Ode as an Active Database: Constraints and Triggers

N. H. Gehani
H. V. Jagadish

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

Ode [2, 3] is a database system and environment based on the object paradigm. The database is defined, queried, and manipulated using the database programming language O++, which is an upward-compatible extension of the object-oriented programming language C++ [Stroustrup 1986]. O++ extends C++ by providing facilities suitable for database applications, such as facilities for creating persistent and versioned objects, defining and manipulating sets, organizing persistent objects into clusters, iterating over clusters of persistent objects, and associating constraints and triggers with objects.

The constraint and trigger facilities in Ode make Ode an active database. Providing integrity constraint facilities in a database is not a new issue since all major commercial databases today provide some level of integrity maintenance. The novel aspect of our work is in providing facilities for object-oriented databases that can be used to specify complex and higher-level integrity constraints. We want to prevent updates to objects that are acceptable from the purely mechanistic concurrency control, serializability perspective, but that leave the database in an inconsistent state. Constraints are used to ensure this kind of data integrity [6]; they go beyond the consistency supported by the type system of a database programming language. Ode provides facilities for constraints that can ensure object consistency on an object update basis (hard constraints) and on a transaction basis (soft constraints). Specified procedures may also be executed in an attempt to ‘‘fix’’ constraints that are not satisfied.

In traditional systems, software components that update objects must themselves recognize the consequences of the update and then perform appropriate actions. This approach compromises software modularity since the updating component is logically performing the task of another software component that should respond to the update. Consider, for example, a radar scanning program that updates enemy aircraft positions. Modularity is compromised if this program must also assume the responsibility of initiating responses (such as firing a missile) to threatening situations (such as an aircraft coming too close). Modularity can be maintained by having a separate software module that periodically examines (‘‘polls’’) the objects of interest and takes appropriate actions when these objects change in specified ways. Thus a missile-firing program may periodically examine the changes to objects updated by the radar scanning program (and any other programs that indicate enemy aircraft position) to determine if any action has to be taken. However, the problem with the ‘‘polling’’ approach is that it wastes resources. Moreover, the response time depends upon the polling period. Immediate responses may require very small polling periods which will increase resource wastage.

Another way of looking at the problem with conventional passive databases is that applications must themselves check object states and initiate actions if appropriate conditions are satisfied. Active databases, on the other hand, provide a data-centric rather than a program-centric view of the world. This simplifies the writing of applications. In an active database, a database operation or an ‘‘external’’ event can, provided the specified conditions are satisfied, automatically trigger the execution of specified actions (including database operations). Applications can update the database without worrying about actions that must be taken when the objects change in a specified way. Applications can also be notified (or invoked) automatically when objects change in a specified way. Active databases are of use in a variety of application areas such as integrated manufacturing, power distribution network management, and air-traffic control. Many databases now support triggers, e.g., POSTGRES [24], HiPAC [11], Sybase [10], Vbase [4], and OOPS [22].

Ode supports three kinds of triggers: once-only, perpetual, and timed triggers. Triggers, like constraints, are associated with objects. However, they are parameterized, and can have multiple invocations active at the same time.

In this paper, after a quick introduction to O++ in Section 2, we state our design goals in providing trigger and constraint facilities for an object-oriented database in Section 3. We then describe the constraint and trigger facilities in Ode and illustrate their use in Sections 4 through 6. We also discuss issues related to constraints and triggers such as intra-object versus inter-object constraints and triggers, and referential integrity. In Section 7 we give details about the implementation of Ode triggers and constraints. Finally, in Section 8, we discuss constraint and trigger facilities in other systems.

2. OBJECTS IN O++: A BRIEF REVIEW

The O++ object facility is based on the C++ object facility and is called the *class*. Class declarations consist of two parts: a specification (type) and a body. The class specification can have a private part holding information that can only be used by its implementor, and a public part which is the class user interface. The body consists of the bodies of the *member* functions (methods) declared in the class specification but whose bodies were not given there. For example, here is a specification of the class `item`:

```
class item {
    Name nm;
    double wt; //in kg
public:
    item(Name xname, double xwt);
    Name name() const;
    double weight_lbs() const;
    double weight_kg() const;
};
```

C++ supports inheritance, including multiple inheritance [25], which is used for object specialization. The specialized object types inherit the properties of the base object type, i.e., the data members and member functions, of the *base* object type. As an example, consider the following class `stockitem` that is derived from class `item`:

```
class stockitem: public item {
   int consumption;  //qty consumed per year
   int leadtime;     //lead time in days
public:
   int qty;
   double price;
   stockitem(Name iname, double iwt, int xqty, int xconsumption,
             double xprice, int xleadtime);
   int eoq() const;  //economic order quantity
};
```

`stockitem` is the same as `item` except that it contains other information such as the quantity in stock, its consumption per year, its price and the lead time necessary to restock the item.

O++ extends C++ by providing facilities to create persistent objects. O++ visualizes memory as consisting of two parts: volatile and persistent. *Volatile* objects are allocated in volatile memory and are the same as those created in ordinary programs. *Persistent* objects are allocated in persistent store 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 [16]. The object identity is referred to as a *pointer to a persistent object*.

Persistent objects are allocated and deallocated in a manner similar to heap objects. Persistent storage operators `pnew` and `pdelete` are used instead of the heap operators `new` and `delete`. Here is an example:

```
persistent stockitem *psip;
...
psip = pnew stockitem(initial-values);
```

`pnew` allocates the `stockitem` object in persistent store and returns its id in `psip`. Note that `psip` is a pointer to a persistent `stockitem` object, and *not* a persistent pointer to a `stockitem` object.

Persistent objects can be copied to volatile objects and vice versa using simple assignments:

```
*sip = *psip; /*copy the object pointed to by psip*/
              /*to the object pointed to sip      */
*psip = *sip; /*and vice versa */
```

Components of persistent objects are referenced like the components of volatile objects, e.g.,

```
w = psip->weight_kg();
```

All `stockitem` objects in the database (i.e., in persistent store) can be examined using the following `for` statement:

```
for (psip in stockitem) {
    ...
}
```

Transactions in O++ have the form

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

Transactions are aborted using the `tabort` statement.

3. DESIGN GOALS

When designing the trigger and constraint facilities in Ode, we kept the following design goals in perspective:

- Trigger and constraints should be specified declaratively.
- Triggers and constraints should be associated with class definitions to reflect object orientation (just as member functions are associated with class definitions).
- Constraints and triggers should work with the inheritance mechanism (including multiple inheritance).
- Trigger and constraint predicate (condition) checking should be minimized. It is clearly infeasible to check every trigger and constraint before a transaction commit. In an object-oriented environment where the operations can be user defined, the system cannot determine automatically which operations will affect the trigger and constraint predicates. Consequently, it must be possible to narrow sufficiently the points at which the predicates have to be checked [6].
- Constraint violations should be able to abort a transaction, raise an exception, or take any other specified recovery action.
- In the Event-Condition-Action terminology of [19], immediate, deferred and separate execution modes should all be supported.

The mechanisms for triggers and constraints are related because one can think of and implement a constraint as a trigger whose action is executed when the negation of the constraint predicate become true. However, we have provided separate facilities for triggers and for constraints since the two are logically different. For example:

1. Constraints ensure consistency of the object (and database) state. If this consistency cannot be maintained (on an object update basis or on a transaction basis), then the transaction is aborted. Triggers are not concerned about object consistency. They are fired whenever the specified conditions become true.

2. Actions associated with a constraint violation are executed as part of the transaction violating the object constraints. On the other hand, trigger actions are initiated as separate transactions. The reason for this is that the transaction violating a constraint must be aborted if the violation cannot be fixed, while the triggering transaction (such as one recording the approach of an enemy aircraft) should be allowed to commit even if the triggered transaction (such as one to fire a missile at the enemy aircraft) aborts for some reason.

3. Constraints apply to an object from the moment it is created to the moment it is deleted. Triggers must explicitly be activated after the object has been created.

4. All objects of a given type have the same constraints. But this is not true for triggers: different triggers may be activated for different objects even though the objects maybe of the same type. For example, an object representing stock A may have an active trigger to sell the stock if its price follows below a certain amount. But the object representing stock B may not have any active triggers.

It should be noted that the distinction between constraints and triggers has been made with a view to providing a ‘‘natural’’ expressive mechanism for commonly required constructs. However, the user always has the full power of the C++ programming language available, should it be desired, for example, to have some triggered action execute as part of the triggering transaction, or to associate parameters with a constraint.

4. CONSTRAINTS

Constraints are used to maintain a notion of consistency beyond what is typically expressible using the type system [20]. Updates that violate the specified constraints should not be permitted. Interpretations of consistency are usually application specific and may be arbitrarily complex. Constraints, which are Boolean conditions, are associated with class definitions. All objects of a class must satisfy all constraints associated with the class.

Violation of a constraint, if not rectified, will abort the transaction causing the violation. Depending upon the mechanism used, the constraint violation may have to be rectified immediately after the violation is reported or may have to be rectified but before the completion of the transaction.

Constraints in Ode consist of two parts: a predicate and an action (or handler). This action is executed when the predicate is *not* satisfied. Constraint checking can be performed after accessing the object or at some later point in time. For example, in design applications, it is sometimes appropriate to defer constraint checking to just before the transaction commit instead of performing it right after accessing the object. This allows for temporary violations of constraints (which is likely to happen when the constraints of two objects depend upon each other’s values and one of the objects is updated) that are rectified in actions following the object update before the transaction attempts top commit. Consequently, to support these two modes of constraint checking we support two kinds of constraints: hard and soft.

4.1 Hard Constraints

Hard constraints are specified in the constraint section of a class definition as follows:

```
constraint:
    constraint_1: handler_1
    constraint_2: handler_2
    ...
    constraint_n: handler_n
```

constraint$_i$ is a Boolean expression that refers to components of the specified class and *handler*$_i$ is a statement that is executed when a constraint is violated. Constraints are checked only at the end of constructor and member (friend) function calls (but not at the end of destructor calls). Although we do not prohibit accessing the public data components of an object directly, it is the programmer's responsibility to ensure that such accesses do not violate any constraints because no constraint checking is performed for such accesses.

If any constraint associated with an object is not satisfied and there is no handler associated with it, then the transaction of which this access is a part is aborted (and rolled back). If there is a handler associated with the constraint, then this handler is executed and the constraint is re-evaluated. If the constraint is still not satisfied, then the transaction is aborted.

The granularity of hard constraint checking is at the member function level. This has two important advantages: objects are always in a consistent state (except possibly during an update operation) and the implementation of constraint checking is simplified. The notion is that each public member function must leave the object in a consistent state.

Here is an example of a hard constraint:

```
class supplier {
    Name state;
    ...
constraint:
    state == Name("NY") || state == Name(""):
        printf("Invalid Supplier State\n");
};
```

After a `supplier` object has been created or accessed, the constraint is checked. The constraint is violated if the supplier's location is specified and it is not in New York (NY). The statement associated with the constraint will be executed and the constraint checked once again. If the constraint is still not satisfied, as it will not be in this particular example, then the transaction is aborted.

As another example, the following code fragment specifies that an employee's salary must always be less than the manager's salary. Note the two complementary constraints, one in each of the two types of objects involved:

```
class manager;
class employee {
    ...
    persistent manager *mgr;
    float sal;
public:
    ...
    float salary() const;
constraint:
    sal < mgr->salary();
};

class manager : public employee {
    persistent employee *emp<MAX>;
    int sal_greater_than_all_employees();
    ...
public:
    ...
constraint:
    sal_greater_than_all_employees();
};

int manager::sal_greater_than_all_employees()
{
    persistent employee *e;
    for (e in emp)
        if (e->salary() > salary())
            return 0;
    return 1;
}
    ...
```

Where there are multiple constraints associated with an object, the placement of constraints does not specify the order in which the constraints will be checked. We believe that it is in spirit of declarative semantics not to specify any ordering even though many programming languages such as Prolog do not follow this principle. Users should write the action parts of the constraints without making assumptions about the order of execution. However, we do guarantee that the condition checking and action parts of each constraint execute atomically (with respect to the other constraints).

Hard constraints are only checked at the boundaries of public member functions that update objects To promote code sharing, any number of private member functions can be defined, and these can execute in environments in which the constraints are not satisfied.

4.2 Soft Constraints

Hard constraints ensure that objects are internally consistent at all times. Thus we can ensure that a bank balance is not be negative even momentarily (outside of a public member function) in the middle of a transaction. In traditional database systems, a transaction is the smallest unit across which integrity must be maintained. Checking integrity constraints at a granularity finer than that of a transaction can lead to problems. For example, in the above salary example if both employee and manager are to be given raises in a single transaction, our implementation choice forces us to give the manager the raise first and then the employee. Giving a raise first to the employee could momentarily cause his/her salary to become greater than the manager's. This will result in a constraint violation and will cause the transaction to abort — in spite of the fact that no constraint would have been violated at the end of the transaction had it been allowed to complete. Forcing the order of events within a transaction is not desirable.
Even worse, consider the following example:

```
class person
{
    ...
    persistent person *spouse;
public:
    ...
constraint:
    (spouse == NULL) || (this == spouse->spouse) ;
};
...
person p1, p2;
```

The above constraint specifies that if a person has a spouse, then the spouse's spouse must be the person himself/herself. Initially, let us suppose that `person` objects `p1` and `p2` were not married. Now consider a transaction to record the fact that `p1` and `p2` have married each other. The moment `p2` is made the spouse of `p1`, the above constraint will be violated because the spouse field of `p2` has not been updated. The reverse problem occurs if `p1` is first made the spouse of `p2`. In either case, the transaction will be aborted. In fact, the same problem occurs whenever a pair of complementary relationships has to be maintained between two objects.

To handle such cases, we need a *deferred* or *transaction-level* constraint checking mechanism. Transaction level constraint checking is supported with *soft* constraints in Ode. Soft constraints are specified like hard constraints except that the keyword `soft` precedes the keyword `constraint`, e.g.,

```
class person
{
    ...
    persistent person *spouse;
public:
    ...
soft constraint:
    (spouse == NULL) || (this == spouse->spouse);
};
```

In general, soft constraints are used when other objects are involved in the constraint. Hard constraints are likely to be used when the constraint condition does not involve other objects.

4.3 Comments on Constraints

Constraint handlers are specified in the class definition and they are the same for all transactions. An alternative approach could be to allow each transaction to specify its own constraint handlers. Not only would this approach be inefficient and notationally inelegant, it would also be inconsistent with our object-oriented approach in which the constraint handlers are specified in the object itself. Transaction dependence can be incorporated in constraint handlers by making their actions depend on the value of an object component that is set by the transaction.

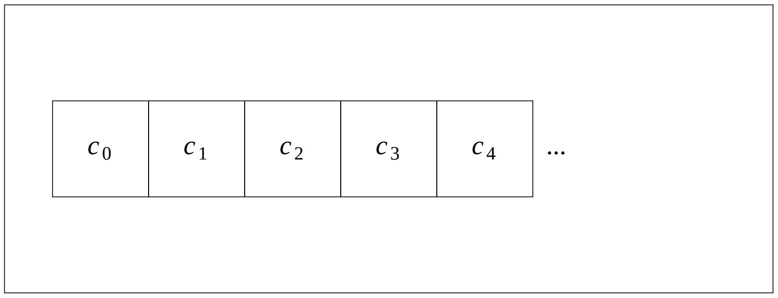
The choice of having immediate (hard) or deferred (soft) constraint checking is made at class definition time and not at run time (during the transaction). Soft constraint checking is delayed until just before the end of the transaction. All soft constraints that need to be checked (i.e., those associated with objects that have been updated) are then checked, in some unspecified order. As in the case of hard constraints, the order in which the constraints have been stated in the class definition is immaterial. Furthermore, the order of occurrence of the updates that require these constraints to be checked is also immaterial. The reason for this is that there could be multiple events in the course of a transaction that cause the same constraint to be evaluated. It is wasteful, and potentially incorrect, to check such a soft constraint several times. On the other hand, there is no clear semantic justification to order it according to say, the first event that requires its checking. Another reason is that in an implementation that permits intra-transaction parallelism (for example, through the use of nested transactions), the order of these events may not be deterministic, and we would not want the results of the program execution to depend on this order.

A derived class inherits the constraints of its parent class and new constraints can be added. Consequently, constraints can be used to specialize classes. Such constraint-based specializations are useful in many applications, e.g., in frame-based knowledge representation systems [7].

Constraints specified in a class definition can conflict with other constraints in the same class definition or with inherited constraints. For example, one constraint may be the negation of another constraint. In general, it is not possible to automate the detection of such conflicts. It is the programmer’s responsibility to ensure that such conflicts do not happen. Otherwise, transactions involving objects of a class with conflicting constraints will always abort.

4.4 The Domino Effect

In CAD applications, constraints often involve other objects such as neighbors. As an example, consider a row of adjacent cells on a chip that are placed next to each other. Except for the end cells, each cell has two neighbors.



A cell must always satisfy the following conditions:

1. It must be on the chip.

2. It should be adjacent to but must not overlap its left neighbor (if any).

3. It should be adjacent to but must not overlap its right neighbor (if any).

These conditions must be satisfied when a cell is created and when a cell is moved. They are specified in the `constraint` section of class `cell`:

```
class cell {
    persistent cell *left, *right;
    …
public:
    int x, y; //Coordinates of the center point
    int width, height;
    cell(int x1, int y1, int width1, int height1);
    void neighbors(persistent cell *left1, persistent cell *right1);
    void shift(int dx);
    …
constraint:
    x-width/2 >= XMIN &&  x+width/2 < XMAX;

    (right == NULL) || x+(width+right->width)/2 == right->x:
        right->shift((width+right->width)/2 - (right->x-x));

    (left == NULL) || x-(width+left->width)/2 == left->x:
        left->shift((x-left->x)-(width+left->width)/2);
    …
};
```

The three constraints ensure that the three conditions for row cells listed above are satisfied when a cell is created or when is moved. A constraint violation causes the transaction to abort. Before the abortion occurs, the statements associated with the constraint are executed in an attempt to rectify the violation. In particular, the constraint actions in the second and third constraints attempt to fix the constraints by shifting the neighbors. The constraint violation domino effect occurs from the fact that moving a cell violates its constraint. To ensure that its constraint is satisfied, the cell must move its appropriate neighbor, which in turn will violate the neighbor’s constraint. And so on. Notice that if any cell is moved outside the chip (x-coordinate of left-end is less than `XMIN` or x-coordinate of right-end is greater than `XMAX`), then the resulting constraint violation cannot be repaired.

Here is the code for the member functions of class `cell`:

```
cell::cell(int x1, int y1, int width1, int height1)
{
    x = x1; y = y1;
    width = width1; height = height1;
    left = NULL; right = NULL;
}
void cell::neighbors(persistent cell *left1, persistent cell *right1)
{
    left = left1; right = right1;
}
void cell::shift(int dx)
{
    x += dx;
}
```

Note the simplicity and the declarative nature of the specification, and compare it to the fairly complex procedural description (not shown here) that would have been required, had the constraint mechanism not been available.

The domino effect can be used not only to maintain integrity of the database in the way that regular constraints can be used, but also for maintaining materialized views, updating derived data that has been cached, and where data values are defined relatively rather than in absolute terms. Another examples where the domino effect can occur is in spreadsheets where changing an element may causes several totals and tallies to be altered.

4.5 Referential Integrity

Referential integrity is a form of integrity constraint supported by many relational databases. If an entity (object) is ‘‘referred to’’ in some relation, then the entity must exist in a *primary* relation listing all entities of that type. For example, if a ‘‘Supplier’’ relation records a certain company as supplying a certain part number, then the part number must be a valid part number recorded in some ‘‘Parts’’ relation. In an object-oriented database framework, referential integrity means that there should not be any references to non-existent objects. The database must thus guarantee that if an object id is recorded in the database, then the corresponding object must be present in the database.

In Ode half the referential integrity problem is solved even without any special effort on our part! Object identifiers cannot be manufactured accidentally. If an object identifier is recorded in the database, it must have been received from the persistent object allocator `pnew` or retrieved from the database itself, and it represents a valid object.

However, the other half of the problem is not so easy. When an object is deleted from the database, we have to make sure that the identifier of this object is not referenced by any other object in the database.

Implementing this requires the use of reference counts, or indexing on object identifier references. Either of these techniques can be used, at the implementation level, to enforce referential integrity.

Our interest here is primarily language constructs for specifying constraints and triggers. Any strategy chosen to implement full referential integrity is beyond the scope of this paper.

5. TRIGGERS

Triggers, like integrity constraints, monitor the database for some conditions, except that these conditions do not represent consistency violations [20]. A trigger, like a constraint, is specified in the class definition and it consists of two parts: a condition and an action. Triggers apply only to the specific objects with respect to which they are activated. Triggers are parameterized, and can be activated multiple times with different parameter values.

If a trigger is active, then when its condition becomes true, the action associated with the trigger is executed. Unlike a constraint handler, which is executed as part of the transaction violating the constraint, a trigger action is executed as a separate transaction. A constraint action must maintain database integrity prior to the transaction commit; trigger actions have not such concerns. In typical applications such as process control, an early warning system, or a stockbroker's trading program, events that cause trigger firing, e.g., events such as excessive boiler pressure, enemy aircraft detection, or stock price changes, can be independent of any consequent actions. Thus trigger actions need not be part of the transaction firing the trigger. Also, aborting a trigger action should not result in the abortion of the transaction firing the trigger. For example, if a stockbroker is unable to fill a customer's buy order because of insufficient margin requirements, we would certainly not want to abort the transaction recording the change in the stock price in the broker's database. Thus, there are semantic requirements for the triggered action to execute as a separate transaction. Another reason in favor of making trigger actions to be separate transactions is that this results in smaller transactions which improves concurrent access and minimizes cascaded aborts, and hence enhances database performance.

Triggers that fire will be recorded and their actions executed (as separate transactions) only if the transaction causing them to be fired commits successfully. Otherwise, the trigger actions will not be executed. On the contrary, a constraint action is executed whether or not the transaction violating the constraint finally commits. Since a constraint action is part of the firing transaction, if the transaction eventually aborts, any updates caused by the constraint action will be rolled back. Thus the trigger action transactions are executed after (but not necessarily immediately after) the triggering transaction, i.e., there is ‘‘weak coupling’’ [11] between the triggering transaction and the trigger action.

Since the action part of a trigger is executed as a separate transaction, it is possible that the condition causing the trigger to fire is no longer true at the time the triggered action is actually executed. For example, the stock price after falling to a level at which a customer's buy order is triggered, could rise above the trigger threshold price before the buy transaction can complete. To prevent purchase of the stock at this now changed (higher) price, the trigger action must check that the stock price is at or below the threshold; otherwise, it should deal with it appropriately. See Sec. 6.3 for a discussion of possible coupling mechanisms.

An associated issue is that of real-time constraints associated with triggers. For example, a trading program that wishes to arbitrage on price differences in two different markets must be able to execute the buy and sell orders essentially simultaneously to avoid the risk of substantial losses resulting from price change. Similarly, if a process control application detects high boiler pressure, then it must be able to initiate timely corrective action before the boiler explodes. Executing trigger actions is similar to executing ordinary transactions, and therefore all the problems and solutions of real-time systems apply. Further discussion of these issues is beyond the scope of this paper. See [1, 18] for a discussion on real-time execution.

5.1 The Mechanism

Ode supports two kinds of triggers: *once-only* (default) and *perpetual* (specified using the keyword `perpetual`). A once-only trigger is automatically deactivated after the trigger has ‘‘fired’’, and it must then explicitly be activated again, if desired. On the other hand, once a perpetual trigger has been activated, it is reactivated automatically after each firing.

Triggers are specified within class definitions:

```
trigger:
    [perpetual] T1(parameter-decl1): trigger-body1
    [perpetual] T2(parameter-decl2): trigger-body2
    ...
    [perpetual] Tn(parameter-decln): trigger-bodyn
```

T_i are the trigger names. Trigger parameters can be used in trigger bodies, which have the form

```
trigger-condition => trigger-action
within expression ? trigger-condition => trigger-action
                                   [ : timeout-action ]
```

The second form is used for specifying *timed* triggers. Once activated, the timed trigger must fire within the specified period (floating-point value specifying the time in seconds); otherwise, the timeout action, if any, is executed.

Triggers are associated with objects; they are activated explicitly after an object has been created. A trigger T_i associated with an object whose id is *object-id* is activated by the call

object-id`->`T_i`(`*arguments*`)`

The trigger activation returns a trigger id (value of the predefined class `TriggerId`) if successful; otherwise it returns `null_trigger`. The object id can be omitted when activating a trigger from within the body of a member function.

An active trigger ‘‘fires’’ when its condition becomes true (as a result of updates by a transaction). Firing means that the action associated with the trigger is ‘‘scheduled’’ for action as a separate transaction. Only active triggers can fire. No performance penalty is incurred for triggers that have not been activated.

Trigger activation must be done explicitly for each individual object. However, the class designer can automate trigger activation by putting the trigger activation code in constructors. Since a constructor function is called at object creation time to initialize the object, the trigger automatically gets activated when an object is created. Because triggers are activated explicitly (by the programmer or by the class designer), different objects of the same type may have different sets of triggers active at any given time.

Triggers can be deactivated explicitly before they have fired using the `deactivate` function:

`deactivate(`*trigger-id*`)`

The trigger with identifier *trigger-id* is deactivated. If successful, `deactivate` returns one; otherwise, it returns zero.

Multiple activations of the same trigger associated with an object (possibly with different arguments) are allowed. For example, there can be multiple activations of the buy trigger associated with a stock object with each buy trigger being activated with different price and quantity arguments.

An active trigger can be fired no more than once by a given transaction, even if the transaction causes several updates to the relevant object, any one of which could by itself have satisfied the trigger condition and caused it to fire. However, there is no limit on the number of activations of the same trigger that could be fired by a single transaction. Trigger conditions may overlap. Consequently, updating an object may result in the firing of one or more active triggers.

Triggers, like constraints and members of a class, are inherited when one class is derived from another. Also, like constraints, neither the order of placement of triggers in a class definition, nor the order in which the triggers are activated, can be used to determine the order in which the triggers will be evaluated or executed. (See Sec. 4.3 for a discussion of some the issues involved.) In fact, since the action part of each trigger is executed as a separate transaction, it is impractical to control the execution schedule to force these transactions to be serializable in a specified order. Unlike in the case of constraints, the conditional evaluation and action parts of a trigger are not executed together atomically, though each individually is.

5.2 Simulating a Perpetual Trigger with Once-only Triggers and Vice-versa

Perpetual triggers cannot be simulated with once-only triggers by re-activating the trigger in the trigger action. A trigger action is executed as a transaction after the transaction firing the trigger commits. Consequently, a once-only trigger will be inactive for the period between when it is fired and when it is reactivated in the trigger action. Similarly, a once-only trigger cannot be simulated by a perpetual trigger that is deactivated in the trigger action since it will remain active until the trigger action is executed and the trigger is deactivated. Effectively, a once-only trigger is automatically and atomically de-activated after it fires. The atomicity is provided by the implementation, and cannot be simulated at the language level.

Recall that a trigger is fired by one transaction, but executes as a separate transaction. Before the fired separate transaction commits, it is possible for the same perpetual trigger to be fired again due to an update of an object by a third transaction. The action part of a perpetual trigger has to take this possibility of multiple firings into account. This problem does not arise in a once-only trigger, since the trigger is atomically deactivated when it is fired, and has to be activated again explicitly, in the action part of the trigger or elsewhere.

5.3 Examples

Consider the following class `inventitem`, derived from class `stockitem` that was shown earlier:

```
class inventitem: public stockitem {
public:
    inventitem (Name iname, double iwt, int xqty, int xconsumption,
              double xprice, int xleadtime, Name sname, Addr saddr);
    void deposit(int n);
    void withdraw(int n);
    ...
trigger:
    order(): qty < reorder_level() ==> place_order(this, eoq());
                              //"this" refers to the object itself
};
```

Trigger `order` is activated in the constructor function `inventitem` and in the member function `deposit`. The action associated with the trigger `order` will be executed after its condition becomes true (as a result of executing the `withdraw` operation).

Here are the bodies of some of the member functions of class `inventitem`:

```
inventitem::inventitem(Name iname, double iwt, int xqty,
    int xconsumption, double xprice, int xleadtime,
    Name sname, Addr saddr):
    stockitem(iname, iwt, xqty, xconsumption, xprice, xleadtime,
            sname, saddr)
{
    …
    order();  //trigger activation
}
void inventitem::deposit(int n)
{
    qty += n;
    order(); //trigger activation
}
void inventitem::withdraw(int n)
{
    qty -= n; //might fire trigger
}
```

Now suppose that we wish to write a complaint if the supplier does not fill our order within the promised lead time. We could achieve this result by using a timed trigger as follows:

```
class cinventitem: public stockitem {
public:
    …
    TriggerId checkarrival;
    int delivered;
    void deposit(int n);
trigger:
    order(): qty < reorder_level() ==> { place_order(this, eoq());
                                        delivered = 0;
                                        checkarrival = complain(); }
    complain(): within leadtime ? delivered ==>;
                                : write_complaint_letter();

};
```

```
void cinventitem::deposit(int n)
{
    qty += n;
    delivered = 1;
    deactivate(checkarrival);
    order(); //trigger activation
}
```

Just as there are innumerable ways of writing non-terminating programs with a general programming language, there are also innumerable different ways of generating conflicts with triggers and constraints. There is considerable theoretical work [8, 17] on detecting such conflicts at compile time, for some limited types of constraints, in general only appropriate programming skill can prevent these problems. Consider, for example, the following trigger that sets the salary of an employee specified as a parameter (`emp`) to the salary of the employee the trigger is associated with:

```
class employee {
    ...
    float salary;
public:
    void change_salary(float sal);
    ...
trigger:
    perpetual update(persistent employee *emp):
         changed(salary) => emp->change_salary(salary);
};
```

The O++ macro `changed` (which is used in class definitions) returns true if its argument is changed in the current access.

Now consider two objects `fred` and `mike` of type `employee`. The trigger associated with `fred` is activated as:

```
fred->update(mike)
```

If the salary of `fred` changes, trigger `update` will set the salary of `mike` to that of `fred`.

And the following trigger activation will set the salary of `mike` to that of `joe`:

```
joe->update(mike)
```

These two triggers together ensure that `mike`'s salary is set equal to that of either `fred` or `joe` depending upon whose salary was changed last. It is conceivable that this may exactly be what the user wants. Now suppose we activate the trigger `update` as follows:

```
mike->update(joe)
```

If `mike`'s salary changes, then `joe`'s salary is also set to the same value, and vice versa. So we have an infinite recursion. The reason is that the `changed` macro simply checks if its argument variable has been assigned a new value in the current update; it returns true even if the new value the same as the old value. To prevent such recursion, the trigger code can be rewritten:

```
perpetual update(persistent employee *emp):
     changed(salary) && emp.salary != salary=>emp->change_salary(salary);
```

6. DISCUSSION

6.1 Intra-Object Versus Inter-Object Constraints & Triggers

A constraint or trigger is said to be *intra-object* if:

i. It is associated with a (single) specific object, and

ii. the condition associated with it is evaluated only when this object is updated.

Otherwise, a constraint or trigger is said to be *inter-object*.

An intra-object constraint or trigger can refer to other objects both in evaluating the condition and in the subsequent action. However, updates to these referenced objects do not require the condition part of the constraint or trigger to be checked. (See discussion below on what is an *event*).

We opted for intra-object constraints and triggers for several reasons. First off, by associating constraints and triggers with class definitions, we have incorporated them in the framework of C++ without violating its object-oriented philosophy. In case of inter-object constraints and triggers, which can refer to objects of different types, it is not clear where they should be specified. It is not appropriate to place such constraints and triggers in only one object type if they involve multiple object types. And we did not want to make constraints and triggers full fledged types which are associated with appropriate object types. Constraint and trigger types would add much semantic complexity to O++. e.g., where can constraint type objects be used, can they passed as parameters, do they have constructors and desctructors associated with them, can pointers refer to them and so forth. One straightforward solution to implementing the functionality of inter-object constraints and triggers is to specify appropriate intra-object constraints and triggers in the definition of the object types involved. Indeed, most inter-object constraints and triggers can be implemented using one or more intra-object constraints or triggers. This was certainly the case in all the examples that we worked out. A typical case is the ‘‘employee's salary no greater than the manager's salary’’ example give in Sec. 4. This is clearly an inter-object constraint, involving two objects: an employee and a manager. This inter-object constraint is then converted into two complementary intra-object constraints, one to be associated with the employee and the other to be associated with the manager. Similarly, we showed, in the previous section, how intra-object triggers could be used to simulate an inter-object periodic trigger. See [15] for a systematic technique to obtain intra-object constraints and triggers from inter-object contraints and triggers.

Secondly, in terms of the E-C-A (event-condition-action) model [19], in Ode the condition and action are explicitly specified for every constraint and trigger. The event is not explicitly specified. For an intra-object constraint or trigger, this event can be assumed to be the updating of the associated object, as discussed in Section 6.3. For an inter-object constraint or trigger, the event might be the update of any one of the objects involved in the constraint or trigger. Checking for these events make inter-object constraints and triggers significantly more expensive than intra-object constraints and triggers.

Finally, physical locality makes intra-object constraints and triggers more efficient to implement than inter-object constraints and triggers. When an event occurs, the condition(s) involve only components of object being updated which means it is in memory.

6.2 Events

Constraints and triggers can be thought of as event-condition-action (E-C-A) triples. Our constraint and trigger facilities require explicit specification of the condition and the action. But events leading to the evaluation of the constraint and trigger conditions are not specified explicitly. We consider object updates as events since only updates affect the constraint and trigger conditions. (Object updates are ‘‘natural’’ candidate events in O++.)

There are many ways to update an object. To get a proper handle on updates and without incurring substantial overhead, we consider as events only object updates caused by public member functions. Specifically, direct updates to objects, such as those caused by changing values of public data members, are not considered to be events. Updates caused by private member functions are not considered to be events since these functions can be called (directly or indirectly) only by public member functions. In such cases, the event is associated with the public member function initiating the update.

Sometimes it may not be clear or possible to determine whether or not a member function updates an object. In our current implementation, we consider each invocation of a non `const` public member function as causing a potential update and we therefore check the constraint and trigger conditions just prior to the termination of the function. Note that `const` member functions cannot update an object.

The action part of a constraint associated with an object could cause updates that will affect constraints and triggers in the same object or in different objects (as happens in the VLSI cell placement example). In particular, this means that the action part of a constraint, in fixing the constraint violation, may violate some other constraint. If the update causing this violation is not done through a public member function, then there is no guarantee that this constraint violation will be detected (immediately).

C++ (and therefore O++) allows expressions to have side effects. It is therefore possible for updates to be embedded in the condition evaluation. Such embedded updates are not treated as events unless they occur via public member functions. Consider as an example the class `loginventitem` shown below that logs information after every `n` updates, by calling function `loginfo`.

```
class loginventitem: public stockitem {
public:
    int count;
    loginventitem(...);
    void loginfo();
    ...
trigger:
    perpetual log(int n): ++count % n == 0 ==> loginfo();
};

loginventitem::loginventitem(): stockitem(...)
{
    int freq;
    ...
    count = 0;
    log(freq);  //trigger activation
}
```

The trigger relies on the fact that the trigger condition will be evaluated every time the object is updated. Such reliance is bad because direct updates will not cause evaluation of the trigger condition and an optimizing O++ compiler may not evaluate the trigger condition when components specified in the trigger condition are not updated.

6.3 Coupling Modes

In [19], three types of coupling between an event (E) and a condition (C) have been identified: the condition evaluation is immediate (when the object is changed), deferred (as part of the transaction but at the end), and separate (in a separate transaction). Similarly, any of these three types of coupling could exist between condition and action (A). In Ode, the condition is evaluated as part of the transaction updating the object (after the object has been updated). In other words, the E-C coupling is always immediate (for hard constraints and for triggers) or deferred (for soft constraints), but never separate. Conversely, the C-A coupling is either immediate (for constraints) or separate (for triggers), but never deferred. We do not provide the separate E-C coupling mode or the deferred C-A coupling mode because we believe that would be overkill. What seems to be of greatest consequence is the coupling between the event and the action (which is the ‘‘weaker’’ of E-C and C-A couplings). This is because only the evet and the action affect the state of the database. For constraints, the C-A coupling is the strongest possible (immediate), so the E-C coupling determines the strength of the E-A coupling (either immediate or deferred). For triggers, the C-A coupling is the weakest (separate), so the E-A coupling is also separate irrespective of the E-C coupling used, and we have chosen to use immediate since that is the most efficient. We thus support the full spectrum of E-A couplings.

In our scheme, a weak E-A coupling is implemented by using a separate C-A coupling, i.e., a trigger. For example, consider the triggered purchase of stock when the stock price falls below a certain threshold. The event is a change in the stock price. The condition is that the stock prices is below a threshold value. The action is its purchase. We want the purchase action to be a separate transaction from the change in price event, therefore we implement this task as a trigger. However, we would like to make sure that the stock price is still below the threshold when the purchase takes place, so we are forced to check the condition a second time as part of the action. An alternative implementation could have been to have a separate E-C coupling, so that a new transaction is fired every time the stock price changes, and then have an immediate C-A coupling so that the purchase occurs immediately if the price is below the threshold. While such an implementation would avoid second condition check, it will be much less efficient than the first one because of the large number of transactions spawned.

Even having selected two types of E-C couplings, and two types of C-A couplings, four combinations are possible, and we support only three of the four. The combination not supported is a deferred E-C coupling with a separate C-A coupling, that is, a ‘‘soft’’ trigger in which the trigger conditions are checked at the end of the transaction rather than immediately after the update in the transaction that could potentially cause the trigger to fire. The reason is that it is always possible to check the trigger condition again as part of the action in a regular (‘‘hard’’) trigger, thereby simulating a ‘‘soft’’ trigger, but not vice versa. By defining triggers the way we have, even momentary spikes in the value of a variable being monitored can be ‘‘captured’’ in that they can cause a trigger to fire, even if this spike is fixed before a transaction commits.

7. IMPLEMENTATION

Our implementation strategy is based on the premise that object updates are performed only by calling public member functions. Constraints and trigger conditions are not checked if objects are updated by directly changing the values of the data members. We encourage the definition of classes whose data members are private and accessed using public member functions. Where appropriate, member functions could be declared `inline` to reduce the execution overhead.

When an object with soft constraints is updated, it is placed in a ‘‘to-be-checked’’ list. An alternative implementation, which we considered and decided against, is to check the soft constraints when an object is updated, and place it on the to-be-checked list only if its constraints are violated. The resulting to-be-checked list will then be smaller, and many objects need not be accessed an extra time (for constraint checking) immediately prior to the transaction commit. However, the constraints will now have to be checked twice (once at object update time, and once at transaction commit time). Another problem would be dealing with situations where one object access violates the constraints but a succeeding second access fixes the violation. Finally, the semantics of the alternative implementation are a little more complex.

Multiple updates to the same object in the course of a transaction will cause a hard constraint to be evaluated once on each update. A corresponding soft constraint will only be checked once at the end of the transaction, since any given soft constraint is permitted to appear no more than once in the to-be-checked list. Note though, that updates using public member functions in the action part of one soft constraint can cause another soft constraint to be re-checked depending on the order of checking selected by the implementation.

Hard constraints, soft constraints, and triggers are encapsulated into member functions `hard_constraints`, `soft_constraints`, and `triggers`, respectively. These functions are called to check constraints and fire active triggers. In addition, each trigger T_i is converted into a member function T_i with the same parameters. This function will be called in response to a trigger activation request.

The `hard_constraints` and `triggers` member functions are called from within each member function as illustrated below while `soft_constraints` are called *once* for each object updated prior to a transaction commit.

A class definition of the form

```
class class {
    ...
public:
    ...
constraint:
    ...
soft constraint:
    ...
trigger:
    [perpetual] T1(...): ...
    [perpetual] T2(...): ...
    ...
    [perpetual] Tn(...): ...
};
```

can be translated into the C++ class (along with definitions of the additional member functions) as follows:

```
class class {
    ...
public:
    ...
    void hard_constraints();
    void soft_constraints();
    void triggers();
    TriggerId T1(...);
    TriggerId T2(...);
    ...
    TriggerId Tn(...);
};
```

Functions `hard_constraint` and `soft_constraint` invoke the transaction abort statement `tabort` if the constraints checked by them are not satisfied, even after execution of any corrective action part specified.

Trigger functions T_i simply activate the triggers and the trigger condition checking function `triggers` fires the triggers if the conditions are satisfied. When a trigger is fired, its action is executed as a separate transaction provided the transaction causing the trigger to be fire commits successfully. Each trigger function T_i creates an entry for the specified trigger and object id in a table of currently active triggers, and returns a key to the entries in the table. Function `deactivate` takes the key to the table as an argument, and deletes the table entry corresponding to the key.

To implement hard constraint checking and trigger firing, member functions of the form

```
class::functionName(parameters)
{
   ...
   return e;
   ...
}
```

are translated to

```
class::functionName(parameters)
{
    ...
    record object state
    ...
    hard_constraints(); trigger(); // before each return statement
    return e;
    ...
    hard_constraints(); trigger(); // at the end
}
```

A timed trigger is implemented as two triggers: one associated with the object for which the timed trigger has been defined, and a second associated with the timer object. The former is of the form

[perpetual] T_1: *trigger-condition* ==> *trigger-action*

The latter is of the form:

T_2: *within-expression-condition-timeout* ==> *timeout-action*

Where T_2 is a special kind of once-only trigger that not only deactivates itself upon firing, but also deactivates T_1. Note that this deactivation of T_1 occurs at the end of the specified time period, whether it is defined to be perpetual or once-only.

8. RELATED WORK

The idea of having triggers and constraints in a database is not new. Facilities for active databases appeared as early as in CODASYL, in the form of ON conditions. System R provided triggers and constraints as a mechanism for enforcing integrity constraints [5, 13]. Simple triggers are now appearing in commercial systems. For example, Sybase [10] provides facilities to specify *rules* and *triggers*. *Rules* are integrity constraints that go beyond those implied by a column's data type. These are a special case of our constraint mechanism, with no actions associated. A *trigger* is a special kind of stored procedure that goes into effect when the specified table is updated. Among other things, it can be used to disallow or roll back changes that violate a constraint, such as an employee's salary being made greater than the manager's, and to keep summary data, such as year-to-date sales. A trigger can be affected by changes to only one table. Each table can have at most three triggers: an update, an insert, and a delete trigger. One trigger cannot call another. If a trigger updates a table with which another trigger is associated, the second trigger will not fire.

In contrast, observe that Ode allows triggers to be associated with objects, which are essentially tuples. Multiple triggers can be associated with any individual object, and these triggers can be parameterized. Also, any number of triggers (and constraint actions) can be fired recursively.

In many systems, integrity constraints (and triggers) are expressed in a separate language that is distinct from the normal database query (and programming) language. As a consequence not only does the programmer have to learn two separate languages, but also, it becomes difficult to invoke arbitrary actions when such triggers and constraints fire, and it may be difficult to check the conditions on every update.

In relational databases, a distinction is sometimes made between a trigger associated with an attribute and a trigger associated with a tuple (or between triggers associated with a column and those associated with the entire relation). This difference appears to be an artifact of the relational model and is not relevant in our context. A trigger is always associated with an object. The trigger condition could involve one or more attributes of the object — it makes no difference. In fact, the condition may even use attribute values from other objects.

Rules have been built into the POSTGRES third generation database system [24]. We believe that the facilities provided in POSTGRES are mechanistic and at a lower level than the facilities described in this paper. There is no difference between constraints and triggers, all are implemented by the single rules mechanism. All rules always execute as part of the same transaction, potentially leading to very long transactions and hence inefficiency, and also potentially leading to an abort of a triggering update because a semantically independent triggered update is unable to commit. Ariel [14] extends the POSTGRES query language, but continues to suffer from the limitations just discussed.

An important efficiency issue to be discussed in this regard is whether the system should be ‘‘forward-chaining’’ (or ‘‘eager’’) or ‘‘backward-chaining’’ (or ‘‘lazy’’ or ‘‘compute-on-demand’’). Clearly, there are circumstances when one should prefer one to the other. For example, the `age` attribute of every object of type `person` may change every time the clock is updated. It is probably inefficient to compute this attribute for every person in the database, every time the clock is updated, so compute on demand may be preferred. On the other hand, pre-computation and caching may be preferred for some attribute that is referenced often and updated infrequently. As such, any reasonable system must provide both options.

Given that both options are provided, the question is how to choose one or the other. A clever enough optimizer may be able to make this choice, in the absence of any direction from the user. This is the POSTGRES strategy. However, it is not clear that such clever optimizers can be written for a sufficiently general situation. In Ode, virtual attributes (member functions) are used to specify computation on demand, and triggers are used to specify computation on update. If a reasonable optimizer is available, it can alter the choice made by the programmer, by rewriting the program: however, such an optimizer is not assumed in the design of the language or its implementation.

HiPac [11] proposes the concept of Event-Condition-Action (ECA) rules. These rules, unlike our triggers and constraints are first class objects. When an event occurs, the condition is evaluated and, if the condition, is satisfied, the action is executed. Integrity constraints, access constraints, alerters, and other active DBMS facilities can be implemented using ECA rules.

Production systems, such as OPS5, use production rules as the programming paradigm. All rules are active all the time. Whenever the conditions for a rule are satisfied, it fires, and executes its action part. Some good work has been done towards designing large production systems in a database context [23, 27]. The difference between these systems and ours is that in these systems the rules become ends in themselves: the entire program is written in terms of rules. In O++, procedural descriptions may be used where they are appropriate, and rules where they are. Also, in these systems, all rules are active all the time, and every update is an event with respect to each rule.

A significant issue in building rule systems for databases is the ability to have operations execute set-at-a-time rather than tuple (or object) at-a-time. O++ provides special constructs for iteration over sets, and these can be used by the compiler to implement the iterated code on a set-at-a-time basis. Any constraint actions that are generated as a part of the code may also be executed set-at-a-time.

A major difficulty with production systems is that they are very hard to debug when there are situations in which multiple rules can fire, since each has its conditions satisfied. Priority levels, whether explicitly stated as in [14, 24], or determined by criteria such as specificity [23, 27], we believe are a bad idea, since their use is against the declarative spirit of constraint specification and can decrease the potential for concurrent execution. Since we do not specify the order in which the rules will fire, the programmer is forced to make a conservative assumption, and we believe that this makes programming easier.

In an object-oriented context, rules and constraints have been proposed in OOPS [22]. Constraints permit only an undo of the update. Rules are used to enforce integrity constraints and to trigger consequent actions. There is no concept of a transaction.

Triggers have also be proposed for the Iris object-oriented database [21]. Any Iris query can be monitored by first defining it as a function and then defining a monitor for that function. Iris provides functions for defining monitors, activating them, deactivating them, and deleting them. In contrast, triggers in O++ are specified declaratively and as part of the object definition.

OZ+ [26] supports self-triggering rules which correspond to the perpetual triggers of O++. The self-triggering rules are parameterless and they execute whenever possible. Each such OZ+ rule consists of a condition and an action. The OZ+ object manager tests the conditions of the self-triggering rules in objects whenever object states change to see if the triggers can be executed. In Ode, trigger conditions are checked at the end of the execution of member functions.

The issue of efficient implementation of triggers in relational databases has been investigated by several researchers, e.g., [9]. But more investigation and experience with trigger implementations is needed in the context of object-oriented databases. [6] discuss issues in the design of constraint and trigger facilities for programming languages. Two problems identified by them relate to minimization of trigger condition and constraint checking and the interaction between constraints and exception handling. In O++, it is necessary to do trigger condition and constraint checking only at the end of each member function. Even this can be reduced by analyzing the member functions so that only the trigger conditions and constraints affected by the member function are checked. We have not addressed the issue of constraint checking and exception handling explicitly in O++ since C++ (and therefore O++) does not have exception handling facilities. The constraint mechanism is a restricted form of a general exception handling mechanism. It will therefore be reasonably straightforward to integrate these facilities with exception handling facilities such as those proposed for C++ [12].

In Vbase [4] triggers can be attached to attributes and to operations. Triggers have often been used to augment creation and deletion member functions. In O++, we do not find it necessary to use triggers to augment object creation and deletion because users can customize object creation and deletion by defining appropriate constructors and destructors.

9. CONCLUDING REMARKS

We have provided facilities for constraints and triggers in O++ that match the object-oriented programming style of C++. Specifically, we support two kinds of constraints: hard and soft. Hard constraints are checked after each object access while soft constraints are checked just before a transaction commit. We support three kinds of triggers: once-only, timed, and perpetual. Triggers, unlike constraints, must be activated explicitly.

Although constraints and triggers can be implemented using similar techniques, we have provided separate facilities for them since are conceptually and semantically different. The purpose of constraints is to ensure data consistency while that of triggers is to perform actions when some conditions are satisfied. Violation of a constraint leads to the abortion of a transaction, if the violation is not rectified, while the satisfaction of a trigger results in new transactions (assuming the transaction commits). Ode constraints do not have parameters because they are intended primarily to ensure object integrity. Triggers have parameters to allow for trigger firing to take place using user-specified values.

In keeping with the static nature of O++, new constraints and new types of triggers cannot be created without changing the class definition. Adding, deleting or modifying trigger and constraints requires changing the class definition, i.e., it requires changing the database schema. We have not addressed the issue of schema evolution. However, trigger parameters can be altered, and individual triggers can be activated or deactivated at will.

Finally, at this time C++ (and therefore O++) does not support exceptions. When these are added to C++, a statement raising an exception can be specified in the statements associated with constraints. We could then use exception handlers to rectify constraint violations.

ACKNOWLEDGMENTS

We are very grateful to Shaul Dar for his comments and suggestions, and for a careful reading of the manuscript. We appreciated the comments and suggestions of the referees. We also had useful discussions with Rick Greer and Xiaolei Qian.

REFERENCES

[1] R. Abbott and H. Garcia-Molina, ‘‘Scheduling Real-Time Transactions: A Performance Evaluation’’, *Proceedings of the 14th Int’l Conf. on Very Large Databases*, Los Angeles, CA, 1988, 1-12.

[2] R. Agrawal and N. H. Gehani, ‘‘Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++’’, *2nd Int’l Workshop on Database Programming Languages*, Portland, OR, June 1989.

[3] 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, 36-45.

[4] T. Andrews and C. Harris, ‘‘Combining Language and Database Advances in an Object-Oriented development Environment’’, *Proc. OOPSLA ’87*, Orlando, Florida, Oct. 1987, 430-440.

[5] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. F. Putzolu, I. L. Traiger, B. W. Wade and V. Watson, ‘‘System R: Relational Approach to Database Management’’, *ACM Trans. Database Syst. 1*, 2 (June 1976), 97-137.

[6] T. Bloom and S. B. Zdonik, ‘‘Issues in the Design of Object-Oriented Database Programming Languages’’, *Proc. OOPSLA*, Orlando, Florida, Oct. 1987, 441-451.

[7] R. J. Brachman and H. J. Levesque, (ed.), *Readings in Knowledge Representation*, Morgan Kaufmann, 1985.

[8] M. Brodie, ‘‘Specification and Verification of Data Base Semantic Integrity’’, Technical Report CSRG-91, Univ. Toronto, Toronto, 1978.

[9] P. Buneman and E. Clemons, ‘‘Efficiently Monitoring Relational Databases’’, *ACM Trans. Database Syst.*, 1979.

[10] M. Darnovsky and G. Bowman, ‘‘TRANSACT-SQL USER’S GUIDE’’, Document 3231-2.1, Sybase, Inc., 1987.

[11] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ladin, D. McCarthy, A. Rosenthal and S. Sarin, ‘‘The HiPAC Project: Combining Active Databases and Timing Constraints’’, *ACM-SIGMOD Record 17*, 1 (March 1988), 51-70.

[12] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[13] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, ‘‘The Notions of Consistency and Predicate Locks in a Database System’’, *Commun. ACM 19*, 11 (Nov. 1976), 624-633.

[14] E. N. Hanson, ‘‘An Initial Report on the Design of Ariel: A DBMS with an Integrated Production Rule System’’, *ACM-SIGMOD Record 18*, 3 (September 1989), 12-29.

[15] H. V. Jagadish and X. Qian, ‘‘Integrity Maintenance in an Object-Oriented Database’’, *Proc. of the 18th Int’l Conf. on Very Large Databases*, Vancouver, BC, Canada, Aug. 1992.

[16] S. N. Khoshafian and G. P. Copeland, ‘‘Object Identity’’, *Proc. OOPSLA ’86*, Portland, Oregon, Sept. 1986, 406-416.

[17] C. Kung, ‘‘On Verification of Database Temporal Constraints’’, *Proc. of the ACM-SIGMOD Int’l Conf. on the Management of Data*, 1985, 169-179.

[18] C. L. Liu and J. W. Wayland, ‘‘Scheduling Algorithms for Multi-programming in a Hard Real-Time Environment’’, *Journal of the ACM 20*, (Jan. 1973), 46-61.

[19] D. R. McCarthy and U. Dayal, ‘‘The Architecture of An Active Database Management System’’, *Proc. ACM-SIGMOD 1989 Int’l Conf. Management of Data*, Portland, Oregon, May-June 1989, 215-224.

[20] R. S. Nikhil, ‘‘Functional Databases, Functional Languages’’, in *Data Types and Persistence*, M.P. Atkinson, P. Buneman and R. Morrison (ed.), Springer Verlag, 1988, 51-67.

[21] T. Risch, ‘‘Monitoring Database Objects’’, *Proc. 15th Int’l Conf. Very Large Data Bases*, Amsterdam, The Netherlands, Aug. 1989, 445-453.

[22] G. Schlageter, R. Unland, W. Wilkes, R. Zieschang, G. Maul, M. Nagl and R. Meyer, ‘‘OOPS – An Object Oriented Programming System with Integrated Data Management Facility’’, *Proc. IEEE 4th Int’l Conf. Data Engineering*, Los Angeles, California, Feb. 1988, 118-125.

[23] T. Sellis, C. Lin and L. Raschid, ‘‘Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms’’, *Proc. of the ACM-SIGMOD Int’l Conf. on the Management of Data*, Chicago, Illinois, 1988.

[24] M. Stonebraker, E. N. Hanson and S. Potamianos, ‘‘The POSTGRES Rule Manager’’, *IEEE Trans. Software Eng. 14*, 7 (July 1988), 897-907.

[25] B. Stroustrup, ‘‘Multiple Inheritance for C++’’, *Proc. European UNIX User’s Group*, Helsinki, May 1987, 189-208.

[26] S. P. Weiser and F. H. Lochovsky, ‘‘OZ+: An Object-Oriented Database System’’, in *Object-Oriented Concepts and Databases*, W. Kim and F.H. Lochovsky (ed.), Addison-Wesley, 1989, 251-282.

[27] J. Widom and S. J. Finkelstein, ‘‘Set-Oriented Production Rules in a Relational Database System’’, *Proc. of the ACM-SIGMOD Int’l Conf. on the Management of Data*, Atlantic City, New Jersey, 1990, 259-270.

Ode as an Active Database: Constraints and Triggers

N. H. Gehani
H. V. Jagadish

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The Ode object-oriented database provides facilities for associating constraints and triggers with objects. Constraints and triggers are associated with class (object type) definitions which makes them easy to read, to implement, and to blend with object inheritance. In this paper, we state our design goals in providing trigger and constraint facilities for an object-oriented database, describe the constraint and trigger facilities in Ode, their implementation, and illustrate their use. Although triggers and constraints can be implemented by similar mechanisms, we point out the significant conceptual differences. Further, we distinguish between integrity constraints that must be maintained on a transaction basis and those that are inherent to an object and are more appropriately maintained on an object update basis. We also discuss related issues such as intra-object versus inter-object constraints and triggers, coupling modes, order and environment of invocation.