A Mechanism for Concurrency Control in a Coupled Knowledge Base Management System

YUH-JIUN CHEN* and WEI-PANG YANG*

A new concurrency control mechanism in a coupled Knowledge Base Management System is presented. The system combines a multi-users logic programming system and a Relational Database Management System. The proposed mechanism, Query-Rule Locking, is inspired by two-phase locking, but it uses the concept of 'relate to' to detect and handle new conflicting problems that arise when multi-users execute Horn clause transactions that access a very large shared knowledge base concurrently. In addition, the number of objects to be locked is minimized by using the 'relate-cover' concept to increase the throughput of transaction execution. This method has been proved to guarantee serializability as well as correctness.

1. Introduction

It is widely predicated that knowledge processing will be a major area of computer applications in the 1990's, where problem solving and logic inference will be the most important part [Li84]. Knowledge Base Management Systems (KBMS) are intended to provide the representation framework and the computational environment for carrying out this task [Brodie85]. A coupled KBMS model that combines a logic programming system (Prolog) and a Relational Database Management System (DBMS) is proposed by Itoh at ICOT [Itoh86]. In this paper, we will present a new concurrency control mechanism in this model.

A common problem in expert systems development is that experts often fail to describe knowledge completely, at least in the initial phase of a project [Kastner86]. Even when the system has been put into use, a human expert is necessary to keep the knowledge base up to date as the nature of the task, the domain of our knowledge of it changes with time, as they inevitably will [Taylor86]. A good example of an expert system that has been implemented in this manner is XCON, the expert system that Digital Equipment Corporation uses to configure new VAX computers. One of the key problems DEC faced was the continuing changes necessitated by new equipment releases, new specifications, etc. Thus there is an expert whose job involves adding new information and modifying rules in XCON's knowledge base to keep it current [Harmon85]. Since the knowledge of our real world progresses every day, the KBMS should provides the function for the knowledge engineering to modify and improve the knowledge base both during the designing and maintaining phases of the KBMS in order to incorporate new experiences and new facts immediately as they become available.

Consider a very large KBMS in the future with frequently update, the maintaining task is more than one person can handle and a team of knowledge engineers is needed. Each of those knowledge engineers may responds for parts (or whole) of the knowledge base and has the right to update that part. Under this environment, it is possible to have some supervisors (knowledge engineer) who update or consult the knowledge base concurrently. Furthermore, since the KBMS is so large and so many users need to use it concurrently, it is unreasonable to stop the queries of all the users waiting for the updating of the supervisors. As a result, a concurrency control mechanism that suspend the minimum set of users that may conflict with the updating of the rules is necessary.

In a large KBMS, multi-users may share data and run transactions concurrently. Serializability is proposed as the criterion for correctness in a multi-users environment [Eswaran76]. A given interleaved execution of some set of transactions is said to be serializable if it produces the same read result and the same final write result as some serial execution of the same transactions does. That is, the interleaved execution produces the same output for each of their read operations and the same final knowledge base state as some serial execution operates on the same initial knowledge base state.

There are several ways in which Prolog transactions may conflict and produce the non-serializable behavior [Carey84]: (1) fact-fact conflicts, (2) query-fact conflicts, (3) rule-rule conflicts, and (4) query-rule conflicts. These problems arise when one transaction updates a fact or a rule being read or updated by another concurrent transaction. Carey et al use an algorithm based on two-phase locking [Eswaran76] to handle (1) and (2) of the above Prolog concurrency control problems. But they do not consider the type (3) and (4) conflicts since

*Institute of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan, ROC

they assume that the queries do not assert or retract rules.

In this paper, we propose a new concurrency control technique to solve the type (3) and (4) of the concurrency control problems in a coupled KBMS. Section 2 describes the coupled KBMS we use, and points out the main concurrency problems in that model. In Section 3, we propose a concurrency control mechanism, based on two-phase locking, to solve those problems. Some conclusions are given in Section 4.

2. The Model and The Problems

2.1 A Coupled Model of Knowledge Base Management System

The model we use is modified from the model developed at ICOT [Itoh86] as shown in Figure 1.

The EDB (Extensional DataBase) contains shared user data which are expressed in relations. The IDB (Intensional DataBase) contains shared user knowledge and control knowledge (meta-knowledge) which are expressed in Horn clauses. An example of IDB and EDB is shown in Figure 2. Every example we use in this paper refers to the EDB and IDB in Figure 2.

The knowledge management program converts the Horn clause query to an equivalent Horn clause which is then converted to relational algebraic expressions and ask the Relational DBMS to accomplish this query [Itoh86]. The knowledge management program is basically an extended Prolog system with the following assumptions: (1) Multiple users can run Prolog transactions concurrently, (2) Users can add or delete facts as well as rules, (3) Users can ask whether a rule is true (exists), and (4) A compilation approach is used in the inference process and the answer is produced set at a time.

User programs are transactions written in Prolog-like Horn clauses. Here are examples of transactions T1 and Horn Clause Conversion.

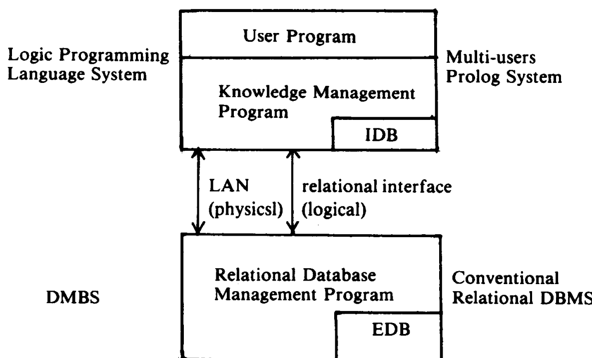


Fig. 1 A coupled Knowledge Base Management System.

IDB

- *r*1 child(*X*, *Y*):-father(*Y*, *X*)
- *r*2 child(*X*, *Y*):-father(*Z*, *X*), marry(*Z*, *Y*)
- *r*3 youngman(*X*):-person(*X*, '*m*', 30)
- *r*4 father(*X*, *Y*):-edb(father(*X*, *Y*))
- *r*5 marry(*X*, *Y*):-edb(marry(*X*, *Y*))
- *r*6 person(*X*, *Y*, *Z*):-edb(person(*X*, *Y*, *Z*))
- *r*7 father_in_law(*X*, *Y*):-father(*X*, *Z*), marry(*Z*, *Y*)

EDB

father

father name	child name
'joe'	'john'
'joe'	'tony'
'bob'	'paul'

marry

husband	wife
'joe'	'mary'
'john'	'judy'
'bob'	'maggie'

person

name	sex	age
'joe'	'*m*'	60
'mary'	'*f*'	58
'john'	'*m*'	32
'judy'	'*f*'	26
'tony'	'*m*'	30
'bob'	'*m*'	42
'paul'	'*m*'	15
'maggie'	'*f*'	41

Fig. 2 An example of IDB and EDB.

Example 2.1. [transactions]

Consider a transaction T1 which consists of four clauses and ends with commit.

T1: assert(child(*X*, *Y*):-father(*Y*, *X*)),
retract(child(*X*, *Y*):-father(*Z*, *X*), marry(*Z*, *Y*)),
youngman(*X*):-person(*X*, '*m*', 30),
youngman(*X*),
commit(T1)

Transaction T1 adds a rule (i.e. child(*X*, *Y*):-father(*Y*, *X*)) into IDB, deletes another rule (i.e. child(*X*, *Y*):-father(*Z*, *X*), marry(*Z*, *Y*)) from IDB, asks whether the rule (i.e. youngman(*X*):-person(*X*, '*m*', 30)) is true or exist, and then asks for the answer of the fact youngman(*X*).

Example 2.2 [Horn Clause Conversion]

By using the IDB in Figure 2, the knowledge management program compiles the query child('judy', *Y*) to a serial of subqueries:

STEP	SUBQUERY	USING RULE
1	child('judy', *Y*)	
2	father(*Y*, 'judy')	*r*1
3	edb(father(*Y*, 'judy'))	*r*4
4	father(*Z*, 'judy'))	*r*2
5	edb(father(*Z*, 'judy'))	*r*4
6	marry(*Z*, *Y*)	*r*2
7	edb(marry(*Z*, *Y*))	*r*5

And then convert useful IDB rules to equivalent Horn clauses:

child('judy', Y):-edb(father(Y, 'judy'))
child('judy', Y):-edb(father(Z, 'judy')), edb(marry(Z, Y))

where edb(father(__, __)) means relation father existing in the EDB.

During the Horn Clause Conversion of a given query that reads *Q*, a rule *R* is chosen if the head (left side or goal part) of *Q* is unifiable with the head of *R*. That is, *Q* relates to *R*. The formal definition of 'relates to' will be given in Section 3. Notice that the scalar equality operation between constants in a conventional Database System is extended to the unification operation between Horn clauses in the IDB of the coupled KBMS. Also, a simple read operation in a Database System is extended to a process of Horn Clause Conversion in the IDB. This extension causes new conflict problems that we will introduce in Section 2.2.

Since the conventional Relational DBMS has provided a concurrency control mechanism, in the coupled KBMS we use, we consider only IDB's concurrency control.

2.2 Concurrency Control Problems in an Intensional DataBase (IDB)

If we don't use the concurrency control mechanism to handle transactions that access the IDB concurrently, there are two kinds of problems which may cause incorrectness:

(1) Rule-rule conflicts. This problem arises when two concurrent transactions attempt to perform updates involving the same rules.

Example 2.3. [update same rules]

Let us consider the following two transactions T1 and T2.

T1: assert(*r*1), retract(*r*2), commit(T1)
T2: assert(*r*2), retract(*r*1), commit(T2)

In Figure 3(a) we see a serial schedule, and only *r*2 is true finally. Figure 3(b) shows a non-serializable schedule because both rule *r*1 and rule *r*2 are false. Notice that we will rewrite the schedule 2 as 'T1(assert(*r*1))→T2(assert(*r*2))→T1(retract(*r*2))→commit(T1)→T2(retract(*r*1))→commit(T2)' in the following examples.

Example 2.4. [lost update]

Consider the following two transactions.

T1: youngman(X):-person(X, Y, Z), assert(youngman(X):-person(X, Y, Z-10)), commit(T1)
T2: youngman(X):-person(X, Y, Z), assert(youngman(X):-person(X, Y, Z-10)), commit(T2)

TIME	T1	T2
*t*1	assert(*r*1)	
*t*2	retract(*r*2)	
*t*3	commit(T1)	
*t*4		assert(*r*2)
*t*5		retract(*r*1)
*t*6		commit(T2)

(a) Schedule 1: a serializable schedule.

TIME	T1	T2
*t*1	assert(*r*1)	
*t*2		assert(*r*2)
*t*3	retract(*r*2)	
*t*4	commit(T1)	
*t*5		retract(*r*1)
*t*6		commit(T2)

(b) Schedule 2: a non-serializable schedule.

Fig. 3 Two schedules with transactions that update the same rules.

If we execute T1 and T2 serially (i.e. T1→T2), or execute T2 and T1 serially (i.e. T2→T1), the rule youngman(X):-person(X, Y, Z-20) is true. Figure 4 shows a non-serializable schedule. Transaction T1's update is lost at time *t*5 because transaction T2 updates the rule based on the value seen at time *t*2.

The outcome is non-serializable because rule youngman(X):-person(X, Y, Z-10) is true and rule youngman(X):-person(X, Y, Z-20) is false.

(2) Query-rule conflicts. This problem arises when one transaction asserts or retracts a rule being read by another concurrency transaction. This problem here is an instance of phantom problem that arises when the request of a transaction refers to a rule which does not (does) exist at the time of the request, but is later created (deleted) due to the action of another transaction.

Example 2.5. [error read result]

Take transactions T1 and T2 as following.

T1: child('judy', Y), commit(T1)
T2: assert(*r*8), assert(*r*9), assert(*r*10), retract(*r*4), commit(T2)

where

*r*8 child(X, Y):-father_in_law(Y, X)
*r*9 child(X, Y):-father_in_law(Z, X), marry(Z, Y)
*r*10 father(X, Y):-edb(father(X, Y)), edb(person(X, '*m*',__))

If we execute T1 and T2 serially, the result of transaction T1 is an empty set. If we execute T2 and T1 serially, the result of transaction T1 is {'joe', 'mary'}. In the following schedule 1, the outcome is non-serializable

TIME	T1	T2
$t1$	youngman(X):-person(X, Y, Z)	
$t2$		youngman(X):-person(X, Y, Z)
$t3$	assert(youngman(X):-person(X, Y, Z-10))	
$t4$	commit(T1)	
$t5$		assert(youngman(X):-person(X, Y, Z-10))
$t6$		commit(T2)

Fig. 4 Transaction T1 lost update at time $t5$.

because T1 will produce the result {'joe'}.

Schedule 1: T2(assert($r8$))→T1→T2(assert($r9$))
→T2(assert($r10$))→T2(retract($r4$))
→commit(T2).

The outcome is non-serializable because T1 will produce the result set {'joe'}.

The problem at hand, then, is to handle rule-rule conflicts and query-rule conflicts through the use of an appropriate concurrency control mechanism. These conflicts are similar to the write-write and read-write conflicts in the conventional database concurrency control discussed in [Ullman82]. However, there are two main differences. First, Prolog programs can insert and delete facts (rules) but they cannot modify existing facts (rules). Second, the conventional database system can't store or manipulate rules and then hasn't the problem of rule-rule and query-rule conflicts.

3. A New Two-Phase Locking Based Concurrency Control Algorithm: Query-Rule Locking

The problems we illustrated in Section 2.2 can be solved by an algorithm called Query-Rule Locking. This algorithm is based on two-phase locking [Eswaran76], but it uses the concept of 'relate-cover' to minimize the number of objects to be locked.

3.1 Main Data Structure for Query-Rule Locking Algorithm

In this Query-Rule Locking method, each transaction T_i maintains a local query set Q_i and a local rule set R_i; besides, the concurrency control mechanism maintains a global query set Q_L and a global rule set R_L. When the concurrency control mechanism wants to execute a read operation q_j (or a write operation r_j) of transaction T_i, it first puts q_j in Q_i (or puts r_j in R_i) and checks if this operation will cause any conflict. If not, the concurrency control mechanism grants the request and puts q_j in Q_L (or puts r_j in R_L); otherwise Transaction T_i is blocked. The local query set, local rule set, global query set, and global rule set are described as below:

(1) Local query set, Q_i, is a set of queries (read operations) $\{q_j\}$ which includes those gueries that T_i has executed, and the next query of T_i that the concurrency control mechanism is checking, or has checked and not granted, or is executing.

(2) Local rule set, R_i, is a set of rules $\{r_j\}$ which includes those rules that T_i has asserted or retracted, and the next rule which T_i wants to assert or retract that the concurrency control mechanism is checking, or has checked and not granted, or is executing.

(3) Global query set, Q_L, is a set of $\{q_j, T_i\}$ or $\{q_j^i\}$ of queries q_j of the transaction T_i for which read-only lock has been granted and not yet released.

(4) Global rule set, R_L, is a set of $\{r_j, T_i\}$ or $\{r_j^i\}$ of rules r_j of the transaction T_i for which write-only lock has been granted and not yet released.

Notice that the Query-Rule Locking algorithm set read-only locks on queries (logical objects) that is not physically existed in the IDB.

Example 3.1. [local and global lock set]

Consider transactions T1 and T2 in Example 2.5. When T2 is executed, the following subqueries and associated contents of Q_2, R_2, Q_L, and R_L will be set:

TIME	SUBQUERY	Q_2	R_2	Q_L	R_L
$t1$	assert(r_8)	{ }	$\{r_8\}$	{ }	$\{r_8^2\}$
$t2$	assert(r_9)	{ }	$\{r_8, r_9\}$	{ }	$\{r_8^2, r_9^2\}$
$t3$	assert(r_{10})	{ }	$\{r_8, r_9, r_{10}\}$	{ }	$\{r_8^2, r_9^2, r_{10}^2\}$
$t4$	retract(r_4)	{ }	$\{r_8, r_9, r_{10}, r_4\}$	{ }	$\{r_8^2, r_9^2, r_{10}^2, r_4^2\}$
$t5$	commit (T2)	{ }	{ }	{ }	{ }

Suppose step 1 and step 2 of transaction T2 have completed, and now T1 begins running. For some reasons that we will describe in the following Section 3.2 (i.e. child('judy', Y) relates to rules $r8$ and $r9$), the query child('judy', Y) is not granted. Thus the contents of Q_1, R_1, Q_2, R_2, Q_L, and R_L at that time are:

Q_1	R_1	Q_2	R_2	Q_L	R_L
{child('judy', Y)}	{ }	{ }	$\{r_8, r_9\}$	{ }	$\{r_8^2, r_9^2\}$

3.2 Procedure Relate-Cover for Query-Rule Locking Algorithm

For a query like child('judy', Y), we may set read-only locks on predicate name—child. This will work, but it is equivalent to lock a relation entirely in a relational database system. We may minimize the number of objects to be locked by introducing a concept of 'relate-

relates to	child (X, Y)	child ('john', Y)	child (X, 'joe')	child ('john', 'joe')
child(X, Y)	Y	Y	Y	Y
child('john', Y)	Y	Y	Y	Y
child(X, 'joe')	Y	Y	Y	Y
child('john', 'joe')	Y	Y	Y	Y
child('bob', Y)	Y	N	Y	N
child(X, 'bob')	Y	Y	N	N
child('bob', 'bob')	Y	N	N	N
p(_,_, . . . ,_) $p \neq$ child	N	N	N	N

Fig. 5 Detailed 'relate' matrix for predicate child.

cover	child (X, Y)	child ('john', Y)	child (X, 'joe')	child ('john', 'joe')
child(X, Y)	Y	Y	Y	Y
child('john', Y)	N	Y	N	Y
child(X, 'joe')	N	N	Y	Y
child('john', 'joe')	N	N	N	Y
child('bob', Y)	N	N	N	N
child(X, 'bob')	N	N	N	N
child('bob', 'bob')	N	N	N	N
p(_,_, . . . ,_) $p \neq$ child	N	N	N	N

Fig. 6 Detailed 'cover' matrix for predicate child.

cover'. Now we use the conception of 'relate' to describe the situation wherefrom the query-rule conflict arises, and the conception of 'cover' for another situation wherein one read operation can be executed without checking query-rule conflict, for there exists another read operation which covers it.

Here we only consider a simple case in which an argument of a predicate contains a variable or a constant. This case could be well suitable for most knowledge-base applications. In this simple case, we expect that the concurrency control mechanism we propose can be implemented efficiently as it is based on a notion of 'relate' which is efficiently put into test.

The head of a clause is the goal of that clause. For instance, the head of child(X, Y):-father(Y, X) is child(X, Y). Notice that the head of a horn clause contains a single predicate. We will say that one Prolog clause that reads Q relates to another Prolog clause that writes rule R if the head of Q relates to the head of R.

Definition 3.1. [predicate $P1$ relates to predicate $P2$]
One prolog predicate $P1$ relates to another Prolog predicate $P2$ if

(1) they have the same predicate names, and

(2) suppose both $P1$'s and $P2$'s i^{th} arguments are constant or instantiated variable, then those two arguments have the same value.

Note: $P1$ relates to $P2$ iff $P2$ relates to $P1$.

The detailed 'relate' matrix of predicate child is shown in Figure 5. The contents of the sixth row of the 'relate' matrix are Y, Y, N, and N because child(X, 'bob') relates to child(X, Y) and child('john', Y); and not relates to child(X, 'joe') and child('john', 'joe').

We say that one Prolog clause that reads Q covers another Prolog clause that reads Q' if the head of Q covers the head of Q'.

Definition 3.2. [predicate $P1$ covers predicate $P2$]
One Prolog predicate $P1$ covers another Prolog predicate $P2$ if

(1) they have the same predicate names, and

(2) every constant or instantiated variable in the ith argument position of $P2$ has the same value as the corresponding argument of $P1$ if $P1$'s i^{th} argument is a constant or an instantiated variable.

If (1) one read operation O_1 covers another read operation O_2, and (2) they belong to the same transaction T_i, and (3) T_i already has a read-only lock on O_1, then T_i can execute O_2 without any further checking. The detailed 'cover' matrix of predicate child are shown in Figure 6.

Algorithm Relate-Cover
input: two queries O_1 and O_2
output: if O_1 covers or relates to O_2 then return(YES) else return(NO)
/*The detailed of Algorithm Relate-Cover is shown in [Chen88]*/
end-of-algorithm Relate-Cover.

3.3 Query-Rule Locking Algorithm

We are now ready to introduce the detailed of the Query-Rule Locking algorithm.

Algorithm Query-Rule-Locking
input: a pair (O_j, T_i) where O_j is the jth subquery of transaction T_i.
data structure (as described in Section 2.3):
local query set Q_i, local rule set R_i, $i=1, n$
global query set Q_L, global rule set R_L.
begin
/*Lock phase 1*/
Case 1: O_j asserts or retracts a rule A. /*rule assertion or retraction*/
/*add A to the local rule set of T_i, so that they can be unlocked easily at end of transaction T_i*/
$R_i=R_i+\{A\}$. /*add to local rule set*/
/*check for rule-rule conflicts*/
if ($\exists r^k_m \in R_L$, $r^k_m=A$, $k\neq i$) or
/*using algorithm Relate-Cover to check for query-rule conflict*/
($\exists q^k_m \in Q_L$, Relate-Cover(q^k_m, A)=YES, $k\neq i$)
then block T_i until r^k_m or q^k_m is unlocked
endif.
$R_L=R_L+\{A\}$. /*add to global rule set*/
return and proceed.
end-of-case 1.
Case 2: O_j read a rule A in IDB. /*read operation*/
/*check if T_i already has a read-only lock which covers the read operation to be deal with*/
if ($\exists q_m \in Q_i$) and (Relate-Cover(q_m, A)=YES)
then return and proceed
endif.
$Q_i=Q_i+\{A\}$. /*add to local query set*/
/*check for query-rule conflicts*/
if ($\exists r^k_m \in R_L$, $k\neq i$) and (Relate-Cover(A, r^k_m) =YES)
then block T_i until such rules r^k_m are unlocked
endif.
$Q_L=Q_L+\{A\}$. /*add to global query set*/
return and proceed.
end-of-case 2.
case 3: O_j reads or writes an EDB fact.
/*query, assert or retract an EDB fact*/
return and proceed.
end-of-case 3.
/*Lock phase 2: end of transaction*/
case 4: O_j is commit statement.
$R_L=R_L-R_i$. /*release rule locks*/
$Q_L=Q_L-Q_i$. /*release query locks*/
end-of-case 4.
case 5: livelock and deadlock.
we may use first-come-first-serviced strategy eliminates livelocks, and use wait-for-graph to detect deadlock.
end-of-case 5.
end-of-algorithm Query-Rule Locking.

In the above algorithm, livelock might happen. Livelock is a problem that occurs potentially in any environment where processes execute concurrently. This problem occurs here when one transaction T is waiting for another transaction to release the lock on data item A, but T would wait forever while some other transactions always had a lock on A even though there are unlimited number of time at which T might have been given a chance to lock A. Now we solve the livelock problem by maintaining for each transaction T a queue of transactions waited for T denoted by $w-f(T)$. When transaction T commits, the first transaction in $w-f(T)$ is waken up and executed. This method is known as the first-come-first-served strategy. The $w-f(T)$ can be used to draw the wait-for graph whose nodes are transactions and whose arc $T_1 \rightarrow T_2$ signify that transaction T_1 is waiting to lock an item on which T_2 holds the lock. Every cycle in the wait-for graph indicates a deadlock. Deadlock can be solved by aborting one transaction in the deadlock cycle.

3.4 Example of Query-Rule Locking Algorithm

In the following, we give an example to show how the Query-Rule Locking algorithm is working.

Example 3.2. [using Query-Rule Locking algorithm]
Consider transactions *T*1, *T*2 in example 2.5. Referring to Figure 7, suppose step 1 of T2 has completed at time *t*1; and now T1 begins running at time *t*2 and needs to set read locks on all rules that relate to child('judy', *Y*)—*r*1, *r*2, and *r*8. Since T2 has already locked rule *r*8, T1 will be blocked at time *t*2 until T2 completes and releases its locks at time *t*6 as shown below. Notice that in step 10 we don't lock q_4 because q_2 covers q_4, and q_2 has been locked in time *t*8.

From the above example, we can see that the Query-Rule Locking algorithm based on two-phase locking prevents non-serializable behavior by blocking one of the two conflicting transactions until the other one has finished.

3.5 Proof of Correctness

To show and explain that the Query-Rule Locking algorithm is successful in guaranteeing serializability (i.e. correctness), we use, referring to the argument in [Ullman82], the following steps:

(1) We introduce an Algorithm Serializability-Test which can determine whether a schedule S is serializable.

(2) By using the Algorithm Serializability-Test, we prove that in this locking model if transactions obey the two-phase protocol [Eswaran76], any legal schedule is serializable.

(3) We show that Query-Rule Locking is two-phase locking, and under this circumstance, if transactions obey the Query-Rule Locking algorithm, then any legal schedule is serializable.

The locking model we use in Query-Rule locking is a kind of read-only, write-only model [Ullman82]. Thus if transaction T2 in schedule S reads a rule A which is written by T1:

(1) T1 must precede T2 in any serial schedule equivalent to S.

TIME	T1	T2	Q_1	R_1	Q_2	R_2	Q_L	R_L
$t1$		assert($r8$)	{ }	{ }	{ }	$\{r_8\}$	{ }	$\{r_8^2\}$
$t2$	q_1		$\{q_1\}$	{ }	{ }	$\{r_8\}$	{ }	$\{r_8^2\}$
$t3$		assert(r_9)	$\{q_1\}$	{ }	{ }	$\{r_8, r_9\}$	{ }	$\{r_8^2, r_9^2\}$
$t4$		assert(r_{10})	$\{q_1\}$	{ }	{ }	$\{r_8, r_9, r_{10}\}$	{ }	$\{r_8^2, r_9^2, r_{10}^2\}$
$t5$		retract(r_4)	$\{q_1\}$	{ }	{ }	$\{r_8, r_9, r_{10}, r_4\}$	{ }	$\{r_8^2, r_9^2, r_{10}^2, r_4^2\}$
$t6$		commit(T2)	$\{q_1\}$	{ }	{ }	{ }	{ }	{ }
$t7$	q_1		$\{q_1\}$	{ }	{ }	{ }	$\{q_1^1\}$	{ }
$t8$	q_2		$\{q_1, q_2\}$	{ }	{ }	{ }	$\{q_1^1, q_2^1\}$	{ }
$t9$	q_3		$\{q_1, q_2\}$	{ }	{ }	{ }	$\{q_1^1, q_2^1\}$	{ }
$t10$	q_4		$\{q_1, q_2\}$	{ }	{ }	{ }	$\{q_1^1, q_2^1\}$	{ }
$t11$	q_5		$\{q_1, q_2\}$	{ }	{ }	{ }	$\{q_1^1, q_2^1\}$	{ }
$t12$	q_6		$\{q_1, q_2, q_6\}$	{ }	{ }	{ }	$\{q_1^1, q_2^1, q_6^1\}$	{ }
$t13$	q_7		$\{q_1, q_2, q_6\}$	{ }	{ }	{ }	$\{q_1^1, q_2^1, q_6^1\}$	{ }
$t14$	commit(T1)	{ }	{ }	{ }	{ }	{ }	{ }	

q_1=child('judy', Y), q_2=father(Y, 'judy'),
q_3=edb(father(Y, 'judy')), q_4=father(Z, 'judy'),
q_5=edb(father(Z, 'judy')), q_6=marry(Z, Y)
q_7=edb(marry(Z, Y)).

Fig. 7 Execution sequence arranged by Query-Rule Locking algorithm.

(2) If T3 is a transaction that writes A, then in any serial schedule equivalent to S, T3 may either precede T1 or follow T2, but may not appear between T1 and T2.

First, consider how to decide whether a schedule is serializable.

Algorithm Serializability-Test
/*Serializability test for schedules with read-only and write-only locks (modified from Algorithm 11.3 of [Ullman82])*/
input: A schedule S for a set of transactions T_1, $T_2, \ldots, T_K$.
output: A determination whether S is serializable, and if so, an equivalent serial schedule is output.
begin

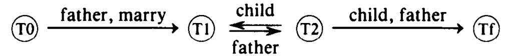
(1) Augment S by appending to the beginning a sequence of steps in which a dummy transaction T_0 writes each rule appearing in S and by appending to the end steps in which dummy transaction T_f reads each of such rule.

(2) Begin the creation of a polygraph P with one node for each transaction, including T_0 and T_f. Temporarily place an arc from T_i to T_j whenever a read operation of T_j relates to a rule A that in the augmented S was last written by T_i.

(3) For each remaining arc$T_i \rightarrow T_j$, and for each rule A such that a read operation of T_j relates to a rule of A written by T_i, consider each other transaction $T \neq T_0$ that also writes A, if $T_i = T_0$ and $T_j = T_f$ add no arcs; if $T_i = T_0$ but $T_j \neq T_f$ add the arc $T_j \rightarrow T$; if $T_j = T_f$, but $T_i \neq T_0$, add the arc $T \rightarrow T_i$; if $T_i \neq T_0$, and $T_j \neq T_f$, then introduce the arc pair $(T \rightarrow T_i,\ T_j \rightarrow T)$.

(4) Determine whether the resulting polygraph P is acyclic. If P is acyclic, let G be an acyclic graph formed from P by choosing an arc from each pair. Then any topological sort of G, with T_0 and T_f removed, represents a serial schedule equivalent to S. If P is not acyclic, then no serial schedule equivalent to S exists.
end-of-algorithm-Serializability-Test.

Example 3.3. [test for serializability]
The final polygraph P of Schedule 1 of Example 2.5 constructed by Algorithm Serializability-Test is shown below:



Since P is not acyclic, no serial schedule equivalent to Schedule 1 exists.

Theorem 3.5.1. Algorithm Serializability-Test correctly determines if a schedule is serializable.

Lemma 3.5.1. In this read-only write-only locking model, if transactions obey the two-phase protocol, then any legal schedule is serializable.

Lemma 3.5.2. Query-Rule Locking is two-phase locking algorithm.

The detailed proofs of Theorem 3.5.1, Lemma 3.5.1, and Lemma 3.5.2 are shown in [Chen88].

Theorem 3.5.2. Algorithm Query-Rule Locking is successful in guaranteeing serializability (i.e. correctness).

Proof. By Lemma 3.5.1, two-phase locking guarantees serializability. By Lemma 3.5.2, Query-Rule Locking is a two-phase locking. Then we have Algorithm Query-Rule Locking guarantees serializability. Q.E.D.

4. Conclusions and Further Works

We have proposed a concurrency control mechanism for use in a coupled KBMS. The proposed Query-Rule

Locking algorithm handles rule-rule and query-rule conflicts that may arise when we concurrently execute Prolog transactions with a very large shared Intentional Database (IDB) which contains knowledge and expressed in Horn clauses. The Query-Rule Locking algorithm has been proved to guarantee serializability as well as correctness. The concurrency control in Extensional Database (EDB) which contains fact as traditional relation is similar to the mechanism in conventional Relational Database management System.

The concurrency control mechanism we have proposed in this paper can be extended and used in other KBMS models. For example, the Relational Knowledge Base (RKB) in [Morita86] which integrates IDB and EDB as term relations. Another example is that Prolog with rule updates and fact updates can be directly handled by the mechanisms which combine what we have proposed and that in [Carey84].

Finally, new concurrency control strategies not constrained by the current database techniques especially suited for KBMSs, various kinds of concurrency control mechanisms in various KBMS models, and the analysis of the advantages/disadvantages of them should be worth studying.

This research was supported by the National Science Council, Taiwan, R. O. C., under contract: NSC77-0408-E009-20 (1988).

References

[Brodie85] BRODIE, N. L. and MYLOPULOS, J. On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies. Springer-Verlag (1985).

[Carey84] CAREY, M. J., DEWITT, D. J. and GRAEFE, G. Mechanisms for Concurrency Control and Recovery in Prolog—A Proposal. Proceeding of First International Workshop on Expert Database System (1984), 271-291.

[Chen88] CHEN, Y. J. and YANG, W. P. A Concurrency Control Mechanism in a Coupling Knowledge-Base Management System. Proceeding of Twenty-Second Annual Conference on *Inf. Sci.* and Syst., Princeton, NJ, USA (Mar. 1988).

[Eswaran76] ESWARAN, K. P., GRAY, J. N., LORIE, R. A. and TRAIGER, I. L. The Notions of Consistency and Predicate Locks in a Data Base System. *Comm. ACM* **19**, No. 11 (November 1976).

[Harmon85] HARMON, P. and KING, D. Expert Systems (1985).

[Itoh86] ITOH, H. Research and Development on Knowledge Base Systems at ICOT. Proceeding of the Twelfth International Conference on Very Large Data Bases (1986), 437-445.

[Kastner86] KASTNER, J. K. et al. A Continuous Real-Time Expert System for Computer Operating. Proceeding of the International Conference on Knowledge Base Systems, London (1986), 89-114.

[Li84] LI, D. A Prolog Database System. John Wiley & Sons Inc. (1984).

[Morita86] MORITA, Y., YOKOTA, H., NISHIDA, K. and ITOH, H. Retrieve-By-Unification Operation on a Relational Knowledge Base. In Proceeding of the Twelfth International Conference on Vary Large Data Base (1986), 52-59.

[Taylor86] TAYLOR, J. M. Expert Systems: Where Do We Go From Here? Proceeding of the International Conference on Knowledge Based Systems, London (1986), 313-324.

[Ullman82] ULLMAN, J. D. Principles of Database Systems. Second Edition, Pitman Publishing Limited (1982).

(Received January 25, 1988; revised May 10, 1989)