

Efficiently Verifying Safety Properties with Idle Office Computers

Marko Mäkelä

Laboratory for Theoretical Computer Science

Helsinki University of Technology

P.O.Box 9205

02015 HUT

Finland

June 24, 2002

Techniques for Verifying Software Systems

- Code inspection and reviews
- Proof techniques
- Simulation and testing: checking some selected behaviours
- Exhaustive state space enumeration: covering all reachable states and behaviours

Symbolic methods represent the set of explored states as a Boolean function

Explicit methods represent each reachable state separately as a bit string

Some Desired Properties of Software Systems

Liveness and fairness express eventualities and require progress:

- a sent message will eventually be received, provided that the channel works. . .
- a process trying to reach a critical section will eventually enter it, provided that. . .

Safety properties require that nothing bad ever happens:

- the system does not lock up (deadlock) unexpectedly
- undefined computations do not occur ($\frac{x}{y}$ with $y = 0$); assertions do not fail
- at most one process is in a critical section at any given time

Prerequisites for Automated Software Verification

- Simple procedures: the formalism should be hidden “under the hood” of the toolset
- Automated translation of the software and its requirements to a model and properties of the verification tool: the model and the formal specification are always up to date
- The modelling formalism should support high-level operations and data types to allow a simple translation of the model and easy interpretation of error traces

All these requirements can be fulfilled by *explicit methods* for exhaustive state space enumeration. *Safety properties* are much easier to formulate than liveness and fairness.

Marko Mäkelä

The Bottlenecks of Explicit Methods: Time and Space

Explicit state space enumeration requires

time to explore all reachable states of the system and

space to store

- the set of states encountered so far,
- a queue or stack of unexplored states, and
- information needed for recovering error traces.

Trading Time for Space in Explicit Methods

The space consumption can be reduced by compressing the set of states tightly. The queue and the error information could be represented with pointers to the data in the set. Alternatively, the set of states can be “stored” lossily by applying hashing techniques.

Also, it is possible to trade time for space, to reduce the set of reachable states:

- partial order reductions: omit uninteresting interleavings of processes
- symmetry reductions: store only *canonic representatives* of states
- path compression, omission of invisible states, state space caching: the “forgotten” states may be revisited several times, from several directions

Saving Time in Explicit Methods: Parallel Processing

Personal computers are idle most of the time, serving only interactive users. A *virtual supercomputer* could be built if the computing application meets some requirements:

small memory and file system usage at the processing nodes: interactive users will be disturbed if the disk rattles constantly \Rightarrow use a centralised “state space server”

dynamic load balancing: older computers may be much slower than newer ones, and the processors may at times be occupied by something else

fault tolerance: if a (laptop) computer is shut down or disconnected from the network without prior notice, its computations should be taken over by the remaining nodes

efficient communications overlapped with processing; low $\frac{\text{Mb/s}}{\text{MIPS}}$ ratio

The Exhaustive State Space Enumeration Algorithm

Sequential Version

```

S := ∅; Q.insert(s0)
while (s := Q.remove()) ≠ NIL do
  for all s' ∈ successors(s) do
    if s' ∉ S then
      if error(s') then report(s')
      S := S ∪ {s'}; Q.insert(s')

```

Server Initialisation and Procedures

```

remote procedure getState():
  return Q.remove() // wait until Q ≠ ∅
  // return NIL if everybody waits here

```

Client Processes

```

while (s := getState()) ≠ NIL do
  S' := ∅
  for all s' ∈ successors(s) do
    if error(s') then report(s')
    else S' := S' ∪ {s'}
  putStates(S')

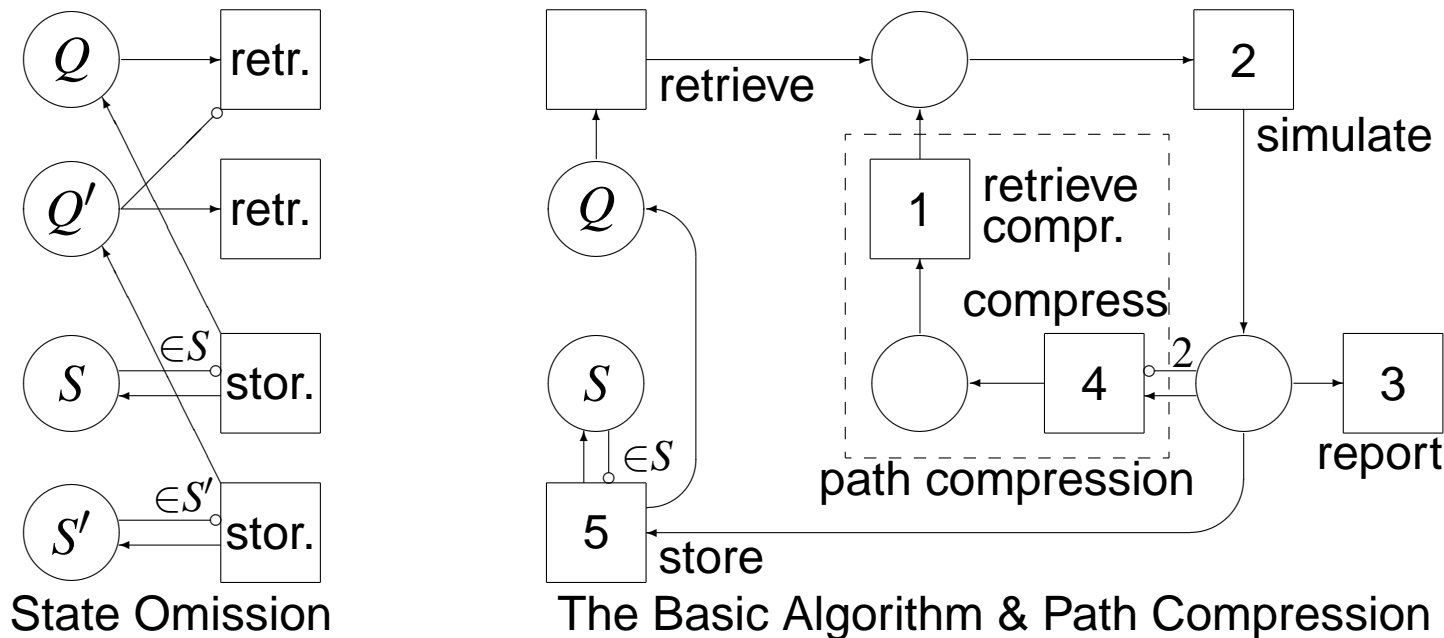
```

```

S := ∅; Q.insert(s0)
remote procedure putStates(S'):
  for all s' ∈ S' \ S do
    S := S ∪ {s'}; Q.insert(s')

```

A Petri-Net-Like Visualisation of the Safety Checker Algorithm

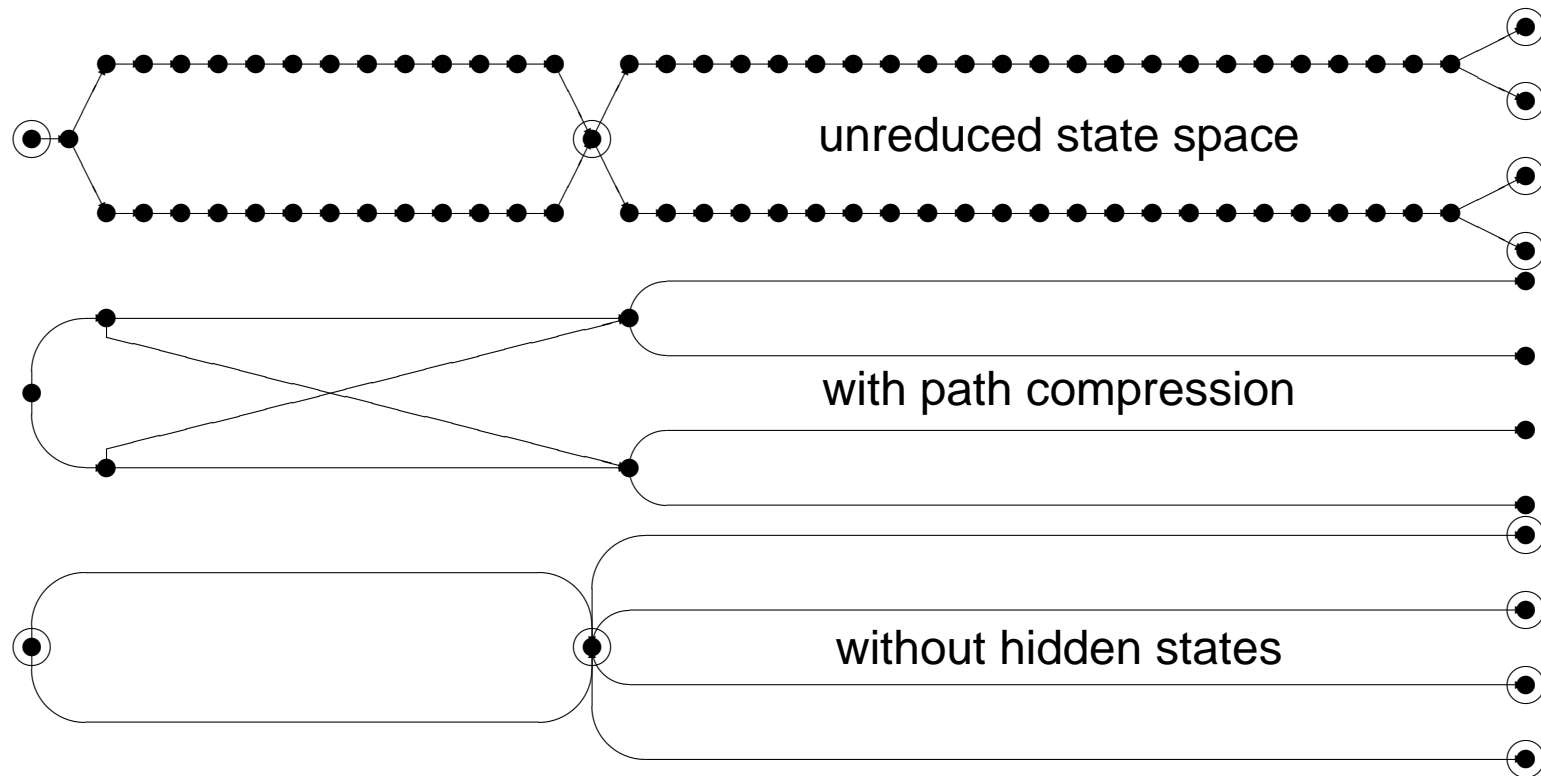


In our parallel algorithm, the actions “store” and “retrieve” are distributed via remote procedures, implemented with message passing. Each client has a local queue of unprocessed states that is supplied from the server’s global queue. The set of reachable states is maintained by the server only. State space reductions are applied locally at the clients.

Marko Mäkelä

State Space Reductions

Safety properties are checked in all reachable states. Reducing the state space only means that the omitted states can be revisited several times, slowing down the analysis.



Generating Error Traces

Breadth first search is bound to find the shortest path from a set of graph nodes (the initial states) to a specified type of node (an erroneous state).

It is essential for usability that the model checking tool does not merely display an erroneous state but also presents a sample execution leading to the error. For safety property violations, the tool only needs to list *one possible path* leading to the error.

To this end, the tool must maintain a mapping from each reachable state to one of its ancestors. When an erroneous state is found, this mapping is applied transitively until a reverse path from the error to an initial state is completed.

The resulting chain of states $s_0s_1s_2 \dots s_n$ still does not show the transitions between the states. They can be derived by re-simulating the model to find one action $s_0 \xrightarrow{t_0} s_1$, an action $s_1 \xrightarrow{t_1} s_2$, etc. In a reduced state space, each t_i may be a sequence of actions.

Marko Mäkelä

Generating Error Traces in Distributed Model Checking

Assuming that a tiny fraction of the reachable states are erroneous and that one error is eliminated at a time, it is acceptable to stop the search after showing the first error trace.

Furthermore, assuming that error traces are relatively short, at most thousands of steps, they can be produced on a single processor—the server. In this way, the model checking process is controlled via a single processor.

In MARIA, the server distributes unprocessed states evenly to the clients. Each state is tagged with an offset to an append-only tree-like *predecessor map file* with pairs of compressed state vectors and file offsets (to predecessor states).

When a client reports an erroneous state or the successors of a state, it also reports the file offset of the state. When an error in state s_n is reported, the server performs random access to this file to retrieve the compressed state vectors $s_0s_1s_2 \dots s_n$.

Marko Mäkelä

An Implementation on TCP/IP Stream Sockets (1/3)

- each client has one bidirectional socket to the server
 - clients send strings **putStates**(S') or **report**(s')
 - server sends unprocessed state vectors (clients do not send **getState**() strings)
 - clients initiate the dialog by sending **putStates**($\{s_0\}$)
- non-blocking I/O with **select**(2) on the server
 - whenever there are u unprocessed states and n clients, and a client having less than $\frac{u}{n}$ unprocessed states can accept data, some states are sent to it
 - client requests are read whenever possible; incomplete packets will be buffered

An Implementation on TCP/IP Stream Sockets (2/3)

- non-blocking I/O with **select(2)** or **poll(2)** on the clients
 - a **putStates()** call will poll the incoming queue of unprocessed states
 - writes will not block unless **reject()** is called or all states have been processed
- simple error handling and **fault tolerance** by utilising the built-in socket mechanisms
 - if a client is abnormally terminated or disconnected from the network, the server will redistribute its unprocessed states upon receiving a socket error
 - the server can exclude clients or terminate the analysis by closing sockets

An Implementation on TCP/IP Stream Sockets (3/3)

dynamic load balancing: the unprocessed states are evenly distributed to all clients

- clients can join and leave at any time
- with bad luck, a model with few branches leaves the fastest clients unemployed

termination detection: the server terminates when there are no unprocessed states and all clients have sent **putStates()** messages for all enqueued states

simple invocation with ordinary shell commands

- the server is started first, then the clients (single command on a multiprocessor)
- a custom job-launching protocol would be essential in the Windows environment

Experimental Results with MARIA: The RLC Protocol

Processor Type		Time						
AMD Athlon, 1 GHz		100 s	210 s					
Intel Pentium III, 450 MHz		183 s	361 s					
Intel Pentium II, 266 MHz		304 s	591 s					
MIPS R12000, 300 MHz		258 s	517 s					

Computer Type	Wall-Clock Time						
N.o. Clients (<i>n</i>):	1	2	3	4	5	6	7
2 Athlon + 2 P III + 3 P II (Linux TCP/IP)	105 s 95 %	54 s 93 %	42 s 94 %	35 s 92 %	32 s 91 %	29 s 92 %	28 s 88 %
<i>n</i> · MIPS R12000 (UNIX domain sockets)	261 s 99 %	132 s 98 %	89 s 97 %	70 s 92 %	59 s 87 %	53 s 81 %	46 s 80 %

Experimental Results with MARIA: RLC & Path Compression

Path compression reduces the state space from 19,890 reachable states and 23,233 arcs to 5,236 arcs and 9,677 states. The workload is approximately doubled, as 45,238 successors will be considered. (See the last column of the first table on last slide.)

System	1	2	3	4	5	6	7	8	9	10	11	12
Linux	212 s	105 s	82 s	67 s	60 s	55 s	50 s	2 Athlon + 2 P III + 3 P II				
PCs	99 %	100 %	99 %	99 %	99 %	99 %	99 %	2 · 1 + 2 · 0.45 + 3 · 0.266 GHz				
SGI	520 s	257 s	174 s	130 s	105 s	88 s	76 s	67 s	60 s	56 s	51 s	47 s
Origin 2000	99 %	100 %	99 %	99 %	99 %	98 %	97 %	96 %	95 %	93 %	92 %	90 %

Due to the added per-state workload and the improved branching factor, more processors can be utilised. With path compression, 90 % of 12 processors are utilised. Without the reduction, 80 % of 7 processors explore the states in roughly the same wall-clock time.

Marko Mäkelä

Experimental Results with MARIA: Data Base Managers

The famous distributed data base manager model generates a large state space with only four high-level transitions. For $d = 8$ modelled data base nodes, the arcs to states ratio is $81,664/17,497 \approx 4.66$, and for $d = 9$, it improves to $314,946/59,050 \approx 5.33$.

<i>d</i>	Seq.	Distributed to <i>n</i> Clients									
	<i>n</i> =1	2	3	4	5	6	7	8	9	10	
8	4.85 s	6.14 s	3.17 s	2.09 s	1.48 s	1.16 s	0.94 s	0.83 s	0.72 s	0.70 s	0.70 s
		79 %	76 %	77 %	82 %	84 %	86 %	83 %	84 %	77 %	69 %
9	20.82 s	25.02 s	12.60 s	8.39 s	6.03 s	4.72 s	3.79 s	3.26 s	2.87 s	2.54 s	2.33 s
		83 %	83 %	83 %	86 %	88 %	92 %	91 %	91 %	91 %	89 %

Although MARIA computes successor states of this model very fast, the server process does not become a performance bottleneck for these numbers of clients.

Marko Mäkelä

Conclusion and Future Work

- A supercomputer is easily outperformed by an office computer network (a Silicon Graphics Origin 2000 with 128 MIPS R12000 processors is 50–60 1 GHz PCs)
- With TCP/IP sockets, computations can be distributed easily (grid computing)
 - + simple management and termination detection; dynamic load balancing
 - single point of failure; possible scalability limitations
 - + scales well in practice, even for simple models
 - + scalability improves with complex models and state space reductions
- The liveness model checker in MARIA could make use of this algorithm
- A SETI@Home-style job launcher should be developed for Windows PCs