FI MU

Faculty of Informatics
Masaryk University Brno

Web Portal for Benchmarking Explicit Model Checkers

by

Radek Pelánek

FI MU Report Series FIMU-RS-2006-03

October 2006

Web Portal for Benchmarking Explicit Model Checkers

Radek Pelánek [*]
Department of Computer Science, Faculty of Informatics
Masaryk University Brno, Czech Republic
xpelanek@fi.muni.cz

October 5, 2006

Abstract

We present BEEM — BEnchmarks for Explicit Model checkers. This benchmark set includes more than 50 models together with correctness properties (both safety and liveness). The benchmark set is accompanied by an comprehensive web portal, which provides detailed information about all models. The web portal also includes information about state spaces and facilities selection of models for experiments.

The report describes the rationale beyond the form of the benchmark set, the design of the web portal and the main aspects of its realization, and also an example of an experimental analysis over the benchmark set: an analysis of a performance of sequential and distributed reachability.

The address of the web portal is `http://anna.fi.muni.cz/models`.

1 Introduction

Model checking field underwent a rapid development during last years. Several new, sophisticated techniques have been developed, e.g., symbolic methods, bounded model checking, or automatic abstraction refinement. However, for several important application domains (e.g., mutual exclusion algorithms, communication protocols, controllers, leader election algorithms) we still cannot do much better than the basic *explicit* model

[*]Supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and by the Academy of Sciences of Czech Republic grant No. 1ET408050503.

```
proc ExplicitSearch($M$)
   add $s_0$ to Wait; add $s_0$ to States
   while $Wait \neq \emptyset$ do
         remove $s$ from Wait
         foreach $s \xrightarrow{a} s'$ do
            if $s' \notin States$ then
                           add $s'$ to Wait
                           add $s'$ to States
            fi od
   od
end
```

Figure 1: Basic explicit search.

checking approach — brute force exhaustive state space search (Figure 1). This technique is used by several of the most well-known model checkers (e.g., Spin [14], Murphi [7]). Even some of the software model checkers (e.g., Java PathFinder [12], Zing [1]) are based on the explicit search.

The explicit model checking technique has gained extensively from the progress in computer speed and memory sizes. There has also been progress in the algorithmic improvement of the method, e.g., heuristics for memory consumption, reduction techniques, directed search, distributed search. Altogether, the application scope of the explicit technique has been extended significantly and many realistic case studies showed practical usability of the method.

There is also a very large body of research work devoted to the improvement of explicit model checking. Unfortunately, many papers fail to convincingly demonstrate the usefulness of newly presented techniques. For both researchers and practitioners, it is rather difficult to judge and compare different improvements. In order to enable better development and evaluation of techniques, we need to study practically used models and to develop a benchmark set. This is the goal of this work.

1.1 State of the Art

In order to support the need for benchmarks, we present an evaluation of experiments in model checking papers. We have used a sample of 80 publications which are concerned

Table 1: Quality of experiments reported in model checking papers. For each quality category, we report number of published papers in years 1994-2006.

	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006
Q1	-	-	1	1	1	1	1	3	2	4	2	1	1
Q2	-	-	3	3	2	3	3	1	2	2	2	1	-
Q3	-	2	1	3	1	2	2	1	3	2	2	4	1
Q4	1	-	-	-	1	-	1	4	1	1	2	-	2
Q5	-	-	-	1	-	-	-	-	1	-	1	-	-

with explicit model checking and contain an experimental section[1]. Experiments in each of these publications were classified into one of the following five categories (the distinction among toy models, simple models, and case studies is described in Section 2.3):

Q1 Random inputs and/or few toy models.

Q2 Several toy models (possibly parametrized) or few simple models.

Q3 Several simple models (possibly parametrized) or one large case study.

Q4 Exhaustive study of parametrized simple models or several case studies.

Q5 Exhaustive study with the use of several case studies.

Table 1 presents the quality of experiments in papers from our sample (list of all used papers and their classification is in Appendix B). Although the classification is slightly subjective, it is clear from Table 1 that there is nearly no progress in time towards higher quality of used models. This is rather disappointing, because more and more case studies are available. Low experimental standards make it hard to assess newly proposed techniques and obstruct the progress of the research in the field. As we have discussed in our evaluation of on-the-fly reduction techniques [17], the practical impact of many techniques can be quite different from claims made in publications. Clearly, a good benchmark set is missing.

[1]The sample was obtained by collecting all relevant papers from SPIN, TACAS and CAV conferences and browsing their citations and references to them. Nevertheless, this sample is certainly not meant to be complete.

The need for benchmarking, better experiments, and thorough evaluation of tools and algorithms is well recognized, e.g., experimentation is a key part of Hoare's proposal for a Grand Challenge of Verified Software [13]. As can be seen from the below given discussion of the related work, there is significant interest in benchmarks in the model checking community. Nevertheless, the progress up to date has been rather slow. The main obstacle in developing model checking benchmarks is the absence of a common modeling language — each model checking tool is tailored towards its own modeling language and even verification results over the same example are often incomparable.

Although the development of benchmarks is difficult and the model checking community will probably never have a universal benchmark set, we should try to build benchmarks as applicable as possible and steadily improve our experimental analysis. This is the goal of this work.

1.2 Our Approach

We present BEEM — a benchmark set which is built over the following principles.

Modeling Language

Models are implemented in a *low-level modeling language* based on communicating extended finite state machines (DVE language, see [18] for syntax and semantics). The adoption of a low-level language makes the manual specification of models harder, but it has several advantages. The language has a simple and straightforward semantics; it is not difficult to write own parser and state generator. It can also be automatically translated into other modeling languages — at the moment, the benchmark set includes also *Promela* models which were automatically generated from DVE sources.

Models and Properties

Most of the models are *well-known* examples and case studies. Models span several *different application areas* (e.g., mutual exclusion algorithms, communication protocols, controllers, leader election algorithms, planning and scheduling, puzzles). In order to make the set organized, models are *classified* into different types and categories. The benchmark set is *large* and still growing (at the moment it contains 56 parametrized models with 276 specified instances). *Source codes* of all models are publicly available.

Models are briefly described and include *pointers to sources* (e.g., paper describing the case study).

The benchmark set includes also *correctness properties* of the models. Since important part of model checking is error detection, the benchmark set includes also *models with errors* (presence of an error is a parameter of the model).

Tool Support

The modeling language is supported by an *extensible model checking environment* — The Distributed Verification Environment (DiVinE) [3]. DiVinE is both a model checking tool and a open and extensible library for a development of model checking algorithms. Researchers can use this extensible environment to implement their own algorithms, easily perform experiments over the benchmark set, and directly compare with other algorithms in DiVinE. Promela models can be used for comparison with the well-known model checker Spin [14].

Web Portal

The benchmark set is accompanied by an comprehensive web portal, accessible at `http://anna.fi.muni.cz/models`, which should *facilitate the experimental work*. The web provides:

- presentation of all information about models, their parameters, and correctness properties,
- detailed information about properties of state spaces of models [16] including summary information,
- verification results,
- web form for selection of suitable model instances according to a given criteria,
- instance generator, which can generate both DVE models and Promela models for given parameter values.

1.3 Content of the Report

This report describes the used modeling language, content of the benchmark set, classification of models, design and realization of the web portal. We also demonstrate several simple experimental applications over the benchmark set. We study the behaviour of simple sequential and distributed reachability. The experiments show how much the experimental results depend on used models, that it can make difference if we use toy or complex models. We also demonstrate the speed of state space traversal, and speed-up in distributed environment.

1.4 Related Work

Corbett [6] was the first to evaluate model checking tools over a large benchmark set. He used models described in Ada; this benchmark set is now rather outdated. There have been several other systematic experimental works [5, 4, 17], but they do not provide any generally usable benchmarks.

Detailed arguments for the need of benchmarking in model checking were given by Avrunin et al. [9]. They even started to build a web portal of models[2]. Their portal is simpler then ours and it was never finished — it contains only very small number of models.

Atiya et al. [2] described benchmark proposal based on benchmarking theory. They provide a carefully selected list of models, describe them, and discuss studies that used these models. Unfortunately, their benchmark proposal does not include source codes of models. Moreover, selected models are from different tools (and hence expressed in different modeling languages). This makes it very difficult to use this benchmark set for practical experiments.

Jones et al. [15] propose a preliminary benchmark set for benchmarking parallel model checkers. Their set consists of just two parametrized artificial models; we consider such set completely insufficient for reasonable experiments.

Most of the well known model checking tools (Spin, Uppaal, Murphi, CADP) have their own sets of examples, either directly included in distribution or as a stand alone collection (Promela model database[3], CADP case studies[4]). These collections, however, lack systematic organization and usually cover only some of part of the explicit model checkers' application domain.

The *Very Large Transition Systems* benchmark suite[5] and our study of properties of state spaces [16] both contain large number of representative examples which are more-

[2] `http://laser.cs.umass.edu/verification-examples/`

[3] `http://web.tiscali.it/ikaria/alberto/promela_models/models.html`

[4] `http://www.inrialpes.fr/vasy/cadp/case-studies/`

[5] `http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html`

over systematically organized. Unfortunately, both these studies provide only state spaces and not models. This makes them inapplicable for most of the model checking experiments.

Specialized web portals have proved very useful in several related disciplines. Specification pattern system [8] provides a database of verification properties. Petriweb portal [10] provides a collection of Petri nets, particularly for pedagogical purposes. Some of its design aspects (e.g., selection of models according to given parameters) are similar to the design of our portal.

2 The Benchmark Set BEEM

In this section we describe the used modeling language, models included in the benchmark set, and the used classification of models. At the moment, the benchmark set contains 56 different models and a 276 specified instances of these models.

2.1 The DVE Modeling Language

Models in BEEM are presented in the DVE modeling language, which is the native input language of DiVinE [3]. DVE language is a low-level language based on communicating extended finite state machines. The language is similar to other modeling languages (Spin's Promela, Uppaal's networks of extended timed automata, CADP's LOTOS, etc.). The language is designed to have straightforward semantics and for easy automatic processing — so it is not very user friendly. Nevertheless, it is expressive enough to model all classical examples and case studies. Figure 2 shows an example of a DVE code.

The basic features of the language are the following:

- Model is comprised of a network of processes; processes execute asynchronously.
- Each process is a finite state machine extended with finite domain variables (including one dimensional arrays). Each transition is comprised from guard, synchronization and update of variables; transitions execute atomically.
- Processes can synchronize via two-party handshake (with optional value passing).

The full (abstract) syntax and semantics of the language is given in Appendix A. Useful features, like more dimensional arrays, buffered communication, committed states, (discrete) timers, etc. can be realized using low-level mechanisms and macros[6]. For macros we use of a powerful, general `m4` preprocessor[7]. The preprocessor is also used to represent parametrized versions of models. In this way we have only one source file for each model.

BEEM contains two types of source codes:

- MDVE files: manually written source code with directives for preprocessor (includes macros and declarations of model parameters),
- DVE file: the low-level input code for the model checker or for automatic translation to other modeling languages; DVE files are generated from MDVE files by the preprocessor.

2.2 Correctness Properties

For each model, BEEM also includes several correctness properties of the model. The properties are expressed over atomic propositions, which are defined by expressions over model variables. At the moment, we support two types of properties: reachability properties and linear temporal logic (LTL) properties.

2.3 Classification of Models

Models are classified according to two criteria: type and complexity. Type of model is one of the following application domains: mutual exclusion; communication protocols; leader election algorithms; controllers; other protocols; puzzles, planning, scheduling; other. Table 2 presents for each class of models their typical characteristics, typical correctness properties used over this class of models, and some characteristics of state spaces of these models. Each type of model is briefly described below.

Beside this classification, models in BEEM are also classified according to the 'complexity' of the model. We use the same classification as in our previous work [16]:

- toy examples: very simple examples which are usually used for teaching or as an explanatory examples in research papers; these models can be specified in just few lines of code,

[6]Note that the current version of DiVinE supports an extended version of the language which includes some higher lever constructs like committed states and buffered channels. Nevertheless, BEEM is kept in the basic version of the language.

[7]`http://www.gnu.org/software/m4/`

```
process Cache_0 {
byte value = 0; int m,m2 = -1;
state   invalid, i_bus_req, i_app_read, i_app_write, iv1, iv2, id1, set_value,
        valid, v_bus_req, v_app_read, v_app_write, wait_bus_ack,
        dirty, d_bus_req, d_app_read, error_st;
init valid;
trans
 invalid -> i_bus_req { sync bus_0?m; },
 i_bus_req -> invalid { guard m == 1 || m == 2; sync bus_0!-1; },
 i_bus_req -> invalid { guard m == 3; },
 invalid -> i_app_read {sync read_0?m; },
 i_app_read -> iv1 { sync bus_0!1; },
 iv1 -> iv2 { sync bus_0?value; },
 iv2 -> valid { sync answer_0!((value & (1<<m))/(1<<m)); },
 invalid -> i_app_write {sync write_0?m;},
 i_app_write -> id1 { sync bus_0!2;},
 id1 -> set_value {sync bus_0?value;},
 set_value -> dirty {
        sync answer_0!-1;
        effect value = value - (value & (1<<((m/16)))) + ((m%16) * (1<<((m/16)))); },
 valid -> v_bus_req { sync bus_0?m; },
 v_bus_req -> valid { guard m == 1; sync bus_0!-1;},
 v_bus_req -> invalid { guard m == 3; },
 v_bus_req -> invalid { guard m == 2; sync bus_0!-1; },
 valid -> v_app_read { sync read_0?m; },
 v_app_read -> valid { sync answer_0!((value & (1<<m))/(1<<m)); },
 valid -> v_app_write { sync write_0?m; },
 v_app_write -> wait_bus_ack { sync bus_0!3; },
 wait_bus_ack -> set_value { sync bus_0?m2; },
 dirty -> d_bus_req {sync bus_0?m; },
 d_bus_req -> valid { guard m == 1; sync bus_0!value; },
 d_bus_req -> invalid { guard m == 2; sync bus_0!value; },
 d_bus_req -> error_st { guard m == 3; },
 dirty -> d_app_read { sync read_0?m; },
 d_app_read -> dirty { sync answer_0!((value & (1<<m))/(1<<m)); },
 dirty -> set_value { sync write_0?m; };
}
```

Figure 2: An example of a DVE code: Synapse cache coherence protocol, code of a process modeling one cache.

- simple models: non-trivial models inspired by real systems, but very simplified,
- complex case studies: models used in case studies to reason about some real system; the description of the system have more than 100 lines of code.

The goal of this classification is to test to what extent do experimental results differ for models of different complexity (see Section 4).

2.4 Overview of Models

Here we provide description of types of models and we also models list currently included in the benchmark set.

2.4.1 Mutual Exclusion Algorithms

The goal of a mutual exclusion algorithm is to ensure an exclusive access to a shared resource. Models of these algorithms usually consist of several nearly identical processes which communicate via shared variables.

Models currently included in the set: Anderson's queue lock algorithm, Alur-Taubenfeld's algorithm, Bakery algorithm, Dining philosophers problem, Driving philosophers problem, Fischer's algorithm, Lamport's algorithms, MCS queue lock algorithm, Peterson's algorithm, and Szymanski's algorithm.

2.4.2 Communication Protocols

The goal of communication protocols is to ensure communication over an unreliable medium. The core of a model of a communication protocol usually comprise of a sender process, a receiver process, and a bus/medium. Processes communicate by handshake; shared variables are not used.

Models currently included in the set: Bounded retransmission protocol, Cambridge ring protocol, Collision avoidance protocol, Layer link protocol of the IEEE-1394, Sliding window protocol, Pragmatic general multicast protocol, and Rether protocol.

2.4.3 Leader Election Algorithms

The goal of leader election algorithms is to choose a unique leader from a set of nodes. Models consist of a set of (nearly) identical processes connected in a ring, tree, or arbitrary graph; communication is via (buffered) channels.

Type	model characteristics	correctness properties	state space
mutual exclusion	processes: similar, simple communication: shared variables structure: clique or ring	safety: mutual exclusion response: wait $\rightarrow$ CS	low BFS height one large SCC component long cycles
communication protocols	processes: different, complex communication: (lossy/buffered) channels structure: linear, tree	safety: correct message response: sent $\rightarrow$ receive	one large SCC
leader election algorithms	processes: similar, simple communication: channels structure: ring, graph	safety: at most one leader liveness: eventually some leader	acyclic graph or small SCCs
controllers	processes: different, complex communication: synchronous channels, shared variables structure: centralized (star)	safety: absence of an error response: signal $\rightarrow$ reaction	small average degree large BFS height complex SCC structure
puzzles, planning, scheduling	processes: one or few communication: none or shared variables	reachability: solution	big average degree small BFS height

Table 2: Overview of basic types of models.

Models currently included in the set: Algorithms based on extinction and on filters, Firewire (IEEE 1394) tree identification protocol, Lann leader election algorithm for token ring, and leader election algorithm on unidirectional ring by Dolev et al.

2.4.4 Controllers

Models of controllers usually have centralized architecture: a controller process communicates with processes representing individual parts of the system. The communication can be both by shared variables and handshake communication.

Models currently included in the set: Audio/video power controller, Elevator controller, Gear controller, Distributed system for lifting trucks, Programmable logic controller program, Production cell case study, Train-gate controller.

2.4.5 Puzzles, Planning, Scheduling

Planning and scheduling problems and puzzles are not the main application domain of explicit model checkers. Nevertheless, there are good reasons to include (some of these problems) in our benchmark set:

- These examples are often used for teaching and as explanatory examples in research papers (for example [19]).
- These examples can be easily modeled by the same formalisms as asynchronous systems.
- Recently, there have been interest in applying model checking methods in artificial intelligence.

Models currently included in the set: Blocks world, Scheduling machines for production, Bridge puzzle, Peg solitaire puzzle, sliding block puzzles (Rushhour, Sokoban).

2.4.6 Other

The set includes several other models, which are not directly covered by any of the above mentioned categories, for example: Milner's cyclic scheduler, Telecommunication service protocol, Needham-Schroeder public key authentication protocol, Sharing SRAM and CAM by lookup processors, Synapse cache coherence protocol.

3 The Web Portal

In order to make BEEM more accessible and usable, we have built a web portal over it. This section present the web portal and its realization.

3.1 Functionality

The web portal provides the following functionality:

- a listing of all models and their classification,
- detailed information about each model:
  - description of a modeled system,
  - reference to the original source of the model,
  - description of parameters,
  - correctness properties,
  - list of instances included in BEEM ,
- additional information for each specified instance:
  - size of the state space,
  - details about properties of the state space (only for instances with a small state space),
  - verification results for the instance including lengths of counterexamples,
  - Promela model, which is automatically generated from DVE source code,
- summary information about properties of state spaces (distribution of numerical values: histograms, quartiles, mean),
- instance generator, which can generate both DVE models and Promela models for given parameter values,
- selection of instances according to a given criteria; this should facilitate the selection of examples for experiments; the user can choose instances according to size and results of verification.

The benchmark set (including model descriptions) is also available for download.

3.2 Properties of State Spaces

The web portal includes information about properties of state space — this follows the line of our previous research [16]. Information about state space properties can be useful for explanation of results of experiments, therefore we consider it important to include it in the web portal.

For each state space we report the following information (for description of these properties see [16]):

- size of the state space: number of states, edges,
- degree information: average degree, distribution of in-degrees and out-degrees,
- structure of strongly connected components: number of components, size and position of the largest component, number of trivial components, height of the SCC quotient graph,
- breadth and depth first search information: maximal size of queue/stack, number of BFS levels, number of back/cross/front edges,
- information about local structure: number of diamonds, clustering coefficient,
- information about labels of states,
- visualization of parts of the state space.

Information about properties of a state space is given for each model instance which have state space smaller than a given fixed limit (60,000). Summary information about the distribution of numerical values is also provided. For model instances with state space larger then the given limit we provide only the number of states and edges.

3.3 Realization of the System

Now we briefly describe the implementation of the web portal. Figure 3 presents sketch of relations among different parts of the system.

3.3.1 Example Description

For each example we have a description file in XML format. This file contains the following informations:

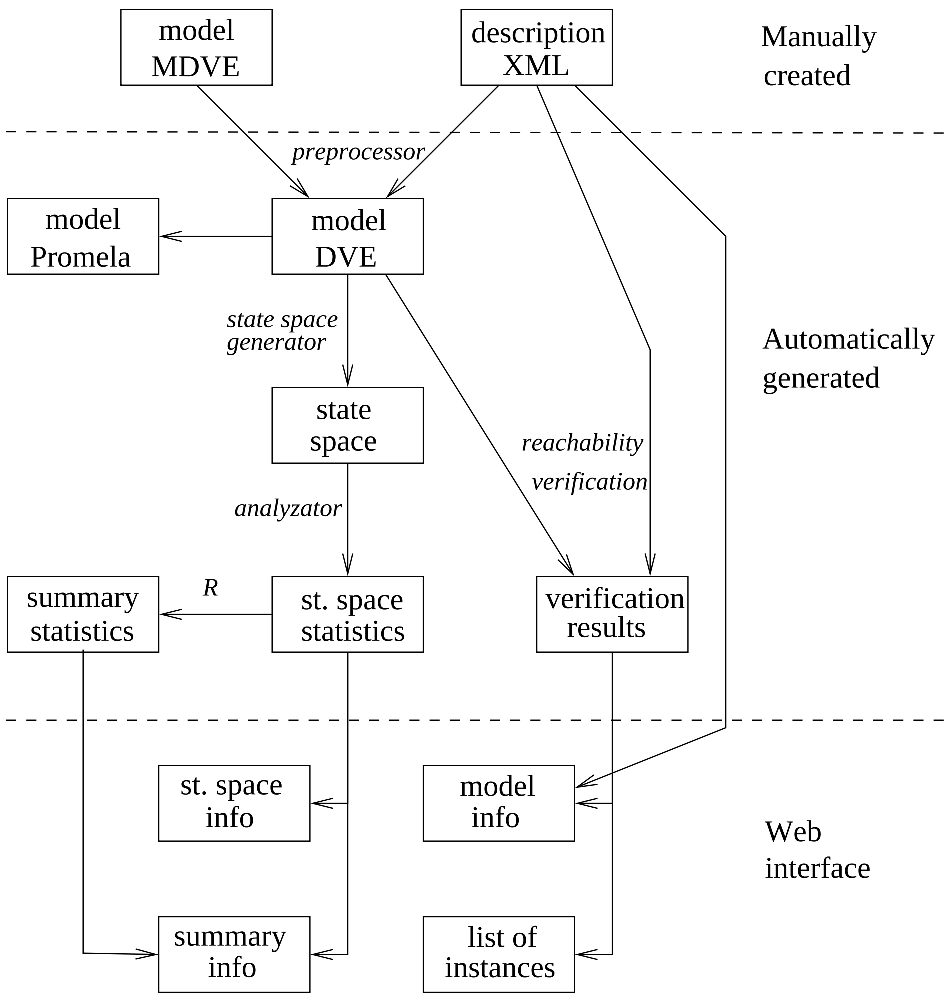


Figure 3: Overview of the realization of the web portal.

- classification of the example (see Section 2.3),
- description of the example, source of the example (e.g., reference to a research paper),
- description of model parameters,
- definition and description of atomic propositions and correctness properties (see Section 2.2),
- definition of specific instances of the model (concrete values of parameters).

Figure 4 gives an example of a model description file.

3.3.2 Programs for Automatic Processing

For each model, the administrator of the web portal provides only an MDVE source code and an XML file with a description of the model. All other files and informations are automatically generated:

- DVE and Promela source codes for instances: generated by `m4` preprocessor and `dve2promela` translator,
- informations about a model and its state space, verification results for specified correctness properties: generated by programs implemented in C++ with the use of the DiVinE library; results are stored as XML files,
- graphical representation of the informations and summary statistics about state spaces: generated using `dot`, `R`, and `gnuplot`.

Scripts for batch execution and processing of models are implemented in Perl.

3.3.3 Web Interface

Both the manually provided data and informations computed by programs are represented as XML files. This information is presented to the user by web interface, which is implemented as CGI scripts in Perl.

```
<model  type="other-protocol" classification="complex">
  <short-description>Synapse cache coherence protocol</short-description>
  <long-description>Synapse cache coherence protocol: several caches are
    connected by a bus, the goal of the protocol is to keep the content of the
    caches coherent. </long-description>
  <source>Manually translated from Promela source code (included in SPIN
    distribution).</source>
  <parameter-description>
    <par-name>Lines</par-name>
    <par-description>Number of lines in a cache</par-description>
  </parameter-description>
  <parameter-description>
    <par-name>N</par-name>
    <par-description>Number of applications and caches</par-description>
  </parameter-description>
  <ap> <ap-name>cerror</ap-name>
    <ap-expression>Cache_0.error_st</ap-expression> </ap>
  <ap> <ap-name>write11</ap-name>
    <ap-expression>written_line==1 &amp;&amp; written_value==1</ap-expression></ap>
  ...
  <property id="1" prop-type="reachability">
    <formula>cerror</formula>
    <description>Cache gets into an error state.</description>
  </property>
  ...
  <property id="3" prop-type="LTL">
    <formula> G (write11 -> ((G ! write10) || ((! read10) U (write10)) ))</formula>
    <description>If we write 1 to line 1 and do not override it until
    next reading then the next reading from this line is 1. </description>
  </property>
  <instance id="1">
    <parameters>Lines=2,N=2</parameters>
  </instance>
  ...
</model>
```

Figure 4: An example of a description file: Synapse cache coherence protocol. Note that only some parts of the XML file are listed.

```
void search() {
  Wait.push(init);
  States.insert(init);
  while (! Wait.empty()) {
    state_t q = Wait.front(); Wait.pop();
    System.get_enabled_trans(q, trans);
    for (enabled_trans_container_t::iterator i = trans.begin();
         i != trans.end(); i++) {
      state_t r;
      System.get_enabled_trans_succ(q,*i, r);
      if (! States.is_stored(r)) {
        Wait.push(r);
        States.insert(r);
      }
    }
    delete_state(q);
  }
}
```

Figure 5: DiVinE implementation of the basic explicit search from Figure 1.

4 Experimental Applications

In this section we demonstrate an example of application of BEEM to an experimental study of a basic sequential and distributed reachability analysis. This analysis demonstrates the dependence of a model checker on an input model. It also illustrates that performance results can differ for toy and complex problems.

Algorithms were implemented in the DiVinE library [3]. The library provides basic primitives for implementation of (distributed) model checking algorithms, so the implementation is very close to the pseudocode of the algorithm (see Figure 5 and compare it with Figure 1). Experiments were run on 2.60GHz processor with 4 GB RAM memory, distributed experiments were run on a cluster of 20 dedicated workstations connected by 1GB Ethernet.

4.1 Sequential Reachability

At first, we study the performance of sequential reachability. We are concerned with the speed of exploration, i.e., the number of states generated per second. We have run the reachability on 210 instances and obtained the following results (see Figure 6):

- The speed of exploration depends quite significantly on the model. The typical value is between 50,000 and 100,000 states per second, extremes are at 7,000 and 230,000, i.e., the difference in the speed on different models can be as large as 30 times.
- The model checker is slower for complex models.

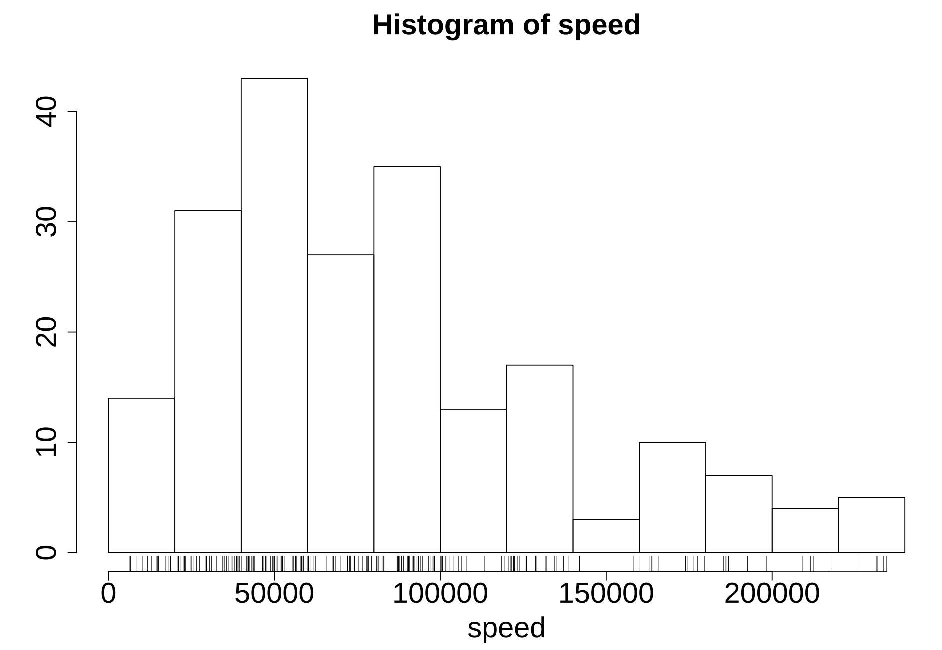
We have also done some profiling of the code. It is a well known rule of thumb that 90% of the execution time of a program is spent in 10% of the code. Since the reachability analysis consist of very large number of repetitions of a simple cycle, this general "90/10 law" demonstrates itself very strongly in this case. Therefore, it is a good idea to target any optimization efforts to a particular part of code.

However, it shows up that the part of code that is most prominently used also depends on a specific model. Here we present only summary analysis that shows the main trend. Operations during the reachability analysis can be roughly divided into two main classes: successor computation operations (e.g., `get_enabled_trans`) and operations for manipulation with storage (e.g, `insert` and `is_stored`). Figure 7 shows the ratio between time spent by these two types of operations. The figure demonstrates that the ratio vary quite significantly and that the time spent by successor generation is higher for complex models.

4.2 Distributed Reachability

At second, we study the performance of distributed reachability. In this case the basic performance parameter in which we are interested is the speedup, i.e., how many times is the computation faster on n workstations than on one workstation. Optimally, we would like to get near to a linear speedup, but this is not realistic due to a communication overhead[8].

[8]Nevertheless, sometimes one can get even superlinear speedup. This can happen in the case that the computation on one workstation uses too small hash table or it starts swapping.



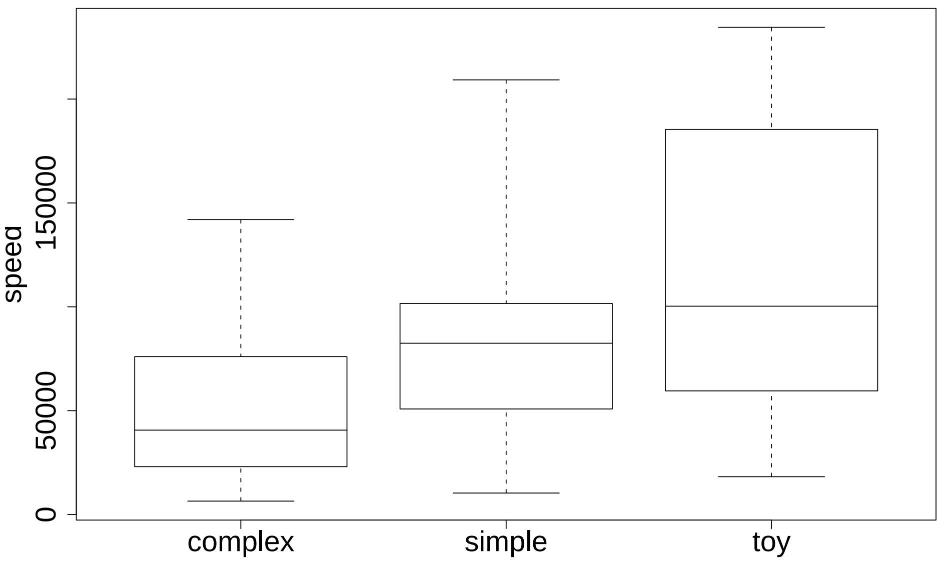


Figure 6: The speed of state generation during the state space search. The histogram shows the number of instances over which the state space generator's speed is in the given interval. The second graph shows the speed according to the type of the model using the boxplot method (minimum, quartiles, and maximum are shown).

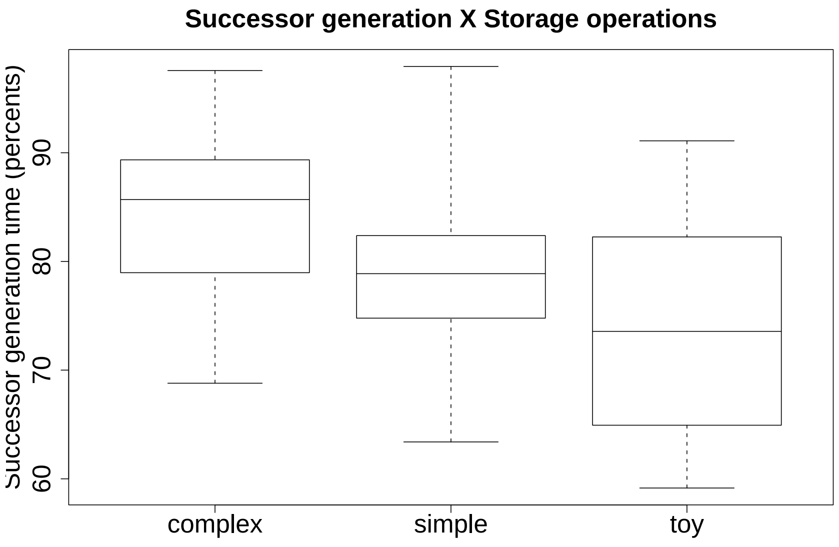


Figure 7: Percentage of time spent by successor generation according to the type of the model (boxplot method).

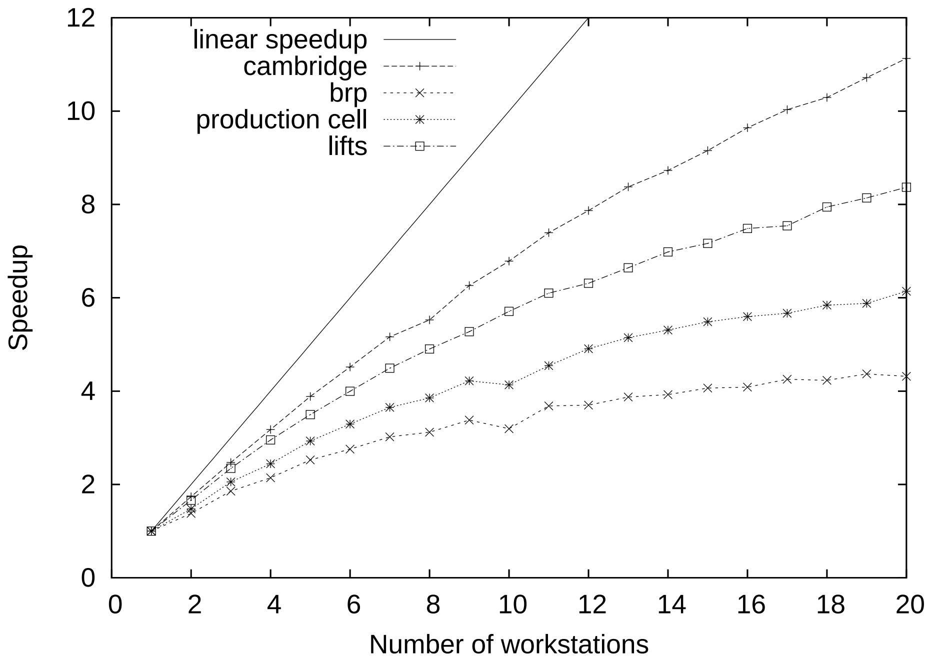


Figure 8: Speedup for 4 sample models for 1 to 20 workstations.

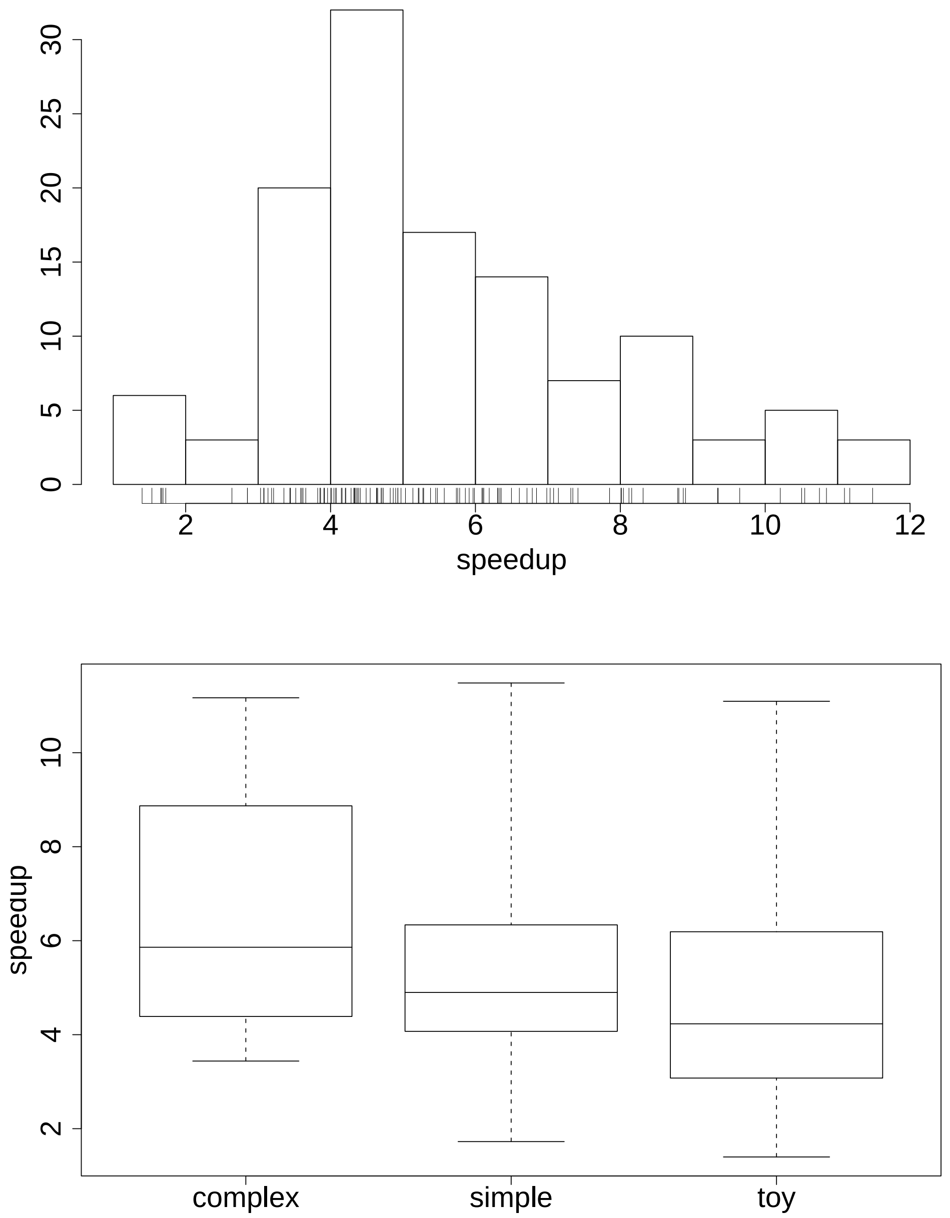


Figure 9: Speedup on 20 workstations. The same type of presentation as in Figure 6 is used.

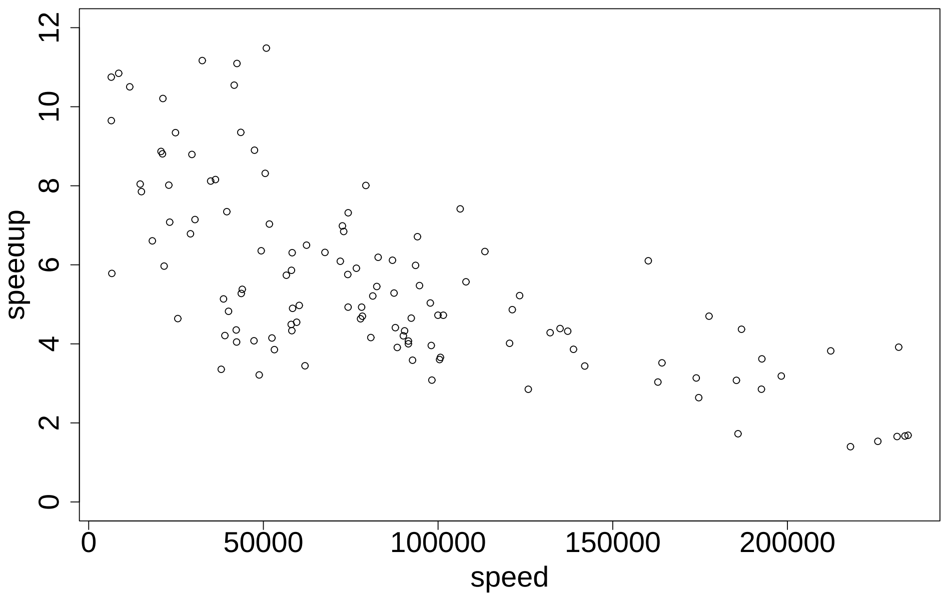


Figure 10: Correlation between speed and speedup.

Figure 8 shows the speedup for several sample models for 1 to 20 workstations (we have run the experiment for more models, these are representative results). The achieved speedup depends on the model, but for a given model it is consistent — it keeps increasing linearly as we add more workstations.

Figure 9 reports speedup achieved on 20 workstations (experiments were run on 120 instances with sufficiently large state space). The speedup vary quite significantly: from 2 to 12, the typical speedup is between 4 and 6. The figure also demonstrates that the type of an input model is again important: for complex models we get a better speedup. This suggests that there may be correlation between speed and speedup.

Figure 10 shows the relation between speed of state generation (as studied above) and the speedup. The correlation is quite clear; the correlation coefficient is -0.68. For models, where the speed is low, the communication overhead is not so significant and thus the speedup is better.

5 Conclusions and Future Work

This report present BEEM — a benchmark set for explicit model checkers. Compared to other similar benchmarks sets, our set has two main advantages:

1. The set is large and contains diverse set of models.

2. The set has a good web support which facilities application of the set.

The report also briefly demonstrates how this benchmark set can be used to get better insight into the behaviour of model checking algorithms.

In future we plan to further extend BEEM and the web portal. Specifically, we would like to do the following:

- add more models,
- add more correctness properties and model variants (particularly more erroneous variants) to existing models,
- add fairness requirements to correctness properties,
- improve the automatic translation to Promela and add other translators (e.g. into LOTOS, Uppaal input language),
- incorporate compact visualization of the whole state space [11],
- create more sophisticated support for experimental analysis.

This benchmark set gives many possibilities for future experimental research, for example:

- thorough comparison of competitive model checking algorithms (e.g., distributed LTL model checking algorithms),
- analysis of practical usefulness of reduction techniques,
- performance analysis of model checking algorithms, with the goal to identify which model/state space attributes (parameters) influence the performance most significantly.

Acknowledgement I thank Pavel Krčál and to members of the DiVinE group, particularly Ivana Černá, Pavel Šimeček and Jiří Barnat, for collaboration, discussions, and feedback.

References

[1] T. Andrews, S. Qadeer, S. K. Rajamani, Y. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proc. of Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 484–487. Springer, 2004.

[2] D. A. Atiya, N. Catano, and G. Lüettgen. Towards a benchmark for model checkers of asynchronous concurrent systems. In *Fifth International Workshop on Automated Verification of Critical Systems: AVOCs*, University of Warwick, United Kingdom, Sept. 12–13 2005.

[3] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Rockai, and P. Šimeček. Divine - a tool for distributed verification. In *Proc. of Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 278–281. Springer, 2006. The tool is available at `http://anna.fi.muni.cz/divine`.

[4] A. Chamillard. An empirical comparison of static concurrency analysis techniques, 1996.

[5] A. T. Chamillard, L. A. Clarke, and G. S. Avrunin. Experimental design for comparing static concurrency analysis. Technical report, Amherst, MA, USA, 1996.

[6] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng.*, 22(3):161–180, 1996.

[7] D. L. Dill. The mur*hi* verification system. In *Proc. of Computer Aided Verification (CAV 1996)*, volume 1102 of *LNCS*, pages 390–393. Springer, 1996.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proc. Workshop on Formal Methods in Software Practice*, pages 7–15. ACM Press, 1998.

[9] M. B. Dwyer G. S. Avrunin, J. C. Corbett. Benchmarking finite-state verifiers. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):317–320, 2000.

[10] R. Goud, K. M. van Hee, R. D. J. Post, and J. M. E. M. van der Werf. Petriweb: A repository for petri nets. In *Proc. of Petri Nets and Other Models of Concurrency (ICATPN 2006)*, volume 4024 of *LNCS*, pages 411–420. Springer, 2006.

[11] J.F. Groote and F. van Ham. Large state space visualization. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *LNCS*, pages 585–590. Springer, 2003.

[12] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[13] T. Hoare. The ideal of verified software. In *Proc. of Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 5–16. Springer, 2006.

[14] G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.

[15] M. Jones, E. Mercer, T. Bao, R. Kumar, and P. Lamborn. Benchmarking explicit state parallel model checkers. In *Proc. of Workshop on Parallel and Distributed Model Checking (PDMC'03)*, volume 89 of *ENTCS*. Elsevier, 2003.

[16] R. Pelánek. Typical structural properties of state spaces. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.

[17] R. Pelánek. Evaluation of on-the-fly state space reductions. In *Proc. of Mathematical and Engineering Methods in Computer Science (MEMICS'05)*, pages 121–127, 2005.

[18] R. Pelánek. Web portal for benchmarking explicit model checkers. Technical Report FIMU-RS-2006-03, Masaryk University Brno, 2006.

[19] A. Valmari. What the small rubik's cube taught me about data structures, information theory and randomisation. *International Journal on Software Tools for Technology Transfer (STTT)*, 2006. To appear.

A Syntax and Semantics of a DVE Modeling Language

We define an abstract syntax and semantics of the DVE modeling language. The concrete syntax is a natural one (see Figure 2 and source codes of models in the benchmark set).

A.1 Abstract Syntax

A.1.1 Expressions

Let V be a set of variables. We treat all variables as one dimensional array variables of a bounded length — scalar variables are treated as a special case of array variables of length 1, more dimensional arrays can be easily simulated by one dimensional.

Set of all expressions over V is denoted $\mathit{Expr}(V)$ and is defined as:

$$e := i \mid V[e] \mid -e \mid e \;\mathit{intop}\; e \mid e \;\mathit{relop}\; e \mid \neg e \mid e \;\mathit{boolop}\; e$$

where:

- *intop* is an integer operator $(+, -, /, \cdot, \ldots)$
- *boolop* is a boolean operator $(\wedge, \vee)$
- *relop* is a relation operator $(<, >, \leq, \geq, =)$

Expressions are used both as integer values (in assignments) and as boolean values (in guards).

A.1.2 Effects

Set of all effects over V is denoted $\mathit{Eff}(V)$ and is defined as follows ($a \in V, e \in \mathit{Expr}(V)$):

$$\mathit{Eff} := a[e] = e \mid \mathit{Eff};\; \mathit{Eff}$$

A.1.3 Synchronization

Let Σ be a finite set of communication channels and τ a special channel not included in Σ. Then the set of all synchronizations over Σ and V is denoted $S(\Sigma, V)$ and is defined as follows:

$$S := \tau \mid \Sigma! \mid \Sigma? \mid \Sigma?V[\mathit{Expr}(V)] \mid \Sigma!\mathit{Expr}(V)$$

A.1.4 Extended Finite State Machines

Extended Finite State Machine (EFSM) over V and Σ is a tuple $A = (Q, q_0, E)$ where Q is a set of states, $q_0 \in Q$ is an initial state, and $E \subseteq Q \times \mathit{Expr}(V) \times S(\Sigma, V) \times \mathit{Eff}(V) \times Q$ is a set of edges. In the concrete syntax we allow machines (processes) to have local variables.

A.1.5 Network

Network of EFSMs over V and Σ is a tuple $\mathcal{N} = (A_1, \dots, A_n, \mathit{dom}, \mathit{length}, \gamma_0)$, where

- $A_i = (Q^i, q_0^i, E^i)$ are automata over V and Σ
- $\mathit{dom} : V \to I(\mathbb{Z})$ assign to each array variable its domain (an interval in $\mathbb{Z}$)
- $\mathit{length} : V \to \mathbb{N}$ assign to each array variable its length (i.e. the size of the array)
- γ_0 is an initial valuation

A.2 Semantics

Let us suppose that we have a network $\mathcal{N} = (A_1, \dots, A_n, \mathit{dom}, \mathit{length}, \gamma_0)$. Variables are interpreted over $\mathbb{Z}_\perp$ – integer numbers extended with $\perp$ (undefined). Any operation with $\perp$ yields $\perp$. A *valuation* is a function $\gamma : V \to (\mathbb{N} \to \mathbb{Z}_\perp)$, such that $\forall a \in V : (\forall 0 \leq i < \mathit{length}(a) : \gamma(a)(i) \in \mathit{dom}(a) \wedge \forall i \geq \mathit{length}(a) : \gamma(a)(i) = \perp)$. Let Γ be a set of all valuations for given $V, \mathit{dom}, \mathit{length}$.

A.2.1 Expressions

Evaluation of an expression e with respect to a valuation $\gamma \in \Gamma$ is denoted $[\![e]\!]_\gamma$ and is defined as follows:

- $[\![\mathit{number}]\!]_\gamma = \mathit{number}$
- $[\![a[e]]\!]_\gamma = \begin{cases} \gamma(a)([\![e]\!]_\gamma) & a \in V, 0 \leq [\![e]\!]_\gamma < \mathit{length}(a) \\ \perp & \text{otherwise} \end{cases}$

- $[\![-e]\!]_\gamma = -[\![e]\!]_\gamma$
- $[\![e_1 \mathit{ intop } e_2]\!]_\gamma = [\![e_1]\!]_\gamma \mathit{ intop } [\![e_2]\!]_\gamma$
- $[\![e_1 \mathit{ relop } e_2]\!]_\gamma = \begin{cases} 1 & [\![e_1]\!]_\gamma \mathit{ relop } [\![e_2]\!]_\gamma \\ 0 & \text{otherwise} \end{cases}$
- $[\![\neg e]\!]_\gamma = \begin{cases} 1 & [\![e]\!]_\gamma = 0 \\ 0 & \text{otherwise} \end{cases}$
- $[\![e_1 \mathit{ boolop } e_2]\!]_\gamma = \begin{cases} 1 & ([\![e_1]\!]_\gamma \neq 0) \mathit{ boolop } ([\![e_2]\!]_\gamma \neq 0) \\ 0 & \text{otherwise} \end{cases}$

As expressions are used in guards as boolean values, we use the notation $\gamma \models e$ with the meaning $[\![e]\!]_\gamma \neq 0$.

A.2.2 Effects

Effects are interpreted as functions $\Gamma \to (\Gamma \cup \mathit{error})$, i.e. effect applied to valuation yields either 'updated' valuation or an error.

- $[\![a[e_1] = e_2]\!](\gamma) = \begin{cases} \gamma[a([\![e_1]\!]_\gamma) \mapsto [\![e_2]\!]_\gamma] & 0 \leq [\![e_1]\!]_\gamma < \mathit{length}(a), [\![e_2]\!]_\gamma \in \mathit{dom}(a) \\ \mathit{error} & \text{otherwise} \end{cases}$
- $[\![\mathit{eff}_1;\ \mathit{eff}_2]\!](\gamma) = [\![\mathit{eff}_1]\!]([\![\mathit{eff}_2]\!](\gamma))$

where $\gamma[a(x) \mapsto y]$ is a valuation γ' such that γ' differs from γ only in value of $a(x)$.

A.2.3 Network

The semantics of a network is a transition system $[\![\mathcal{N}]\!] = (S, s_0, \longrightarrow)$, with a finite set of states S, an initial state s_0, and transition relation $\longrightarrow \subseteq S \times S$, where:

- $S = (Q^1 \times \ldots \times Q^n) \times (\Gamma \cup \mathit{error})$
- $s_0 = (q_0^1, \ldots, q_0^n, \gamma_0)$

We use the following notation: $\vec{q}$ is a vector of states $(q_1, \ldots, q_n)$, $\vec{q}[x/y]$ for a state vector $\vec{q}$ with state x substituted for y, $\gamma \in \Gamma$. There is a transition $(\vec{q}, \gamma) \longrightarrow (\vec{q'}, \gamma')$ iff $\gamma \neq \mathit{error}$ and one of the following conditions is satisfied:

1. There exists an edge $(q^1, g, \tau, \mathit{eff}, q^2) \in E_i$ such that:
   - $\vec{q}' = \vec{q}[q^2/q^1]$
   - $(\vec{q}, \gamma) \models g$
   - $\gamma' = [\![\mathit{eff}]\!](\gamma)$
2. There exists edges $(q_1^1, g_1, c!, \mathit{eff}_1, q_1^2) \in E_i$, $(q_2^1, g_2, c?, \mathit{eff}_2, q_2^2) \in E_j$ such that:
   - $i \neq j$
   - $\vec{q}' = \vec{q}[q_1^2/q_1^1, q_2^2/q_2^1]$
   - $(\vec{q}, \gamma) \models g_1 \land g_2$
   - $\gamma' = [\![\mathit{eff}_1]\!]([\![\mathit{eff}_2]\!](\gamma))$
3. There exists edges $(q_1^1, g_1, c!e_1, \mathit{eff}_1, q_1^2) \in E_i$, $(q_2^1, g_2, c?a[e_2], \mathit{eff}_2, q_2^2) \in E_j$ such that:
   - $i \neq j$
   - $\vec{q}' = \vec{q}[q_1^2/q_1^1, q_2^2/q_2^1]$
   - $(\vec{q}, \gamma) \models g_1 \land g_2$
   - $\gamma' = [\![\mathit{eff}_1]\!]([\![\mathit{eff}_2]\!]([\![a[e_2] = e_1]\!](\gamma)))$

B Quality of Experiments: Classification of Papers

In this appendix we provide a list of publication used in Section 1 to study quality of experimental work in model checking publications.

Publications with Experiments of Quality Q1

- R. Alur and B.-Y. Wang. Next heuristic for on-the-fly model checking. In *International Conference on Concurrency Theory*, volume 1664 of *LNCS*, pages 98–113. Springer, 1999.
- B. Bollig, M. Leucker, and M. Weber. Parallel model checking for the alternation free μ-calculus. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 543–558. Springer, 2001.
- Benedikt Bollig, Martin Leucker, and Michael Weber. Local parallel model checking for the alternation-free μ-calculus. In *Proceedings of the 9th International SPIN Workshop on Model checking of Software (SPIN '02)*. Springer-Verlag Inc., 2002.
- Sébastien Bornot, Rémi Morin, Peter Niebert, and Sarah Zennou. Black box unfolding with local first search. In *TACAS*, Lecture Notes in Computer Science, pages 386–400. Springer, 2002.
- Dragan Bosnacki. A light-weight algorithm for model checking with symmetry reduction and weak fairness. In *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 2003.
- L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed ltl model checking based on negative cycle detection. In *Proc. Foundations of Software Technology and Theoretical Computer Science (FST TCS 2001)*, volume 2245 of *LNCS*, pages 96–107. Springer, 2001.
- Stefan Edelkamp and Shahid Jabbar. Large-scale directed model checking ltl. In *SPIN*, volume 3925 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.
- J. C. Fernandez, M. Bozga, and L. Ghirvu. State space reduction based on live variables analysis. *Journal of Science of Computer Programming (SCP)*, 47(2-3):203–220, 2003.

- M. D. Jones and J. Sorber. Parallel search for ltl violations. *Software Tools for Technology Transfer (STTT)*, 7(1):31–42, 2005.

- Victor Khomenko and Maciej Koutny. Branching processes of high-level petri nets. In *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 458–472. Springer, 2003.

- Denis Lugiez, Peter Niebert, and Sarah Zennou. A partial order semantics approach to the clock explosion problem of timed automata. In *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 296–311. Springer, 2004.

- Hillel Miller and Shmuel Katz. Saving space by fully exploiting invisible transitions. In *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 336–347. Springer, 1996.

- Karsten Schmidt. Using petri net invariants in state space construction. In *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 473–488. Springer, 2003.

- Karsten Schmidt. Automated generation of a progress measure for the sweep-line method. In *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 192–204. Springer, 2004.

- A. Prasad Sistla and Patrice Godefroid. Symmetry and reduced symmetry in model checking. In *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 91–103, 2001.

- Scott D. Stoller, Leena Unnikrishnan, and Yanhong A. Liu. Efficient detection of global properties in distributed systems using partial-order methods. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 264–279. Springer, 2000.

- H. van der Schoot. Partial-order verification in spin can be more efficient. In *Proc. of SPIN Workshop*. Twente University, 1997.

- Frank Wallner. Model checking ltl using net unforldings. In *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 207–218. Springer, 1998.

Publications with Experiments of Quality Q2

- T. Basten and D. Bosnacki. Enhancing partial-order reduction via process clustering. In *Proc. of Automated Software Engineering*, 2001.
- G. Behrmann, P. Bouyer, E. Fleury, and Kim G. Larsen. Static guard analysis in timed automata verification. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2003)*, volume 2619 of *LNCS*, pages 254–277. Springer, 2003.
- Dragan Bosnacki, Natalia Ioustinova, and Natalia Sidorova. Using fairness to make abstractions work. In *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 198–215, 2004.
- E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(3):279–287, November 1999.
- Rance Cleaveland, Gerald Lüttgen, V. Natarajan, and Steve Sims. Priorities for modeling and verifying distributed systems. In *TACAS*, Lecture Notes in Computer Science, pages 278–297. Springer, 1996.
- C. Daws and S. Tripakis. Model-checking of real-time reachability properties using abstractions. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 313–329. Springer, 1998.
- Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of mcmillan's unfolding algorithm. In *TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 1996.
- C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of Principles of programming languages (POPL'05)*, pages 110–121. ACM Press, 2005.
- P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 266–280. Springer, 2002.

- Jaco Geldenhuys and Antti Valmari. A nearly memory-optimal data structure for sets and mappings. In *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2003.
- G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. of Protocol Specification, Testing, and Verification*, 1992.
- Gerard J. Holzmann. The engineering of a model checker: The gnu i-protocol case study revisited. In *SPIN*, pages 232–244, 1999.
- C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1–2):41–75, 1996.
- Radu Iosif. Symmetry reduction criteria for software model checking. In *SPIN*, volume 2318 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2002.
- Radu Iosif and Riccardo Sisto. Using garbage collection in model checking. In *SPIN*, Lecture Notes in Computer Science, pages 20–33. Springer, 2000.
- J.P. Krimm and L. Mounier. Compositional state space generation from Lotos programs. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1997)*, volume 1217 of *LNCS*, pages 239–258. Springer, 1997.
- R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static partial order reduction. In *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS 1998)*, volume 1384 of *LNCS*, pages 345 – 357. Springer, 1998.
- Ilkka Kokkarinen, Doron Peled, and Antti Valmari. Relaxed visibility enhances partial order reduction. In *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 328–339. Springer, 1997.
- Bengi Karaçali and Kuo-Chung Tai. Model checking based on simultaneous reachability analysis. In *SPIN*, pages 34–53, 2000.
- F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. of SPIN workshop*, volume 1680 of *LNCS*. Springer, 1999.
- Thomas Mailund and Michael Westergaard. Obtaining memory-efficient reachability graph representations using the sweep-line method. In *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2004.

- Atanas N. Parashkevov and Jay Yantchev. Space efficient reachability analysis through use of pseudo-root states. In *TACAS*, volume 1217 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 1997.
- Karsten Schmidt. Integrating low level symmetries into reachability analysis. In *TACAS*, Lecture Notes in Computer Science, pages 315–330. Springer, 2000.

Publications with Experiments of Quality Q3

- D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. In *Proc. of SPIN Workshop*, volume 1885 of *LNCS*, pages 1–19. Springer, 2000.
- Dragan Bosnacki and Gerard J. Holzmann. Improving spin's partial-order reduction for breadth-first search. In *SPIN*, volume 3639 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 2005.
- Tonglaga Bao and Mike Jones. Time-efficient model checking with magnetic disk. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 526–540. Springer, 2005.
- I. Černá and R. Pelánek. Distributed explicit fair cycle detection. In *Proc. SPIN workshop*, volume 2648 of *LNCS*, pages 49–73. Springer, 2003.
- Rubén Carvajal-Schiaffino, Giorgio Delzanno, and Giovanni Chiola. Combining structural and enumerative techniques for the validation of bounded petri nets. In *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 435–449, 2001.
- Claudio Demartini, Radu Iosif, and Riccardo Sisto. dspin: A dynamic extension of spin. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 261–276, London, UK, 1999. Springer-Verlag.
- Peter C. Dillinger and Panagiotis Manolios. Fast and accurate bitstate verification for spin. In *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 57–75, 2004.
- Sami Evangelista and Jean-François Pradat-Peyre. Memory efficient state space storage in explicit software model checking. In *SPIN*, volume 3639 of *Lecture Notes in Computer Science*, pages 43–57. Springer, 2005.

- J. Geldenhuys and P. J. A. de Villiers. Runtime efficient state compaction in SPIN. In *Proc. of SPIN Workshop*, volume 1680 of *LNCS*, pages 12–21. Springer, 1999.
- P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. of Computer Aided Verification (CAV 1992)*, volume 663 of *LNCS*, pages 178–191. Springer, 1992.
- Keijo Heljanko, Victor Khomenko, and Maciej Koutny. Parallelisation of the petri net unfolding algorithm. In *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 371–385. Springer, 2002.
- G. J. Holzmann. An analysis of bitstate hashing. In *Proc. of Protocol Specification, Testing, and Verification*, pages 301–314. Chapman & Hall, 1995.
- G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. of Formal Description Techniques VII*, pages 197–211. Chapman & Hall, Ltd., 1995.
- R. P . Kurshan, V. Levin, and H. Yenigün. Compressing transitions for model checking. In *Proc. of Computer Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, pages 569–581. Springer, 2002.
- Alberto Lluch-Lafuente, Stefan Edelkamp, and Stefan Leue. Partial order reduction in directed model checking. In *SPIN*, volume 2318 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2002.
- Kim G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proc. of Real-Time Systems Symposium (RTSS'97)*, pages 14–24. IEEE Computer Society Press, 1997.
- Fredrik Larsson, Paul Pettersson, and Wang Yi. On memory-block traversal problems in model-checking timed-systems. In *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 127–141. Springer, 2000.
- Madanlal Musuvathi and David L. Dill. An incremental heap canonicalization algorithm. In *SPIN*, volume 3639 of *Lecture Notes in Computer Science*, pages 28–42. Springer, 2005.
- Igor Melatti, Robert Palmer, Geoffrey Sawaya, Yu Yang, Robert M. Kirby, and Ganesh Gopalakrishnan. Parallel and distributed model checking in eddy. In

SPIN, volume 3925 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2006.

- Stephan Melzer and Stefan Römer. Deadlock checking using net unfoldings. In *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 352–363. Springer, 1997.
- Gordon J. Pace, Frédéric Lang, and Radu Mateescu. Calculating-confluence compositionally. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 446–459, 2003.
- U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In *Proc. of Computer Aided Verification (CAV 1997)*, volume 1254 of *LNCS*, pages 256–267. Springer, 1997.
- U. Stern and D. L. Dill. Using magnetic disk instead of main memory in the murphi verifier. In *Proc. of Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 172–183. Springer, 1998.
- Claus Schröter and Victor Khomenko. Parallel ltl-x model checking of high-level petri nets based on unfoldings. In *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 109–121. Springer, 2004.
- W. Visser. Memory efficient state storage in SPIN. In *Proc. of SPIN Workshop*, pages 21–35, 1996.

Publications with Experiments of Quality Q4

- G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and upper bounds in zone based abstractions of timed automata. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 312–326. Springer, 2004.
- Gerd Behrmann, Thomas Hune, and Frits W. Vaandrager. Distributing timed model checking - how the search order matters. In *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 216–231, 2000.
- G. Behrmann, K. G. Larsen, and R. Pelánek. To store or not to store. In *Proc. of Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*. Springer, 2003.

- S. Christensen, L.M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001.
- S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz. Application and experimental evaluation of state space reduction methods for deadlock analysis in ada. *ACM Trans. Softw. Eng. Methodol.*, 3(4):340–380, 1994.
- Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. In *SPIN*, volume 3925 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2006.
- Javier Esparza and Keijo Heljanko. Implementing ltl model checking with net unfoldings. In *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2001.
- Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed explicit model checking with hsf-spin. In *SPIN*, Lecture Notes in Computer Science, pages 57–79. Springer, 2001.
- Jaco Geldenhuys. State caching reconsidered. In *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 23–38, 2004.
- H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proc. SPIN Workshop*, volume 2057 of *LNCS*, pages 217–234. Springer, 2001.
- A. Groce and W. Visser. Heuristics for model checking java programs. *Software Tools for Technology Transfer (STTT)*, 6(4):260–276, 2004.
- G. J. Holzmann and A. Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer (STTT)*, 3(1):270–278, 1998.
- Sebastian Kupferschmid, Jörg Hoffmann, Henning Dierks, and Gerd Behrmann. Adapting an ai planning heuristic for directed model checking. In *SPIN*, volume 3925 of *Lecture Notes in Computer Science*, pages 35–52. Springer, 2006.

Publications with Experiments of Quality Q5

- S. Blom and J. van de Pol. State space reduction by proving confluence. In *Proc. of Computer Aided Verification (CAV 2002)*, number 2404
- G. J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proc. of SPIN Workshop*, 1997.
- G. D. Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli. Exploiting transition locality in automatic verification of finite state concurrent systems. *Software Tools for Technology Transfer (STTT)*, 6(4):320–341, 2004.