

Experimental Work in Explicit Model Checking

Radek Pelánek

2008

Context

- domain: **explicit finite state model checking**
- applications: asynchronous systems, particularly protocols
- tools: Spin, Murphi, CADP, mCRL2, DiVinE,
 Java Pathfinder, Helena, Zing, ...
- research: reduction techniques, heuristics, optimisations
- this work: experimental evaluation

Outline

- 1 BEEM — BEncmarks for Explicit Model checkers
 - motivation
 - realization, content
- 2 applications of BEEM:
 - properties of state spaces
 - evaluation of techniques
- 3 summary, outlook

BEEM: Modeling Language

- low-level modeling language
- communicating extended finite state machines (DVE language)
- simple and straightforward semantics
- easy to write own parser and state generator
- automatic translation to other modeling languages is possible
- BEEM contains also automatically generated Promela (mCRL models comming soon)

```
process phil_0 {
  state think, one, eat, finish;
  init think;
  trans
    think -> one {guard fork[0] == 0; effect fork[0] = 1; }
    one -> eat {guard fork[1] == 0; effect fork[1] = 1; }
    eat -> finish {effect fork[0] = 0; },
    finish -> think {effect fork[1] = 0; };
}
```

```

process Sender {
byte ab=0, n, i, counter;
state idle, next_frame, wait_ack, send, success, q_error, ret;
init idle;
trans
  idle -> next_frame {sync Sin?n; effect i = 1; },
  next_frame -> send {effect counter=0;},
  send -> wait_ack { guard i==1 && i==n; sync toK!(4+2+ab); },
  send -> wait_ack { guard i>1 && i==n; sync toK!(2+ab);},
  send -> wait_ack { guard i==1 && i<n; sync toK!(4+ab); },
  send -> wait_ack { guard i>1 && i<n; sync toK!ab; },
  wait_ack -> success { sync fromL?; effect ab = 1-ab;},
  wait_ack -> q_error {guard counter == 3; sync timeout?;},
  wait_ack -> send {guard counter < 3; sync timeout?; effect coun
  success -> next_frame { guard i<n; effect i = i +1; },
  success -> ret { guard i==n; sync Sout!1;},
  q_error -> ret { guard i < n; sync Sout!2; },
  q_error -> ret { guard i == n; sync Sout!3; },
  ret -> idle {sync shake!;};
}

```

BEEM: Models

- 57 parametrized models, 300 specified instances
- well-known academic examples, case studies
- different application areas: mutual exclusion algorithms, communication protocols, controllers, leader election algorithms, planning and scheduling, puzzles
- model source codes
- pointers to sources (research papers)
- models with errors
- correctness properties (reachability, LTL)

BEEM: Tool Support

- DVE models → The Distributed Verification Environment (DiVinE)
 - an extensible model checking environment
 - easy to implement own model checking algorithms and perform experiments
- Promela models → SPIN
- (mCRL models)

BEEM: Web Portal

`http://anna.fi.muni.cz/models`

- presentation of models
- information about state spaces of models
- verification results
- selection of instances
- instance generator

Communication protocols

Name	Description	Size
brp	Bounded retransmission protocol	38
brp2	Bounded retransmission protocol (with timing)	72
cambridge	Cambridge ring protocol	46
collision	Collision avoidance protocol	20
firewire_link	Layer link protocol of the IEEE-1394	117
iprotocol	Optimized sliding window protocol	44
pgm_protocol	Pragmatic General Multicast (PGM) protocol	83
protocols	Simple communication protocols	36
rether	Real-time Ethernet protocol	24

Mutual exclusion algorithms

Name	Description	Size
anderson	Andersons queue lock mutual exclusion algorithm	5
at	Alur-Taubenfeld mutual exclusion algorithm	12
bakery	Bakery mutual exclusion algorithm	8
driving_phils	Mutual exclusion of processes accessing several resources	50

Model: peterson

Basic information

Source
code

[peterson.mdve](#) (author of the DVE model: Radek)

Short
description

Peterson's mutual exclusion protocol for N processes

Long
description

Situations, where two or more processes are reading and/or writing some shared data and the final result depends on who runs precisely when, are called race conditions. Code sections containing race conditions can be regarded as "critical", because such code can lead to inconsistent data. To avoid inconsistency in critical sections, exclusive access to shared data must be granted. This is called mutual exclusion, because if two processes compete for access then they have to exclude each other mutually. See also other mutex examples.

Source

Classical example

Parameters

N Number of processes
ERROR Presence of an (artificial) error (0/1)

Properties to
check

reachability Violation of mutual exclusion (more than one process in critical section)
LTL If P₀ waits for CS then it will eventually get there.
LTL If P₀ isn't in CS then it will eventually reach it.
LTL Infinitely many times someone in critical section.

[show formulas](#)

Interesting Instances of the Model

Source	Parameter values	Model info	State space info
peterson.1.dve	N=3	Processes 3	
peterson.1.pm (*)		Locations 15	Reach. time (**) 0.6s
		Transitions 21	States 12,498
		Variables 8	Error state No
		Channels 0	Deadlock No
		State size 18	Details
		Drawings of automata	
peterson.2.dve	N=3, ERROR=1	Processes 3	
peterson.2.pm (*)		Locations 15	Reach. time (**) 2.2s
		Transitions 21	States 124,704
		Variables 8	Error state No
		Channels 0	Deadlock No
		State size 18	Details
		Drawings of automata	
peterson.3.dve	N=3, ERROR=2	Processes 3	
peterson.3.pm (*)		Locations 15	Reach. time (**) 2.8s
		Transitions 21	States 170,156
		Variables 8	Error state No
		Channels 0	Deadlock No
		State size 18	BFS levels
		Drawings of automata	

Verification results

Number in parenthesis is the length of the shortest path to a reachable goal, respectively the length of some counterexample (not necessary the shortest one).

Instance	Property 1 (reach)	Property 2 (LTL)	Property 3 (LTL)	Property 4 (LTL)
<u>peterson.1</u>	<u>goal.1.1</u> no	<u>peterson.1.prop2.dve</u> no (19)	<u>peterson.1.prop3.dve</u> no (19)	<u>peterson.1.prop4.dve</u> yes
<u>peterson.2</u>	<u>goal.2.1</u> yes (23)	<u>peterson.2.prop2.dve</u> no (19)	<u>peterson.2.prop3.dve</u> no (19)	<u>peterson.2.prop4.dve</u> yes
<u>peterson.3</u>	<u>goal.3.1</u> yes (17)	<u>peterson.3.prop2.dve</u> no (19)	<u>peterson.3.prop3.dve</u> no (19)	<u>peterson.3.prop4.dve</u> no (19)
<u>peterson.4</u>	<u>goal.4.1</u> no	<u>peterson.4.prop2.dve</u> no (28)	<u>peterson.4.prop3.dve</u> no (28)	<u>peterson.4.prop4.dve</u> yes
<u>peterson.5</u>	<u>goal.5.1</u> yes (34)	<u>peterson.5.prop2.dve</u> unknown	<u>peterson.5.prop3.dve</u> unknown	<u>peterson.5.prop4.dve</u> unknown
<u>peterson.6</u>	<u>goal.6.1</u> yes (23)	<u>peterson.6.prop2.dve</u> unknown	<u>peterson.6.prop3.dve</u> unknown	<u>peterson.6.prop4.dve</u> unknown
<u>peterson.7</u>	<u>goal.7.1</u> unknown	<u>peterson.7.prop2.dve</u> unknown	<u>peterson.7.prop3.dve</u> unknown	<u>peterson.7.prop4.dve</u> unknown

- Sort:
- Alphabetically
 - By reachability time
- Show reachability problems:
- Reachable
 - Not reachable
 - Unknown
- Show LTL verification problems:
- Satisfied
 - Not satisfied (counterexample)
 - Unknown
- Restrict size:
- Larger than states
- Smaller than states
- Only instances with at least one verification problem of a selected type:
- yes
 - no
- Output:
- HTML table
 - Plain list of instances

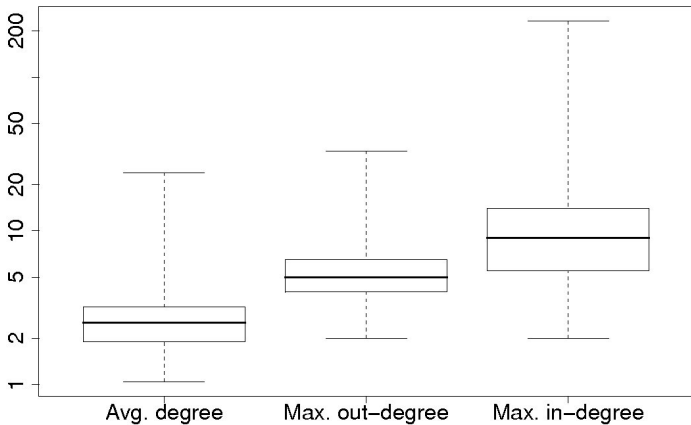
Applications of BEEM

- properties of state spaces
- evaluation of techniques
 - error detection techniques
 - state caching and state compression techniques

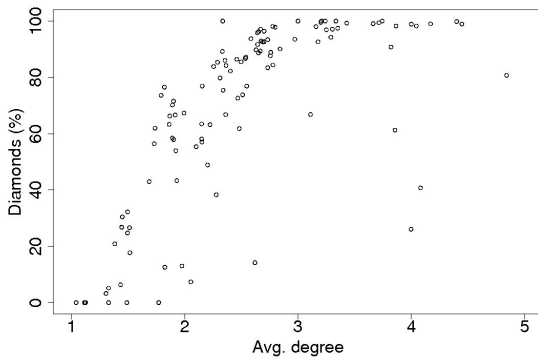
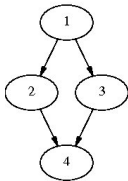
Properties of State Spaces: Motivation

- usual approach: state space = arbitrary directed graph
- but: short description \Rightarrow special class of directed graphs
- what are the common properties of state spaces?
- how can we exploit them?

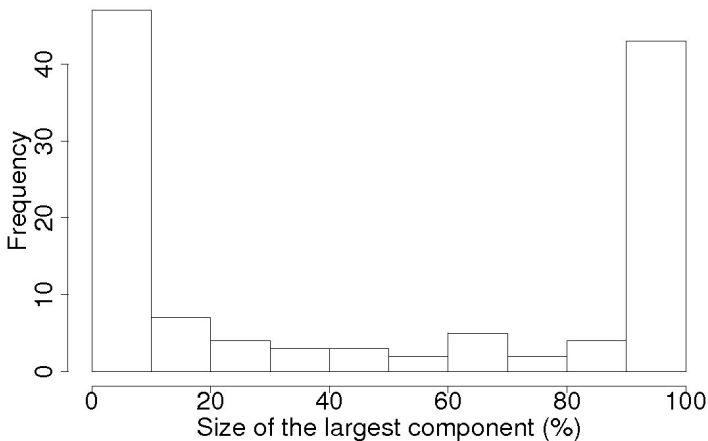
Degrees



Average Degree and Diamonds



Strongly Connected Components



Error Detection Techniques

```

proc ErrorDetection( $M, \varphi$ )
  insert initial state to Wait
  while not finished do
    get  $s$  from Wait
    if  $s$  violates  $\varphi$  then return path to  $s$  fi
    foreach  $s' \in$  selected successors of  $s$  do
      if  $s'$  not matched in Visited
        then insert  $s'$  to Wait
        update Visited with information about  $s'$  fi
    od od
end

```


Error Detection Techniques

- BFS** breadth-first search
- DFS** depth-first search
- RDFS** randomized DFS
- RW** random walk
- ERW** enhanced random walk
- BITH** bitstate hashing with repetition
- DIRS** directed search with structural heuristic
- DIRG** search directed by heuristic function given by the goal
- UPOR** under-approximation refinement based on partial order reduction

Performance Classification

- N_T = the number of states processed by a technique T
- N_B = the number of states processed by the best technique for a given model

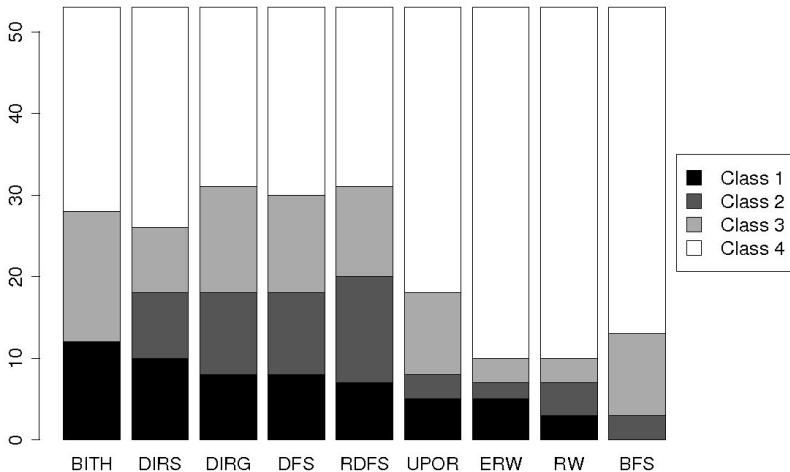
$$\text{Class 1} \quad N_B = N_T$$

$$\text{Class 2} \quad N_B < N_T \leq 2 \cdot N_B$$

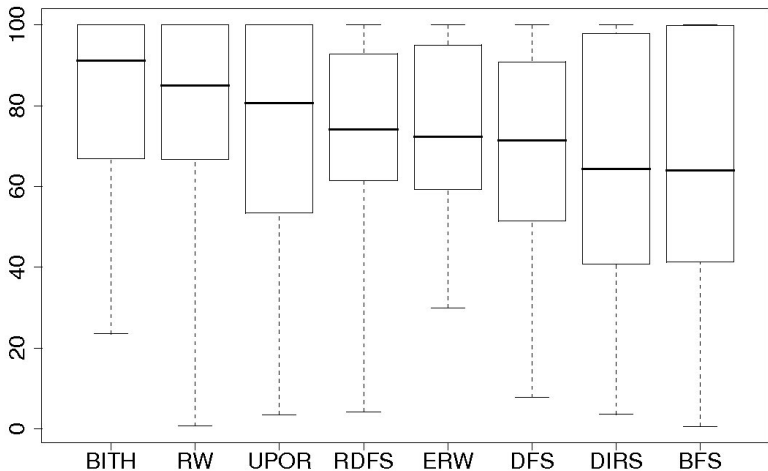
$$\text{Class 3} \quad 2 \cdot N_B < N_T \leq 10 \cdot N_B$$

$$\text{Class 4} \quad 10 \cdot N_B < N_T$$

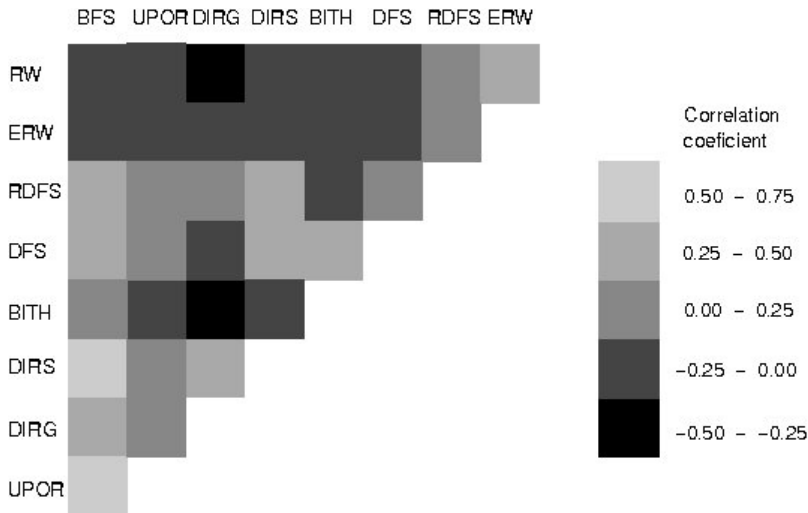
Comparison of Techniques: Number of Visits



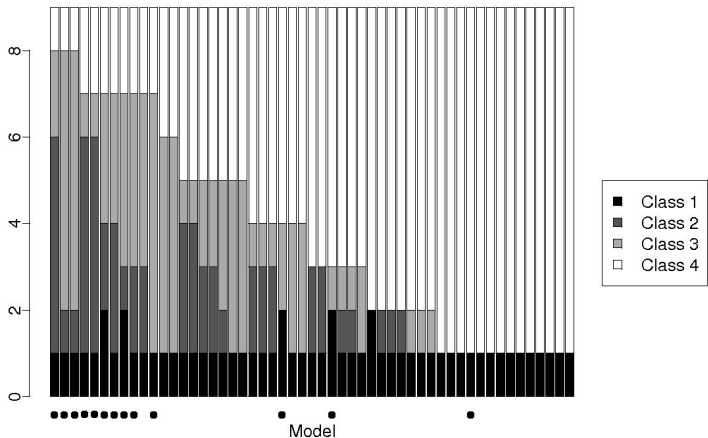
Comparison of Techniques: Condition Coverage



Correlations Among Techniques



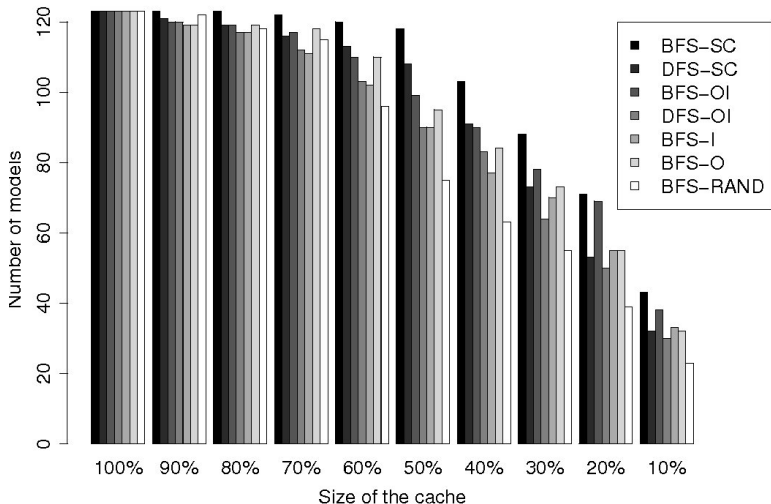
Impact of Model



Error Detection Techniques: Summary

- There is no single best technique. Never mind, it does not matter.
- It is important to focus also on complementarity of techniques, not just on their perfectness.
- It is important to compare a new technique with a large number of previously known techniques.
- Both models and goals have significant and hard to predict impact on the performance of techniques.
- *Use several simple techniques, run them in parallel.*

State Caching: Results



Summary: Benchmarks

- experimental standards in model checking are rather low (small number of models, toy models)
- poor experiments can distort results
 - impact of model selection
 - toy vs complex models
- we need benchmarks
- BEEM — proposal of benchmarks for explicit model checkers

Summary: Evaluation of Techniques

- simple techniques achieve similar performance as sophisticated ones
- each technique works on different models; it is difficult to estimate the performance in advance
- *Use several simple techniques, run them in parallel.*

Outlook: EMMA

Explicit Model checking MAnager

- runs several techniques in parallel
- collects (intermediate) results, makes adjustments according to them
- uses *recipe* to decide:
 - selection and order of techniques to be executed
 - parameter values to be used
- recipe
 - provided by human
 - learnt by some machine learning algorithm