Concurrency Control for Knowledge Bases

Vinay K. Chaudhri vinay@ai.toronto.edu
Vassos Hadzilacos vassos@db.toronto.edu
John Mylopoulos jm@ai.toronto.edu
Department of Computer Science, University of Toronto
Toronto, M5S 1A4, Ontario, CANADA

Abstract

As the demand for ever-larger knowledge bases grows, knowledge base management techniques assume paramount importance. In this paper we show that large, multi-user knowledge bases need concurrency control. We discuss known techniques from database concurrency control and explain their inadequacies in the context of knowledge bases. We offer a concurrency control algorithm, called the Dynamic Directed Graph (DDG) policy that addresses the specific needs of knowledge bases. The DDG policy exploits the rich structure of a knowledge base to support the interleaved, concurrent execution of several user requests, thereby improving overall system performance. We give a proof of correctness of the proposed concurrency control algorithm and an analysis of its properties. We demonstrate that these results from concurrency control interact in interesting ways with knowledge base features and highlight the importance of performance-oriented tradeoffs in the design of knowledge-based systems.

1 INTRODUCTION

> *Very large knowledge-based systems will soon be commonly upon us. With this, issues that have occupied the database world will come to concern KR developers, although perhaps complicated in interesting ways by the logical interpretation of KR languages.*
> – Ron Brachman, AAAI-90 Invited Lecture.

As we build ever larger knowledge bases, it is reasonable to expect that the time will soon arrive when it will no longer be viable or desirable to maintain multiple copies of the same knowledge base for each one of its users, nor will it be economically feasible to restrict access to the knowledge base to one user at a time. Instead, it is expected that multiple users will share a single knowledge base which receives queries and updates and interleaves their execution against the knowledge base, thereby optimizing the deployment of computing resources, both CPU cycles and space.

To make the problem more concrete, consider the construction of a large knowledge base, part of which is stored in the primary storage and the rest in the secondary storage[1]. If query/update requests from users are processed sequentially, the system will remain idle while waiting for a disk access to complete (see Figure 1(a): dark dots and vertical lines correspond to the requests of two different users). However, if requests are processed in an interleaved fashion, i.e., concurrently as suggested in Figure 1(b), the idle periods can be reduced resulting in a higher system throughput[2] and a quicker response time to the user (see Figure 2). This improvement in throughput or response time will grow with the number of users until resources available to the system become saturated[3]. Beyond this point, interleaved execution of user requests does nothing to enhance the system performance and may cause it to deteriorate because of the overhead of concurrency control.

In addition to improving the performance of a system, concurrency promotes sharing of knowledge, which is expected to be increasingly important in the future (Neches et al. 1991).

In the present paper, we concentrate on the problem of concurrency control. We propose a concurrency control algorithm that suits the requirements of knowl-

[1] It will not be possible to fit a large knowledge base such as CYC (Lenat and Guha 1989) in main memory.

[2] Measured by the number of user requests executed on average per unit of time.

[3] Concurrent processing can lead to arbitrary improvements in performance — sometimes of the order of *one hundred times* (Gray 1992). Even when there are no disk waits, performance studies indicate that a round robin scheduling discipline that allows interleaved processing of user requests gives best overall performance (Lazowska et al. 1984).

Appears in the Proceedings of the Third International Conference on Knowledge Representation and Reasoning, October 25-29, 1992, Cambridge, MA.

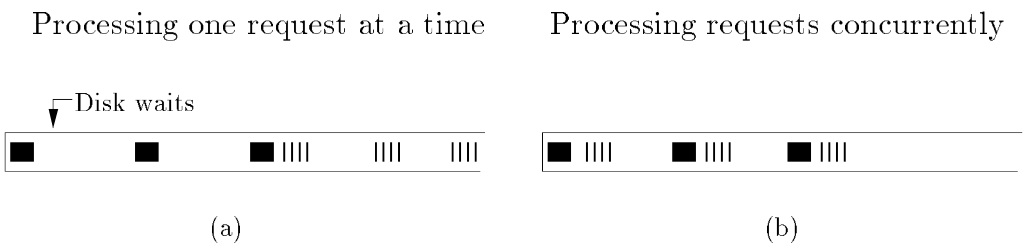


Figure 1: Concurrent processing reduces idle times

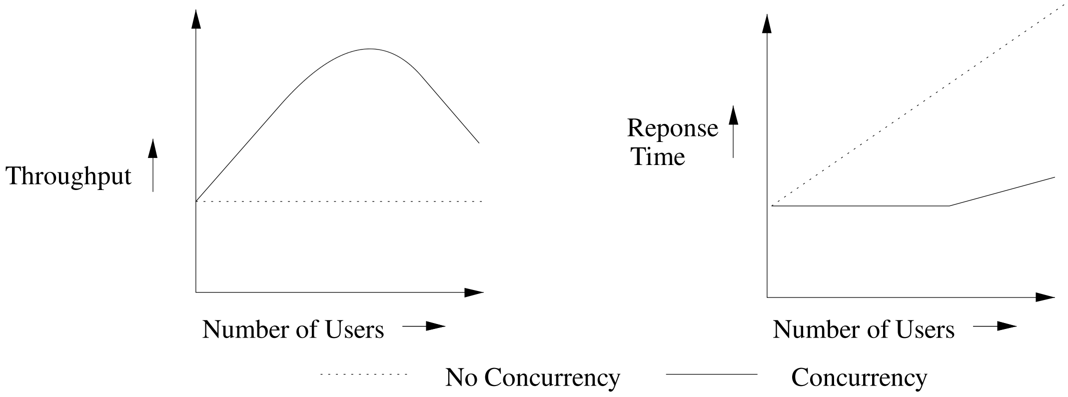


Figure 2: Concurrent processing improves performance

edge bases. We analyze its correctness and discuss how the interaction between the design of this algorithm and knowledge base features.

The organization of the paper is as follows: Section 2 contains a quick overview of concurrency control algorithms for databases and difficulties that arise in applying them to knowledge bases. In Section 3, we present a new algorithm for concurrency control for knowledge bases. We conclude the paper in Section 4 with a discussion on related work, the impact of concurrency control considerations on the design of future knowledge-based systems and a summary of the paper.

2 THE CONCURRENCY CONTROL PROBLEM

In this section, we give an example knowledge base, describe the problems caused by arbitrary interleaving and the concurrency control problems that need to be solved for knowledge bases.

2.1 AN EXAMPLE KNOWLEDGE BASE

For the purposes of this paper,[4] knowledge bases are assumed to support an object-oriented representational framework with an assertional sub-language used for both deductive rules and constraints. Also, possibly, they might support facilities for representing special kinds of knowledge (for example, temporal knowledge, incomplete knowledge, *etc.*). Where we need to talk about a specific knowledge representation notation, we will be using the language Telos (Mylopoulos et al. 1990).

As an example, consider a knowledge base that has a class `Employee` which is a specialization of class `Person`. The class `Employee` has attributes `Name`, `Manager` and `Salary`. Each `Salary` attribute indicates the salary of an employee during a certain time interval (for example, `before 1989`). `Emp01` is an instance of `Employee` and is an identifier for the employee `Adam` whose `Manager` is `John`. The salary of `Adam` takes two different values in the history — `20000` during time interval `I1` and `30000` during time interval `I2`. These intervals in the history, `I1` and `I2`, are left unspecified. There is an integrity constraint that requires that the value of the salary must always increase and, therefore, we can infer that `I1` must precede `I2`. In addition, it is assumed that the the knowledge base contains the following deductive rules:

`DR1`: $\forall p/\mathsf{Employee}, \forall s/\mathsf{Salary}, \forall t1/\mathsf{TimeInterval}$
  $\texttt{Salary}(p,s)(\texttt{at } t1) \wedge \texttt{GreaterThan}(s, 25000) \Rightarrow$
    $\texttt{WellPaid}(p)\ (\texttt{at } t1)$

`DR2`: $\forall p/\mathsf{Employee}, \forall t1, t2/\mathsf{TimeInterval}$
  $\texttt{WellPaid(p)}\ (\texttt{at } t2) \wedge\ \texttt{StartsBefore}(t2, t1) \Rightarrow$
    $\texttt{WellPaid}(p)\ (\texttt{at } t1)$

[4]For an extensive survey, see (Brachman and Levesque 1985).

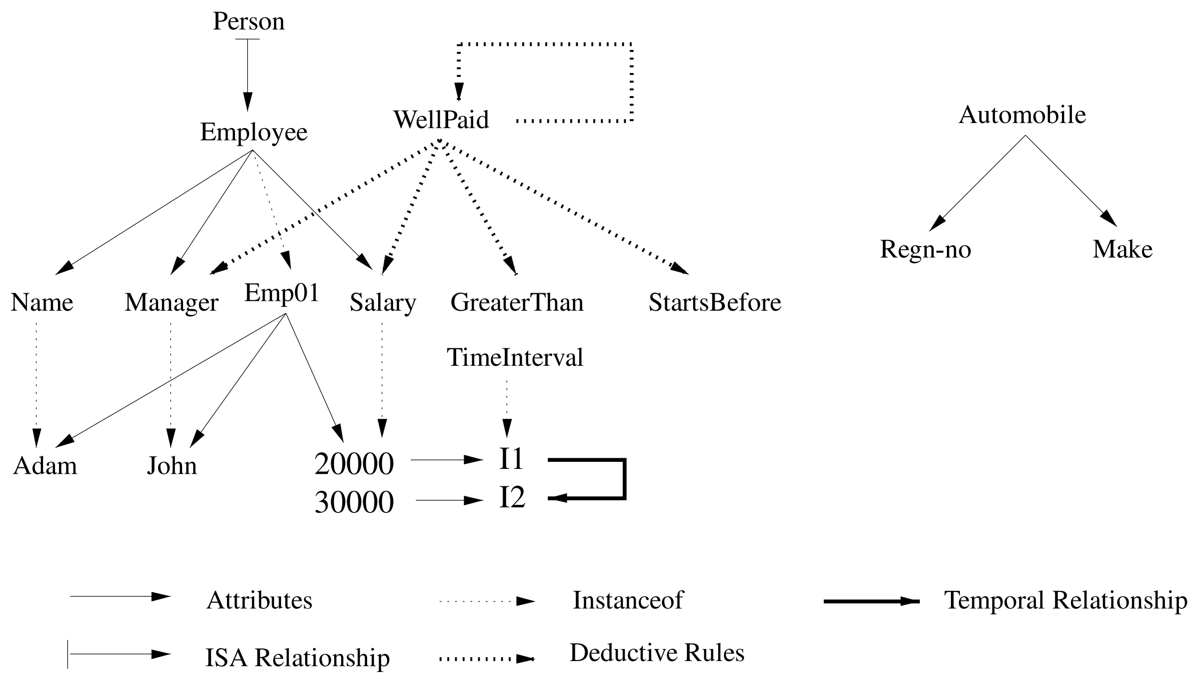


Figure 3: The knowledge base KB_1 and its structuring dimensions

The first rule states that if an employee's salary is over 25000 then the employee is well-paid. The second rule asserts that if an employee is well-paid during a time interval $t2$, she remains well-paid during any time interval that starts after $t2$. Finally, the knowledge base contains a class called `Automobile` which has attributes `Regn-no` and `Make`. For rest of the paper, we will refer to this knowledge base as KB_1.

Figure 3 shows a semantic net representation of KB_1. This knowledge base is a directed graph with five different types of edges, each corresponding to a different structuring dimension. The observation that most knowledge base features can be visualized as graphs has been the driving force in our research. As we will describe in later sections, we have taken graphs as an abstract representation for knowledge bases. We have selected our solution techniques in such a way that they are most suitable for graph structures. Such an approach helps us in understanding the interaction between knowledge base characteristics, which are abstracted in graph-theoretic terms, and the concurrency control requirements.

2.2 PERILS OF ARBITRARY INTERLEAVING

Suppose two users want to simultaneously access KB_1. The first user, called T_1, wants to change the definition of `DR1` to `DR3`, asserting that if an employee is a manager then she is well-paid:

`DR3`: $\forall p, q/\mathsf{Employee}, \forall t1/\mathsf{TimeInterval}$
`Manager`(p, q)(`at` $t1$) $\Rightarrow$ `WellPaid`(q) (`at` $t1$)

The second user, called T_2, wants to find all `WellPaid` employees during the time interval `I2`.

T_1 and T_2 will perform these operations using `UNTELL`, `TELL` and `ASK` commands (Mylopoulos et al. 1990). Consider the following interleaving of their operations:

1. T_1 executes an `UNTELL` removing `DR1` from the knowledge base.

2. T_2 executes an `ASK` to find all `WellPaid` employees. Currently, the knowledge base contains only `DR2`. Therefore, T_2 gets an answer that there are no `WellPaid` employees.

3. T_1 completes its job by a `TELL` command and adds `DR3`.

If we execute the operations of T_1 and T_2 without any interleaving and in the order T_1 after T_2, we get an answer `Adam`. If the order is T_2 after T_1, we will get an answer `John`. The answer returned in the above execution does not correspond to any state of the knowledge base and, therefore, such an interleaving is *incorrect*. The notion of *correctness* of concurrent executions has been formalized through the concept of *serializability* (Eswaran et al. 1976; Bernstein et al. 1978; Papadimitriou 1979). Let *transaction* refer to the execution of a user program on a knowledge base. Two executions are *equivalent* if they leave the knowledge base in the same state, and if each operation returns the same value in both executions. An interleaved execution of transactions is *serializable* if it is equivalent to some serial execution of the same collection of transactions. This example shows that not all concurrent interleaved executions of transactions are correct (serializable). The mechanism that controls the order in which the operations of concurrent transactions are processed, so that the overall execution is serializable, is called *concurrency control algorithm* or *policy.* The next sub-section describes some concurrency control algorithms from databases.

2.3 CONCURRENCY CONTROL ALGORITHMS FROM DATABASES

There is a vast body of literature on concurrency control algorithms (Papadimitriou 1986; Bernstein, Hadzilacos and Goodman 1987). There are three broad classes of such algorithms: locking, timestamps and serialization graphs. For each of these classes, there are variations based on multiple versions and optimistic methods. Locking-based algorithms have been most successful in practice and their performance is better understood. They also have special solutions for graph structures — the abstraction of knowledge base that appears to be the most appealing. Therefore, we have adopted the locking class of methods for knowledge bases. In the rest of the paper, we will focus on locking algorithms. The discussion on the other methods can be found in (Chaudhri, Hadzilacos and Mylopoulos 1992).

We will describe here two well-known locking algorithms. The first, known as two phase locking, does not make any assumption about the structure of the underlying data. The second, known as the DAG policy, assumes that the underlying data is structured as a directed acyclic graph.

Two-phase locking (2PL) (Eswaran et al. 1976) in simplified terms works as follows:

TP1. Associated with each data item is a distinct "lock". A transaction must acquire a lock on a data item before accessing it.
TP2. While a transaction holds a lock on a data item, no other transaction may access that data item.
TP3. A transaction cannot acquire any additional lock once it has released some lock (hence the name two-phase locking).

It can be shown that two-phase locking ensures serializability. In the example of the previous section, T_1 will lock `WellPaid` before changing its definition so that T_2 will not be able to read the partially updated value. This prevents the incorrect execution. If a transaction must acquire a lock (because of the rule TP1), but cannot do so (because of rule the TP2), it must wait until the transaction that owns that lock releases it. It is easy to construct scenarios in which locks are acquired in such a manner that a *deadlock* arises (Yannakakis 1982b): a cyclical sequence of transactions each waiting for the next to release a lock it must acquire. Such deadlocks may be resolved by choosing one of the transactions, aborting it (*i.e.*, undoing any effects it had on the knowledge base state), releasing its locks and restarting it at a later time.

The DAG policy may be specified by the following rules (Silberschatz and Kedem 1980):

D1. A transaction may begin execution by locking any item.
D2. Subsequently, it can lock an item if it has locked all the predecessors (i.e., parents) of that item in the past and is currently holding a lock on at least one of them.
D3. It may lock an item only once.

Unlike 2PL, the DAG policy is deadlock-free, that is, if transactions follow the DAG policy then a deadlock never arises. Furthermore, the DAG policy allows a transaction to release certain locks before it has acquired all locks it will ever need. The freedom of transactions to release locks earlier, often results in a greater degree of concurrency than would be possible under 2PL.

The next sub-section explores the applicability of these two locking policies to knowledge bases.

2.4 INADEQUACIES OF EXISTING METHODS AND FOCUS OF PRESENT WORK

Both 2PL and the DAG policy are possible candidates for a concurrency control algorithm for knowledge bases. However, as we describe below, both of them are inadequate for direct application to knowledge bases.

2PL requires that a transaction must hold all its locks until it has finished acquiring all of them. This has serious performance implications for the type of transactions likely to be applied to knowledge bases. For example, in the knowledge base KB_1 (Figure 3), while proving a goal through backward chaining (Genesereth and Nilsson 1987), a transaction is likely to access all the items that are below that goal in the inference graph, potentially a set including all deductive rules. Other examples of such potentially global knowledge base operations include truth maintenance systems (de Kleer 1986), temporal reasoning (Allen 1983) and recursive queries (Naqvi and Tsur 1989). In such situations, if we use 2PL, transactions will end up locking large portions of the knowledge base for long periods of time, thus significantly reducing concurrency.

The DAG policy, which holds only a small number of locks at any given time, could be a possible answer to this problem. We can use the DAG policy, if we view the underlying structure of the knowledge base

T_1= ⟨(D (WellPaid,Greaterthan)) (D (WellPaid,Salary)) (I (WellPaid,Manager)) ⟩

Figure 4: An example transaction

```
T1: (D (WellPaid,Greaterthan)) (D (WellPaid,Salary))        (I (WellPaid,Manager))
T2:                                                 (A WellPaid) ...
```

Figure 5: An example schedule

(for example, through the presence of generalization hierarchies or deductive rules structure) as a graph. However, the DAG policy assumes that there are no cycles in the underlying structure and the structure does not undergo any change (Yannakakis 1982a). Unfortunately, the structure of a knowledge base will contain cycles (*e.g.*, in the inference graph generated for a collection of recursive rules) and will undergo change (*e.g.*, when rule definitions are changed or rules are added or deleted). This means that the DAG policy cannot be directly applied to knowledge bases.

Thus, 2PL is not likely to give good performance for knowledge bases whereas the DAG policy does not offer enough functionality. Motivated by this, in the first phase of our research, we have extended the DAG policy to the Dynamic Directed Graph (DDG) policy that can handle cycles and updates in a knowledge base. In the next phase of our research, we plan to undertake a performance analysis of this policy which will give us guidelines for tuning the knowledge base structure for a multi-user environment and the comparative evaluation of the DDG and 2PL policies. The present paper describes the Dynamic Directed Graph (DDG) policy and its properties.

3 DYNAMIC DIRECTED GRAPH POLICY — AN ALGORITHM FOR KNOWLEDGE BASES

We begin this section by describing our assumed framework. Then, we describe the Dynamic Directed Graph (DDG) policy and its formal properties. We conclude this section by giving a locking policy that allows more concurrency than the DDG policy — but only under special circumstances.

3.1 AN ABSTRACT MODEL OF KNOWLEDGE BASES

We assume that a knowledge base is a directed graph $G = (V, E)$ where V is a set of nodes v_i (for example `WellPaid`), and E is a set of edges which are ordered pairs (v_i, v_j), of nodes (for example, (`WellPaid,Salary`)). We will use the generic term *entity* to denote both nodes and edges. A more precise representation of the knowledge base KB_1, shown in Figure 3, would have been a directed graph whose edges are of different colors, corresponding to the different types of relationships between the nodes. In the initial phase of our investigation, we do not distinguish amongst different types of edges.

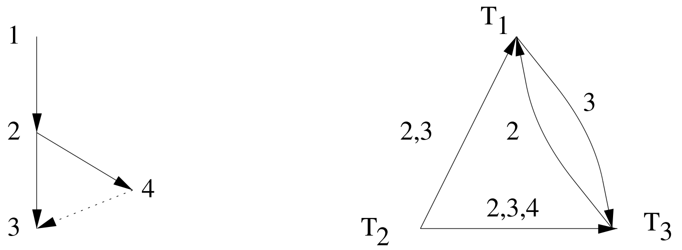
A user interacts with the knowledge base by means of transactions. Each transaction is a sequence of `TELL, UNTELL, RETELL` and `ASK` operations (Mylopoulos et al. 1990). These operations are implemented by means of more primitive operations. For example, in terms of a graph representation, the transaction T_1 of Section 2.2 consists of several primitive operations: delete the edges `(WellPaid,Salary), (WellPaid,Greaterthan)` and insert the edge `(WellPaid,Manager)`.

In our model, we will represent only these primitive operations and we will consider them to be *atomic*. Formally, an operation is a pair (`a` e), where `a` is an action (one of `INSERT, ACCESS, DELETE`, abbreviated by `I, A` and `D` respectively) and e is an entity, which is a node or an edge. For example, the transaction T_1 of Section 2.2, that changes the definition of the rule `DR1` to `DR3`, could be specified as in Figure 4.

In addition to the operations introduced above, we also define lock and unlock operations for an entity e, denoted (`L` e) and (`U` e) respectively. (`L` e) denotes the acquisition of that lock and (`U` e) the release of that lock. A *locked transaction* is a sequence of `ACCESS, INSERT, DELETE, LOCK` and `UNLOCK` operations. All the locked transactions are *well-formed* in the sense that a transaction cannot perform any operation on an entity unless it holds a lock in it. It is possible to generalize our results to the case of non-well-formed policies (Yannakakis 1982a) but we make this assumption to keep the model simple and intuitive.

We say that a transaction T (or an operation (`a` e)) is *defined* in a knowledge base state D if it does not insert (delete, access) an entity that already exists (does not exist) in the knowledge base.

A transaction system is a finite collection τ of transactions. A *schedule* S of a transaction system τ, at some instant t, is an ordering of the steps of some transactions of τ that preserves the order of actions of each transaction. The interleaving discussed in Section 2.2, and as shown again in Figure 5, is an example of a schedule. We assume that all the schedules are *legal* in the sense that while a transaction holds a lock on some



(a) The knowledge base (b) The serializability graph $D(S_{DAG})$

T_1: `(L 1) (A 1)` `(L 2,3) (A 2,3) (U 1,2,3)`
T_2: `(L 2,3,4) (I (3,4)) (U 2,3,4)`
T_3: `(L 2,4) (A 2,4) (U 2)` `(L 3) (A 3) (U 3,4)`

(c) The schedule S_{DAG}

Figure 6: An example of non-serializable schedule produced by the DAG policy

entity, no other transaction may acquire a lock on that entity. A schedule is *proper* if all the steps of all the transactions are defined in the knowledge base state they are executed. A *partial schedule* S of a transaction system τ is a legal and a proper schedule of any prefixes of the transactions of τ.

In our model, the *correctness* of a schedule is equivalent to *serializability*. Serializability of a schedule can be easily decided as follows (Papadimitriou 1986). Let $\overline{e}$ denote the set of nodes in an entity e (recall that an entity is a node or an edge). Construct a directed graph $D(S)$ by associating a node v_i with each transaction T_i and including an arc (v_i, v_j) if in schedule S, T_i acts on an entity e_x before T_j does on entity e_y and $\overline{e}_x \cap \overline{e}_y \neq \emptyset$. For the sake of clarity, the arcs of $D(S)$ are labelled with the entities in $\overline{e}_x \cap \overline{e}_y$. S is serializable if and only if the digraph D is acyclic.

3.2 A NAIVE APPLICATION OF THE DAG POLICY TO DYNAMIC GRAPHS

In this section, we illustrate by means of an example that the DAG policy might produce incorrect results in face of updates to the underlying DAG structure. Figure 6(a) shows a knowledge base which is manipulated by three transactions T_1, T_2 and T_3 running by the locking rules D1-D3 of the DAG policy. The schedule S_{DAG} produced by these transactions is shown in Figure 6(c) (`(L 2,3,4)` is a compact representation of three operations: `(L 2)`, `(L 3)` and `(L 4)`. The other operations in this schedule should be interpreted in a similar way.). When T_1 begins execution, the edge $(4,3)$ is non-existent. The DAG policy pre-computes the locked transaction, requiring T_1 to be holding a lock on node 2 at the time it acquires a lock on node 3. In the meantime, T_2 inserts the edge $(4,3)$ and T_3 completes a part of its execution. If T_1 continues using the lock steps that it has pre-computed, we will get the schedule S_{DAG} as shown in Figure 6(c). The serializability graph of this schedule is shown in Figure 6(b) which contains a cycle and thus S_{DAG} is not serializable. With this motivation, let us give the description of the DDG policy which in addition to handling cycles, overcomes this problem of updates.

3.3 DESCRIPTION OF THE DYNAMIC DIRECTED GRAPH (DDG) POLICY

A *locking policy* is a collection of rules which specifies how the transactions should acquire locks. Formally, a locking policy P is a relation such that $P(T, \overline{T})$, only if transaction T is a subsequence of well-formed locked transaction $\overline{T}$.

The locking rules of the DDG policy assume that the underlying graph is always connected and has a single root[5]. In the first sub-section, we will show how any arbitrary graph is converted and maintained in this restricted form. In the second sub-section, we will specify the locking rules of the DDG policy.

3.3.1 Restricting the Knowledge Base to a Rooted and a Connected Graph

Restricting the knowledge base to a rooted and a connected graph is a two step process. First, the knowledge base has to be in this form to start with, and second, this shape has to be maintained as the knowledge base undergoes updates. The first step is implemented by pre-processing rules and the second step by structure maintenance rules. We will specify these rules and then give justifications for keeping the graph rooted and connected.

[5] Root is a node with no incoming edges.

Pre-processing Rules

The pre-processing rules are applied when the knowledge base is initially started. These rules take directed graph G of the knowledge base as input and generate a graph $\overline{G}$ as output. They compute some information which is later used by locking rules and ensure that the graph has a single root and is connected. In specifying these rules and in the rest of the paper, we will deal with a cycle by considering the strongly connected component (SCC) which contains all the nodes on that cycle.

P1. Partition G into $G_i(V_i, E_i)$ $1 \leq i \leq k$, where the underlying undirected graph of each G_i is a connected component of the underlying undirected graph of G. For each component, identify the non-trivial strongly connected components[6] G_{ij} $1 \leq j \leq b_i$. For each connected component i, identify the sources s_{ij}, $1 \leq j \leq l_i$.

P2. For each $G_i(V_i, E_i)$, add a *control node* c_i. Add edges (c_i, s_{ij}) $1 \leq i \leq k$, $1 \leq j \leq l_i$. Add another control node C and the edges (C, c_i) $1 \leq i \leq k$.

P3. Call the resulting graph $\overline{G}(\overline{V}, \overline{E})$. Thus, $\overline{V} = V \cup \{c_i | 1 \leq i \leq k\} \cup \{C\}$ and $\overline{E} = E \cup \{(c_i, s_{ij}) | 1 \leq j \leq l_i, 1 \leq i \leq k\} \cup \{(C, c_i) | 1 \leq k\}$.

For example, if we apply the above process to KB_1, we will need to add nodes c_1 and c_2, one for each connected component, and a control node C for the whole graph. $\overline{G}$ will contain the original graph G, corresponding to the knowledge base KB_1 of Figure 3 and the following edges: (c_1, `Person`), (c_1, `WellPaid`), (c_2, `Automobile`), (C, c_1) and (C, c_2). The resulting graph $\overline{G}$ is rooted and connected.

Structure Maintenance Rules

Insert and delete operations applied to $\overline{G}$ may cause it to become dis-connected or to acquire new sources. Furthermore, the information about the connected components of the graph needs to be updated. All this is implemented by the following rules.

M1. When a new source, s_i, is created in the connected component, G_j, add the edge (c_j, s_i)

M2. When an existing source, s_i, is removed from the component G_j, remove the edge (c_j, s_i). If the removal of s_i results in the creation of new sources, then do as in M1.

M3. Two connected components G_i and G_j are merged. This will happen if an edge (v_i, v_j) is inserted with $v_i \in G_i$ and $v_j \in G_j$. Let $s_{i1}, s_{i2}, \ldots s_{il_i}$ and $s_{j1}, s_{j2}, \ldots s_{jl_j}$ be the sources of G_i and G_j respectively. Remove c_j (and therefore the edges, (C, c_j) and (c_j, s_{jm}) $1 \leq m \leq l_j$). Recompute the sources of the merged component as: $\{s_{i,1}, s_{i,2}, \ldots s_{il_i}\} \cup \{s_{j1}, s_{j2}, \ldots s_{jl_j}\} - \{v_j\}$. Add edges (c_i, s_{ij}) for the new set of sources.

M4. A connected component $G_i(V_i, E_i)$ is split into $G_j(V_j, E_j)$ and $G_k(V_k, E_k)$. Compute new sources and add or delete appropriate edges.

M5. As there are updates in the graph, keep updating the information on connected components[7].

For example, if we want to store in KB_1, the information about the automobiles owned by an employee, we can define `Vehicle` as an additional attribute of `Employee` and make it an instance of `Automobile` (achieved by inserting a node `Vehicle` and adding the edges `(Employee,Vehicle)` and `(Automobile,Vehicle)`). This will merge the two components of the graph G. After applying M3, the new $\overline{G}$ will no longer have the control node c_2 and instead, will have the edge (c_1 ,`Automobile`). The resulting graph is still rooted and connected.

Now, let us give some justification for this process. If the graph corresponding to a knowledge base is not connected, a transaction can span more than one component. To guarantee the correctness of all the schedules in such a situation, we will have to ensure that the transactions that access some components in common follow the same serialization order in these components. This could be achieved by maintaining a graph external to the knowledge base, in which there is a node v_i corresponding to each transaction T_i, and an edge (v_i, v_j) if T_i precedes T_j in some component. Assuming the executions are serializable within each connected component, the schedules produced in such a situation will be correct if this external graph is acyclic. This external graph is not necessary if there is only one connected component in the knowledge base. Thus, if we keep the underlying graph of the knowledge base connected at all times, we save the cost of maintaining and checking cycles in this external graph. On the other hand, we incur some cost in pre-processing and structure maintenance which is comparable to the cost of maintaining the external graph. Overall, our proposed design results in a net saving, and therefore, is computationally more efficient.

In the above construction, we assume that there will be only two levels of control nodes — one level of control node for each connected component and one control node for the whole graph. The number of levels of these nodes can be varied to improve the performance of the knowledge base. The exact improvement will depend on the environment in which the knowledge

[6] A non-trivial strongly connected component is a strongly connected component that has more than one node. From now on, whenever we mention a strongly connected component, we will assume that it is non-trivial.

[7] One can use incremental graph algorithms. Such algorithms can dynamically maintain certain kinds of information about a graph in the face of updates to the graph without recomputing the information from scratch (Italiano 1986; Italiano 1988).

base will be used. We plan to explore this issue in our future research.

From now on, we will assume that the knowledge base is always connected and has a single root.

3.3.2 Acquiring Locks

The rules presented in this section are the core of the DDG policy as they specify how the transaction should acquire locks.

There are two key differences between the locking rules of the DAG policy and the DDG policy. First, the locking rules of the DDG policy are applied to the current state of the graph, whereas the locking rules of the DAG policy are applied to the state of the graph when the transaction begins execution. In case of the DAG policy, there is no need to distinguish between the initial and the current state of the graph, because the graph never changes. As we saw in the schedule S_{DAG} of the previous sub-section, this results in an undesirable behaviour in face of the updates. Second, the locking rules of the DDG policy provide a solution for cycles. It is easy to see that using the locking rules of the DAG policy, it is not possible to lock any node on a cycle. The DDG policy solves both of these problems.

Let us first give some definitions and then specify the locking rules.

Let $R(T)$ be the set of nodes to be accessed by a transaction T. Let us define a *dominator* of a set of nodes U in a rooted and connected graph to be a node d, such that all paths from the root node to each node $v \in U$ pass through d. The root node dominates all the nodes in the graph, including itself. *Entry point* of a strongly connected component (SCC), G_{ij}, is a node v of G_{ij}, such that, there is an edge (w, v) of $\overline{G}$ so that w is not in G_{ij}.

Locking Rules

L1. The first node to be locked by T is D, a dominator of $R(T)$ with respect to $\overline{G}$.

L2. Before T performs any operation (`INSERT/DELETE` or `ACCESS`), on a node v (or an edge (u, v)), T has to lock v (both u and v).

L3. A node v can be locked if and only if all its predecessors in the present state of $\overline{G}$, that do not lie on the same non-trivial strongly connected component as v, have been locked by the transaction in the past, and the transaction is presently holding a lock on at least one of them. All the nodes on a strongly connected component are locked together in one step, provided all the entry points of that SCC have been locked. A node that is being inserted can be locked at any time.

L4. Each node can be locked at most once.

For example, the transaction T_1 of Figure 4 would start by locking `WellPaid`. Then it would lock `Salary` and then the edge `(WellPaid,Salary)`. After deleting this edge, T_1 could release the lock on `Salary` and proceed to lock `Greaterthan`. Thus, the transaction is able to acquire locks even after releasing some of the locks — a clear improvement over two-phase locking.

In the example of Figure 6, T_1 will not be allowed to lock node 3 in the new state of the knowledge base unless it has locked node 4 as well. This will prevent the incorrect schedule S_{DAG}. In the next sub-section we will prove that this is true for any schedule produced by the DDG policy.

This completes the description of the DDG policy.

3.4 PROPERTIES OF THE DYNAMIC DIRECTED GRAPH (DDG) POLICY

We will begin this section by showing that that the DDG policy always produces correct schedules. Then, we will prove that the DDG policy is deadlock-free and well-structured.

3.4.1 Correctness

The correctness of a locking policy can be formalized by the notion of *safety*.

A locked transaction system is *safe* if any legal schedule S of it is correct. A locking policy P is safe if for any transaction system $\tau = \{T_1, \ldots, T_m\}$ and $\overline{\tau} = \{\overline{T}_1, \ldots, \overline{T}_m\}$ where $P(T_i, \overline{T}_i)$ for all $1 \leq i \leq m$, the locked transaction system $\overline{\tau}$ is safe.

We will first state a theorem that characterizes the unsafe transaction systems. Intuitively, this theorem says that if a locked transaction system is not safe then it is *always* possible to construct a *canonical* non-serializable schedule, which is legal and proper, and in which all transactions except one are executed serially. This result is the generalization of a similar result for transaction systems that do not contain insert/delete operations (called static systems) (Yannakakis 1982a) to systems that contain insert/delete operations (called, dynamic systems). For the static systems, the structure of the serialization graph corresponding to the canonical schedule is linear whereas, for the dynamic systems the corresponding graph is not necessarily linear — it can be a general graph. This characterization of canonical schedules is a very useful tool in proving the correctness of the policy.

The *interaction graph* $G(\tau)$ of a transaction system τ is an undirected graph with one node T_i corresponding to each transaction of τ and an edge between any two nodes whose corresponding transactions have an entity in common. We can now state the theorem on canonical schedules.

Theorem 1 *A transaction system τ is not safe if and only if there are transactions $T_1, \ldots, T_k$ in τ ($k > 1$) and entities $A_1, \ldots, A_k$, not necessarily distinct, such that*

(1) A subsequence of $T_1, \ldots, T_k$ forms a chordless cycle in the interaction graph $G(\tau)$.

(2) In T_1 the entity A_k is locked after A_1 is unlocked.

(3a) The following partial schedule is legal and proper. Let $T_1^{'}$ be the prefix of T_1 up to the ($\mathtt{L}$ A_k) step, and $T_i^{'}$, for $i \neq 1$, the prefix of T_i up to and including the ($\mathtt{U}$ A_i) step. $S^{'}$ is the serial execution of $T_1^{'}, \ldots, T'_k$ in this order.

(3b) Furthermore, $S^{'}$ can be extended to a complete schedule, that is, can avoid deadlock.

The details of the proof of this theorem can be found in (Chaudhri, Hadzilacos and Mylopoulos 1992). □

Theorem 2 *Dynamic Directed Graph policy is a safe policy.*

Proof Outline: Suppose it is not. Then, choose transactions $T_1, \ldots, T_k$, entities $A_1, \ldots, A_k$ and the corresponding schedule $S^{'}$ as in Theorem 1.

The serialization graph $D(S^{'})$ of the schedule up to (but not including) the ($\mathtt{L}$ A_k) step of T_1 is an acyclic graph over $T_1, \ldots, T_k$. Let d be the length of the shortest path from T_i to T_k in the serialization graph $D(S^{'})$. Let B_i be the first entity locked by the transaction T_i and G_i be the state of the graph when T_i begins execution. By induction on d, it can be shown that for $1 \leq i \leq k$, B_i is a descendant of A_k in G_i. Let G_{k+1} be the state of the graph after T_k finishes its execution. Then, B_1 is a descendant of A_k in G_1, which is a contradiction to the fact that T_1 holds a lock on a parent of A_k in G_{k+1}. Hence the assumed non-serializable schedule as claimed does not exist and the DDG policy is safe. □

3.4.2 Deadlock-freedom

As described in Section 2.3, a deadlock is a situation when a cyclical sequence of transactions are each waiting for the next to release a lock it must acquire (Yannakakis 1982b). Such deadlocks may be resolved by choosing one of the transactions, aborting it (*i.e.*, undoing any effects it had on the knowledge base state), releasing its locks and restarting it at a later time. Thus, if a locking policy is deadlock-free, it would mean that it will never have to abort any transaction unnecessarily.

Let us consider a typical case of deadlock for a set τ of transactions. Deadlock arises in a partial schedule S of τ when every transaction wants to lock an entity in the next step that is already locked by some other transaction. This means that there is a set of transactions $\{T_1, \ldots, T_k\}$ such that the next step of T_i is ($\mathtt{L}$ x_i) where x_i is currently locked by T_{i+1} (where we take $k+1 = 1$).

Thus, in the partial schedule S, transaction T_i accesses x_{i-1} before T_{i-1}; if S could possibly finish in any way then the resulting schedule would not be correct. In other words, deadlocks prevent some wrong schedules from finishing. Let us show that the DDG policy is deadlock-free using this fact.

Theorem 3 *The Dynamic Directed Graph (DDG) policy is deadlock-free.*

Proof Outline: In the deadlock state, none of the transactions can release the locks on entities that are causing the deadlock. This means that these entities must be parents of the entities that need to be locked. But this would imply that these entities lie on a cycle. The DDG policy locks a cycle in one step which could not be locked by different transactions simultaneously. Hence, the DDG policy must be deadlock-free. □

3.4.3 Well-structured-ness

A locking policy P is *well-structured* if it allows a transaction to access an arbitrary set of entities in the knowledge base. Formally, P is well-structured if for any set of entities D in the knowledge base and any transaction T, such that T is defined in D[8], there is a locked transaction $\overline{T}$ such that $P(T, \overline{T})$ and $\overline{T}$ is also defined in D.

Theorem 4 *The DDG policy is well-structured.*

Proof Outline: It will be possible for T to lock an arbitrary set of entities in the underlying graph if we can identify one entity by starting from which the locking rule L3 will be satisfied for all the entities in the transaction. Since the rules P1-P3 and M1-M5 ensure that the graph is always connected and has a single source, the node C of $\overline{G}$ satisfies this property for all the nodes in the graph. Hence, the DDG policy is well-structured. □

3.5 A MORE LIBERAL VARIANT OF THE DYNAMIC DIRECTED GRAPH (DDG) POLICY

The locking rule L3 of the DDG policy requires that once all entry points have been locked, nodes of a strongly connected component (SCC), and therefore, all the nodes on a cycle, should be locked together in one step. This will not permit any concurrency within

[8] Recall that a transaction T is defined in a knowledge base state D if it does not insert (delete, access) an entity that already exists (does not exist) in the knowledge base.

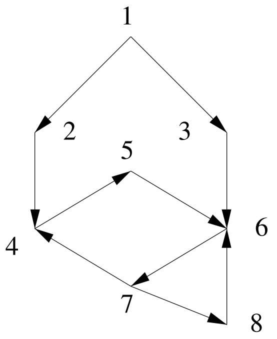


Figure 7: Permitting concurrency within cycles

the cycles. A natural question is whether this is the best one can do or is there a locking policy that can allow concurrency within cycles. We present a rule L3′ which permits concurrency within an SCC.

L3′ If a node does not lie on an SCC or it is not an entry point of an SCC, it can be locked if and only if all its predecessors in the present state of $\overline{G}$ have been locked by the transaction in the past and the transaction is presently holding a lock on at least one of them.

If a node is an entry point of an SCC, it can be locked if and only if all its predecessors in the present state of $\overline{G}$, that do not lie on the same SCC, have been locked by the transaction in the past and the transaction is presently holding a lock on at least one of them.

Replacing L3 by L3′ in the DDG policy gives a locking policy DDG′, which allows concurrency within cycles. For example, consider a graph G, shown in Figure 7. This graph has one non-trivial strongly connected component namely 4,5,6,7,8. The DDG′ policy will permit transactions $T_1 = \langle(\mathtt{A}\ 2,4,5)\rangle$ and $T_2 = \langle(\mathtt{A}\ 3,6,7)\rangle$ to concurrently access this SCC. However, transactions $T_3 = \langle(\mathtt{A}\ 2,4,5,6)\rangle$ and $T_2 = \langle(\mathtt{A}\ 3,6,7,4)\rangle$ cannot concurrently access this SCC. Each of them will begin execution by locking the node 1. Suppose, T_3 locks the node 1 before T_4 locks it. T_3 will be able to lock the entry points 4 and 6 before T_4 and it will not release locks on them until it has finished processing the whole SCC. Thus, T_3, will be the only transaction executing within the SCC (Similar arguments can be made when T_4 locks node 1 before T_3.). Thus, the DDG′ policy permits concurrency within an SCC to a limited extent. The DDG′ policy satisfies the same properties as the DDG policy:

Theorem 5 *The DDG′ policy is a safe, deadlock-free and well-structured policy.*

Proof Outline: A simple generalization of the proofs for the DDG policy works for the DDG′ policy. □

Let us analyze the relative merits of the DDG and the DDG′ policies. The DDG policy is intuitive and simple as it treats strongly connected components (and cycles) as a unit of locking. In the presence of updates, the DDG′ policy will require more bookkeeping effort than the DDG policy, and therefore, the DDG policy appears more suitable for the case of updates. Furthermore, in knowledge base applications such as recursive rules, it is highly likely that a transaction will access *all* the entities on a cycle and probably access them *more than once*. Since, a transaction is allowed to lock an entity only once[9] (L4), it means that it will have to retain all the locks on an SCC until it has finished processing it. Thus, if an SCC has only one entry point, then there is no difference in the concurrency permitted by the two policies. Furthermore, when an SCC has more than one entry point, the concurrency within an SCC is possible only if some nodes can be accessed through different entry points. Therefore, we feel that, in general, we cannot take advantage of the more concurrency allowed by the DDG′ policy. Based on these arguments, we have adopted the DDG policy as our initial design.

4 RELATED WORK, CONCLUSIONS AND SUMMARY

There has been very little work on concurrency control with a specific reference to knowledge bases. (Raschid, Sellis and Lin 1988) uses concurrency to improve the performance of rule execution in the context of the OPS5 system (Forgy 1982). Their proposal is to parallelize the inference process of one user, whereas our focus is on parallelizing the operations of different users. Similar efforts have been made in (Filman 1989; Ishida, Yokoo and Gasser 1990; Schmolze and Goel 1990).

(Garza and Kim 1988) studies concurrency control problem for object-oriented databases and proposes a locking method that has provisions for variable units of locking. (Elkan 1990) looks at deductive databases and proposes efficient algorithms for checking conflicts between transactions. Neither of these papers addresses the problem that the large portions need to be locked in knowledge bases, and that the internal structure of a knowledge base may be used to do more efficient concurrency control. There is a flurry of work on flexible transaction models (Berghouti and Kaiser 1991) focusing on the problems that arise in domains such as software engineering and CAD.

Multidatabase concurrency control looks at the situations when a transaction can span more than one system (Breitbart, Garcia-Molina and Silberschatz 1992). This research will be useful for knowledge bases because we can easily visualize situations when a part of a

[9] This condition is necessary to guarantee the safety of a locking policy (Yannakakis 1982a).

transaction is executed in a knowledge base (inference) and a part in a database (data retrieval) (Brodie 1989). We do not address the problem of multidatabase transactions in our research.

According to our knowledge, concurrency control has not received the attention of AI community.

The design of locking policy for cycles is an example of how the knowledge base features might interact with techniques such as concurrency control. The discussion in Section 3.5 illustrates that, in general, our locking policy will not allow transactions to concurrently access entities along a cycle. This means that for performance reasons, we should try to design a knowledge base with few and small cycles. This suggests the existence of an *expressiveness vs performance* tradeoff meaning that good performance considerations dictate ruling out certain knowledge base designs (for example, a knowledge base with too many large cycles).

The expressiveness vs performance tradeoff is analogous to the *expressiveness vs tractability* tradeoff (Levesque and Brachman 1985) where complexity considerations dictate ruling out certain knowledge base designs. Performance, measured in terms of throughput and response time (Lazowska et al. 1984), is a measure of desirability, similar and related to complexity — but may be of more pragmatic nature. Furthermore, the designs ruled out by the complexity considerations are not likely to be the same as those ruled out for performance reasons. For example, negations cause intractability in the reasoning (Levesque and Brachman 1985; Nebel 1988), but it will be possible to view the knowledge base containing negations as a graph and use the algorithm presented in this paper. On the other hand, the reasoning with cycles is tractable but the presence of cycles has a potential of causing performance bottlenecks. We suggest that, in addition to the complexity considerations, performance requirements should be a design criteria for knowledge-based systems of future.

There seem to be many interesting extensions to this work which we believe will affect the way we think about the design of knowledge-based systems. For example, we are beginning work on the development of a performance model that will tell us the desired features of a graph for the best performance. This will give us recommendations on structuring of a knowledge base for optimal performance in a multi-user environment. We are planning to generalize this algorithm to handle graphs with edges of different "colors" and to have more general locking modes. We are also considering to implement this algorithm as part of a knowledge base so as to verify its applicability in a knowledge base environment. Further down the road, we expect that issues of fault tolerance such as recovery (Bernstein, Hadzilacos and Goodman 1987) will play an important role in the extension of our research results.

To summarize, this paper has argued that concurrency control is a necessity for large multi-user knowledge bases. The paper then focused on a locking-based approach that seems most viable from a practical viewpoint. A locking algorithm, called the Dynamic Directed Graph (DDG) policy, was presented that exploits the structure of a knowledge base and selectively locks only a small number of entities at any one time. The DDG policy can handle situations where the knowledge base contains cycles and undergoes changes over time. Our results include a proof of correctness of the proposed algorithm, an analysis of its properties and a discussion of how concurrency control requirements might affect the knowledge-based systems of the future.

In conclusion, we would like to claim that the paradigms for the design of knowledge-based systems should include *expressiveness vs performance* tradeoffs, where performance is measured in terms of throughput and response time. We believe that algorithms such as the ones presented in this paper constitute a modest contribution towards this direction and can provide an important enabling technology to the goal of knowledge sharing, as articulated in (Neches et al. 1991).

Acknowledgments

This research was supported by the University of Toronto, the Information Technology Research Centre of Ontario, the Natural Science and Engineering Research Council of Canada and the Institute of Robotics and Intelligent Systems.

References

Allen, J. F. 1983. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843.

Berghouti, N. S. and Kaiser, G. E. 1991. Concurrency Control in Advanced Database Applications. *Computing Surveys*, 23(3):269–318.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Welssley Publishing Company.

Bernstein, P. A., Rothnie, J. B. J., Goodman, N., and Papadimitriou, C. H. 1978. The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case). *IEEE Transactions on Software Engineering*, 4(3):154–168.

Brachman, R. J. and Levesque, H. J., editors 1985. *Readings in Knowledge Representation and Reasoning.* Morgan Kaufmann Publishers, Inc., Los Altos, CA.

Breitbart, Y., Garcia-Molina, H., and Silberschatz, A. 1992. Overview of Multidatabase Transaction Management. Technical Report STAN-CS-92-1432, Stanford University.

Brodie, M. L. 1989. Future Intelligent Information Systems: AI and Database Technologies Working Together. In Mylopoulos, J. and Brodie, M. L., editors, *Artificial Intelligence and Databases*. Morgan Kaufmann Publishers Inc., CA.

Chaudhri, V. K., Hadzilacos, V., and Mylopoulos, J. 1992. Concurrency Control for Knowledge Bases. Technical Report Forthcoming, University of Toronto.

de Kleer, J. 1986. An assumption-based TMS. *Artificial Intelligence – An International Journal*, 28(2):127–162.

Elkan, C. 1990. Independence of Logic Database Queries and Updates. In *Proceedings of the 1990 Principles of Database Systems Conference*, pages 154–160.

Eswaran, K., Gray, J. N., Lorie, R. A., and Traiger, I. L. 1976. The Notions of Consistency and Predicate Locks in Database Systems. *Communications of the ACM*, 19(9):624–633.

Filman, R. E. 1989. New Generation Knowledge System Development Tools. Technical Report DARPA Contract No. F30602-85-C-0065, IntelliCorp Inc.

Forgy, C. L. 1982. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence – An International Journal*, 19(1):17–37.

Garza, J. F. and Kim, W. 1988. Transaction Management in an Object-Oriented Database Systems. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 37–45, Chicago, IL.

Genesereth, M. R. and Nilsson, N. J. 1987. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Los Altos, CA.

Gray, J. N. 1992. Personal Communication.

Ishida, T., Yokoo, M., and Gasser, L. 1990. An Organizational Approach to Adaptive Production Systems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 52–58, Boston, MA.

Italiano, G. F. 1986. Amortized Efficiency of a Path Retrieval Data Structure. *Theoretical Computer Science*, 48(2,3):273–281.

Italiano, G. F. 1988. Finding Paths and Deleting Edges in Directed Acyclic Graphs. *Information Processing Letters*, 28(1):5–11.

Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C. 1984. *Quantitative System Performance: Computer System Analysis using Queueing Network Models*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

Lenat, D. B. and Guha, R. 1989. *Building Large Knowledge-based Systems: Representation and inference in the Cyc Project*. Reading, Mass.:Addison-Wesley Publishing Co.

Levesque, H. J. and Brachman, R. J. 1985. A Fundamental Tradeoff in Knowledge Representation and Reasoning. In Brachman, R. J. and Levesque, H. J., editors, *Readings in Knowledge Representation*, pages 42–70. Morgan Kaufmann Publishers, CA.

Mylopoulos, J., Borgida, A., Jarke, M., and Koubarakis, M. 1990. Telos: A Language for Representing Knowledge About Information Systems. *ACM Transactions on Information Systems*, 8(4):325–362.

Naqvi, S. and Tsur, S. 1989. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, New York.

Nebel, B. 1988. Computational Complexity of Terminological Reasoning in BACK. *Artificial Intelligence – An International Journal*, 34(3):371–383.

Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T., and Swartout, W. 1991. Enabling Technology for Knowledge Sharing. *AI Magazine*, 12(3):36–56.

Papadimitriou, C. 1979. Serializability of Concurrent Database Updates. *Journal of the Association for Computing Machinery*, 26(4):631–653.

Papadimitriou, C. 1986. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, MD.

Raschid, L., Sellis, T., and Lin, C.-C. 1988. Exploiting Concurrency in a DBMS Implementation for Production Systems. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, pages 34–45, Austin, TX.

Schmolze, J. G. and Goel, S. 1990. A Parallel Asynchronus Distributed Production System. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 65–71, Boston, MA.

Silberschatz, A. and Kedem, Z. M. 1980. Consistency in Hierarchical Database Systems. *Journal of the Association for Computing Machinery*, 27(1):72–80.

Yannakakis, M. 1982a. A Theory of Safe Locking Policies in Database Systems. *Journal of the Association for Computing Machinery*, 29(3):718–740.

Yannakakis, M. 1982b. Freedom from Deadlock of Safe Locking Policies. *SIAM Journal of Computing*, 11(2):391–408.