International Journal of Computer and Communication Technology

Volume 5 | Issue 4 Article 2

October 2014

A Correctness Criterion for Eager Approach Validation for Transactional Memory System

Ravi Kumar
Department of Computer Science and Engineering Indian Institute of Technology Patna, Patna-800013,, r.kumar@iitp.ac.in

Sathya Peri
Department of Computer Science and Engineering Indian Institute of Technology Patna, Patna-800013, Bihar, sathya@iitp.ac.in

A Correctness Criterion for Eager Approach Validation for Transactional Memory System

Ravi Kumar
Department of Computer Science and Engineering
Indian Institute of Technology Patna, Patna-800013,
Email: r.kumar@iitp.ac.in

Sathya Peri
Department of Computer Science and Engineering
Indian Institute of Technology Patna, Patna-800013, Bihar
Email: sathya@iitp.ac.in

Abstract : With rise of multicore systems, software transactional memory (STM) has garnered significant interest as an elegant alternative for developing concurrent code. A (memory) transaction is an unit of code in execution in memory. A software transactional memory system (STM) ensures that a transaction appears either to execute atomically (even in presence of other concurrent transactions) or to never have executed at all. To achieve this property, a commonly used approach by STM systems is eager validation approach. In this approach, when a transaction performs a write operation the update is immediately written to the memory and is visible to other transactions. If the transaction aborts, then the writes performed by it previously are undone with the aid of undo logs. In this paper, we have presented a new correctness criterion EAC for eager approach validation systems. We have developed a STM system that implements this criterion. The STM system uses database recovery criterion strictness and the notion of conflict graphs used for testing conflict serializability.

I. INTRODUCTION

In the recent years with rise of multicore systems, parallel programming has become very important. As a result, software transactional memory has garnered significant interest as an elegant alternative for developing concurrent code. Software transactions are units of execution in memory which enable concurrent threads to execute seamlessly [3]. Traditionally locks have been used for developing parallel programs. But programming with locks has many disadvantages such as deadlocks, priority inversion etc. These disadvantages make it difficult to build scalable software systems. Importantly, lock based software components are difficult to compose i.e. building larger software systems using simpler software components [2]. Software transactions address many of the short comings of lock based systems.

A (memory) transaction is a unit of code in execution in memory. A *software transactional memory system* (STM) ensures that a transaction appears either to execute atomically (even in presence of other concurrent Transactions or to never have executed at all. If a transaction executes to completion then it is committed and its effects are visible to other transactions. Otherwise it is aborted and none of its effects are visible to other transactions. Thus the values written by a live (incomplete) transaction to the memory are not visible to other transactions.

To achieve this property, two approaches are commonly used by STM system: eager and lazy approach. In lazy approach, each transaction tX maintains a local buffer for every data-item d it accesses. Any write onto a d by tX is first written to the local buffer associated with d. When tX commits, the updates are written to the memory. If the tX aborts then the contents of the buffer are discarded. In case of eager approach, every update is written directly to the memory and is immediately visible to other transactions. If the transaction aborts then the writes performed by it are undone with the aid of undo logs. In this paper we focus on eager approach.

When transactions accessing common data-items execute concurrently it is imperative that they execute correctly. Hence, it is important to characterize the right correctness criteria. The correctness criterion used in traditional databases is *serializability* [5]. But serializability concerns itself only with the events of committed transactions. It does not require that the aborted transactions read consistent values. A commonly accepted correctness requirement for concurrent executions in STM systems is that all transactions including aborted ones read consistent values [1] and is referred to as *allreadsconsistency*[7]. *Opacity* [1] is a correctness criterion that satisfies all-reads-consistency.
In this paper, we have presented a new correctness criterion EAC for eager approach validation systems. We have developed a STM system that implements this criterion. For implementing this criterion, we used database recovery criterion *Strictness* [8] and the notion of conflict graphs used for testing conflict serializability.

Roadmap: Section II presents background and system model, Section III presents EAC, a correctness criterion for eager approach validation systems, and the algorithm implementing it. Section IV concludes this paper and finally Section V describes the pseudo code of the algorithm.

II. BACKGROUND AND SYSTEM MODEL

In a multicore system, threads execute concurrently and invoke transactions whenever they want to access shared data. A *transaction* is finite sequence of instructions executed. Different transactions execute in an interleaved manner, to access shared variables. The operations invoked by transactions are read and write operations. We assume that these are operations executed instantaneously. But it is necessary that the outcomes of operations of different interleaving transactions on the shared memories leave the system in a consistent state. The STM system is a layer between parallel interleaving transactions and shared memory which ensures memory is in a consistent state.

When a transaction tX performs a read operation on a data item d it is denoted as rX(d). The write on d is denoted as rX(d). If the date-item is not relevant, we simply ignore it. When a transaction terminates, the STM validates the actions of the transaction. If a transaction tX executes successfully to completion, it terminates with a commit operation denoted as cX. The effects of a committed transaction are visible to other transactions. Otherwise it aborts, aX and none of it effects are visible to other transactions that follow it. Abort and commit operations are called terminal operations.

Some properties of the STM presented in this paper are:

a) Schedule and History: A schedule is a totally ordered sequence (in real time order) of operations and terminal operations of transactions in a computation. These operations are referred to as events of the schedule. Schedule includes both terminated and live (which have not yet terminated) transactions. A history consists only of terminated transactions. We will use schedules in the examples in the rest of the document. The example of a schedule is:

S1 : r1(x) r2(y) w1(y) w2(z) c1 c2 r3(x) w3(x) c3.

For a schedule S, we denote its transactions as trans(S), its operations are ops(S). For S1, trans(S) = t1, t2. We capture the order among the events of the schedule S by <S. Thus, r1(x) <S1 w1(y). The relation <S partially orders transactions in S. For instance, in S1, t1 <S1 t3 and t2 <S1 t3. But nothing can be said about the order of t1 and t2 since they execute in interleaved manner.

b) Blind read and write: Blind read implies that a transaction reads from other live transaction while blind write means that a transaction writes onto a shared data without reading it.

c) Transaction Validation - Eager approach: A STM system can employ two approaches for transaction validation: lazy and eager. In lazy approach, each transaction tX maintains a local buffer for every data-item d it accesses. Any write onto a d by tX is first written to the local buffer associated with d. When tX commits, the updates are written to the memory. If the tX aborts then the contents of the buffer are discarded. In case of eager approach, every update is written directly to the memory and is immediately visible to other transactions. Normally in case of eager approach, transactions maintain undo logs to take care of aborts. When a transaction aborts, the undo logs help restore the memory to the previous state and completely remove the affects of the aborted transaction.
The STM presented in this paper is based on eager approach.

d) *All-Reads-Consistency*: An important property required of STM systems is that every transaction including aborted transactions read consistent values [1]. We denote this property as all reads consistency. For a read operation rX(d) in a schedule S we denote its lastWrite [6], [7]as the previous closest write wY (d) on d in the S. It is the write operation whose write value is read by rX.

Opacity [1] is a commonly used correctness criterion by STM systems that ensures all-reads-consistency. A schedule S consisting of transactions involving read and write operations is be said to be opaque if there exists a serial schedule SS such that (i) the set of transactions in S are same as in SS (ii) the order of transaction execution in S and SS are the same. If a transaction tX completes before tY begins in S then tX also completes before tY begins in SS (iii) the *lastWrites* for every read operation (including the reads of the aborted transactions) are the same in S and SS. By ensuring all reads including reads of aborted transactions be the same as in serial schedule, opacity follows all-reads-consistency. Imbs and Raynal gave an implementation of opacity with read and write operations for lazy approach [4].

In addition to these conditions, opacity also requires that each aborted transaction read only from committed transactions. In other words no aborted transaction tj can read from another aborted transaction ti (which possibly was live during the read operation of tj).

e) Conflict Serializability: *Serializability* (View Serializability or VSR) is a correctness criterion for databases which concerned it only with committed transactions. But testing for membership of VSR has been proved to be NP-Complete [5]. Conflict Serializability

(CSR) [8] is a well known subclass of serializability (VSR) whose membership can be tested in polynomial time.

Two operations of a schedule S are said to be in conflict if they access same shared data and at least one of them updates the data. On the basis of conflicts the class criterion CSR can be defined. A schedule S is said to be in CSR if there exists a serial schedule SS such that (1) the set of operations in S and SS are the same (2) the set of conflicts in S and SS are the same.

Checking for membership of CSR can be performed in polynomial time using conflict graphs [8]. For a schedule S, a conflict graph G(S) can be constructed as follows: (i)Vertices: all the committed transactions of S are the vertices of G(S). The vertex for a transaction tI is denoted as vI. (ii)Edges: If two transactions, tI, tJ , have operations that conflict then there is an edge from vI to vJ . If the resulting graph G(S) constructed is acyclic, then the schedule S is in CSR.

Opacity generalizes serializability by considering even aborted transactions. Similar to conflict serializability, the class conflict opacity or CO can be defined whose membership can be tested in polynomial time. The class CO includes the conditions (1) and (2) of CSR described above. In addition to that CO also imposes transaction order like opacity: (3) the order of transaction execution in S and SS are the same. If a transaction tX completes before tY begins in S then tX also completes before tY begins in SS. The membership of CO can be tested similar to CSR by constructing the conflict graph described as above and checking for it acyclicity.

f) Strictness: Strictness is a correctness criterion defined in the context of databases for ensuring recoverability. A schedule S is strict if the following holds for all transaction tI □ trans(S) and for all pI (x) □ op(tI), p □ {r,w}: if wJ(x) <s pI(x), i != j, then aJ <s pI(x) V cJ <s pI(x). Let ST denotes the class of all strict schedules. Informally, ST says that if a transaction tJ writes onto a data-item d in a schedule S, then tJ must abort or commit before any other transaction tI can read or write onto d. Consider the schedules
S2 = w1(x) w1(y) r2(u) w2(x) w1(z)c1 r2(y) w2(y) w3(u) c3 c2 and
S3 = w1(x) w1(y) r2(u) w1(z)c1 w2(x)r2(y) w2(y) w3(u) c3 c2.
It can be clearly seen that S2 !□ ST because of absence of c1/a1 between w1(x) and w2(x) while S2 □ ST.

III. EAGER APPROACH CONSISTENCY

We define a new consistency criterion which is well suited for eager approach validation: *Eager Approach Consistency* or EAC. A history S is said to be in EAC if: (1A) if transaction tJ reads x from transaction tI in S, and if tI is aborted then tJ should also be aborted. (1B) there exists a sequential history SS such that (1B.1) the set of transactions in S are same as in SS (1B.2) the order of transaction execution in S and SS are the same.(1B.3) the lastWrites for every read operation (including the reads of the aborted transactions) are the same in S and SS.

It can be seen that EAC generalizes opacity by allowing transactions to read even from aborted transactions. The rule (1B) is same as opacity. Consider the following schedule which is in EAC but is not in opacity:

S4 = r1(x) r2(y) w1(z) r2(z) a1 a2

Thus, EAC allows more flexibility for schedulers. Next, consider the following schedule:

S5 = rI (y)wI (x)rJ (x)aIwJ (x)cJ

The schedule S5 exhibits dirty-read problem [8]. The read rJ of tJ reads x from tI . But transaction tI is aborted and its writes are undone. The transaction tJ writes x based on dirty-read rJ (x) which is undesirable. The rule (1A) of EAC disallows such schedules. To implement (1A) of EAC efficiently we employ strictness or ST in our STM system which accepts only schedules that are in ST. Next, consider a schedule S6 that satisfies the rule (1A) of EAC.

S6 = rI (x) rJ (x) wJ (x) cI wI (x)

In S6, both tI and tJ read from terminated transaction but does not exist any serial schedule such that the lastWrites of every read is same in both schedules. Thus, S6 does not satisfy rule (1B) of EAC. The rule (1B) can be ensured by any algorithm that satisfies opacity such as [4]. We ensure this rule by employing conflict opacity (CO) which allows for more concurrency.

g) Relationship between Strictness and Conflict Opacity: Strictness and Conflict Serializability are independent classes. Then not all members of ST also belong to CO and vice-versa. For example:

S7 = r1(x) w1(x) r2(x) c1 c2,

and

S8 = r1(x) w2(x) c2 w1(x) c1

Here, S7 □ CSR but S7 !□ ST while S8 □ ST but S8 !□ CO.

h) Strictness and Conflict Opacity ensures EAC: As described earlier, it can be seen that Strictness and Conflict Opacity ensures EAC. From the definition of strictness, we get that strictness ensures rule '(1A)' of EAC. In fact strictness implements a subset of schedules allowed by rule '(1A)'. Strictness does not allow a transaction to read from aborted transactions while according to rule '(1A)', a transaction can read from aborted transactions. Given below is an example for this:

S9 = rI (x) wI (x) rJ (x) aI aJ

According to rule '(1A)', schedules of type S9 can be possible but strictness does not allow transaction tJ to read

from live transaction and abort tJ . Conflict opacity takes care of rule '(1B)'. Thus ST ∩ CSR □ EAC.

A. Implementation

In this subsection, we will discuss our STM implementation of EAC. There are many tradeoffs in implementing a STM, can be either memory efficient or time efficient. To make an STM which is optimal in both memory and time complexity is really a difficult task. In this paper, an STM is based on page model and uses eager writes. Since consistency of system is important, so before executing any step, TM system must check the consistency. This section describes programming model, data structure and algorithms used to implement the STM.

1) Programming Model:

A process can lead to many parallel threads, which invoke transactions. Whenever a transaction starts, terminates or tries to execute any operations, these functions are called:

BEGIN TRANSACTION: Every transaction is started by executing BEGIN TRANSACTION, which returns transaction identifier which can further be used by transaction to invoke another function.

TRY TO COMMIT: Whenever a transaction tries to commit, it has to invoke TRY TO COMMIT function. The identifier of calling transaction is passed as a parameter.

ABORT: Whenever some consistencies arises or execution of a transaction leads to no more serializability of the history then ABORT method is invoked for that transaction. The identifier of calling transaction is passed as a parameter.

STM READ: To read any shared memory, transaction has to invoke STM READ method. The parameters passed to this method are identifier of invoking transaction and name of the base object (variable) to be read. The method return the read structure after read is allowed by transaction manager.

STM WRITE: If any transaction wants to update any shared memory, transaction has to invoke STM WRITE method. The parameters passed to this method are identifier of invoking transaction, name of the base object (variable) to be updated, current value of the base object and the new value of the base object. The method return nothing after write is allowed by transaction manager.

Whenever any operation is called, STM attempts to satisfy the correctness and record the accesses.

2) Data Structure:

Data structure plays an important role in time and space complexity. The data structures used for the STM are linked list and vector.

Transaction: Information about transactions is kept in form of a linked list. Each node (i.e. transaction) keeps track of its ID, a Boolean variable (representing active or terminated node), a pointer vector of base objects which it updates (to undo if it aborts), list of all its head transactions (to keep track of edges in conflict graph).Transaction also keeps track of all the base objects that it has accessed to read and write in two different lists.

Base object: Base object is the shared data accessed by transaction. Base objects are stored as linked list. Each node represents a base object and keep track of its name, a Boolean variable (representing whether it is updated by any live transaction or not, helpful in strictness), another Boolean variable (representing whether the base object has ever been updated or not, initially set to false) and a list of all the transactions which has accessed it along with the operation operated (either read or write) on it.

3) Algorithm for ensuring Strictness:

Strictness implies that any live transaction should not conflict with other live transaction.

(3A) Each base object has a Boolean variable (initially set to false) which represents whether it is updated by any live transaction or not.

(3B) If any transaction updates it, variable is set to true.

(3C) If any transaction commits or aborts then TM system set the Boolean variable associated to each base object updated by the transaction to false.

So if any transaction tries to access any base object then transaction manager first check whether the Boolean variable of the base object is true or false. If variable is true then TM system should abort the transaction otherwise pass the operation to Data Manager and set the variable to true (if operation is write). For example:

Let

$$SH = r1(x)\ w2(x)\ c2\ w1(x)\ c1$$

(1) When, transaction t1 invoke r1(x), then TM system create a node for x and set the Boolean to false, and since variable is false so read operation is passed to Data Manager.

(2) When w2(x) is invoked by transaction t2, and since Boolean variable for x is still false, so TM system pass the operation to DM but set the variable to true.

(3) Now when t2 tries to commit, then TM system set the

Boolean variable associated with x to false (as x is updated by t2).

(4) And since variable associated with x is false, so TM system pass the w1(x) operation invoked by t1 to Data Manager and set the variable to true.

(5) And finally, when t1 tries to commit, then TM system set the Boolean variable associated with x to false (as x is updated by t1).

This is the way of implementing strictness.

4) Algorithm for implementing Conflict Opacity:
Conflict opacity takes care of preserving order and serializability. It is implemented using serializability graph testing (SGT). In the serializability graph G(S) of a schedule S, nodes represent transactions and directed edges represent conflicts between transactions. Serializability graph testing implies (1) testing of cycle(s), if any, in the graph, (2) updating the graph, (3) remove write-edges (edges associated with the write operation) of the aborted transactions, (4) order preservation and (5) removing unnecessary nodes and edges. Cycle(s) represents, schedule S is not in CSR. Acyclic conflict graph G(S) implies that there exists a sequential schedule SS which is conflict equivalent to given schedule and such serial schedule can be found by topologically sort the graph.

The algorithm for SGT takes use of graph traversal algorithm. Every time an operation is invoked TM system first checks if any conflict(s) arises or not. If any conflict arises, then TM system checks whether even after adding the corresponding directed edge, acyclicity is still maintained or not. If the conflict graph is still acyclic, then the operation is passed to Data Manager System and the graph is updated by adding the edge, otherwise transaction is aborted.

Since CO also includes aborted transactions but only read steps. So it is necessary to remove the edge, if any, corresponding to write operation of aborted transaction that is edge representing conflict in which one operation is write operation of the aborted transaction.

In process, total number of transactions can be of order tens of thousands each is represented by a vertex in the conflict graph. And thus lead to a big set of vertices and edges in graph. It is a daunting task to traverse whole graph with so many number of vertices. So it would be better if unnecessary nodes can be removed somehow and so all its incident edges. Unnecessary nodes represent here the nodes which can never play role in forming any cycle. Given below is a rule to remove nodes in a graph:

(4A) Terminated source node(s) of the graph can be removed. And so the outgoing edge(s) if any.
Terminated source node implies terminated transaction that does not have any incoming edges from other nodes. So this node can never play any role in forming cycle(s) in graph.

Order can also be preserved using SGT. Preserving order simply implies that if a transaction, say tI , completely precedes other transaction, say tJ , in a schedule S then in the equivalent conflict serial schedule SS also tI should precedes tJ . A transaction completely precedes other transaction in a schedule means that in the topologically sorting of the conflict graph of the schedule they should appear in same order or can be said that there is an edge between them. So to preserve the order between transactions, (4B) when a new transaction begins, traverse the list of terminated transactions and add a directed edge between every node representing terminated transactions and node representing currently started transaction. For example:
Let

SI = rI (x) wJ (x) cJ rK(y) cI aK

In SI, when transaction tK starts after transaction tJ commits add a directed edge from node tJ to tK

IV. DISCUSSION AND CONCLUSION

In this paper we presented a new correctness criterion EAC for eager approach validation STM systems. This criterion is generalization of opacity. It allows more flexibility by allowing an aborted transaction to read from another transaction. We also gave the implementation of a eager validation STM system that implements EAC. Our STM system implements EAC by accepting only those schedules that are in both ST and CO.
As a part of the future work, we are planning to explore to see if we can find a more efficient criterion that implements rule (1A) of EAC. Coming to rule (1B), we are looking for a more efficient implementation of CO.

REFERENCES

[1] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.

[2] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[3] Maurice Herlihy and J. Eliot B.Moss. *Transactional memory: architectural support for lock-free data structures*. SIGARCH Comput. Archit. News, 21(2):289–300, 1993.

[4] Damien Imbs and Michel Raynal. *A lock-based stm protocol that satisfies opacity and progressiveness*. In *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 226–245, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Christos H. Papadimitriou. *The serializability of concurrent database updates*. J. ACM, 26(4):631–653, 1979.

[6] Sathya Peri and K.Vidyasankar. Correctness of concurrent executions of closed nested transactions in transactional memory systems. In *12th International Conference on Distributed Computing and Networking*, pages 95–106, 2011.

[7] Sathya Peri and K.Vidyasankar. An efficient scheduler for closed nested transactions that satisfies all-reads-consistency and non-interference. In *13th International Conference on Distributed Computing and Networking*, 2012.

[8] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

V. PSEUDOCODE

Algorithm 1 STM Read(): A transaction tI performs a read operation rI (d). The return value is read data or abort.

```
1: procedure STM READ(tI , d)
2:          if (tI.active == false) then
3:                    return abort;
4:          end if
5:          if (d.update flag == true) then
6:                    return abort;
7:          end if
8:          for all (node of transaction lists which has
            accessed) do
9:                    if (node.write flag == true) then
10:                            node.head.add(tI );
11:                   end if
12:         end for
13:         //traverse the graph for cycle(s)
14:         stack.push(tI );
15:          while (!stack.isEmpty()) do
16:                   node = stack.pop();
17:                   node.visited = true;
18:                   for all (headNode of node) do
19:                            if (headNode.visited! = true)
            then
20:                                    stack.push(headNode);
21:                           end if
22:                           if (headNode == tI ) then
23:                                    return abort;
24:                           end if
25:                   end for
26:         end while
27: .       //add d to the read base-object list of tI .
28:          tI.rbase object:add(d);
29: .       //add node(tI ) to the transaction list of d and set
            the write flag to false.
30:         d.transaction list.add(node(tI ));
31:         node(tI ).write flag = false;
32:         return d;
33: end procedure
```

Algorithm 2 BEGIN TRANSACTION(): A transaction tn with identifier n is started and its ID is returned.

```
1:  procedure BEGIN TRANSACTION
2:          //add a node newNode to the list of transaction
3:          newNode.t id = t id + +;
4:          newNode.active = true;
5:          // traverse the list of terminated transactions and
add newNode.t_id to the head list of each transaction
6:          for all (node of terminated transaction) do
7:                    node.head.add(newNode.t id);
8:          end for
9:          return newNode.t id;
10: end procedure
```

Algorithm 3 STM Write(): A transaction tI performs a write operation wI (d). The return value are OK or abort.

```
1: procedure STM WRITE(tI , d, oldd, newd)
2:          if (tI.active == false) then
3:                    return abort;
4:          end if
5:          if (d.update flag == true) then
6:                    return abort;
7:          end if
8:          for all (node of transaction lists which has
accessed) do
9:                    node.head.add(tI );
10:         end for
11: .       //traverse the graph for cycle(s)
12:         stack:push(tI );
13:         while (!stack.isEmpty()) do
14:                   node = stack.pop();
15:                   node.visited = true;
16:                   for all (headNode of node) do
17:                            if (headNode.visited! = true)
then
18:                                    stack.push(headNode);
19:                           end if
20:                           if (headNode == tI ) then
21:                                    return abort;
22:                           end if
23:                   end for
24:         end while
25: .       //add d to the write base-object list of tI .
26:          tI.wbase object:add(d);
27:         d.update flag = true;
28: .       //add node(tI ) to the transaction list of d and set
the write flag to true.
29:         d.transaction list.add(node(tI ));
30:          node(tI ).write flag = true;
31:         return OK;
32: end procedure
```

Algorithm 4 Abort(): Whenever a transaction is aborted by TM system or transaction tries to abort. Return value is OK.

```
1: procedure ABORT(tI )
2:          //Remove all the write edges of tI .
3:          for all (d of tI:wbase object) do
4:                    for all (node of d.transaction list) do
5:                              node.head.remove(tI );
6:                    end for
7:          end for
8:          for all (d of tI.rbase object) do
9:                    for all (node of d.transaction list) do
10:                             if (node.write flag == true V
Node.getHead(tI )! = true) then
11:                             node.head.add(tI );
12:                             end if
13:                   end for
14:         end for
15:         //set the update flag of each write base object d
updated by tI to false.
16:         for all (d of tI.wbase object) do
17:                   d.update flag = false;
18:         end for
19:         tI.active = false;
20:         return OK;
21: end procedure
```

Algorithm 5 try to commit(): A transaction tI tries to commit. OK is returned.

```
1: procedure TRY TO COMMIT(tI )
2:  //set the update flag of each write base object d
updated by tI to false.
3:          for all (d of tI.wbase object) do
4:                    d.update flag = false;
5:          end for
6:          tI.active = false;
7:          return OK;
8: end procedure
```