rules are fired "in parallel."

The second dimension of the space of execution models concerns the explicit use of "multistate logic." A central element in active database execution models is the fact that multiple (virtual) database states are created during the application of a rulebase. Indeed, this forms one of the key differences between active database technology and expert systems technology: expert systems typically react only to the current state of the underlying knowledge base, whereas many active database systems react both to the "current" state, and also to how this state came about (see Section 6.2). An early system in which conditions explicitly refer to two virtual states is AP5; more recent systems include the Starburst Rule System and A-RDL [Simon and Kiernan 1995] (a descendant of RDL1 [deMaindreville and Simon 1988]). The execution models of each of these systems uses a deferred coupling mode between event and condition, but each supports a different convention concerning what virtual states are accessible by rule events and conditions, and the syntax used to refer to them.

As noted previously, HiPAC does not support access to multiple virtual states in rule conditions. POSTGRES and ARIEL do permit access to the original and new versions of individual tuples that trigger rules, but do not support more general access to the original and new virtual states created during the course of rule execution.

Almost all the execution models described in the literature can be specified in terms of a particular combination of coupling modes and access (by events and by conditions) to virtual states created during the process of rule execution. In the remainder of this section we describe how Heraclitus[Alg, C] can be used to precisely specify several of the execution models found in the literature, and many variations of them. At this time Heraclitus[Alg, C] does not support nested transactions, and so it cannot be used to specify those aspects of the execution models of HiPAC or Beeri and Milo [1991] that rely on nested transactions. (For example, Heraclitus[Alg, C] can support the spawning of a rule action to be a separate and independent transaction. However, it cannot support the spawning of a rule action to be a separate but dependent transaction.)

6.2 Transaction Boundary Rule Firing

The primary contribution of the Heraclitus paradigm in specifying execution models concerns the representation and accessing of multiple virtual states. To illustrate this, we focus in this section on the somewhat simplified situation where the *event* of each rule is simply "true," the coupling between event and condition is deferred, and the coupling between condition and action is immediate. (This implies that rules can respond only to changes to the database state.) We refer to such execution models as having *transaction boundary rule firing*, because rule firing occurs only after the execution of all atomic commands in a user-requested update. We return to the more general ECA model in the following section.

Under transaction boundary rule firing, rule application constructs a sequence of "virtual states"

$$S_{orig}, S_{prop}, S_2, S_3, \ldots, S_{curr}$$

of the database, where S_{orig} is the "original" state and S_{prop} is the result of applying to S_{orig} the set of user-proposed updates collected during the transaction. The subsequent virtual states result from a sequence of rule firings according to the execution model. S_{curr} denotes the "current" virtual state that is being considered by the execution model. Execution either aborts when an erroneous situation arises, or terminates when the execution model reaches a fixpoint (i.e., $S_{curr} = S_{curr-1}$), in which case the final virtual state replaces S_{orig}. Prominent systems[9] following essentially this paradigm include AP5, A-RDL, LOGRES, the Starburst Rule System, and ARIEL, and also expert systems such as OPS5.

To illustrate, recall the inventory control example of Section 3. Consider the referential integrity constraint stating that if there is an order for part p from supplier s, then the pair (s, p) should appear in relation `Suppliers`. A possible rule for enforcing this might be written as

```
R1: if Orders(part, qty, sup, exp) and not Suppliers(sup, part)
      then -Orders(part, qty, sup, exp).
```

(We use a pseudocode here, and describe in the following how such rules are specified in Heraclitus[Alg, C].) With rules such as this it is implicit which virtual state(s) are being considered by the conditions and actions. In typical active database systems, if at some point in the application of rules the state S_{curr} satisfies the condition of R1 for some assignment of variables, then the action may be fired, depending on the presence of other rules whose conditions are true. We say that Rule R1 uses a "one-state" logic, because the rule condition examines a single state, namely the

[9]Some of these systems explicitly support nontrivial *events*.

"current" one. RDL1 [de Maindreville and Simon 1988], LOGRES, and most expert systems (e.g., OPS5) support only a one-state logic.

In the context of databases, a problem with Rule R1 is that the appropriate response to a constraint violation may depend on how the violation arose. Rule R2 deletes all violating orders if a pair is deleted from the `Suppliers` relation, but if the violation is the result of an update to `Orders`, then R3 undoes that individual update and transmits a warning.

```
R2: if -Suppliers(sup, part)
    then -Orders(part, *, sup, *).
R3: if +Orders(part, qty, sup, exp) and not Suppliers(sup, part)
    then -Orders(part, qty, sup, exp) and send_warning(part, qty,
    sup, exp).
```

The signed atoms in the conditions of these rules refer to proposed updates rather than any database state. (The use of wildcards ("*"s) in the action of R2 is analogous to their use in Heraclitus[Alg, C].)

In essence, the conditions of Rules R2 and R3 make explicit reference to the delta between two virtual states. Of course, some design choice needs to be made about which pair of virtual states should be considered. The AP5 system focuses on the delta between S_{orig} and S_{curr}:

$$\underbrace{S_{orig}, S_{prop}, S_2, S_3, \cdots, S_{curr}}_{\Delta}.$$

Assuming this semantics for a moment, note that a one-state execution model cannot simulate the effect of Rules R2 and R3 without using "scratch paper relations" that essentially duplicate the contents of S_{orig}. Another natural semantics for rule conditions supporting explicit access to a delta would be to use the delta between S_{prop} and S_{curr}. The Starburst Rule System is more intricate: it uses the delta between virtual states S_i and S_{curr}, where i is determined by the rule under consideration and the history of previous firings of that rule (see Widom and Finkelstein [1990]; Hull and Jacobs [1991]).

Under one approach to specifying execution models using Heraclitus, the original state S_{orig} remains unchanged during rule firing, and deltas are used to represent the sequence of virtual states resulting from the user update and rule firing. In this context, rules are represented as functions that have as input zero or more deltas, and produce as output a delta corresponding to the effect of the rule firing. For example, Rule R2 can be expressed in Heraclitus[Alg, C] (assuming the type declarations given in Section 3) as follows:[10]

[10]Technically, the type of input variable curr should be declared in a separate line of the type declaration of `rule_R2`, given the version of C that was extended to form Heraclitus[Alg, C]. To simplify the exposition here, we use the ANSI-C style of declaring the types of input parameters.

```
delta rule_R2(delta curr)
{
    return bulk(del Ord(part, *, sup, *), peekdel(Supp, curr));
}
```

Consider finally the rule

R4: if the firing of rules results in a 20% drop in orders
 then `inventory_warning` .

Here we need to consider the change in orders between S_{prop} and S_{curr}:

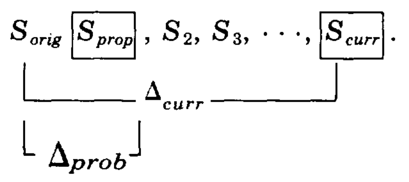
$$S_{orig}, \boxed{S_{prop}}, S_2, S_3, \cdots, \boxed{S_{curr}}.$$

Although this could be expressed using explicit access to a delta, it is much easier to express it in terms of the virtual states, that is, to write:

$$\text{R4}': \text{if } \frac{\texttt{count(Orders)} \text{ “in } S_{curr}\text{”}}{\texttt{count(Orders)} \text{ “in } S_{prop}\text{”}} < .8$$

then `inventory_warning()`.

In current DBPLs there is no mechanism to write expressions such as the condition of R4′, because they do not provide explicit access to virtual states. The Heraclitus paradigm provides this by using deltas and the when operator. One way to express Rule R4′ in the Heraclitus paradigm is to construct deltas corresponding to the virtual states S_{prop} and S_{curr} as follows:



Rule R4′ can be expressed in Heraclitus as:

$$\text{R4}'': \text{if } \frac{\texttt{count(Orders) when } \Delta_{\text{curr}}}{\texttt{count(Orders) when } \Delta_{\text{prop}}} < .8$$

then `inventory-warning()`.

Using the Heraclitus[Alg, C] syntax, this rule might be written as follows:

```
delta rule_R4(delta prop, curr)
{
        if ( (float)(count(Ord) when curr) / (float)(count(Ord)
           when prop) < .8) inventory_warning ( );
     return *empty_delta;
}
```

This rule is more difficult to express in active database systems such as POSTGRES and ARIEL, because they support explicit access only to the previous and current versions of individual tuples, rather than to different versions of the full virtual state.

We now describe how the Heraclitus paradigm can specify a large family of execution models that use transaction boundary rule firing. We continue with the simplifying assumption that the *event* of each rule is simply "`true`". (We briefly consider the incorporation of nontrivial events into this framework near the end of this section.) Hull and Jacobs [1991] show in much greater detail how Heraclitus can simulate the kernels of two prominent active database systems that use transaction boundary rule firing, namely, the Starburst Rule System and AP5.

As suggested before, in the approach described here the original database state S_{orig} remains untouched until termination, and deltas are constructed to represent the virtual states corresponding to rule firing. (An alternative would be to update the database state with each rule firing, and maintain "negative" deltas that simulate previous virtual states in the sequence.) Rules are represented as functions that have as input zero or more deltas (corresponding either to virtual states or deltas between them), and produce as output a delta corresponding to the effect of the rule firing. The rules might also invoke additional procedures such as `inventory_warning( )`. Rules can be arranged to provide either "tuple-at-a-time" or "set-at-a-time" operation [Widom and Finkelstein 1990]. The Heraclitus operators merge and smash are used so that deltas corresponding to new virtual states can be constructed from previous deltas and rule outputs. Using this approach, the execution models of AP5, RDL1, LOGRES, the Starburst Rule System, and ARIEL can be specified within Heraclitus[Alg, C].

To provide a simple illustration of how Heraclitus[Alg, C] can be used to specify execution models with transaction boundary rule firing, assume that a total of 25 rules are written to capture the purchasing policy for the inventory control application, all using input variables corresponding to S_{prop} and S_{curr} (or more precisely, using input variables of type delta that correspond to Δ_{prop} and Δ_{curr}). In Heraclitus[Alg, C] these can be combined into an array of delta functions as follows:

```
delta (*policy[24])();
policy[0] = rule_R0;
policy[1] = rule_R1;
  :
policy[24] = rule_R24;
```

The following function specifies an execution model that takes in a delta corresponding to a user-requested update and applies the rules according to a specific algorithm. Here, the assignment `temp = *empty_delta` initializes `temp` as a transient delta holding the empty delta.

```
boolean apply_policy(delta prop)
{
    delta curr, prev, temp;
    if (prop == fail) return (false);
    curr = prop;
    do {
         prev = curr;
         temp = *empty-delta;
         for (i=0; i<25; i++) temp = temp &
         (*policy[i])(prop, curr);
         curr = curr ! temp;
    } while (curr != fail && prev != curr);
    if (curr == fail)
     return (false);
     else {
       apply curr;
       return (true); }
}
```

Here the inner `for` loop corresponds to a single, simultaneous, and independent (set-oriented) application of each rule in `policy`, and combines the results using merge. This is "simultaneous" application of the rules, because each rule is evaluated on `prop` and `curr`; the resulting deltas are accumulated in variable `temp`. The outer `while` loop repeatedly performs the inner loop, using smash to fold the results of each iteration into the value of `curr` already obtained. The outer loop is performed until either a fixpoint is reached, or the inner loop produces the delta *fail* (either because one of the rules explicitly called for an abort by producing *fail*, or because in some execution of the inner loop, two rules produced conflicting deltas).

In some active databases such as the Starburst Rule System, the rule condition creates a relation which is passed to the rule action. This might be called the "witness" relation, because the tuples created by the condition serve as witnesses that the rule action should be fired. As illustrated in Section 6.3, this is easily accommodated in Heraclitus[Alg, C]. In particular, relation variables in Heraclitus[Alg, C] can hold relations of different signatures over the course of the program. Thus the Heraclitus[Alg, C] code for an execution model can use the same variable for the witness relations of all the rules.

Suppose now that there is a second array `keys` that includes 15 rule functions that capture key constraints, and that the preceding execution model is to be modified so that after each execution of the inner loop, the rules in `keys` are to be fired until a fixpoint is reached. Suppose further that these rules use only a single input delta, corresponding to S_{curr}. Now let function `apply_rules` have the following signature

```
delta apply_rules(delta curr, delta *rule_base[](), int size),
```

and suppose that it applies the rules in rule_base until a fixpoint is reached. Then the desired modification to apply_policy can be accomplished by adding

```
    curr = curr ! apply_rules(curr, keys, 15);
```

as the last line of the inner for loop.

These examples provide a very brief indication of the kind of flexibility that Heraclitus[Alg, C] provides in specifying active database execution models with transaction boundary rule firing. Variations on this theme can be developed. As a simple example, a rule-base can be "stratified," and the execution model can fire each layer to a fixpoint before moving to the next layer. More complex firing patterns subsuming the rule algebra of Imielinski and Naqvi [1988] are expressed easily (see Section 6.4).

We now briefly consider how nontrivial events might be incorporated into the framework just presented. As noted earlier, both events and conditions are Boolean expressions. Typically, events are specified using a language with considerably less expressive power than the language for conditions. As a practical matter, in several prototype active database systems the mechanism for testing events is deep within the physical implementation so that it is inexpensive to identify the set of rules whose events are true. We are concerned here primarily with capturing the logical semantics of an execution model, and do not consider such optimizations here. It is typical that the selection of a rule to be fired occurs in two phases, first identifying the set of rules whose event is true, and from that set identifying one or more rules whose condition is true. In some execution models, the semantics of how an event or condition is tested or a rule action interpreted may be affected by when the rule event has been true in the past. For example, in the Starburst Rule System events refer to a delta that reflects some of the previous rule actions (and possibly the user transaction), and rule conditions can refer to the "current" state and also the delta just mentioned. To capture this kind of semantics in Heraclitus[Alg, C], separate functions can be created that correspond to testing rule events, testing rule conditions, and constructing deltas corresponding to rule actions. In the execution model written in Heraclitus[Alg, C] it is easy to store relevant information about when the events and/or conditions of rules became true, and thus provide the appropriate deltas to the events, conditions, and actions. This is illustrated in Section 6.3 in the context of immediate-immediate coupling (see also Hull and Jacobs [1991]).

6.3 Specifying Immediate-Immediate Coupling

In this section we return to the case where nontrivial *events* are permitted, and focus on the case of immediate-immediate coupling. This follows the spirit of the execution model of POSTGRES as described in Stonebraker et al. [1990]. From this and the discussion of Section 6.2 it is clear how families of rules with diverse coupling modes can be supported in Heraclitus.

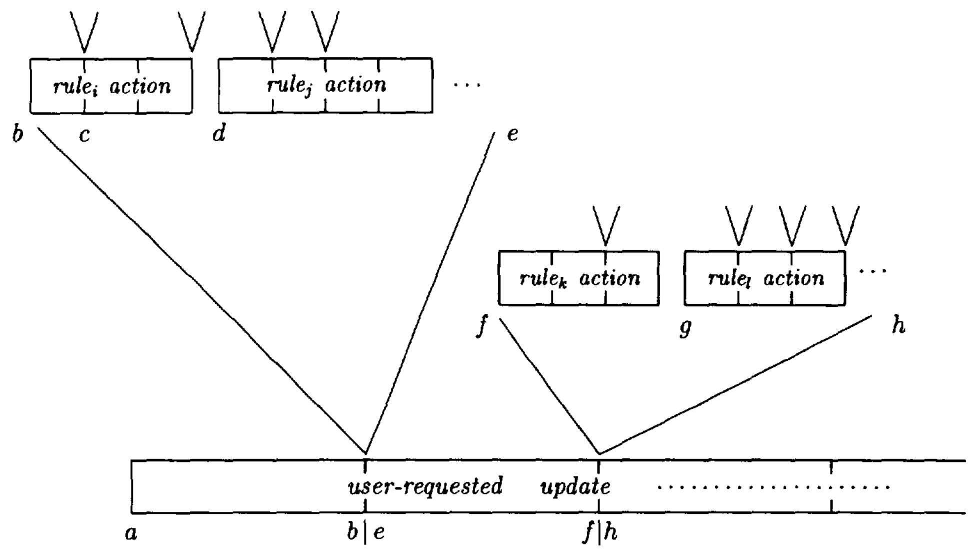


Fig. 8. Schematic illustration of immediate rule firing.

In the context of immediate-immediate coupling, a user-requested update will generally be a sequence of several atomic update commands. In the general case, some rules might fire before all these atomic commands are executed. In the sequence of virtual states created by rule execution, there may not be a state corresponding to S_{prop} as in Section 6.2. Thus we now consider a sequence of virtual states

$$S_{orig}, S_1, S_2, S_3, \ldots, S_{curr},$$

where each state is the result of applying a single atomic update command to the previous one.

As in the case of transaction boundary rule firing, an important design issue concerns the virtual states that are accessible to rule events and conditions. We consider now, rather microscopically, the execution of a few steps of a user-requested transaction along with some immediate rule firings.

Figure 8 presents a schematic representation of the first part of an execution[11] of a user-requested update, along with the effects of some rules that are fired along the way. The lowest long rectangle represents the sequence of atomic commands that make up the user-requested update; each shorter rectangle within the lowest one represents a single atomic update command within that update request. The two long rectangles in the upper left represent the actions of two rules (let us call them rule i and rule j) that fire after the first atomic update. The other long rectangles represent the actions of rules k and l, that fire after the second atomic

[11]We mean "execution" in a virtual sense; these updates will not be committed until rule firing has successfully completed without aborting.

update of the user request has been executed. The figure also suggests that rules are fired as the result of some of the atomic updates of rules *i*, *j*, *k*, and *l*. This is possible because immediate firing of rules may be recursive.

The letters *a* through *f* depict different points in time during the execution; for example, *b* indicates the point in time after the first atomic update of the user request has been executed and before any other atomic update has been executed, and *e* indicates the point in time just before the second atomic update of the user request is executed. In the discussion that follows, letting *x* range over $\{a, \ldots, h\}$, we use S_x to denote the virtual state that corresponds to time *x*. In particular, S_a is a synonym for S_{orig}. Also, given two (virtual) states S and S', we let *diff*(S, S') denote the (minimal) delta value that maps S into S'.

As noted previously, the *event* of a rule is generally focused on an event external to the database, or on the difference *diff*(S, S') between two virtual states arising during execution of a user-requested update and rule firing. Likewise, in most active database systems supporting immediate-immediate rule firing, the *condition* makes reference to two states, the "current" one and some previous one. (In some cases this is accomplished by permitting explicit access only to the values of changed tuples or objects, rather than permitting explicit access to an entire virtual state.)

Consider now rule *i*. Presumably its *event* is true in *diff*(S_a, S_b), and its *condition* is true relative to S_a and S_b. What about rule *j*? Is rule *j* to be considered because its *event* is true in *diff*(S_a, S_b), or because it is true in *diff*(S_a, S_d)? Likewise for the condition of rule *j*: need it be true relative to S_a and S_b, or S_a and S_d? Continuing with the example, design choices must also be made in connection with time *f*: what states should be considered when determining what rules should be fired between time *f* and *h*?

As with transaction-boundary execution models, the Heraclitus paradigm can be used to specify a wide range of alternatives in connection with immediate-immediate rule firing, and permit explicit specification of what virtual states are to be accessed by rule events and conditions. To illustrate, we present in pseudocode a very simple execution model that supports recursive immediate-immediate rule firing.

We assume that the rulebase is represented using three arrays of size `rulebase_size`, one each for the events, conditions, and actions of rules. Events and conditions will take as input two deltas corresponding to virtual states, and return Booleans. Conditions will also produce a "witness" relation, that is passed to the rule action. The witness relation might be used, for example, to hold the set of tuples that violate a constraint that the rule is supposed to maintain (cf. Ceri and Widom [1991]). In the case where the events of two rules become true simultaneously, priority is given to the rule having the lower number in the rulebase.

In the code we assume that there is a type `atomic_command` that holds atomic database commands as C structures. The user-requested update and rule actions are represented as arrays of such atomic commands. These arrays have fixed size max_size, and array entries not containing atomic commands hold a special null marker.

As before, we assume that the stored database state does not change until the processing is completed, and that global variable curr holds the delta corresponding to the "current" virtual state that the execution has reached. The main routine is:

```
void invoke_rulebase(atomic_command *user_request[max_size])
{
     curr = *empty_delta;
     apply_rules(user_request);
     if curr != fail apply curr;
}
```

The subroutine for applying the rulebase is given by the following pseudocode. Here net_effect(curr, c) is the delta value corresponding to the effect that atomic command c would have if applied to the result of applying curr to the underlying database state; and net-diff returns the delta corresponding to the net difference between the states represented by two deltas. (More precisely, let S be a database state and let the value of δ_i in S be Δ_i for i = 1, 2. Then the expression net_diff(δ_1, δ_2) evaluated in S yields the minimal delta Δ such that $apply(apply(S, \Delta_1),\Delta) = apply(S, \Delta_2)$, that is, the minimal delta that takes state $apply(S, \Delta_1)$ to state $apply(S, \Delta_2)$.)

```
void apply_rules(atomic_command *command_array[max_size])
{
     delta local_start, local_event_end;
     relation *witness;
     for (i=0; more commands in command_array; i++)
     {
          local_start = curr;
          curr = curr ! net_effect(curr, command_array[i]);
          local_event_end = curr;
          j = 0;
          while (j < rulebase_size)
          {
               if (event[j](net_diff(local_start, local_event_
                         end)) && cond[j](local_start, curr,
                         &witness))
                    {
                         apply_rules(action[j](witness));
                         j = 0;
                    }
                    else j++;
          }
     }
}    |
```

In this execution model, referring to Figure 8, the event of rule j is tested in connection with S_a and S_b, and the condition of rule j is tested in connection with S_a and S_d. In this example the event and condition are satisfied, and so relevant data is passed to the rule action via the relation variable witness.

It is clear that a variety of other design choices in connection with what events and conditions access can be specified using the Heraclitus paradigm. Furthermore, the conditions could be designed to permit access to more than two virtual states.

6.4 Novel Execution Models

A fundamental contribution of the Heraclitus paradigm and Heraclitus[Alg, C] is to provide a very flexible and relatively straightforward mechanism for specifying and experimenting with new execution models. In this section we briefly consider two novel kinds of execution models that have been developed using the Heraclitus paradigm, and that generalize previous research on active databases. This illustrates a "value-added" contribution of Heraclitus to the field of active databases: we have already seen that Heraclitus provides a unifying perspective on existing execution models; we show now that Heraclitus can be used to introduce fundamentally new kinds of execution models.

The first kind of execution model is to combine more explicitly the declarative style of rules with conventional imperative programming. We illustrate with a simple example, and then mention a much richer use of this approach.

To set the stage, recall that the semantics of a rulebase stem from both the rules themselves and the execution model. Speaking intuitively, this allows a partial separation of the specification of the *logic* of an application from the specification of the *flow-of-control*. In some cases it may be convenient to reduce this separation by supporting a two-tier approach that uses rulebases for performing subtasks but uses the imperative paradigm for sequencing and controlling the performance of these subtasks. An active database system that supports this perspective in a limited fashion is LOGRES, which provides mechanisms for specifying separate rule modules that can be enabled or disabled. In contrast with LOGRES, the Heraclitus paradigm affords great flexibility in how the imperative and declarative paradigms can be combined. The Coral [Ramakrishnan et al. 1992, 1993] and Glue-Nail [Derr et al. 1993] database programming languages also combine the imperative and declarative paradigms. The emphasis there is on efficient implementation of subprograms that are written in variants of Datalog. A fundamental feature of Heraclitus, namely, the ability to specify a wide variety of alternative semantics for rule application, is not supported.

As a simple example of specifying flow-of-control in an execution model, suppose that in a business, based on certain conditions, rules will be fired in order to remedy problems, for example, if sales volume is too low then increase advertising, or lower prices by 5%, or lay off 10% of the employees. 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.[12] The following procedure assumes that there are c rule clusters, ordered by increasing cost, and attempts to find the cheapest solution for the current state of the database. Here the function apply_rules(j, d) returns a delta corresponding to the application of the j^{th} cluster of rules in the context of delta d.

```
void apply_cheapest_solution(int c, delta prop)
{
     delta attempt;
     /*. . .declarations for functions included here. . .*/
     for (i = 0; i < c; i++)
     {
          attempt = apply_rules(i, prop);
          if satisfies_constraints(attempt)
          {
             apply attempt;
             return
          }
     }
     print_message("no cluster offers a solution")
}
```

It is clear that more complex firing patterns can be developed.

A much more comprehensive investigation of combining the imperative and active paradigms is described in Dalrymple [1995]. That work explores three different interoperability problems, focused on interoperating databases, interoperating software systems, and message-based interoperation between diverse computer systems. In each case, the overall task is broken into submodules following standard software engineering principles. For example, in the database application there are different submodules that focus on incremental update propagation between databases holding overlapping information, remote object retrieval (for views maintained in a virtual manner), security and access rights, and responding to external events. Several different execution models were found useful in Dalrymple [1995] for supporting different submodules arising in the three example applications. A working prototype was developed using Heraclitus[Alg, C] in support of the application for interoperating software systems.

Related investigations are found in Zhou et al. [1995] and Hull and Zhou [1995], which concern the construction of mediators that maintain materialized integrated views of multiple heterogeneous databases. A customized execution model has been developed using Heraclitus[Alg, C] to support the incremental update propagation needed for these mediators.

A second kind of execution model generalizing previous active database research is described in Chen et al. [1996]. This kind of execution model is based on the use of "limited ambiguity rules" (LARs), that may have disjunctions in their actions. The LAR execution model explores a directed

[12]The authors thank Serge Abiteboul for suggesting this example.

acyclic graph (DAG) of possibilities, where a fork in the DAG corresponds to the firing of a rule whose action involves a disjunction. In Chen et al. [1996] LARs and their associated execution model are applied to updating derived data in semantic models (see, e.g., Hull and King [1987] for a description of semantic data models). As a simple example, suppose that the class `Employed_Student` is defined to be the intersection of classes `Employee` and `Student`, that x is in the class `Employed_Student`, and that an explicit request to delete x from `Employed_Student` has been made. This implies that either x should be deleted from `Employee`, or from `Student`, or from both these classes. Under the LAR approach, this is captured by the rule (expressed here in pseudocode):

```
IF z is deleted from Employed_Student
THEN [delete z from Employee]
     OR [delete z from Student]
     OR [delete z from Employee and delete z from Student]
```

The execution model for LARs creates a DAG of virtual states (represented as deltas), with branching corresponding to rule actions that have more than one disjunct. Branch merging can occur if rules applied to two different states yield the same state. Execution stops when no rules can be fired. Successful leaves of the tree correspond to valid "completions" of a user-requested update; if there are two or more successful leaves then a separate policy can be used to choose between them. Chen et al. [1996] describe the execution model in full, and provide a mechanism for compiling semantic data model schemas into LAR rulebases.

The execution model of the LAR approach has been implemented both directly in C [Chen et al. 1996], and using Heraclitus[Alg, C]. There is considerable nondeterminism in the basic LAR execution model. Because the Heraclitus code was at a higher level than C, it was easy to write and compare different optimizations in the Heraclitus version. However, for a given strategy the C version was more efficient.

We expect that the LAR approach can be used in a variety of application areas, where the repair of a violated constraint is not easily expressed using a single (conventional) rule. For example, in some minimization problems more than one local minimum may exist, but other constraints in the application might disqualify some of them. The LAR approach might also be useful in the context of backward chaining, in the sense of Heineman et al. [1992], where there may be more than one way to cause a given rule to fire. We expect that the LAR approach will be most useful in the previously described hybrid framework of Dalrymple [1995], where some subtasks are solved using the LAR approach, and others are solved using more conventional execution models.

7. CONCLUSION AND FUTURE RESEARCH

The Heraclitus paradigm is a language extension for DBPLs that elevates deltas to be first-class citizens. The paradigm provides constructs to create, combine, and access deltas. Most important is the `when` operator, which permits hypothetical access to deltas, enabling a programmer to consider "what-if" scenarios and to reason about their impact. The Heraclitus paradigm can be used to support a variety of database applications.

This article describes a concrete realization of the Heraclitus paradigm, in the form of the prototype language Heraclitus[Alg, C], an implemented DBPL that extends C with relations and deltas for them. The article describes (1) the syntax and semantics of Heraclitus[Alg, C], (2) the subtleties that arise when manipulating deltas, including algebraic properties of the when operator, (3) two alternative implementations for operators that access deltas, based on hashing and sort-merge techniques, (4) the tradeoffs associated with these two alternatives, and (5) how Heraclitus[Alg, C] can and has been used to specify a wide variety of execution models for active databases. The article also (6) develops a formal and model-independent framework for the core of the Heraclitus paradigm that can be used when developing other realizations of it.

We are extending Heraclitus in several ways. From an implementation perspective, we are currently investigating two enhancements. The first is to further optimize both the compiler and the storage manager, primarily through the use of B^+-tree index structures. Both relations and deltas may have B^+-trees, so that the hypothetical relational operators can be more efficient. The second enhancement is the development of high-level macros to simplify the specification of execution models of active databases. As detailed in Section 6, we have been using Heraclitus[Alg, C] to develop and experiment with novel active database execution models; this work is continuing with application to database interoperation [Zhou et al. 1995].

The model-independent perspective of Heraclitus as described in Section 5 is a starting point to extend this paradigm to models other than the pure relational model. For example, the Heraclitus[OO] (abbreviated H2O) project [Boucelma et al. 1995] is a broad project that includes the development of a Heraclitus-based DBPL for object-oriented databases (see Boucelma et al. [1995] and Doherty et al. [1995a]). Work is also in progress [Doherty et al. 1995b] on developing a coherent understanding of how the Heraclitus paradigm can be generalized to a variety of common database data structuring constructs (including tuples, sets, bags, lists, and sets with keys or object identifiers) and for complex data types built from these. The work on bags is particularly relevant to incorporating the Heraclitus paradigm into practical relational DBMSs, which typically view relations as bags of tuples, that is, with duplicates permitted.

An interesting application of the Heraclitus paradigm concerns the detection and resolution of conflicting updates. One application area involves providing complex services, such as telephone or transportation services, to large organizations. Given a desired upgrade in performance, multiple alternative hypothetical updates to the current service might be considered. Further, updates proposed to perform service upgrades concerning different but related aspects of the organization might conflict in some way (e.g., due to limitations of the underlying infrastructure). The Heraclitus paradigm provides a useful starting point for representing multiple hypothetical updates and detecting conflicts among them. It will be natural to use the rule-based paradigm of active databases to provide a relatively declarative basis for specifying how conflicts should be resolved. It is important to note that the proposed updates themselves can be naturally represented as deltas. Thus the rules of an update conflict resolution system will explicitly manipulate and analyze multiple deltas. This capability is not supported by any existing system. A preliminary investigation into this area is presented in Doherty and Hull [1995] and Doherty et al. [1995a].

Another preliminary investigation concerns the application of the Heraclitus paradigm to multimedia information systems [Ghandeharizadeh 1995; Escobar-Molano et al. 1995]. The basic idea is to (i) represent a complex scene as a collection of related atomic objects using spatial and temporal constructs, and (ii) use delta operators to define how the relationship between the objects evolves as a function of time. This will enable a system to reason about the changes to the objects as a function of time and space, and to construct logical concepts that might be queried, for example, the concept of one car chasing another.

ACKNOWLEDGMENTS

The Heraclitus[Alg, C] prototype was developed primarily at the University of Southern California, with some of the final effort occurring at the University of Colorado, Boulder. We are very thankful to four key members of the implementation team, namely, Jaime Castillo, Martha Escobar-Molano, Junhui Luo, and Gang Zhou, who participated in the research and greatly contributed to the management and implementation of the project. We are especially grateful to Gang Zhou, who almost single-handedly implemented the current version of Heraclitus[Alg, C], and who has been maintaining and extending that prototype in connection with the research described in Dalrymple [1995] and Zhou et al. [1995]. We would also like to thank Hsun-Ko Chan, Elton Chen, Vera Choi, Yuji Enzaki, Sanjay Joshi, Yoram Landau, Michel Liu, Shih-Hui Lu, Aris Prassinos, Babar Saeed, Chiu (Simon) Tsang, Howard Ungar, and Jackie Yu for their help with the implementation. We also thank Serge Abiteboul, I-Min A. Chen, Jim Dalrymple, Mike Doherty, Jean-Claude Franchitti, Roger King, and Eric Simon for useful discussions on the topics presented in this article. We also thank the anonymous referees of an earlier version of this article, who made many valuable suggestions regarding the presentation of this material.

REFERENCES

ABITEBOUL, S. 1988. Updates, a new frontier. In *Proceedings of the International Conference on Database Theory* (Bruges, Belgium, Aug.), 1–18.

ABITEBOUL, S., HULL, R., AND VIANU, V. 1994. *Foundations of Databases*. Addison-Wesley, Reading, MA.

ATKINSON, M. AND BUNEMAN, P. 1987. Types and persistence in database programming languages. *ACM Comput. Surv. 19*, 2 (June), 105–190.

BEERI, C. AND MILO, T. 1991. A model for active object oriented database. In *Proceedings of the International Conference on Very Large Data Bases* (Barcelona, Spain, Sept), 337–349.

BONNER, A. J. 1995. The logical semantics of hypothetical rulebases with deletion. *J. Logic Program.* (to appear).

BONNER, A. J. 1990. Hypothetical datalog: Complexity and expressibility. *Theor. Comput. Sci. 76*, 3–51.

BOUCELMA, O., DALRYMPLE, J., DOHERTY, M., FRANCHITTI, J. C., HULL, R., KING, R., AND ZHOU, G. 1995. Incorporating active and multi-database-state services into an OSA-compliant interoperability framework. In *The Collected Arcadia Papers, 2nd ed.*, University of California, Irvine, May.

BRATBERGSENGEN, K. 1984. Hashing methods and relational algebra operations. In *Proceedings of the Conference on Very Large Data Bases* (Singapore, Aug.), 323–333.

CACACE, F., CERI, S., CRESPI-REGHIZZI, S., TANCA, L., AND ZICARI, R. 1990. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proceedings of the ACM SIGMOD Symposium on the Management of Data* (Atlantic City, NJ, May), 225–236.

CAREY, M. J., DEWITT, D. J., RICHARDSON, J. E., AND SHEKITA, E. J. 1986. Object and file management in the EXODUS extensible database system. In *Proceedings of VLDB* (Kyoto, Japan, Aug.), 91–100.

CERI, S. AND WIDOM, J. 1991. Deriving production rules for incremental view maintenance. In *Proceedings of the International Conference on Very Large Data Bases* (Barcelona, Spain, Sept.), 577–589.

CHEN, I-M. A., HULL, R., AND MCLEOD, D. 1996. An execution model for limited ambiguity rules and its application to derived data update. *ACM Trans. Database Syst 20*, 4 (Dec.), 365–413.

CHEN, I-M. A., HULL, R., AND MCLEOD, D. 1994. An execution model for limited ambiguity rules and its application to derived data update. Tech. rep., Computer Science Dept., Univ. of Southern California, June. Also Derived data update via limited ambiguity. In *Fourth International Workshop on Research Issues in Data Engineering (RIDE-ADS '94)* (Houston, TX, Feb.), 77–86.

CHOU, H-T., DEWITT, D., KATZ, R., AND KLUG, T. 1985. Design and implementation of the Wisconsin Storage System (WiSS). *Softw. Pract. Exper. 15*, 10 (Oct.).

COHEN, D. 1989. Compiling complex database transition triggers. In *Proceedings of the ACM SIGMOD Symposium on the Management of Data* (Portland, OR, June), 225–234.

COHEN, D. 1986. Programming by specification and annotation. In *Proceedings of AAAI*.

COLLET, C., HABRAKEN, P., COUPAYE, T., AND ADIBA, M. 1994. Active rules for the software engineering platform GOODSTEP. In *Proceedings of the Second International Workshop on Database and Software Engineering* (Sorrento, Italy, May).

DALRYMPLE, J. 1995. Extending rule mechanisms for the construction of interoperable systems. University of Colorado, Boulder, Ph.D. thesis.

DAYAL, U. ET AL. 1988. The HIPAC project: Combining active database and timing constraints. *SIGMOD Rec. 17*, 1 (March), 51–70.

DE MAINDREVILLE, C. AND SIMON, E. 1988. Modelling non-deterministic queries and updates in deductive databases. In *Proceedings of the International Conference on Very Large Data Bases* (Los Angeles, CA, Aug.) 395–406.

DERR, M. A., MORISHITA, S., AND PHIPPS, G. 1993. Design and implementation of the Glue-Nail database system. In *Proceedings of the ACM SIGMOD Symposium on the Management of Data* (Washington D.C., May), 147–156.

DEWITT, D. AND GERBER, R. 1985. Multiprocessor hash-based join algorithms. In *Proceedings of the Conference on Very Large Data Bases* (Stockholm, Sweden, Aug.) 151–164.

DEWITT, D., GHANDEHARIZADEH, S., SCHNEIDER, D., BRICKER, A., HSIAO, H., AND RASMUSSEN, R. 1990. The gamma database machine project. *IEEE Trans. Knowl. Data Eng. 2*, 1 (March), 44–62.

DEWITT, D., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D. 1984. Implementation techniques for main memory database systems. In *Proceedings of SIGMOD* (Boston, MA, June, 1–8).

DOHERTY, M. AND HULL, R. 1995. Towards a framework for efficient management of potentially conflicting database updates. In *Proceedings of the IFIP WG2.6 Sixth Working Conference on Database Semantics (DS-6)* (Atlanta, GA, May).

DOHERTY, M., HULL, R., DERR, M., AND DURAND, J. 1995a. On detecting conflict between proposed updates. In *Proceedings of the International Workshop on Database Programming Languages* (Gubbio, Umbria, Sept.) (to appear).

DOHERTY, M., HULL, R., AND RUPAWALLA, M. 1995b. A framework for defining deltas on bulk data types. (in preparation).

ESCOBAR-MOLANO, M., GHANDEHARIZADEH, S., AND IERARDI, D. 1995. An optimal resource scheduler for continuous display of structured video objects. *IEEE Trans. Knowl. Data Eng.*

ESWARAN, K. P., GRAY, J., LORIE, R., AND TRAIGER, I. L. 1976. The notation of consistency and predicate locks in a database system. *Commun. ACM 19*, 11, 624–633.

GABBAY, D. M. 1985. N-Prolog: An extension of Prolog with hypothetical implications. II. Logical foundations and negation as failure. *J. Logic Program. 2*, 4, 251–283.

GATZIU, S. AND DITTRICH, K. R. 1994. Detecting composite events in active database systems using petri nets. In *Proceedings of Fourth International Workshop on Research Issues in Data Engineering: Active Database Systems* (Houston, TX, Feb.), 2–9.

GEHANI, N. H. AND JAGADISH, H. V. 1991. ODE as an active database: Constraints and triggers. In *Proceedings of the International Conference on Very Large Data Bases* (Barcelona, Spain, Sept.), 327–336.

GHANDEHARIZADEH, S. 1995. Stream-based versus structured video objects: Issues, solutions, and challenges. In *Multimedia Database Systems: Issues and Research Directions*. S. Jajodia and V. S. Subrahmanian, Eds., Springer Verlag.

GHANDEHARIZADEH, S., HULL, R., JACOBS, D., ET AL. 1993. On implementing a language for specifying active database execution models. In *Proceedings of the International Conference on Very Large Data Bases* (Dublin, Ireland, Sept.), 441–454.

GHANDEHARIZADEH, S., HULL, R., AND JACOBS, D. 1992. Implementation of delayed updates in Heraclitus. In *Proceedings of the International Conference on Extending Data Base Technology* (Vienna, Austria, March), 261–276.

GRAY, J. AND REUTER, A., EDS. 1993. *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, San Mateo, CA, Chapter 7, 403–406.

HANSON, E. N. 1989. An initial report on the design of Ariel: A DBMS with an integrated production rule system. *SIGMOD Rec. 18*, 3 (Sept.), 12–19.

HANSON, E. AND WIDOM, J. 1992. An overview of production rules in database systems. Tech. Rep. RJ 9023 (80483), IBM Almaden Research Center, October 12.

HEINEMAN, G. T., KAISER, G. E., BARGHOUTI, N. S., AND BEN-SHAUL, I. Z. 1992. Rule chaining in MARVEL: Dynamic binding of parameters. *IEEE Expert 7*, 6 (Dec.), 26–32.

HSU, M., LADIN, R., AND MCCARTHY, D. R. 1988. An execution model for active data base management systems. In *International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness*, 171–179.

HULL, R. AND JACOBS, D. 1991. Language constructs for programming active databases. In *Proceedings of the International Conference on Very Large Data Bases* (Barcelona, Spain, Sept.), 455–468.

HULL, R. AND KING, R. 1987. Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv. 19*, 3 (Sept.), 201–260.

HULL, R. AND ZHOU, G. 1995. A framework for optimizing data integration using the materialized and virtual approaches. Tech. rep., Computer Science Department, University of Colorado, May.

IMIELINSKI, T. AND NAQVI, S. 1988. Explicit control of logic programs through rule algebra. In *Proceedings of the ACM Symposium on Principles of Database Systems* (Austin, TX, March), 103–116.

JACOBS, D. AND HULL, R. 1991. Database programming with delayed updates. In *International Workshop on Database Programming Languages* (Nafplion, Greece, Aug.), Morgan-Kaufmann, San Mateo, CA, 416–428.

KITSUREGAWA, M., TANAKA, H., AND MOTO-OKA, T. 1983. Application of hash to database machine and its architecture. *New Gen. Comput. 1*, 1.

LLOYD, J. W. 1987. *Foundations of Logic Programming (2nd ed)*. Springer-Verlag, Berlin.

MCCARTHY, D. R. AND DAYAL, U. 1989. The architecture of an active data base management system. In *Proceedings of the ACM SIGMOD Symposium on the Management of Data* (Portland, OR, June), 215–224.

MORGENSTERN, M. 1983. Active databases as a paradigm for enhanced computing environments. In *Proceedings of the International Conference on Very Large Data Bases* (Florence, Italy, Sept.), 34–42.

RAMAKRISHNAN, R., SRIVASTAVA, D., AND SUDARSHAN, S. 1992. CORAL: Control, relations and logic. In *Proceedings of the International Conference on Very Large Data Bases* (Vancouver, B.C., Aug.), 238–250.

RAMAKRISHNAN, R., SRIVASTAVA, D., SUDARSHAN, S., AND SESHADRI, P. 1993. Implementation of the CORAL deductive database system. In *Proceedings of the ACM SIGMOD Symposium on the Management of Data* (Washington D.C., May), 165–176.

SCHMIDT, J. W. 1977. Some high level language constructs for data of type relation. *ACM Trans. Database Syst. 2*, 3 (Sept.), 247–261.

SCHNEIDER, D. AND DEWITT, D. 1989. A Performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of SIGMOD* (Portland, OR, June), 110–121.

SEVERANCE, D. G. AND LOHMAN, G. M. 1976. Differential files: Their application to the maintenance of large databases. *ACM Trans. on Database Syst. 1*, 3 (Sept.).

SHAPIRO, L. 1986. Join processing in database systems with large main memories. *ACM Trans. Database Syst. 13*, 2 (Sept.).

SIMON, E. AND DE MAINDREVILLE, C. 1988. Deciding whether a production rule is relational computable. In *Proceedings of the International Conference on Database Theory*, 205–222.

SIMON, E. AND KIERNAN, J. 1995. A declarative approach to active databases: The A-RDL language and system. In *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom and S. Ceri, Eds., Morgan-Kaufmann, San Francisco, CA (to appear).

STONEBRAKER, M. 1992. The integration of rule systems and database systems. *IEEE Trans. Knowl. Data Eng. 4*, 5 (Oct.), 415–423.

STONEBRAKER, M., JHINGRAN, A., GOH, J., AND POTAMIANOS, S. 1990. On rules, procedures, caching and views in data base systems. In *Proceedings of the ACM SIGMOD Symposium on the Management of Data* (Atlantic City, NJ, May), 281–290.

STOY, J. E. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA.

WIDOM, J. 1993. The Starburst active database rule system, 1993. Available by anonymous ftp to db.stanford.edu in the file pub/widom/1993/starburst-rule-system.ps.

WIDOM, J. AND CERI, S. 1995. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, San Francisco, CA.

WIDOM, J. AND FINKELSTEIN, S. J. 1990. Set-oriented production rules in relational database systems. In *Proceedings of the ACM SIGMOD Symposium on the Management of Data* (Atlantic City, NJ, May), 259–264.

WOODFILL, J. AND STONEBRAKER, M. 1983. An implementation of hypothetical relations. In *Proceedings of the International Conference on Very Large Data Bases* (Sept.), 157–165.

ZHOU, Y. AND HSU, M. 1990. A theory for rule triggering systems. In *International Conference on Extending Data Base Technology*, 407–421.

ZHOU, G., GHANDEHARIZADEH, S., AND HULL, R. 1994. Benchmarking the Heraclitus[Alg, C] prototype. Tech. Rep. USC-CS-94-582, University of Southern California, August 15. Available in file HERACLITUS/TR-582 at ftp@perspolis.usc.edu.

ZHOU, G., HULL, R., KING, R., AND FRANCHITTI, J-C. 1995. Using object matching and materialization to integrate heterogeneous databases. In *Proceedings of the Third International Conference on Cooperative Information Systems (CoopIS-95)* (Vienna, Austria, May).

Received September 1994; revised August 1995; accepted December 1995