A Formal Lazy Replication Regime for Spreading Conversion Functions Over Objectbases

Clara Smith
LINTI, Universidad Nacional de La Plata,
and Consejo Nacional de Investigaciones
Científicas y Técnicas, Argentina.
e-mail: csmith@ada.info.unlp.edu.ar

Carlos A. Tau
LINTI, Universidad Nacional de La Plata,
and L.B.S. Informática.
La Plata, Buenos Aires, Argentina.
Fax: +54 21 258816

Abstract. This paper introduces a functional-flavored formalization of objectbase evolution processes. It also determines features of the failure equivalence concept between the two main approaches for the fulfillment of database conversions: immediate and lazy updates.

1 Introduction

Schema evolution commands produce a significant impact on the bases, as objects have to accomodate theirselves to the new specifications. The updates are usually expressed using *system* or *user-defined* conversion functions. The objective of this study is to clarify and formally specify objectbase evolution concepts and change replication regimes, within a mathematical framework. Our work combines some of the strongest features of the objectbase literature and formal methods of specification and design. The source objectbase model, SIGMA, is the result of our grade thesis [ST93] at the Informatics Department, Universidad Nacional de La Plata, Buenos Aires, Argentina. SIGMA's formal description, including semantic domains, interpretation functions and denotations for the Object Manipulation Language (OML) appear in [Tau94] [Tau95]. Despite the *ad hoc* choice of SIGMA, should be noticed that the notions introduced in this article fit any current typical objectbase model, due to the simplicity and precision achieved on the definitions.

The rest of the paper is organized as next: Section 2 briefly recalls core SIGMA components. It explains how adjustments and adaptations are applied to concrete objectbase stores in order to suit new schema organizations, using the renowned immediate and lazy update replication policies. Furthermore, we analyze two equivalences between both approaches: observational and failure equivalence. Section 3 introduces the *conversion queue*, a representation for retaining updates at hand. The *mix_map* regime for implementing lazy base updates is also presented. Section 4 shows how the mix_map regime can be considered failure equivalent to the immediate approach. Finally, our conclusions are exposed in section 5.

2 Mapping Conversions on the Bases

A SIGMA **schema** is defined as a 5-tuple <*sid*, T, I, A, R>; where *sid* is the schema identifier, T is a certain set of types and relationships selected to conform the schema, I is a parallel set of well-formed implementations for types in T, R is a set

of general integrity constraints defined for the schema, and A is a function providing customizations for types in T. SCH = (ID x T x I x CUSTOM x AXIOMS) is a semantic domain for SIGMA schemas, where CUSTOM and AXIOMS are primitive domains for custom-built axioms and integrity constraints respectively [Tau95], and ID is a primitive domain for names [Wir86]. A SIGMA **base** is defined as a 4-tuple <*bid*, π, σ, ζ>, where *bid* is the base identifier, π is an object composition environment (with *is-part-of* and client references among objects), σ is the concrete object store, and ζ is the type-extention (or **class**) **system**. The semantic domain BASE = (ID x COMP_ENV x STORE x CLASS_SYS) for SIGMA bases and its constituent domains are strictly analyzed and defined in [Tau95]. Finally, we describe an **objectbase system** as a sequence of schemas, each of which governs a list of bases. This domain is expressed as OBASE_SYS = (SCH x (BASE)*)*.

One single SIGMA schema may serve as the conceptual reference for several bases. Therefore, there is a need to spread every schema evolution over each subordinate base, without exception, as they have to be updated to be brought up to a consistent state with respect to the new governor schema. This kind of propagation is usually expressed using a *map* operation, that receives a conversion function and a group of bases and *effectuates* every update. Its functional definition follows:

map :: FUNCTION $\rightarrow$ BASE* $\rightarrow$ BASE*

map cf [ ] = [ ]

map cf b:tail = *apply*(cf, b):*map*(cf tail)

The symbol : above stands for item chaining. A*pply* is a high order function that receives a function and an object (a base in this case), and executes the function using such object as a real parameter. Update mappings are also present in [Mon92].

Following the main two strategies for the accomplishment of database renewals, namely immediate and lazy updates [Kim89] [Tre93] [Fer94], we can give at least two meanings from an implementation viewpoint to the map effect: *immediate map* and *lazy map*. The former instantaneously replicates the conversions in every subordinate base, while the latter keeps pending changes until a base activation is solicited. But from a formal viewpoint, map should be interpreted within a unique final semantics: the immediate one. The lazy map effect is guaranteed to be the same as the immediate map effect if we are sure all bases will be touched once again during their lifetime.

Definition 2.1 *Lazy maps are always equivalent to immediate maps, as bases accomodate theirselves to changes when they are going to be observed (activated).*

This definition (which improves the one given in [Fer94]) determines that every base conforms to the schema change, due to the fact that we are merely spectators of base conversions. If we care about the application of conversion functions in a more detailed way, we will probably conclude things look not so plain. To illustrate, suppose each of the several serviceable parts of an objectbase system has its own identity, which we shall call **agent**. This term is used broadly, but we will refer to agents when talking about a discrete set of (possibly atomic) actions. Each action is either an interaction with a neighbour agent, or it may occur independently of them. Agents are always observable to the system. Definition 2.1 states an observational

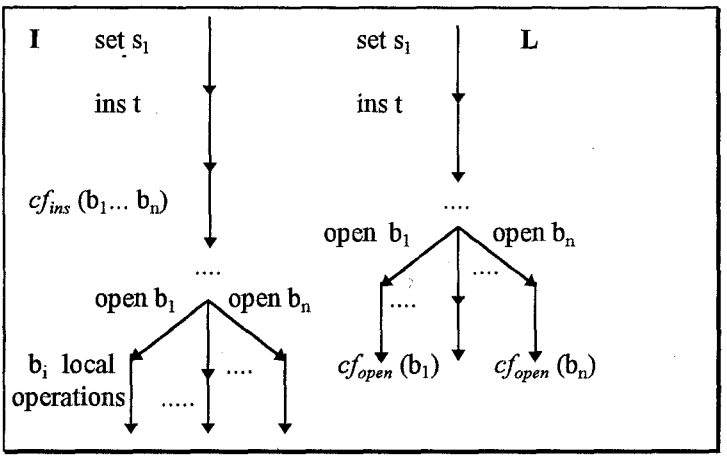

Figure 1 - Traces for a Type Insertion: Immediate and Lazy Policies

equivalence that is perhaps the simplest of all we can deal with: it equates two agents if and only if they can perform exactly the same sequences of observable actions. It can also be called **trace equivalence** (written $\approx$) [Mil89].

Let a schema designer write the sequence *...; set s_1; insert t; ...* in a transaction body, in which first schema s_1 is activated, and then a type t is placed in s_1. Next, suppose s_1-governed bases, say b_1 ... b_n, are activated. Both agents I and L in figure 1 outline what it may occur during the transition from one consistent base configuration to another in identical copies of a single objectbase system, using immediate and lazy map policies respectively. As conversion functions are unobservable to end users, we conclude I $\approx$ L.

The main disadvantage of trace equivalence is that it may equate a deadlocking agent to one which does not deadlock. This certainty indicates that trace equivalence concept, applied to our agents, is too weak. An interesting and stronger equivalence is Hoare's **failure equivalence** ($\approx_f$) [Mil89], that equates two agents if both have the same set of failures. A failure is a pair *(t,{l})* where *t* is a trace and *{l}* is a set of action labels, meaning *t* can be performed and thereby reach a state in which no further activity is possible if the environment will only permit actions in *{l}*. It turns out that I and L do not possess the same failures. Trace I has, among others, the failure *(set ins,{l})* $\forall$ *{l}* such that $cf_{ins} \notin$ *{l}*. This one does not belong to L's failure set, thus I $\approx_f$ L is false. Therefore, this couterexample shows how the lazy execution of a conversion function cannot be considered failure equivalent to the execution of the same conversion function treated immediately.

3 Lazy Maps Need Conversion Queues

In the worst case, evolutions may be deferred for lazy applications. Besides, a schema may undergo several changes before its bases are touched. We will associate to each base a *conversion queue* to maintain pending updates. Such a queue is merely an entry that contains FIFO-arranged references to conversion functions.

$$\frac{\begin{array}{c} \textit{ass\_schemas}(\beta) \rightarrow l\varepsilon \\ \textit{get}(l\varepsilon, \textit{sch\_named(s)}) \rightarrow \xi \\ \textit{ass\_bases}(\beta,\xi) \rightarrow l\mu \\ \textit{include}(\ (\textit{new\_base}\ b\ \xi)\ l\mu) \rightarrow l\mu' \\ (\textbf{replace}\ <\xi, l\mu>\ \textbf{in}\ \beta\ \textbf{with}\ <\xi, l\mu'>) \rightarrow \beta' \end{array}}{(\textbf{NEW\_BASE}\ b\ \textbf{FOR SCHEMA}\ s\ \beta) \rightarrow \beta'}$$

Figure 2 - Operational Semantics for the Base Creation Operation

We embody a CQUEUE primitive domain for queues with the BASE domain to configure (BASE x CQUEUE), a domain for bases and conversion queues. Under this representation, when generating a new base we just point out its name and the handpicked governor schema's identifier. Its new empty conversion queue will be attached automatically, it is imperceptible and intangible to end base-users. Figure 2 depicts an operational semantics [Plo81] for such SIGMA's OML command. The symbol $\rightarrow$ in the equation means *reduces to*, or should be more specifically understood as an evaluation relation [Hen90]. The formula has the following interpretation: *ass_schemas* picks up the various current valid schemas ($l\varepsilon$) for the objectbase (β). Next, once verified the schema name, *ass_bases* takes all s-governed bases ($l\mu$). *Include* does the item piercing job. A definition for *new_base* follows:
new_base:: ID $\rightarrow$ SCH $\rightarrow$ (BASE x CQUEUE)
new_base b ξ = (<b, π_0, σ_0, ζ_0>$_\xi$, [ ]$_b$); where:
$\pi_0 = \lambda(x,y).$*unbound* (empty ξ–composition environment), $\sigma_0 = \lambda x.$*unused* (vacant ξ–store), $\zeta_0 = \lambda$**id.if** *type*(ξ,id) **then** <*type_of*(x)=id,{}> **else** *unbound*, (unoccupied ξ–class system); and [ ]$_b$ is b allied empty conversion queue. Moreover, *type*(ξ,id) returns the semantic object true (*tt*) if id is the name for a type in ξ. *Type_of*(x) retrieves the type name of an object x. We use the **replace in with** constructor to partially replace some sorts in a specification and generate a new one.

Definition 3.1: *Let mix_map be a lazy replication policy defined over SIGMA bases and conversion queues. Its formal functionality is specified as:*
mix_map :: FUNCTION $\rightarrow$ (BASE x CQUEUE)* $\rightarrow$ (BASE x CQUEUE)*
mix_map cf [ ] = [ ]
mix_map cf (b,cq):tail = (b, *push* cf cq):*mix_map*(cf tail)

4 Immediate vs. Mix_Map: Failure Equivalence

We consider a **complex agent** as a set of actions organized in a unit of atomicity. This notion is related to the one of transaction [Cat94]; both can be simply blended.

Definition 4.1: *Complex agents are deduced from user-transaction traces, and inserted in system traces as follows:*
• For the immediate map policy, given an update *uc* for a schema *s* in a transaction body, a complex agent is built as ***<uc s; map cf$_{uc}$ ass_bases(s)>***, where the second

component stands for the effective instantaneous map (section 2) of the suitable conversion function onto the s-governed bases. The system trace is expanded with this complex agent. For base activation commands, no complex agents need to be built, as any immediate base activation ignores the remaining bases.

• For the mix_map regime, i) due to an update *uc* applied over a schema *s*, the system derives a complex agent of the form ***<uc s ; mix_map cf_{uc} assoc_bases(s)>***. *Assoc_bases* is a redefinition of *ass_bases*, its outcome fits the domain defined for pairs base-queue in section 3. ii) From an activation command *ac*, the system obtains a complex agent configured as ***<ac b ; dequeue b cq_b>***. Function *dequeue* clears cq_b via the performance of its queued renewals over b.

Proposition 4.2: *A system trace built up from a user-transaction trace using the immediate policy is failure equivalent to a system trace under the mix_map regime, built up from the same user-transaction using complex agents.*

Proof sketch: the proof uses the induction principle over the trace structure and the number of schema updates and base openings ocurring in the trace.

One schema update. Let T be a user transaction and let *uc* be the unique update command (for a given schema s) in T. Two system traces, say I and M, are built identically for the immediate and mix_map policies respectively, except for the agent involving *uc*. Trace I holds a duple of the form *<uc s; map cf_{uc} ass_bases(s)>* as a complex agent, expanding *uc*'s original place. M contains the complex agent *<uc s; mix_map cf_{uc} assoc_bases(s)>*, replacing *uc*'s earliest location. Thus, traces I and M are identical in structure for those commands which are not updates. For the only update in T they both have a complex agent in the same place, and complex agents are atomic. Thus, I and M have the same set of failures. Therefore I $\approx_f$ M.

One base opening. Let T be a user transaction, let *ac* be the unique base activation command in T. System traces I and M are built exactly alike for the immediate and mix_map regimes, except for the agent involving *ac*. For trace I, the agent is atomic and it is namely *ac*, as the base opening is executed directly over the desired base, without any mapping over other bases. Trace M holds, covering *ac*'s position, the complex agent *<ac b, dequeue b cq_b)>*. Thus, I and M are identical in structure for those commands that are not base openings. For the only *ac* in T, I holds the atomic agent *ac* and M maintains the complex agent built above, and complex agents are atomic. We conclude I $\approx_f$ M.

Multiple schema updates and multiple openings. Let uc_1, ..., uc_n be a sequence of updates for a given schema s arbitrarily found in T, and let ac_1, ..., ac_m be base activation commands arbitrarily spread over T. If the failure equivalence between both regimes is proven for n updates and also holds for m openings, then:

i) The equivalence is proven also in case of an additional update uc_{n+1} included in T following the first n updates. Traces I and M upto the agent immediately before uc_{n+1} are by hypothesis failure equivalent. For every uc_i in T, trace I holds the complex agents *<uc_i s; map cf_{uci} ass_bases(s)>*, and M contains the complex agents *<uc_i s; mix_map cf_{uci} assoc_bases(s)>* in places where T originally has a uc_i.

ii) In addition, the trace equivalence between I and M is also proven in case of an additional opening ac_{m+1}, as by hypothesis I and M are failure equivalent upto the

agent placed just before ac_{m+1}. Trace I holds the original ac_i agents, and M possesses $<ac\ b_i;\ dequeue\ b_i\ cq_{bi}>$ as complex agents for each ac_i.

Therefore, traces I and M for a transaction T involving certain uc_1, ..., uc_{n+1} update commands and ac_1, ..., ac_{m+1} base openings are failure equivalent. □

5 Conclusions

Immediate and lazy (mix) maps can be used as alternate policies for the concretion of objectbase evolutions. For a particular situation, the most convenient regime may be applied, and the other left for a better occasion. Conversion queues act as an appropriate bank for change sequences, as they do not interfere with the original objectbase model's base format. This observation suggest a notable facility for adapting the use of conversion queues in current objectbase systems. We think the endeavor to give rigorous definitions and earnestly remark the formalities related to the trace equivalence concept in the framework of objectbase updates is valuable: we believe the relevance of this type of equivalence is significant because it strongly cares in the efficacy of the system.

References

[Cat94] - *The Object Database Standard: ODMG-93 Release 1.1.* R. Cattell, ed. Morgan Kaufmann Series in Database Management, San Francisco, CA, 1994.

[Fer94] - *Implementing Lazy Database Updates for an Object Database System.* F. Ferrandina, T. Meyer, R. Zicari. Proceedings of the 20th VLDB Conference, 261-272. Chile, 1994.

[Hen90] - *The Semantics of Programming Languages.* M. Hennessy. J. Wiley & Sns., 1990.

[Kim89] - *Features of the ORION OODBMS.* W. Kim, N Ballou et al. In Object-Oriented Concepts, Databases and Applications, W. Kim, F. Lochovsky, eds. ACM Press, 1989.

[Mil89] - *Communication and Concurrency.* R. Milner. M. Hoare series, Prentice Hall, 1989.

[Mon92] - *Lazy Evaluation of Intensional Updates in Constraint Logic Programming.* D. Montesi, R. Torlone. Proceedings of the 2nd International Computer Science Conference, 502-508. Hong Kong, 1992.

[Plo81] - *A Structural Approach to Operational Semantics.* G. Plotkin. Lecture Notes, Aarhus University, 1981.

[ST93] - *A Unified Model for Object-Oriented Databases.* C. Smith, C. Tau. Grade Thesis. Informatics Department, Universidad Nacional de La Plata, Argentina. February 1993.

[Tau94] - *Formalization of Object Manipulation Concepts in the Denotational Semantics Framework.* C. Tau, C. Smith, C. Pons, A. Monteiro, G Baum. 20th VLDB Conference Poster Paper Collection, 47-56. Santiago, Chile, September 1994.

[Tau95] - *Formally Speaking About Schemata, Bases, Classes and Objects.* C. Tau, C. Smith, C. Pons, A. Monteiro. 4th International Symposium on Database Systems for Advanced Applications, Singapore. World Scientific Publishing Co., 308-317. April 1995.

[Tre93] - *Schema Transformation Without Database Reorganization.* M Tresch, M. Scholl. ACM SIGMOD RECORD 22 (1), 1993.

[Wir86] - *Structured Algebraic Specifications: A Kernel Language.* M. Wirsing. Theoretical Computer Science, 123-249, 1986.