Integrity Maintenance in an Object-Oriented Database

H. V. Jagadish
Xiaolei Qian

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We present an approach for integrating inter-object constraint maintenance seamlessly into an object-oriented database system. We develop a constraint compilation scheme that accepts declarative global specification of constraints, including relational integrity, referential integrity, and uniqueness requirements, and generates an efficient representation that permits localized processing. We demonstrate the feasibility of our approach by designing a constraint pre-processor for O++, the programming language interface to the Ode object-oriented database.

1. INTRODUCTION

By its very definition, a database must serve as a faithful and incorruptible repository of data. Applications that consult the database expect a "warranty" that the database is supplying the correct values. As such, it is not surprising that much attention has been paid to the maintenance of integrity in relational databases. Object-oriented databases are rapidly gaining popularity, and show a promise of supplanting relational databases [15]. It is therefore imperative that we explore the maintenance of integrity in object-oriented databases.

By virtue of object orientation, some integrity constraints are represented naturally and maintained "for free" in an object-oriented database, in that they are directly captured by the type system and the object class hierarchy. Typical examples of this sort are the constraints that every `employee` is a `person` and that every `child` of a `person` is a `person`. Other forms of integrity constraints apply to a single object, and clearly belong as part of an object class specification. An example of such a constraint for a `person` object is that `years-of-schooling` be at least 5 less than `age`. See [11] for a discussion of how such *intra-object* constraints can be integrated into an object-oriented database programming language.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada, 1992

However, by taking the object-centered position, it also becomes unnatural and difficult to represent and maintain many *inter-object* constraints, which apply across objects. For example, we may have a constraint that the `age` of a `person` must be at least 12 greater than the `age` of any `child` of `person`. This constraint compares the `age` attributes of two objects: a person and the person's child. (Actually, it compares pairwise the `age` attribute of a `person` with each of the person's children). When an integrity constraint enforces some relationship across object boundaries in this fashion, it is no longer clear how or where to record such a constraint in an object-oriented system. For instance, even though the constraint above was stated in terms of the age of a person with respect to that of the children, there is a *complementary* constraint on the age of a person with respect to that of the parents. This conflict, between supporting shared access for many applications and facilitating efficient representation and localized processing for a specific application, often results in the redundant representation of different views of the same knowledge in object-oriented databases [26].

In Section 4, we develop a constraint compilation approach to resolve this conflict. We show how inter-object constraints can be stated declaratively once and then integrated with the rest of the object-oriented system by a compiler.

A major motivation for the work described in this paper is that given the flexibility and power of object-oriented systems, it should be possible to capture within the system integrity constraints that traditionally, in a relational system, have not been part of the database itself. At the same time, a key issue in the maintenance of integrity constraints is a careful circumscription of the set of constraints to be verified after each update. The recommended approach in an object-oriented database [11] is to associate constraints with classes, and upon the update of an object to check each constraint associated with its class and none others. The constraint compilation approach we develop here generates efficient representations and localized consistency maintenance, by appropriately transforming a specified declarative constraint and associating it with exactly the relevant set of class definitions, where each of a small number of relevant constructs can efficiently be checked (procedurally). Our methodology encourages reuse, since after schema modification the constraints need only be recompiled – there is no need to respecify them.

A few special cases of inter-object constraints are of particular importance. One is *relational integrity*, or the maintenance of "inverse" pointers. In an object-oriented database, a binary relationship between two objects is not represented as a single tuple in a relation, but rather as a reference, at each of the two objects, to the other object involved in the relationship. If a change is made in one reference (in one object), a corresponding update is usually required in the other reference to maintain integrity. For example, when a man and a woman marry, they record a reference to each other in their respective `wife` and `husband` fields. Observe that two separate updates are required: one in the man object and one in the woman object. Similarly, if they decide to divorce at some later date, an update is once again required in both objects: an update only in one would violate relational integrity. Relational integrity is discussed in depth in Section 5.

The second special case of inter-object constraints is *referential integrity*. This issue has been studied extensively in the context of relational databases and has recently been incorporated in some commercial products. In object-oriented terms, we wish to ensure that a reference to an object in the database is always valid. Referential integrity is studied in Sec. 6.

A third special case of inter-object constraints is *uniqueness*. Often some attribute is required to have a unique value for every object (in a class). Such uniqueness constraints are considered in Sec. 7.

We begin in Section 2 with a quick review of O++, the primary programming language interface to Ode, describing in particular its constraint facilities. We introduce a language in Section 3, CIAO++, which is a small extension of O++, suitable for declaratively expressing integrity constraints. Additional CIAO++ constructs are discussed in Sections 5-7. In Section 8, we discuss how the ideas of Sections 3-7 can be integrated with the Ode object-oriented database and the O++ language [1].

Related Work

A constraint maintenance facility has to answer two types of questions. Given a constraint, (1) when is the constraint violated? (2) how to fix the problem when there is a constraint violation?

To answer the first question, a constraint compilation approach is taken in [7, 12, 13, 19, 22]. State transitions are abstracted into sets of inserted and deleted tuples. Assuming that the constraint is true before a state transition, the objective is to derive an equivalent condition to be checked after the state transition.

When state transitions are specified as transaction programs, constraint maintenance takes the form of verifying that a transaction preserves the truth of a constraint. Various programming logics have been used, such as Hoare Logic [10], Dynamic Logic [4], and Boyer-Moore Logic [23].

Constraints expressed in logical formulas are often very expensive to check. Finite differencing techniques have been used in [3, 16, 20] to transform complex constraint checking to simple data manipulation. A more general constraint reformulation approach is taken in [21], which simplifies constraint formulas using knowledge about database semantics and organization.

To answer the second question, input from database designers is often needed to decide what to do when a constraint is violated. Query modification [25] represents an early attempt to handle this problem, where a state transition is aborted if the constraint is not properly maintained. This approach has evolved into various constraint monitoring schemes that either require database designers to specify maintenance actions as part of the constraint [8], or query database designers interactively to acquire such actions [5, 6, 18]. In [24], transaction compilation rather than transaction execution is aborted if a potential constraint violation is detected.

Some special cases of integrity maintenance in an object-oriented database are discussed in [2, 17]. However, our work represents the first comprehensive approach that combines isolated constraint maintenance techniques, in particular constraint compilation, finite differencing, auxiliary data structures, and monitoring, into an integrated constraint maintenance facility for object-oriented databases. Constraint maintenance in object-oriented databases differ from that in relational databases in three critical aspects, which makes the techniques developed for relational databases not directly applicable to object-oriented paradigm.

Firstly, control in object-oriented databases is localized rather than centralized – there is no centralized place where constraints can be stated, reasoned about, and maintained. Instead, every object is responsible to maintain the constraints attached to it with respect to changes to its attributes. Our constraint compiler is capable of compiling every global constraint into several local constraints attached to different objects, such that, by maintaining the local constraints instead, the global constraint is guaranteed to be valid, and redundant maintenance effort is minimized.

Secondly, the object-oriented model is much richer than the relational model in terms of data modeling constructs. Every modeling construct supports the maintenance of some implicit constraints. By transforming explicit constraints stated by users into implicit constraints embedded in the object-oriented hierarchy, constraint maintenance is more efficient. Our constraint compiler is capable of compiling explicit constraints into auxiliary structures such as new object classes, new attributes, and new object references, such that non-local access is minimized.

Thirdly, the object-oriented model supports the attachment of monitors to individual objects. Our constraint compiler utilizes this feature, together with finite differencing techniques, to compile global corrective actions into local triggers in O++ that efficiently maintain the global constraints.

The approach suggested here is to reduce inter-object constraints into sets of equivalent local constraints, and is exactly the opposite of the approach suggested in [2].

2. OBJECTS IN O++: A BRIEF REVIEW

The O++ object facility is based on the C++ object facility and is called the *class*. *Volatile* objects are allocated in volatile memory and are the same as those created in ordinary programs. *Persistent* objects are allocated in persistent memory and they continue to exist after the program creating them has terminated. Each persistent object is identified by a *unique* identifier, called the object identity [14]. The object identity is referred to as a *pointer to a persistent object*. For example, here is a specification of the classes Dept, Emp, and Mgr:

```
class Dept {
    int budget;
public:
    char dname[20];
    persistent Mgr *head();
      //reference to dept. head object
    persistent Emp *emps[[MAX_EMPS]];
      //double brackets denote unordered set
};

class Emp {
    int salary ;
public:
    Name name;
    char sex;
    int sal() const {return salary; }
    persistent Dept *dept;
    Emp(Name n, char s, int salfig,
       persistent Dept *d);
    void update_salary(int new_salary);
constraint:
   sex == 'F' || sex == 'M' ;
   (sal() >= 10000 || sal() == 0 ):
     update_salary(0) ;
};

class Mgr: public Emp {
    Mgr(Name n, char s, int salfig,
        persistent Dept *d);
};
```

O++ provides facilities for associating *constraints* with an object. These are specified as part of a class definition, and are treated as *members* of the class. The specified constraint conditions are checked every time an instance of that class is updated (through a public member function), a new instance is created, or an old instance is removed. If the condition is found to have been violated, the constraint "fires", executing the action part associated with it, if any. After the action part is executed, the constraint is checked again. If it is still not satisfied, then the transaction attempting the update causing the constraint violation is aborted, and all its updates undone. The order in which constraint conditions are checked and actions executed is implementation dependent, but repeatable.

The syntax for specifying constraints in O++ is:

```
constraint:
   constraint_condition1 [: action1] ;
   constraint_condition2 [: action2] ;
   ...
```

For instance, there are two constraints in the definition of class Emp. The first specifies that the sex field, which we know is of type char, should have a value exactly one of F and M. The second constraint also has an action part, and specifies that the salary of an employee should be recorded as zero, if it does not meet the minimum requirement of 10000. Observe that the constraint condition is satisfied once the salary has been recorded as zero.

The constraint facility provided in O++ is *intra-object* in that when an object is updated only the constraints associated with it, through its class definition, are checked. This restriction has been placed for reasons of efficiency, as well as in accordance with the spirit of localized processing of object-oriented programming. It is not practical to check every constraint with every object, every time that any update is made in the system. Note that there is no restriction on referencing (or even modifying) other objects in the condition or action part of a constraint.

Constraints can be *hard* or *soft*. Hard constraints are checked as soon as the object is updated, and must be satisfied immediately. Soft constraint checking is deferred until the end of the transaction causing the update. Inter-object constraints almost always must be soft since the constraint may be violated after one object has been updated, but before the other has. In this paper, we shall assume that all constraints are soft – some of these may later be hardened, as an optimization.

Transactions in O++ have the form

```
trans {
    ...
}
```

Transactions are aborted using the tabort statement. The macro old(X) can be used within a transaction, to return the value of X at the beginning of the transaction, where X is any persistent object. Similarly, the macro changed(X) returns TRUE if X has been modified from old(X) within the course of the current transaction, and returns FALSE otherwise.

3. LANGUAGE DESIGN

C++ is a procedural language. O++, being based on C++, is also a procedural language, except for the introduction of a set facility, and declarative intra-object constraints and triggers. However, O++ does not provide a declarative mechanism, for instance, to express a constraint of the form: "there exists p in set S such that $e(p)$", for some logic expression e. An explicit temporary variable is required that "collects" the evaluation of the expression for each element of the set. An O++ routine to evaluate this condition may be:

```
{
   cond = FALSE ;
   for (p in S) {
      cond = cond || e(p) ;
   }
   return cond ;
}
```

We believe that there is a value in having a cleaner and more declarative expression of constraints, both in terms of a user understanding code that has been written, and in terms of a compilation process that has to recognize particular constructs to be able to apply the transformation procedure or any of the optimizations discussed below. To this end, we define a language CIAO++ (short for "Constraints In An O++ program"). CIAO++ is a (minor) extension of O++, just as O++ is an extension of C++. In fact, CIAO++ programs "look" exactly like O++ programs, except that more powerful and declarative constraint specification facilities are available to the user. A simple "compiler" accepts CIAO++ code and emits O++ code. See Sec. 8 for an overview of this compiler. In this and the next three sections, we describe CIAO++ constructs as we go along.

The primary new functionality required is the ability to identify the two types of quantifiers. We do this by means of the keywords `foreach` and `thereis`, standing for universal and existential quantification respectively. All C++ logic expressions are also O++ and CIAO++ logic expressions. A simple BNF for CIAO++ logic expressions is as follows:

CIAO++_log_exp := *C++_log_exp* |
  `foreach` *variable* `in` *set* (*CIAO++_log_exp*) |
  `thereis` *variable* `in` *set* (*CIAO++_log_exp*)

It is easy to see that the set of logic expressions that can be defined using CIAO++ is exactly the set of range-restricted prenex formulas[1]. A couple of examples are given below, with regard to the classes `Emp`, `Dept`, and `Mgr` defined in the previous section.

```
foreach d in Dept (thereis e in d->emps[[]]
  (e->sal() > d->head->sal()/2))
```

In words, the constraint above (we shall call it constraint A) says that in each department there is at least one employee whose salary is more than half the department head's salary. We now specify another constraint, called constraint B, to the effect that there is at least one department in which each employee's salary is more than one half the manager's salary. This is written:

```
thereis d in Dept (foreach e in d->emps[[]]
  (e->sal() > d->head->sal()/2))
```

In Ode, constraints have action parts associated with them, to be executed if the constraint condition is violated. It is sometimes convenient to refer to the quantified variable(s) in the action part as well. We permit this in CIAO++, with respect to universally quantified variables. The action part is executed for each instantiation of the universally quantified variable for which the constraint condition is violated. Thus, we could fix a violation of constraint A, for instance, by lowering the salary of the department head. We would write this:

```
foreach d in Dept (thereis e in d->emps[[]]
  (e->sal() > d->head->sal()/2)):
    lower_salary (d->head) ;
```

A central principle of CIAO++ is that inter-object constraints can be associated with any of the objects that participate in the constraints, or even specified separately in a distinguished constraint specification file. The equivalent O++ program will have this constraint divided into an equivalent set of intra-object constraints, one constraint being associated with each relevant class definition.

1. A range-restricted prenex formula has the form $(Q_1 x_1 \in S_1)...(Q_n x_n \in S_n)\, e$, where Q_i is either $\forall$ or $\exists$, and e is quantifier-free. For every range-restricted first-order formula there exists an equivalent range-restricted prenex formula.

In O++, constraints, like other members of a class, are permitted to reference `private` members of the specific object they are associated with. In CIAO++, an inter-object constraint, even if physically incorporated into a class definition, is not a member of the class, and is not permitted to reference private or protected members. Its association with the class is merely a notational convenience.

In addition to the general declarative inter-object constraint construct, CIAO++ also offers convenient short-hand facilities for describing relational integrity, referential integrity, and uniqueness. A description of these facilities is deferred until Sections 5, 6, and 7 respectively. First, we develop a theory of inter-object constraint maintenance.

4. CONSTRAINT COMPILATION

Inter-object constraints are expressed in CIAO++ as described in the previous section. Our task is to implement each inter-object constraint as an equivalent set of (intra-object) constraints to be associated with the appropriate class definitions, that need be checked only when an object of that class is updated, created, or deleted. Clearly, it is sufficient, though unnecessarily profligate, to associate each such inter-object constraint with every class definition. On the other hand, it may not be sufficient to associate an inter-object constraint only with the classes mentioned explicitly in its quantification, because an object mentioned in the constraint may refer to objects in other classes, and changes to the referenced objects could violate the constraint.

In the first subsection below we develop a transformation technique that correctly associates an inter-object constraint with the appropriate classes. The following subsections present useful optimizations.

4.1 Identifying Object References

We distinguish two kinds of object references in an inter-object constraint: those appearing explicitly in the constraint expression, and those appearing implicitly in user-defined functions that are called within the constraint expression. A *reference expression* has the form *e.a*, where *e* is an expression that evaluates to an object, and *a* is an attribute name. A reference expression *e.a* is *primitive* if *e* itself is not a reference expression.

4.1.1 Explicit References

The first stage, in correctly associating an inter-object constraint with the appropriate classes, is to identify all the classes mentioned in explicit object references. To do this, we transform an inter-object constraint into a logically equivalent one such that all objects referenced explicitly in the constraint expression are "brought to attention" in the quantification. The transformation is defined as follows.

Let $(Q \cdots)^*$ denote a sequence of zero or more quantifications. Every inter-object constraint has the form $(Qo_1 \in S_1) \cdots (Qo_n \in S_n)\, e(o_1, \ldots, o_n)$, where Q is either $\forall$ or $\exists$, $n > 0$, e is a (quantifier-free) boolean expression, $o_1, \ldots, o_m$ are all the variables occurring in e, and $S_1, \ldots, S_n$ are set-valued expressions. For each constraint of this form, we first transform it into

$(Qo_1 \in S_1) \cdots (Qo_n \in S_n)(true \rightarrow e(o_1, \ldots, o_n))$.
(The symbol "$\rightarrow$" stands for logical implication, and is equivalent to writing $\neg P \vee R$).

Then the following transformation step is applied repeatedly until no more application is possible. Suppose the constraint has the form $(Qo_1 \in S_1) \cdots (Qo_m \in S_m)(Q \cdots)^*(P \rightarrow R)$ for some $m \leq n$, and there is a non-primitive reference expression $(e.a_1).a_2$ in R such that $e.a_1$ is primitive. The result of a single transformation step with respect to $e.a_1$ is $(Qo_1 \in S_1) \cdots (Qo_m \in S_m)(\forall o \in S)(Q \cdots)^*[e.a_1/o]$ $(P \wedge o = e.a_1 \rightarrow R[e.a_1/o])$, where S is the class of $e.a_1$, o is a fresh variable not occurring anywhere in the constraint before the transformation step, and $(Q \cdots)^*[e.a_1/o], R[e.a_1/o]$ denote the expressions obtained from $(Q \cdots)^* \cdot R$ respectively where all sub-expressions of the form $e.a_1$ are replaced by object reference o.

Lemma 1:

The transformation process applied to inter-object constraint $(Qo_1 \in S_1) \cdots (Qo_n \in S_n) e(o_1, \ldots, o_n)$ produces a constraint that evaluates to true iff the original constraint does.

Proof:

We prove by induction on the chain of transformation steps. The base case is obvious. Let the constraint before and after the n^{th} transformation step respectively be:

$$(Qo_1 \in S_1) \cdots (Qo_m \in S_m)(Q \cdots)^*(P \rightarrow R) \quad (1)$$

$$(Qo_1 \in S_1) \cdots (Qo_m \in S_m)(\forall o \in S)(Q \cdots)^* [e.a_1/o](P \wedge o = e.a_1 \rightarrow R[e.a_1/o]) \quad (2)$$

and P does not contain any reference expression occurring in R. By the induction hypothesis, (1) evaluates true iff the original constraint does. Notice that $e.a_1$ must be non-null (else we cannot evaluate $e.a_1.a_2$). That is, $(\exists o \in S)(o = e.a_1)$ is true. Hence (1) is equivalent to

$$(Qo_1 \in S_1) \cdots (Qo_m \in S_m)((\exists o \in S)(o = e.a_1) \rightarrow (Q \cdots)^*(P \rightarrow R))$$

which in turn is equivalent to

$$(Qo_1 \in S_1) \cdots (Qo_m \in S_m)(\forall o \in S)(o = e.a_1 \rightarrow (Q \cdots)^*(P \rightarrow R)[e.a_1/o])$$

Since e does not contain reference to any variables quantified in $(Q \cdots)^*$, and P cannot contain reference expression $e.a_1$, the above formula is equivalent to (2). □

This transformation captures all explicit object references by identifying each with an (additional) universal quantifier. There are no non-primitive reference expressions in the new constraint expression, and only non-primitive reference expressions could possibly denote object dereference.

We call the form of a constraint obtained after the above transformation process, its *canonical representation*. The canonical representation of a constraint must be associated with every object class that is quantified in it either universally or existentially.

4.1.2 Implicit References

Next, we need to identify all the implicit object references in user-defined functions called (directly or indirectly) in the constraint expression. We cannot expect the same transformation method to work here for several reasons. First, we cannot always expand function calls by inline code due to the existence of recursive functions. Second, reference expressions in user-defined functions might involve local variables, which are meaningless outside the function context.

Instead in the case of implicit object references we simply associate the constraint with the class of each of them. Assuming that no user-defined predicate functions are compiled separately, this step captures all implicit object references, provided it is applied recursively through all function definitions encountered. Any separately compiled (or library) functions must advertise what classes they refer to, and we associate the constraint with each of these classes. The above discussion leads to the following theorem.

Theorem 1

To guarantee the validity of a constraint, it is sufficient to associate its canonical representation with every class that either is quantified over in the canonical representation, or is the class of an implicit non-primitive reference expression.

For example, the two constraints shown in section 3.2 are transformed as follows. The canonical representation of constraint A is:

```
foreach d in Dept (foreach m in Mgr
  (thereis e in d->emps[[]]
    (!(m == d->head) || e->sal() > m->sal()/2)))
```

and of constraint B is:

```
thereis d in Dept (foreach m in Mgr
  (foreach e in d->emps[[]]
    (!(m == d->head) || e->sal() > m->sal()/2)))
```

Both constraints must be attached to all three classes, `Dept`, `Emp`, and `Mgr`.

While the canonical representations of these constraints are useful in correctly identifying the classes with which the constraint must be associated, more efficient representations of the constraints are clearly possible. In fact, no human programmer, having identified the classes with which to associate these constraints, would proceed to specify them in the baroque form of the canonical representation. In the next several sub-sections, we present optimizations that may be applied to the canonical representation of a constraint *after it has been instantiated in a class definition*. The attempt is to obtain (constraint specification) code from the constraint compiler that is of quality comparable to what a human programmer could have produced, were the human to determine all the classes a constraint should be associated with and write each instantiation of the constraint by hand.

4.2 The One-Copy Property

It is possible that there is more than one range variable quantified over the same class after the transformations of the previous step. In that case, only one copy of the constraint need be associated with the class definition. There is no value in repeating the same constraint multiple times. We'll refer to this as the *one-copy property*.

Note that the one-copy property arises from the fact that a constraint in its canonical representation is perfectly symmetric with respect to all the quantified variables that participate in it, in the sense that the two or more constraints being associated with an object class due to two different quantified variables or implicit reference expressions are logically equivalent (in fact they are identical). If some transformation, such as most of the optimizations presented below, loses this symmetry, then the one-copy property no longer holds.

4.3 Optimization via Specialization

Inter-object constraints are in general very expensive to maintain, and the more quantifier nestings there are in a constraint, the more expensive it is to check its validity. For the class of inter-object constraints of the form $(\forall o \in S)e(o)$, the constraint can be *specialized* with respect to the object to which it is attached. Specifically, the constraint associated with class S could be: $e[o/\texttt{this}]$, where `this` refers to the current object (that is being changed). (Recall that the notation a/b is meant to denote the replacement of a by b). Thus the cost of maintaining the constraint is reduced by $|S|$ times because the universal quantification over S is removed[2]. This simplification is correct because the constraint is checked for validity whenever some object of S is changed. Assuming that the database is valid before the change, other objects in S, which have remained the same, do not have to be checked against this change. This leads us to the following theorem:

Theorem 2

With respect to changes in an object o' of class S, if the database is valid before the state transition, then the original constraint $(\forall o \in S)e(o)$ is valid iff the simplified constraint $e[o/o']$ is valid, after the state transition.

For example, constraint A, when attached to the `Dept` class, is specialized into the formula (A1):

```
foreach m in Mgr (thereis e in emps[[]]
  (!(m == head) || e->sal() > m->sal()/2))
```

As stated above, only the outermost universal quantifier may be removed by means of this optimization. However, we know that universal quantifiers commute. So the general rule is to take the copy of the constraint associated with some class and to see if, by commuting quantifiers, it can be written in a form with the universal quantifier over this class being outermost. If so, this quantifier can be eliminated, as discussed above.

For instance, the same constraint A, when attached to the `Mgr` class, is specialized into the formula below by commuting the universal quantifiers on classes `Dept` and `Mgr` (A2):

```
foreach d in Dept (thereis e in d->emps[[]]
  (!(this == d->head) || e->sal() > sal()/2))
```

2. We use $|S|$ to denote the *cardinality* of a set S

Once a universal quantifier is eliminated using this optimization, the constraint is no longer symmetric, and therefore the one-copy property no longer holds. To see why, let us suppose the constraint is $(\forall o_1 \in S_1)(\forall o_2 \in S_2)e(o_1, o_2)$, and $S_1 = S_2$. The constraints to be associated with S_1 and S_2 (they happen to be the same) are: $(\forall o_2 \in S_2)e(\texttt{this}, o_2)$ and $(\forall o_1 \in S_1)e(o_1, \texttt{this})$ respectively, which are not equivalent in general. Given an inter-object constraint, if $S_1, \cdots, S_m$ are all the classes mentioned in the quantification that are equal to S and are not nested in any existential quantification, then the cost ratio between checking the one unsimplified constraint and checking the m simplified constraints would be $\dfrac{\prod_{i=1}^{m} |S_i|}{\sum_{i=1}^{m} \prod_{j \neq i} |S_j|}$, which is equivalent to $\dfrac{|S|}{m}$. The decision on whether to use the simplified version depends on whether this cost ratio is greater than one, given that size information about object classes is available. Typically, one expects $|S| >> m$, and it is worthwhile to use the simplified version.

4.4 Optimization via Variable Binding

The transformation procedure applied to capture all the classes with which a constraint is to be associated introduces universal quantifiers, as we saw above. Once the transformed constraint has been associated with appropriate classes, it is often possible to "undo" some of the transformation individually for each instantiation of the constraint in a class. By this means, the extra universal quantification can often be eliminated altogether. To be more specific, if a constraint associated with some particular class has the form $(Q_1 \cdots)^*(\forall o \in S)(Q_2 \cdots)^*(o = e \wedge P \rightarrow R)$ and $o = e$ does not contain any variables quantified in $(Q_2 \cdots)^*$, then it can be transformed into the equivalent form $(Q_1 \cdots)^*(Q_2 \cdots)^*(P \rightarrow R)[o/e]$. The correctness of this optimization is guaranteed by Lemma 1.

For instance, constraint A, when associated with class `Dept`, can be simplified further after the optimization via specialization shown in formula A2, and written as follows:

```
thereis e in emps[[]]
  (e->sal() > head->sal()/2)
```

Constraint B, when associated with class `Dept`, can be simplified to:

```
thereis d in Dept (foreach e in d->emps[[]]
  (e->sal() > d->head->sal()/2))
```

This simplified form is exactly the same as the original specification, so it may appear that all our work thus far has been superfluous. Note, however, that this simplification is not possible in the case of constraint A associated with class `Mgr`. The transformation technique permitted us to identify these classes, and to associate the correct constraint with all of them.

4.5 Optimization via Redundant Data

For another class of inter-object constraints of the form $(\exists o \in S)e(o)$, the efficiency of maintaining its validity may be improved by maintaining redundant data. We create a variable

S' whose type is $set(S)$, and an object $o' \in S$ is a member of S' if and only if $e(o')$ is true. We can replace the original constraint by two constraints: $(\exists o \in S)(o \in S')$ and $(\forall o \in S)(o \in S' \Leftrightarrow e(o))$. The action part of the latter is such that o is inserted into (deleted from) S' whenever $e(o)$ becomes true (false). Given that S' is initialized appropriately, the action part correctly maintains the validity of the second constraint. We thus have the following theorem:

Theorem 3

The original constraint $(\exists o \in S)e(o)$ is true iff the two constraints $(\exists o \in S')$ and $(\forall o \in S)(o \in S' \Leftrightarrow e(o))$ are both true for some $S' \subseteq S$.

Any class to be associated with the original constraint is instead associated with the second constraint above. S is associated in addition with the first constraint above. Other optimizations can be applied to these constraints. In particular, the optimization of Section 4.3 is often applicable to the the second constraint associated with S.

For example, constraint B can be simplified by the introduction of a class `Set_of_Dept`, and an object, `const_B_Dept`, instance of this class. Each object of this new class represents a set of (references to) departments[3]. With class `Dept` we associate the constraints:

```
thereis d in Dept (d in const_B_dept) ;
foreach e in emps[[]]
  !(e->sal() > head->sal()/2) :
    const_B_dept += this ;
!(foreach e in emps[[]]
  !(e->sal() > head->sal()/2)) :
    const_B_dept -= this ;
```

Constraint B associated with other classes, must also be transformed for this to work. For instance, with class Emp, we must write (recall that the `d` in the action part refers to the specific departments for which the given constraint condition is violated):

```
foreach d in Dept (foreach e in d->emps[[]]
  !(e->sal() > d->head->sal()/2)):
    const_B_dept += d ;
foreach d in Dept !(foreach e in d->emps[[]]
  !(e->sal() > d->head->sal()/2))):
    const_B_dept -= d ;
```

With the original constraint, there is no integrity maintenance overhead to create objects in the existentially quantified class that we try to remove, but it costs $|S|$ to delete an object in S. With this optimization, the overhead when an object is created in S would be the same as the cost of evaluating e to check if it is also in S', while deleting an object in S takes constant time to check if set S' is non-empty. Let $|P|$ be the cost of evaluating the (quantified) logic formula to determine membership in set S'. Usually $|P| >> 1$ since evaluating the condition may involve iterating over other sets. Let $|S|$, $|S'|$ be the sizes of the sets S and S'. Clearly, $|S'| \leq |S|$. The total cost without the optimization for x insertions and y deletions is $y * |S| * |P|$. The total cost with the optimization is $x * |P| + y * |S'|$. So, statistically, this optimization is of value when $x * |P| + y * |S'| < y * |S| * |P|$. This is certainly the case when $x * |P| + y * |S| < y * |S| * |P|$. Since $|P| >> 1$, the above inequality holds when $x < y * |S|$. The cost of maintaining the constraint with respect to changes in other classes is not affected. Therefore, this optimization is likely to be of value unless the expected number of insertions is greater than the expected number of deletions by a large factor.

3. Operator += has the semantics "add to set if not already an element", and operator -= has the semantics "if an element of the set, remove".

5. RELATIONAL INTEGRITY

5.1 Basics

Consider a binary relationship that is known at schema definition time. In a relational database, it would be stored as a table with two columns, each column holding a foreign key representing one of the participants in the relationship. In an object-oriented database, this relationship (assuming it is known at schema definition time) is stored as a directional reference (or set of references) from either participant in the relationship to the other.

When such a relationship is to be updated, multiple updates have to be performed, one for each participant in the relationship, giving rise to the possibility that the relation is recorded differently at the different logical locations. *Relational integrity* in an object-oriented database is the proper maintenance of relationships recorded at multiple logical locations, ensuring that the recording is consistent.

For example consider a "husband-wife" relationship. In a relational database, this would be stored in a table with each couple stored as a tuple, with the husband key (name or other identifier) in one column, and the wife key in the other column. In an object-oriented database, corresponding to this tuple, the husband object would have the wife's id recorded in the wife attribute, and the wife object would have the husband's id recorded in the husband attribute. Relational integrity ensures that if A records B as his wife, then B records A as her husband and vice versa.

There is no way to express an n-ary relationship directly in an object-oriented model. There is also no need, since any n-ary relationship can be expressed as a set of n binary relationships, one between each participant in the original relationship and a special "relationship" object. As such, we shall focus on binary relationships.

Whenever, for some objects a and b, and some directional relationship R, we say $a\ R\ b$, we also imply that $b\ R^{-1}\ a$ holds. R^{-1} is called the inverse of R. For example if R is the relationship "manager of", then R^{-1} is the relationship "managed by". We know that if a is the manager of b, then b is managed by a, and vice versa. Observe that $R = (R^{-1})^{-1}$. In the example class definitions we have been using, this relation is between the attribute `dept` in class `Mgr` and the attribute `head` in class `Dept`.

5.2 CIAO++ Constructs

We provide a facility for declaring the inverse of an attribute in O++ as follows:

type attr `inverse` *inv-attr* ;

Thus, wherever an attribute is being declared in the definition of some class *C*, its inverse can also be specified. For such an inverse specification to be meaningful, the *type* must be of the form "*class-name* `*`". The attribute itself may be an individual value, a set, or an array. The class *class-name* must have an attribute named *inv-attr*. *inv-attr* may be an individual value, a set, or an array, but in all cases its type must be a reference to the current class. The declaration above is understood to mean that *C.attr* and *class-name.inv-attr* are inverses of each other. That is, each attribute is a directional representation of the same relation, (pointing to the other object), with the two attributes expressing the relation in different directions[4].

Thus far, our attention has been focussed solely on the condition part of an inter-object constraint. The action part has not been paid much attention, and has been specified just as in O++. Here, for the first time, we would like to have a convenient shorthand notation for different action possibilities. We handle these by introducing the keywords `ripple` and `abort`, meaning respectively that the action is to fix the reverse pointer and that the action is to abort the transaction.

A declaration of inverse is required in the definition of only one of the two attributes involved in the inverse relationship. That is, if A declares B as its inverse, B need not declare an inverse, but if it does, that is fine too, as long as the inverse it declares is A and not anything else. By default, the action part of one applies to the other as well. However, it is permissible to have two different action policies for the two directions. See discussion in section 5.3.

A few sample inverse declarations are given below:

```
class Dept {
  ...
  Mgr*  head()          inverse dept abort ;
  Emp*  emps[[50]]      inverse dept ripple ;
  ...
}

class Emp {
  ...
  Dept* dept            inverse emps[[]] abort ;
  Emp*  mentees[[10]]   inverse mentors ripple ;
  Emp*  mentors[[2]] ;
  Emp*  officemates[[4]]
               inverse officemates[[]] abort ;
  ...
}
```

The first inverse declaration above relates a manager and the department he or she heads. Notice that `head()` is a computed attribute. The second inverse declaration relates employees to the department they work in. This is a many-one relation. The third inverse declaration (the first one in `Emp`)) is the complement of the previous declaration. Only one of these is required. However, the actions specified are different in the two directions. If a `Dept` object modifies its set of `emps[[]]`, then a corresponding modification is automatically made to the `dept` attribute of each employee affected, as part of constraint maintenance. On the other hand, an `Emp` object is not permitted to change its `dept` attribute unilaterally: an attempt to do so will cause the transaction to abort.

The next inverse declaration is with regard to a many-many relationship between mentors and mentees. Observe that the inverse was declared only with one of the two attributes involved – `mentees`. There is no need to declare an inverse with `mentors` as well. Finally, we declare `officemates` to be an inverse of itself; that is, if *a* records *b* as an officemate then *b* must record *a* as an officemate as well.[5]

5.3 Maintaining Relational Integrity

Inverse declaration in CIAO++ is a request to maintain relational integrity. In general, each relational integrity constraint is expressible as a pair of constraints in the canonical form of the previous section, with both constraints in the pair quantified identically. Consequently, the quantifier-free logic expressions can be combined to form a single conjunctive expression. For instance, the second relational integrity constraint in the example above is expressed by the pair of constraints:

```
foreach e in Emp (foreach d in Dept
  ((d != e->dept) || (e in d->emps[[]])))
foreach e in Emp (foreach d in Dept
  (!(e in d->emps[[]]) || (d == e->dept)))
```

These can be combined and written:

```
foreach e in Emp (foreach d in Dept
  (((d != e->dept) || (e in d->emps[[]])) &&
    (!(e in d->emps[[]]) || (d == e->dept))))
```

or equivalently,

```
foreach e in Emp (foreach d in Dept
  (((d != e->dept) && !(e in d->emps[[]])) ||
   ((d == e->dept) && (e in d->emps[[]]))))
```

Once the relational integrity constraint is placed in this form, all the optimizations discussed in the preceding section can be applied.

If the action associated with a relational integrity constraint is to abort the offending transaction, then no special action part need be written in O++. However, if the action associated is ripple, then an action specifying this ripple must be written. This action is different in the two classes involved. The

4. Our facility for declaring inverses is similar to the `inverse_member` facility in ObjectStore [17]. The difference is that our facility is not a special ad hoc construction, but rather syntactic sugar on top of the general inter-object constraint facility.

5. Relational integrity does not ensure transitivity. Thus, if *a* records *b* and *c* as officemates, then all we ensure is that *b* and *c* each record *a* as an officemate. However, *b* and *c* need not record each other as officemates. If we wished to enforce such a transitive constraint, we would have to write an additional explicit inter-object constraint:

```
foreach p in officemates[[]]
  (foreach r in p.officemates[[]]
    (r in officemates[[]])) ;
```

example above becomes, when associated with class Emp:

```
!changed(dept) :
  if (old(dept) != NULL)
    old(dept)->emps[[]] -= this ;
  if (dept != NULL)
    dept->emps[[]] += this ;
```

Each time an object of class Emp has its dept attribute modified, the object is removed from the set of employees in the old department and added to the set of employees in the new department. The macro changed returns TRUE whenever the value of its argument has changed in the course of the current transaction. The macro old returns the value of its argument at the beginning of the current transaction.

A similar constraint is required with the class Dept. Since each of these constraints independently specify the action to be taken when the inverse relationship is modified at one end, no additional complications are caused by having the action be abort in one direction and ripple in the other.

5.4 Optimization via Inverse

By virtue of two attributes a_1 of class S_1 and a_2 of class S_2 being inverses, representing a binary one-one relationship, several equivalent assertions are true:

$$(\forall o_1 \in S_1)(o_1.a_1 = NULL \vee o_1.a_1.a_2 = o_1)$$
$$(\forall o_2 \in S_2)(o_2.a_2 = NULL \vee o_2.a_2.a_1 = o_2)$$
$$(\forall o_1 \in S_1)(\forall o_2 \in S_2)(o_1.a_1 = o_2 \Leftrightarrow o_1 = o_2.a_2)$$

We can derive four simplification rules corresponding to the three assertions above. (A simplification rule of the form $P[e_1] \rightarrow P[e_2]$ says that if we have a constraint formula with some subexpression that matches e_1, then we could replace it by an equivalent formula in which the matched subexpression is replaced by e_2).

$$P[o_1.a_1.a_2] \rightarrow P[o_1]$$
$$P[o_2.a_2.a_1] \rightarrow P[o_2]$$
$$P[o_1.a_1 = o_2] \rightarrow P[o_1 = o_2.a_2]$$
$$P[o_1 = o_2.a_2] \rightarrow P[o_1.a_1 = o_2]$$

The first two rules always result in optimization. The last two rules, while in themselves are usually not optimizations, can often make further optimization possible. For example, the constraint A attached to the Mgr class (as shown in formula A2) is:

```
foreach d in Dept (thereis e in d->emps[[]]
  (!("this" == d->head) || e->sal() > sal()/2))
```

If we know that the dept attribute of the Mgr class is the inverse of the head attribute of the Dept class, then the above constraint could be simplified to:

```
thereis e in dept->emps[[]] (e->sal() > sal()/2)
```

For inverse attributes a_1 of class S_1 and a_2 of class S_2 that represent some binary many-one or many-many relationships, the corresponding attributes a_1 and/or a_2 are set-valued. The first two simplification rules do not apply. The other two rules are derived from an appropriate modification of the third assertion. For instance, if a_1 is single-valued but a_2 is set-valued, the equivalent assertion is

$$(\forall o_1 \in S_1)(\forall o_2 \in S_2)(o_1.a_1 = o_2 \Leftrightarrow o_1 \in o_2.a_2)$$

and the applicable simplification rules are

$$P[o_1.a_1 = o_2] \rightarrow P[o_1 \in o_2.a_2]$$
$$P[o_1 \in o_2.a_2] \rightarrow P[o_1.a_1 = o_2]$$

5.5 Optimization via Relational Constraints

Inter-object constraints of the form $(\forall o_1 \in S_1)(\exists o_2 \in S_2)e(o_1, o_2)$, can be transformed with the help of redundant data, like the constraints discussed in Sec. 4.5, provided that relational integrity is being maintained.

For object class S_1 and S_2, add new attributes a_1 and a_2 respectively, which are of type $set(S_2)$ and $set(S_1)$. For each object o_1 of class S_1, $o_1.a_1$ denotes the set of objects o_2 of class S_2 that makes $e(o_1, o_2)$ true. Similarly, for each object o_2 of class S_2, $o_2.a_2$ denotes the set of objects o_1 of class S_1 that makes $e(o_1, o_2)$ true.

Associated with classes S_1 and S_2 respectively are two new constraints. One says that for every object o_1 of S_1, $o_1.a_1$ is computed as the set: $\{o_2 | o_2 \in S_2 \wedge e(o_1, o_2)\}$. Another says that for every object o_2 of S_2, $o_2.a_2$ is computed as the set: $\{o_1 | o_1 \in S_1 \wedge e(o_1, o_2)\}$. A third intra-object constraint is associated with S_1 which says $\neg(a_1 = NULL)$. Finally, a new relational constraint claims that a_1 and a_2 are inverses of each other.

Theorem 4

The original constraint $(\forall o_1 \in S_1)(\exists o_2 \in S_2)e(o_1, o_2)$ is valid iff the following four new constraints are valid:

$$(\forall o_1 \in S_1)(\forall o_2 \in S_2)(o_1 \in o_2.a_2 \Leftrightarrow e(o_1, o_2)) \quad (1)$$
$$(\forall o_1 \in S_1)(\forall o_2 \in S_2)(o_2 \in o_1.a_1 \Leftrightarrow e(o_1, o_2)) \quad (2)$$
$$(\forall o_1 \in S_1)\neg(o_1.a_1 = NULL) \quad (3)$$
$$(\forall o_1 \in S_1)(\forall o_2 \in S_2)(o_1 \in o_2.a_2 \Leftrightarrow o_2 \in o_1.a_1) \quad (4)$$

Constraints (1) and (3) together are equivalent to the original constraint. and require the maintenance of the computed attribute a_1. Constraints (2) and (4) assist in the maintenance of this computed attribute with the help of relational integrity.

To maintain constraint A this way, we would have a new attribute associated with class Dept, named d_emps[[]], which is the set of those "distinguished employees" in the department who earn more than half of the department head's salary. A corresponding new attribute associated with class Emp, named distinguished, which is a boolean-valued attribute true if and only if the employee is a distinguished member of his department (it is not a set-valued attribute because the relationship between employees and departments is many-to-one). Besides maintaining d_emps[[]] and its inverse distinguished, constraint A ensures that d_emps[[]] is always non-empty.

In the cost table below, we assume that the objects of S_2 that satisfy the relationship $e(o_1, o_2)$ with respect to specific o_1 are uniformly distributed; and the same with objects of S_1. From the table, it is obvious that the more frequently S_2 is deleted from, the more savings our optimization provides. On the other hand, if the creation of objects in S_2 and the deletion of objects in S_1 are more frequent, the original constraint is cheaper to maintain. In particular, consider a sequence of x insertions into and y deletions from S_2, while S_1 remains unchanged. The total cost for this sequence of operations is $x + y|S_1||S_2|$ with the original constraint, and $x|S_1| + y|S_1|/|S_2|$ with the optimization. Assuming $|S_1|$ and $|S_2|$ are both >> 1, the optimized constraint is cheaper to maintain iff $x/y < |S_2|$. In other words, the optimization is to be preferred unless the total number of insertions exceeds the total number of deletions by a factor that is larger than the size of the set itself.

	Before	After
Create S_1	$\|S_2\|$	$\|S_2\|$
Delete S_1	constant	$\|S_2\|/\|S_1\|$
Create S_2	constant	$\|S_1\|$
Delete S_2	$\|S_1\|\times\|S_2\|$	$\|S_1\|/\|S_2\|$

6. REFERENTIAL INTEGRITY

Referential integrity requires that any object referenced by another object actually exist. In an object-oriented system, references are recorded by means of object identifiers. Since the user has no way of generating or modifying object identifiers accidentally, the system can easily guarantee that a reference is valid at the time at which it is recorded. What requires work is to ensure that there are no references left to an object anywhere in the system when it is deleted. This question of maintaining referential integrity at object deletion time is our primary concern in this section.

Suppose that an object to be deleted still has a reference to it. There are three standard maintenance options [9]. The reference can be deleted as part of the transaction deleting the object (by placing a NULL in the reference pointer), the referencing object can be deleted, or the deletion of the object can be disallowed.

Which of the three we want for each reference is specified once as part of the class definition, and is applicable to all instances of the class. This specification is inherited in derived classes. We use the keywords `nullify`, `ripple`, and `abort`, respectively for the three possible actions. We also introduce the keyword `off` to indicate explicitly that referential integrity is not to be maintained. The general statement for referential integrity is of the form:

<attr-decl> `reference` *policy* `;`

and is included in the class definition, one for each reference. The *policy* is one of the four keywords mentioned above. The *<attr-decl>* is the declaration of an attribute in the usual form. The default policy is `off`, if nothing is specified for some attribute. Here is an example:

```
class Dept {
  ...
  Mgr* head()
    inverse dept abort    reference abort ;
  Emp* emps[[50]]
    inverse dept ripple   reference nullify ;
  ...
}
```

The example above states that when a deletion is attempted on a `Mgr` object, the transaction should be aborted if this object is listed as the `head` of some `Dept` object in the database. When a `Emp` object is deleted, any reference to this object from the `Dept` this employee works should be nullified and the deletion allowed to commit.

In general, referential integrity requires that at the time deletion is attempted on an object $\bar{o}$ of class $\bar{S}$, for every class S in the database, for every attribute a of S that is a reference to an object of class $\bar{S}$, we check $\forall o \in S(o.a \neq \bar{o})$. The possible S and a values are known at compile time, and are independent of the size of the database. For each such S and a, one constraint of the form just shown is placed in the definition of class $\bar{S}$, and associated with every instance of the class.

In Section 5.5 we discussed how relational integrity can be used to improve the enforcement of constraints of the form $\forall$ ($\exists$ ()). Since referential integrity has this form, the same technique is applicable here. Suppose we know that $\bar{S}.\bar{a}$ is the inverse of $S.a$. Then the constraint associated with the class $\bar{S}$ simply reduces to checking that $\bar{a}$ is NULL (or the empty set, if it is set valued). This check can be performed cheaply, in constant time. If relational integrity is not maintained, other techniques, such as reference counts, can be used to get rid of the universal quantifier. The key point to note is that referential integrity can be implemented, and optimized, using the general inter-object constraint mechanisms described in Sec. 4.

7. UNIQUENESS

Another special type of inter-object constraint is *uniqueness*, requiring that every object of a certain class have a unique value for some attribute. CIAO++ introduces the keyword `unique`, which can be used when declaring any attribute in a class definition. When applied to a set-valued attribute, it is taken to mean that the corresponding sets are *disjoint*. When applied to an array-valued attribute, it is taken to mean that the corresponding arrays differ in at least one element. These constraints can be written in the canonical inter-object form in a straightforward fashion, and the techniques discussed in the preceding sections used. For instance, we could have:

```
class Dept {
  unique persistent Mgr *head() ;
  unique persistent Emp *emps[[MAX_EMPS]] ;
     ...
}
```

The first constraint above states that no two departments can have the same head (that is, no one manager can simultaneously head two departments), and is equivalent to:

```
foreach m in Dept ( foreach n in Dept
  ((m == n) || (m->head() != n->head())));
```

The second constraint states that no two departments can have any employees in common (that is, each employee works in no more than one department), and can be written as:

```
foreach m in Dept (foreach n in Dept
  (foreach e in m->emps[[]]
    (foreach f in n->emps[[]]
      ((m == n) || (e!=f))))) ;
```

When a uniqueness constraint is violated, the action is assumed to be an abort of the transaction violating the constraint.

As an aside observe that when attributes on which uniqueness is specified have inverses declared, then the maintenance of relational integrity automatically also maintains uniqueness. For instance, if a `Mgr` object has a single-valued `Dept` attribute, then (with relational integrity maintained) no two departments can have the same manager. Similarly, if an `Emp` class has a single-valued `dept` attribute, inverse of the `emps[[]]` attribute in the `Dept` class, then it is not possible for any `Emp` object to be recorded in the `emps[[]]` set of more than one department.

8. IMPLEMENTATION

CIAO++ is accepted by the `ciaofront` preprocessor, which generates equivalent O++ code. The O++ code generated is then compiled with the O++ compiler, comprising `ofront` and the C++ compiler. If O++ is input to `ciaofront`, then it is output with no modification: only the constructs specific to CIAO++ are processed[6].

Since all the new constructs in CIAO++ deal with class definition, only the class definitions need be processed through `ciaofront`. This pre-processor works in two passes. In the first pass, it collects all the constraints and transforms them into their canonical representations. It then ensures that it has available to it all the class definitions that require one or more of these constraints to be included. In the second pass, the transformed constraints are placed in all the appropriate classes, performing optimizations where appropriate. In C++, it is necessary that declaration precede use. In particular, it is not permissible for a member of a class A to be referenced in a class B unless the definition of A precedes that of B. A typical inter-object constraint is likely to involve members of multiple classes, such as A and B, so that we seem to be in trouble whichever we place first. `ciaofront` resolves this problem by encapsulating the condition evaluation in a function. The body of this function is placed after the declarations of all the classes involved.

All the optimizations described in this paper are at the language level, and are incorporated into the CIAO++ language compiler. Lower level optimizations can also be of value, and may work in cases where language level optimizations are not possible. For instance, if objects of class S are indexed on the a attribute, then the constraint $\forall o \in S(o.a \neq \texttt{val})$ translates to $\nexists o \in S(o.a = \texttt{val})$, and can be evaluated by a single index look-up to determine whether there is indeed an object o that has an attribute a with the prescribed value. In other words, due to the existence of the index, the universal quantifier does not render the constraint expensive to maintain. We are currently studying how such "lower-level" optimizations may be incorporated into `ciaofront`.

6. At a future date, we intend to merge `ciaofront` with `ofront` to create a single pre-processor. The eventual goal is to have a single efficient compiler for a stable language, without piping through multiple layers.

By the very nature of inter-object constraints, it is not possible to compile interconnected classes separately. Now suppose that we have a database with an existing class A, already populated with several objects. What happens if one wishes to add a class B to the database, and create inter-object constraints between these two classes? Even if the inter-object constraints are stated with class B, class A also has to be recompiled. The objects existing in the database may not satisfy the new constraint: what do we do about them? These and other such problems compound the already hard question of schema evolution. We are currently studying this problem.

9. CONCLUSIONS

Object-oriented databases pose new challenges to semantic integrity, both in terms of constraint representation, and in terms of constraint maintenance. We have developed a constraint compilation approach that facilitates efficient representation and localized processing on one hand, and ensures global declarative specification and consistency maintenance on the other hand. Constraints are specified declaratively in the shared logical language. We have demonstrated the feasibility of our approach by designing a constraint preprocessor for the Ode object-oriented database system.

Constraint compilation is a kind of knowledge compilation, where the generic constraint knowledge expressed in logic is compiled into object-oriented representations. In general, a knowledge representation scheme always provides constructs that ease the expression of certain types of knowledge, while making the expression of others hard. But no schemes are perfect for every possible application. We believe that our approach properly resolves the conflict between shared generic knowledge specification and localized efficient representation.

A lot of work remains to be done. In particular, we have only scratched the surface of optimization. With the rich semantics of object-oriented paradigm, more optimization techniques can be developed for constraint compilation. We are also working on the implementation of the constraint compiler for Ode. Our work can be generalized to constraint compilation into other kinds of semantic data models, and to knowledge compilation in knowledge-based systems.

Acknowledgements

We are grateful to Narain Gehani for several useful discussions. We are also grateful to Shaul Dar, Narain Gehani, and Dan Lieuwen for a careful reading of a draft of this paper.

REFERENCES

[1] R. Agrawal and N. H. Gehani, "ODE (Object Database and Environment): The Language and the Data Model," *Proc. ACM SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989.

[2] A. Albano, G. Ghelli, and R. Orsini, "A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language," *Proc. of the 17th Int'l Conf. on Very Large Databases*, Barcelona, Spain, September 1991, 565-576.

[3] P. Bernstein, B. Blaustein, and E. Clarke, "Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data," *Proc. 6th Int'l Conf on Very Large Databases*, 1980 , 126-136..

[4] M. Casanova and P. Bernstein, "A Formal System for Reasoning about Programs Accessing a Relational Database," *ACM Transactions on Programming Languages and Systems*, **2**(3), July 1980, 386-414.

[5] M. Casanova, L. Tucherman, and A. Furtado, "Enforcing Inclusion Dependencies and Referential Integrity," *Proc. 14th Int'l Conf. Very Large Data Bases*, 1988, 38-49.

[6] S. Ceri and J. Widom, "Deriving Production Rules for Constraint Maintenance," *Proc. 16th Int'l Conf. Very Large Data Bases*, 1990, 566-577.

[7] U. S. Chakravarthy, J. Grant, J. Minker, and Logic-Based Approach to Semantic Query Optimization, *ACM Trans. on Database Systems*, **15**(2), June 1990, 162-207.

[8] D. Cohen , "Compiling Complex Database Transition Triggers," *Proc. ACM-SIGMOD 1989 Int'l Conf. on Management of Data*, 1989, 225-234..

[9] C. J. Date, "Referential Integrity," *Proc. 7th Int'l Conf. Very Large Data Bases*, 1981.

[10] G. Gardarin and M. Melkanoff, "Proving Consistency of Database Transactions," *Proc. 5th Int'l Conf. Very Large Data Bases*, 1979, 291-298.

[11] N. H. Gehani and H. V. Jagadish, "Ode as an Active Database: Constraints and Triggers," *Proc. 17th Int'l Conf. Very Large Data Bases*, Barcelona, Spain, 1991, 327-336.

[12] L. Henschen, W. McCune, and S. Naqvi, "Compiling Constraint-Checking Programs from First-Order Formulas," in *Advances in Database Theory, Vol.2*, , H. Gallaire, J. Minker, and J-M. Nicolas (ed.), Plenum Press, 1984, 145-170.

[13] A. Hsu and T. Imielinski, "Integrity Checking for Multiple Updates," *Proc. ACM-SIGMOD 1985 Int'l Conf. on Management of Data*, 1985, 152-168.

[14] S. N. Khoshafian, G. P. Copeland, and 406-416, "Object Identity," *Proc. 1st Int'l Conf. Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, Sept. 1986.

[15] W. Kim, "OBJECT-Oriented Databases: Definition and Research Directions," *IEEE Transactions on Knowledge and Data Engineering*, June 1990, 327-341.

[16] S. Koenig and R. Paige, "A Transformational Framework for the Automatic Control of Derived Data," *Proc. 7th Intl. Conf. on Very Large Data Bases*, 1981, 306-318.

[17] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System," *Communications of the ACM*, **34**(10), October 1991, 51-63.

[18] G. Moerkotte and P. C. Lockemann, "Reactive Consistency Control in Deductive Databases," *ACM Trans. on Database Systems*, **16**(4), December 1991, 670-702.

[19] J-M. Nicolas, "Logic for Improving Integrity Checking in Relational Data Bases," *Acta Informatica* , **18**, 1982, 227-253.

[20] R. Paige, "Applications of Finite Differencing to Database Integrity Control and Query/Transaction Optimization," in *Advances in Database Theory, Vol.2*, H. Gallaire, J. Minker, and J-M. Nicolas (ed.), Plenum Press, 1984, 171-209..

[21] X. Qian and D. Smith, "Integrity Constraint Reformulation for Efficient Validation," *Proc. 13th Int'l Conf. Very Large Data Bases*, 1987, 417-425.

[22] X. Qian, "An Effective Method for Integrity Constraint Simplification," *Proc. IEEE 4th Int'l Conf. Data Engineering*, 1988, 338-345.

[23] T. Sheard and D. Stemple, "Automatic Verification of Database Transaction Safety," *ACM Transactions on Database Systems*, **14**(3), September 1989, 322-368.

[24] D. Stemple, S. Mazumdar, and T. Sheard, "On the Modes and Meaning of Feedback to Transaction Designers," *Proc. ACM-SIGMOD 1987 Int'l Conf. on Management of Data*, 1987, 374-386.

[25] M. R. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification," *Proc. ACM-SIGMOD 1975 Int'l Conf. on Management of Data*, 1975, 65-78.

[26] G. Wiederhold, "Views, Objects, and Databases," *IEEE Computer* , **19**(12), December 1986, 37-44.