ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Paolo Atzeni and Val Tannen (Eds)

Database Programming Languages (DBPL-5)

Proceedings of the Fifth International Workshop on Database
Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995

Paper:

On Detecting Conflict Between Proposed Updates

Michael Doherty, Richard Hull, Marcia Derr and Jacques
Durand

Published in collaboration with the
British Computer Society

On Detecting Conflict Between Proposed Updates

Michael Doherty, Richard Hull *

Computer Science Department, University of Colorado
Boulder, CO 80309-0430
{doherty,hull}@cs.colorado.edu

Marcia Derr, Jacques Durand

U S WEST Advanced Technologies
4001 Discovery Drive
Boulder, CO 80303

Abstract

In the Heraclitus paradigm, a *delta value* or more simply, *delta*, is a concrete value that corresponds to a difference between database states. This paper presents the Heraclitus[OO] (abbreviated H2O) DBPL, which provides a syntax and semantics for working with deltas in the context of object-oriented databases.

The paper also considers the use of deltas in connection with detecting conflict between pairs of proposed updates to a database. This is useful in contexts where multiple users are each creating and choosing between multiple possible updates. A semantically based notion of potential conflict between proposed updates is developed, along with several conservative approximations based on the use of different kinds of delta.

1 Introduction

There are a variety of application areas involving the creation and analysis of complex updates that may or may not be applied to the database. This frequently arises in systems that support the delivery of complex services (e.g., telecommunication or transportation services to corporations). In these contexts, the proposed updates are characterized by the following:

- Long duration – Multiple, potentially conflicting, updates may be in progress for extended periods, perhaps days or longer. The partial updates may need to be accessed and modified multiple times.
- Hypothetical access – Many of the updates may never actually be applied to the database. For example, a customer may explore different possible changes to their service. It is useful to access and compare the hypothetical database states corresponding to the application of different proposed updates.
- Potential conflict – Updates proposed by different users might conflict with each other, e.g., because they modify the same object. Moreover, the very notion of conflict and how to resolve it is typically dependent on the application.

In this paper we develop a versatile framework for creating and managing multiple long duration proposed updates, and for efficiently detecting potential conflict between them.

The first contribution of this paper is to present the syntax and semantics of the Heraclitus[OO] (abbreviated H2O) DBPL [1]. The H2O DBPL generalizes the Heraclitus paradigm [2, 3, 4] to the object-oriented database model. In particular, the H2O DBPL supports the creation and manipulation of *delta values* (or more simply, *deltas*); these are concrete, tabular values that correspond to differences between object-oriented database states. Deltas are first-class citizens in the H2O DBPL, and can be accessed in a hypothetical fashion. Importantly, the delta corresponding to an expression ϵ specifying a proposed update allows for efficient repeated access to the proposed change, without re-evaluating ϵ each time. Deltas are used in our framework to represent the net effect of proposed updates, and provide the starting point for our framework for detecting potential conflict between them. The H2O DBPL is also of independent interest, because of the broad family of applications for which deltas can be applied, including hypothetical database access, version management, and active databases [4].

*This research was supported in part by NSF grant IRI-931832, ARPA grants BAA-92-1092 and 33825-RT-AAS, and a sponsored research agreement from U S WEST Advanced Technologies.

The second contribution of this work is the development of a semantically based notion of *potential conflict* that addresses the issue of detecting conflict between proposed updates, along with a variety of approximations of it. The semantic notion of conflict focuses primarily on (a) the database values read during execution of the expressions specifying the proposed updates, and (b) the specific database modifications that the proposed updates call for. Testing this “pure” notion of potential conflict will typically be quite expensive. To remedy this, we use the deltas of the H2O DBPL to develop a family of conservative approximations of this notion. (Although not considered here, we expect that this framework can be extended to provide automatic and semi-automatic mechanisms for resolving conflicts between proposed updates.)

The work presented here is motivated in part by problems arising in the processing of customer change requests at U S WEST, a regional telephone company (see [5] for more on the application of Heraclitus in telecommunications). We believe that the framework will be useful to other applications requiring long transactions, including the broad area of engineering design.

Related Work A variety of systems (see [6, 7]) provide mechanisms for the disciplined creation and modification of versions, so that conflicts can be isolated and manually resolved before the base data is corrupted. The H2O DBPL provides language primitives that would support the relatively easy specification of customized version management systems.

Research on managing conflict between deltas has been done in the context of software engineering using “program dependence graphs” [8, 9], and in the field of configuration management using “change sets” [10, 11]. In the first case the analysis of conflict and the algorithms for merging deltas are restricted to the specific application of interest, and in the second case the notion of conflict used is primarily text-based. In contrast, our work develops a general framework for using deltas and detecting potential conflict between them. Speaking intuitively, we exploit the structure provided by a database schema to assist in providing relatively tight approximations of the semantically based notion of conflict.

Our ideas for adding application specific information to refine our approximations of potential conflict between updates is related to research in adding semantic information to concurrency control protocols [12, 13]. The main distinction is that our conflict analysis is performed on the delta values which represent partial or complete updates, while the semantic concurrency control is based on traditional notions of locking. Also, we provide for the possibility of modifying proposed updates in case of conflict, rather than simply aborting the update.

Organization of the Paper The remainder of this paper is organized as follows. A motivating example stemming from the U S WEST application is presented in Section 2. In Section 3 we present an overview of the current H2O DBPL prototype. The semantically based definition of potential conflict is given in Section 4, and approximations for this notion based on deltas are given in Section 5. Brief conclusions are presented in Section 6. Due to space limitations, some of the technical details concerning the H2O DBPL are presented in the Appendix (see also [1]).

2 A Motivating Example

This section introduces a motivating example, which will be used to illustrate various concepts in this paper. The example concerns managing requests from large business customers to change their telephone service. It is a simplified and abstracted version of a real-world example encountered at U S WEST.

The telephone needs of business clients can be very complex, and, as a result, requests to upgrade or modify the telephone service provided to businesses can be intricate. In many cases a business client may wish to explore several alternative ways to upgrade their service before committing to any one of them. For example, if a business is thinking of acquiring voice-mail service, it may wish to contract with the phone company to provide voice-mail, or it may wish to purchase additional lines from the phone company but contract with a third party to provide in-house voice-mail services. After the costs of these alternatives are estimated, the business client may need time (possibly several hours or days) to consider and/or refine them before making a final decision.

A partial, simplified schema of a database that might be used by a telecommunications company for managing customer service is shown in Figure 1. The `BusinessCustomer` class holds information about business customers that contact the telephone company for service. This class defines attributes such as name, customer id, and billing address. The `Location` class holds information about customer sites that receive telephone service. The `LocalExchange` class holds information about the “local exchanges” controlled by the telephone company; these provide switching services at a macro level. The number of local exchanges in an area depends on the size and density of the area. To

```

interface BusinessCustomer {
    attribute string name;
    ...
};

interface Location {
    attribute string address;
    ...
};

interface LocalExchange {
    attribute string id;
    ...
};

interface PBX {
    attribute string id;
    attribute BusinessCustomer customer;
    attribute Location loc;
    attribute LocalExchange le;
    attribute int maxTrunks;
    attribute int noTrunks;
    attribute Set<PBX> pbxSet;
};

```

Figure 1: Part of schema for telecommunications example.

keep the example simple, we omit the full details for the `BusinessCustomer`, `Location`, and `LocalExchange` classes.

The `PBX` class holds information about Private Branch Exchanges (PBXs); these are switches that are installed in customer locations. (Speaking loosely, a local exchange considers the first three digits of a seven-digit phone number, and a PBX considers the remaining four digits.) Each PBX connects to at most one local exchange via one or more trunks. The maximum number of trunks is given by attribute `maxTrunks`. The number of trunks currently assigned is given by attribute `noTrunks`. Each PBX can also connect to PBXs located at other customer sites. These connections are represented by the set attribute `pbxSet`. The lines that connect these PBXs are not explicitly represented in this example schema.

Figure 2 shows an example of (part of) a state of the database, along with two proposed updates against it. More specifically, the upper left corner of the diagram shows the current state of a subset of the database for a single customer. In the current state, the ACME Brewing Company has two PBXs located at two different sites. Each PBX is connected to its own local exchange by several trunks. In addition, the two PBXs are directly connected to each other. This state is shown textually in Figure 3. The notation for describing an object is the name of the object's class followed by the object's OID and a list of attribute values in parentheses. Attributes are listed in the same order as the attribute names in the class declarations in Figure 1.

The first update in Figure 2 increases by four the number of trunks in the PBX named 'pbx2'. The second update installs a new PBX in a third location, and connects it to both 'pbx1' and 'pbx2', removing the existing connection between 'pbx1' and 'pbx2' (see also Example 4.1).

3 An Overview of the H2O DBPL

In this section we describe the syntax and semantics used for deltas in the H2O DBPL¹. A more complete description can be found in [1].

The current H2O DBPL implements a straightforward kind of delta, which we term *object-granularity* deltas. These deltas can be viewed as a natural generalization of the deltas developed in Heraclitus[Alg,C] [4] for the relational model, with objects replacing tuple values. Section 5 introduces refinements of object-granularity deltas, which are more effective in the context of merging deltas and detecting conflicts between them.

3.1 Object-granularity deltas and their operators

We begin our discussion by describing deltas and some algebraic operations on them; these form one part of the semantic domain of H2O. This will prepare us for describing the H2O constructs for creating, accessing and manipulating deltas,

¹The H2O DBPL is one part of the larger H2O project [14], which includes research on multi-database-state services, activeness, and interoperation between databases and software systems.

Figure 2: A (partial) state and two proposed updates.

```

BusinessCustomer(b1: 'ACME Brewing Company', ...)

Location(l1: '101 Main St', ...)
Location(l2: '622 Elm St', ...)
Location(l3: '851' Second Ave', ...)

LocalExchange(e1: 'lex1', ...)
LocalExchange(e2: 'lex2', ...)

PBX(p1: 'pbx1', b1, l1, e1, 64, 3, {p2})
PBX(p2: 'pbx2', b1, l2, e2, 64, 6, {p1})
    
```

Figure 3: Current state for the telecommunications schema.

i.e., the syntactic domain of H2O.

In the context of object-granularity deltas, there are three kinds of *atomic updates*. These are now listed with their intuitive semantics.

- *ins* $c(o : v)$ - Insert a new object of class c with oid o and value v .
- *mod* $c(o : v)$ - Modify the object of class c with oid o to have the new value v .
- *del* $c(o)$ - Delete the object of class c with oid o .

In the first two cases, v is assumed to have type corresponding to c . In each case, o is called the *target* of the atomic update.

Definition: An *object-granularity delta value*, or ' delta_{og} ', is a set of atomic updates, such that no oid is the target of two atomic updates. There is also a special delta_{og} denoted *fail*, which corresponds intuitively to inconsistency.

To illustrate, consider the two updates shown in Figure 2. The corresponding delta_{og} s are:

$$\begin{aligned}
 \Delta_{update_1} &= \{ \text{mod } PBX(p2 : 'pbx2', b1, l2, e2, 64, 10, \{p1\}) \} \\
 \Delta_{update_2} &= \left\{ \begin{array}{l} \text{mod } PBX(p1 : 'pbx1', b1, l1, e1, 64, 3, \{p3\}), \\ \text{mod } PBX(p2 : 'pbx2', b1, l2, e2, 64, 6, \{p3\}), \\ \text{ins } PBX(p3 : 'pbx3', b1, l3, NULL, 64, 0, \{p1, p2\}) \end{array} \right\}
 \end{aligned}$$

(In the examples here, only one class is involved in each delta. In general, a delta_{og} can include atomic updates that affect multiple classes.)

A fundamental algebraic operator in connection with delta_{ogs} is *apply*, which has the effect of applying the changes called for in the delta_{og} to a database state. Since H2O can create and manipulate hypothetical states, the intuitive definitions for the semantics of atomic updates are not sufficiently precise to immediately yield the definition of *apply*. For example, it is possible that a delta containing an atomic update $\text{mod } c(o : v)$ may be applied to a state in which o is not present. Figure 4 in the Appendix defines the semantics of the *apply* of a delta Δ on a state DB , by describing the impact of Δ at an object by object level. (Due to space limitations, we cannot dwell on many of the motivations and subtleties for the definitions presented here.)

The computation of delta values corresponding to update expressions is specified by the *trace* operator. The value of *trace* captures the sequence of changes to the objects in the database that would occur if the expression were actually executed. The resulting delta value is dependent on the types of the affected objects and the delta form being used. For object-granularity deltas, the trace function can be informally described as follows: (i) the update expression, ϵ , is executed in hypothetical mode, (ii) for each OID o of type τ mentioned by ϵ , atomic commands to insert, delete, or modify the value of o are recorded, and (iii) the net effect of ϵ on o is incorporated into the final delta value. The *trace* operator is used for defining deltafication (see next subsection) and a similarly defined operator, *retrace*, is used for defining reverse deltafication.

One operator for combining deltas is *smash*, denoted using ‘!’. This is loosely related to the composition operator ($;$) found in imperative programming languages. The basic property of smash is that for each state DB and each pair Δ_1, Δ_2 of deltas, $\text{apply}(\text{apply}(DB, \Delta_1), \Delta_2) \equiv \text{apply}(DB, \Delta_1 ! \Delta_2)$ (see Figure 5).

The result of *smash* on the two delta_{ogs} given above is:

$$\begin{aligned} \Delta_{update_1} ! \Delta_{update_2} &= \left\{ \begin{array}{l} \text{mod } PBX(p1 : 'pbx1', b1, l1, e1, 64, 3, \{p3\}), \\ \text{mod } PBX(p2 : 'pbx2', b1, l2, e2, 64, 6, \{p3\}), \\ \text{ins } PBX(p3 : 'pbx3', b1, l3, NULL, 64, 0, \{p1, p2\}) \end{array} \right\} \\ \Delta_{update_2} ! \Delta_{update_1} &= \left\{ \begin{array}{l} \text{mod } PBX(p1 : 'pbx1', b1, l1, e1, 64, 3, \{p3\}), \\ \text{mod } PBX(p2 : 'pbx2', b1, l2, e2, 64, 10, \{p1\}), \\ \text{ins } PBX(p3 : 'pbx3', b1, l3, NULL, 64, 0, \{p1, p2\}) \end{array} \right\} \end{aligned}$$

(Smash is not commutative for this example, even though it would appear intuitively that the two updates as originally described should commute. The non-commutativity here stems in part from the use of object-granularity deltas; the “attribute-granularity” deltas of Section 5 overcome this problem.)

Another binary operator on deltas is *merge*, denoted using ‘&’. Intuitively, this operator returns the union of the effects of the two deltas, if there is no conflict between them. The semantics of merge is given in Figure 6 in the Appendix. In general, the merge operator is defined to satisfy the following property:

$$\Delta_1 \& \Delta_2 \equiv \begin{cases} \Delta_1 ! \Delta_2 & \text{if } \Delta_1 ! \Delta_2 = \Delta_2 ! \Delta_1 \text{ and no object is deleted} \\ & \text{by one delta and modified by the other} \\ fail & \text{otherwise} \end{cases}$$

As illustrated above, the *smash* on Δ_{update_1} and Δ_{update_2} is not commutative, because the two delta_{ogs} have conflicting *mod* atoms targeting object $p2$. Thus, $\Delta_{update_1} \& \Delta_{update_2} = fail$. (If attribute-granularity deltas were used the merge would be defined.)

3.2 Creating, accessing, and manipulating deltas

In this subsection, we briefly present some elements of the syntax and semantics of the H2O DBPL constructs for the creation, access and manipulation of deltas. The H2O DBPL is defined as an extension to the C++ binding of the ODMG standard [15] for object-oriented databases. The syntax of the H2O extensions to the ODMG object manipulation language (OML) for manipulating deltas is given in Figure 7 in the Appendix. Deltas are implemented by the system-defined class `delta`, and are treated as first-class database objects.

As presented in this paper, deltas in the H2O DBPL range over differences between database states. Thus, in defining the semantics of the H2O DBPL we carefully separate² the database state from the program state. Speaking

²The separation presented here is a simplification of how the H2O DBPL actually works (see [1]).

loosely, we view the class hierarchy to consist of two separate pieces, one being the database schema and the other being the program schema. (Persistence is orthogonal to the distinction between database schema and program schema.) A *database state* is a function from classes in the database schema to sets of oids with values (subject to the usual constraints). A *program state* is a similar function for the program schema, along with a function from all program variables to values (which may be oids from either schema) of appropriate type. We model the full state in a program execution as an ordered pair (PS, DB) with program state PS and database state DB .

Because the H2O DBPL is an extension of C++, all syntactic objects in the language are expressions. The semantics of these are defined by the *expression valuation function*, $\mathcal{E}[\![\]\!]$, which maps expressions from the *syntactic domain* of the H2O DBPL to the *semantic domain* of algebraic objects. More precisely, the signature of $\mathcal{E}[\![\]\!]$ is given by

$$\mathcal{E} : Expression \rightarrow (ps \times db) \rightarrow (val \times (ps \times db)),$$

which means that an expression yields a function that maps each program state/database state pair into a value and a program state/database state pair. In general, we will be interested in only one of these components, and will use the shorthand notation $\mathcal{E}_{val}[\![\]\!]$, $\mathcal{E}_{ps}[\![\]\!]$, and $\mathcal{E}_{db}[\![\]\!]$ respectively to indicate application of the valuation $\mathcal{E}[\![\]\!]$ and extraction of the particular component of interest.

One subtlety not immediately apparent from the semantic definitions is the treatment of oids, in particular, the generation of new oids. To simplify our presentation, we model the oid generator as an (infinite) set of unconsumed oids that exists as a persistent object, and view it as part of the program state. Whenever a new oid is needed (either in a regular or a hypothetical expression) an oid is removed from this set. The set of unconsumed oids is kept in the program state, rather than the database state, so that it is not affected by hypothetical operations, and so that uniqueness of oids is guaranteed.

The H2O DBPL provides basic operators for creating, accessing, and combining deltas. We briefly describe three important operators here; these are used in the discussion of potential conflict which follows. Additional operators are described in the Appendix.

Deltafication: This provides one way to create deltas. The *deltafication* of an arbitrary expression ϵ (which may have side-effects³) is the expression $[\![\epsilon]\!]$. Intuitively, evaluation of this expression yields the delta that corresponds to the execution of ϵ . The delta $\Delta = \mathcal{E}_{val}[\![\epsilon]\!](PS, DB)$ (the delta_{og} associated with $[\![\epsilon]\!]$ in state (PS, DB)) is $\text{trace}(\epsilon, PS, DB)$ and has the property that $\text{apply}(DB, \Delta) = \mathcal{E}_{db}[\![\epsilon]\!](PS, DB)$. This means that the result of applying the delta Δ to DB is the same as the result of executing the expression ϵ on (PS, DB) .

Apply: If δ is a delta expression, then the statement `apply  $\delta$` has the effect of first evaluating δ , and then applying it to the database state.

Hypothetical Application: Hypothetical access to a delta is obtained using the `when` operator. Speaking loosely, if δ is a delta expression and α an arbitrary side-effect free expression, then evaluating `when  $\delta$   $\alpha$` in (PS, DB) yields the value that *would* be obtained if α were evaluated against the database state⁴ $\text{apply}(DB, \mathcal{E}_{val}[\![\delta]\!](PS, DB))$, i.e., against the state obtained by applying δ to the current state.

The primary use of `when` is for hypothetical evaluation of side-effect free expressions. If the expression α in `when  $\delta$   $\alpha$` has side effects on the database, these side effects will not be realized outside of the context of the expression.

4 Pairs of Delta Expressions that are Semantically Conflict Free

This section attempts to lay out a general, semantically-based framework for understanding when proposed updates are conflict free. The following section discusses how delta values can be used to approximate this.

The very notion of ‘conflict’ is quite controversial – exactly what should we take as the definition of conflict between proposed updates? Because the notion of conflict is highly dependent on the underlying application, almost any general definition we choose will be an approximation of the “true” notion of conflict. In this section we propose a formal, semantically-based notion of “conflict free” that focuses on the actual code used to define the proposed updates, and which is independent of particular application domains. More specifically, this notion examines the values read by the expressions specifying proposed updates and the values actually changed by them. Although this notion has short-comings, it provides a starting point for discussion.

In this section and the next, we focus on proposed updates specified using delta expressions of the form $\delta = [\![\epsilon]\!]$; these correspond to the hypothetical execution of a proposed update specified by the expression ϵ . At the end of the

³There are subtleties around whether the side-effects of ϵ have effect when the expression $[\![\epsilon]\!]$ is executed. In general, side-effects on PS do occur, but the side-effects on DB do not.

⁴ δ might have impact on PS ; this is not reflected in the equation here. See the Appendix for the complete definition.

section we briefly consider “interactive delta expressions”, which might model more closely the change requests arising in the telephone company example. We have not yet investigated the notion of conflict free for delta expressions built using variables, the binary operators, or reverse deltafication.

Suppose now that $\delta_1 = [\langle \epsilon_1 \rangle]$ and $\delta_2 = [\langle \epsilon_2 \rangle]$ are delta expressions, i.e., expressions that describe proposed updates to a (partial) database state. Both δ_1 and δ_2 might involve multiple reads of the database state, in addition to multiple writes (and modifies) against it.

Our main focus is on how ϵ_1 and ϵ_2 affect the database state. For this reason we generally assume that the computations represented by ϵ_1 and ϵ_2 are not dependent on program state variables (i.e., program state variables are just scratch-pad-like objects). Thus, in this section and the next we typically write $\mathcal{E}_{val}[\epsilon_1](DB)$ rather than the technically more correct $\mathcal{E}_{val}[\epsilon_1](PS, DB)$.

A preliminary notion is that of “read interference”. Speaking informally, $\delta_2 = [\langle \epsilon_2 \rangle]$ “interferes” with $\delta_1 = [\langle \epsilon_1 \rangle]$ if the sequence of values read by δ_1 when executed in DB is different⁵ from the sequence of values read by δ_1 when executed in the result of applying δ_2 to DB . But what is meant by the notion of “values read”? We now define two “modes” for reading: “answer-based read monitoring” and “envelope-based read monitoring”. These modes can be viewed as the ends of a spectrum of possibilities.

Suppose first that there is a strict separation between database queries, database updates, and other processing, as for example occurs in C with embedded SQL. In this case the “sequence of values read” by a delta expression $\delta = [\langle \epsilon \rangle]$ can refer to⁶ the sequence of answers to database queries executed. A separation between database accesses and other processing might also be enforced when using, e.g., CO₂ or some other commercial OODB language. We shall call this *answer-based read monitoring*.

In many cases a discipline of separating database access from other computations may not be followed. In this case a more crude notion of “values read” might be used. A very basic approach is to define the “sequence of values read” to be the sequence of all objects explicitly retrieved along with their full values. Speaking intuitively, this sequence corresponds to an “envelope” around the set of objects and values that actually affect the evaluation of the delta expression. We shall call this (*object-granularity*) *envelope-based read monitoring*.

Before continuing we briefly consider the impact of constraints. In this paper we consider any automatic constraint checking and/or repair mechanisms in use to be an implicit part of the expression ϵ . Thus, any data read as a part of automatic constraint checking will be included in the “values read”, and any database modifications automatically performed to repair constraints will be included as updates called for by ϵ . The issue of constraints is a topic for further research.

We now have:

Definition: Let $\delta_1 = [\langle \epsilon_1 \rangle]$ and $\delta_2 = [\langle \epsilon_2 \rangle]$, and let DB be a database state. Then δ_2 (*read*) *interferes* with δ_1 in DB under answer-based [envelope-based] read monitoring if the sequence of answers to queries asked [objects accessed and their values] by δ_1 when executed in DB is different from the sequence of answers to queries asked [objects accessed and their values] by δ_1 when executed in $\mathcal{E}_{db}[\epsilon_2](DB)$.

When testing whether δ_2 interferes with δ_1 in DB , we expect that $\mathcal{E}_{val}[\delta_2](DB)$ will be evaluated to yield the delta value Δ_2 . If d is a delta variable holding Δ_2 , then testing for interference can be done by comparing the execution of δ_1 and the execution of δ_1 when d .

Our semantically based definition of conflict free is:

Definition: Delta expressions $\delta_1 = [\langle \epsilon_1 \rangle]$ and $\delta_2 = [\langle \epsilon_2 \rangle]$ are *semantically conflict free* in state DB , under answer-based [envelope-based] read monitoring if

- (i) δ_2 does not interfere with δ_1 in DB under answer-based [envelope-based] read monitoring,
- (ii) δ_1 does not interfere with δ_2 in DB under answer-based [envelope-based] read monitoring,
- (iii) ϵ_1 and ϵ_2 commute on DB , i.e., executing $\epsilon_1 ; \epsilon_2$ and executing $\epsilon_2 ; \epsilon_1$ on DB yield the same result, and
- (iv) there is no oid o such that ϵ_1 has the effect in DB of deleting o and ϵ_2 has the effect in DB of modifying the value of o (or visa versa).

Two delta expressions have *semantic potential conflict* (under a given mode of read monitoring) if they are not semantically conflict free.

⁵A subtlety not elaborated on here is that in this context equality is modulo isomorphism of oids.

⁶There are three subtleties in connection with the notion of “values read” that we do not address in this paper. First, we should also include here the sequence of error messages possibly generated during database updates that occurred during the hypothetical execution of ϵ . Second, for technical reasons, we assume that all subexpressions of the form $\tau = v$ (i.e., assignment of value v to term τ) occurring in ϵ include an explicit database access of the form $read(\tau)$. And third, we assume that there is no explicit derived data in the schema. Intuitively, this implies that each value in the database is represented in only one place.

Speaking loosely, the first two conditions focus more on whether there is conflict between δ_1 and δ_2 in terms of what they *access* from the database. The last two conditions focus more on conflict in terms of how they *change* the database. The primary motivation for the fourth condition is the intuition that there is a conflict between attempting to delete an object and attempting to modify it. (See also the semantics of merge in Figure 6.)

Example 4.1: We now present three examples that illustrate these notions, using Figure 2 of Section 2.

(a) For this case, assume that the code for ϵ_1 selects all PBX objects having `id` equal to `'pbx2'` and increases the number of trunks by four. For simplicity, we will ignore the constraint `noTrunks  $\leq$  maxTrunks` in this example. Expressed in pidgin OQL and OML, the code for ϵ_1 is:

```
change_set_a = select y from y in PBX where y.id = 'pbx2';
foreach y in change_set_a { y.noTrunks = y.noTrunks + 4; };
```

Assume further that the code for ϵ_2 installs a new PBX in location 13 and connects it to both `'pbx1'` and `'pbx2'`, removing the existing connection between `'pbx1'` and `'pbx2'`. The code for ϵ_2 (i) poses two queries, one for PBXs with attribute `id = 'pbx1'` and one for PBXs with attribute `id = 'pbx2'`; (ii) inserts a new PBX object; and (iii) updates all `pbxSet` attributes to reflect the new connections.

In this case, under answer-based read monitoring δ_1 does not interfere with δ_2 nor visa-versa. (This is true in part because all of the queries in ϵ_1 and ϵ_2 return only the oids of various objects, but none of their attribute values.) Under envelope-based read monitoring both interfere with the other, because both expressions read the object `p2`, and both modify values of attributes of `p2`. In summary, δ_1 and δ_2 are semantically conflict-free under answer-based read monitoring, but have potential semantic conflict under envelope-based read monitoring.

(b) For this case, suppose that ϵ_1 is:

```
change_set_b = select y from y in PBX, x in PBX
                where x.id = 'pbx1' and y.le != NULL
                and y in x.pbxset and x in y.pbxset;
foreach y in change_set_b { y.noTrunks = y.noTrunks + 4; };
```

Intuitively, `change_set_b` holds all PBX objects that are directly connected to `'pbx1'` and also connected to a local exchange. Let ϵ_2 be as before. In this case, under answer-based read monitoring δ_1 does not interfere with δ_2 , but δ_2 does interfere with δ_1 . Also, in this case the executions of ϵ_1 and ϵ_2 do not commute. In sum, δ_1 and δ_2 have potential semantic conflict under both modes of read monitoring.

(c) Suppose now that the code for ϵ_1 computes both `change_set_a` and `change_set_b` as in the above cases, but then uses `change_set_a` when increasing the number of trunks. In this case the database access to compute `change_set_b` is essentially ignored in the code of ϵ_1 , but answer-based read monitoring is unable to discern that. Now, δ_1 and δ_2 have potential semantic conflict under both modes of read monitoring, even though ϵ_1 and ϵ_2 commute. $\square$

Suppose now that delta expressions $\delta_1 = [\langle \epsilon_1 \rangle]$ and $\delta_2 = [\langle \epsilon_2 \rangle]$ are semantically conflict free. This means that ϵ_1 and ϵ_2 can be applied to the database state in either order to yield the same result, and that they do not conflict with regards to deleting vs. modifying the same object. At an intuitive level, application of ϵ_1 and ϵ_2 in either order yields a result that captures the intention of both.

Suppose now that we do not want to apply these, but rather, we want to create a delta expression that combines the intention of both. If they are semantically conflict-free, both of the delta expressions $[\langle \epsilon_1 ; \epsilon_2 \rangle]$ and $[\langle \epsilon_2 ; \epsilon_1 \rangle]$ yield the desired “merge” of the two delta expressions.

We noted earlier that if a notion of delta value is available, then testing for interference between delta expressions can be performed by using delta values and hypothetical evaluation under when. If the algebraic merge operator is defined appropriately for delta values, this can be extended to form a sufficient condition for semantic conflict free. For example (proof omitted):

Proposition 4.2: Let DB be a state, $\delta_1 = [\langle \epsilon_1 \rangle]$, and $\delta_2 = [\langle \epsilon_2 \rangle]$. Assume the use of δ_{alg} values and the merge operator of Section 3. Suppose that $\Delta_i = \mathcal{E}_{val}[\delta_i](DB)$ for $i = 1, 2$. Then, using either mode of read

monitoring:

$$\left. \begin{array}{l} \delta_1 \text{ does not interfere with } \delta_2 \text{ in } DB \\ \delta_2 \text{ does not interfere with } \delta_1 \text{ in } DB \\ \Delta_1 \ \& \ \Delta_2 \neq \text{fail} \end{array} \right\} \Rightarrow \delta_1 \text{ and } \delta_2 \text{ are semantically conflict free in } DB$$

In this case, $\mathcal{E}_{val}[\llbracket \langle \epsilon_1 ; \epsilon_2 \rangle \rrbracket](DB) = \Delta_1 \& \Delta_2$. (This result remains true for the other notions of delta value and merge presented in this paper.)

The notion of semantic potential conflict can be generalized to apply to sets of delta expressions.

In some applications a proposed update might be the result of an extended interaction between human and computer, rather than the result of evaluating an expression of the form $\llbracket \langle \epsilon \rangle \rrbracket$. We have developed a notion of *interactive delta expression*, which is a sequence of queries, actions, and pauses (for human reflection). There are natural generalizations to interactive delta expressions of the notion of semantically conflict free, and of the notions developed in Section 5.

5 Customizing Delta Values

In the previous section we introduced the notions of semantically conflict free and semantic potential conflict. We expect that testing for these properties will typically be prohibitively expensive. This leads to the development of conservative, syntactically-based approximations of conflict free which are much less expensive to test. There will be two basic dimensions in the creation of these approximations: (a) different notions of delta values developed for different classes of objects, and (b) different approaches to approximating the “read set” of a delta expression. The primary focus in this section is on issue (a).

We first show how the object-granularity delta values of Section 3 can be used to approximate the notion of semantically conflict free. We then develop two refinements of delta values that provide closer approximations of semantically conflict free. The first refinement is quite general and the other is much more application specific. We are developing a framework [16] in which that different kinds of delta values can be used for different classes in a single object-oriented database schema.

5.1 Object-granularity conflict free

The driving idea of the notion of object-granularity conflict free is to use object-granularity deltas when approximating conditions (iii) and (iv) in the definition of semantically conflict free. In this subsection we develop this notion in the context of answer-based read monitoring, and briefly indicate how the notion is defined for envelope-based read monitoring.

In order to test for possible conflict, we extend the notion of delta to include information about the set of values read. In the context of answer-based read monitoring, we shall use the following approach.

Definition: A *query-extended delta_{og} (value)* is a pair (Q, Δ) where

- (i) Q is a sequence of⁷ closures of queries, and
- (ii) Δ is a delta_{og} as in Section 3.

Intuitively, Q corresponds to the sequence of database queries asked by a delta expression δ , and Δ captures the specific modifications called for by δ .

Definition: Let $D_1 = (Q_1, \Delta_1)$ and $D_2 = (Q_2, \Delta_2)$ be two query-extended delta values. Then D_1 and D_2 are (*object-granularity*) *conflict free (under answer-based read monitoring)* in state DB if (abusing notation somewhat):⁸,

- (i) $\mathcal{E}_{val}[\llbracket Q_1 \rrbracket](DB) = \mathcal{E}_{val}[\llbracket Q_1 \text{ when } \Delta_2 \rrbracket](DB)$,
- (ii) $\mathcal{E}_{val}[\llbracket Q_2 \rrbracket](DB) = \mathcal{E}_{val}[\llbracket Q_2 \text{ when } \Delta_1 \rrbracket](DB)$, and
- (iii) $\Delta_1 \ \& \ \Delta_2 \neq \text{fail}$.

⁷Let ϵ be an expression, and q a query occurring in ϵ , and consider an execution of ϵ for state DB . The *closure* of an occurrence of q is the assignment of variables to values that holds for the variables occurring in q at the point in the execution of ϵ where q is invoked.

⁸The notion of evaluation of a query is extended to sequences of queries in the natural fashion.

Otherwise they have *object-granularity potential conflict* in DB (under answer-based read monitoring).

Speaking intuitively, condition (iii) here captures the spirit of both conditions (iii) and (iv) of the definition of semantic conflict free.

Example 5.1: Recall Example 4.1, and consider case (a) of that example. Let $D_i = (Q_i, \Delta_i)$ be the query-extended delta_{og} corresponding to δ_i for $i = 1, 2$. Then conditions (i) and (ii) in the definition of object-granularity potential conflict between D_1 and D_2 are not violated. However, $\Delta_1 \& \Delta_2 = \text{fail}$, because both deltas modify the value of $p2$. Thus, condition (iii) is violated. (In Example 5.3 we shall see that if attribute-granularity deltas are used, then condition (iii) is not violated.)

Consider now case (b) of Example 4.1. In that case condition (i) of object-granularity conflict free between D_1 and D_2 is violated. This is also true for case (c). $\square$

It is easily verified that object-granularity conflict free under answer-based read monitoring is a conservative approximation of semantic conflict free:

Proposition 5.2: Assume object-granularity delta values. Let DB be a state, and $\delta_1 = [\langle \epsilon_1 \rangle]$ and $\delta_2 = [\langle \epsilon_2 \rangle]$ be two delta expressions. Let $D_i = (Q_i, \Delta_i)$ be the query-extended delta_{og} corresponding to δ_i in DB for $i = 1, 2$. Then, relative to DB ,

$$(Q_1, \Delta_1) \text{ and } (Q_2, \Delta_2) \text{ are object-granularity conflict free} \Rightarrow \delta_1 \text{ and } \delta_2 \text{ are semantically conflict free.}$$

Analogous results can be developed for envelope-based read monitoring. Briefly, an *(object-granularity) envelope-extended delta_{og} value* is a pair (R, Δ) where R is a set of oids along with their values, and Δ is a delta_{og} value. Given a delta expression δ and state DB , the corresponding envelope-extended delta_{og} value will be $(R, \mathcal{E}_{val}[\delta](DB))$, where R is the set of all database oids and their values that are accessed and/or hypothetically changed in the evaluation δ . Two such delta_{og} s are *(object-granularity) conflict free (under envelope-based read monitoring)* if (i) $R_1 \cap R_2 = \emptyset$, and (ii) if $(\Delta_1 \& \Delta_2) \neq \text{fail}$. The analog of Proposition 5.2 is easily developed.

5.2 Attribute-granularity delta values and conflict

A straightforward refinement of object-granularity deltas is to redefine atomic modifies so that they focus on the modification of a single attribute value, rather than the full value of an object. Delta values constructed in this way are called *attribute-granularity delta values*, denoted delta_{ag} . The notions of apply, smash and merge are defined for delta_{ag} s in the natural manner, as are attribute-granularity conflict free and potential conflict. Also, the notion of envelope-based read monitoring can be refined to focus on individual attribute values rather than full object values. The analog of Proposition 5.2 is easily demonstrated.

Example 5.3: Under the delta_{ag} semantics, the delta values corresponding to the delta expressions of Figure 2 are:

$$\begin{aligned} \Delta'_{update_1} &= \{ \text{mod } PBX(p2 : \text{noTrunks} : 10) \} \\ \Delta'_{update_2} &= \left\{ \begin{array}{l} \text{mod } PBX(p1 : \text{pbxSet} : \{p3\}), \\ \text{mod } PBX(p2 : \text{pbxSet} : \{p3\}), \\ \text{ins } PBX(p3 : \text{'pbx3'}, b1, l3, NULL, 64, 0, \{p1, p2\}) \end{array} \right\} \end{aligned}$$

In this case, we have $\Delta'_{update_1} ! \Delta'_{update_2} = \Delta'_{update_2} ! \Delta'_{update_1} = \Delta'_{update_1} \& \Delta'_{update_2} =$

$$\left\{ \begin{array}{l} \text{mod } PBX(p1 : \text{noTrunks} : 10), \\ \text{mod } PBX(p1 : \text{pbxSet} : \{p3\}), \\ \text{mod } PBX(p2 : \text{pbxSet} : \{p3\}), \\ \text{ins } PBX(p3 : \text{'pbx3'}, b1, l3, NULL, 64, 0, \{p1, p2\}) \end{array} \right\}.$$

Recall Example 4.1(a), and let $D_i = (Q_i, \Delta_{update_i})$ be the query-extended delta_{ag} corresponding to δ_i ($i = 1, 2$). Then D_1 and D_2 are attribute-granularity conflict free. $\square$

The approach of attribute-granularity delta values can be refined further, by going deeper into the structure of individual attributes. For example, the `pbxSet` attribute is set-valued. Modifications to the value of this attribute might be represented as a family of individual inserts and deletes to the set. Under this approach, the first modify of Δ'_{update_2} would be represented as

$$mod\ PBX(p1 : pbxSet : ins\ p3, del\ p1).$$

Another direction is possible if a family of operators are defined on the type of some attribute. For example, addition and subtraction are defined on the type non-negative integer, i.e., the type of `noTrunks`. We might permit atomic delta values of the form

$$mod\ PBX(p1 : noTrunks : +4).$$

This notion of defining specialized delta operators for specific data types has been formalized as *delta forms* and delta form templates have been defined for standard complex value types [16]. In the next subsection, we illustrate by presenting one specific example of a specialized delta form.

5.3 Deltas for objects consisting of undifferentiated sets

We now present a specialized kind of delta value that is useful in contexts where there are undifferentiated sets of objects. Such undifferentiated sets arise in some applications at U S WEST. More generally, this discussion illustrates how delta values can be customized for a particular data type. We present this specialized delta in terms of a concrete example; we have also developed a formal specification.

The class `PBX` presented in Figure 1 in Section 2 is simplified in a variety of ways. In reality, a record must be kept, for each `PBX`, of which individual trunks are connected to which individual ports at the local exchange. Thus, a slightly more realistic interface for `PBX` is given by:

```
interface PBX' {
  attribute string id;
  attribute BusinessCustomer customer;
  attribute Location loc;
  attribute LocalExchange le;
  attribute int maxTrunks;
  attribute Set<Tuple<trunk_id,le_line_id>> trunk_assignment;
  attribute Set<PBX> pbxSet;
};
```

Here the `trunk_assignment` attribute holds a set of ordered pairs⁹ (t, l) , where t is a `trunk_id` for the `PBX` and l is a `line_id` for the local exchange that the `PBX` is connected to, and this forms a 1-1 correspondence. To simplify this discussion, we shall omit the `noTrunks` attribute. Using the class `PBX'`, the object corresponding to $p1$ in the running example might be

```
PBX'(p1: 'pbx1', b1, l1, e1, 64, { <t1,lep25>, <t2,lep26>, <t3,lep27> }, {}),
```

indicating that the trunk `t1` is attached to the local exchange port `lep25`, etc.

Consider now two proposed updates δ_1 and δ_2 , that both assign an additional trunk line of `'pbx1'`. Under a straightforward choice of δ_{a_g} s that permit explicit inserts and deletes to set-valued attributes, the following might be associated with these two updates:

$$\begin{aligned}\Delta_1 &= \{mod\ PBX'(p1 : trunk_assignment : ins\ <t4,lep88>)\} \\ \Delta_2 &= \{mod\ PBX'(p1 : trunk_assignment : ins\ <t4,lep253>)\}\end{aligned}$$

Speaking intuitively, when creating each delta the system apparently chose the “next available” `trunk_id` from `'pbx1'`. This has the unfortunate side effect that Δ_1 and Δ_2 have semantic conflict, since the requirement that there be a one-to-one correspondence in `trunk_assignment` will be violated if both δ_{a_g} s are applied. However, the two

⁹The ODMG ODL does not actually support a `Tuple` type, rather the use of a `struct` (as in C) is required. We take the liberty of using `Tuple` here to specify ordered tuples, in order to simplify the notation.

delta expressions are compatible, because the total number of trunks used by the original state and the two deltas is less than 64.

Our solution to this problem is based on the fact that the `trunk_id`'s should be viewed as undifferentiated. We suppose that a new class `virtual_trunk_id` is created, with an infinite number of elements. If creating a delta value for the class `PBX'` for which an unused `trunk_id` is needed, then we use an unconsumed element of `virtual_trunk_id` in its place. Thus, the two delta expressions above would now be evaluated into:

$$\begin{aligned}\Delta'_1 &= \{mod\ PBX'(p1:trunk_assignment:ins < v\bot1,lep88 >)\} \\ \Delta'_2 &= \{mod\ PBX'(p1:trunk_assignment:ins < v\bot2,lep253 >)\}\end{aligned}$$

(The exchange line `id`'s might also be undifferentiated; we ignore that possibility here to simplify the discussion.)

Briefly, a delta Δ using such undifferentiated values is consistent relative to a database state DB for a given PBX p if the total number of trunks already in use for p plus the number of `virtual_trunk_id`'s for p occurring in Δ does not exceed the value of $p.maxTrunks$. The semantics for applying such a delta involves a non-deterministic assignment of actual `trunk_id`'s to `virtual_trunk_id`'s. The smash of two such delta values is defined in the natural fashion, but yields inconsistency (*fail*) if the total number of `trunk_id`'s used in the smash exceeds the bound given by `maxTrunks`. Analogous comments apply to merge. Importantly, we permit the use of `virtual_trunk_id`'s in the same places as `trunk_id`'s – this permits the construction of complex deltas with attributes of other objects referring to newly assigned trunk lines.

6 Conclusions

This paper has two foci: (a) introducing the syntax and semantics for deltas in the H2O DBPL, and (b) laying some of the groundwork for using deltas to help detect potential conflict between proposed database updates. An example arising in a regional telephone company was used to motivate and illustrate the development.

This paper raises a host of issues. In Section 5 we described two refinements of object-granularity deltas, and indicated how they provided better approximations of the notion of semantic conflict free. We are also developing customized notions of delta that are appropriate for trees and graphs. An important open problem is to develop a framework for specifying customized delta values in a uniform fashion, that can be compiled into implementations for them (see [16] for initial results in this direction).

Another issue concerns constraints. Suppose that an active database rulebase is used to monitor constraint violations and to repair them. It might be appropriate for the delta associated with the expression $\delta = [\epsilon >]$ to include the impact of any constraint repairs that would have occurred were ϵ executed. Thus, our prototype implementation of the H2O DBPL would need to be extended to incorporate hypothetical rule firing. The merge and smash operators on delta values would also need to be generalized. (Some difficulties would arise if the rule base is not “semantically consistent” in the sense of [17], i.e., if the rule base yields different results on update requests that are semantically equivalent but syntactically different.)

In this paper we focused on detecting conflict between proposed updates. A natural direction is to develop mechanisms that would help to resolve conflicts that are detected. Active database technology looks promising here. Unlike previous applications of active databases, in this context the rules should be able to explicitly refer to deltas as independent objects.

Although not discussed in this paper, we have found that using an equivalent but different database schema can impact the various approximations of semantic conflict free. The presence of derived data can also impact the various approximations. Thus, the decision to use (a particular variety of) deltas to help detect conflicting updates may have an impact on the schema design process.

Acknowledgements

The Heraclitus project has evolved dramatically since its inception five years ago, and has been impacted by the many researchers involved in discussions on various aspects of the project. We are especially grateful to Dean Jacobs for his extensive insight concerning the Heraclitus paradigm. We also thank Mohammed Rupawalla and other members of the H2O research group at CU Boulder, including especially Omar Boucelma, Jean-Claude Franchitti, Roger King, and Gang Zhou, for many interesting discussions on deltas for object-oriented databases and related topics.

References

- [1] M. Doherty, R. Hull, and M. Rupawalla. The Heraclitus[OO] database programming language, 1996. Technical Report in preparation.
- [2] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 455–468, 1991.
- [3] D. Jacobs and R. Hull. Database programming with delayed updates. In *Intl. Workshop on Database Programming Languages*, pages 416–428, San Mateo, Calif., 1991. Morgan-Kaufmann, Inc.
- [4] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus[Alg.C]: Elevating deltas to be first-class citizens in a database programming language. Technical Report USC-CS-94-581, Computer Science Department, Univ. of Southern California, 1994. revised August, 1995.
- [5] M. Derr, J. Durand, M. Doherty, R. Hull, and M. Rupawalla. Applications of Heraclitus in telecommunications information processing. Technical report, University of Colorado, Boulder, 1996. Submitted for publication, available via anonymous ftp at <ftp://ftp.cs.colorado.edu/users/hull/h2o:telecom.ps>.
- [6] N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.
- [7] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, December 1990.
- [8] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Trans. on Programming Languages and Systems*, 11(3):345–387, July 1989.
- [9] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proc. of Fourteenth ACM Intl. Conf. on Software Engineering*, pages 392–411, 1992.
- [10] Inc. Software Maintenance & Development Systems. Aide-De-Camp software management system, product overview, 1989.
- [11] R. D. Cronk. Tributaries and deltas. *Byte*, 1992.
- [12] M. P. Herlihy and W. E. Weihl. Hybrid concurrency control for abstract data types. In *Proc. ACM Symp. on Principles of Database Systems*, pages 201–210, 1988.
- [13] R. F. Resende, D. Agrawal, and A. El Abbadi. Semantic locking in object-oriented database systems. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 388–402, 1994.
- [14] O. Boucelma, J. Dalrymple, M. Doherty, J. C. Franchitti, R. Hull, R. King, and G. Zhou. Incorporating active and multi-database-state services into an OSA-compliant interoperability framework. In *The Collected Arcadia Papers, Second Edition*. University of California, Irvine, May 1995.
- [15] R.G.G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, San Mateo, California, 1993.
- [16] M. Doherty, R. Hull, and M. Rupawalla. Structures for manipulating proposed updates in object-oriented databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 306–317, 1996.
- [17] P. Picouet and V. Vianu. Semantics and expressiveness issues in active databases. In *Proc. ACM Symp. on Principles of Database Systems*, 1995.
- [18] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

		occurrence of o in Δ			
presence of o in DB		absent	$ins\ c(o : v_2)$	$mod\ c(o : v_2)$	$del\ c(o)$
	absent	absent	$c(o : v_2)$	absent	absent
	$c(o : v_1)$	$c(o : v_1)$	$c(o : v_2)$	$c(o : v_2)$	absent

 Figure 4: Semantics of the apply operator on individual oid o

		occurrence of o in Δ_2			
occurrence of o in Δ_1		absent	$ins\ c(o : v_2)$	$mod\ c(o : v_2)$	$del\ c(o)$
	absent	absent	$ins\ c(o : v_2)$	$mod\ c(o : v_2)$	$del\ c(o)$
	$ins\ c(o : v_1)$	$ins\ c(o : v_1)$	$ins\ c(o : v_2)$	$ins\ c(o : v_2)$	$del\ c(o)$
	$mod\ c(o : v_1)$	$mod\ c(o : v_1)$	$ins\ c(o : v_2)$	$mod\ c(o : v_2)$	$del\ c(o)$
	$del\ c(o)$	$del\ c(o)$	$ins\ c(o : v_2)$	$del\ c(o)$	$del\ c(o)$

 Figure 5: Semantics of the smash operator (!) on individual oid o

A Syntax and Semantics of the H2O DBPL

This appendix gives some additional details on the H2O DBPL; see [1] for a full specification.

The specification for the apply, smash and merge operations on object-granularity deltas are given in Figures 4, 5¹⁰ and 6.¹¹

The syntax of the H2O extensions to the ODMG object manipulation language (OML) is presented in the BNF grammar in Figure 7. The productions *h2o_stmt*, *when_exp* and *delta_exp* define new H2O constructs. The remaining productions show how these constructs are integrated with C++ and come from the specification given in Appendix A of [18].

The semantics of the H2O DBPL syntactic constructs is given in Figure 8. The semantics are defined by the expression valuation function, $\mathcal{E}[\cdot]$, which maps expressions from the *syntactic domain* of the H2O DBPL to the *semantic domain* of algebraic objects. As noted in Section 3, states are viewed to be pairs (PS, DB) where PS denotes the *program state* and DB denotes the *database state*. Since the H2O DBPL is based on C++, we follow the convention that expressions can have side-effects on the state. Thus, under $\mathcal{E}$ each expression yields a mapping from a state to a value (the result of the expression) and a new state. Similarly, statements are treated as expressions that yield the null value.

In Section 3 we described the deltafication, apply, and hypothetical access constructs of the H2O DBPL. The other major constructs for creating, accessing and combining deltas are now described.

Creating deltas: Deltas are created in the H2O DBPL by “deltafication” and “reverse deltafication”.

Reverse Deltafication: The *reverse deltafication* of a statement ϵ (which may have side-effects) is the expression $[> \epsilon <]$. Intuitively, the semantics is to execute ϵ on the current state, and to return a delta that can “undo” the effect of ϵ .

Accessing deltas: As described in Section 3, the H2O DBPL provides two ways to access a delta, using *apply* and *when*.

Combining delta expressions: The H2O DBPL provides keywords corresponding to the binary operators smash and merge. It also provides a third binary operator called “compose”.

¹⁰In this and the following table, we have not considered examples such as $ins\ c(o : v)$ in the first delta and $ins\ c'(o : v')$ in the second, i.e., where an OID is associated with distinct classes in the two deltas. This is because we assume that OIDs are typed according to which class they can be assigned.

¹¹If the resulting delta contains any occurrences of *fail*, then the delta itself takes on the special value *fail*.

		occurrence of o in Δ_2			
		absent	$ins\ c(o : v_2)$	$mod\ c(o : v_2)$	$del\ c(o)$
occurrence of o in Δ_1	absent	absent	$ins\ c(o : v_2)$	$mod\ c(o : v_2)$	$del\ c(o)$
	$ins\ c(o : v_1)$	$ins\ c(o : v_1)$	$v_1 = v_2 : ins\ c(o : v_1)$ $v_1 \neq v_2 : fail$	$v_1 = v_2 : ins\ c(o : v_1)$ $v_1 \neq v_2 : fail$	<i>fail</i>
	$mod\ c(o : v_1)$	$mod\ c(o : v_1)$	$v_1 = v_2 : ins\ c(o : v_1)$ $v_1 \neq v_2 : fail$	$v_1 = v_2 : mod\ c(o : v_1)$ $v_1 \neq v_2 : fail$	<i>fail</i>
	$del\ c(o)$	$del\ c(o)$	<i>fail</i>	<i>fail</i>	$del\ c(o)$

 Figure 6: Semantics of the merge operator (&) on individual oid o

smash: Loosely¹² speaking, the keyword `smash` is translated into the `smash (!)` operator:

$$\mathcal{E}_{val} \llbracket \delta_1 \text{ smash } \delta_2 \rrbracket (PS, DB) = \mathcal{E}_{val} \llbracket \delta_1 \rrbracket (PS, DB) ! \mathcal{E}_{val} \llbracket \delta_2 \rrbracket (PS', DB)$$

(where $PS' = \mathcal{E}_{ps} \llbracket \delta_1 \rrbracket (PS, DB)$). Note here that δ_2 is evaluated in the initial database state DB , rather than in a context with database state $apply(DB, \mathcal{E}_{val} \llbracket \delta_1 \rrbracket (PS, DB))$.

merge: The keyword `merge` is analogous to `smash`.

compose: This operator does not have an analog in the algebra of deltas. The expression $\delta_1 \text{ compose } \delta_2$ yields a delta that corresponds to the result of evaluating δ_1 and then applying it, followed by evaluating δ_2 and then applying it.

Intuitively, we would expect evaluation of a delta expression to have no side effects on the state. Consider for example a delta expression $\delta = [< \epsilon >]$. As indicated by the semantics of Figure 8, evaluation of δ leaves the database state unchanged. However, the program state may change, both because (i) new oids might be used, or because (ii) ϵ involves explicit modification to program variables. The latter case reflects a design decision motivated primarily by pragmatic considerations.

Some delta expressions can have side-effects on the database. One example is reverse deltafication: $[> \epsilon <]$ yields a delta value and also executes ϵ against the current state. Other examples arise as the result of the C++ language. For example, if x holds an object o_{17} whose value has type integer, then the right-hand side of the H2O expression $d = [< \epsilon >], x = 2;$ yields a delta, but has the effect of changing the value of o_{17} to 2. Although we discourage this kind of expression, we do not prohibit it. For this reason, when defining the semantics of $\delta_1 \text{ smash } \delta_2$ we allow the possibility that evaluating δ_1 might have side-effects on the database state. Analogous remarks apply to the semantics of `merge`, `compose`, and `when`.

¹²We ignore here the possibility that evaluation of δ_1 can have side-effect on the database state; see below.


```

stmt ::=      h2o_stmt
              || exp_stmt
              || comp_stmt
              || other_stmt

exp_stmt ::= {exp}

exp ::=      assgn_exp
              || exp , assgn_exp

assgn_exp ::= h2o_exp
              || unary_exp assgn_op assgn_exp
              || cond_exp

h2o_stmt ::= apply delta_exp ;           /* apply statement */

h2o_exp ::=  when_exp
              || delta_exp

when_exp ::= when delta_exp exp         /* when expression */

delta_exp ::= delta_var                 /* delta variable */
              || [< stmt >]              /* deltafication */
              || [> stmt <]              /* reverse deltafication */
              || delta_exp smash delta_exp /* delta combination */
              || delta_exp merge delta_exp /* delta combination */
              || delta_exp compose delta_exp /* delta combination */
              || ( delta_exp )

```

Figure 7: Syntax of H2O DBPL extensions to ODMG OML

$$\begin{aligned}
\mathcal{E} : \text{Expression} &\rightarrow (ps \times db) \rightarrow (val \times (ps \times db)) \\
\mathcal{E}[[l < \epsilon >]](PS, DB) &= \\
&\quad \text{let } (V, (PS', DB')) = \mathcal{E}[[\epsilon]](PS, DB) \text{ in} \\
&\quad (trace(\epsilon, PS, DB), (PS', DB)) \\
\mathcal{E}[[l > \epsilon <]](PS, DB) &= \\
&\quad \text{let } (V, (PS', DB')) = \mathcal{E}[[\epsilon]](PS, DB) \text{ in} \\
&\quad (revtrace(\epsilon, PS, DB), (PS', DB')) \\
\mathcal{E}[[\text{apply } \delta ;]](PS, DB) &= \\
&\quad \text{let } (\Delta, (PS', DB')) = \mathcal{E}[[\delta]](PS, DB) \text{ in} \\
&\quad (NULL, (PS', apply(DB', \Delta))) \\
\mathcal{E}[[\text{when } \delta \epsilon]](PS, DB) &= \\
&\quad \text{let } (\Delta, (PS', DB')) = \mathcal{E}[[\delta]](PS, DB) \text{ in} \\
&\quad \text{let } (V, (PS'', DB'')) = \mathcal{E}[[\epsilon]](PS', apply(DB', \Delta)) \text{ in} \\
&\quad (V, (PS'', DB')) \\
\mathcal{E}[[\delta_1 \text{ smash } \delta_2]](PS, DB) &= \\
&\quad \text{let } (\Delta_1, (PS', DB')) = \mathcal{E}[[\delta_1]](PS, DB) \text{ in} \\
&\quad \text{let } (\Delta_2, (PS'', DB'')) = \mathcal{E}[[\delta_2]](PS', DB') \text{ in} \\
&\quad (\Delta_1 ! \Delta_2, (PS'', DB'')) \\
\mathcal{E}[[\delta_1 \text{ merge } \delta_2]](PS, DB) &= \\
&\quad \text{let } (\Delta_1, (PS', DB')) = \mathcal{E}[[\delta_1]](PS, DB) \text{ in} \\
&\quad \text{let } (\Delta_2, (PS'', DB'')) = \mathcal{E}[[\delta_2]](PS', DB') \text{ in} \\
&\quad (\Delta_1 \& \Delta_2, (PS'', DB'')) \\
\mathcal{E}[[\delta_1 \text{ compose } \delta_2]](PS, DB) &= \\
&\quad \text{let } (\Delta_1, (PS', DB')) = \mathcal{E}[[\delta_1]](PS, DB) \text{ in} \\
&\quad \text{let } (\Delta_2, (PS'', DB'')) = \mathcal{E}[[\delta_2]](PS', apply(DB', \Delta_1)) \text{ in} \\
&\quad (\Delta_1 ! \Delta_2, (PS'', DB'')) \\
\mathcal{E}[[x = \epsilon ;]](PS, DB) &= \\
&\quad \text{let } (V, (PS', DB')) = \mathcal{E}[[\epsilon]](PS, DB) \text{ in} \\
&\quad (V, updatevar(x, V, (PS', DB')))
\end{aligned}$$

Figure 8: Portion of H2O DBPL semantics