Efficiently Verifying Safety Properties with Idle Office Computers

Marko Mäkelä[*]

Laboratory for Theoretical Computer Science
Helsinki University of Technology
Espoo, Finland,
Email: `marko.makela@hut.fi`

Abstract

Assuring the quality of safety-critical software systems requires more rigorous methods than testing. Model checking by exhaustive state space enumeration, "testing all executions," is an alternative, but the use of state and memory reduction techniques makes runtime a major limiting factor. We describe a simple parallel version of a state space enumeration algorithm that utilises the unused computing power of office workstations while not congesting their memories. In an experiment with a complex data link protocol, our implementation of the algorithm achieves close to linear speedups on a heterogeneous network of workstations.

Keywords: model checking, distributed algorithm, state space enumeration

1 Introduction

Complex protocols or distributed software systems are often verified by examining all reachable global states of a system from a specified set of initial states. Depending on the way in which the system has been described and on the type of properties that are to be verified, different methods can be applied. This article focuses on verifying *safety* properties, checking if the system can reach a "bad" state or if a finite execution of the system violates a desired property.

Search algorithms for verifying safety properties are built around a data structure that represents a set of encountered states. When the search starts, the set corresponds to the initial states. At the end, the set contains all the reachable states of the system.

Symbolic methods represent the set of reachable states with a modifiable Boolean function that maps assignments of state variables to truth values. A constant function of "false" represents the empty set. When the set grows, the function is modified to map the state variable assignments representing the added states to "true". As adding states may make the function independent of some variables, the representation of the function can shrink even though the set grows. The biggest disadvantage of symbolic methods is that their peak memory usage greatly depends on how the system has been specified and partitioned (Ciardo, Lüttgen & Siminiceanu 2000, Table 3). Also, it is difficult to apply symbolic methods to complex data types, which are essential in software systems.

[*]This research was financed by the Helsinki Graduate School on Computer Science and Engineering, by Jenny and Antti Wihuri Fund, and by Academy of Finland (Project 47754).

In *exhaustive state space enumeration*, the reachable states of a model are enumerated by evaluating all possible successor states of the known states of the model. In the beginning of the search, only the initial state is known. At the end, all reachable states of the model are known. To ensure that the search completes, the successors of each state should be evaluated only once. One way of ensuring this is to store each state in a table that grows during the search. The total memory usage of the table is the product of the number of reachable states and the average length of the state vectors, plus the bookkeeping overhead. In order to speed up state look-ups, a mapping of hash signatures of states to indices in the table can be maintained.

Stern and Dill (Stern & Dill 1997) mention three changes that allow more complex models to be analysed via state space enumeration. First, the state space can be reduced while ensuring that errors will be detected. Second, the state set can be approximated with hashing techniques, potentially causing errors to go undetected. Third, the exploration time can be reduced by applying parallel processing.

This paper describes a scalable safety property checker that is based on exhaustive state space enumeration. The search can be distributed both on multiprocessor systems and on networks of ordinary workstations. Our implementation of the algorithm uses standard programming interfaces (IEEE 2001) and performs notably well on a heterogeneous network of office workstations.

Our benchmark for the safety checker is the radio link control (RLC) protocol of the third-generation mobile telephone system. A verbatim translation of the protocol specification (ETSI 2000) into an algebraic system net (Kindler & Völzer 2001) in the modelling language of Maria (Mäkelä 2002) comprises over 20,000 lines of text. Displaying a state vector of the model in readable form to the user requires around 8,000 characters or nearly 500 lines of text. The internal storage requires 170–200 bytes. All variations of the model have more than 700 transitions and 100 places. It is practically impossible to unfold this high-level model to a place/transition system, since the protocol makes heavy use of complex data types, such as queues of unions of large structures. Also, it is impractical to tell Maria to compile the model to machine executable code in order to speed up analysis, as a compilation run usually lasts over half an hour. If there is a trivial modelling error that the high-level net interpreter of Maria finds in a minute, compiled code will find it it in a few seconds, but only after the compiler has completed its job.

The analysis of the RLC protocol (Tynjälä, Leppänen & Luukkala 2002) concentrates on safety properties, as the Maria liveness property checker (Latvala 2001) is practically limited to systems having at most some millions of reachable states, while the RLC protocol model can reach tens of millions of states, depending on the configuration. One source of capacity problems are the edges of the reachability graph, which are needed for checking liveness properties. Safety checking does not need to know about the edges until an erroneous state is found and a path to the error needs to be reproduced. Even then, it is sufficient to have access to a "spanning tree" of the reachability graph rather than all its edges. Also, it is unnecessary to store the names and parameters of the transitions leading from a state to another, as these can be recomputed if an error is found.

1.1 Related Work

The idea of speeding up explicit state space exploration by utilising multiple processors is not new. Parallel processing can be applied at different levels of the exploration algorithm.

TrailBlazer (Holzmann & Smith 2001) distributes the verification procedure at the highest possible level by invoking several independent runs of the protocol verifier Spin on dedicated computing servers. This is very efficient when checking multiple models or properties are at the same time, since the only communications between the front-end computer and the computing servers take place when verification processes are started or completed. However, the approach does not speed up the analysis of a single property in a single model.

Lorentsen and Kristensen (Lorentsen & Kristensen 2001) experimented with a state reduction method based on symmetries. Each state is mapped to a canonical representative. This processor intensive subtask of the state space exploration algorithm was distributed to a set of computing servers. According to (Lorentsen & Kristensen 2001, Table 2), this distributed algorithm was able to utilise up to 4 or 5 slave processors at 80 % efficiency. The performance of this method depends heavily on the model being analysed: in particular, if symmetry reductions cannot be applied, only one processor will be utilised.

Parallel Murφ, a distributed version of the Murφ model checker for a guarded-command language, was implemented on a dedicated computing cluster. The algorithm (Stern & Dill 1997) is symmetric; the set of explored states is distributed among the nodes. Each node "owns" a part of the state space. The owner of a state is determined by computing a hash signature of the state and mapping it to a node identifier. In the reported experiment (Stern & Dill 1997, Table 2), 84 % of the computing power of the 32-processor cluster was utilised.

We take a different approach. Instead of employing expensive supercomputers or dedicated computing clusters, we use the idle processing time of regular office computers. Normal interactive use of such computers is not affected by low-priority background processes, as long as they do not consume much memory. When the state collections—the explored and the unprocessed states—are maintained on a central server, the worker processes need little memory. In our experiments, up to 12 processors were utilised at over 90 % efficiency.

1.2 Outline

The rest of this paper is organised as follows. Section 2 presents a basic algorithm for checking safety properties by explicit state space enumeration. Parallel versions of the algorithm are discussed in Section 3. Section 4 presents experimental results for our benchmark model, and Section 5 gives some concluding remarks.

2 Explicit State Space Enumeration

2.1 A Basic Algorithm

Algorithm 1 is the basic procedure for checking safety properties. It has two parameters: the initial state s_0 of the system, and the state transformation rules, a relation that maps a given state s to the set $\mathsf{successors}(s)$ of its successor states. The function $\mathsf{error}(s)$ identifies erroneous states.

Algorithm 1 *Verify a safety property by explicit state space enumeration.*

```
$S := \emptyset$        // empty set of processed states
$Q := \langle s_0 \rangle$   // buffer of unprocessed states
while $Q \neq \langle\rangle$ do
  $s := Q.\mathsf{remove}()$
  for all $s' \in \mathsf{successors}(s)$ do
    if $s' \notin S$ then
      if $\mathsf{error}(s')$ then show trace from $s_0$ to $s'$
      $S := S \cup \{s'\}; Q.\mathsf{insert}(s')$
```

The search algorithm makes use of two data structures: the set of states it has explored, and a collection of states that are waiting to be processed. It must be possible to insert items into the set and to check if the set contains a given item. The collection must support insertion and removal of items, and emptiness check. If the collection is a queue, then the search proceeds *breadth first*, guaranteeing that a shortest path to an error is found first.

This basic algorithm can be varied in several ways. There can be several initial states. The search can terminate at the first encountered error, or it can display an execution trace from the initial state to each faulty state, or all execution traces to each faulty state.

The implementation of this algorithm in Maria operates on two kinds of models: algebraic system nets as such, or synchronised with a finite state automaton that corresponds to a safety property expressed in a subset of linear temporal logic. There are four kinds of erroneous states:

1. states that satisfy a "reject" or "deadlock" formula,
2. accepting product states of a system and a property,
3. states that cannot be compacted due to a violation of a capacity constraint or a proposed invariant, and
4. states that cannot be computed due to an evaluation error, such as arithmetic overflow or buffer underflow.

2.2 Probabilistic Storage

In probabilistic verification techniques, the set of explored states is approximated by not storing the state vectors themselves, but by computing and storing short hash signatures of them. In this way, the set can be "stored" in much smaller amount of memory.

The disadvantage of probabilistic methods is that new states can be mistaken for explored ones, meaning that entire branches of the state space can remain uncovered. The probability of missing erroneous states can be reduced by repeating the search with a different hash signature function.

Unfortunately, representing the state set in less space does not shrink the list of unprocessed states. In the worst case, the list may need to hold all reachable states of the system simultaneously. To avoid memory shortages, Spin has a "stack cycling" option (Holzmann 1999) for keeping portions of the list on disk.

2.3 Explicit Storage

Algorithm 1 will explore all reachable states of the model if all membership tests on the state set succeed. One way of guaranteeing this is to represent the set as a table. Membership tests can be sped up by maintaining a mapping from hash signatures of state vectors to indices into the table.

The lossless state set storage in Maria maintains the state table and the hash map in two disk files. The state table is a string of compacted state vectors, which are sequences of bytes. The map is a B-tree from hash signatures to offsets in the state table file.

As the look-up structure and the state table are typically accessed in random order, it would be better to keep them in main memory rather than on disk. However, Maria supports memory mapping (IEEE 2001, Section 2.8.3.2) of random access files. The overhead is negligible compared to dynamically allocated memory, as no system calls are made unless the file needs to be grown. To reduce the amount of these calls, Maria always doubles the allocated file size when more space is needed.

When Maria uses this type of state set storage, it presents the collection Q of unprocessed states in less memory by not storing state vectors but offsets into the state table.

2.4 Error Trace Generation

Typically, verification tools are used in order to prove that a system is free of errors—that it behaves according to its specification. It is often adequate to stop the analysis when the first error is found. The work of a verification tool is only half done when an erroneous state is found. If there are millions of reachable states, merely displaying the erroneous state to the user is as frustrating as reporting "there is an error in your system, but I won't tell you how to get there." A *counterexample trace* is a sequence of model actions that leads from the initial state to the error.

The information needed for producing error traces should be stored in as little space as possible, so that most of the available memory can be used for accommodating the set of encountered states and the collection of unprocessed states. Not all information needs to be stored in advance—when an error is detected, the omitted information can be recomputed. This allows the verification of larger systems and more complex properties.

Efficient production of a counterexample trace, a shortest execution to an erroneous state,[1] requires a function that maps each state s' in the trace to the state from which s' was obtained in Algorithm 1: $s = \mathsf{ancestor}(s')$, such that $s' \in \mathsf{successor}(s)$. All states on the counterexample trace can be enumerated by repetitively applying this function on the ancestor s of the error state s' until the initial state s_0 is reached.

Algorithm 1 does not maintain such a function explicitly. When the unprocessed states Q are arranged as a stack and the operation $s := Q.\mathsf{remove}()$ is split into two, so that s is read from Q before entering the inner loop and removed from Q after the loop, the contents of the stack Q almost supplies a counterexample trace from s_0 on its bottom to s on its top. If a state on the explored path from s_0 to s' has multiple successors, Q may contain states that do not belong to the counterexample trace. These states can be identified by tagging each state in Q with its distance from s_0.

Unfortunately, using a search stack (depth-first search) may produce unnecessarily long counterexample traces. Breadth-first search, implemented by arranging Q as a queue, produces shortest paths, but the function $\mathsf{ancestor}(s)$ cannot be computed with the information that is present in the queue. Stern and Dill (Stern & Dill 1997) propose an addition to the algorithm: whenever a state s' is inserted into Q, it is also appended to an auxiliary file together with the position of its ancestor s in the said file. The collection Q must associate each unprocessed state with these file positions. This auxiliary file provides the mapping $\mathsf{ancestor}(s)$ for producing error traces.

Maria implements the above mentioned scheme. It applies a variable-length code to reduce the size of the file. Small offsets are represented with one byte, bigger ones with two bytes, and so on. Furthermore, when the state set is stored as a table (Section 2.3), the trail file will contain offsets to the table instead of state vectors.

Writing the counterexample recovery information to a file does not significantly affect the performance, since sequential file access is fast. Only when a counterexample trace is produced, random access is needed. Even at that point, the input/output overhead may be insignificant, if discovering the actions that lead from one state to another in the trace takes a long time.

[1]The shortest path to the error may not be unique. It is often practical to display only one trace to the user if an error is found.

3 Parallel Processing

An execution time profiler is a good tool in alleviating performance bottlenecks. It points out those parts of the program code where most of the execution time is spent. We have profiled a run of safety property analysis of the RLC protocol model (Tynjälä et al. 2002). Well over 90 percent of the time is spent in the model interpreter that computes compacted successor states of a state.

Normally, one would compile the model to executable code that bypasses the interpreter, but it is not reasonable in this case, as the 500 kilobytes of high-level Maria modelling language are translated to almost 800,000 lines or 20 megabytes of low-level C program code. Especially when the model is under development, it would be frustrating to wait thirty minutes for the compilation to find a trivial error that the interpreter would have found instantly.

3.1 A Client–Server Algorithm

Algorithm 2 *Client process for verifying a safety property by exhaustive state space enumeration.*

```
do until server terminates
    $s := \mathsf{getState}()$                         // remote procedure
    $S' := \emptyset$
    for all $s' \in \mathsf{successors}(s)$ do
        if $\mathsf{error}(s')$ then $\mathsf{report}(s')$    // remote procedure
        else $S' := S' \cup \{s'\}$
    $\mathsf{putStates}(S')$                          // remote procedure
```

Algorithm 3 *Server process for verifying a safety property by exhaustive state space enumeration.*

```
$S := \emptyset$            // empty set of processed states
$Q := \langle s_0 \rangle$         // buffer of unprocessed states
do until all clients wait for $\mathsf{getState}()$ to finish
    serve client requests

remote procedure $\mathsf{getState}()$:
    return $Q.\mathsf{remove}()$// atomically wait until $Q \neq \langle\rangle$

remote procedure $\mathsf{report}(s')$:
    show trace from $s_0$ to $s'$

remote procedure $\mathsf{putStates}(S')$:
    for all $s' \in S' \setminus S$ do
        $S := S \cup \{s'\}; Q.\mathsf{insert}(s')$
```

Our distributed version of Algorithm 1 is presented in Algorithms 2 and 3. The message exchange between clients and the server is abstracted as remote procedure invocations.

The implementation of this distributed algorithm in Maria uses two variants of stream-oriented POSIX sockets (IEEE 2001, Section 2.10). Local domain sockets are used on multi-processor systems, while TCP/IP sockets are applied when clients are executed on multiple computers.

The remote procedure calls are an abstraction of our implementation, where the server has one input and one output buffer per client. When a client invokes $\mathsf{report}(s')$ or $\mathsf{putStates}(S')$, it sends a byte sequence to the server and does not await any response. When the client invokes $\mathsf{getState}()$, it does not send anything, but it waits for a byte sequence from the server. To avoid latency problems, the client has an input queue for $\mathsf{getState}()$ data, and the server tries to keep its queue Q of unprocessed states empty by distributing its contents evenly to the clients. The server uses entirely non-blocking input and output; a client may block when it runs out of states or when it flushes its output buffer when invoking $\mathsf{report}(s')$.

3.2 Dynamic Load Balancing

Office networks tend to be heterogeneous, since computers are installed at different times, and newer computers tend to be more powerful. Also, some computers are in heavier use than others, and thus the amount of available processing power varies. The static load balancing mechanism of Parallel Murφ does not work very well in such environments.

The load balancing mechanism of our client–server algorithm is simple: a fast computer will communicate with the server more often than a slower one and thus receive and explore more states per time unit.

3.3 Termination Detection

Occasionally, the collection Q of unprocessed states may become empty. When a client invokes $\mathsf{getState}()$ and Q is empty, the server puts the client on a waiting list and waits for $\mathsf{putStates}(S')$ calls that could fill up Q. When all registered clients are on the waiting list, the server decides that all states have been processed and tells the clients to terminate.

3.4 Resource Management

Compared to a symmetrically distributed peer-to-peer algorithm, the client–server algorithm is easier to manage, as n worker processes need only n connections instead of $n \cdot (n-1)$. Starting the distributed state space exploration does not require any special software. The server process and the client processes can be started with ordinary shell commands.

The server handles client requests one at a time. When no requests are pending, the server waits for new client connections and incoming data from old clients. Because of this, new clients can join the computation at any time. Clients can also exit the computation safely by not invoking $\mathsf{getState}()$ once they have processed a state. In our implementation of the algorithm, the server maintains copies of the client-side input queues. In the event of a communication error, the connection is closed and the states that the failing client was processing are distributed to the remaining clients.

The server can host a user interface and show accurate progress measures (numbers of explored states and events). The computation can be interrupted easily. When the server terminates, clients will terminate as soon as they attempt to exchange the next batch of states with the server.

4 Results

The benchmark of our distributed safety property verification algorithm is a complete model (Tynjälä et al. 2002) of the radio link control protocol (ETSI 2000) of third-generation mobile telephone networks. With the chosen parameters, the model has 19,890 reachable states and 23,233 arcs.

We explored the protocol on a 128-processor SGI Origin 2000 in 32-bit mode and on a selection of GNU/Linux computers in a 100 Mb/s local area network. The first column of Table 1 indicates the speed of the sequential algorithm,[2] and Table 2 shows the average execution time of the parallel algorithm.

The two fastest computers in our office network are 160 % faster than a single processor of the supercomputer. The socket interface is efficient: running a server and a client process on the Athlon is about 5 % slower than running the sequential algorithm. On the SGI, the distributed algorithm with one client utilises about 99 % of the theoretically available power.

The processor utilisation factor declines, as clients are added. We define this factor as the ratio between the execution time of the sequential algorithm and the execution time of a distributed algorithm times the number of worker clients. With six clients on the supercomputer, the utilisation factor is $258\,\mathrm{s}/(6\cdot 53\,\mathrm{s}) \approx 81\,\%$, which means that a fifth of the computing resources is wasted. For up to 3 clients, the figure is close to 100 %. This is due to the relatively small branching factor of the model: at times, the collection of unprocessed states is almost empty, and clients have to wait for $\mathsf{getState}()$ calls to complete.

Of the workstations in the local area network, clients #1 and #2 process the state space in 100 s using the sequential algorithm, while clients #3 and #4 use 183 s and clients #5, #6 and #7 need 304 s. For 2 clients, we get a factor of $(54\,\mathrm{s}\cdot 2/100\,\mathrm{s})^{-1} \approx 93\,\%$. For 3 clients, the factor is $(42\,\mathrm{s}\cdot(2/100\,\mathrm{s} + 1/183\,\mathrm{s}))^{-1} \approx 94\,\%$, and so on. For up to six clients, the communication overhead eats only about eight percent of the computing power. With seven clients, the processor on one client is utilised for only about half the time.

Our distributed algorithm performs best when all clients are constantly employed, that is, their $\mathsf{getState}()$ queues do not become empty until the whole state space has been explored. In other words, the model being analysed should be

- rather nondeterministic, so that most states have several successors, or
- complex, so that the execution time of the verification is dominated by computing the function $\mathsf{successors}(s)$.

Clearly, the RLC model fulfils the latter condition, but its state space is rather deterministic, $23,233/19,890 \approx 1.17$ successors per state. Luckily,

[2]The second column of Table 1 is explained in Section 4.1.

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

Table 1: Time to explore the RLC model with Algorithm 1 and with path compression.

Computer Type	Wall-Clock Time						
N.o. Clients (n):	**1**	**2**	**3**	**4**	**5**	**6**	**7**
2 Athlon + 2 P III + 3 P II	105 s	54 s	42 s	35 s	32 s	29 s	28 s
	95 %	93 %	94 %	92 %	91 %	92 %	88 %
$n \cdot$ MIPS R12000	261 s	132 s	89 s	70 s	59 s	53 s	46 s
	99 %	98 %	97 %	92 %	87 %	81 %	80 %

Table 2: Time to explore the RLC model with Algorithm 3 and n copies of Algorithm 2.

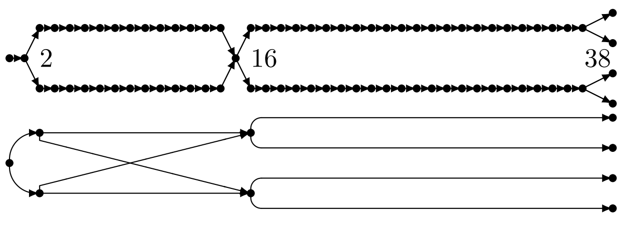

Figure 1: The first 39 levels of the RLC state space without and with path compression.

the state space branches rather early, as Figure 1 illustrates. The first three successors are computed by only one client, but from there on the analysis can employ another client process. At level 39, there is work for four clients.

4.1 Improving Scalability with Reductions: Path Compression

To reduce the state spaces of models of distributed software systems, we implemented a path compression option that eliminates deterministic behaviour:

Algorithm 4 *Algorithm 2 with path compression.*

```
do until server terminates
  $s :=$ getState(); $S'' := \emptyset$
compress :
  $S' := \emptyset$
  for all $s' \in$ successors($s$) do
    if error($s'$) then report($s'$)
    else $S' := S' \cup \{s'\}$
  if $|S'| = 1$ and $s' \notin S''$ then
    $s := s'; S'' := S'' \cup \{s\}$; go to compress
  putStates($S'$)
```

In this modified algorithm, the client does not invoke putStates(S') or getState(), as long as it is exploring a non-branching chain of states. The modified algorithm can significantly reduce a state space, but it can also require more processor time. Every time there is a branch to the middle of a sequential state chain, the states in the chain are explored again.

Figure 1 illustrates how Algorithm 4 affects the RLC state space. The original subgraph of 56 states and 79 arcs is reduced to 9 states and 10 arcs. In this case, the branching factor decreases from 79/56 to 10/9. For the whole state space, the factor improves from $23,233/19,890 \approx 1.17$ to $9,677/5,236 \approx 1.85$.

The auxiliary state set S'' in Algorithm 4 is needed for detecting cyclic behaviour in the non-branching chains of states. When a state with one successor is encountered, the successor will be explored only if it has not occurred earlier along the path being compressed. Obviously, the modified algorithm will visit all reachable states of the system; it just omits some of them from the main set of states S. Thus, all safety properties can be checked with this algorithm. It is more complicated to preserve other properties, such as liveness, as Miller and Katz (Miller & Katz 1999) have shown.

For the RLC model, the path compression algorithm computes 45,238 successor states instead of the original 23,233. Enabling path compression in the sequential algorithm approximately doubles the processor time requirement, as can be seen in the second column of Table 1. The good news is that with path compression, the analysis of this model can be distributed more efficiently to a larger number of processors, as a comparison between Tables 2 and 3 shows.

4.2 Performance on Simple Models

When exploring a high-level model where computing the successor relation clearly dominates the execution time, the processor time requirement of the server process (Algorithm 3) is negligible, as our implementation of the server deals with compacted states. One might wonder how well our distributed algorithm performs on a lower-level model where the speed of the server process and the network capacity might become limiting factors.

The distributed data base management algorithm that was presented by Jensen (Jensen 1981) as a Coloured Petri Net has one parameter, the number of peer nodes. With 8 nodes, the model has a branching factor of $81,664/17,497 \approx 4.66$, and with 9 nodes, the factor is $314,946/59,050 \approx 5.33$. The times in Table 4 demonstrate that the larger instance of the model utilises the processors better, scaling well for up to 11 or 12 processors.

5 Conclusion

State and memory reduction techniques have shifted the bottleneck of verification by explicit state space enumeration. The biggest limitation is no longer the memory consumption, but the execution time.

In many organisations, there is a huge capacity in the personal computers that are sitting idle most of the time, serving only interactive users. Users would not notice if their computers ran small background processes at low priority. An efficient parallel algorithm that minimises the inter-processor traffic can use almost all available processing power for useful computations.

The presented method can be applied to create a distributed version of any explicit state space verification tool for checking safety properties, since it is compatible with techniques that reduce the number of states or the memory consumption of the explored state set, such as the path compression method.

According to our measurements, the implementation of the algorithm in Maria (Mäkelä 2002) adapts well to heterogeneity and dynamically changing load of the processors, as well as to dynamic machine configuration changes.

Acknowledgements

The author would like to thank the anonymous referees and Charles Lakos for their helpful comments, and CSC – Scientific Computing Ltd. for providing access to the 128-processor SGI Origin 2000 supercomputer.

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
212 s	105 s	82 s	67 s	60 s	55 s	50 s					
99 %	100 %	99 %	99 %	99 %	99 %	99 %					
520 s	257 s	174 s	130 s	105 s	88 s	76 s	67 s	60 s	56 s	51 s	47 s
99 %	100 %	99 %	99 %	99 %	98 %	97 %	96 %	95 %	93 %	92 %	90 %

Table 3: Time to explore the RLC model with Algorithm 3 and n copies of Algorithm 4 (path compression) on a network of workstations, and on $n \cdot$ MIPS R12000. Compare to Table 2 and the last column of Table 1.

N.o.	**Alg. 1**	**Alg. 3 + $n \cdot$ Alg. 2**									
Nodes		n =**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 %

Table 4: Average time to explore the data base model on the SGI Origin 2000.

References

Ciardo, G., Lüttgen, G. & Siminiceanu, R. (2000), Efficient symbolic state-space construction for asynchronous systems, *in* M. Nielsen & D. Simpson, eds, 'Application and Theory of Petri Nets 2000, 21st International Conference', Vol. 1825 of *Lecture Notes in Computer Science*, Springer-Verlag, Århus, Denmark, pp. 103–122.

ETSI (2000), Universal Mobile Telecommunications System (UMTS); RLC protocol specification, ETSI TS 125 322 V3.5.0 (2000-12), European Telecommunications Standards Institute.

Holzmann, G. J. (1999), The engineering of a model checker: the Gnu i-protocol case study revisited, *in* D. Dams, R. Gerth, S. Leue & M. Massink, eds, 'Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops, Trento, Italy, July 1999, Toulouse, France, September 1999. Proceedings', Vol. 1680 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 232–244.

Holzmann, G. J. & Smith, M. H. (2001), 'Software model checking: extracting verification models from source code', *Software Testing, Verification & Reliability* **11**, 65–79.

IEEE (2001), Standard for information technology—portable operating system interface (POSIX®), IEEE Std 1003.1-2001, Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA.

Jensen, K. (1981), 'Coloured Petri nets and the invariant method', *Theoretical Computer Science* **14**(3), 317–336.

Kindler, E. & Völzer, H. (2001), 'Algebraic nets with flexible arcs', *Theoretical Computer Science* **262**, 285–310.

Latvala, T. (2001), Model checking LTL properties of high-level Petri nets with fairness constraints, *in* J.-M. Colom & M. Koutny, eds, 'Application and Theory of Petri Nets 2001, 22nd International Conference', Vol. 2075 of *Lecture Notes in Computer Science*, Springer-Verlag, Newcastle upon Tyne, England, pp. 242–262.

Lorentsen, L. & Kristensen, L. M. (2001), Exploiting stabilizers and parallelism in state space generation with the symmetry method, *in* A. Valmari & A. Yakovlev, eds, '2nd International Conference on Application of Concurrency to System Design', IEEE Computer Society, Newcastle upon Tyne, England, pp. 211–220.

Mäkelä, M. (2002), Maria: Modular reachability analyser for algebraic system nets, *in* J. Esparza & C. Lakos, eds, 'Application and Theory of Petri Nets 2002, 23rd International Conference', Vol. 2360 of *Lecture Notes in Computer Science*, Springer-Verlag, Adelaide, Australia, pp. 427–436.

Miller, H. & Katz, S. (1999), 'Saving space by fully exploiting invisible transitions', *Formal Methods in System Design* **14**(3), 311–332.

Stern, U. & Dill, D. L. (1997), Parallelizing the Murφ verifier, *in* O. Grumberg, ed., 'Computer Aided Verification 1997, 9th International Conference (CAV97)', Vol. 1254 of *Lecture Notes in Computer Science*, Springer-Verlag, Haifa, Israel, pp. 256–267.

Tynjälä, T., Leppänen, S. & Luukkala, V. (2002), Verifying reliable data transmission over UMTS radio interface with high level Petri nets. Unpublished manuscript.