ISBN 978-83-60810-14-9
ISSN 1896-7094

Modeling Real-Time Database Concurrency Control Protocol Two-Phase-Locking in Uppaal

Martin Kot
Center for Applied Cybernetics
Dept. of Computer Science,
Technical University of Ostrava
17. listopadu 15,708 33 Ostrava – Poruba
Czech Republic
Email: martin.kot@vsb.cz

***Abstract*—Real-time database management systems (RT-DBMS) are recently subject of an intensive research. Model checking algorithms and verification tools are of great concern as well. In this paper we show some possibilities of using a verification tool Uppaal on some variants of pessimistic concurrency control protocols used in real-time database management systems. We present some possible models of such protocols expressed as nets of timed automata, which are a modeling language of Uppaal.**

I. Introduction

MANY real-time applications need to store some data in a database. It is possible to use traditional database management systems (DBMS). But they are not able to guarantee any bounds on a response time. This is the reason why so-called real-time database management systems (RTDBMS) emerged.

Research in RTDBMS focused on evolution of transaction processing algorithms, priority assignment strategies and concurrency control techniques. But the research was based especially on simulation studies. Hence at Technical university of Ostrava, Václav Król, Jindřich Černohorský and Jan Pokorný designed and implemented an experimental real-time database system called V4DB [6], which is suitable for study of real time transaction processing. The system is still in further development but some important results were obtained already.

Formal verification is of great interest recently and finds its way quickly from theoretical papers into a real live. It can prove that a system (or more exactly a model of a system) has a desired behavior. The difference between testing and formal verification is that during testing only some possible computations are chosen. Formal verification can prove correctness of all possible computations. A drawback of formal verification is that for models with high descriptive power are almost all problems undecidable. It is important to find a model with an appropriate descriptive power to capture a behavior of a system, yet with algorithmically decidable verification problems.

In this paper we consider so called model checking (see e.g. [3], [8]). This form of verification uses a model of a system in some formalism and a property expressed usually in the form of formula in some temporal logic. Model checking algorithm checks whether the property holds for the model of a system. There are quite many automated verification tools which implement model checking algorithms. Those tools use different modeling languages or formalisms and different logics.

The idea of the research described in this paper came from authors of V4DB. They were interested in using a verification tool on their system. They would like to verify and compare different variants of algorithms and protocols used in RTDBMS. To our best knowledge, there are only rare attempts of automated formal verification of real-time database system. In fact we know about one paper ([9]) only where authors suggested a new pessimistic protocol and verified it using Uppaal. They presented two small models covering only their protocol.

There is not any verification tool intended directly for real-time database systems. We have chosen the tool Uppaal because it is designed for real-time systems. But, it is supposed to be used on so-called reactive systems, which are quite different from database systems. So we need to solve the problem of modeling data records of the database and some other problems. Then we would like to check some important properties of used protocols and algorithms, for example: absence of a deadlock when using an algorithm which should avoid deadlock in the transaction processing, processing transaction with bigger priority instead of transactions with smaller priority and so on.

Big problem of verification tools is so called state space explosion. Uppaal is not able to manage too detailed models. On the other hand, too simple models can not catch important properties of a real system. So we need to find a suitable level of abstraction.

One of the most important and crucial parts of RTDBMS is concurrency control. There were many different concurrency control protocols suggested. In this paper, we will concentrate on variants of a pessimistic protocol called two-phase-locking (2PL). First variant is basic 2PL protocol, then slightly modified version where deadlines are used to abort waiting transactions and finally 2PL – high priority where a

Author acknowledges the support by the Czech Ministry of Education, Grant No. 1M0567.

transaction with higher priority can restart a transaction with a smaller priority. We will show that it is possible to model those protocols, to some level of abstraction, using modeling language of Uppaal. These examples will show possibilities of modeling other similar pessimistic protocols and even some other parts of RTDBMS. The models are inspired by the RTDBMS system V4DB in some way. V4DB is experimental so it has some simplifications which we can use to obtain simpler models yet with important behavior covered (as V4DB has). But it is possible to use ideas shown in this paper for verification of concurrency control algorithms in general.

We will also mention a few simple formula with an Uppaal's answer to show model checking possibilities on suggested models.

II. Verification tool Uppaal

Uppaal ([2], [4]) is a verification tool for real-time systems. It is jointly developed by Uppsala University and Aalborg University. It is designed to verify systems that can be modeled as networks of timed automata extended with some further features such as integer variables, structured data types, user defined functions, channel synchronization and so on.

A timed automaton is a finite-state automaton extended with clock variables. A dense-time model, where clock variables have real number values and all clocks progress synchronously, is used. In Uppaal, several such automata working in parallel form a network of timed automata.

An automaton has locations and edges. Each location has an optional name and invariant. An invariant is a conjunction of side-effect free expressions of the form $x < e$ or $x \leq e$ where x is a clock variable and e evaluates to an integer. Each automaton has exactly one initial location.

Particular automata in the network synchronize using channels and values can be passed between them using shared variables. A state of the system is defined by the locations of all automata and the values of clocks and discrete variables. The state can be changed in two ways - passing of time (increasing values of all clocks) and firing an edge of some automaton (possibly synchronizing with another automaton or other automata).

Some locations may be marked as committed. If at least one automaton is in a committed location, time passing is not possible, and the next change of the state must involve an outgoing edge of at least one of the committed locations.

Each edge may have a select, a guard, a synchronization and an assignment. Select gives a possibility to choose non-deterministically a value from some range. Guard is a side-effect free expression that evaluates to a boolean. The guard must be satisfied when the edge is fired. Synchronization label is of the form $Expr!$ or $Expr?$ where $Expr$ evaluates to a channel. An edge with $c!$ synchronizes with another edge (of another automaton in the network) with label $c?$. Both edges have to satisfy all firing conditions before synchronization. There are urgent channels as well – synchronisation through such a channel have to be done in the same time instant when it is enabled (it means, time passing is not allowed

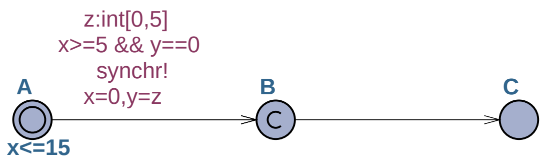

Fig. 1. Graphical representation of a timed automaton in Uppaal

if a synchronisation through urgent channel is enabled). An assignment is a comma separated list of expressions with a side-effect. It is used to reset clocks and set values of variables.

Figure 1 shows how the described notions are represented graphically in Uppaal. There are 3 locations named `A`, `B` and `C`. Location `A` is initial and `B` is committed. Moreover `A` has an invariant `x<=15` with the meaning that the automaton could be in this location only when the value of the clock variable `x` is less or equal 15. The edge between `A` and `B` has the select `z:int[0,5]` – it nondeterministically chooses an integer value from the range 0 to 5 and stores it in variable `z`. This edge also has the guard `x>=5 && y==0`. This means that it can be fired only when the value of the clock variable `x` is greater or equal 5 and the integer variable `y` has the value 0. Data types of variables are defined in a declaration section. Further it has synchronization label `synchr!` and an assignment `x=0, y=z` reseting the clock variable `x` and setting the value of `z` to the integer variable `y`.

Uppaal has some other useful features. Templates are automata with parameters. These parameters are substituted with given arguments in the process declaration. This enables easy construction of several alike automata. Moreover, we can use bounded integer variables (with defined minimal and maximal value), arrays and user defined functions. These are defined in declaration sections. There is one global declaration section where channels, constants, user data types etc. are specified. Each automaton template has own declaration section, where local clocks, variables and functions are specified. And finally, there is a system declaration section, where global variables are declared and automata are created using templates.

Uppaal's query language for requirement specification is based on CTL (Computational Tree Logic, [5]). It consist of path formulae and state formulae. State formulae describe individual states and path formulae quantify over paths or traces of the model.

A state formula is an expression that can be evaluated for a state without looking at the behavior of the model. For example it could be a simple comparison of a variable with a constant `x <= 5`. The syntax of state formulae is similar to the syntax of guards. The only difference is that in a state formula disjunction may be used.

There is a special state formula `deadlock`. It is satisfied in all deadlock states. The state is deadlock if there is not any action transition from the state neither from any of its delay successors.

Path formulae can be classified into *reachability*, *safety* and *liveness*. Reachability formulae ask if a given state formula is satisfied by some reachable state. In Uppaal we use syntax `E<>` φ where φ is a state formula.

Safety properties are usually of the form: "something bad will never happen". In Uppaal they are defined positively: "something good is always true". We use `A[]` φ to express, that a state formula φ should be true in all reachable states, and `E[]` φ to say, that there should exist a maximal path such that φ is always true.

There are two types of liveness properties. Simpler is of the form: "something will eventually happen". We use `A<>` φ meaning that a state formula φ is eventually satisfied. The other form is: "leads to a response". The syntax is φ `-->` ψ with the meaning that whenever φ is satisfied, then eventually ψ will be satisfied.

The simulation and formal verification are possible in Uppaal. The simulation can be random or user assisted. It is more suitable for verification whether the model corresponds with the real system. Formal verification should confirm that the system has desired properties expressed using the query language. There are many options and settings for verification algorithm in Uppaal. For example we can change representation of reachable states in memory. Some of the options lead to less memory consumption, some of them speed up the verification. But improvement in one of these two characteristic leads to a degradation of the other usually.

For more exact definitions of modeling and query languages and verification possibilities of Uppaal see [2].

III. Pessimistic protocol two-phase-locking

In this section we suggest one model of pessimistic concurrency control protocol. Of course, it is not the only one possible.

Two-phase-locking protocol is based on data locks. Before access to data the transaction must have a lock. All locks granted to a transaction are released after all operations of this transaction are executed. There are two types of locks—for read and write. The first is used for the operation select and the latter for update, delete and insert. Either one write lock or several read locks can be on a particular record (for simplicity, in our model will be only one read lock allowed for one record). If a transaction can not get a lock for a request it is placed in a queue of this record. After an existing lock is released, a new lock is granted to the first transaction in the queue.

The suggested model consists of several timed automata created using two templates. One type of automata represents data records in a database. The template is shown on the Figure 2. Each record automaton has an integer ID stored in `rec_id`. There are three locations corresponding to two types of locks and to an unlocked state. Channels `rd_ch[x]` and `wrt_ch[x]` are used for requests for read and write locks on record x. Channel `rls_ch[x]` is for release (unlock) request.

The second template shown on the Figure 3 is intended to create automata representing active transactions in the system. In V4DB is a number of active transactions bounded (predispatcher module of RTDBMS holds the queue of incoming transactions and passes them to a dispatcher in such a way that it avoids overloading). So it is possible to represent one active (i.e. currently in execution) transaction as one automaton. After successful end of a transaction the same automaton represents some other transaction.

For simplicity, all transactions are supposed to have the same number of operations (given by a constant `OPERATIONS`). Each operation accesses one record (i.e. needs one lock). A type of operations and an accessed record is for a real RTDBMS in fact random because it is determined outside the RTDBMS. We do not need to model concrete operations, only locks. The record is chosen nondeterministically using select `rec:rec_id_t`. The operation is then immediately (due to a committed location) chosen nondeterministically by using one of three possible edges. If a transaction owns the demanded type of a lock on the accessed record, it does not asks the lock again. If it has only a read lock, it can ask change to a write lock. In the array `locked` is stored the information about owned locks, variable `locks` contains the number of operations for which locks are gained and `real_locks` the number of records locked by this transaction.

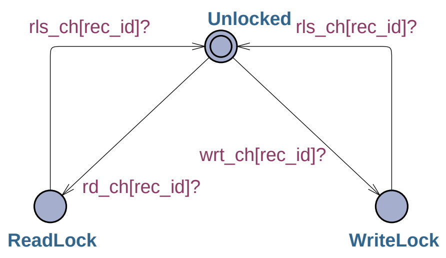

Fig. 2. Automaton representing a record in a database

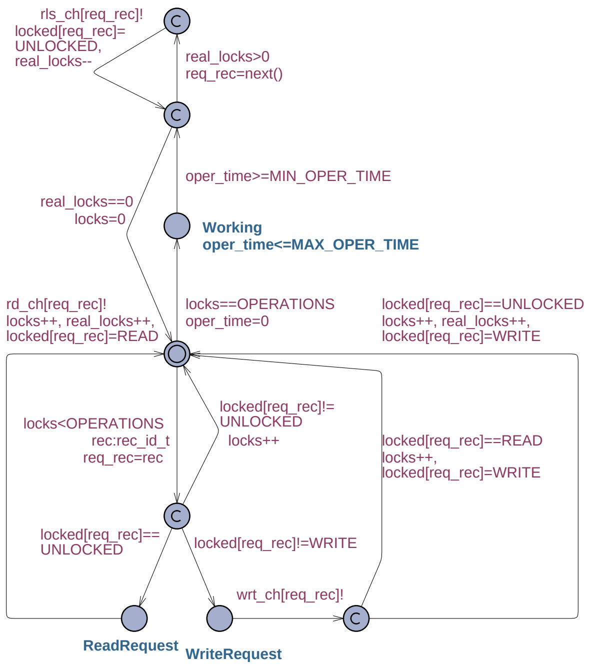

Fig. 3. Transaction automaton for two-phase-locking protocol

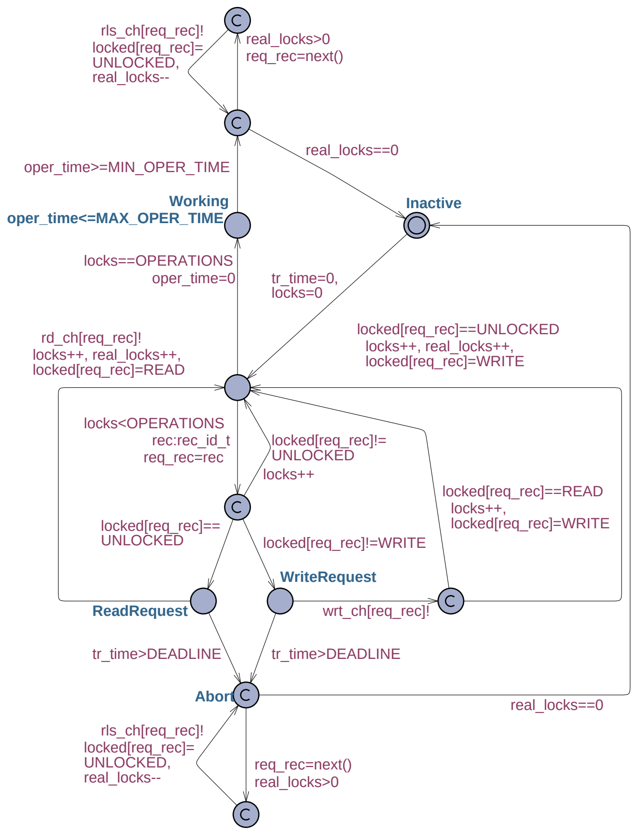

Fig. 4. Transaction automaton for modification of two-phase-locking protocol

If the transaction has all necessary locks, the automaton is in a location `Working`. This represents execution of database operations. The time spent in this location is bounded by constants `MIN_OPER_TIME` and `MAX_OPER_TIME`. After the execution, all locks are released instantly (using committed states and edges between them).

The described model simulates basic variant of 2PL protocol where a deadlock can arise when some transactions wait mutually for locks granted to other transactions. A small modification where transactions exceeding their deadline may be aborted can solve the problem with deadlocks.

IV. Modification of a model of two-phase-locking protocol

A template for database record automata remains the same as in the previous model (Figure 2). A changed transaction automata template is shown on Figure 4.

There is a clock variable `tr_time` added. It measures time from the beginning of transaction execution. If a transaction is waiting for a lock and it reaches its deadline (for simplicity same for all transactions given by a constant `DEADLINE`), it can be aborted. This means that all locks previously granted to this transaction are released.

We can use Uppaal to verify that this solution is really sufficient to avoid deadlock. For Uppaal, reachability properties are more suitable. So the formula

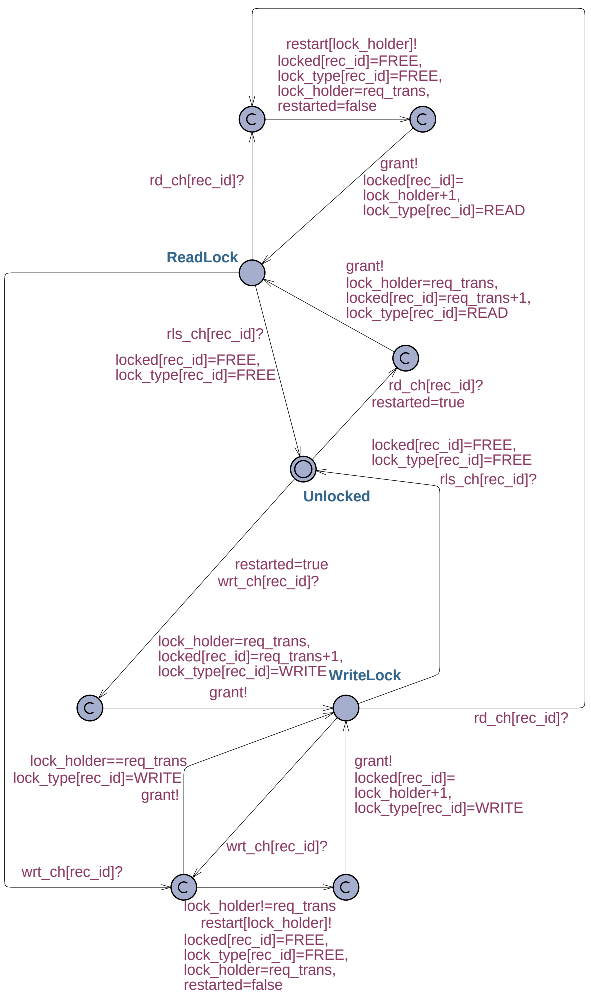

Fig. 5. Automaton representing a record in a database for 2PL-HP protocol

```
E<> deadlock
```

means that deadlock is reachable in the model and this property is not satisfied. Hence it is verified that the system is deadlock-free.

V. Pessimistic protocol two-phase-locking high-priority

The last modification of our model is for a protocol two-phase-locking high-priority (2PL–HP). If a lock is requested by a transaction with a higher priority the transaction with a lower priority holding this lock may be restarted.

For this model we change both automata templates. A new template for database record automata is depicted on Figure 5.

In the global declarations are defined two arrays – `locked` and `lock_type`. The first one contains information about transactions holding locks for particular records and the latter one contains information about types of particular locks. `lock_holder` is a local variable of one record automaton used for the ID of transaction holding the lock on this record. As almost all is chosen nondeterministically (including the order of activating particular transaction automata), we can model priorities using ID numbers of transaction automata – higher ID means higher priority.

If the automaton is in the location `Unlocked`, all requests passed through channels `rd_ch` and `wrt_ch` are answered immediately through a channel `granted` and informations about this lock are saved to above mentioned arrays and variable.

If the automaton is in the location `WriteLock` or `ReadLock` and a new request arrives, it has to restart a transaction holding the lock (priorities are checked before the request in a transaction automaton). Restarted transaction x is contacted using a channel `restart[x]`. Then the lock is granted to a requesting transaction using channel `grant`. If a write lock is requested from the location `ReadLock`, there is a possibility to grant it without any other activity (except for the change of a type of lock in `lock_type` array). This is done when requesting transaction (`req_trans`) is the same as the current holder of the read lock (`lock_holder`).

The transaction automata template has to be changed as well. The modified version is depicted on the Figure 6.

There are added edges leading to a new location `Restart` from all locations where an automaton can be during passing of time. All those edges have synchronization label `restart[trans_id]`. In this way a transaction (with an ID stored in a variable `trans_id`) can be restarted anytime by a record automaton. In the location `Restart` all previously gained locks are released and the waiting transaction with higher priority is notified using global boolean variable `restarted`. A function `next()` returns the smallest ID number of a record on which is the transaction actually holding a lock.

Requests for locks are guarded. A requested record (specified by the variable `req_rec`) has to be unlocked or locked by a transaction with smaller priority. It comes handy to use 0 (constant `FREE` is defined as 0) in the array `locked[]` for unlocked records and ID of transaction automaton (i.e. the priority of transaction) plus one for a lock holder. Than the guard

```
trans_id+1 > locked[req_rec]
```

is true whenever the lock on `req_rec` is hold by a transaction with a smaller priority or this record is unlocked.d by a transaction with a smaller priority or this record is unlocked.

As in the previous case, although for this model Uppaal can verify that it is deadlock-free. We can use the same formula

```
E<> deadlock
```

and the answer is negative (i.e. no deadlock is reachable).

Furthermore we can check e.g. if the transaction with the highest priority could be possibly restarted. The number of transaction automata is given using a constant `TRANSACTIONS`. Hence the greatest ID number (this means priority too) is `TRANSACTIONS-1`. The formula is

```
E<> Transaction(TRANSACTIONS-1).Restart
```

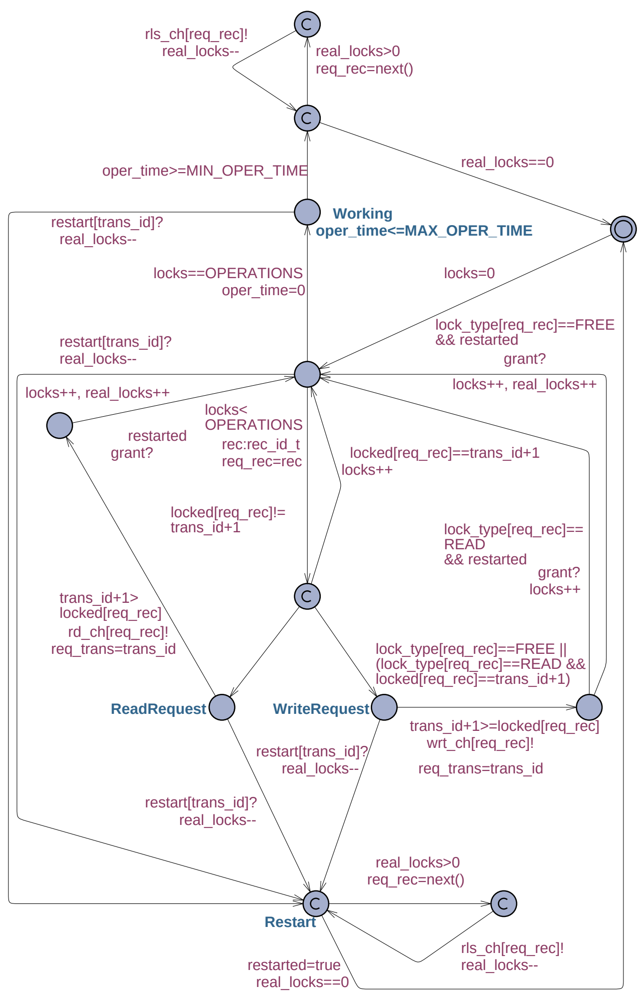

Fig. 6. Transaction automaton for 2PL-HP protocol

and it is not satisfied, i.e. this transaction could not be restarted. For all other transactions x the formula

```
E<> Transaction(x).Restart
```

is satisfied.

VI. Conclusion

In the previous sections, several timed automata were shown. They form models of three variants of pessimistic concurrency control protocols used in real-time database management systems. Of course, this were not the only possible models. The purpose was to show that some important aspects of the real-time database system such as a concurrency control can be modeled using such a relatively simple model as nets of timed automata are. The models can be extended in many different ways to capture more behavior of those protocols and thus allow many properties to be described as a formula in the logic of Uppaal and then checked using its verification algorithms. Even on presented models (without any extensions or modifications) different properties have been checked and some simple samples of them were presented in this paper.

Some properties can not be expressed using Uppaal's modification of CTL. The possible solution to this problem is to try some other verification tool with other query language.

Other parts of real-time database system or other concurrency control protocols can be modeled too. For example priority assignment algorithms have significant influence on performance database management system. This is our potential future work.

REFERENCES

[1] Alur, R., Dill, D.L.: Automata for modeling real-time systems. Proc. of Int. Colloquium on Algorithms, Languages, and Programming, volume 443 of LNCS, pages 322-335, 1990.

[2] Behrmann, G., David, A., Larsen, K. G.: A Tutorial on Uppaal. Available on-line at http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf (September 7, 2007)

[3] Berard, B., Bidoit, M., Petit, A., Laroussinie, F., Petrucci, L., Schnoebelen, P.: Systems and Software Verification, Model-Checking Techniques and Tools. ISBN 978-3540415237, Springer, 2001.

[4] David, A., Amnell, T.: Uppaal2k: Small Tutorial. Available on-line at http://www.it.uu.se/research/group/darts/uppaal/tutorial.ps (September 7, 2007)

[5] Henzinger, T.A.: Symbolic model checking for real-time systems. Information and computation, 111:193-244, 1994.

[6] Król, V.: Metody ověřování vlastností real-time databázového systému s použitím jeho experimentálního modelu. Dissertation thesis. VSB—Technical university of Ostrava, 2006 (in Czech).

[7] Król, V., Pokorný, J., Černohorský, J.: The V4DB project—support platform for testing the algorithms used in real-time databases. WSEAS Transactions on Information Science & Applications, Issue 10, Volume 3, October 2006.

[8] McMillan, K. L.: Symbolic Model Checking. ISBN 978-0792393801, Springer, 1993.

[9] Nyström, D., Nolin, M., Tesanovic, A., Norström, Ch., Hansson, J.: Pessimistic Concurrency-Control and Versioning to Support Database Pointers in Real-Time Databases. Proc. of the 16^{th} Euromicro Conference on Real-Time Systems, pages 261-270, IEEE Computer Society, 2004.