Semantic Integrity Dependencies and Delayed Integrity Checking

Gilles M. E. Lafue

Computer Science Department
Rutgers University
New Brunswick, N.J. 08903

Abstract
This paper's approach to semantic integrity management is that in order to maintain an integrity constraint, some variables of the constraint may be operated on while others may not. This defines integrity dependencies between variables. Various examples of integrity dependencies and their meanings are discussed. In addition to corresponding to real world practice, integrity dependencies can be used to improve the efficiency of checking constraints. This is achieved by delaying the checking (and maintenance) of data which depends on, but does not affect, the data currently operated on. It also gives delayed checking and maintenance a chance to be performed in parallel with applications. Simulation results are presented to support the claim that delayed checking is generally more efficient[1]

1. Introduction

Once semantic integrity constraints have been expressed about a database, managing the database's semantic integrity involves *checking* the constraints after database updates which might violate them, and *maintaining* those which have been found to be violated. Given integrity constraints, the DBMS can automatically check integrity. The DBMS's involvement in maintenance however, is much cruder. It essentially consists either of (i) executing *violation recovery actions* coded by the user for specific violations, or (ii) rejecting database updates which lead to integrity violations, or (iii) ignoring the violations.

The reason for crude automatic integrity maintenance is that in general, there are many ways to maintain a violated constraint. For example, maintaining an instance of the following constraint (expressed in relational tuple calculus)

$$\text{EMPLOYEE}(e) \wedge \text{EMPLOYEE}(f) \wedge f.\text{POSITION}=\text{PRESIDENT} \Rightarrow e.\text{SALARY} \leq f.\text{SALARY}$$

may consist of decreasing the current employee's salary or increasing the president's. It is up to the user, not the DBMS, to choose a solution.

Often however, some solutions are preferred to others, regardless of the context in which they are applied, i.e., of the current database state. In order to maintain the above constraint for example, the enterprise's policy may be to always increase the president's salary. Such preferences can indicate the order in which maintenance solutions are to be tried, or that some solutions must be discarded. In this paper, we shall only consider the simpler case that in order to maintain a given constraint, some solutions are possible and others impossible, that is, some variables may be operated on and others not. This defines *dependencies* between the constraint's variables: If a variable cannot be operated on to maintain the constraint, it is called an *independent* variable of the constraint (its integrity is not affected by the constraint). If a variable can be operated on to maintain the constraint, it is called a *dependent* variable of the constraint. The existence and/or value of a constraint's dependent variables depend on the existence and/or value of the constraint's independent variables. Of course, a variable may be dependent in a constraint and independent in another. We shall call these dependencies *integrity dependencies*.

Such dependencies are explicit in other areas of computer science. In abstract data types for instance, one can include in a type definition a constraint which involves an attribute exported from another type, and this other type is unaffected by its use in the constraint. Knowledge representation systems in Artificial Intelligence often provide this feature too, particularly with generalization hierarchies, in which the attributes of a specialization can depend on those of its generalization, but not vice versa.

If available, user coded violation recovery actions can embed integrity dependencies. However, there are several advantages in explicitly declaring integrity dependencies to the DBMS:

- Their declarations would enrich database schemas, especially if they are straightforward and integrity dependencies are context independent. This paper is not concerned with the syntax of such declarations but assumes it to be as simple as, say, specifying key attributes of relations in relational DBMSs.
- Since integrity dependencies limit the combinatorial explosion of maintenance solutions, DBMSs could become more active partners in integrity maintenance. For example, they could automatically carry out solutions which are unique, or present the user with solutions and estimates of the extent of their propagation through the database. Speculations regarding this are presented in [8].

[1] This work has been partly supported by the National Science Foundation, grant no. MCS-81-10100.

- The DBMS can take advantage of these dependencies to improve the efficiency of automatic integrity checking. This is the emphasis of the paper.

This paper bases its approach to automatic integrity checking on integrity dependencies as follows. A constraint need not be checked immediately after operations on instances of its independent variables, if the corresponding instances of its dependent variables are of no interest to the current user/applications. Instead, this checking can be *delayed* until the dependent instances become of interest, i.e., are accessed, since they are the only instances to be affected by this constraint. This approach, which was described in [7], is called *delayed integrity checking (and update propagation)*, as opposed to the more traditional *immediate integrity checking (and update propagation)*.

For example, suppose a constraint in a database for building architecture that every horizontal pipe must be supported by a beam, and that the pipe's location is constrained by the beam's, and not the opposite (i.e., the pipe is the dependent variable). When a structural engineer wants to move or delete a beam, he doesn't care about the pipes that the beam may support, and wouldn't want to be stopped just because some pipes may violate integrity as a result, especially if the mechanical engineer in charge of piping is not currently active. This integrity violation becomes important only when the pipes, or other things depending on them, are accessed, at which point the effects of the beam update must be checked.

Delayed integrity checking makes sense only if integrity dependencies can be defined. It also assumes (i) that cycles of dependencies are localized in the database and (ii) that the history of data values is not important. It is based on integrity dependencies defined on static constraints (i.e., which regulate states of the database) rather than on dynamic constraints (i.e., which regulate transitions between states).

Due to the above assumptions, delayed checking fits some databases better than others, and as we shall see, *Computer-Aided Design (CAD) databases* are prime candidates. A CAD database is a database which supports the design of a complex artifact (e.g., a building, an aircraft, a complex digital circuit,...), that is, an evolving model of this artifact, from high level specifications down to low level implementation details. Integrity management is particularly crucial to CAD databases due to their frequent updates [4]. The examples in this paper come from simple design situations.

In addition to corresponding to real world practice, e.g., in CAD, delayed integrity checking can significantly improve checking efficiency. This paper shows simulation results which support this claim. Efficiency is measured in terms of numbers of accesses to the database on secondary memory. Such accesses are commonly assumed to be by and large the major cost of database integrity management. Much research has aimed at improving checking efficiency [12], [5], [10], [3], [1], [2], [6]. Delayed checking is orthogonal to these approaches in that it can be combined with them.

The next section develops examples, and section 3 expands on the notion of delayed integrity checking. Section 4 outlines possible implementations for supporting delayed checking. Section 5 is a general comparison of immediate and delayed checking. Section 6 presents the simulation model, and section 7 interprets simulation results. The conclusion assesses this approach for CAD databases.

2. Examples of Semantic Integrity Dependencies

The following example is borrowed from [9] and [8]. It shows a CAD database used by an Expert System for understanding and modifying digital circuits. The data not only represents the components and connections which make up a circuit, but also the *reasons* why these components and connections are there, i.e., their *roles* in the circuit. The role of a component is to (partly) *implement* a *specification*. In turn, that specification may implement another higher-level specification. The whole database is therefore a hierarchy (not necessarily a tree) of *modules*, in which a module implements the module(s) above it, and is implemented by the module(s) underneath it. This is often called a *design (or implementation) hierarchy*.

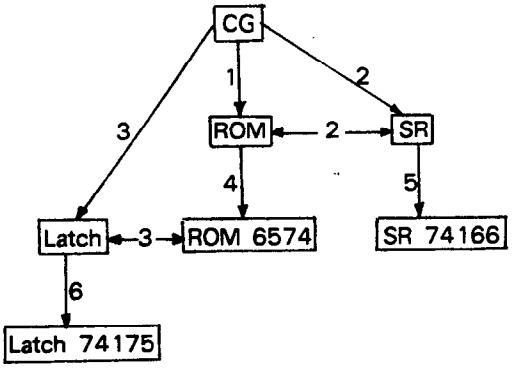


Figure 2-1: Design hierarchy for the CG

Figure 2-1 shows four levels of specification-implementation modules for the Character Generator (CG) of a computer video terminal. The CG is the circuit which converts input ASCII codes into the dot matrices to display on the screen. For each ASCII character and matrix row, it outputs a string of bits. Its specification defines its interface with the rest of the circuit. That is, (i) it is a function from a set of <character, row index> pairs to a set of bit strings (i.e., the font function), (ii) output bit strings are to be encoded serially, and (iii) the input characters and row indexes are held for at least 50 nanoseconds.

The CG is implemented with a Read Only Memory (ROM) whose contents represents the font function, and a shift register (SR) to transform the parallel output of the ROM into a serial bit string. Then, a particular ROM (no. 6574) and a particular SR (no. 74166) are chosen, and a latch is inserted to hold the ROM 6574 input. The latch is implemented with a latch 74175.

The data of a module represents what is known about the module. For instance, the CG module contains its specification, and the ROM module contains a description of what is common to all ROM's, e.g., that its output is parallel encoded.

What enables the implementation of modules is a set of rules which are called *implementation rules* and represent general knowledge about circuit design. For example, figure 2-2 shows the (simplified) rules used in figure 2-1.

1. If the specification is a function that alters data values then use a ROM whose contents is the table definition of the function or a network of random logic.

2. If the specification is a parallel encoded bit string and the current bit string is serial then use a SR whose input bus size is equal to the input string length and has the appropriate load input and clock input.

3. If the specification is a sequence of bit strings such that each string lasts until the next string and the strings of the current sequence do not last this long then use a latch whose input bus size is equal to the sequence length and has the appropriate clock input.

4. If the specification is a ROM then use a ROM 6574 or a EPROM 2716 or ...

5. If the specification is a SR then use a SR 74166 or a SR 74165 or ...

6. If the specification is a latch then use a latch 74175 or a latch 74171 or ...

Figure 2-2: Implementation rules for the CG

At any point in the database development, the application (or checking) of an implementation rule may reveal its violation, i.e., that its condition is satisfied, but not its consequence. For instance, the insertion of the CG specifications and of the ROM satisfy the condition of rule 2. If checking the rule reveals the lack of a SR satisfying its condition, then maintenance consists of inserting the appropriate SR.

Implementation rules can therefore be seen as integrity constraints. In addition to these rules which relate modules of different levels, there can also be integrity constraints which apply to modules of the same level. For example, it may be stated that no component output can be connected to more than 10 other component inputs.

Edges in figure 2-1 represent instantiations of implementation rules, as identified by the edge labels which refer to the rules of figure 2-2. A rule can be instantiated by several edges if it involves more than two modules, e.g., rules 2 and 3.

Edge directions represent integrity dependencies between modules. For example, the edge from the CG to the ROM module indicates that in rule 1, the ROM depends on the function specification (here, the one specified in the CG). The edges for rule 2 indicate that the independent variable in that rule is the desired serial bit string (here, the one specified in the CG), and the dependent ones are the current parallel bit string (here, the one ouput by the ROM) and the SR. This means that neither the ROM nor the SR has priority over the other as implementation of the CG. If the current parallel bit string was independent, the edge between the ROM and the SR module would go only from the former to the latter, indicating the priority of the ROM.

Thus, in general, integrity dependencies cannot be implied from the logical structures of constraints, but are defined "on top" of constraints. The meanings of integrity dependencies depend on the meanings of the constraints to which they apply. Consider again the example constraint about beams supporting pipes. Declaring the pipe variable dependent and the beam variable independent means that beams cannot be moved or created in order to support pipes (they have another purpose). This is usual in housing architecture where the building structure determines the piping. Now, if the beam variable is declared dependent and the pipe variable independent, then a beam can be created or updated especially for supporting a pipe. This would indicate a dominant piping system, as in chemical process plants for instance[2].

Figure 2-1 illustrates a major reason why CAD databases are good candidates for delayed checking. The graph of integrity dependencies is a hierarchy (the design hierarchy), i.e., cycles of dependencies are confined within layers, thus satisfying the assumption about localized cycles of dependencies.

3. Delayed Integrity Checking

The following terminology will be used in the rest of the paper. We shall refer to a *record* as data which corresponds to an instance of a constraint variable and is swapped as a unit between core and secondary memory (e.g., a tuple in relational databases). If A is a record which constrains another record B because A is an instance of a constraint's independent variable and B is an instance of one of the constraint's dependent variables, then A is called B's *parent* in this constraint, and B is called A's *child*. Parents and children are linked by directed integrity dependencies. Instances of dependent variables related in the instantiation of a constraint are called *siblings*. Siblings depend on each other and are linked by undirected integrity dependencies[3].

The general principle of delayed integrity checking is that the integrity of a record is checked only when the record is in core, the current focus of attention of some application(s). In other words, integrity is checked only when strictly necessary. A given record update, insertion or deletion is not immediately propagated to the record's children since these children do not affect the record. Therefore, this operation must be recorded somewhere so that when the children are accessed, they can be notified. This is done by sending to a *bulletin board* a *message* consisting of a description of the record, the nature of the operation and a timestamp. In addition, the effects of operations on parents can be propagated to children in parallel with applications execution.

Deleting a record first requires accessing the record. Then, it is checked whether this record can be deleted without violating a constraint in which it is dependent. A successful

[2] Future work includes extending the notion of dependency to resource protection. In this paper, integrity dependencies are expressed in terms of constraint variables. However, to be more realistic, they should also take into account the users or applications that operate, or can operate, on the variables. This is best illustrated in the case of existentially quantified dependent variables. Suppose that in the above example constraint, the pipe has been declared independent and the beam dependent. A pipe can be created or updated regardless of whether there is a beam to support it, and maintenance consists of creating or moving beams. The responsibility for creating beams or choosing which existing beams to move is more meaningful if it applies to the users authorized to perform such actions rather than to the current beam records.

[3] From the viewpoint of integrity checking, one can also consider records involved in a cycle of directed dependencies as siblings of each others although they may appear in different constraints, since each one depends on the others, however indirectly.

deletion results in sending a message which identifies the deleted record as a (current) parent.

When a record is inserted, its parents and siblings are identified in the database and its initial value is checked against these parents and siblings. When the checking is positive, the record is inserted into the database, and a message is sent which identifies the new record as a (potential) parent.

For simplification, a record update will be modelled as one deletion followed by one insertion. This paper will ignore the transaction issue that the intermediate checking after the deletion can be useless due to the fact that the insertion follows.

When a record is accessed from the database, messages sent more recently than the record's last access, and indicating the deletion of current parents or the insertion of potential parents, are looked for in the bulletin board. If such an insertion message is found, the parent identified by the message is accessed to check whether its value affects the accessed record. Thus, accesses can be triggered recursively. If an inserted or deleted parent implies deleting or updating the accessed record, the application is prompted to do so. Records carry the timestamp of their last access.

Records can be accessed either by applications or by a special DBMS component which accesses records precisely to check their integrity with respect to recent operations on their parents. This DBMS component is called the *flusher* because it flushes messages from the bulletin board. The flusher executes in parallel with applications. In general, it accesses records with lower priority than applications. Under certain circumstances however, it may gain higher priority; for instance, if an application wants to check the consequences of an update[4]. A message is deleted when it has been "seen" by all the children of its record which existed at the time of its insertion[5].

4. The messages and the bulletin board

Although the main purpose of this paper is not the implementation of delayed integrity checking, the outline of implementation strategies for the bulletin board presented in this section may help the reader get a clearer appreciation of the approach and of the simulation results shown later on.

The bulletin board can be implemented as a structure distinct from the database itself. To the extent that messages represent database updates, this resembles a differential file [11] which is dynamically cleared by the flusher.

An alternative is to intermingle the bulletin board and the database implementation. Consider for instance, a relational database implemented as tables or files (one per relation) with indexes for some attributes. Every relation has three additional attributes, each indexed on: insertion time (IT), deletion time (DT), and access time (AT). The bulletin board now consists of the IT and DT indexes and possibly, indexes for some other attributes. Suppose for example the following relational schema:

BEAM (END-PT1, END-PT2, X-SECTION)
PIPE (END-PT1, END-PT2, DIAMETER)

Suppose further that (horizontal) pipes must be supported by beams and depend on the beams which support them. When a pipe p_1 is accessed, recently deleted beam tuples are checked to see whether they used to support p_1. These beam tuples are identified as:

$$\{b \mid BEAM(b) \wedge b.DT > p_1.AT\}$$

An important question concerns the choice of attributes to include in a message, that is, in the second implementation alternative, the attributes to index for integrity checking purpose. This question is illustrated here using the second implementation alternative, because of its impact on checking efficiency as discussed later in the simulation model. In the above example, the deleted beams to consider can be further selected as those which were close enough to the pipe p_1 to possibly support it. This requires an index, say, for END-PT1:

$$\{b \mid BEAM(b) \wedge b.DT > p_1.AT \wedge distance(b.END\text{-}PT1, p_1.END\text{-}PT1) < 100\}$$

The trade-off is between an earlier selection (i.e., in the bulletin board, or indexes) but more indexes to maintain, versus a later selection (i.e, in the database).

5. General comparison of immediate and delayed checking

A major argument in favor of delayed integrity checking is that it often corresponds to real world practice. For example, by decoupling the treatment of specifications and of their depending implementations, designers can concentrate on the former with no immediate concern for the latter. Purposes of multi-purpose objects can be similarly decoupled, as illustrated by the example of beams used primarily for structural purpose, and secondarily for supporting pipes.

Delayed checking assumes tolerance of temporary integrity violations. The integrity of a record may be violated by operations on its parents, and left unchecked until the record is accessed. Another case of undetected violations may occur due to the fact that records are directly sensitive to changes in their parents only, not in their further ancestors. For example, suppose beams depend on the columns which support them. Suppose further that a column is deleted, and then a pipe supported by a beam which is itself supported by that column, is accessed before the beam. The pipe does not know that its supporting beam is no longer supported. Special provisions for sensitivity to changes in ancestors must be offered if specifically requested. They are related to flushing activities, but no solution will be outlined here.

Delayed checking redistributes part of the cost of integrity checking incurred by applications from update time to

[4] If requested immediately after the update, this is equivalent to immediate propagation, which therefore, can be implemented as a special case of delayed checking.

[5] If in some implementations, it is significantly cheaper to compute the number of such children than to access them (e.g., by maintaining redundant cardinality data), then a counter could be associated with each message and initialized to the appropriate number of children. Then, for every child which "sees" the message, whether it is accessed by applications or by the flusher, the counter is decremented, and the message is automatically deleted when the counter reaches zero.

access time. In addition, it can save applications some checking at access time if this checking has been performed in parallel with them by the flusher. In the best case of delayed checking, children checking is all done in parallel, and in the worst case, it is all done serially with applications.

Now, consider the expected numbers of database (and bulletin board) accesses for integrity management, regardless of when these accesses take place. A disadvantage of immediate checking is that a record can be accessed and modified several times, due to several modifications in its ancestors, before any application is interested in it. All but the last modifications will be ignored, and are therefore useless (unless history is a concern, of course). On the other hand, delayed checking requires as many accesses to an updated record as this record has children. Furthermore, it incurs the cost of accessing and inserting messages. Neither alternative appears superior a priori.

6. Simulation of delayed and immediate checking

This section presents the simulation model used to compare the efficiency of delayed and of immediate integrity checking in terms of numbers of database accesses.

6.1. Modelling the database operations

For the sake of simplicity, the results reported here concern only record accesses and updates. Equivalently, this consists of modelling all database updates as record updates. Record updates have been chosen because they are the costliest database updates (again, they can be seen as pairs of record deletions and insertions).

It is assumed that integrity checking is left entirely to the DBMS, and that applications maintain integrity as they are notified of violations by the DBMS. Accordingly, two sorts of database operations are distinguished. *Spontaneous* operations are performed by applications regardless of integrity, and *integrity* operations are performed for integrity management. Integrity database accesses are performed by the DBMS, and integrity database updates by applications.

The simulation aims at estimating the average number of database and bulletin board accesses for integrity management per spontaneous access and update. A record access costs one database access, and so does a record update (for the record's re-insertion). Similarly, a message lookup or insertion costs one bulletin board access. If database and bulletin board accesses concern sets of records rather than individual records, then the cost of accesses is assumed to be proportional to the size of their sets. In order to avoid core size as a modelling parameter, it is assumed that a record is accessed from the database at most once, and (re-)inserted at most once, due to a spontaneous access. The same assumption holds for message lookups and insertions. See figure 9-9 for the simulation recursive model of delayed checking operations.

One parameter of the simulation, called RSU, is the rate of spontaneous updates, i.e., the proportion of spontaneous record accesses to be followed by an spontaneous update, (RSU$\leq$1). These spontaneous updates take place whether or not the records are found to violate integrity at access time with delayed checking. Spontaneous accesses and updates are performed independently of each other, and are uniformly distributed over the database, as if many different applications interacted with the database simultaneously.

Two cases are considered regarding parallel checking with delayed checking. In the best case, messages from parents are searched for at access time, but none is found which has not been checked. In the worst case, all the parents which have been updated since the last access to the current record, must be accessed.

6.2. Modelling databases

Databases are abstracted as graphs where nodes are records and edges integrity dependencies. A database is the superposition of a directed graph of parent-child dependencies, the *digraph*, and of undirected graphs which connect siblings, the *sibling graphs*. The digraph is a hierarchy, i.e., an acyclic digraph in which layers are distinguished. It can be for instance, a design hierarchy. There are as many sibling graphs as there are parents. A *layer graph* is an undirected graph made of all the sibling graphs of the same layer. Graphs in the simulation are static, i.e., they do not grow or shrink, which is consistent with the modelling of database updates as record updates. Simulation was conducted on graphs made of several hundred nodes and five or six layers.

The graphs of a database are characterized by the following parameters (which are not all independent):

ADD: avg. in-degree of the digraph nodes.
ADS: avg. degree of the sibling graph nodes.
ADL: avg. degree of the layer graph nodes.
ECL: edge connectivity of the layer graphs.
PUP: probability of update propagation from a parent to a child or between 2 siblings. PUP is the same for all edges.

ADD is the average number of parents, or of children, per record, i.e., for delayed checking, it is the average number of messages looked for at access time. In the simulation runs, the sibling graphs were either non-existent (ADS=0), or rings, in which case every child is linked to exactly two of its siblings if it has at least two siblings (ADS=2), or cliques, in which case every child is linked to all its siblings (ADS approaches ADD-1). Both ADL and ECL depend on ADD and ADS. If ADD$\leq$1, the sibling graphs do not intersect, and ADL=ADS. If ADD$\geq$1 and ADS$\geq$2, the sibling graphs intersect, the layer graphs become connected, i.e., ECL$\geq$2, and ADL>ADS.

7. Simulation results

Some simulation results are plotted in figures 9-1 to 9-8. They show the average costs of delayed checking as percentages of the corresponding average costs of immediate checking. Theses costs are averaged per spontaneous *event*, i.e., spontaneous access or update. That is, they are computed as (x+(y*RSU))/(1+RSU), where x is the average number of database (or bulletin board) accesses per spontaneous access, and y is this average number per spontaneous update. Both the best and the worst case of delayed checking are represented. The heavy lines represent database accesses, and the lighter lines bulletin board accesses. The bulletin board accesses are added to the database accesses.

A bulletin board access is considered as expensive as a database access, and all database accesses are considered equally expensive. These two simplifications are generally more detrimental to delayed checking than to immediate checking. Firstly, if the bulletin board is a distinct structure from the database, it is likely to be smaller than the database, since a message is at most as long as the record it represents (neglecting the timestamp), and there are at most as many messages as there are, or have recently been, records. It is generally assumed that the expected time to access a (paged) structure is proportional to the size of this structure. Now, if the bulletin board is part of the database access structure (see section 4), then the cost of accessing the bulletin board is included within the cost of accessing the database. Plotting cheaper bulletin board accesses would result in pushing the light lines down towards their corresponding heavy lines. Secondly, if the bulletin board and the database share their access structure, databases accesses of delayed checking which are prompted by a bulletin board search are faster than others, since most of the job was done for accessing the bulletin board.

The plots show sensitivity to the five parameters which describe the databases and the way they are manipulated: ADD, ADL, ECL, PUP and RSU. The amount of integrity to manage grows with the value of each of these parameters. For each parameter, first is considered the case where the layer graphs are not connected, and then the case where they are.

The only way ADL and ECL can keep constant while ADD varies is if ADS=0 (figure 9-1). As the number of parents per record grows, the increase in useless checks and updates of descendants for immediate checking is faster than the increase in repeated accesses to updated parents for delayed checking, but about as fast as the increase in messages for the worst case of delayed checking.

In order to help interpret the plots in which the digraph is a tree and ADS$\geq$2 (figures 9-2, 9-3 and 9-4), consider the maximum subgraph which can be accessed following a spontaneous access or update to a given node. For delayed checking, it is made of the siblings hanging on the path from that node to the root. Its size is NC*A, where NC is the number of children of the nodes which *have* children, and A is the number of layers from this node to the root. For immediate checking, the maximum subgraph size is $\sum_{i=1}^{D} NC^i$, where D is the number of layers from this node to the leaves.

An increase in ADL means an increase in NC, and immediate checking is generally more sensitive to that than delayed checking (figure 9-2). If delayed checking costs do not decrease more sharply, it is because as NC increases, the average number of ancestor layers per node slowly increases while the average number of descendant layers per node slowly decreases. As PUP increases (figure 9-3), more of the maximum subgraphs is likely to be accessed. As RSU increases (figure 9-4), immediate checking accesses the subgraphs more often.

Basically, immediate checking is more sensitive to parameter increases, and the cost of delayed checking is the closest to that of immediate checking for low parameter values, that is, when integrity is cheap to manage anyway.

Connected layer graphs make things more complicated because of the dependence of ADL and ECL on ADD. The essential novelty is that the maximum subgraphs are now made of entire layers. As a result, delayed checking immediately propagates updates to more records at each layer, and the difference with immediate checking weakens. Thus, the two alternatives are more similarly sensitive to parameter increases, as can be seen from figures 9-5, 9-6 and 9-8.

Figure 9-7 which shows delayed checking generally cheaper than immediate checking except when PUP tends to 0 or 1, can be explained as follows. When PUP tends to 0, updates hardly propagate and the two alternatives are hardly different. When PUP tends to 1, delayed checking tends to propagate integrity checks and updates throughout the whole graph from the spontaneously updated records up, and immediate checking throughout the whole graph from the spontaneously updated records down.

8. Conclusion

Specifying integrity dependencies between the variables of integrity constraints can improve checking efficiency by delaying some of the checking and giving it a chance to be performed in parallel with applications. Delayed checking is cheaper than immediate checking in database accesses, and often cheaper even when the number of bulletin board accesses is added to the number of database accesses. This is especially encouraging as several aspects of delayed checking have been considered under the worst light. Again, the cost of a bulletin board access could be reduced to a fraction of the database access cost, and the average database access is cheaper for delayed checking if the bulletin board and the database share their access structure.

The simulation can help characterize the databases which are good candidates for delayed checking. The higher their update rate, the better. Their average probability of update propagation, i.e., the "tightness" of their integrity constraints, should be high enough to make integrity management significant, but not to the point that everything completely determines everything else. Also, it seems that delayed checking applies particularly to trees of directed integrity dependencies, although it performs comparatively well for other acyclic digraphs too.

CAD databases match closely these characteristics. They are frequently updated, since their purpose is primarily to be built, as opposed to more static databases which are primarily consulted. The elements of a complex design are often tightly related, but not completely determined by each other (otherwise design would not be needed). Finally, trees usually make life easier for designers. They allow reasoning at various levels of abstraction (i.e., specification) and top-down design, because the role of every design element as a specification implementor is known, and its implementation is not affected by another. In reality of course, the role of elements is not always known or expressed, and optimization sometimes consists of making elements implement several specifications.

9. Acknowledgements

The author is grateful to Alex Borgida and Tom Mitchell for many fruitful discussions and useful comments about this paper, and to the VLDB referees for their suggestions.

References

1. Bernstein P., Blaustein B., Clarke E. Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data. Sixth International Conference on Very Large Data Bases, , 1980.

2. Blaustein B. *Enforcing Database Assertions: Techniques and Applications.* Ph.D. Th., Aiken Computation Laboratory, Harvard University, 1981.

3. Buneman O.P., Clemons E.K. "Efficiently Monitoring Relational Databases." *ACM Transactions on Database Systems 4*, 3 (Sept. 1979).

4. Eastman C., Lafue G. Semantic Integrity Transactions in Design Databases. Working Conference on CAD Databases, IFIP, 1981.

5. Hammer M., Sarin S. Efficient Monitoring of Database Assertions. International Conference On Management of Data, ACM/SIGMOD, 1978.

6. Koenig S., Paige R. A Transformational Framework for the Automatic Control of Virtual Data. Seventh International Conference on Very Large Data Bases, , 1981.

7. Lafue G. An Approach to Automatic Maintenance of Semantic Integrity in Large Design Databases. National Computer Conference, AFIPS, 1979.

8. Lafue G.M.E., Mitchell T.M. Data Base Management Systems and Expert Systems for CAD. Working Conference on CAD Systems Frameworks, IFIP WG 5.2, 1982.

9. Mitchell T. et al. Representations for Reasoning about Digital Circuits. Seventh International Joint Conference on Artificial Intelligence, 1981.

10. Nicolas J.M. Logic for Improving Integrity checking in Relational Data Bases. ONERA-CERT, Toulouse, France, Feb., 1979.

11. Severance D., Lohman G. "Differential Files: Their Application to the Maintenance of Large Databases." *ACM Transactions on Database Systems 1* (Sept. 1976).

12. Stonebraker M. Implementation of Integrity Constraints and Views by Query Modification. International Conference On Management of Data, ACM/SIGMOD, 1975.

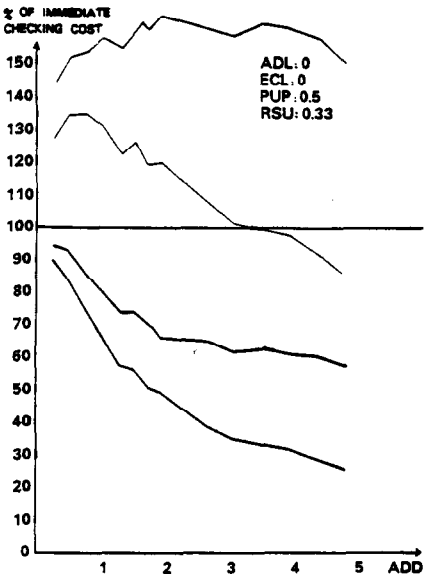


Figure 9-1: Delayed checking costs vs. ADD for non-connected layers

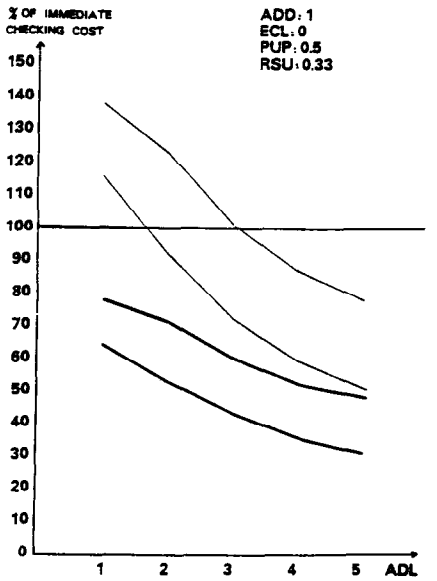


Figure 9-2: Delayed checking costs vs. ADL for non-connected layers

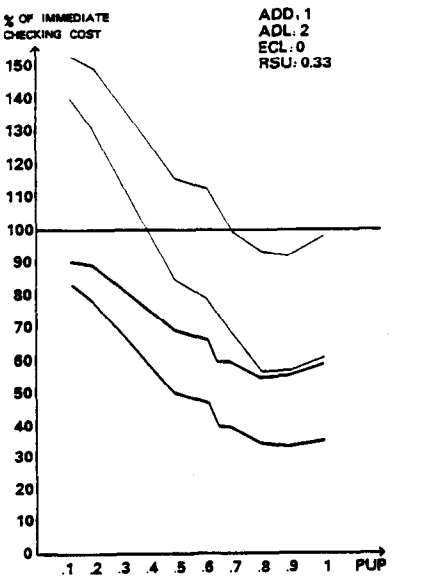
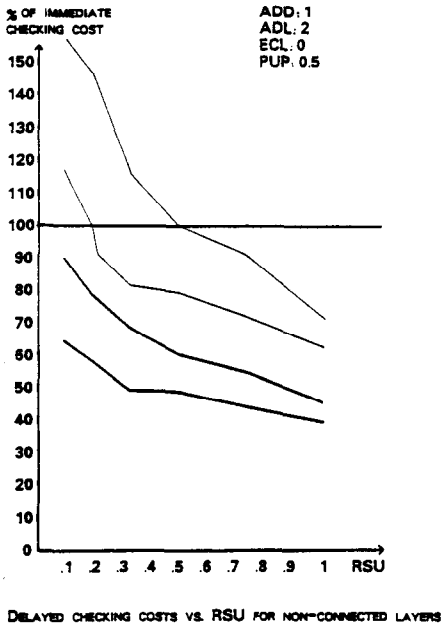
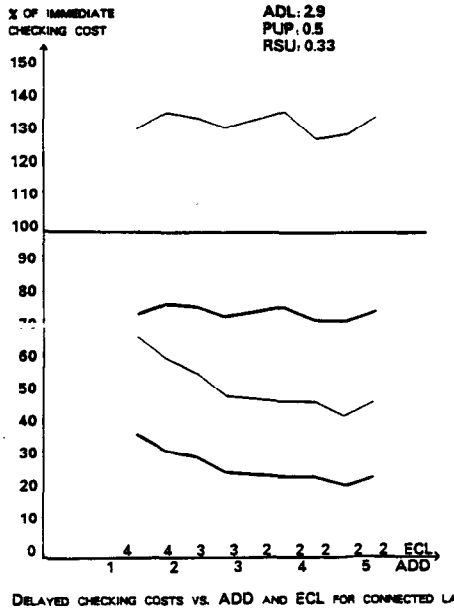


Figure 9-3: Delayed checking costs vs. PUP for non-connected layers



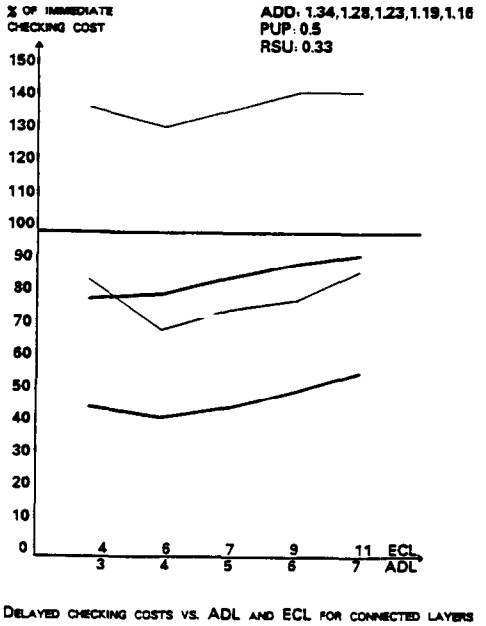
DELAYED CHECKING COSTS VS. RSU FOR NON-CONNECTED LAYERS

Figure 9-4:



DELAYED CHECKING COSTS VS. ADD AND ECL FOR CONNECTED LAYERS

Figure 9-5:



DELAYED CHECKING COSTS VS. ADL AND ECL FOR CONNECTED LAYERS

Figure 9-6:

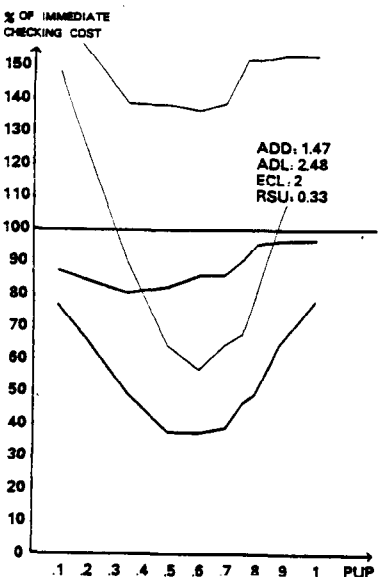
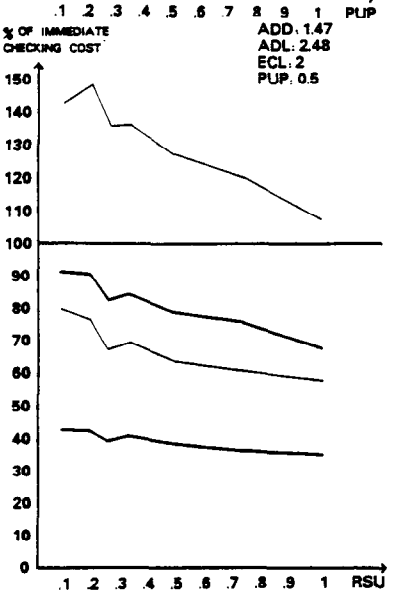


Figure 9-7: DELAYED CHECKING COSTS VS. PUP FOR CONNECTED LAYERS



DELAYED CHECKING COSTS VS. RSU FOR CONNECTED LAYERS

Figure 9-8:

```
access(r)= if a(r) ≤ t0 and u(r) ≤ t0
           then ( for every parent(r) do
                  ( if m(parent(r)) ≤ t0 then ( m(parent(r)):=t;
                                                b:=b+1 );
                    if u(parent(r)) > a(r) then ( access(parent(r));
                                                  if random ≤ PUP then update(r) ) );
                  a(r):=t:=t+1 );

update(r)= if u(r) ≤ t0
           then ( u(r):=t:=t+1;
                  for every parent(r) do access(parent(r));
                  for every sibling(r) do
                  ( access(sibling(r));
                    if random ≤ PUP then update(sibling(r)) );
                  b:=b+1 );

spontaneous access to record r: t0:=t; access(r);
spontaneous update of record r: update(r);
```

where:

- t is time. It is incremented for each record access or update.
- t_0 is the time at which the current spontaneous access has started.
- b is the number of bulletin board accesses.
- a(r) is the time of last access to record r.
- u(r) is the time of last update of record r.
- m(r) is the time of last lookup for the message representing record r.
- *random* generates random numbers between 0 and 1.

After a certain number of spontaneous accesses and updates, the number of database accesses is t, and the number of bulletin board accesses is b.

Figure 9-9: *Recursive model of record access and update for delayed checking*