Formal Verification of an Optimistic Concurrency Control Algorithm using SPIN

Achraf Makni
Faculté des Sciences Economiques et de Gestion de Sfax, Tunisia
Achraf.makni@fsegs.rnu.tn

Rafik Bouaziz
Faculté des Sciences Economiques et de Gestion de Sfax, Tunisia
Raf.bouaziz@fsegs.rnu.tn

Faïez Gargouri
Institut Supérieur d'Informatique et de Multimédia de Sfax, Tunisia
Faïez.gargouri@fsegs.rnu.tn

Abstract

To contribute to the promotion of concurrency controllers for temporal databases, we propose in this paper to formally check the access concurrency control algorithm proposed in [4], using SPIN. This algorithm is based on the optimistic approach and must guarantee strong consistency for transaction time relations. SPIN provides a software model checking with a powerful tool to detect errors. It is an appropriate tool for analyzing the logical consistency of concurrent systems. The main target consists of retrieving and correcting blocking error type, on the one hand, and ensuring the validity of the considered system properties specified by temporal logic formulae, on the other hand.

1. Introduction

The critical mission of the concurrency control (CC) access to DB consists in guaranteeing the consistency of any DB. The CC is then an essential component in a database management system. It must guarantee that any simultaneous execution of transactions produces the same results as a sequential execution [10].

The CC takes new dimensions when applied to temporal DB (TDB). The objective of TDB is the management of the data history. This paper deals with transaction time relations (TTR). The main goal of TTR consists in providing for applications, not only the current data, but also all the previous DB states which succeed in time. To be able to maintain these states, the updating operations should not be destructive. The TTR store data versions by stamping them using the two following physical times:

- *transaction start time*: the execution time of the transaction which inserts the corresponding tuple. This time is *a priori* known.
- *transaction end time*: the execution time of the transaction which updates or removes the considered tuple. It can not be *a priori* known.

In order to have transaction consistent views of the TDB [9] [15], the CC mission must be reinforced to ensure the strong consistency (SC). To this purpose, the CC of such environments must guarantee that any simultaneous execution of transactions produces the same results of the sequential execution of these transactions in their strict order of arrival [16].

We propose in this paper to formally check the optimistic CC algorithm OCCA_SC/TTR [4], which ensures the SC for TTR. For this purpose, we choose to use the SPIN tool [14], which is one of the most powerful model checkers. It is an appropriate tool for analysing the logical consistency of concurrent systems, especially for data communication protocols. SPIN is largely used, not only in research areas, since it is freeware, but also in industrial areas [2] [3] [6] [11] [13].

The systems analyzed by SPIN are described with the PROMELA language (PROcess MEta LAnguage), which is a specification language for finite state systems. Under PROMELA, a system is represented by a set of parallel processes which communicate via global variables and/or communication channels. PROMELA also allows checking properties which are specified in linear temporal logic (LTL).

This paper is organized as follows. We describe in section 2 the structure of the OCCA_SC/TTR algorithm. In section 3, we explain the PROMELA specification used as formal model, and how we express, in PROMELA, the components of our system. After that, we report in section 4 some results of the verification step. A discussion about related works constitutes the object of section 5. Section 6 concludes the paper and gives perspectives for further research.

2. Description of OCCA_SC/TTR

The CC methods are classified in two main categories: pessimistic methods and optimistic ones. The consistency checking is carried out, in the case of pessimistic methods, at the time of each transaction operation. It is carried out only at the end of each transaction, in the case of optimistic methods. For TTR, we can find, in the literature, some CC algorithms based on the pessimistic approach [5] [8] [9]. But, to our knowledge, only the OCCA_SC/TTR algorithm [4] was proposed to study the CC for these relations according to the optimistic approach in order to ensure the strong consistency of the DB. In the following subsections, we briefly present the OCCA_SC/TTR algorithm.

2.1. Objectives of OCCA_SC/TTR algorithm

The OCCA_SC/TTR [4] algorithm allows:

- Maintaining the SC in a TTR environment.
- Exploiting the TTR past versions instead of managing specific versions for the CC.
- Minimizing the abortion degree of transactions.
- Avoiding the starvation problem.
- Detecting conflicts as soon as possible.

2.2. The OCCA_SC/TTR algorithm

For each transaction T_i, the concurrency controller maintains two sets: RS_i (Read Set), the set of objects read by T_i, and WS_i (Write Set), the set of objects written by T_i.

During the transaction execution, when the concurrency controller receives:

- a Read (T_i, g) operation, it adds the granule g (g is a tuple) to RS_i;
- a Read (T_i, g, pt) operation, it adds the granule g to RS_i only if pt indicates the current version of g. If pt indicates a previous version, this operation should not be taken into account in the validation test, as it cannot produce conflicts in any case;
- a Write (T_i, g) operation, it adds the granule g to WS_i;
- a Rollback operation, it eliminates all the read and written objects from RS_i and WS_i;
- a Commit operation, it checks if there is, or not, a conflict between the transaction to be validated and the ones which are not yet validated.

In our work, we started from the validation strategy of broadcast optimistic method with critical section. This strategy stipulates that, at any execution of a COMMIT order of a transaction T_i, each concurrent transaction T_c, which is still in its reading phase, must do a validation test with T_i. Then, the concurrency controller must examine $RS_c \cap WS_i$. It announces a conflict between the two transactions when this intersection is not empty. The transaction to be aborted is the one having the lowest priority.

To be able to ensure strong consistency, we propose to proceed **to the stamping of transactions by their arrival moments** and to attribute to the last coming one the lowest priority. However, this stamping is not sufficient to guarantee the DB consistency for TTR [4].

In order to solve this problem, we propose to define **a validation order** between transactions, which must be respected by the COMMIT execution of any transaction T_i. Consequently, the concurrency controller must check, before starting the validation test, that T_i has the highest priority regards to transactions which are not yet validated. Then, we propose to add **a certification phase** which precedes the validation one of each transaction. In this phase, the concurrency controller checks that T_i has the highest priority. Two cases are possible:

- If T_i has the highest priority, then the concurrency controller passes it to the validation phase.
- Otherwise, T_i is put in a waiting list to be certified later on.

Once arrived at its validation phase, T_i will be automatically validated. The new versions of granules manipulated by this transaction will be stamped by the transaction time of T_i (equal to its arrival moment) and then stored in the DB.

A research of conflicts, which can exist between T_i and any transaction T_j not yet validated, is then carried out. T_j is necessarily younger than T_i and thus having the lowest priority. Consequently, if there is a conflict, T_j must be aborted to be re-executed with the same stamp.

After the T_i validation, the concurrency controller must always check if there is a transaction T_k, waiting for certification, that has now the highest priority. We then propose to define an awakening function which has to select the transaction T_j having the highest priority in the set of transactions waiting for certification. This function must be called after the validation phase and allows T_j, if it exists, to sit again for the certification test. It is executed out of the critical section of the validation test, in order to increase the parallelism.

The critical section, during which all the manipulated granules in writing by T_i must be locked, normally covers the writing and validation phases of T_i. But we successfully reduced this period using the "EOT (end of transaction) marker" technique. When T_i passes the certification test, the concurrency controller marks its end in the RS_j of each transaction T_j not yet validated. To check the absence of conflicts between T_i and each transaction T_j, it uses the $RS_j(T_i)$ which is the set RS_j limited to the objects read from the beginning to the end of transaction mark of T_i. So, we propose to reduce the critical section to the writing phase and the period of marking the transaction end. Indeed, this period is considerably much shorter than the whole validation phase, also including the conflict checking and the awakening function.

3. Specification of OCCA_SC/TTR using PROMELA

We use, in the following, an example consisting of three transactions (T_1, T_2 and T_3) and two granules (x and y). The transaction T_1 manipulates these two granules in reading and writing. The transactions T_2 manipulates the granule x in reading and writing. T_3 manipulates the granule y in reading and writing. Thus, the transaction conflict risk is limited between T_1 and T_2, on the one hand, and T_1 and T_3, on the other hand. However, we tested our system using other examples with two or three transactions. The example selected for this paper includes conflict with one transaction (T_2 with T_1 or T_3 with T_1), with two transactions (T_1 with T_2 or T_1 with T_3) and also includes a case of non conflict (T_2 with T_3). Note that we can define, for other examples, more than three transactions. Indeed, since the writing phase and the period of the marking of the transaction end are executed within a critical section, the number of active transactions can not introduce inconsistency. Likewise, there is no risk of inconsistency if any other transaction arrives during the execution period of active transactions. To modify the number of transactions or granules, we must modify the two "nb_tr" and "size" constants, which respectively represent the transaction number and the array size related to the sets of the read and written objects by each transaction.

3.1. Declaration of constants, types and variables

We present, in this section, some constants defined in our PROMELA program.

- The granule manipulated by a TTR is the tuple, which includes temporal attributes, in addition to user attributes. Since SPIN is a model checking tool and not an implementation language, we will make abstraction on some details which have no effect on the CC of our system, in order to simplify it and to facilitate its checking. Thus, we define x and y constants as being two granules of a TTR.
  #define ***x*** *10*
  #define ***y*** *11*
- The "typedef" declaration allows declaring a user defined type of data. The following new *transaction* type gathers the transaction characteristics.

typedef **transaction {**
byte **nom**;
byte **order**; /* Represents the transaction stamp (arrival moment). */
byte **order_validation**; /*Indicates a transaction validation order. */
byte **state**; /* The transaction state takes one of the following values: "read" if the transaction is in reading phase; "certified" if the transaction is blocked in certification phase; "finished" if the transaction finished its execution. */
byte **rs**[size]; /* Represents the set of objects read by the transaction. */
byte **ws**[size]; /* Represents the set of objects written by the transaction. */
bit **restart**; /* takes the value "true" when the transaction is aborted. */
bit **certified**; /* A transaction is authorized to pass to its validation phase only if it is certified (certified = "true"). */
bit **awake**; /* Indicates that the transaction is awaked if it takes the value true. */
};

- The *liste_tr* array contains the whole transactions.
  transaction ***liste_tr****[nb_tr];*
- Knowing that each process of each transaction communicates with the concurrency controller, we then declared a message channel of the type "rendez-vous" (represented by a size equal to zero).
  chan ***trans_inst*** *= [0] of {byte,byte};*

Each process must inform the concurrency controller each time that it executes a reading, writing, aborting or validation operation. For that it uses the *trans_inst* channel. The concurrency controller also uses the same channel to inform the concerned process if the transaction must be aborted, blocked or if it is certified to be validated.

The choice of the type "*rendez-vous*" for the message channel is justified essentially by the fact that it allows avoiding the risk of the non detection of conflicts. Indeed, if we use the type "*buffer*", a given transaction can finish the execution of its statements without marking them in its RS nor WS. This is due to the fact that each message, which arrives at the channel, is recorded in the tail of the channel and its reception can be made only if previous messages leave the channel. During this time, if a transaction having the highest priority checks the absence of conflicts with that transaction, the risk of the non detection of conflict occurs.

The messages transmitted between any process and the concurrency controller are composed of two fields. The first indicates the transaction index in the *liste_tr* array. This element allows identifying the sender or the receiver of the message. It ensures that when the concurrency controller sends an answer, it will be received only by the concerned transaction. The second indicates the statement or the answer which is sent.

- The *N_valid* global variable, initialized to 1, is incremented after each transaction validation. When the value of this variable becomes higher than the number of transactions, we can finish the CC process.

    *byte **N_valid** = 1;*

3.2. Process definitions

Our system is composed of the *init* process, the *concurrent processes* and the *concurrency controller* process.

3.2.1. **The init process.** The *init* process is the starting point of the program. It starts its execution with the initialization of the transaction characteristics and assigns a priority order to each transaction in a non deterministic way. This allows checking the validity of the result whatever the order of the transactions. The *init* process launches, thereafter, three processes. Each of them deals with one transaction. It is useful to start a process sequence in a critical section marked by the keyword "atomic". Thus, a process begins its execution only after the initialization of all the others. This allows reducing the complexity of the checking.

```
Init  {
.
.
.
If
/* Without conditions, "if" allows executing one of the
   statement sequences in a non-deterministic way. */
::liste_tr[0].order = 1;
  liste_tr[1].order = 2;
  liste_tr[2].order = 3
::liste_tr[0].order = 1;
  liste_tr[1].order = 3;
  liste_tr[2].order = 2
.
.
.
fi;
atomic {  run cc( );  run p1( );  run p2( );  run p3( ) }
     }
```

3.2.2. **The concurrent processes.** Processes p1, p2 and p3 deal with the transactions T_1, T_2 and T_3, respectively.

Each process initializes the RS and WS of the concerned transaction before beginning the execution of the transaction operations. The element "*restart*", defined in the type "*transaction*", takes the value "false" initially. It could take the value "true" later if there is a conflict with another transaction having the highest priority. This leads to re-executing the transaction.

After executing all its read and write operations, a transaction executes its COMMIT statement. Its process informs the concurrency controller of this statement and waits for an answer. Three cases are possible:

- If the answer is "valid", then the transaction is certified and validated.
- If the answer is "abort", then the transaction must be aborted.
- If the answer is "block", it is because the transaction was not considered as the one having the highest priority, at the time of its certification phase, and it was blocked, as a transaction waiting for certification. It must wait until it is awaked by another transaction having the highest priority.

We present hereafter an example of the process p1.

```
proctype p1() {
 byte rec_response;
 begin:
 initialisation(0);
 liste_tr[0].restart = false;
/* beginning of operations. */
 if
 ::liste_tr[0].restart == true ->    goto begin
 ::liste_tr[0].restart == false ->   trans_inst!0,readx
       /* "!" specifies a message sending operation. */
 fi;
       .
       .
       .
 beginvalid:
 trans_inst!0,commit;
 trans_inst?0,rec_response;
       /* "?" specifies a message reception operation. */
 if
 ::rec_response == valid -> true
 ::rec_response == abort -> goto begin
 ::rec_response == block ->
  if
  ::liste_tr[0].awake == true -> liste_tr[0].certified=true;
    goto beginvalid
  fi
 fi;
  liste_tr[0].state=finished
        }
```

3.2.3. **The CC process.** The CC process is ready to receive messages as long as there is a transaction in execution, i.e. the number of validated transactions (held in the variable *N_valid*) is not higher than the number of transactions. Following the message reception, the CC process identifies the statement to be executed and calls the corresponding procedure in order to do the necessary treatments. All our system's procedures are developed using the *inline definition* SPIN.

```
proctype cc() {
 byte index,statement;
 do
 ::N_valid > nb_tr  ->  break
 ::else ->  trans_inst?index,statement;
  if
  ::statement == readx -> Inscription_read(index,x)
  ::statement == writex -> Inscription_write(index,x)
 .
 .
 .
  ::statement == commit ->  Treat_commit(index)
  fi
 od }
```

Each **Inscription_write**/**Inscription_read** inline definition adds a granule to WS/RS of the writer/reader transaction (arranged in the array element in the position "index").

Validation request, through the COMMIT statement, is treated by the concurrency controller using the **Treat_commit** procedure described below:

```
inline Treat_commit(index) {
 byte sent_response;
/* In the beginning, this procedure checks if the
     transaction has the highest priority. */
      .
      .
      .
 if
 ::liste_tr[index].certified ==   true ->
  validation(index);
  awaking();
  sent_response = valid
 ::else ->sent_response = block
 fi;
 trans_inst!index,sent_response
     }
```

The certification test is carried out in two steps. The concurrency controller checks if a transaction has the highest priority regards to the list of transactions waiting for certification. If it is the case, then the concurrency controller checks if this transaction has the highest priority regards to the list of reading transactions. Each transaction which passes these two tests is certified. Otherwise, the concurrency controller puts it in the list of transactions waiting for certification.

The **Validation** inline definition adds the EOT of the transaction T_i, which asks to be validated, in the RS of all concurrent transactions. Then it searches conflicts. Whenever it detects a conflict with a concurrent transaction T_j, this latter must be aborted and re-executed. So, this procedure assigns the same value "true" to the two elements "*restart*" and "*awake*" of T_j, before the end of the validation phase of T_i. T_j will be considered as a transaction in reading phase; the element "*state*" takes the value "read", also before this end.

4. OCCA_SC/TTR validation

The SPIN model checker proceeds in two steps. In the first one, "deadlock" or "unreachable code" errors are detected. In the second step, the validity of the system's quality properties is checked through the application of an appropriate LTL formula. For each detected error, SPIN gives the shortest way which leads to this error.

4.1. First OCCA_SC/TTR version

With the first version of our system, SPIN detects the possible blocking situations. The shortest way which leads to this error is described below.

Suppose that the priority order allotted to our three transactions is: $T_1 > T_2 > T_3$. Suppose also that T_3 is the first to ask for validation and that T_2 is the second one.

The transaction T_3 is certified regards to the list of transactions waiting for certification, which is still empty. But its certification test regards to the reading transactions fails. It is then blocked. For the same reason, T_2 is also blocked. T_1 continues its execution, it is certified and it starts its validation phase. Since there is a conflict between T_1 and T_2, on the one hand, and between T_1 and T_3, on the other hand, T_2 and T_3 are aborted. Then, the elements "*restart*" and "*awake*" of T_2 and T_3 take the same value "true", and their elements "*state*" take the value "read". After the validation of T_1, the concurrency controller executes the awakening procedure; but no transaction is awaked because the list of transactions blocked for certification is empty. The transaction T_1 is finished (*state*="finished"). If the transaction T_3 starts again its execution and demands its validation from the concurrency controller before T_2, it will be blocked again in the certification test, since T_2 has now the highest priority. So, when T_2 finishes again its execution, it passes the two certification tests and starts its validation phase. Since there is no conflict between T_2 and T_3, the latter is awaked by the concurrency controller after the T_2 validation. T_3 will sit for the certification test, but only regards to the transactions in reading phase, as it has the highest priority of the transactions blocked for certification (in fact, it is the only one). But the result of this test is negative, because T_2 is still considered in reading phase (*state*="read"). When the element "*state*" of T_2 takes the value "finished", the transaction T_3 will be blocked, despite having the highest priority in the system. Besides, it will persist in this state, since the concurrency controller cannot awake it any more.

To avoid such blocking situation, the attribution of the value "finished" to the element "*state*" should not be carried out after the awakening of the concurrent transaction which has now the highest priority.

4.2. Second version

In order to solve the problem of the previous blocking situation, we propose a new version of the OCCA_SC/TTR. This version will place the statement of attribution of the value "finished" to the element "*state*" before calling the awakening procedure. This statement must be placed in the **Treat_commit** inline definition and not in the end of each process.

```
inline Treat_commit(index)  {
      .
      .
      .
 if
  ::liste_tr[index].certified == true ->
  validation(index);
  liste_tr[0].state = finished;
  awaking();
  sent_response = valid
  ::else->
      .
      .
      .
  }
```

No error was reported by SPIN for this new version. The checking of the model was done using the exhaustive search mode and the partial order reduction algorithm.

4.3. Definition and application of LTL formulae

We use here the two elements "*order*" and "*order_validation*" defined in the "*transaction*" type. "*order*" takes the transaction stamp value which is assigned in accordance with transaction arrival order. This assignment is carried out before starting the parallel execution of all transactions. However, "*order_validation*" is used to represent the validation order of the transaction. When a transaction is validated, the concurrency controller assigns to this element the value representing the validation order. To achieve this aim, we use the global variable "*N_valid*", initialized with 1. After validating each transaction T_i, the procedure "Validation" attributes the value of "*N_valid*" to the element "*order_validation*" of T_i, and increments the global variable "*N_valid*".

```
inline Validation(index) {
  /* Beginning of critical section. */
 .
 .
 .
  liste_tr[index].order_validation = N_valid;
 N_valid = N_valid+1
  /* End of critical section. */
 .
 .
 .
  }
```

To make sure that our system guarantees strong consistency, we must verify, at the end of the execution of any transaction T (placed in the position *i* in the liste_tr array), the following property p:

(liste_tr[i].order == liste_tr[i].order_validation).

"p" means that the two elements "*order*" and "*order_validation*" of T have the same value.

To reduce the state number of our system, we propose to check this property only for one transaction. Indeed, it is not necessary to define a formula for each transaction since SPIN can generate and verify all the scenarios of the transaction priority order. So, we defined the property "p" as follows:

```
#define p
 (liste_tr[0].order == liste_tr[0].order_validation)
```

The LTL Formula which we applied is as follows: "<>[]p".

We used in this formula two temporal logic operators; the *guarantee operator* "eventually: <>" and the *safety operator* "always: []". "<>[]p" means that there is at least a state from which we will have the property "p" true forever.

In our system, the priority and validation orders are initially different. This is true since the assignment of priority order is carried out at the beginning of the execution, but the assignment of validation order is carried out at the end of the execution. This justifies the use of the operator eventually.

No error is detected in this checking phase when applying the formula <>[]p.

After having checked that strong consistency is ensured, we will check, hereafter, that in the case of a conflict, the transaction with the lowest priority will be aborted. Our second formula is then based on the values which can be taken by the granules x and y. So, we define two global variables "*xval*" and "*yval*":

```
      byte xval;
      byte yval;
```

These two variables memorize the values taken by x and y, respectively. We suppose that each transaction, when modifying a variable, gives it a specific value: when T_1 modifies x, this granule will take the value "tr1". At the end of the execution, each granule's final value must be equal to the value assigned by the transaction having the lowest priority. If it is not the case, it means that there is a non solved conflict between two transactions on this granule (the transaction having the lowest priority was not aborted). So, we modify the process body as follows:

```
proctype p1()  {
 .
 .
 .
 if
 ::liste_tr[0].restart == true -> goto begin
 ::liste_tr[0].restart == false -> trans_inst!0,writex;
       xval = tr1
 fi;
 .
 .
 .
     }
```

Let us remember that the two transactions T_1 and T_2 manipulate the granule x in reading and writing. If these two transactions are executed simultaneously, a conflict may occur. The LTL formula, described below, allows checking if this conflict is solved or not (if it appears). The LTL formula is as follows:

[]((<>(a&&b)-><>c) && (<>(!a&&d)-><>e))

The properties a, b, c, d and e are defined as follows:

#define a (liste_tr[0].order < liste_tr[1].order)
#define b (liste_tr[0].state == finished)
#define c (xval == tr2)
#define d (liste_tr[1].state == finished)
#define e (xval == tr1)

This LTL formula treats the two possible cases between T_1 and T_2 according to their priority orders. T_1 and T_2 correspond respectively to the index 0 and 1 in the *liste_tr* array.

Case 1:
if $T_1 > T_2$ (a = true) and if T_1 is finished (b = true) $\rightarrow$
we must be sure to have:
c = true in a future state (xval = "tr2").

In the "<> (a&&b)-> <>c" formula, we used the eventually operator "<>" for the condition (a&&b). Indeed, "a&&b" is not true in the initial state. The element "*state*" of a transaction can have the value "finished" only after the end of the transaction execution. This formula implies that if the condition "a&&b" is true in a state i different from the initial state, we must be sure to verify the property "c" (c = true) in a state j where j > i. The use of the guarantee operator for the property "c" ("<> c") is justified by the fact that "j" is not necessarily equal to "i+1".

Case 2:
if $T_1 < T_2$ (a != true) and if T_2 is finished (d = true) $\rightarrow$
we must be sure to have:
e = true in a future state (xval = "tr1").

The application of this formula gives a valid result.

5. Positioning vis-à-vis of related work

Although a lot of work has been devoted to TDB, little work has paid attention to the CC problems.

To our knowledge, OCCA_SC/TTR [4] is the first optimistic CC algorithm proposed for TTR. The 2PL algorithm is the most adopted one. But we see that applying an optimistic CC is advantageous in the case of TTR since old versions of data are maintained. This allows data readers to see the state of the data before the modifications take place. So, the abortion degree can be reduced.

The OCCA_SC/TTR algorithm is based on the broadcast strategy and then detects conflicts as soon as possible. By using EOT marker technique, it has the merit to reduce to the maximum the period during which resources are locked in the validation phase. The most optimistic CC algorithms of the literature adopt the broadcast strategy of validation, like the one proposed in [12]. But, those using the EOT marker technique are rare.

Compared to the CC algorithms proposed for TTR [5] [8] [9] [15], OCCA_SC/TTR is characterized by a high degree of parallelism. Indeed, it does not lock any resource at its use, since it is not pessimistic. All of these algorithms define a granule as a tuple in order to reduce the conflict objects without complicating the data management. OCCA_SC/TTR ensures the strong consistency, like the algorithms proposed in [8] [9] [15] by stamping the transactions by the moment of their arrival and synchronizing them according to this order. The advantage of our algorithm consists in the fact that it does not need resource knowledge, contrary to the algorithms of [8] and [9].

Both OCCA_SC/TTR and the algorithm of [9] define a certification phase. Unlike OCCA_SC/TTR, the certification phase of [9] is used to extend the 2PL algorithm with a scheduling mechanism of maturities.

Moreover, both OCCA_SC/TTR and the speculative algorithms, proposed for real time DB [1], aim to increase the parallelism degree. Indeed, OCCA_SC/TTR successfully avoids the disadvantages of optimistic CC algorithms. It detects conflicts as soon as possible and reduces to the maximum the period during which resources are locked in the validation phase. However, speculative algorithms use both pessimistic and optimistic approaches. Any conflict transaction continues its execution. But a conflict point is maintained using a shadow transaction initially blocked. The conflict transaction must be aborted only if it is not the first arrived to its validation phase. In this case, the shadow transaction starts its execution from the conflict point.

In other respects, the formal verification of the CC algorithms, proposed for TTR [5] [9], is achieved with a theoretical formalism. Whereas, the formal verification of OCCA_SC/TTR algorithm is achieved automatically using the SPIN model checker. Indeed, model checking [7] [14] is a powerful and useful technique, allowing an automated verification of finite state systems. It is an ideal tool to verify consistency, i.e. deadlock detection and unreachable states. These kinds of errors need to be detected in an automatic exhaustive search.

6. Conclusion

In this paper, we have shown how OCCA_SC/TTR operates correctly. This algorithm is an optimistic CC algorithm which ensures the SC for TTR. Based on the broadcast validation strategy with the integration of the EOT marker technique, it allows avoiding the problem of the aborting transaction later, on the one hand, and reducing the period of time during which resources are locked at the transaction validation, on the other hand.

The formal verification, under the PROMELA language and the SPIN tool, allows us to find some insufficiencies and to solve an error problem relating to the moment when a transaction must have the finished state. We have shown that the state of a transaction T_i must have the value "finished" before making another transaction T_j awake.

In addition, the definition and the application of the two LTL formulae, using SPIN, enable us to check that the strong consistency of the DB is maintained, on the one hand, and that, in case of a conflict between two transactions, this conflict is solved by aborting transaction having the lowest priority, on the other hand.

Our future work aims at an experimental validation of our algorithm using a complete case study, and at showing that it ensures better performances compared to those of pessimistic algorithms.

7. References

[1] L. Amanton, B. Sadeg, and J. Haubert, "Precision for Timeliness in Distributed Real-Time Databases", *Proceedings of the ICEIS Conference*, Angers, France, Vol. 1, 2003, pp. 558-561.

[2] J. Berstel, S. C. Reghizzi, G. Roussel, and P, S. Pietro, "A Scalable Formal Method for Design and Automatic Checking of User Interfaces", *ACM TOSEM*, Vol. 14(2), 2005, pp. 124-167.

[3] E. Brinksma, A. Mader, and A. Fehnker, "Verification and Optimization of a PLC Control Schedule", *STTT Journal*, Vol. 4 (1), 2002, pp. 21-33.

[4] R. Bouaziz, and A. Makni, "ACCO_CF/RTT: Un algorithme de contrôle de concurrence optimiste pour les relations temporelles de transaction", *ISDM Journal*, n°19, 2005, article n°236.

[5] C. Castro, "On concurrency management in temporal relational databases", *Proceedings of the SEBD Conference*, 1998, pp. 189-202.

[6] T. Cattel, "Process Control Design using SPIN", *The First SPIN workshop*, Montréal, Quebec, 1995.

[7] E.M. Clarke, A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications", *ACM TOPLAS*, Vol. 8(2), 1986, pp. 244-263.

[8] S. D. Elloumi, R. Bouaziz, and M. Moalla, "Contrôle de concurrence multiversion dans les bases de données temporelles", *Proceedings of the BDA Conference*, Hammamet, Tunisie, 1998, pp. 135-155.

[9] M. Finger, and P. McBrien, "Concurrency Control for Perceivedly Instantaneous Transactions in Valid-Time Databases", *International Workshop*, 1997, pp. 112-119.

[10] G. Gardarin, *Base de données : Les systèmes et leurs langages*, EYROLLES Edition, 1988.

[11] S. Gnesi, G. Lenzini, D. Latella, C. Abbaneo, A. Amendola, and P. Marmo, "An Automatic SPIN Validation of a Safety Critical Railway Control System", *Proceedings of the DSN Conference*, pp. 119-124. Published by IEEE Computer Society Press, 2000.

[12] J. Harista, M. Carey, and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems", *Real-Time Systems Journal*, Vol. 4, n°3, 1992, pp. 203-241.

[13] K. Havelund, M. Lowry, and J. Penix, "Formal Analysis of a Space Craft Controller using SPIN", *IEEE TSE*, Vol. 27(9), 2001, pp. 749-765.

[14] G, J. Holzmann, "The model cheker spin", *IEEE TSE*, Vol. 23(5), 1997, pp. 279-295.

[15] C. S. Jensen and D. B. Lomet, "Transaction timestamping in (temporal) databases", *Proceedings of the VLDB Conference*, Roma, Italy, 2001, pp. 441-450.

[16] M. Rahgozar, *Contrôle de concurrence par gestion des événements*, PhD thesis, 1987.