Preprints of the 30th IFAC Workshop on Real-Time Programming and
4th International Workshop on Real-Time Software pp. 155–160

On the Implementation of State-space Exploration Procedure in a Relational Database Management System*

Jaroslav Rab   Ondrej Rysavy   Miroslav Sveda

Brno University of Technology, Bozetechova 2
612 66 Brno, Czech Republic

Abstract: An examination of discrete system's behavior can be done by exhaustive exploration of the state space that is generated according to the assigned domain semantics. Model-checking is the matured discipline that allows to explore state space as large as several millions of states. In this paper, we describe a novel approach to the implementation of state exploration procedure using PL/SQL, the language of Oracle relational database system. The high efficiency of database systems when dealing with large amounts of data and relatively cheap hardware available nowadays advocates the use of relational database as an implementation platform for practical exhaustive state exploration algorithm with the hope that this platform may scale up the model checking method to hundreds of millions of explorable states.

Keywords: Formal Specification, Temporal Logic of Actions, State exploration, Relational database systems

1. INTRODUCTION

Practical verification of hardware and software systems is based on algorithmic methods, which are able to explore large state-spaces that exhaustively describes the behavior of these systems. While algorithms for state space explorations are rather simple and well explored the issue of handling very large state spaces is actively researching. The methods for efficient representation of data in main memory and various abstraction techniques allows explore the systems consisting of hundreds of millions of states. Often the very sophisticated and complex methods are used to deal with storing and indexing the data describing states. In this paper, we present an idea to use relational database system to manipulate the data describing state space and to provide a system for the exploration of these data in order to verify the required properties of a hardware or software system being modeled. Although, the layer manipulating with data is much heavier than that usually implemented in state of the art model checking tools, we believe that the following statements provide enough sound arguments to justify the rationality this idea.

Virtually unlimited memory. Database systems are designed to accommodate a large amount of data. The active databases can have hundreds of millions of rows in tables, and their total size can be hundreds of gigabytes.

Time to result. Various techniques have been implemented to speed up data processing in the database systems. For example, indexes help to optimize various operations containing selecting the data or merging two tables. Often the speed optimization requires to use more space. As first assumption claims that we can have a lot of space for the database, the speed of processing may be increased. We assume that time required to get at least partial results is more important to users than memory requirement considerations.

Presistence. The database systems are primarily used for storing the data. The state space generated for a specification which is stored in the database is ready for further exploration until someone explicitly decides it should be erased from the database. The persistence balance the overall costs of model generation that may be very high for a large models. Moreover as database data may be altered as needed, the techniques that modify or refine the model as specification evolves can be applied.

We demonstrate the idea using a system description given in the formalism called Temporal Logic of Actions developed by Lamport (2003). In the rest of the section, a brief description of this formalism is provided. Note that an adequate system description as assumed in this paper can be provided by any state-based formalism employing some form of guard/action predicates.

1.1 TLA+

Temporal Logic of Actions (TLA) is a variant of linear-time temporal logic. It was developed by Lamport (2003) primarily for specifying distributed algorithms, but several works shown that the area of application is much broader. The system of TLA+ extends TLA with data structures

* The research has been supported by the Czech Ministry of Education in the frame of Research Intention MSM 0021630528: Security-Oriented Research in Information Technology and by the Grant Agency of the Czech Republic through the grants GACR 102/08/1429: Safety and Security of Networked Embedded System Applications and GACR 201/07/P544: Framework for the deductive analysis of embedded software.

allowing for easier description of complex specification patterns. TLA+ specifications are organized into modules. Modules can contain declarations, definitions, and assertions by means of logical formulas. The declarations consist of constants and variables. Constants can be uninterpreted until an automated verification procedure is used to verify the properties of the specification. Variables keep the state of the system, they can change in the system and the specification is expressed in terms of transition formulas that assert the values of the variables as observed in different states of the system that are related by the system transitions. The overall specification is given by the temporal formula defined as a conjunction of the form $I \wedge \Box[N]_v \wedge L$, where I is the initial condition, N is the next-state relation (composed from transition formulas), and L is a conjunction of fairness properties, each concerning a disjunct of the next-state relation. Transition formulas, also called actions, are ordinary formulas of untyped first-order logic defined on a denumerable set of variables, partitioned into sets of flexible and rigid variables. Moreover, a set of primed flexible variables, in the form of v', is defined. Transition formulas then can contain all these kinds of variables to express a relation between two consecutive states. The generation of a transition system for the purpose of model checking verification or for the simulation is governed by the enabled transition formulas. The formula $\Box[N]_v$ admits system transitions that leave a set of variables v unchanged. This is known as stuttering, which is a key concept of TLA that enables the refinement and compositional specifications. The initial condition and next-state relation specify the possible behavior of the system. Fairness conditions strengthen the specification by asserting that given actions must occur. The TLA+ does not formally distinguish between a system specification and a property. Both are expressed as formulas of temporal logic and connected by implication $S \implies F$, where S is a specification and F is a property. Confirming the validity of this implication stands for showing that the specification S has the property F. The TLA+ is accompanied with a set of tools. One of such tool, the TLA+ model checker, TLC, is state-of-the-art model analyzer that can compute and explore the state space of finite instances of TLA+ models. The input to TLC consists of specification file describing the model and configuration file, which defines the finite-state instance of the model to be analyzed. An execution of TLC produces a result that gives answer to the model correctness. In case of finding a problem, this is reported with a state-sequence demonstrating the trace in the model that leads to the problematic state. Inevitably, the TLC suffers the problem of state space explosion that is, nevertheless, partially addressed by a technique known as symmetry reduction allowing for verification of moderate size system specifications.

2. MODEL CONSTRUCTION

The state space construction demonstrated on an introductory example is shown in this section. The example is taken from Lamport's book (see Lamport (2003)). It represents a specification of clocks enriched with minutes (variable `mn`) that makes the specification less trivial but still small enough for complete presentation.

2.1 Table Preparation

Listing 1. Creating s-table and t-table:

```
create table s_hourclock(
  id integer primary key,
  hr integer, mn integer);

create table t_table(
  src integer references s_hourclock(id),
  trg integer references s_hourclock(id),
  act integer);

create sequence seq_s_hourclock
  start with 1 nocycle;

alter table s_hourclock add constraint
  s_hourclock_unique unique(hr,mn)
```

2.2 Initial States

To enumerate and store all initial state in state table the PL/SQL procedure shown in listing 2 is executed. It loops over the variable `hr` and inserts each value in state table, which corresponds to predicate *Init* of the specification.

Listing 2. Generating initial states:

```
for hr in 1..12 loop
insert into s_hourclock values(
  seq_s_hourclock.nextval,hr,0);
end loop;
```

2.3 Action definition

An action consists of guard and computable expression for determination of values in a successive state. The action $A3$ is realized as a stored procedure as shown in listing 3. The evaluation of a guard expression yields to result set that is bound to cursor `c1`. The select statement contains `where` clause expressing that minutes are in interval `(0..58)`. The inner select prevents to get states that where already examined. This is achieved by testing that there is not a transition (i.e. relation in transition table) carried by the action `a3` that starts in the selected state.

Listing 3. Procedure implementing Action 3:

```
create procedure hourclock_a3 as
  dupid s_hourclock.id%type;
  cursor c1 is
    select * from s_hourclock
    where mn>=0 and mn<=58
      and id not in
      (select distinct src
       from r_hourclock
       where act=3);
  rec c1%rowtype;
begin
  loop
    open c1;
    fetch c1 into rec;
    exit when c1%notfound;
    begin
```

MODULE *HourClock*

EXTENDS *Naturals*
VARIABLE hr, mn

$$Init \triangleq hr \in (1 \,..\, 12) \land mn = 0$$
$$A1 \triangleq hr \in (1 \,..\, 11) \land mn = 59 \land hr' = hr + 1 \land mn' = 0$$
$$A2 \triangleq hr = 12 \land mn = 59 \land hr' = 1 \land mn' = 0$$
$$A3 \triangleq mn \in (0 \,..\, 58) \land mn' = mn + 1 \land \text{UNCHANGED } \langle hr \rangle$$
$$Next \triangleq A1 \lor A2 \lor A3$$

$$Spec \triangleq Init \land \Box[Next]_{\langle hr,\, mn \rangle}$$

Fig. 1. HourClock TLA Specification

```
      insert into s_hourclock values
        (seq_s_hourclock.nextval,
         rec.hr, rec.mn+1);
      insert into r_hourclock values
        (rec.id,seq_s_hourclock.currval,3);
    exception when others then
      select id into dupid from s_hourclock
        where hr=rec.hr and mn=rec.mn+1;
      insert into r_hourclock values
        (rec.id,dupid,3);
    end;
    close c1;
  end loop;
end;
```

The loop in the action procedure inserts new states in state table and new transitions in transition table. A new state is computed from the values of the current state as pointed by the cursor. If newly computed state already exists in the state table an exception is raised because the value uniqueness constraint is violated. In this case only the transition is inserted in the transition table. Note that the transition is marked with identification of action `a3`.

2.4 Main Loop

In main loop, which intuitively corresponds to $Next$ predicate, the actions are executed until the set of states stops growing. The implementation is straightforward in PL/SQL by the loop that compares the size of state table before and after the execution of action procedures.

Listing 4. Main Loop:

```
declare
  i integer;
  pi integer;
begin
  select count(*) into pi from s_hourclock;
  loop
    hourclock_a1;
    hourclock_a2;
    hourclock_a3;
    select count(*) into i from s_hourclock;
    exit when pi=i;
    pi:=i;
  end loop;
end;
```

By executing the main loop, the `s_hourclock` contains all reachable states and `r_hourclock` contains all possible transitions of the hourclock specification. These tables can be readily used for querying properties of the model, e.g. checking the type invariant amounts to select all states that violates the type invariant property (see listing 5).

Listing 5. Type Invariant Checking:

```
select * from s_hourclock where not
(
  hr >= 1 and hr <= 12
  and
  mn >= 0 and mn <= 59
)
```

Nevertheless for deeper analysis, if properties are given as formulas fo temporal logic, the state space needs to be considered together with the transition graph to form a transition system. It allows for answering the question of whether the given temporal logic formula holds in this transition system.

2.5 An Issue of Transitive Closure

Before we proceed to define a systematic method for state space exploration, we examine the role of transitive closure (TC) of transition table. Having precomputed TC would greatly simplify algorithms for state exploration. The naive iterative implementation is shown in listing 6.

Listing 6. Transitive Closure of T-Table:

```
create table tc_hourclock as
  select * from r_hourclock;

loop
  insert into tc_hourclock
   (select G.src, TC.trg
    from r_hourclock G, tc_hourclock TC
    where G.trg = TC.src);
  exit when sql%rowcount = 0;
end loop;
```

The (time) complexity of this implementation is $O(n^3)$ for n edges and if appropriate indexes are used the complexity can be reduced to $O(n^2 log\ n)$. These values seem not to be very optimistic if considering large state tables. Although several improvements and alternative methods were studied, e.g. by Libkin and Wong (1997) and Dong et al. (1999), we attempt to avoid the computation of full TC. Note that also exiting database management systems offers for limited implementation of recursive queries, for instance, Oracle's `connect by` query.

3. MODEL EXPLORATION

Although SQL-based querying over the state and transition tables is possible, the usual way of validating reactive models is to check properties defined by terms of a temporal logic. The most straightforward algorithm adaptable for SQL implementation is CTL model checking algorithm based on state labeling.

The algorithm for checking validity of CTL formula ϕ in a (Kripke-style) model M operates by labeling states according to markers that correspond to subformulas of ϕ. The state s is labeled, $s \in label_\psi$, iff the subformula ψ is true in that state. Once the algorithm completes the $M, s \models \phi$ iff $s \in label_\phi$. For further explanation see, e.g. Clarke et al. (1999).

As any CTL formula can be expressed in terms of atomic expression, $\neg$, $\vee$, EX, EU and EG we provide the corresponding labeling procedures only for those cases.

The first three cases are straightforward to implement. An example of labeling an atomic proposition or a proposition consisting of non-temporal subformula is shown on listing 7. The idea is to create a new table that consists of indexes of states that satisfy the given proposition, in this case, $hr = 12 \wedge mn \in (0..30)$.

Listing 7. Labeling atomic expressions:

```
create table l_hourclock_1 as
  select id from s_hourclock where
    hr = 12 and mn >= 0 and mn <= 30
```

Labeling disjunction consists of creating a new table that merges rows of the two subtables that correspond to the subformulas. We only need to guarantee that the resulting table will not contain duplicities.

Listing 8. Labeling g=$f_1 \vee f_2$:

```
-- input: l_hourclock_f1, l_hourclock_f2
-- output: l_hourclock_g
create table l_hourclock_g as
 (select id from l_hourclock_f1
  union
  select id from l_hourclock_f2)
```

Also procedure for the labeling of EX f is easy to implement. To do this we select all states labeled with f and label their predecessors with EX f. The listing 9 provide an example of such labeling. Note that predecessor is accessed in transition table if we consider the current state being indexed by `dst` field.

Listing 9. Labeling g=EX f expressions:

```
-- input: l_hourclock_f
-- output: l_hourclock_g
create table l_hourclock_g as
 (select src
  from r_hourclock, l_hourclock_f
  where dst = id)
```

In the following subsections, we concentrate on non-trivial cases that involves iterative computations.

3.1 Procedure for E[f_1Uf_2]

To handle formulas of the form g = E[f_1Uf_2], the algorithm first finds all states labeled with f_2 (these states are immediately labeled with g). Then the algorithm goes backward, i.e. in the opposite way the transition relation is defined, to find all reachable states labeled with f_1 and labels them with g. The PL/SQL code is in listing 10.

Listing 10. Labeling g=E[f_1Uf_2]:

```
-- input: l_hourclock_f1, l_hourclock_f2
-- output: l_hourclock_g
begin
  create table l_hourclock_g as
    select id from l_hourclock_f2

  loop
    insert into l_hourclock_g
     (select r.src
      from r_hourclock r,
           l_hourclock_g g,
           l_hourclock_f1 f1
      where r.trg = g.id
      and f1.id = r.src
      and r.src not in
        (select id from l_hourclock_g)
     )

    exit when sql%rowcount = 0;

  end loop;
end
```

3.2 Procedure for EG f

The most complicated part is represented by the case EG f that requires analyzing the graph to determinate nontrivial strongly connected components (SCC).

First, we provide a PL/SQL code for computation of SCC adapting the algorithm devised by Tarjan (1971). The algorithm performs depth-first-search traversal in order to find a sink node or a loop. The procedure `visit` (see listing 11) is in the core of the algorithm. It works on `scc` table, which has four column:

id identifies a node, it refers to state table.
root identifies the candidate root node of a strongly connected component of the given node.
comp identifies the strongly connected component.
stack is a number that represents a stack index, i.e. the order on the stack.

The procedure pushes root node passed as the only argument in the `scc` table. In a main loop, it marks the top node as visited and pushes all its children on the stack of nodes waiting for the processing. The processed node is put into the other stack that determines an order, in which

Listing 11. Visit procedure:
```
create or replace
procedure visit
( vertex in number )

visited integer;
leftptr integer := 1;
rightptr integer := 2**31;
node integer;
cursor c1(node integer) is select trg from r_hourclock where src = node and
  trg not in (select id from scc);
rec c1%rowtype;
begin
  delete from scc;
  -- push(root)
  insert into scc values (vertex, vertex, 0, rightptr);
  loop
  -- pop(node)
    begin
      select id into node from scc where stack = rightptr;
    exception when no_data_found then exit;
    end;
    update scc set stack=leftptr where stack = rightptr;
    rightptr := rightptr + 1;
    leftptr := leftptr + 1;
    open c1(node);
    loop
      fetch c1 into rec;
      exit when c1%notfound;
      -- push(child)
      rightptr := rightptr - 1;
      insert into scc values (rec.trg, rec.trg, 0, rightptr);
    end loop;
    close c1;
  end loop;
end visit;
```

the nodes were examined. To simulate two stacks, used by Tarjan's algorithm, it is enough to have only one `stack` column and two sets of indexes (`leftptr, rightptr`) as the node cannot be in both stack simultaneously.

The algorithm for g = `EG` f considers that strongly connected components were determined in the previous step and attempts to find all paths that lead to these SCCs. To do this it proceeds by incrementally increase the labeled set by adding in each step states for those there are transitions ending in the labeled set.

Listing 12. Labeling g=`EG` f:
```
-- input: l_hourclock_f,
-- output: l_hourclock_g
begin
  SCC(f) -- it produces l_hourclock_g
         -- with states in SCC(f)
  loop
    insert into l_hourclock_g
     (select r.src
      from r_hourclock r,
           l_hourclock_g g,
           l_hourclock_f f
      where r.trg = g.id
        and f1.id = r.src
        and r.src not in
          (select id from l_hourclock_g)
      )
    exit when sql%rowcount = 0;
  end loop;
end
```

4. DISCUSSION AND FURTHER WORK

In the present work we introduced an idea of implementing state exploration procedure in the language of relation database. In particular, we demonstrated the idea on examples given in PL/SQL that is the language of Oracle database system. We showed that, in particular, the full implementation of CTL model checking algorithm is straightforward. In the presentation, we did not consider any optimization of state space generation procedure nor the model checking algorithm, although using accompanied profiling tools it is possible to find the performance problems in SQL queries and come with optimization improvements. As the experiments indicate the used underlaying database system offers promising practical platform for automated verification of large scale models. Although, it is not possible to directly compare this implementation with other model checking tools, on several examples we obtain results in time similar to the TLC model-checker accompanied with TLA tool suite. We were also unable to practice bigger case studies as the tool that would automatically generate PL/SQL statements from TLA specification is not fully implemented yet, therefore for all the experiments code was entered manually.

The immediate observations can be split to two classes. The first class considers the practical aspect on the use of the method. To gain advantages of the method one needs to be provided with a set of tools that allows to automatic generation of state space models from specifications, system for property description and state space analysis. For efficiency reasons, the advantage user should be allowed to see preliminary results or to modify generated SQL statements. The second class considers the implementation aspect of the method. The crucial issue behind the implementation of the state exploration method in the environment of relational database is efficient procedure for recursive query evaluation, which appears behind any all non-trivial computations.

As the research done so far only points out the basic ideas, there is a a huge room for further development and improvements of the method. The following list contains the most appealing items for immediate research:

- SQL optimization and DBS-specific optimization techniques should be applied as along the line considered by the second assumption. Currently, only the principle was shown but the further improvements on efficiency need to be done in order to demonstrate that the method can be really considered as a practical tool for validation of industrial scale problems.
- Reusing auxiliary results of CTL model checking procedure is possible as formulas may share same subformulas. The technique that allows to identify the same labeling should be studied in order to use this option transparently to the user. The matching atomic formulas with respect to their logical equivalency is a premise for implementation of efficient reusing technique.
- Incremental model construction or modification that reduces the costs associated with a recomputation of complete state space or sets of labeled states. The incremental approach can increase the methods efficiency but requires more sophisticated approach in state space generation and verification algorithm design. There are numerous work on incremental computation of views generated by recursive algorithms for relational database systems (e.g.), which may help in this course of research.
- Support for component verification that exploits the natural operation of relational databases, e.g. join and intersection. A design consisting of components may easier to treat as for each individual component the state space can be generated and then the state spaces can be combined according to compositional operation defined for a containing component. In this phase, the native SQL operations, which implementations are optimized in relational database can be exploited.

REFERENCES

Clarke, E., Grumberg, O., and Peled, D. (1999). *Model Checking.* The MIT Press.

Dong, G., Libkin, L., Su, J., and Wong, L. (1999). Maintaining the transitive closure of graphs in sql. *In Int. J. Information Technology*, 5.

Lamport, L. (2003). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Professional.

Libkin, L. and Wong, L. (1997). Incremental recomputation of recursive queries with nested sets and aggregate functions. In *In LNCS 1369: Proceedings of 6th International Workshop on Database Programming Languages, Estes Park*, 222–238. Springer-Verlag.

Tarjan, R. (1971). Depth-first search and linear grajh algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, 114–121. doi: 10.1109/SWAT.1971.10.