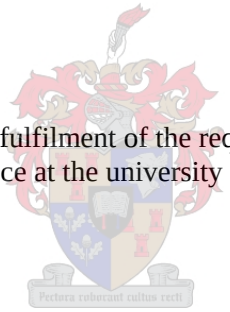
Reducing Communication in Distributed Model Checking

By

Jean Francois Fourie



Thesis presented in partial fulfilment of the requirements for the degree of
master of science at the university of stellenbosch

Supervised by: Jaco Geldenhuys
Co-supervised by: Cornelia Ingg

December 2009

Declaration

By submitting this dissertation electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

December 2009

Abstract

Model checkers are programs that automatically verify, without human assistance, that certain user-specified properties hold in concurrent software systems. Since these programs often have expensive time and memory requirements, an active area of research is the development of distributed model checkers that run on clusters. Of particular interest is how the communication between the machines can be reduced to speed up their running time.

In this thesis the design decisions involved in an on-the-fly distributed model checker are identified and discussed. Furthermore, the implementation of such a program is described. The central idea behind the algorithm is the generation and distribution of data throughout the nodes of the cluster.

We introduce several techniques to reduce the communication among the nodes, and study their effectiveness by means of a set of models.

Abstract

Modeltoetsers is programme wat outomaties bevestig, sonder enige hulp van die gebruiker, dat gelopende sagteware aan sekere gespesifiseerde eienskappe voldoen. Die feit dat hierdie programme dikwels lang looptye en groot geheues benodig, het daartoe aanleiding gegee dat modeltoetsers wat verspreid oor 'n groep rekenaars hardloop, aktief nagevors word. Dit is veral belangrik om vas te stel hoe die kommunikasie tussen rekenaars verminder kan word om sodoende die looptyd te verkort.

Hierdie tesis identifiseer en bespreek die ontwerpsbesluite betrokke in die ontwikkeling van 'n verspreide modeltoetser. Verder word die implementasie van so 'n program beskryf. Die kernidee is die generasie en verspreiding van data na al die rekenaars in die groep wat aan die probleem werk.

Ons stel verskeie tegnieke voor om die kommunikasie tussen die rekenaar te verminder en bestudeer die effektiwiteit van hierdie tegnieke aan die hand van 'n lys modelle.

Acknowledgements

I am grateful to the people who supported me in finishing this thesis:

- First and foremost, I would like to thank my supervisor Jaco Geldenhuys for his guidance, support, and patience. His attitude is resolute in that he is *always* positive, helpful, and friendly.
- I would also like to thank my co-supervisor Cornelia Inggs for her advice and kindness, and for helping me add structure and purpose to my schedule.
- I want to express my gratitude to the Department of Computer Science at Stellenbosch University for granting me enough time to finish this work.
- I gratefully acknowledge the financial support I received from the Stellenbosch bursaries.
- I am indebted to Deon Borman, Wessel Venter, and Van Aarde Krynauw for their patience and time in making sure the Beowulf cluster is always up and running.

Finally, I thank my parents for their love and continuous support, and my brothers and friends for their encouragement and motivation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software reliability has become critically important in today's technology-driven world. The consequences of improperly-tested software can range from loss of money to even the loss of human life. For example, between 1985 and 1987 at least three hospital patients died due to malfunctions in a radiation therapy machine called Therac-25 [44]. The malfunctions were caused by race conditions that allowed sequences of events that activated x-ray mode without the machine's tungsten shield in place. As a result, the patients were exposed to massive overdoses of radiation. Finding all such incorrect sequences of events manually is practically impossible for concurrent systems due to the complexity of the software and the large number of possible execution sequences that such systems typically have. Instead, programs called model checkers are designed and used to automatically verify the correctness of large concurrent software systems.

However, in the time-intensive process of model checking, memory is often exhausted. In such cases complete verification results are sacrificed and we have to accept incomplete or no results. Many techniques have been successfully applied to model checkers to reduce the time and memory requirements. A recent, more direct approach is to increase the available hardware memory and processing power by implementing a model checker to run on a cluster of machines, which we will refer to as "nodes" in this thesis.

Usually, and in our case, model checking examines the software system as a set of system states. The use of distributed or parallel model checking relies on the partitioning of all such

states amongst the nodes in the network. Thus, every state is processed by a unique node, its owner. This requires a means of communication, for example, message passing. However, communication time is expensive and restricted by the speed of the network.

One way to reduce communication is to make use of the fact that in some cases state partitioning operates independently from the model checking. For example, alterations are made on how states are partitioned so that nodes generate fewer remote states [7, 8, 43]. In other approaches the model checking algorithms themselves are modified [8, 9, 10, 24].

The goal of this thesis

In this thesis we discuss the design and implementation of two independent strategies, both aimed at decreasing the communication between nodes. In both cases fewer states are sent between nodes. Briefly, we investigate the following:

- Method 1: each node keeps a fixed amount of work, belonging to other nodes, for itself.
- Method 2: a distance value is calculated for every state, and distances are assigned to a range and each range to a node.

In other words, Method 1 circumvents the state partitioning to a certain extent, and Method 2 uses a more localized state partition function.

A distributed model checker, using a static partitioning function, is implemented for state enumeration. This algorithm is then extended into three model checking algorithms; two based on the algorithm presented by Courcoubetis et al. [30] (which we shall refer to as the CVWY algorithm) and one based on bounded model checking [41]. Method 1 is implemented on each of the algorithms, while Method 2 is used as an alternative partitioning function in the state enumeration algorithm. Finally the two methods are independently evaluated.

Thesis outline

Chapter 2: Background on model checking presents a brief overview of both sequential and distributed model checking. It discusses the model checking problem and specifically

examines the state graph, the requirement properties, and the model checking algorithm. Furthermore, the so-called state explosion problem, which means that the number of states grows exponentially in the number of variables and processes in a system, is discussed.

Chapter 3: Distributed design issues is essential to the thesis, because it defines the issues related to model checking, and more specifically distributed model checking. The focus is primarily on delays in communication and on how states are efficiently assigned to nodes so that all nodes are kept busy and memory is efficiently utilized. Furthermore, aspects that are important to both distributed and sequential model checking algorithms are discussed. Finally, an overview is presented of work related to the distributed issues.

Chapter 4: Design and implementation describes in detail the distributed state enumeration algorithm we use, as well as the extension of the algorithm into existing distributed model checking algorithms. Furthermore, two techniques are introduced to reduce communication between nodes, and several strategies are examined to reduce the lengths and memory-requirements of receiver queues on nodes. Finally, this chapter discusses the format of input models and then elaborates on further implementation details.

Chapter 5: Evaluation presents the results of experiments conducted to measure the performance of the various techniques implemented on the model checker. This includes the performance effects of the queue-reduction strategies, the communication-reduction strategies, and the algorithm itself. The chapter concludes with an interpretation of the results.

Lastly, a summary and conclusions are given in ***Chapter 6: Conclusion.***

Chapter 2

Background on model checking

Conventional testing is not exhaustive and requires some human assistance, thus making it prone to human error. This is especially true for concurrent programs where multiple processes can, at the same time, modify and read the values of shared variables. Depending on the number of processes and variables, the scope of the execution paths is often very large. Instead of conventional testing, model checking can be used to verify the correctness of such programs. A user inputs a system specification and a correctness property, and the model checker checks if the property holds in the system. In this context, the *system* is a hardware or software implementation, or some combination thereof. The model checking process is automatic and the user requires no knowledge of how the model checker works. The earliest model checking algorithms were developed by Clarke and Emerson [15], and by Queille and Sifakis in the early 1980's [50].

A system is a set of states and transitions. Every state is a representation of the system in time, and every transition moves the system from one state to another. The complete set of states of a system is called its state space. Our approach to formal verification is explicit model checking, where each of the system states is generated explicitly, explored, and evaluated. This is also known as *state enumeration*. Another approach, one not taken here, is symbolic model checking [13]. In this case the state space is represented implicitly using, for example, binary decision diagrams [2, 12]. Symbolic model checking is typically used for hardware verification and explicit model checking for software verification, and therefore we focus on the latter.

We also assume that all systems have a finite number of reachable states. However, computers have memory limitations and only limited-sized state spaces can be explored. The cause of large state spaces is the state explosion problem and is discussed in Section 2.2. In distributed or parallel model checking we can combine the physical memory from a cluster of machines and thus verify larger state spaces. In both distributed and parallel model checking the states are divided amongst the machines. In Section 2.3 we discuss the details of distributed model checking, why we have opted to use distributed model checking, and how it differs from parallel model checking.

The different model checking components are discussed in Section 2.1, and an overview of related work on distributed model checking is postponed until Chapter 3.

2.1 Model checking components

A functional model checker requires the following components:

- *an input model* representing a software system,
- *a correctness property* that should hold for the system, and
- *a model checking algorithm* to verify that the property holds for the system.

Sections 2.1.1, 2.1.2, and 2.1.3 deal with these individually.

2.1.1 State graph

We focus on the case where systems are comprised of concurrent processes coupled asynchronously and can be modeled as automata. We will not distinguish between automata and graphs. This means that we shall use the terms “vertex” and “state”, and “edge” and “transition” interchangeably. The vertices and edges of the state graph are the states and transitions. To obtain this graph we first construct a simplified model of the system [38]. Each of the asynchronous processes is represented as a finite state automaton where the states hold the values

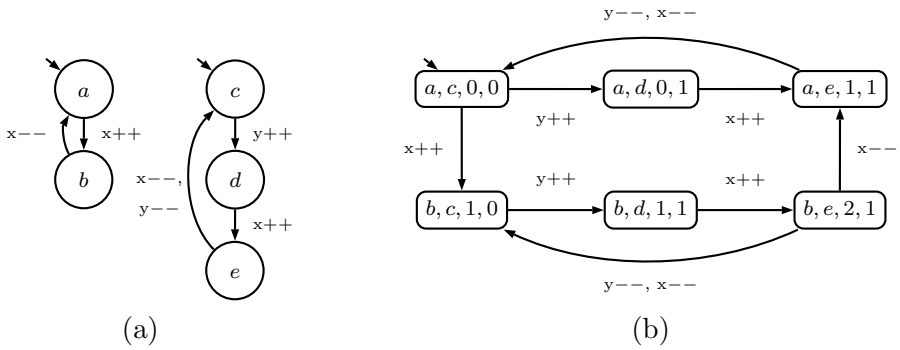


Figure 2.1: The state automata of (a) two concurrent processes and (b) their product.

of the variables local to the process. Finally, the product automaton of the process automata forms the state graph.

The models are nondeterministic. This means that a state can have more than one transition, and the order in which a state's transitions are explored is not specified. We only deal with asynchronous communication between processes, and thus each transition only represents a change in a single process.

The example in Figure 2.1, the *IncDec* model, illustrates how the state graph in (b) is formed from a system with two concurrent processes in (a). The variable x is global and accessible by both processes while only the process on the right has access to variable y. In (a) the first process only increments and decrements x while the second process first increments y and then x. Once both variables have been decremented, the system returns to the initial state. In the state graph in (b) every state is marked by the local states of the component processes and the values of x and y. For example, the label $\{a, c, 0, 0\}$ means that the first process is in local state a, the second process in local state c, and both x and y are 0. Such a model is typically expressed in a high-level specification language. In the case of the model checker Spin, the Promela language is used [38].

2.1.2 Requirement property

The second input to the model checker is the correctness property, which is expressed in temporal logic. There are many types of temporal logic, but in this thesis we focus on *LTL* (linear

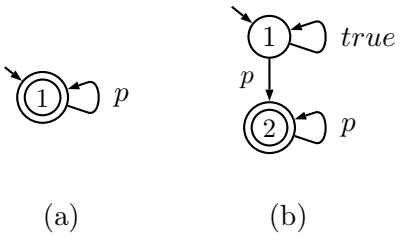


Figure 2.2: The corresponding Büchi automaton for (a) Gp and (b) F(Gp).

temporal logic) [23, 49]. LTL formulae are constructed by combining propositions, standard logic operators, and temporal operators. There are five of the latter: G (always or globally), F (eventually or finally), U (until), R (release), and X (next).

The meaning of the temporal operators is defined with respect to an execution path. Starting at the initial state, an execution path is any sequence of enabled transitions in the state graph leading from state to state. A path that includes a cycle is an infinite path. An execution path satisfies the formula Gp when the proposition p is satisfied for every state in that path. On the other hand, the formula qUr means that proposition q holds for every state in the path up until a state in which r holds, after which q and r can have any value. In this case, r is guaranteed to eventually hold. The other temporal operators are defined in a similar way.

The correctness property is checked for every path of the system until eventually the formula is false in a path, or until all paths are exhausted, in which case the property is satisfied by the system. Properties can be classified as either *safety* or *liveness* properties. A safety property expresses that "something bad will never happen", and a liveness property states that "something good must eventually happen". The formula Gp is an example of a safety property, while the formula G($p \Rightarrow$ Fq) is a liveness property which asserts that whenever p is true, q will also eventually become true.

LTL formulae are translated into finite automata, known as *Büchi* automata. The details of Büchi automata are not important here, but will be discussed in Section 4.3.1. Figure 2.2 depicts two automata translated from LTL formulae. The Büchi automaton takes part in the construction of the product automaton (the state graph). It is important to note that the correctness property is first negated. This means that the model checker will not explicitly

check that every execution of the product satisfies the property (the so-called "language subset problem"), but rather it will search for an execution that violates its negation (the so-called "emptiness problem"). On a high-level, this amounts to the same thing, but at the implementation level, the latter approach is more practical than the former.

2.1.3 Model checking algorithm

The model checking algorithm is used to explore the state graph and to check whether the correctness property is valid or invalid. In most model checkers either a breadth-first search (BFS) or a depth-first search (DFS) is implemented to search through the state graph. In Figure 2.3 both algorithms are used to search through a small state graph and in each case the edges are marked with the order in which they are first visited. When the DFS algorithm in (b) finds an already visited state, the last edge is backtracked and the search follows a new path. On the other hand, in the BFS algorithm in (a) a state's children are all queued and then visited in order.

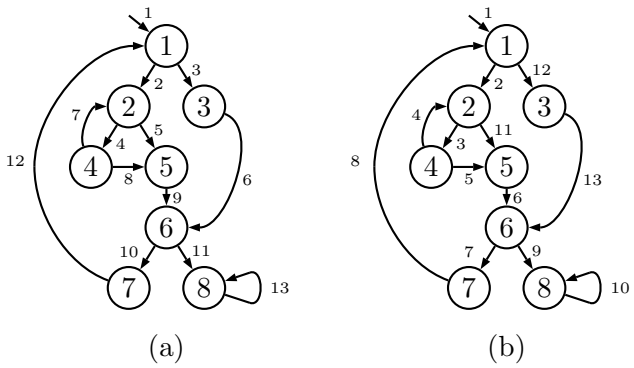


Figure 2.3: The search through the state graph in (a) breadth-first and (b) depth-first manner.

A model checking algorithm can be classified by two characteristics. The first is whether the algorithm is *structural* or *automata-based*. We use an automata-based approach [6, 16, 54] where each temporal logic formula is translated into an equivalent automaton as mentioned in the previous section. Thus, existing methods from automata-theory can be applied to check if a formula holds in the system. Structural algorithms are based on the structure of the formula and formulae are simplified and expanded into parts that are individually evaluated [45].

The second characteristic is the choice between a *global* and a *local* algorithm. Local algorithms check if the property is satisfied in a specified state, for example the initial state. Global algorithms check if the property is satisfied in every state. In our case, we focus on local, on-the-fly algorithms, where "on-the-fly" means states are explored as they are generated. States are explored only until a property is found invalid. In the worst case every state in the state space is visited one or more times. To avoid redundant explorations of a state, every state is stored in memory. The alternative is an "offline" algorithm where the state space is generated before the verification process starts.

For some properties it is enough to examine individual states, while for other properties it is necessary to investigate entire paths. To solve the emptiness problem (which demonstrates that the system satisfies the correctness property) the model checker looks for accepting cycles (i.e., cycles that contain an accepting state of the product automaton). As soon as such a cycle is detected, the model checker may terminate and report a violation of the correctness property.

Two model checking algorithms are the Tarjan algorithm [31, 52] and the CVWY algorithm [16]. In the latter, a second, nested depth-first search is started once an accepting state is found to discover if it belongs to a cycle.

2.2 The state explosion problem

Every process in the system has a number of local states, and unique states are formed based on the combinations of different local states. The number of these global states grows exponentially as the number of processes increases. A system with n asynchronous, independent processes, each with k local states, will consist of k^n global states. This leads to a memory problem in larger models, because there is not enough room to store all of the states. When only a part of the state space is stored, some states may be explored more than once, and the search may take an exponentially long time to complete.

Many techniques have been devised to alleviate this problem. In partial order reduction [32, 38, 48, 53], work is decreased by eliminating redundant paths. An example is given in Figure 2.4 where the order of transitions is different, but both paths end in the same state. Every state is

labeled by the current values of $\{x, y\}$. The dotted path is omitted and not considered in the evaluation. In other techniques the space used to store states is minimized. These include state compaction [29], hash compaction [20], state compression [34, 38], and bitstate hashing [37, 38]. Alternatively, in symbolic model checking the state space is also compressed [13]. In another approach states are kept on a magnetic disk instead of in memory [22]. However, writing to a disk is much slower than writing to memory.

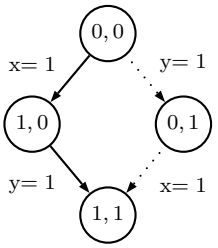


Figure 2.4: Partial order reduction omits redundant paths.

One of the most recent ideas to deal with state explosion is the use of parallel or distributed model checking, thereby increasing the capacity for the number of states to be stored.

2.3 Distributed model checking

Nodes coupled together asynchronously, are exploited either by using a shared-memory architecture (in the case of parallel model checking), or a message passing paradigm (in the case of distributed model checking). In the former, every node has both local memory and global memory. The global memory is shared and nodes can communicate by reading from and writing to it.

In a distributed environment nodes have only local memory and they communicate with each other via message passing. We adopt this approach, because it is widely available to users and cheaper to come by. A distributed cluster was available to us.

Distributed and parallel model checkers are highly diverse in how the workload is partitioned and how the algorithms deal with verification. A large group of partitioning strategies assigns states to nodes. An algorithm runs sequentially on each node and when a state is found that

belongs to another node, it is sent there. This approach is used here and the partitioning is discussed in Section 2.3.1. Section 2.3.2 briefly discusses the distributed algorithms.

2.3.1 Partitioning of the work

To avoid wasting memory, a state must not be stored on more than one node. If state s is a child of both *p1* and *p2* it will be generated twice, once on the node exploring *p1* and once on the node exploring *p2*. Thus, s is assigned to one owner node and will be explored and stored only on this node. For a network of N nodes the state space S is partitioned into N partitions $\{S_1, S_2, ...S_N\}$, where S_i belongs to node i. Here i is a unique node number from 1 to N. This means that every state s belongs to one state partition, $s \in S_i$, and a partitioning function *owner(s)* calculates the associated node number for s.

During exploration a node will encounter both *local* and *foreign* states. Foreign states belong to a remote node while local states belong to the exploring node. The foreign states are sent to their respective owners. Transitions to foreign states are called *cross-transitions* and each of these transitions forces one SEND operation. Local states are normally queued and explored locally.

Some model checkers do not assign states to nodes, but instead run algorithms independently on different nodes. For example, when networks are not permanently available for one computational task, independent bounded searches can be used [39]. In general, running independent tasks increases the chance of duplicate work.

2.3.2 Algorithms in parallel

A distributed algorithm incorporates the same characteristics and ideas as sequential algorithms (see Section 2.1.3). Both a BFS and DFS can be distributed [21, 43]. However, in distributed model checking, as a consequence of cross-transitions, paths are interrupted and jump between nodes. The nodes in the cluster contributes to a combined parallel search.

The interruptions in paths present complications that are discussed in Section 3.1.3. These complications are largely related to the use of a nested search in sequential algorithms. The

searches are used to find accepting cycles by examining the current path. Since these paths are split amongst nodes, the entire path history is not immediately available.

Apart from split paths, there are more issues that we have to deal with when implementing distributed algorithms, as opposed to sequential algorithms. In the next chapter we elaborate on these, as well as issues that are related to both sequential and distributed algorithms.

Chapter 3

Distributed design issues

In the design of distributed model checkers we are concerned with two limitations:

- *Not enough memory*: As mentioned before, distributed model checking is used to verify models with large memory requirements, otherwise not possible on a single machine. Even if each node of the cluster has only a fraction of the memory of a single machine, the combined memory capacity is usually much greater. In this case it is more desirable to obtain complete results even though we have to wait longer for it.
- *Not enough time*: In this case, distribution is used to implement individual verification tasks in parallel. This is usually efficient for algorithms that lack coordination and communication between nodes, and are designed to find results fast. Thus, redundant work is not eliminated.

The bottleneck in both cases is communication, but we focus only on the first case. Since every cross-transition results in one SEND operation, it is possible to estimate the effect communication has on time. When a model with state space S is explored on a network with N nodes, a good partition function will assign approximately $\frac{S}{N}$ of the states to every node. If a node generates M states in total, we assume that $\frac{M}{N}$ states of these are local. The other $M \times \frac{(N-1)}{N}$ states are foreign and result in SEND operations. According to the LogP model, every SEND suffers network delays [17]. This includes the time a node is engaged in transmission or reception of a message, the latency delay in communicating a small message, and the time to

transfer the state's bytes to the recipient node. Assume that we explore 1 million states on a network with 7 nodes, and that every SEND costs 0.2ms. If each node generates 143 000 states ($M \approx \frac{S}{N}$), 858 000 SEND operations are needed and 172s is spent on communication.

The time in this example is only an approximation; redundantly generated states are not taken into account, nor do nodes generate the same number of states throughout the network. However, the example illustrates that communication is expensive, even for a small model.

Other than communication, it is important that states are evenly distributed amongst the nodes. If one node's state partition is too large in comparison to the partitions of the other nodes, the node may exhaust its memory which means the verification will terminate. This type of balance is referred to as spatial balance and is essential when memory capacity is a problem [14].

However, an even state distribution does not guarantee that all nodes will be busy most of the time. Temporal balance is the available, unexplored states on every node [14]. Every node must always have available states, otherwise nodes waste time idly waiting for work to arrive.

In this chapter we first elaborate on the distributed issues in Section 3.1. Then issues, that are relevant to both sequential and distributed model checkers, are discussed in Section 3.2. Finally, related work is listed in Section 3.3 and is based on partitioning and communication problems in distributed model checking.

3.1 Distributed design issues

Problems that relate to networking and state partitioning entail the following:

- partitioning of the state space,
- temporarily storing received states until they are explored,
- splitting a path and continuing it on a different node,
- avoiding redundant distribution of the same state on the same node,
- communication between nodes, and

- termination detection.

These are discussed in Sections 3.1.1 - 3.1.6 respectively.

3.1.1 Partitioning: locality and spatial balance

A partition function maps every state to a unique owner node, and thus every state is only explored and stored once. This entails the following:

- a routine that calculates the *owner* of every state, i.e., finds a unique node index between 0 and $N - 1$ for every state in a network with N nodes.

To minimize communication the partitioning must minimize cross-transitions and increase *locality*. Locality means that a state is mapped to the same node as its parent. Figure 3.1 illustrates three ways to partition a small state space. In (a), nodes A and B both receive an equal share of 3 states, and memory is most effectively utilized. However, there are 6 cross-transitions. On the other hand, in (b) each node still explores only 3 states, but the cross-transitions are reduced to 2. A third case in (c) also offers 2 cross-transitions, but the state partitions are now unbalanced. We refer to how the states are partitioned as *load balancing*.

Designing a partition function that results in the second case is challenging. To find which states should be grouped together we need to know more about the state graph. However, this knowledge can only be obtained by a search through the state space, and this defeats the purpose of on-the-fly exploration. A common solution is dynamic load balancing in which the

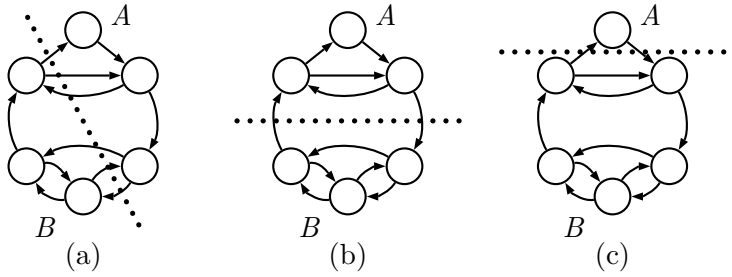


Figure 3.1: A state space is split to (a) ignore locality, (b) take locality into consideration, and (c) take locality into consideration but with bad load balancing.

state partitions are continually changed and adapted to the shape of the state graph. To make sure that the partitions are equally sized, states are also dynamically shifted between nodes. However, in most dynamic implementations nodes meet in synchronization points (this is called barrier synchronization) and this can lead to further delays. In dynamic implementations that omit synchronization points, situations may arise where nodes have outdated information about the workload of the other nodes in the cluster, and this results in work being sent to the wrong nodes. It is hard to find a balance between locality and good spatial balance and in most cases the focus of partitioning is on only one of the two.

3.1.2 Temporal balance and waiting queues

If states received from other nodes are not kept until they are explored, the exploration will not be complete. Nodes temporarily store these states in waiting queues and this entails the following:

- routines to *enqueue* and *dequeue* a state into and from the queue respectively.

The length of these queues is a good estimate of the available workload, or temporal balance, on each node. Queues can be unbalanced and this often happens when some nodes receive too many states in a given period and cannot remove and explore these states quickly enough. When queues are too long, memory is exhausted and verification is stopped.

With dynamic queue balancing, queues can be shorter, because states in large queues are moved to shorter queues on nodes other than their owners. However, this approach suffers from synchronization delays in some implementations when nodes meet at synchronization points. In addition, duplicate visits are made to the same state when a local state is moved to and explored by a remote node, before it is eventually explored again on its owner. In a simpler approach states are only queued when they are not in the store (see Section 3.2.2). In some cases queue balancing is also referred to as load balancing. When the term is used and the context is not clear, we shall indicate to what it refers.

It is desirable that the reception and management of incoming states do not delay the exploration. In the same way, if nodes poll too often for new messages, the exploration will be

delayed.

3.1.3 Distributed exploration

The difference between sequential and distributed exploration is that a distributed search is broken up into parts. Normally the same search algorithm runs sequentially on every node, and as soon as a foreign state is encountered on one node the search is continued on a different node, i.e., the owner of the state. If the purpose is to explore every state, any search algorithm is acceptable, and safety checking can be applied to every state.

On the other hand, checking liveness properties is much more complex. In sequential algorithms, like Tarjan's algorithm [31] and the nested-DFS algorithm used in Spin [38] (adapted from the *CVWY* algorithm), cycles are detected using DFS. By its nature a DFS cannot be distributed [51]. It is also infeasible to run a DFS on every node, because in order to find a cycle we need the path history of every state. Even though it is possible to attach to every state its history, it is too expensive. A BFS is much easier to distribute. In Section 3.3.1 we briefly mention how algorithms accomplish distributed liveness checking.

3.1.4 Redundant sends

It is possible for the same state to be generated more than once on the same node. If the generated state belongs to another owner, it serves no purpose to send it more than once. Instead, sent states are recorded in a cache and every foreign state is checked against the cache before it is sent. This cache entails the following:

- a routine that *inserts* a foreign state into the cache and, if full, replaces an old state.
- a routine that *checks* if a state was already cached, and thus already sent.

The cache uses memory otherwise reserved for the store, and therefore the cache is bounded. When the cache is full, the memory used by older states is reassigned to new states. Different strategies exist to select old states [28, 35, 36].

If the size of the cache bound is too large, insufficient store memory is available, and if too small, caching overhead can slow down the time saved on communication. The size depends on the number of cross-transitions, and thus indirectly on the lengths of the waiting queues, because these queues store the states that result from the cross-transitions. A system with a smaller percentage of cross-transitions requires a smaller cache.

3.1.5 Communication

A distributed model checker requires routines to communicate data between nodes and these include:

- a routine to *send* states to remote nodes, and a routine to *receive* states sent from other nodes,
- routines to relay *termination* information between nodes, and
- routines to relay *partial results* between nodes.

Communication further entails the choice of a message passing paradigm that is reliable and fast; a survey on how to model a good communication strategy by Joubert [40] was used to make some of the choices involved; these are summarized in the rest of this section.

High-level paradigms like MPI/PVM are easier to implement but slower than a low-level TCP/IP socket implementation. Though more complex, the low-level implementations are more suited for resource-intensive applications such as model checking.

Message passing is either synchronous or asynchronous. In synchronous operations the sending node is blocked until a reception acknowledgement of the sent message arrives, and a receiving node is blocked until a message arrives. Asynchronous operations continue to take place in parallel with other operations on the same node after being called. Asynchronous message passing can be dealt with in three ways:

- **Blocking**: Each sending call must synchronize with one receiving call before being unblocked. Here the processes are blocked at system level.

- **Non-blocking with an unbounded buffer**: Each sending call buffers a new message without considering memory limitations, and is immediately unblocked.
- **Non-blocking with a bounded buffer**: Each sending call buffers a new message, but a limit is placed on the size of the buffer. The call is immediately unblocked.

In the blocking case there will be delays should a receiving call not react quickly enough. In the second case memory may be exhausted if more messages than fit in the available buffer space are sent. However, there will not be any delays once a message is sent. There are also no delays in the case of the bounded buffer, but the implementation is difficult since the success of each sending call must be checked and, if not successful, the call must be made again.

3.1.6 Termination detection

Termination detection is required by all distributed software. It is therefore possible to adopt a known strategy that works efficiently. However, in some termination detection strategies, nodes are stopped at synchronization points and some nodes are idle and wait for others to synchronize. These strategies are best avoided in distributed model checkers. It is preferable that nodes stay busy and terminate when there are no more states to receive or explore.

3.2 Extra design issues

Some design issues relate to both sequential and distributed model checkers. These are essentially the same as described elsewhere in the literature [27].

For a model checker the focus is on selecting an algorithm to explore every state in the state space as fast as possible. This entails selecting between a global or local algorithm, and between a structure-based or automata-based algorithm. Expressing requirement properties, demands a choice between a branching or linear time logic, for example, between CTL and LTL, respectively. In addition, the system models must be expressed in a specification language.

Once an algorithm is selected, it is used to generate the state graph. The system explores one state at a time, referred to as the current state. The functionality of the algorithm is divided

into specific tasks: the generation of states, the detection of revisited states, and the efficient representation of states. These are discussed in Sections 3.2.1, 3.2.2, and 3.2.3, respectively.

3.2.1 Generation of states

In this section we describe one example of a state generator that is relevant to this thesis. The state graph is constructed through the state generation process which starts at the initial state and applies the following to every state:

- a routine that generates the *children* of the current state, one at a time. Each transition associated with the current state is checked if enabled (See the end of this subsection) until one is successful or all the remaining ones are disabled.

Once the children of the current state have been explored, the search backtracks to the parent of the current state. Here the children include the children of the children, and so on. The search terminates once the children of the initial state have been explored. We can find and evaluate the transitions of the current state by examining in which state each process is. Four structures are maintained for this:

- *a state table* to link transitions to individual process states,
- *a transition table* to link transitions to a set of commands,
- *a list of commands* that implements every transition, and
- *a stack* to keep track of the current state's last transition explored.

The first and second structures relate to the states and transitions of every process. The command list consecutively describes the transitions, each in the form *guard* $\rightarrow$ *action*. A transition is only enabled if the Boolean guard expression evaluates to *true*. Once enabled, the action commands are used to transform the state into the new child state.

When properties that require cycle detection are checked, a new state is only formed if both a process transition and a property transition are enabled. (We assume here, for simplicity,

that there are no deadlocking process transitions.) Since every property is translated into an automaton, we can use this to find the property transitions. Every property transition is checked in combination with every process transition.

3.2.2 Detection of revisited states

It is important that the same state is not stored or explored redundantly. A storage mechanism must be implemented that detects when states are revisited. The state *store* is such a data structure and has the following functionality:

- a routine to *insert* a state into the store, and
- a routine to *lookup* whether a state is already present in the store.

For fast lookup the store is usually implemented as a hash table where each slot is in itself a sorted linked list. Two techniques that apply directly to the store implementation can be used to reduce the memory utilization: bitstate hashing [37, 38] and state caching [35]. The former uses a large array of bits as a hash table. Every state is hashed and once explored the bit is set, and the store would be unable to detect the collision. However, this may produce incomplete results, because two states might hash to the same bit. In state caching states are also revisited, because states are overwritten when the cache is full.

3.2.3 Representation of states

States need to be represented efficiently to save memory. This is essential in distributed model checkers where the size of states contribute to message overheads. State management entails the following functions:

- a routine to *set* a value at any location in a state, and a routine to *get* a value at a location in the state,
- a routine to *compare* two states for equality, and
- a routine to *replicate* a state.

States are either stored *explicitly* or *implicitly*. In the first case states are typically vectors, or arrays, and each slot represents a property or variable of the system. Here the set and get routines require a simple overwrite and lookup at a desired vector location. The vector size is either *fixed* or *variable*. In variable-sized vectors the size is specific to every state, and dynamic process creation is easier, because new state components are dynamically added to a state. When vectors are fixed the size must be large enough to accommodate all state sizes. This means that for some states, slots are left empty and memory is poorly utilized. Variable-sized vectors require dynamic memory allocation which is more expensive to implement, but no (or little) memory is wasted.

State compaction is a technique in which we reserve only those bits required for every state location. The number of bits depends on the range of values possible for a location. This greatly reduces the memory requirements [29].

Alternative presentations are to store states implicitly. For example, symbolic model checking employs BDDs to represent the state space [2, 12]. Implicit representation techniques are more expensive time wise even though it reduces the memory requirements.

3.3 Related work

The idea of distributing model checking across a network of computers was first proposed by Stern and Dill in 1997 [21]. Considerable research effort has been devoted to the problem, and an annual workshop PDMC (Parallel and Distributed Methods in verifiCation) that focuses on this, has been held since 2002. In this section we describe some of the work that is relevant to the issues in this thesis.

In Stern and Dill's model checker, the Murphi verifier [21], the focus is on state enumeration and not property verification. A hash function randomly calculates an owner node for every state to which the state is sent for exploration. Every node runs a BFS algorithm and polls for new messages when the waiting queue is empty. Received states are only queued if they have not been stored. Times are improved when the number of nodes is increased, and the state space is split evenly among the nodes, but the number of cross-transitions is not taken into

account. This penalizes this approach with communication overheads. Message aggregation is mentioned, but not implemented. It combines small groups of messages into one larger message to reduce the communication.

Garavel et al. [26] implements a partition function, similar to the one in the Murphi verifier, on top of a hashing function. In this case the hashing function is part of the OPEN/CÆSER environment [25] in which their model checker is built. The hashing function $f(s, P)$ with a prime number P, calculates a value between 0 and $P - 1$ for every state s. The state space is divided into P subsets and these are distributed among the nodes in the network by $f(s, P)$ *mod* N (where N is the number of nodes in the network). This method distributes the state space into N almost-even subsets. However, once again the relationships between parent and children states are ignored.

In this section we concern ourselves with approaches to improve locality (by minimizing cross-transitions), or on the other hand, to improve spatial and temporal balance. However, we first consider algorithms in which distributed cycle detection is solved, in Section 3.3.1. Methods for load balancing are discussed in Section 3.3.2, and methods for locality in Section 3.3.3.

3.3.1 Distributed cycle detection

This subsection presents a brief overview on how cycle detection has been implemented in the distributed environment.

Brim et al. reduces the cycle detection problem to a negative cycle detection problem by assigning negative weights to accepting cycles [11]. The nested DFS algorithm has also been distributed in other ways. Barnat et al. distributes a nested DFS by using data structures on every node that keep track of individual nested searches continued on from remote nodes [5]. Geldenhuys et al. successfully distributes the nested DFS in the distributed CVWY algorithm [30]. With every state is attached a root, and once an accepting state is found, the root is set to it and the path is split in two; one with the original root and one with the new root. When the root is the same as the state, an accepting cycle is found. Thus, an exhaustive nested search is simulated by using the roots to keep track of accepting states. In a different approach, Černá et al. assigns nodes to states based on strongly connected components and

nested searches need not cross onto other nodes at all [3]. Finally, Krčál uses bounded model checking in four distributed model checking algorithms, two of which are based on running nested searches sequentially [41].

3.3.2 Load balancing

In this section we look at algorithms in which either spatial or temporal balancing is addressed. Load balancing techniques are implemented either statically or dynamically. In static partitioning a predefined function maps states to nodes. In dynamic balancing on the other hand, queues or state partitions are calculated and adjusted during runtime.

In a mapping and remapping strategy by Ciardo and Nicol, states are mapped to classes during state enumeration [46]. A search tree is constructed from random walks during an initialization phase. Every new state is propagated down the tree and every tree exit point belongs to a class. States in this tree belong to class 0. Classes are mapped to nodes. In addition, a search tree is constructed for every class on each node to allow remapped classes to be indentified and removed. In remapping, nodes meet at synchronization points where they calculate the average spatial load. Above-average nodes are matched with below-average nodes and states and class search trees are exchanged so that all nodes are moved closer to the average. This strategy requires constant remapping, because once the classes are largely unbalanced, it requires too much remapping overhead to correct. However, constant synchronization slows down exploration.

This mapping and remapping strategy can aid locality, because state components (states maintain a component for every process) are compared from left to right. A propagating state will end up near states with similar state components, and between a parent and child state only a few components change.

Allmaier et al. implemented a different partitioning strategy that groups the state space into contiguous, ordered blocks, one for every node [1]. A master node initiates a load balancing phase once one block is a predefined percentage larger than the others. The states at the start of a contiguous block are called the minimal states and those at the end are called the maximal states. Nodes are ordered according to the order of their blocks, and exchange minimal and

maximal states respectively with their left and right neighbours. The master node controls how much memory each block should use.

Černá et al. assigns strongly connected components (SCCs) to nodes [3]. The LTL formula part of every state is checked to find in which strongly connected component the state belongs. The state is then assigned to the node owning that SCC. The decomposition into SCCs is approximated using information from the property automaton to form maximal SCCs, because a full decomposition requires a search through the entire state space. In this scheme, cycles are not split across nodes.

Heyman et al. investigated a symbolic model checking algorithm [33] that runs a sequential BFS and, when the memory usage exceeds a threshold, the state space is dynamically partitioned into slices, each assigned to a node. Furthermore a coordinator node initiates a load balancing phase when the load becomes unbalanced, and matches nodes with large memory requirements to nodes with small memory requirements. The union of the slices of two such nodes is divided into two equal-sized slices. In this algorithm states are sent directly between nodes, but this communication requires coordination from the coordinator.

Queue imbalances in the distributed UPPAAL algorithm [7] are countered by redirecting foreign states to a node other than the owner. This happens if the owning node's load exceeds the average load by too much. Thus, on top of the partition function that computes a unique random owner for every state, an extra load balancing layer is implemented. In addition, the hash tables of the waiting queue and the queue of unexplored states are merged; this idea is also documented by David et al. [18] who use it to reduce the memory requirements.

Kumar and Mercer deal with queue imbalances that result from static partitioning functions in explicit model checking, by grouping nodes into relationships based on hypercubes, and nodes only equalize their workload with dimensional neighbours (see Figure 3.2) [42]. The algorithm has been implemented in the Murphi verifier [21] and at iterative steps every node communicates its queue size to its neighbours. If this size exceeds a neighbour's queue size, it sends states to that neighbour.

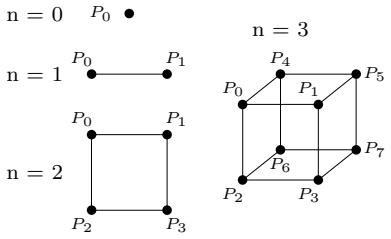


Figure 3.2: Hypercube structure for 1 to 8 processors [42].

3.3.3 Locality

The approaches in this section are directed towards minimizing cross-transitions, either by improving locality, or by eliminating unnecessary SEND operations.

Lerda and Sisto improve the static partitioning function of the Murphi verifier [21] by making use of the fact that many transitions involve a change in only one of the concurrent processes of the system [43]. Thus, only a small fraction of transitions is a result of any of the processes, and can minimize cross-transitions. One of the processes is therefore selected as *designated* (as in the expression "designated driver"), and a table is used to map the local states of the process to different nodes. Any transition that changes this local state is a cross-transition, and the new state is sent to the owner node, based on the information in the table. All other transitions — those that do not change the state of the designated process — stay on the same node.

Similarly, the distributed version of UPPAAL, a model checker for timed automata [8], implements partitioning strategies that only hash over a part of the state. In addition, the algorithm only stores loop-entry points instead of all states. Non-loop entry points are explored locally, because they are not stored.

Two more solutions based on locality are used by Behrman [7] for timed automata. The first method again calculates the owner node over only part of the state. The second method explores transient states (states not necessary to explore to ensure termination) locally since they will not be stored remotely.

Zeus [9] is the distributed implementation of the timed model checker *KRONOS* [19]. Nodes

request regions associated with foreign states if these states are needed in the fix-point calculation. A pushing and polling schema is used to reduce communication. A node requesting a foreign state will in addition request all regions from the state's owning node it needs. In turn, it will send all regions that the same remote node requires. Some models are not scalable due to the asynchronous behaviour of Zeus and this is remedied in a synchronous version [10]. This version iteratively performs the fix-point calculation and at the end of every step exchanges regions. However, time is wasted on synchronization due to uneven workload on nodes leaving some nodes idle. This is solved by dynamically migrating regions at every step of the iteration; first by predicting the workload for the next iteration, and secondly by redistributing the work. Regions are thus migrated to new owners based on the difference between the previous partition and the current one. The normal region exchange follows at the end of every step [9].

In the model checking algorithm by Orzan et al. [24] the state space is approximated by using abstract interpretation, resulting in a system with fewer cross-transitions. The connectivity of the approximated abstract states is used to predict the connectivity of the concrete states. Cycles in the abstract states correspond to cycles in the concrete states and can each be handled by a single node. However, this process requires generation of the full state space before initiating the model checking phase, and the states must be written to file.

Chapter 4

Design and implementation

This chapter is divided into two parts:

- first we explain the design and implementation of the model checking algorithm used, and
- second, starting at page 51, we introduce a technique to reduce communication, and refer to it as *local search*. The technique is implemented on top of the algorithm.

A detailed apprehension of the distributed algorithm is not important to the understanding of local search, but distributed model checkers differ from one to the other. Thus an explanation of the algorithm will help clarify how local search is used in this case. This also allows for some of the algorithm components to be challenged with alternative implementations. These include the following:

- *queuing*: queues are used to temporarily hold states received from remote nodes and these queues grow exceedingly long in some cases. Strategies are proposed to reduce their lengths.
- *partitioning*: the partition function used does not cater for locality, and a different partitioning strategy that takes locality into account is introduced.

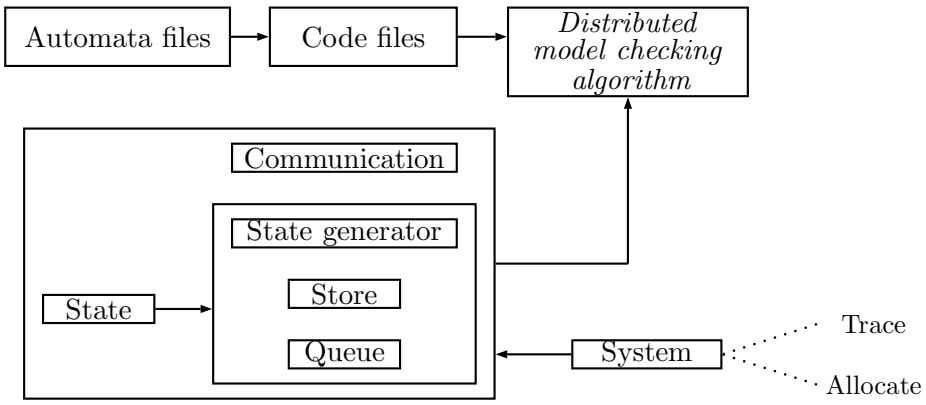


Figure 4.1: Component layout for the implementation.

Section 4.1 deals with the distributed algorithm, Section 4.2 introduces local search, and finally in Section 4.2 extra implementation details are given.

4.1 The distributed algorithm

The following choices were incorporated into the algorithm:

- correctness properties are expressed in *LTL*,
- the algorithm is *automata-based*,
- *local on-the-fly* search is used, and
- states are represented *explicitly*.

Since model checkers are time-intensive programs, the algorithm was implemented in the C language. As a mid-level language it allows fast manipulation of low-level elements such as bits and addresses, and it adds the necessary high-level structures required in writing complex software such as model checkers. C is widely available and there are specialized libraries for communication.

The algorithm runs on every network node and the algorithm components are depicted in Figure 4.1. The model checker calls upon the different components to perform specific functions. The

State Generator is responsible for the generation of states. States are stored or cached in memory through the *Store* component. The *Queue* component is used in managing the waiting queues. Communication is implemented in the *Communication* component, and makes use of TCP/IP sockets. The global *System* component handles output messages and memory allocation. Lastly, global state data are kept in the *State* component which also handles state manipulation.

One node is chosen as the *manager* and is responsible for obtaining the input files (the system model and the correctness property) and distributing them to the rest of the nodes. These files are read in automata form, and then interpreted into code files that can be used by the model checker for state generation and cycle detection. We elaborate on the code files in Section 4.3.1. The manager does not participate in the model checking process, only in detecting termination and collecting results.

Interaction between nodes

The manager node only runs one manager thread, but every other node runs two threads in parallel:

- a *worker* thread that explores the states and sends foreign states to their owners, and
- a *receiver* thread that receives state from other nodes and queues them for exploration by the worker thread.

Having two threads allows states to be received and queued without interrupting the exploration. The receiver thread inserts all incoming states into a waiting queue R. The worker thread queues local states into a waiting queue W. When W is empty the worker swaps the contents of R and W.

Figure 4.2 illustrates the communication schema between threads. A socket connection is established between every node's receiver thread and all of the worker threads on the other nodes. Every node is assigned a unique *id* during initialization and one of these ids is chosen for every state during partitioning. The worker sends a foreign state to the receiver on the node with the state's calculated id. The manager node connects to all of the other nodes.

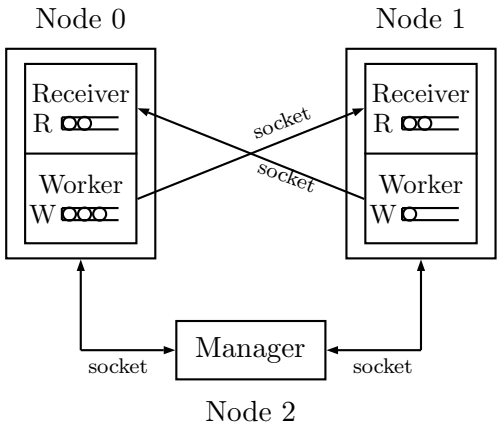


Figure 4.2: Interaction between nodes.

The rest of the section explains the algorithm in more detail and is directed towards simple state enumeration (Sections 4.1.1 to 4.1.6). Finally, Sections 4.1.7, 4.1.8, and 4.1.9 extend the algorithm towards three model checking algorithms.

4.1.1 Algorithm in detail

For state enumeration a BFS is run on every node. The pseudo-code of the algorithm is shown in Figure 4.3 and is structured similarly to the distributed CVWY algorithm [30]. The worker thread starts the WORKER() function (lines 5-21) while the receiver thread starts the RECEIVER() function (lines 1-4). The receiver polls for incoming messages (such as states) and inserts them at tail of R (line 3).

The worker first checks if the node is the owner of the initial state with the OWNER() function, and if so, inserts the state into W (lines 6-7). In the WHILE-loop (lines 9-21) messages are removed from the head of W (line 12) and then handled according to the type of the message. In the case of a state, its children are generated in EXPLORECHILDREN() (line 17). This function is outlined in lines 22-28 and every state is stored, or *marked*, in line 22. Before a state is explored it is first checked against the store in line 17.

The owners of states are found in line 24, and if a state is foreign it is sent to its owner (line 27). Otherwise the state is queued in W (line 24). Sent states are cached (line 26) to avoid their

```
   Receiver()
1  repeat
2    x ← Receive()
3    R.Enqueue(x)
4  until x = DieToken

   Worker()
5  ŝ ← InitState()
6  if Owner(ŝ) = Me then
7    W.Enqueue(ŝ)
8    W.Enqueue(TermToken)
9  while true do
10   token ← Empty
11   while ¬W.IsEmpty() do
12     msg ← W.RemoveFirst()
13     if msg = DieToken then terminate
14     else if msg = TermToken then token ← msg
15     else
16       s ← msg
17       if ¬Marked(s) then ExploreChildren(s)
18   endwhile
19   if token ≠ Empty then HandleToken(token)
20   W.Refill(R)
21 endwhile

   ExploreChildren(s)
22 Mark(s)
23 for s′ in s.Children() do
24   if Owner(s′) = Me then W.Enqueue(s′)
25   else if ¬Cached(s′) then
26     Cache(s′)
27     Send(s′) to Owner(s′)
28 endfor
```

Figure 4.3: The distributed BFS algorithm.

unnecessary retransmission. When W is empty the worker refills it in line 20 with messages queued in R.

Termination detection

We implement a token-based approach for termination detection, as used in the CVWY algorithm [30]. A token is circulated through the network and can be of two types: TERM or DIE. The token starts as a TERM token (line 8) and once no more messages are in transit, a node may change the type to DIE inside the HANDLETOKEN() function in line 19. Every node will receive the DIE token and send its results to the manager before terminating (line 13). Apart from the type, the token message also consists of an *id* of the node that initiated the last termination cycle. The id is used in combination with a *colour* variable on every node to determine if the token type should change, but these details are not important here.

Manager thread

The manager thread starts the *Manager()* function which is depicted in Figure 4.4. The manager polls for either a *result* message or an *error* message and only performs its tasks, other than polling, when no more states are in transit and exploration is finished. Thus the manager is usually one of the active exploration nodes. Polling is done with the *NetProbe()* function (line 4) which returns the message type and also the id of the sending node, *who*. This function blocks until a message is found on the list of sockets, which in this case is the socket connections between the manager and the other nodes. The actual reception of the messages is done independently.

When a result message is received from a node (line 5), it means that the node has finished its exploration and is ready to send all of its results to the manager. The counter i in lines 3 and 7 ensures that the manager only terminates after receiving results from every node; *num_nodes* is the number of nodes participating. Once results have been received from all of the nodes, the manager displays the results and terminates. The result messages are divided into subtypes for specific results. These are not important to the design of the algorithm and are therefore not discussed here.

```
1 void Manager() {
2   int msg_type, who, e_type, error_found = 0, i = 0;
3   while (i < num_nodes) {
4     msg_type = NetProbe(&who, sockets, num_nodes);
5     if (msg_type == RESULT) {
6       /* read results from node who */
7       i++;
8     }
9     else if (msg_type == ERROR) {
10      error_found = 1;
11      ReceiveError(&e_type, who);
12    }
13  }
14  /* report the results and terminate */
15 }
```

Figure 4.4: Function Manager().

In the case of an error message (line 11) the manager will display the type of the error along with the results. Types of errors are, for example, deadlocks or runtime errors.

Receiver()

A more thorough outline of the *Receiver()* function is given in Figure 4.5. The receiver also polls (line 5) for incoming messages on the array of sockets *in_sock*. This is a list of socket connections between the receiver and each of the worker threads on the other nodes. The receiver can receive a *state*, a *token*, or an *error* message. States and the token are queued in R (lines 6-8 and 10-16). The receiver will exit the loop ($busy = 0$) and terminate in the case of a DIE token. When an error is found, a flag is set (line 20) which will stop any more messages from being queued and normal termination will ensue.

```
1 void Receiver() {
2   int msg_type, who, e_type, id, error_found = 0, busy = 1;
3   State s;
4   while (busy) {
5     msg_type = NetProbe(&who, in_sock, num_nodes);
6     if (msg_type == STATE) {
7       ReceiveState(&s, who);
8       if (!error_found) /* lock R, enqueue s and unlock R */
9     }
10    else if (msg_type == TERM) {
11      ReceiveToken(&id, who);
12      /* lock R, enqueue token and unlock R */
13    }
14    else if (msg_type == DIE) {
15      ReceiveToken(&id, who);
16      /* lock R, enqueue token and unlock R */
17      busy = 0;
18    }
19    else if (msg_type == ERROR) {
20      error_found = 1;
21      ReceiveError(&e_type, who);
22    }
23  }
24 }
```

Figure 4.5: Function Receiver().

4.1.2 Communication

The communication component is implemented on a TCP/IP socket system which makes use of asynchronous communication. Read and write operations are blocking at the system level. The write operation is blocked until there is enough buffer space to accept some or all of the

data being sent. The read operation is blocked until data can be read from the internal buffer. This applies independently to threads; for example, a blocked call on the worker thread will not block the receiver thread. All communication makes use of the following functions, each taking as argument the socket id (or file descriptor) over which data is sent:

- *void NetWrite(int fd, void* buffer, size_t count)*: send *count* bytes from the buffer to the socket descriptor *fd*. Each write is blocked until some or all of the *count* bytes are delivered to the kernel buffer. When fewer than *count* bytes are sent, an error is reported.
- *void NetRead(int fd, void* buffer, size_t count)*: receive *count* bytes on the socket descriptor *fd* and puts them into the buffer. A read call will be blocked until there are bytes available to be read. If fewer than *count* bytes are read, the function will retry a number of times after which an error is reported.

We implement the sending and receiving routines on top of these functions. Each send routine includes the type of the message, and *out_sock* refers to the socket connections between the worker and each of the receiver threads on the other nodes. The send routines are:

- *void SendState(State s, int node)*: send the message type (*STATE*), the state size, and the state vector (see Section 4.3.2) to the node with index *node*.
- *void SendToken(int t_type, int id, int node)*: send the token type *t_type* (DIE or TERM) and the token id to the node with index *node*.
- *void SendError(int e_type, int node)*: send the message type (*ERROR*) and the type of error *e_type* to the node with index *node*.

The receive routines do not receive the message types. Instead a function *NetProbe()* makes use of the *select* system call to search for an incoming message on a list of sockets. This function always precedes a receiving routine and the type is then used to determine what receive routine to call. The receive routines are:

- *void ReceiveState(State* s, int node)*: receive state *s* from the node with index *node*.
- *void ReceiveToken(int* id, int node)*: receive the token *id* from the node with index *node*.

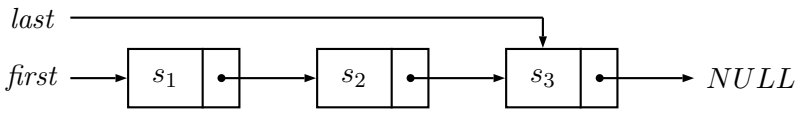


Figure 4.6: The queue structure for both R and W.

- *void ReceiveError(int* e_type, int node)*: receive the error type *e_type* from the node with index *node*.

4.1.3 Queues

Messages are added at the tail and removed from the front of both the worker queue W and the receiver queue R. Since these messages contain either a state or a token, each queue entry consists of a token id, a message type, a state, and a pointer to the next entry in the queue. Both queues have a *first* and *last* pointer to the head and the tail of the queue respectively, as illustrated in Figure 4.6. The queuing routines are all $O(1)$ and utilize functions from the Queue component. The queuing routines are:

- *void WorkerQueueAdd(State s)*: perform a tail-insertion of state s into W.
- *void ReceiverQueueAdd(int type, State s, int id)*: perform a tail-insertion of message type *type* into R, where type is *STATE* in the case of a state message, and *DIE* or *TERM* for a token message. In the case of a token, *id* is the token id, and in the case of a state, s is the state.
- *void WorkerQueueRemove(int* type, State* s, int* id)*: remove the first item from the queue and return the type in *type*. If the type is a token the function returns the token id in *id*, otherwise it returns the state in s.
- *void WorkerQueueRefill()*: the first and last pointer fields of W are set to the first and last fields of R respectively and the fields of R are set to NULL.

Thread synchronization on R

When W is empty and messages are moved from R to W, access to R must be synchronized to avoid race conditions. We must ensure that neither the worker nor the receiver thread reads an old value of R, such as a pointer to a removed queue entry, for example. Two synchronization techniques, *mutual exclusion* and *conditional variables*, are used and we give a brief summary of each:

- **Mutual exclusion**: locks are associated with data shared by various threads to only allow one thread accessing it at a time. A thread needs to gain the *lock* to modify the shared variables and *unlock* it if done to allow other threads access to the *critical section*.
- **Condition variables**: threads *wait* on a condition variable until another thread exits the critical section and *signals* them on the condition variable to wake up, allowing one of them to enter the critical section.

Figure 4.7 shows how a mutual exclusion strategy is applied in both the Receiver() and Worker() functions. A variable *mutex* controls access to R. Only one of the two threads can enter its own critical section. Both threads will block in line 1 until the other thread unlocks *mutex* in

```
1 LOCK(mutex)
2 /* enter critical section */
3 R.ENQUEUE(s)
4 CONDSIGNAL(cond)
5 /* exit critical section */
6 UNLOCK(mutex)
```

(a)

```
1 LOCK(mutex)
2 /* enter critical section */
3 if R.EMPTY() then
4     CONDWAIT(cond, mutex)
5 Q.REFILL(R)
6 /* exit critical section */
7 UNLOCK(mutex)
```

(b)

Figure 4.7: The working of *mutex* and conditional variable *cond* to synchronize access to R in (a) the receiver thread and (b) the worker thread.

line 6 in (a) and line 7 in (b). The receiver thread inserts a new message (line 3 in (a)) and the worker thread moves the contents of R into W (line 5 in (b)).

To stop the worker from continually re-entering the critical section when both R and W are empty, the condition variable *cond* is used to delay the worker in line 4 of (b) and allow the receiver to enter its critical section. When a state arrives the receiver can insert it into R and will then signal in line 4 of (a) to the worker that it can continue to swap the queues.

Queue-length reduction strategies

Received states are always directly inserted into R even if they are already queued or stored. Thus, queues can become very long, and due to timing issues may lead to imbalances in queue lengths from node to node. This is illustrated in Figure 4.8. As an example we run our algorithm on Lamport's mutual exclusion model with 6 processes. The queue lengths range between 17 647 and 477 077. Memory depletion is a possibility on nodes with very long queue lengths.

We apply three strategies to reduce the queue lengths. The first two are implemented in the receiver thread and take the contents of the store into account when adding new states to the queue. The third strategy, similar to the idea described by David et al. [18], applies to both the worker and receiver, and never allows a state to be queued more than once in either queue. These strategies are:

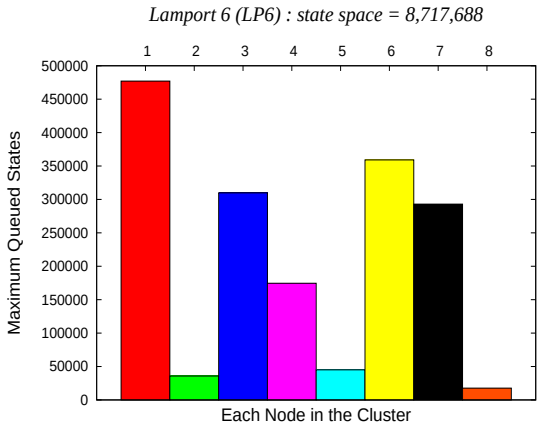


Figure 4.8: R is unbalanced.

(a) In the *store* strategy received states are added to the queue only if they are not already present in the store.

(b) In the *limit* strategy, which we introduce here, an upper bound L is applied to the length of R. Every time this limit is reached all states in the queue are rechecked: if a particular state appears in the store, it is removed from the queue. Only "new" states remain in the queue. The following rules are applied:

1. L is set to U (a user-defined value) during initialization.
2. When a state is received and the number of states in the receiver queue is greater than L:
    1. insert the new state into R,
    2. for every state s in R: if s is in the store remove it from R, and
    3. $L = L \times 1.1$.
3. When R and W are swapped reset L to U.

(c) In the *hash* strategy, states are inserted in the store as soon as they are received from another node or generated locally. Every queue entry keeps a pointer to a state in the store and does not explicitly store the state itself. However, if a newly-arrived state is found to be present in the store even before its insertion, it means that the state has either been queued or explored already. In this case it is not inserted again, neither in the store, nor in the queue (as a pointer) — thus avoiding doubly-queued states and reducing queue lengths. This strategy is called "hash" since the store is implemented as a hash table (see Section 4.1.5). The state exploration is not affected by this strategy, except for when states are compared against the store.

Synchronization on the store is required in the hash strategy, because both threads can modify the store. A mutual exclusion lock *store_mutex* is associated with the store. The limit strategy locks access to R in step 2.1, and also in the removal process in step 2.2 once a stored state is found in R.

In the store and limit strategies, line 3 in Figure 4.3 is modified, and in the hash strategy every state insertion into R or W is preceded by a store insertion.

4.1.4 Partitioning

The state space is divided amongst the nodes by a hash function that randomly calculates an owner for every state [21, 26]:

- *int Owner(State s)*: return for state s a unique owner index between 0 and $N - 1$ for a network with N nodes.

The owner of a state is calculated based on the contents of the state. Therefore a state's owner is not "remembered".

Memory is used as efficiently as possible, because the states are uniformly distributed. The function can be written more specifically as follows:

$$Owner(s) = (f(s)\ mod\ P)\ mod\ N$$

A value is calculated (by fast bitwise operation) for every state s, and *(f mod P)* divides the state space into P chunks. These chunks are then divided amongst the N nodes.

P is a prime number, because it minimizes the chance that patterns repeat in the range of values *f(s)*. For example, if we divide the range {200, 204, 208, 212, ... 800} by a non-prime 132 and a prime 131, it results in 68 collisions for the non-prime and 0 collisions for the prime.

This randomized partition function can be replaced by an alternative, distance partition function that we introduce next.

Distance table

Every node keeps a *distance table*, and for every state a distance value is calculated. We subtract this value from the distance value of the initial state in the state space. If these distances are representative of the changes made in the values in the state, we can group together states with similar distances and assign them to the same node knowing that they are close to each other in the breadth-first tree hierarchy. Thus, fewer states are foreign and sent between nodes.

The distance table, illustrated in Figure 4.9, is a set of ranges, each assigned to a node. If r is the range length of every bucket, the first bucket includes distances 0 to $r - 1$, and the second

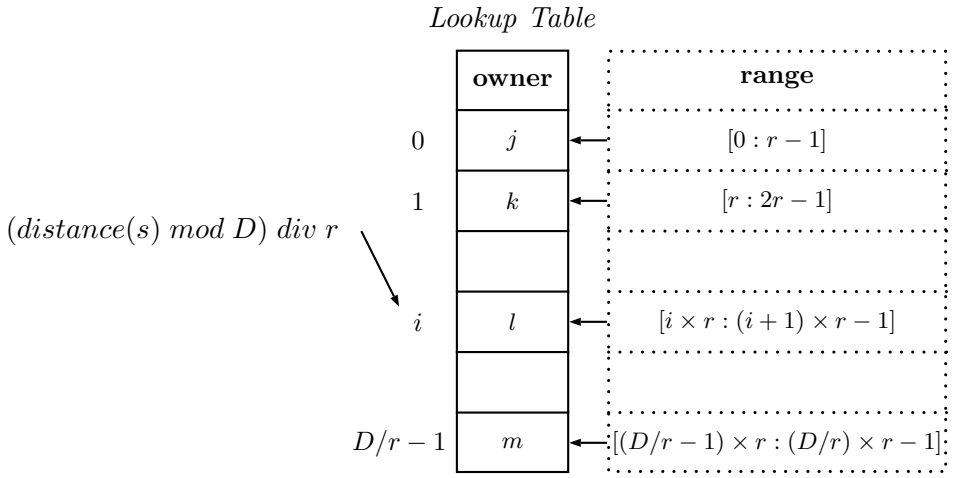


Figure 4.9: The distance table: states are mapped to distances and distances to owners (node indices). The table on the right displays the value range of every bucket in the distance table, and r is the range size.

bucket includes distances from r to $2r - 1$, and so on. A maximum distance D is selected and the size of the table is D/r. The *Owner(s)* function now looks like the following:

$$Owner(s) = table[(distance(s)\ mod\ D)\ div\ r].owner$$

When an owner is not defined yet for the bucket to which the *distance(s)* belongs, the node sends a query to the manager node that then calculates it. It does so by selecting the node with the least amount of work, based on the following weighted function:

$$Load\_on\_node_i = (|states_i|) \times w_1 + (|R_i|) \times w_2$$

Weights $w1$ and $w2$ define the importance of the number of stored (explored) states and the number of queued (unexplored) states. The load is based only on memory balance if $w1 = 1$ and $w2 = 0$, and based only on temporal balance if $w1 = 0$ and $w2 = 1$. The manager keeps a small table to store the load amount of every node. Every node updates this value by sending its current load to the manager after exploring every x states. Once the owner is selected, it is broadcasted to all of the nodes who then update their own tables. Initially no bucket has an owner.

We can expect buckets to be assigned owners mostly at the start of execution and thus the

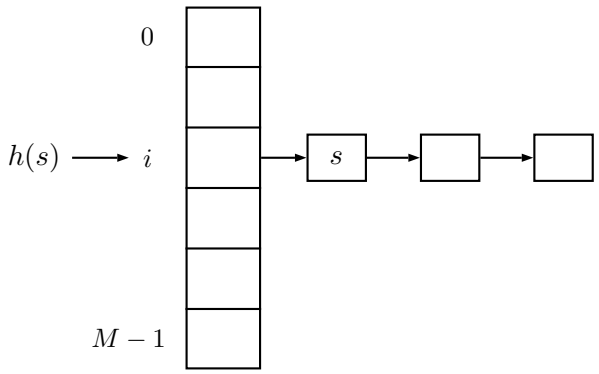


Figure 4.10: Each state is hashed to a store bucket and queued in the collision list.

value of x can be increased to avoid the nodes from constantly updating the manager with the progress when buckets are already filled.

The success of this partitioning strategy is based on the size of r and the type of the distance function. For a large r, states will be well grouped, but this may result in vastly unbalanced node partitions when a lot of states fall only in a few buckets. For a small r the number of cross-transitions will increase.

Our distance function interprets the values of the state, from left to right, as one large distance value. Since the size of states ($size(s)$, i.e., the number of values in the state vector) of some models is very large, the distance value will be too large and undefined. Thus we approximate this value with a smaller 5-valued number in which every $size(s)/5$ state values are combined into one value.

We combine these values in a number of different ways, for example, with a bitwise OR, a bitwise XOR, or addition.

4.1.5 Store

Every node records the states in its partition in a store in memory. The stores are implemented as hash tables and the buckets are ordered collision lists. Figure 4.10 illustrates this, and also that every state is assigned by the hash function, $h(s) = g(s) \; mod \; M$, to a bucket and then inserted, or marked, into the corresponding bucket queue. The hash function is the same as

the partition function in the previous section, and we choose M, the size of the hash table, as the prime number P.

The store interface entails two functions, referred to as MARK(s) and MARKED(s) in Figure 4.3:

- *void StoreInsert(State s)*: store state s in the store by hashing s to a bucket and inserting s into the corresponding collision list.
- *int StoreLookup(State s)*: check if state s is in the store by hashing s to a bucket and searching for s in the collision list.

4.1.6 State caching

On every node we implement two separate caches, but in the same way. The first cache is called the *sent* cache and in it we record sent states to avoid redundant sends. The second cache is called the *search* cache, and its function is explained later in Section 4.2 where it is applicable. The cache interface entails one function that combines state lookup and insertion (CACHE() and CACHED() in Figure 4.3):

- *int SentCacheInsert(State s)*: check if state s is cached in the sent cache, and if not, insert it. A 1 is returned if it is already cached, else a 0. If the cache is full an old state is replaced.
- *int SearchCacheInsert(State s)*: check if state s is cached in the search cache, and if not, insert it. A 1 is returned if it is already cached, else a 0. If the cache is full an old state is replaced.

Both caches are implemented as hash tables with collision lists like the store in the previous section. When the cache is full, we use a *rotating bucket* principle to select an old state for removal. An index value is kept and initialized to 0 (the first bucket). In the event of a full cache, a state is removed from the head of the list associated with the bucket at index *index*. The index is incremented by one, and empty buckets are skipped. The old state's memory is recycled and used for the new state which is then added to the tail of its own bucket list.

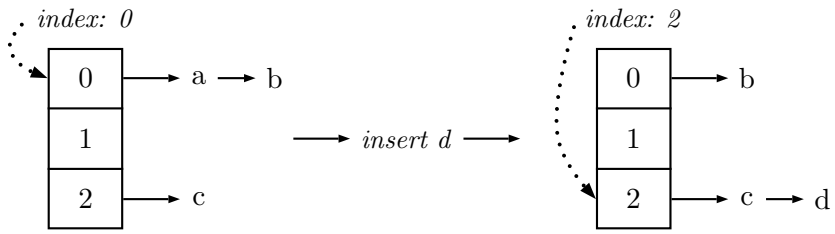


Figure 4.11: State d is inserted into a full cache using the rotating index method.

Figure 4.11 demonstrates when state d is cached into an already full cache. State d is hashed to bucket 2. The index is at 0 and thus state a is removed. The index is then incremented to 2, because bucket 1 is empty. Finally, state d is inserted at the end of the list in bucket 2.

The next three sections show how the state enumeration algorithm is extended into three known model checking algorithms. The first two algorithms are both versions of the distributed CVWY algorithm [30], while the third is a bounded model checking algorithm [41].

4.1.7 Naive distributed CVWY algorithm

The algorithm is outlined in Figure 4.12. The algorithm operates in a breadth-first manner and handles states in pairs, formed by a root state and the current state. When an accepting state is generated and the root is empty, a new path is followed in which the root is set to the accepting state in order to start a nested search (lines 14 and 17). When the root and the state is the same, a path exists back to the root, an accepting cycle is found (line 10), and the algorithm reports the counterexample.

Every state may be associated with many roots and the store is extended so that every entry contains a list of roots, as illustrated in Figure 4.13. In addition, the *SendState()* and *Receive-State()* functions are extended to send and receive both the state and the root, and the waiting queues list pairs of states.

The cache is extended in the same way by adding root lists to every state. The rotating bucket method is still used, but every pair (r, s) in the cache counts as one entry. For example, if a state is cached with four roots it adds four to the size of the cache. However, instead of

removing only a single state when the cache is full, we now remove an entire collision list. When only the root r is new and the new state s is in the list to be removed, then we only replace one of the old roots of s with r.

The downside of this algorithm is that every accepting state initiates a new nested search. In the worst case this leads to a quadratic number of state pairs that needs to be investigated. The next algorithm improves on this by only starting nested searches with back-level edges.

```
   Worker()
 1 ŝ ← InitState()
 2 if Owner(ŝ) = Me then Q.Enqueue((−, ŝ))
 3 while true do
 4   while ¬W.IsEmpty() do
 5     (r, s) ← W.RemoveFirst()
 6     if ¬Marked(r, s) then ExploreChildren(r, s)
 7   endwhile
 8   W.Refill(R)
 9 endwhile

   ExploreChildren(r, s)
10 if r = s then report counterexample
11 Mark(r, s)
12 for s′ in s.Children() do
13   if Owner(s′) = Me then
14     if (r = −) ∧ s.Accepting() then W.Enqueue(s, s′)
15     W.Enqueue(r, s′)
16   else
17     if (r = −) ∧ s.Accepting() then
18       Send(s, s′) to Owner(s′)
19     Send(r, s′) to Owner(s′)
20 endfor
```

Figure 4.12: The distributed naive CVWY algorithm.

4.1.8 Improved distributed CVWY algorithm

This algorithm does not work in combination with local search (which we describe shortly), but we nevertheless describe how it works and we show in Section 4.2.3 why cycles are missed when we try to add local search.

Based on the observation that any cycle must contain at least one back-level edge from a deeper to a shallower state (or to a state of equal depth), a heuristic is implemented that only initiates new searches when back-level edges are encountered. The improved algorithm is depicted in Figure 4.14.

The message pairs (r, s) are extended with the suspected depth d of s, and a "frozen" flag f that indicates that the search has passed through a back-level edge. When the flag is true the root is called "frozen". Once the flag has been set, in line 8, a new nested search cannot be started in line 15. This preserves the integrity of the breadth-first search tree constructed during the search, because a counterexample is reported only when a cycle is found with at least one accepting state reachable from the initial state. Secondly, if the state space contains one or more accepting cycles the algorithm will eventually report the error. The MARK() function in line 14 will not duplicate ($-$, s) if r is blank, because it is already handled in line 6.

This method uses a store in the same way as in Figure 4.13 as was used for the naive method. However, the blank root state is not explicitly stored in the root list, because a state is always first stored with a blank root (line 6) which is not always the case in the naive method. The

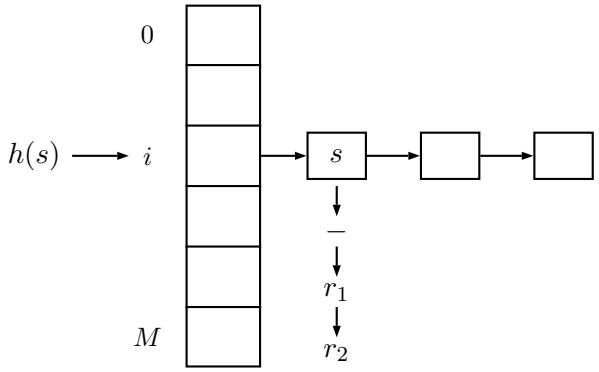


Figure 4.13: In the store each state keeps a list of roots.

depth d is recorded with every state. The *StoreLookup(r, s)* function returns the depth of s if both r and s are present in the store, otherwise it returns -1. The waiting queues R and W list entries of the form $(r,\ s,\ d,\ f)$.

The *SendState()* and *ReceiveState()* functions send and receive both the state and the root, but also the associated d and f. The cache is implemented in the same way as with the naive algorithm, but the depth d is saved with every state.

```
   Worker()
 1 ŝ ← InitState()
 2 if Owner(ŝ) = Me then W.Enqueue((−, ŝ, 0, false))
 3 while true do
 4   while ¬W.IsEmpty() do
 5     (r, s, d, f) ← Q.RemoveFirst()
 6     if ¬Marked(−, s) then s.Depth ← d ; Mark(−, s)
 7     else if Marked(r, s) then continue
 8     else if s.Depth ≤ d then f ← true
 9     ExploreChildren(r, s, d, f)
10   endwhile
11   W.Refill(R)
12 endwhile

   ExploreChildren(r, s, d, f)
13 if r = s then report counterexample
14 Mark(r, s)
15 if ¬f ∧ s.Accepting() then r ← s
16 for s′ in s.Children() do
17   if Owner(s′) = Me then W.Enqueue(r, s′, d + 1, f)
18   else
19     Send(r, s′, d + 1, f) to Owner(s′)
20 endfor
```

Figure 4.14: The distributed improved CVWY algorithm.

4.1.9 Distributed bounded nested BFS algorithm

A BFS is run on every node and when an accepting state is encountered a sequential nested BFS is started locally. The algorithm is outlined in Figures 4.15 and 4.16. To every state is

```
   Receiver()
 1 repeat
 2   (s, d) ← Receive()
 3   if d < b then R.Enqueue(s, d)
 4 until x = DieToken

   Worker()
 5 ŝ ← InitState()
 6 if Owner(ŝ) = Me then W.Enqueue((−, ŝ, 0))
 7 while true do
 8   while ¬W.IsEmpty() do
 9     (s, d) ← W.RemoveFirst()
10     if ¬Marked(s) ∨ d < s.Depth() then ExploreChildren(s, d)
11   endwhile
12   W.Refill(R)
13 endwhile

   ExploreChildren(s, d)
14 if s.Accepting() then
15   NestedBFS(s, d)
16 s.Depth ← d ; Mark(s) ; d ← d + 1
17 for s′ in s.Children() do
18   if Owner(s′) = Me then
19     if d < b then W.Enqueue(s′, d)
20   else
21     Send(s′, d) to Owner(s′)
22 endfor
```

Figure 4.15: The first part of the bounded nested BFS algorithm, continued in Figure 4.16.

```
   NestedBFS(s, d)
23 goal ← s
24 steps ← b − s.Depth()
25 N.Enqueue(s, 0)
26 while ¬N.IsEmpty() do
27   (s, d) ← N.RemoveFirst() ; d ← d + 1
28   for s′ in s.Children() do
29     if ¬MarkedInNestedSearch(s′) then
30       if s′ = goal then report counterexample
31       MarkForNestedSearch(s′)
32       if d < steps then N.Enqueue(s′, d)
33   endfor
34 endwhile
```

Figure 4.16: The last part of the bounded nested BFS algorithm, continued from Figure 4.15.

attached a suspected BFS depth d. This depth is restricted by an upper bound, b (lines 3 and 19), and a state is not queued when its depth exceeds b. In the nested search, *steps* (lines 24 and 32) is used to the same effect as b and is the difference between b and the depth of the accepting state that initiated NestedBFS() in line 15.

It is possible to process the same state at different depths; during the exploration the minimal depth of every state is remembered. When an already visited state is found at a smaller depth (the second condition in the if-statement in line 10), it is processed again.

An additional queue N stores states found in the nested search in NestedBFS(). The depth d is not queued here, because states explored sequentially in breadth-first manner are ordered by their true depths. In addition, an extra store is used to record states found in a nested search, and this store is cleared after every nested search. If the store is not cleared, states from an earlier nested search may be confused with states in a later nested search, and possible cycles may be missed. Moreover, once a local nested search is complete, the information is never used again.

The store remains the same as in Section 4.1.5, except that every state is stored along with its suspected depth d. The waiting queues also store the depth d, and the *SendState()* and *ReceiveState()* functions send and receive both a state and d. The *StoreLookup(s)* function now returns the depth of s if it is present in the store, otherwise it returns -1. The cache also remains unchanged, except that d is cached with every state.

4.2 Local search

Local search is a technique implemented to reduce cross-transitions (and thus improve locality). It is introduced in Section 4.2.1 with the state enumeration algorithm, and then employed with the three model checking algorithms in Sections 4.2.2, 4.2.3, and 4.2.4. Finally the implementation of local search is dealt with in Section 4.2.5.

```
   Worker()
1  while true do
2    receiveWork(s)
3    if marked(s) then continue
4    mark(s)
5    for s' in s.children() do
6      Send(s') to owner(s')
7    endfor
8  endwhile
```

(a)

```
   LocalSearchWorker()
1  while true do
2    receiveWork(s)
3    ExploreChildren(s, depth)
4  endwhile

   ExploreChildren(s, dep)
6  if marked(s) then return
7  mark(s)
8  for s' in s.children() do
9    if dep > 1 then ExploreChildren(s', dep − 1)
10   else Send(s') to owner(s')
11 endfor
```

(b)

Figure 4.17: Distributed state exploration (a) without and (b) with local search.

4.2.1 Local search and state enumeration

Figure 4.17 shows the original algorithm on the left (see Section 4.1.1) and the same algorithm "enhanced" with local search on the right. The new algorithm is based on *depth*, which specifies how deep the local search will go; if $depth = 1$, then the two algorithms are exactly the same.

However, a value of $depth > 1$ introduces a crucial difference. Whereas algorithm (a) only explores its own states, the local search algorithm (b) explores all the states it encounters before reaching the *depth* limit. Because it is prepared to explore foreign states, it eliminates a large number of SEND operations. In this way, the balance between state generation and communication is improved.

The disadvantage of this scheme is that it may lead to redundant work: the same state may be explored by many different nodes. This situation is illustrated in Figure 4.18. On the left is the original state graph, where every state is numbered and labeled by its owner, either A or B. The next three columns show how state exploration is distributed among two nodes, A

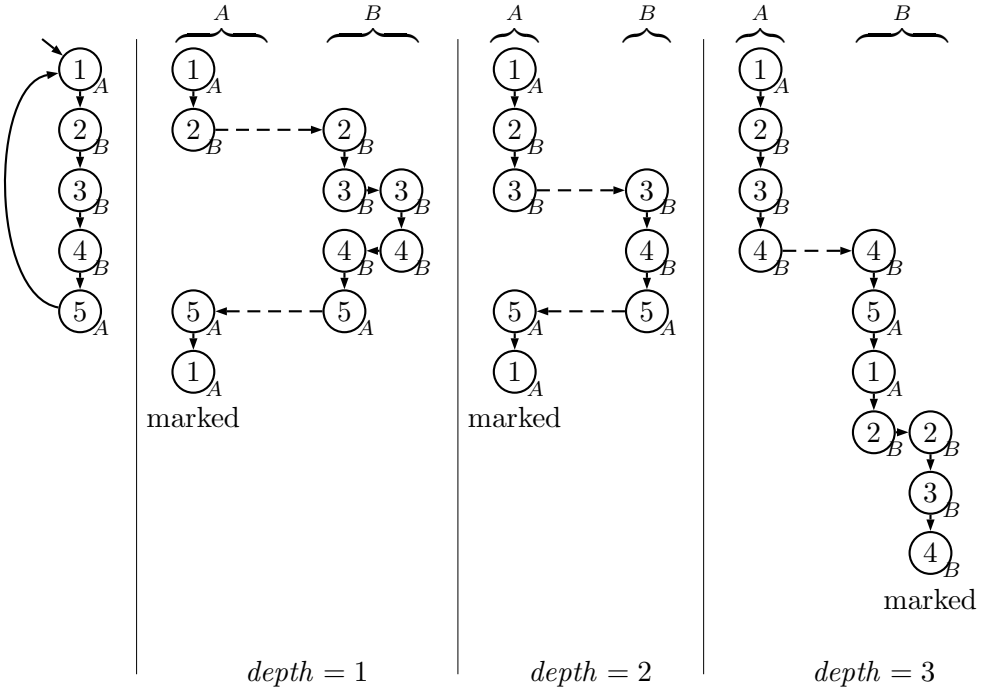


Figure 4.18: Local search can be both advantageous and disadvantageous: for the state graph on the left, states 1, 2, and 3 will be explored redundantly when $depth = 3$, but when $depth = 2$, state 2 is never sent to its owner.

and B, for depths 1, 2, and 3. Vertical arrows are used when a child state is generated and horizontal arrows when a state is sent to another node. The horizontal arrows are solid when a node sends a state to itself, and dashed when a node sends a state to a different node. In the $depth = 3$ column, states 1, 2, and 3 are explored by both nodes A and B.

On the other hand, this disadvantage is offset by the fact that some of the states may be explored only once, without the overhead of being sent to their owners. In the $depth = 2$ column, state 2 is explored by node A, even though it belongs to B. The latter never explores state 2, and we save the sending of the state.

For small values of *depth*, the improvement in communication is not significant. When the value of *depth* is large, we might expect an increase in the amount of redundant work. There is also the danger that nodes may have to idly wait for others when the computation time is too large. In practice, there is a tradeoff between too small and too large values for *depth*; the size and nature of the model plays an important role in the optimal depth of the local search.

The correctness of state enumeration with local search is guaranteed if all states are reached. This is so, because the only difference between exploring a foreign state and a local state locally, is that the foreign state is not placed in the store of its owner, which may lead to redundant visits. Checking of safety properties and deadlocks remain unchanged from the case in which local search is not used.

There are different ways of recording both local and foreign states during the local search, and we explore this next.

The cache/store/ignore option

Consider Figure 4.19. The local search reaches local states L and L', and foreign states F and F'. States L' and F' are generated when *depth* is reached and the local search ends. However, states L and F can be:

1. immediately placed in the queue W and not explored further (for the moment),
2. explored but not marked in the store,

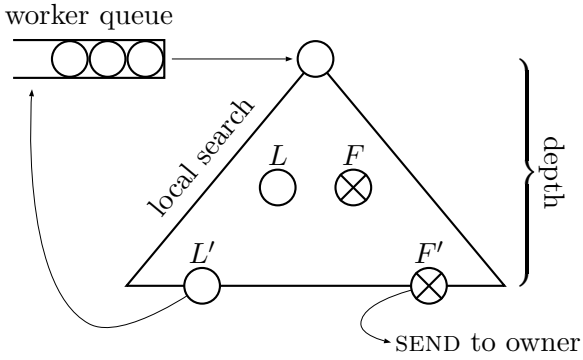


Figure 4.19: Different ways of handling local and foreign states.

3. explored and marked, or

4. explored and cached in a special cache (the *search* cache) dedicated to local searches.

Since the worker queue only holds local states, option 1 only applies to local states. It is difficult to predict the difference option 1 makes against the case in which it is not used. Figure 4.20 depicts two scenarios; in (a) option 1 fares better and in (b) it fares worse. To remain consistent with the algorithm pseudo-code, *depth* counts down towards 1 in both cases.

In Figure 4.20 (a) $depth = 2$, and local states are labeled with an L and foreign states with an F. Here a SEND is avoided, because state 2 is queued and, when removed again, starts a new local search in which state 3 is explored locally. If state 2 is not queued state 3 is sent to its owner node.

However, option 1 also increases redundant work, as demonstrated in Figure 4.20 (b). Local states are again labeled with an L and foreign states with an F. When option 1 is not used and $depth = 3$, five foreign states are sent to their owning nodes. On the other hand, with option 1 these foreign states are explored locally in local searches that start from the three local states. In this scenario the node performs extra work, and so do the other nodes. The use of option 1 forces all local states found during a local search to start their own searches and this increases the number of foreign state explored locally.

Options 2 and 3 — whether states are marked or not — are more significant. The advantage of marking states is clearly that it may avoid future redundant work (should the marked state

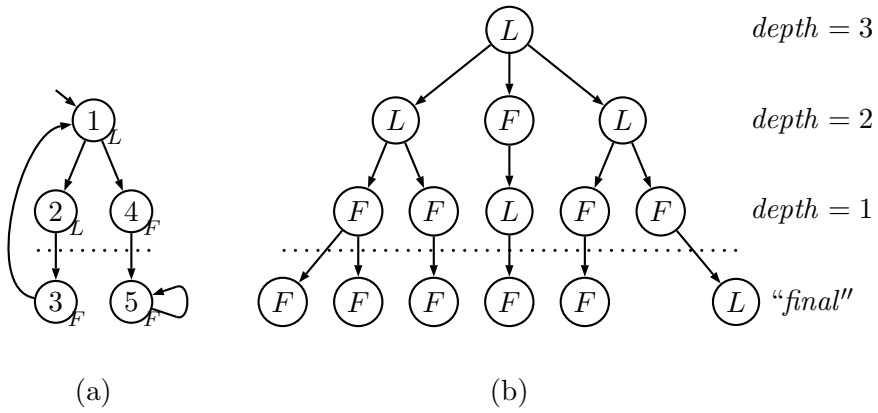


Figure 4.20: Depiction of two small state graphs during a local search: in (a) $depth = 2$, and states 3 and 5 are "final" and sent to their owners, while in (b) $depth = 3$, and 5 foreign states and 1 local states are "final". However, when option 1 is used, only state 5 is "final" in (a), and no states are "final" in (b).

be revisited), even though the marking may waste time and space. Not marking states has exactly the opposite effect: it may save time and space, but it makes revisits more expensive.

Like option 1, both options 2 and 3 only apply to local states. A node's store is reserved for local states and foreign states are only marked on their respective owners to avoid wasting aggregate distributed memory. If foreign states are recorded during a local search, we only allow them to be recorded in the cache where they may eventually be replaced. This leads to the next option.

The final choice, option 4, applies to both local and foreign states. However, local states are not added to the *sent* cache (see Section 4.1.6), because this consumes too much of the limited space available for sent states, and may even increase communication; fewer duplicate sends are avoided when the sent cache is full. This can also lead to some local states being stored twice, once in the store and once in the cache. However, caching local states in the *search* cache (this is where the search cache is used and it is used solely for local search) can save memory if the local state is encountered again before it is replaced in the cache. When replaced, a record of the state will not exist.

Foreign states can be cached in both the sent and/or search cache. If in the former, duplicate sends are avoided should the same foreign state be encountered in a local search and also as a "final" state. On the other hand, caching foreign states in the search cache frees up space in the sent cache for only sent states, because it is possible that many foreign states are only encountered during local searches and never as "final" states.

The relative performance of these options is presented in Section 5.3.5. In the next three sections local search is integrated into the three model checking algorithms.

```
   Worker()
1  while true do
2    receiveWork(r, s)
3    if marked(r, s) then continue
4    if r = s then counterexample
5    mark(r, s)
6    for s' in s.children() do
7      if (r = −) ∧ s.Accepting() then
8        Send(s, s') to Owner(s')
9      Send(r, s') to owner(s')
10   endfor
11 endwhile
```

(a)

```
   LocalSearchWorker()
1  while true do
2    receiveWork(r, s)
3    if ¬Marked(r, s) then
4      ExploreChildren(r, s, depth)
5  endwhile

   ExploreChildren(r, s, dep)
6  if r = s then report counterexample
7  mark(r, s)
8  for s' in s.children() do
9    if dep > 1 then
10     if (r = −) ∧ s.Accepting() then
11       ExploreChildren(s, s', dep − 1)
12     ExploreChildren(r, s', dep − 1)
13   else
14     if (r = −) ∧ s.Accepting() then
15       Send(s, s') to Owner(s')
16     Send(r, s') to owner(s')
17 endfor
```

(b)

Figure 4.21: The distributed naive CVWY algorithm (a) without and (b) with local search.

4.2.2 Local search and the naive CVWY algorithm

Figure 4.21 depicts the naive CVWY algorithm in Section 4.1.7 (without local search) on the left and with local search added on the right. The variable *dep* is the local search depth and while the local search remains active, $dep > 1$ (line 9), ExploreChildren() is called recursively both for $(r,\ s')$ in line 12 and $(s,\ s')$ in line 11.

The correctness of the algorithm with local search relies on a full state exploration of all pairs $(r,\ s)$ generated. This is satisfied, because the local search does not change the working of the algorithm and foreign pairs are recursively explored instead of remotely. Although it may

```
   Worker()
 1 while true do
 2   receiveWork(r, s, d, f)
 3   if ¬marked(r, −) then Mark(−, s, d)
 4   else if marked(r, s) then continue
 5   else if s.depth ≤ d then f ← true
 6   if r = s then counterexample
 7   mark(r, s)
 8   if ¬f ∧ s.accepting() then r ← s
 9   for s′ in s.children() do
10     Send(r, s′, d + 1, f) to owner(s′)
11   endfor
12 endwhile
```

(a)

```
   LocalSearchWorker()
 1 while true do
 2   receiveWork(r, s, d, f)
 3   if ¬marked(−, s) then Mark(−, s, d)
 4   else if marked(r, s) then continue
 5   else if s.depth ≤ d then f ← true
 6   ExploreChildren(r, s, d, f, depth)
 7 endwhile

   ExploreChildren(r, s, d, f, dep)
 8 if r = s then counterexample
 9 mark(r, s)
10 if ¬f ∧ s.accepting() then r ← s
11 for s′ in s.children() do
12   if dep > 1 then
13     ExploreChildren(r, s′, d + 1, f, dep − 1)
14   else Send(r, s′, d + 1, f) to owner(s′)
15 endfor
```

(b)

Figure 4.22: The distributed improved CVWY algorithm (a) without and (b) with local search.

result in redundant work, the algorithm remains correct.

4.2.3 Local search and the improved CVWY algorithm

Figure 4.22 depicts the improved CVWY algorithm (presented in Section 4.1.8) without local search on the left and with local search on the right. The variable *dep* is the local search depth in (b), and while $dep > 1$ the search continues recursively (line 12). The variable d is a state's suspected BFS depth; here d counts up while *dep* counts down.

However, some cycles may be missed with the algorithm on the right. The algorithm relies on a state to arrive twice on the same node, but with different roots. The flag will be "frozen" and a root will remain unchanged in line 10. Consider the scenario in Figure 4.23 (a) where every state in the cycle is an accepting state. We set the depth to 3 and the cluster size to 2. Pairs are explored as shown in Figure 4.23 (b). State 0 is the initial state and belongs to node A. Every local search is indicated by a dotted box that includes the pairs searched in that local search. The suspected depth of every state s is placed to the right of the pair (r, s) in brackets. The first local search, starting from ($-$, 0), will include pairs (0, 1) and (1, 2). The flag is not "frozen" and thus roots are switched with every new child, because all of the states are accepting states. Pair (2, 3) is sent to node B, because 3 belongs to B. The search

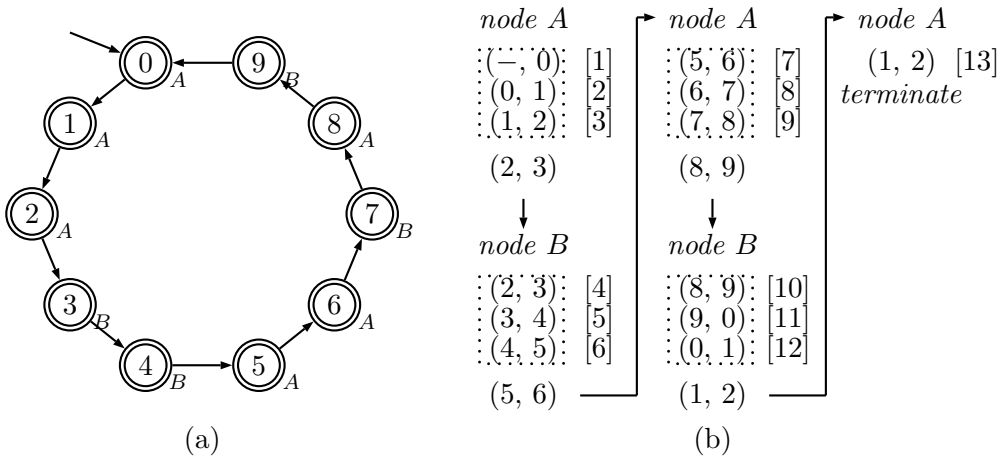


Figure 4.23: An example of when local search does not work with the improved algorithm: (a) a state graph with one accepting cycle, and (b) how two nodes run local searches (in dotted boxes) that miss the cycle.

continues to (5, 6), (8, 9) on A, and finally to (1, 2) on B. Note that (9, 0) was found in the local search on B and 0 belongs to A, meaning that we would have frozen the flag and a cycle would have been detected at (9, 9) in the algorithm on the left. However, now we send (1, 2) to A; this pair has already been explored and stored, and thus the search terminates without finding the cycle.

The storage options have no effect on this outcome, because it is the state 2 that was explored twice on its owner both times with the same root.

4.2.4 Local search and the bounded nested BFS algorithm

We can also apply local search to the bounded NBFS algorithm presented in Section 4.1.9. Local search is only used in the distributed BFS and not in the sequential nested search. The algorithm is outlined in Figure 4.24 without local search on the left and with local search on the right. The variable *dep* is the local search depth. The algorithm remains correct, because all accepting states are still discovered, either locally or remotely.

4.2.5 Local search implementation

Local search is implemented on the worker thread inside the ExploreChildren() function. A global variable *depth* is set at initialization to the user-defined local search depth. After the generation of every child state we add the check ($dep > 1$) and call the function recursively. Two additional option variables, *foreign_option* and *local_option*, are used for selecting marking options (see Section 4.2.1) for foreign and local states respectively. The value of *foreign_option* ranges between the following:

- `LEAVE`: foreign states generated during a local search are not marked,
- `CACHE_SENT`: foreign states generated during a local search are added to the sent cache, and
- `CACHE_SEARCH`: foreign states generated during a local search are added to the search cache.

```
Worker()
1  while true do
2    receiveWork(s, d)
3    if marked(s) ∧ d ≥ s.depth then
4      continue
5    if s.accepting() then NestedBfs(s, d)
6    mark(s, d)
7    for s′ in s.children() do
8      Send(s′) to owner(s′)
9    endfor
10 endwhile
```

(a)

```
LocalSearchWorker()
1  while true do
2    receiveWork(s, d)
3    if ¬marked(s) ∨ d < s.depth then
4      ExploreChildren(s, d, depth)
5  endwhile

ExploreChildren(s, d, dep)
6  if s.accepting() then NestedBfs(s, d)
7  mark(s, d)
8  for s′ in s.children() do
9    if dep > 1 ∧ d + 1 < bound then
10     if ¬marked(s′) ∨ d + 1 < s′.depth then
11       ExploreChildren(s′, d + 1, dep − 1)
12   else Send(s′, d + 1) to owner(s′)
13 endfor
```

(b)

Figure 4.24: The bounded nested BFS algorithm (a) without and (b) with local search.

The value of *local_option* ranges between the following:

- `LEAVE`: local states generated during a local search are not marked,
- `STORE`: local states generated during a local search are marked in the store,
- `CACHE_SEARCH`: local states generated during a local search are added to the search cache, and
- `ENQUEUE`: local states generated during a local search are added to the worker queue.

Once in the ExploreChildren() function, a flag indicates that a local search is in progress, and if set, the Mark() function is called to handle the storage options according to the values of *foreign_option* and *local_option*. The flag is only set if ExploreChildren() is called recursively.

4.3 Extra implementation details

This section deals with details that are not directly related to the distributed design and implementation in the first part of this chapter, but are important nontheless. We first discuss the input files in Section 4.3.1, then the state structure in Section 4.3.2, and finally state generation in Section 4.3.3.

4.3.1 Importing the system

The model checker reads two types of automata, the model automaton and the Büchi automaton representing the requirement property. Note that the properties are negated, because it is more efficient to find one counterexample than to ensure that all paths satisfy a property. When only a model is provided the model checker performs state enumeration. When both files are presented, the model checker performs property verification. Both automata files are first externally transformed into instruction files, as shown in Figure 4.25. The model files end in *.dmc* while the Büchi files end in *.buc*. The instruction files end in *.code* and consist of a sequence of instruction codes.

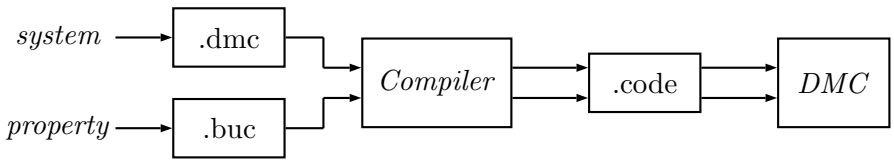


Figure 4.25: The automata files are interpreted to *.code* files and read into the distributed model checker.

Automata files

Both the model and Büchi automata are expressed in simple automaton grammars. The language omits record types but is not difficult to add. The grammar for the model automaton is given below in EBNF:

Model → "model" id ':' [*VarGlobalDecl*] { *Process* } [*Initialize*] "end" '.'
Initialize → "init" ':' { *NewStatement* | *Statement* } "end" ';'
NewStatement → "new" id ';'
VarGlobalDecl → "var" { id ':' *GlobalType* ';' }
VarLocalDecl → "var" { id ':' *LocalType* ';' }
GlobalType → "int" | *ArrayType* | *QueueType*
LocalType → "int" | *ArrayType*
ArrayType → "array" '[' number ']' "of" *LocalType*
QueueType → "queue" '[' number ']' "of" "int"
Process → "process" id ':' [*VarLocalDecl*] [*Initialize*] { *State* } "end" ';'
State → "state" id ':' { *Trans* }
Guard → "guard" *BooleanExpr*
SendStatement → "send" '(' id ',' *Expr* ')' ';'
ReceiveStatement → id '=' "recv" '(' id ')' ';'
SStatement → *SendStatement* | *ReceiveStatement*
Statement → id '=' *Expr* ';'
Trans → "trans" { *Guard* } [*SStatement*] { *Statement* } "goto" id

Anything that changes the values of a state results in a transition. Every transition ends with a *goto* command which forces the system to change to a different state. The guard statements are *guard*, *recv* and *send*. The latter two are queue operations and are only successful when there is a value available to read or there is enough buffer space in the relevant queue.

The *BooleanExpr* rule is not explicitly written, but it includes any Boolean expressions (involving the system variables) that evaluates to *true* or *false*. An example is:

- $(a = 2)\ and\ (b < a)\ or\ (1)$

Similarly, the *Expr* rule is not shown, but it includes any arithmetic expressions, for example:

- $(3 \times 3\ div\ 2\ mod\ 2) + x - 1$

Every process can initialize values in a special *init* process by means of the *Initialize* rule. The model itself can initialize global values or instantiate instances of processes in a global init process. Dynamic process creation is not supported and processes cannot themselves instantiate other processes.

The requirement property is meant as an extension of the model, and its grammar lacks the need for variable declaration. Variables required in the property are expressed in the model file. The Büchi grammar, given below, closely resembles rules of the model grammar:

BState → "bstate" ['*'] id ':' { *BTrans* }
BGuard → "guard" *BBooleanExpr*
BTrans → "btrans" { *BGuard* } "goto" id

Note that it does not include processes, because there is only one Büchi automaton. Additionally, Büchi transitions do not modify the state values and require no statements, which are always assignments. An accepting state is marked with a '*' next to *bstate*.

The specified grammars encourage simple visual design of models without the need to recreate them in a programming language. Our grammar is similar in many respects to the DiVinE specification language (DVE) [4] in that models are directly expressed in the form of their finite state automata. Every automaton node relates to one *State* rule and every edge to a *Trans* rule. Since models do not explicitly express every different state permutation, it is often simpler to do this in the automaton file too, but this is not necessary for correctness. Figure 4.26 demonstrates how the system in Figure 2.1 (b) is expressed in the grammar.

Staying with the same model, to test the property that x will always eventually be greater than 1, i.e. G(F($x > 1$)), we negate it to F(G($x \leq 1$)) and substitute $x \leq 1$ into p in Figure 2.2 (b). The corresponding .buc automata file is shown in Figure 4.27.

```
1 model IncDec:
2   var x : int;
3
4   process P1:
5     state A:
6       trans guard (x = 0) x = x + 1; goto A
7       trans guard (x = 1) x = x - 1; goto A
8   end;
10
11  process P2:
12    var y : int;
13    state B:
14      trans guard (y = 0) and (x = 0) y = y + 1; goto B
15      trans guard (y = 1) and (x = 0) x = x + 1; goto C
16    state C:
17      trans x = x - 1; y = y - 1; goto B
18  end;
19
20  init: new P1; new P2; end;
21 end.
```

Figure 4.26: Illustration of a .dmc file for the system in Figure 2.1.

```
1   bstate A:
2     btrans guard (1) goto A
3     btrans guard (x <= 1) goto B
4   bstate* B:
5     btrans guard (x <= 1) goto B
```

Figure 4.27: Illustration of a .buc file for the Büchi automaton in 2.2 (b).

Code files

Both types of automata files are translated to a sequence of low-level instructions, read by the model checker. The format of the .code file for the *IncDec* model is shown in Figure 4.28. Lines with descriptive comments start with a hash. The instructions are stored as integers; symbolic names are given in the comments.

The .code file is divided into two parts: the model information at the top and the instructions at the bottom. The model information includes types such as arrays, queues, and processes (lines 4–6) that are translated into *entities*. The entire model corresponds to a *global* entity while each array, queue, and process corresponds to one *array*, *queue*, and *process* entity, respectively. Eventually these entities are used to create building blocks in the state vector (see Section 4.3.2). The entities have five attributes:

- **type**: the entity type, i.e., *global*, *process*, *array*, or *queue*.
- **size**: the number of values required for a building block associated with the entity.
- **buffer length**: the size of the array or length of the queue buffer.
- **init instruction address**: the address of the instructions for the init fragment in the instruction list. This applies to every *process* entity and the *global* entity.
- **initial state**: the index of the state a process starts in.

We also list the number of states and transitions, the starting transition of every state, and the starting address of every transition as model information (lines 7-10). The .code files for the Büchi automata work in much the same way.

4.3.2 State structure

Every state consists of a size and an integer vector. The values in the vector are grouped into blocks called entity instances. We can therefore think of a state as a sequence of entity instances. Within a state, each value has a unique address.

```
1 # MODEL DATA:
2 3              # the number of state entities
3 # each entity → type size buffer_length init_address_instruction initial_state
4 0 1 0 0 0   # the global entity
5 1 2 0 7 0   # the entity for process P1
6 1 3 0 8 1   # the entity for process P2
7 # the number of states and each state's starting transition
8 3 0 2 4
9 # the number of transitions and the instruction address of each transition
10 5 9 27 45 70 95
11 115           # the number of instructions
12 # INSTRUCTIONS: (→ address 0:)
13 # initialization for the model
14 20 1 16       # new P1 pop
15 20 2 16       # new P2 pop
16 17            # halt
17 # initialization for process P1
18 17            # halt
19 # initialization for process P2
20 17            # halt
21 # transition 1: guard (x = 0) x = x+1; goto A
22 15 0 23       # push x gload
23 15 0 6        # push 0 eq
24 18            # guard
25 15 0 23       # push x gload
26 15 1 1        # push 1 add
27 15 0 24       # push x gstore
28 19 0          # goto A
29 # ... rest of the transitions here ...
```

Figure 4.28: The .code file for Figure 4.26.

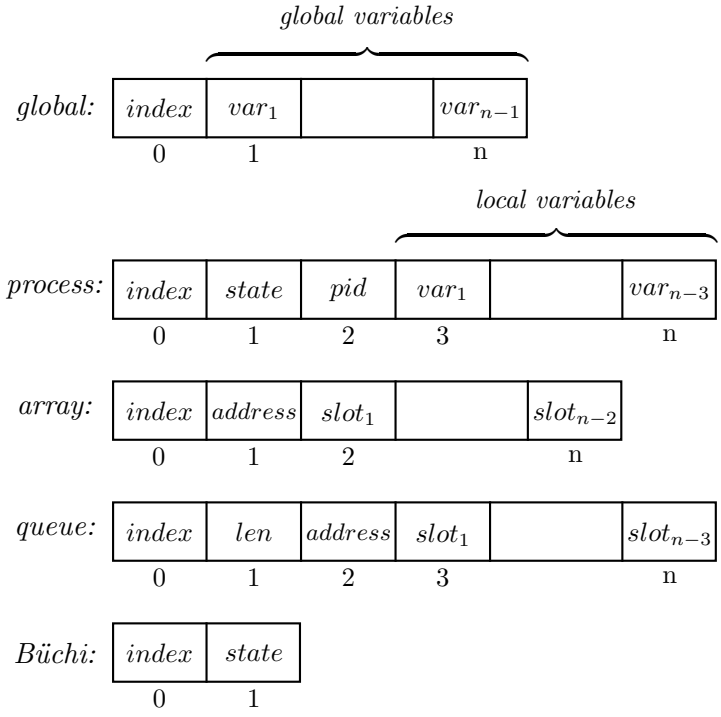


Figure 4.29: The layout of the entity instances.

As we mentioned in the previous subsection, an entity is a description of an array, queue, process, etc. An entity is like a class definition whereas an entity instance is like an instance of the class. For example, there is one entity for each of the processes defined in a .dmc file, but a particular state may contain several instances of a given process entity. An entity instance can refer to another entity instance by storing its vector address. For example, a process instance may refer to an array instance by its address. An entity lookup table records the attributes related to each entity, i.e., type, size, buffer length, etc. Figure 4.29 depicts the differences between entity instances based on the entity type. The value at the first address of every entity instance is the *index* of the entity into the entity lookup table, making it easier to access relevant information of each entity. The remaining values are described below based on the type of the entity instance:

- **global**: there can only be one global instance and it contains one slot for every global variable. In the case of an array or queue variable, it stores an address to the array or queue instance in the vector.

- **process**: multiple process instances of one *process* entity may exist, depending on the number of times the process is initiated. Every instance stores the current automaton *state* that the system is in, and the process identifier, *pid*. These identifiers are assigned to process instances at initialization. Finally the *local variables* and buffer entities (arrays and queues) are handled in the same way as for the global entity.

- **array**: every *array* instance stores the *address* of the array variable, which is either local in a process instance or global and in the global entity. The array instance also contains one slot for each of the *array values*. It is possible to declare multidimensional arrays, and in this case the array values themselves each store an address to another array instance.

- **queue**: every *queue* instance stores the number of buffered items, *len*, and also the *address* of the queue variable. Finally it contains a slot for each of the *queue values*.

- **Büchi**: there can only be one Büchi instance and it contains only one slot to store the current Büchi *state* that the system is in.

Figure 4.30 (a) illustrates the initial state vector in the *IncDec* model shown in Figure 4.26. It consists of a global entity and two process entities (one for $P1$ and one for $P2$), with indices of 0, 1, and 2 into the entity table. Both x and y are initialized to 0 (addresses 1 and 8). The *pid*s of $P1$ and $P2$ are 0 and 1 (addresses 4 and 7). Process $P1$ starts in state A with index 0

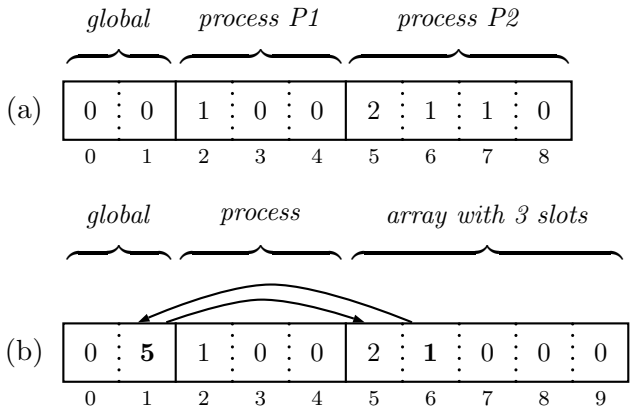


Figure 4.30: The state vector of (a) the initial state from the *IncDec* model, and (b) a state with an array entity instance.

(address 3) while process $P2$ starts in state B with index 1 (address 6). In (b) a system with one process and one global array shows how the only global variable, at address 1, stores the address to the array instance which starts at address 5. The array instance itself stores 1 in its second slot (address 6) to reference the location of its variable.

4.3.3 State generation

Besides the entity lookup table, the data read from the input files are written to two sets of data structures, the first set for the model automaton and the second for the Büchi automaton:

- *state table*: stores the index of the first and last transition of every automaton state. These indices are used for lookups into the transition table.
- *transition table*: stores the first instruction address of every transition. These addresses are used for lookups into the instruction table.
- *instruction table*: stores the instructions in the same order as in the input files.

The State Generator manages these structures and uses them in generating new states.

The next child

For every state we need to find all possible transitions in each of the processes. Thus, the state vector is searched for all process instances, and the *state* slots are used to find the relevant transitions in the state table. Every transition is used to find the relevant instructions in the transition table, to evaluate whether the transition is *enabled* or *disabled*.

Consider the initial state, {0, 0, 1, 0, 0, 2, 1, 1, 0}, of the *IncDec* model in Figure 4.30 (a). The next state is generated by first finding the next process instance, process $P1$. This process is in state 0, or A (address 3), which owns transitions 1 and 2 as the first and last transition. We know from Figure 4.28 line 10 that the first transition maps to address 9 in the transition table. Thus, evaluation starts at index 9 in the instruction table. This is depicted in Figure 4.31.

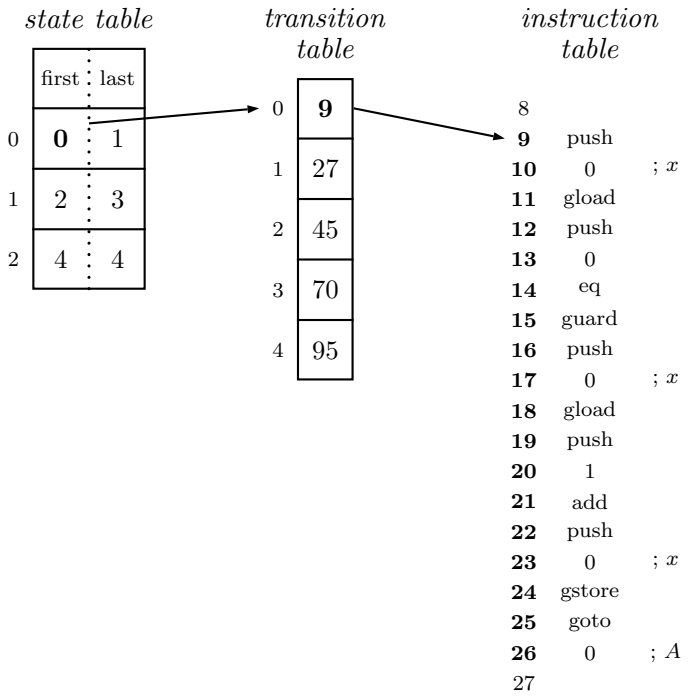


Figure 4.31: The access to the instructions of the first transition in the *IncDec* model.

The instructions of a transition are consecutively interpreted by the State Generator until the transition is found to be disabled, a runtime error occurs, or a *goto* command is found which indicates the transition is enabled. The instructions are stack-based. In this case x and 0 are pushed onto the stack and checked for equality. Since they are equal, 1 is pushed and the *guard* command at address 15 does not abort the transition. Once x is incremented the *goto* command is found at address 25 signaling the end of the transition. The next state is thus {0, 1, 1, 0, 0, 2, 1, 1, 0}, because only x changed and the process remains in state A.

Once all transitions in process $P1$ are explored the next process instance is searched, process $P2$ which starts in state B. This continues until no more transitions remain.

The *NextChild()* function, outlined in Figure 4.32, evaluates every transition to generate one state at a time. Every process instance is examined and the address e hops from entity to entity in the vector. The address *next_addr* saves the location of the next entity when a process has no more transitions (line 3) while the address *last_addr* contains the location of the current process (line 13). We use *cur_trans* to remember the index of the transition under

```
1 int NextChild(State* s, State* child)
2 {
3   int e = next_addr;
4   child = ReplicateState(s);
5   /* all transitions of the current process are exhausted */
6   if (cur_trans > last_trans) {
7     /* find the next process */
8     while ((e < s->size) &&
9       (entity_table[s->vector[e]].type != PROCESS)) {
10      e = e + entity_table[s->vector[e]].size;
11    }
12    if (e == s->size) return NO_MORE_TRANSITIONS;
13    last_addr = e;
14    next_addr = e + entity_table[s->vector[e]].size;
15    cur_trans = state_table[s->vector[e + 1]].first_trans;
16    last_trans = state_table[e->vector[e + 1]].last_trans;
17  }
18  return ExecuteCode(trans_table[cur_trans++], last_addr, child);
19 }
```

Figure 4.32: Function NextChild().

evaluation, and *last_trans* is the index of the last transition possible for the current process. When there are no more transitions in a process (line 6) we need to find the next process (lines 8–11) and the variables are updated (lines 13–16). When we reach the end of the vector all transitions are exhausted for state *s* (line 12). The *ExecuteCode()* function in line 18 evaluates the current transition by interpreting its instructions and, if enabled, saves the new state in *child*. *NextChild()* will return one of following values:

- `SUCCESS`: the next transition was evaluated and found to be enabled after reaching a *goto* command. The new state is saved in *child*.
- `TRANSITION_DISABLED`: a *guard*, *send*, or *recv* command failed and the transition was

aborted before reaching the *goto* command or modifying the state.

- `NO_MORE_TRANSITIONS`: there are no more transitions to evaluate for state s.
- `RUNTIME_ERROR`: the State Generator found some illegal operation and was unable to complete the evaluation of the current transition.

NextChild() is actually slightly more complex, because the interaction of the system automaton and Büchi automaton is omitted here, but included in the actual C implementation. This only includes testing every Büchi transition alongside every model transition, i.e., the union of the Büchi and model transitions. In the evaluation of Büchi transitions we only look at the Büchi entity instance. Büchi transitions are evaluated first, because they are generally fewer and we do not check model transitions with disabled Büchi transitions.

Chapter 5

Evaluation

We introduced local search in the previous chapter, as well as strategies to reduce the lengths of the receiver queue, R. In this chapter we evaluate the effectiveness of local search, the distance partition function, and the queue-reduction strategies. However, the focus is primarily on local search and it spans most of this chapter.

Measurements are performed on selected ESML models [27], chosen so that the set has models with different *average degrees* (branching factors). Table 5.1 lists the selected models; and Table B.1 in Appendix B lists their average branching factors, which fall between 1 and 11. A state space that has on average more children states will initiate local searches that encapsulate a larger number of states than a state space with a smaller degree; the average degree is typically around 3 [47]. The selection of models is often a contentious issue and so, apart from the branching factor, we have tried to make the models as varied as possible.

We use a nine-node Beowulf cluster for the experiments. The cluster uses a T1 network. Every node has 1 GB of physical memory and Intel Xeon 3.06GHz dual-processors. We do not always utilize every node in the cluster and in this way we can observe the effect of the cluster size on experiments. A cluster size of n refers to a network with n nodes and $2n$ processors.

We measure runtime in one of two ways; different sections will make clear which of these they refer to:

- *CPU time*: the sum of the system and user times, i.e., the sum of the total processor

Model	*Description*	*Instances*
LPn	Lamport's mutual exclusion for n processes	$n = 5, 6, 7$
LFn	Leader election based on filters for n processes	$n = 5, 7$
BAn	Bakery mutual exclusion for n processes	$n = 1, 4, 5, 6, 7$
ADn	Concurrent adding puzzle with maximum $n \times 100$	$n = 6, 8$
FIn	Fischer's mutual exclusion for n processes	$n = 6$
DPn	Dining Philosophers problem for n philosophers	$n = 15$
ELn	Elevator2 controller for n floors	$n = 2, 3$
HAn	Tower of Hanoi puzzle for n discs	$n = 3$

Table 5.1: Models selected for measurements.

time the model checker uses in executing its instructions, and the total processor time the system uses on behalf of the model checker.

- *Real time*: the time from when the model checking process starts until it finishes. This time includes the CPU time, communication time, and the time used for scheduling the operating system processes.

In particular, real time is used when we evaluate the impact of local search on communication. Since every node in the Beowulf cluster is a dual-processor, the total system and user times on both processors are included in the CPU time calculation which then includes some overlapped time. Consequently, the CPU time may be larger than the real time. CPU time is used when we evaluate the impact of local search on redundant work; in this case the time of the slowest node is reported. The times are measured in seconds (s).

We first examine speedups in our state enumeration algorithm in Section 5.1. Then the queuing strategies are evaluated in Section 5.2, local search in Section 5.3, the distance partitioning in Section 5.4, and the grouping together of sent states in Section 5.5. The results are summarized in Section 5.6. We always use the randomized partition function, unless stated otherwise.

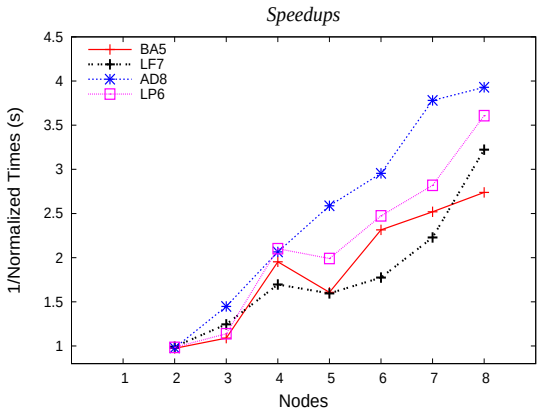


Figure 5.1: The speedups for models BA5, LP6, LF7, and AD8.

5.1 The distributed algorithm

In this section we measure the real time against the number of nodes in the network for the algorithm in Section 4.1.1. The purpose of these measurements is to establish a baseline for other experiments. Note that local search is switched off. Models BA5, LP6, AD8, and LP7, consisting of 7.8, 8.7, 13.5, and 26.3 million states, respectively, are measured on networks with 2 to 8 nodes. We use state compaction [29] with LP7 to save memory while non-compacted states are used for the other three models.

Apart from an anomaly at 5 nodes, the graph in Figure 5.1 depicts speedups for every model as the number of nodes increase. Times are improved by 2.73 times for BA5, 3.67 times for LP6, 3.93 times for AD8, and 3.32 times for LP7 from cluster sizes 2 to 8. This amounts to a saving of 24 minutes in the case of LP6.

5.2 Queue monitoring

The *store* (S), *limit* (L), and *hash* (H) strategies for queue-reduction from Section 4.1.3 are compared to determine which of them saves the most queue memory (the memory used for storing the states in the receiver queue on every node) without slowing down the runtime. These strategies are applied to the normal case (N) where received states are always queued, like the previous section. In both S and L the reduction percentage in queue lengths and queue

	N	*S*	*L*	*LE*	*LS*	*LES*	*H*
Nodes	Kilobytes	%	%	%	%	%	%
Queue Memory for LP6, state space = 8,717,688							
5	481453	65.2	65.5	72.3	47.2	62.2	0.7
6	461085	67.9	66.0	70.0	54.5	66.5	0.8
7	474664	67.8	53.9	78.2	50.2	64.8	0.7
8	329227	67.9	51.7	60.2	40.9	64.0	0.6
Queue Memory for BA5, state space = 7,866,401							
5	317543	82.6	67.7	78.8	79.8	71.6	3.0
6	261897	76.3	70.8	79.8	63.8	77.0	0.7
7	354316	70.2	58.4	72.0	64.5	75.4	0.7
8	199583	60.0	61.8	67.6	46.2	66.5	0.7
Queue Memory for AD8, state space = 13,534,005							
5	10256	80.5	59.2	72.1	69.4	78.9	6.8
6	8364	70.6	58.3	62.3	70.8	81.6	6.0
7	7034	81.4	72.2	87.2	72.2	83.1	6.5
8	7217	70.7	63.5	76.7	63.4	70.8	5.7

Table 5.2: The queue memory required for LP6, BA5, and AD8 with and without the queue strategies applied.

memory is the same, and we show results only for memory. Where necessary, we mention how queue lengths are affected for H.

By trying various values for L it was found that a limit of 100 000 was the most consistent in time and queue-length reduction when applied to BA5. Another model LP6 shares similar queue lengths (600 000 to 150 000 from 5 to 8 nodes) and thus also makes use of this limit. However, an 80 000 limit is used for a third model AD8 with characteristically shorter queues (200 000 to 100 000). Experiments were also run with E, an extended limit strategy that places a *marker* on the last state of the receiver queue when all the queued states are checked against the store. When the limit is reached again, states queued before the marker are not rechecked. This avoids unnecessary store comparisons if the limit is reached frequently. The combination of the S and L strategies is also tested.

Table 5.2 depicts the differences in queue memory on clusters sizes 5, 6, 7, and 8 when these

Nodes	*N*	*S*	*L*	*LE*	*LS*	*LES*	*H*
Times (s) for AD8, state space = 13,534,005							
5	751.5	1163.2	2026.8	1050.4	2257.5	1127.8	1422.8
6	670.0	966.5	1804.9	886.5	1757.9	957.5	1164.0
7	517.5	824.5	1418.1	731.6	1422.6	810.3	1334.5
8	747.1	1019.9	1311.2	966.7	1201.1	1035.7	1158.9

Table 5.3: The CPU times for AD8 with and without the queue strategies applied.

strategies are compared to the case without queue reduction (N). The H strategy clearly outperforms the rest, because every state is only queued once on its owner node. Queue lengths are reduced by approximately 28% and 33% for LP6 and BA5, and for LP6 this relates to 1.5 million fewer queued states on 8 nodes. Since we only queue a pointer reference to the store location of a state, only a small fraction of the original queue memory is required. Results are not quite as good for AD8, because it has much shorter queue lengths due to its smaller branching factor of 1.5, and fewer redundant states are generated. Although all of the strategies always reduce queue lengths, the most reduction in queue lengths (42%) for AD8 is achieved through the L strategy. For the three models, queues are 72% shorter in the best case and 13% in the worst case.

In all strategies the real and CPU times remain unchanged for LP6 and BA5. However, Table 5.3 depicts a slowdown in the CPU times for AD8. All of the strategies are slower than the N strategy (the case with no reduction). The L and LS strategies are the slowest — between 1.6 and 3.0 times slower than N. For both strategies a common scenario is that a state is checked against the store multiple times. Slow times for the H strategy results from the need for store synchronization between the receiver and worker threads.

We can assume that if time is an issue, the queue-reduction strategies are not useful when applied to models like AD8 with already-short queues and small branching factors. Otherwise, the H strategy saves the most memory. In addition, using the correct limit in the L strategy relies on the target model.

5.3 Local search

The purpose of the local search technique is to reduce communication. This leads to the following questions: Are speedups obtained, and if so, at what depth? Is the number of sent states reduced? And what is the impact on redundant work?

We discuss these answers in the summary at the end of the chapter in Section 5.6. First we examine local search applied to the state enumeration algorithm in Section 5.3.1, and in the subsequent sections we consider how the sent cache, queuing strategies, state compaction, and the storage options affect local search. Finally, local search is used with the model checking algorithms in Section 5.3.6. Unless stated otherwise, during a local search foreign states are saved in the sent cache while local states are saved in the store.

5.3.1 Local search and state enumeration

The behaviour of local search is first examined through two small models (that both run faster sequentially than on a distributed cluster). LF5 consists of 1.5 million states and FI6 of 2.9 million states. In each case a sent cache is used and restricted to 300 000 states. Note that when $depth = 1$ local search is not used. Measurements are made over cluster sizes 2 to 9.

The model LF5 is a clear indicator of how we expect local search to affect state exploration. Some states are revisited, as depicted by the state and transitions growths in Figures 5.2 (a) and (b). This redundant work is more severe on larger clusters and with greater depth. On a larger cluster, every node explores a smaller fraction of the state space, and initiates fewer local searches. In a smaller cluster, the same number of local searches is distributed among fewer nodes, and therefore the possibility is higher that some foreign states repeat in multiple local searches on every node. When the depth is greater, every local search generates a larger fraction of foreign states, and therefore chances are greater for a foreign state to repeat on different nodes. In this case the redundant work factor grows from 2.6 at $depth = 3$ to 6.2 at $depth = 17$ on 8 nodes, and from 1.5 to 1.8 on 2 nodes.

Real times are consistently improved in Figure 5.2 (c) as the depth increases. Times are 9.3 and 8.0 times faster on 2 and 8 nodes respectively at $depth = 17$ than $depth = 1$. Due to the

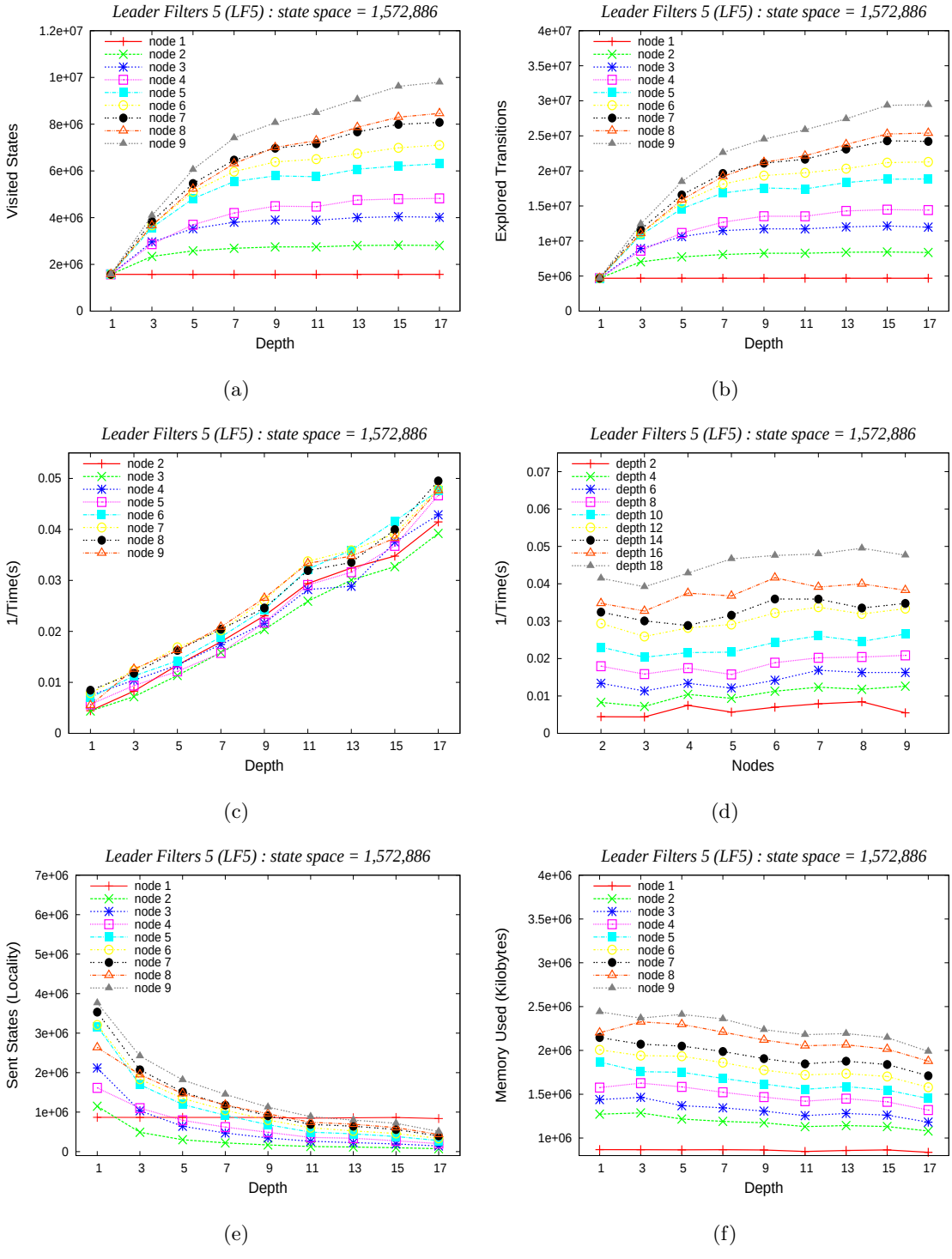


Figure 5.2: Measurements for LF5: (a) total states, (b) total transitions, (c) real time speedups against *depth*, (d) real time speedups against cluster size, (e) sent states, and (f) total memory.

Times (s) for FI6 (77.77s on 1 node), state space = 2,898,705								
	Depth							
Nodes	1	3	5	7	9	11	13	15
2	440.73	320.73	282.63	250.31	238.11	226.11	250.51	235.16
3	254.14	430.86	510.83	630.50	807.82	1159.1	DNF	DNF
4	195.67	382.69	479.26	632.74	852.09	DNF	DNF	DNF
5	132.61	349.13	570.45	720.54	DNF	DNF	DNF	DNF
6	100.49	276.31	515.41	743.23	DNF	DNF	DNF	DNF
7	88.92	260.57	505.11	742.91	DNF	DNF	DNF	DNF
8	66.64	244.04	443.70	714.13	DNF	DNF	DNF	DNF

Table 5.4: Real Times for FI6.

small state space of LF5, an increase in the cluster size forces only a small speedup in (d) when depth is kept consistent. It is found that on larger models, where the cluster size plays a larger role, this speedup is more significant. In (e) the sent states are depicted and it drastically drops which results in less communication and the improved times in (c). The figure in (f) shows how memory is slightly reduced, because some states are not stored locally, but are instead temporarily cached on remote nodes during a local search.

In contrast, real times in Table 5.4 for FI6 are considerably worse, and for larger numbers of nodes the memory is exhausted even though the model completes on a single node (DNF means "did not finish"). This is caused by the drastic increase in queue lengths. A large fraction of the states received on nodes has already been stored or queued. The same state might even be in the worker and receiver queue at the same time. Due to the increased redundant work that results from local search, this repetition is compounded. It is not only that the same state is investigated on several different nodes — a phenomenon we have been referring to as "redundant work". It may also happen that the same state is explored on the same node many times over — this is known as *repetitions*. Every node generates many redundant foreign states and the sent the cache is too small to cache all of them. Time is wasted sending these states as well as redundantly checking them against the store on their owner nodes.

In both models the CPU times are reminiscent of the redundant work factors. For LF5 the times are relatively consistent with increasing depth on all cluster sizes. However, for FI6 times

are slower with increasing depth; on 8 nodes the CPU time is 14.1 times slower at $depth = 7$ than $depth = 1$.

5.3.2 The influence of the cache size

In the previous subsection we found that the receiver queues contain redundant states. In this section we try to alleviate this problem by increasing the cache size (bound) to minimize sent states, i.e., states that will eventually be stored in the receiver queues. Measurements are made on an 8-node cluster (the results are similar for other cluster sizes) for model FI6 and two other models each with 7.6 million states.

Table 5.5 depicts the results for FI6, and as we would expect, it indicates that the larger the cache size, the faster are the real times and the shorter are the queue lengths. The columns *Time* and *Queue* contain the real times and average queue lengths at $depth = 1$. The consecutive *Best* and *Worst* columns depict the fastest (shortest) and slowest (longest) times (queue lengths) as a result of local search. The number in brackets is the depth at which each instance occurred. Initially, in the first row, queue lengths are 75 times the length of the original queues without local search. This means that the number of queued states was 2.68 times the size of the state space. By simply enlarging the cache, and thereby eliminating all of the redundantly sent and locally-searched states on every node (as is the case in the last row), queues are only 3 to 6 times longer as in the $depth = 1$ case. Eventually all explorations are able to complete, because the memory is no longer exhausted by large queues anymore.

FI6 (77.77s on 1 node), state space = 2,898,705, Nodes = 8						
Cache Size	*Time*	*Best*	*Worst*	*Queue*	*Best*	*Worst*
300000	66.42 (1)	221.82 (3)	DNF (9)	101896 (1)	1452287 (3)	7685060 (7)
900000	54.85 (1)	123.75 (3)	DNF (15)	103960 (1)	545905 (3)	7358179 (13)
1500000	58.63 (1)	96.85 (3)	332.56 (17)	99341 (1)	379825 (3)	1658740 (17)
2100000	54.04 (1)	91.44 (3)	119.10 (17)	98816 (1)	377700 (3)	683868 (5)
2600000	52.76 (1)	92.62 (3)	112.65 (17)	96114 (1)	375247 (17)	677434 (5)

Table 5.5: The local search real times (seconds) and the queue lengths (states) when the cache size is varied for model FI6.

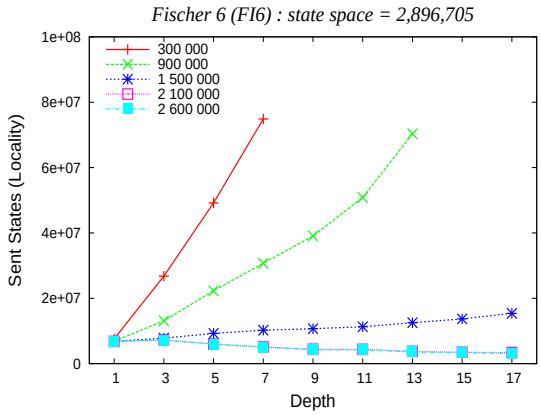
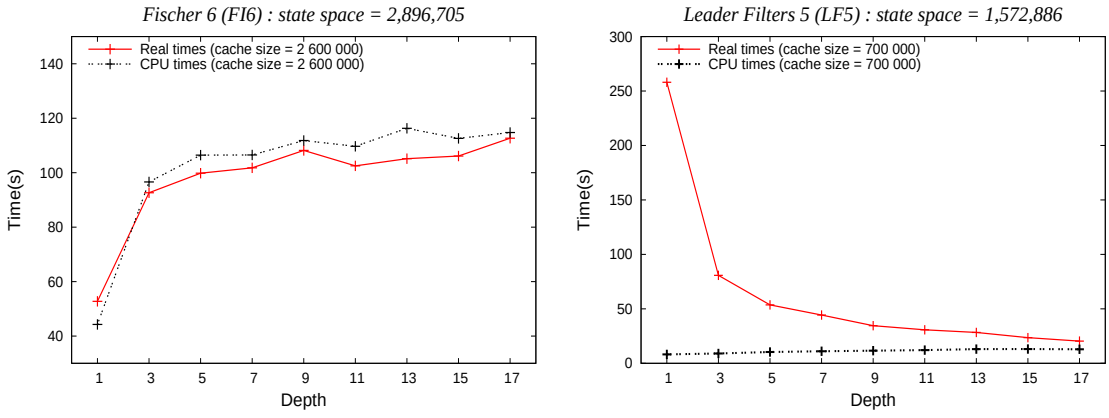


Figure 5.3: The total sent states against the local search depth for different cache sizes for FI6.

However, local search never improves on the times in the *Time* column, even in the last row; the redundant work is too severe and is enlarged by 3.6 times at $depth = 3$, and 6.4 at $depth = 17$. The reason is partly seen in Figure 5.3 where the growth in sent states, when the cache is smaller than 2 million, contradicts with what we want to achieve with local search; instead of fewer, more states are sent. FI6 simply produces too many foreign states in local searches (as we have seen in the previous subsection), and the cache is too small to record all of these in addition to the sent states. This leads to repetitions in both the foreign and sent states on every node. Redundancies in foreign states also lead to extra sent states, as more final states are reached in local searches.



(a) (b)

Figure 5.4: The differences in real times and CPU times against the depth for the models (a) FI6 and (b) LF5.

In the case where the cache size equals 2.6 million, all repetitions are eliminated and the sent states do decrease, but only slightly, and this improvement on real times is negated by the time wasted on generating and caching the abundance of foreign states. Figure 5.4 (a) depicts the CPU and real times when the cache size equals 2.6 million states. The arcs are very similar and indicate that nodes are busy most of the time. If we compare this with LF5 (where local search works) in (b), CPU times are small in relation to real times. This means that nodes are idle more of the time and this allows them to process work from other nodes without much delay.

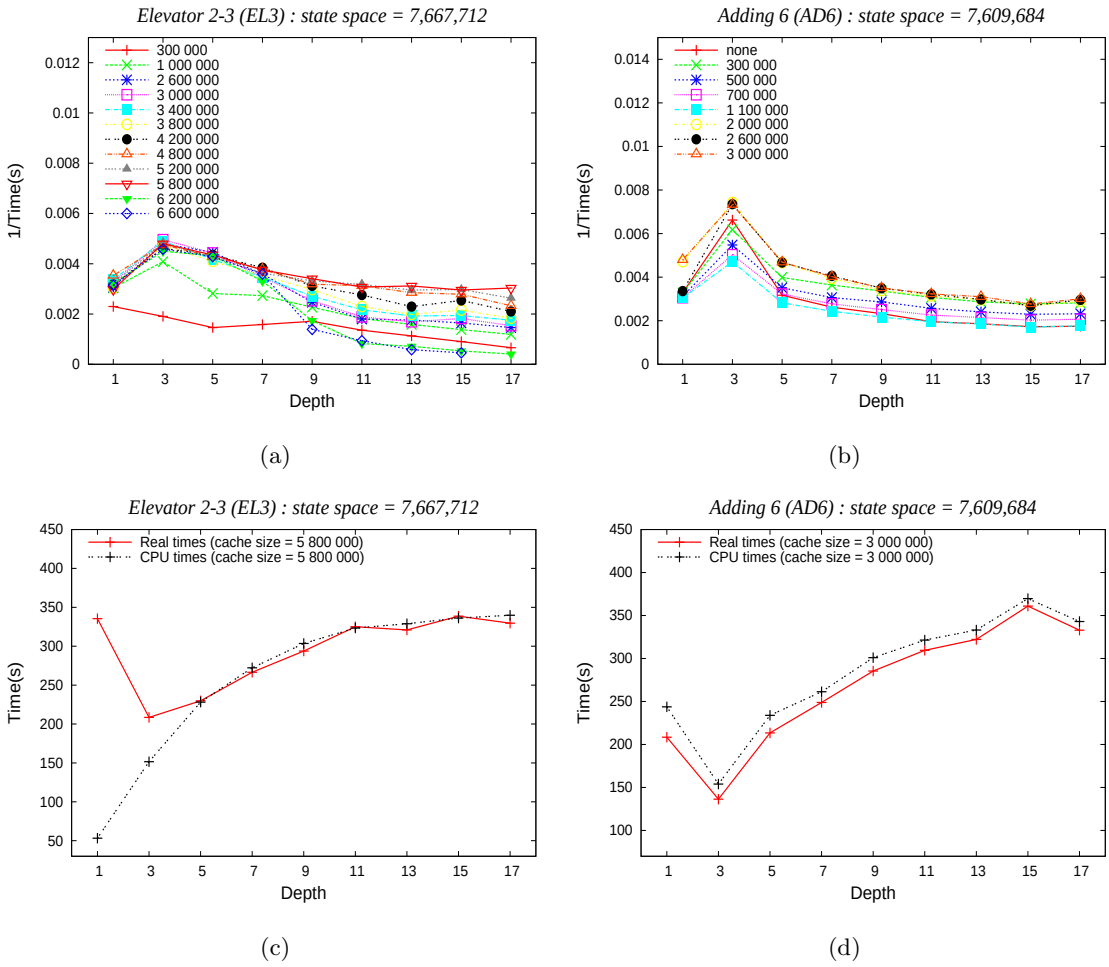


Figure 5.5: Measurements of how the cache size affects the running times in local search: the real times for (a) EL3 and (b) AD6, and the difference between real and CPU times for (c) EL3 and (d) AD6.

Since redundant work has an additive effect on the number of sent states, we assume local search is more viable when the depth is relatively small, where redundancy is at its lowest. This is validated by models EL3 and AD6 in Figures 5.5 (a) and (b) where the real times are plotted with each cache size. For EL3 the real times are improved when $depth < 7$, and deteriorate when $depth >= 7$. In addition, if the cache is too large, times can slow down. For AD6 the times improve at $depth = 3$, and deteriorates for larger depths.

Like for FI6 and LF5, times are the fastest for EL3 when the difference between the real time and CPU time is the largest (depicted in Figure 5.5 (c)). AD6 in (d) is an exception; the CPU times are larger than the real times. The reason for the real time improvement at $depth = 3$ is due to a very small branching factor of 1.5. The redundant work factor is only 1.3 at $depth = 3$ and 3.6 at $depth = 17$, and has a much smaller impact on sent states. Therefore local search can reduce the SENDS (by 1.65 times), and improve times, because the fraction of foreign states that delay CPU times are much smaller. Beyond depth 3 the differences in sent states are too small to improve the times further.

A model's branching factor, however, is only one contributor to how local search impacts redundant work. EL3 generates less redundancy than FI6, but its branching factor of 7.2 is larger than that of FI6 at 4.3. The other contributor, which affects the number of repetitions, is how often and how many states are generated as foreign states. In models where most repetitions surround only a select few states, the redundancy will be much lower. We have not yet identified the properties that contribute to how often states are revisited in a model.

We have found in this subsection that local search relies on the cache to eliminate redundancy, improve times, and reduce queue memory. It is hard to predict the exact size required for the cache, but it is possible to find a benchmark for every model. Based on a large set of models, local search was only found successful when the redundant work falls into the ranges in Table 5.6. At every depth d the cache size should be set so that the redundant work is at most d. Although this may be difficult to predict a priori, it could be adjusted as the model checker is running.

Unfortunately local search has only shown very small, if any, time improvements. Too much time is spent on reducing the redundancies created by local search. However, we know it has

	Redundancy Range	
Depth	*low*	*high*
2	1.2	2.2
3	1.2	3.2
4	2.8	4.1
5	3.2	4.9
6	3.6	5.5
7	3.8	7.0
8	5.4	8.0
9	4.2	9.0

Table 5.6: Each row contains the range of redundant work factors in which local search worked the best at every depth.

the greatest impact on models where the real times are much larger than CPU times, i.e., time is spent mostly on communication. We shall validate this later in another model in Section 5.3.4.

5.3.3 The influence of queue reduction strategies

We next investigate if the queuing strategies from Section 4.1.3 can further reduce queue lengths (and memory) without slowing down times during local search. The cache is kept consistent at 2 million, and models BA5, BA6, and LP6 are used alongside the N (normal), H (hash), S (store), and L (limit) strategies on an 8-node cluster. The limits 100 and 100 000 are used with L and abbreviated to a and b, respectively.

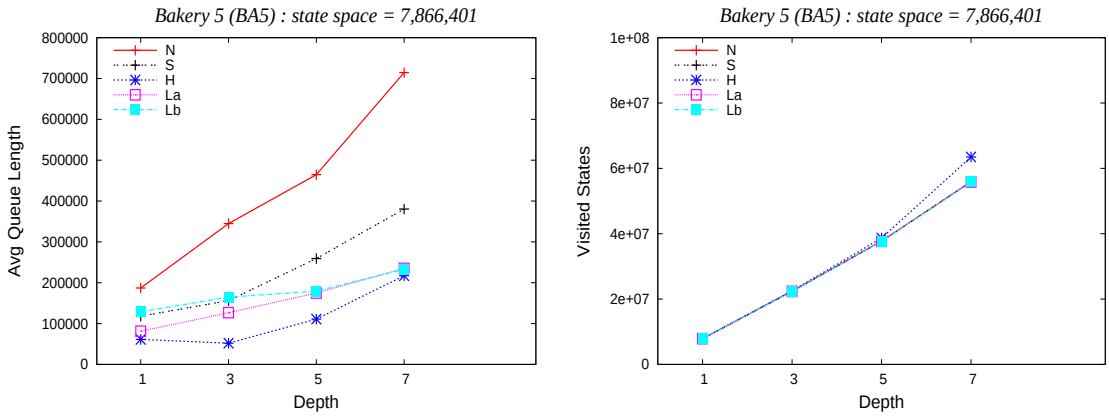
Table 5.7 depicts the real times for BA5, and they are slightly improved in the first row when $depth > 1$. All of the other methods show similar results, but overall they are slower than N. Time is wasted on the extra store comparisons in S and L, and on store locking in H. Figure 5.6 (a) depicts the queue lengths of BA5; H has the shortest queues and uses the least amount of memory. Queues are reduced by 2.3 times when $depth = 1$ and between 3 and 4 times for depths 3 to 7, compared to N.

H also shows a small increase in redundant work per depth as seen in Figure 5.6 (b). This

Times (s) for BA5, nodes = 8 state space = 7,866,401				
	Depth			
Type	1	3	5	7
N	381.12	256.30	236.11	265.02
La	489.26	319.10	240.74	284.25
Lb	570.01	322.89	286.97	280.30
S	402.85	341.80	318.68	314.95
H	408.54	382.89	328.67	371.51

Table 5.7: The local search real times when the queue strategies are used with local search for BA5.

anomaly originates around local states: in H a received state is first stored and then queued, while in the other strategies the state is queued, and only stored when taken from the queue. If such a state, while still in the queue, is visited in a local search in N, it will be checked against the store and explored, because it is not in the store yet. But in H the state is already in the store, thus it is not explored and the local search backtracks. However, this state has not yet been explored, and is eventually taken from the queue to initiate a new local search. At this stage in N the state is ignored, because it was explored in a local search. This difference means



(a) (b)

Figure 5.6: Local search measurements for BA5 with different queue strategies: (a) average queue length and (b) total states.

Times (s) for LP6, nodes = 8 *state space = 8,717,688*				
	Depth			
Type	1	3	5	7
N	355.65	270.50	415.62	557.30
La	430.10	328.76	483.25	549.12
Lb	483.44	377.52	514.29	562.43
S	458.18	417.96	587.93	612.62
H	443.86	449.61	664.73	774.55

Table 5.8: The local search real times when the queue strategies are used with local search for LP6.

that H generates more redundant foreign states, because the depths of the local searches are essentially extended.

The reverse happens when local states are not stored in local searches: H generates less redundant work than N. In this case, local states that are generated in a local search in N will not stop queued states from initiating local searches, while in H these states stand a larger chance of already being stored. We illustrate this with four independent runs of LP5 at $depth = 8$: using N and H where local states are stored, and N and H when local states are not stored. We add *not* as an indication of the latter case. The cache size is set to 900 000 so that no states are ever replaced in the cache. The total states explored are as follows:

$$N < H < H_{not} < N_{not} \Rightarrow 5384228 < 5661636 < 6469732 < 8252160$$

The percentage of local states explored rises from 28% in H_{not} to 46% in N_{not} in this case.

Results for LP6 are similar to BA5, except that times deteriorate when $depth > 3$. Table 5.8 depicts this. N is the fastest when employed with local search. For both BA5 and LP6 La is quicker than Lb, and has slightly shorter queue lengths. This is due to the extra redundantly generated foreign states in local searches, and it means that frequent checks on queued states against the store are more often successful.

5.3.4 The influence of state compaction

All of the queuing strategies investigated in the previous section incur penalties on runtimes, but they save memory. A simpler, different approach to save memory is to compress states using state compaction [29]. The states of the three models BA5, HA15, and LF7 are reduced from sizes 196, 320, and 260 bytes to 16, 36, and 20 bytes, respectively. HA15 consists of 14.3 million states and LF7 of 26.3 million states.

Table 5.9 shows the real times for each model measured on an 8-node cluster with and without state compaction (SC). The memory was insufficient for LF7 without SC, and thus it is listed only with SC. In each case the hash (H) and normal (N) strategies are used in combination with SC, and a 1 suffix refers to a 1-million cache bound and a 2 suffix to 2 million. The first thing to note is that with SC the times are up to 7.5 and 23.7 times faster for models BA5 and HA15, respectively. The second thing is that the times worsen with local search in the case of SC, and the best times are found when $depth = 1$.

The reason for SC's deterioration at larger depths is that states are smaller and are relayed across the network more quickly. Consequently, less time is saved if a state is explored locally instead of remotely. In fact, every model we tested with SC and local search resulted in time deterioration with increasing depth. This even applies to HA15 that is 4.4 times faster with local search (at $depth = 7$) than without. If we compare the differences between the SC and

	Depth, nodes = 8			
Model	1	3	5	7
BA5N2	381.12	256.30	236.11	265.02
BA5scN2	41.51	88.98	119.99	163.06
BA5scH2	43.45	85.68	125.63	198.50
HA15N1	9834.61	4762.23	3076.07	2301.54
HA15scN1	130.31	168.56	191.65	232.10
HA15scH1	94.27	147.23	226.89	240.61
LF7scN2	209.50	335.41	437.93	453.30
LF7scH2	156.04	293.32	353.56	430.20

Table 5.9: Real times (seconds) with and without state compaction.

non-SC instances for HA15, the results validate our findings in Section 5.3.2: when the CPU times are larger than the real times, local search performs poorly. The CPU time is 1.3 times larger than the real time for HA15 with SC, when $depth = 1$ (when local search is not used), indicating that the nodes are rarely idle. In contrast, the real time is 55 times larger than the CPU time at $depth = 1$ for the non-SC instance. The average queue lengths are also much shorter and ranges only between 870 and 3500 states. Nodes are never overburdened with a large workload, and are frequently idle waiting for states. The idle time is therefore best used for local search in the non-SC case. The visited, cached, and sent states are identical between the two instances.

With SC, times for H are faster than N at $depth = 1$ in our implementations. The reason for this is that local states that are queued are converted to non-compact states. Since fewer states are queued in H, we save time on such conversion overheads. We require non-compact states for fast partitioning and generation of children states. Without this the algorithm is much slower, because it is expensive to retrieve values from a compacted state.

Even though local search performs well for some models, like HA15, times are much faster when we apply SC, and local search becomes redundant. In the models we measured, SC eliminates precisely the circumstances under which local search is successful. However, it may be that some models still contain large states and idle times, even after SC is applied.

5.3.5 The influence of store/cache options

It can be assumed, based on the importance of the cache size as found earlier, that less work will be repeated when sent states, local states, and locally-searched foreign states are recorded. Runtimes will be faster because the running time is a fraction of the redundant work. The following options are considered:

1. local states are stored while foreign states are cached in the sent cache,
2. local states are queued while foreign states are cached in the sent cache,
3. local states are not stored while foreign states are cached in the sent cache,
4. local states are stored while foreign states are cached in the search cache,

	Depth for LP6, state space = 8,717,688, nodes = 8					
	3		5		7	
Option	*Time*	*Work*	*Time*	*Work*	*Time*	*Work*
1	270.50	3.50	415.62	7.56	557.30	11.74
2	292.90	4.02	542.32	9.16	717.40	15.86
3	322.57	4.65	772.63	13.48	DNF	DNF
4a	459.63	4.40	949.13	10.64	1137.60	17.92
4b	434.03	5.00	884.37	13.23	1119.34	23.94
4c	481.79	4.08	927.81	8.92	1021.67	14.08

Table 5.10: Real time (seconds) and redundant work factor comparisons when altering storage options during local search for LP6.

5. local states are stored while foreign states are not stored, and

6. local states and foreign states are not stored.

The model LP6 is examined in the case of options 1 - 4. The results are displayed in Table 5.10 where for each depth the real times and redundant work factors are shown. Option 4 is repeated three times. In 4a both the sent cache and search cache can hold 2 million states, in 4b the sent cache holds 1.5 million and the search cache 0.5 million, and in 4c vice versa.

Local search consistently performs best with option 1 where all states are saved and redundant work is minimized. In each of the other cases more redundant work induces slower times. Interestingly, this is true for option 2 as well where local states are queued instead of stored. A local search backtracks when a local state is found and queued. Eventually these local states initiate new local searches each which results in a scenario similar to the hash strategy (see Section 5.3.3). The new searches reach a larger portion of the state space than option 1. The time improvements (at depths 3 and 5) of option 4b over 4a and 4c indicate that it is more valuable to stop redundant sends than redundant locally-searched foreign states.

Local searches with options 5 or 6 increase the redundant work considerably. To examine this, a smaller model is selected, BA4, consisting of only 150000 states. Table 5.11 illustrates clearly how in each case the redundant work is vastly increased with increasing depth and the real times are affected as a result. When BA4 is run with option 1, 11.7 times the original state

	Depth for BA4, state space = 157,003, nodes = 8							
	3		5		7		9	
Option	*Time*	*Work*	*Time*	*Work*	*Time*	*Work*	*Time*	*Work*
5	9.33	3.86	8.46	17.03	10.95	78.53	30.86	342.29
6	14.37	8.12	12.09	51.17	30.90	316.91	158.65	1922.70

Table 5.11: Real time (seconds) and redundant work factor comparisons when altering storage options during local search for BA4.

space is explored while the redundant work factor in option 6 climbs to 1922.7! In larger models the extra redundancy leads to very slow times and also long queue lengths which in most cases leads to memory exhaustion. This is avoided in the case of the hash strategy, but the times are still very slow.

It is clear that local search works best when as much of the redundant work is eliminated by saving either all or some of the local and foreign explored states. The non-storing methods increase the redundancy to such an extent that it can lead to memory exhaustion, even though the goal is to save memory.

Clearly this, alongside the need for a large cache in Section 5.3.2, indicates that to store foreign states locally in the store is also a viable option. However, this is not considered as a practical solution, because foreign states will use permanent memory reserved for local states. If a state is stored on many nodes the memory requirements are increased, worse so depending on the redundant work factor and on how many nodes every state is stored. For many models every node could end up storing nearly the entire state space, which when large means memory will be exhausted and the distributed environment made redundant.

5.3.6 Local search and model checking

In this section we discuss how local search affects the two model checking algorithms discussed in Sections 4.2.2 and 4.2.4. Measurements are made on an 8-node cluster. For the naive CVWY algorithm we use state compaction, because the algorithm functions with pairs of states and the state space is greatly enlarged. For example, one property checked against model BA1 that

consists of 1506 states enlarges the state space 65 times. In addition, safety properties that do not require cycle detection is checked with the normal state enumeration algorithm in Section 4.2.1, with state compaction. We only describe the effect of properties that result in errors (and produce counterexamples). Measurements with correct properties resemble the results found in the previous sections.

Measurements were based on small and large models for the naive algorithm and in most cases local search either delays completion, or the real times remain consistent with the times found without local search. For example, when the property G(*floor* 1 *requests elevator* $\Rightarrow$ *floor* 1 *is serviced*) is tested with EL3, the resulting state space consists of 350 000 states. The local search redundant work factor is 3.5 at $depth = 2$, 4.0 at $depth = 3$, and 6.1 at $depth = 4$. The real times are 4.6, 7.9, and 10.6 times slower at depths 2, 3, and 4. Similarly, for a second property the redundant work eventually reaches 6.3 at $depth = 4$ while real times are 4.9 times slower.

Using the bounded algorithm in Section 4.2.4, real times deteriorate for every model tested with local search. The redundant work is consistently larger, because local searches now include accepting states that initiate nested searches, and states that are revisited when found at a new, smaller depth. For example, when the property GFp is tested against the mutual exclusion algorithm in BA4, where p means that the critical section is reached, the redundancy jumps to 2.2 at $depth = 2$, 8.7 at $depth = 3$, 19 at $depth = 4$, and 22 at $depth = 6$. In the case of LP5 and the property G($q \Rightarrow$ F$\neg q$), the redundant work factor increases to 38.9 at $depth = 2$ and real times are 37 times slower. Here q refers to some process being in the critical section.

Finally, when we measure the safety properties that do not require cycle detection, we select those that are violated late during enumeration, and we check if we can find these violations earlier with local search. However, for all models tested, real times deteriorate with increasing depth and violations are found later rather than earlier.

Since our model checking algorithms severely enlarge redundant work, local search is a poor tool when approaching communication reduction in model checking, at least here. Both the CVWY and the bounded algorithm generate a lot of states without local search, and this contributes to the high redundant work factors with local search.

5.4 Distance partitioning

In this section we measure how the distance partitioning introduced in Section 4.1.4 fairs against the normal randomized partitioning. An 8-node cluster is used, but results are similar for other sizes. The cache size is kept consistent at 1 000 000.

Table 5.12 illustrates how real times and the number of sent states are improved for the three models LP6, BA5, and AD8. For LP6 the communication is reduced by 3.7 times, for BA5 by 4.9 times, and for AD8 by 1.3 times. We are able to save 4 minutes, 7 minutes, and 5 minutes for the three models, respectively.

However, the state partitions are unbalanced, ranging from 0.08 million states on one node to 3.6 million on a second node in the case of LP6. When $r = 10$ the sent states are slightly larger and times are slightly slower, but the state partitions are better balanced. Overall though, states are grouped in only a small portion of the buckets in the distance table.

When state compaction (SC) is applied, real times are closer to the CPU times, as we have mentioned in Section 5.3.4, because less time is required for communication. Thus, SC works best when every node receives an equal share of the state space, like in random partitioning. If one node explores a larger state partition than the others, the CPU time on it dominates the real times, and the time saved on communication is redundant.

Therefore we need states to be spread uniformly across the buckets. So far we have used the OR operation to combine the different values of a state into one distance value (see Section 4.1.4). However, using XOR balances the partitions better with SC. Even so, we found that

	Normal Partitioning		*Distance Partitioning*	
Model	*Time*	*Sends*	*Time*	*Sends*
LP6 ($r = 20$)	528.58	17740921	282.48	4776565
BA5 ($r = 20$)	669.65	15631564	252.58	3172784
AD8 ($r = 20$)	786.90	14076919	449.41	10540681

Table 5.12: Real times (seconds) and sent states when we apply randomized and distance partitioning to LP6, BA5, and AD8.

for the three models in the table, as well as a range of other models, real times are either consistent with or slower than the times where random partitioning is combined with SC. The improvements on communication are still negated by inefficient balancing. The sent states for LP6, BA5, and AD8, are now 1.5 times, 1.8 times, and 2.1 times smaller than with randomized partitioning.

Like with local search, in most cases SC eliminates any need for distance partitioning. However, the distance partitioning can benefit models that have large states after compression. Furthermore, the distance function used here is not ideal; we need to improve the distance function such that fewer cross-transitions exist between the buckets in the distance table. This is something to consider as future work.

5.5 The influence of buffered messages

So far we have seen that when states are compacted, both local search and distance partitioning can be ineffective. Since the compacted models run faster when the state partitions are equally sized, we use the randomized partitioning and show that it is possible to improve on these times through the buffering and grouping together of foreign states as suggested by Stern and Dill [21]. We have already seen in Section 5.3.4 that our hash (H) implementation is faster than the normal (N) algorithm when state compaction is applied, and we measure both.

On every node, foreign states are buffered until x of them belong to the same owner. These are then sent in a single message, because every message suffers some network delays. When a node is idle it sends all remaining buffered states. This implementation has only two overheads: states are copied in and out of the buffers, and every message includes an additional value indicating how many states are contained in the message. We expect the number of SENDS to be reduced roughly by x times.

Table 5.13 depicts results for various models. BA7 consists of 29.0 million states, LP7 of 38.7 million states, LF7 of 26.3 million states, and both HA15 and DP15 of 14.3 million states. In each case we list the x for which the real times were the fastest, and these times are improved for most of the models when the messages are buffered. The only exception is HA15, but it has

	No Buffer			*Buffer of size x for N*			*Buffer of size x for H*		
Model	*Time N*	*Time H*	*Sends*	*x*	*Time*	*Sends*	*x*	*Time*	*Sends*
HA15	153.78	79.60	27292764	5	270.00	14522089	5	79.63	14085072
DP15	339.71	282.53	75212832	30	273.39	2500217	30	476.67	2556457
LF7	196.24	142.36	51414861	60	124.15	1847657	60	79.57	1643220
BA7	218.44	172.02	60416619	60	145.72	2331738	70	115.35	1467578
LP7	363.68	272.95	99487691	50	221.24	2961034	70	223.59	2314899

Table 5.13: Real times (seconds) and the number of SENDS (messages) when foreign states are buffered and sent as large messages.

exceptionally small queue lengths at multiples of 100 instead of 100 000 like the other models. This means that for HA15, nodes are idle a lot of the time, and before a node can collect x states that belong to the same node, it is idle again. Thus, every SEND transfers fewer than x states, as is indicative of the large number of SENDS. On average, every message consists of 1.9 states instead of 5. The time deterioration for DP15 in H might be a result of timing issues, because in H access to the store and the receiver queue is synchronized. DP15 also has a large branching factor of 10.0 and this may also play a role. We do not investigate it further in this thesis.

5.6 Summary

In summary, this chapter contains the following results that are split into four groups; queuing strategies, local search, distance partitioning, and message buffering.

Queue-reduction strategies

- The store (S), limit (L), hash (H), and combination (LS) strategies all reduce queue lengths by at least 13%.
- The hash strategy uses only 0...5% of the queue memory required when no strategies (N) were used.

- Runtimes are not greatly affected, except for models with small branching factors. These have naturally short queues.

Local search

- Based on the results, local search does reduce communication, but in most cases the runtimes are negatively affected. Apart from HA15 and LF5, times do improve for some of the models tested, but only slightly.

- In local search some states are redundantly generated, and this slows down the times. This redundancy can be reduced by storing as many states as possible. When the redundant work factor r at depth d is such that $r \leq d$, local search works well. If not, chances are that that particular model is not compatible with local search at all.

- Since redundancy is affected by both the local search depth and the size of the cluster, local search consistently works best at small depths (between 2 and 7), and on smaller clusters.

- The models that do benefit from local search have characteristics that indicate nodes are idle a lot of the time, and this means that nodes have available time to process states from other nodes. The characteristics are large state sizes, relatively short queues, and at $depth = 1$ small CPU times in comparison with the real times. The evidence of the last characteristic is depicted in Table 5.14 for five models.

	real time/CPU time	
Model	*at depth* $= 1$	*real time speedup*
LF5	31.50	8.00
HA15	55.07	4.41
BA5	7.72	1.49
EL3	5.83	1.46
FI6	1.19	0.67

Table 5.14: The table depicts that in instances where nodes are idle frequently, local search contributes to faster speedups in running times (seconds).

- Local search has a worsening effect on times when states are small or compressed (state compaction), because the previous characteristics are made less dominant in a model. Local search is only useful when compressed states are still large.
- The queue-reduction strategies can help with local search by reducing the length of the queues, but the cost is that times are slightly slower.
- Due to higher redundancies, local search does not function well with the chosen model checking algorithms.

Distance partitioning

- This partitioning strategy reduces sent states and improves real times based on the size of the range r, but nodes receive imbalanced state partitions. When the states are small and compressed, the nodes with larger partitions will have slow CPU times which can possibly negate any time improvements on communication, because these times will be faster when states are smaller.
- Distance partitioning is only useful when states are still large after compression, and when memory is sufficient on every node to store the largest state partition.

Message buffering

- Only with message buffering can we improve on state compaction times, and the good spatial balance from the randomized partitioning function is maintained.

Chapter 6

Conclusion

In this thesis we examine issues involved in on-the-fly distributed model checking. The main focus is on reducing communication between nodes, but we also focus on methods to reduce the lengths of receiver queues. From the implementation in Chapter 4 and the results in Chapter 5 the following were achieved:

- We implemented a standard distributed state enumeration algorithm in Section 4.1.1, and extended it into two existing model checking algorithms in Sections 4.1.7 and 4.1.9. Input models and properties are written in grammars that closely resemble the layout of the state graph. When evaluated in Section 5.1, the standard algorithm showcased good speedups when we increased the size of the network.

- A number of strategies in Section 4.1.3 are implemented to reduce the lengths of receiver queues and thus avoid memory depletion when these lengths are too long on nodes. We evaluated these strategies in Section 5.2 and found that they successfully reduce the queue lengths. The hash strategy requires nearly 0% of the original queue memory. These strategies are unnecessary when models have short queues, because in such cases they incur runtime penalties.

- To reduce communication we introduced local search in Section 4.2 and implemented it on all of the algorithms. Nodes keep states from other nodes by implementing nested searches up until a threshold depth after which the children states are sent to their

owners. In Section 4.2.3 we showed how local search is not applicable to all model checking algorithms. In the evaluation in Section 5.3 we discovered that for most models local search is unnecessary as runtimes slow down and redundant work is increased, even though communication is consistently reduced. Models with short queue lengths (nodes are idle a lot of the time) and large state sizes are the only exceptions.

- We implemented an alternative partition function in Section 4.1.4 that reduces communication by assigning groups of states to the same node. When evaluated we found runtimes are improved for models with large state sizes, but because the spatial balance is lost, this partitioning is redundant when states are compressed. Future work includes finding a more efficient distance function that more closely reflects the changes between states.
- Finally, in Section 5.5 we showed that it is possible to improve the runtimes when states are compacted by buffering foreign states with the same owner into large messages.

To a large extent, our two communication-reduction techniques failed to perform well and in most cases slowed running times down. The queuing methods did provide good results; queue lengths are reduced, and running times are only slightly affected.

Final thoughts

Local search did not work as well as we expected it would. The idea is that nodes explore some foreign states locally instead of sending them to their owners. If such a state is not explored on its owner, we will have saved time on a SEND. However, local search simply generates *too many states too fast*. The time wasted on this redundancy negates the time saved by minimizing communication. In a similar fashion, distance partitioning wastes too much time due to spatial imbalances.

These two communication-reduction strategies (local search and distance partitioning) only perform well when the verification models have characteristic traits such as large state sizes and when nodes have large idle times. Thus it is important to consider whether these strategies are worth the effort. The combination of state compaction and buffered messages makes communication and memory-use efficient, and if memory is still a concern, the hash queue strategy

can be applied to further reduce memory requirements.

Appendix A

Examples of model source code

A.1 Model LP5

This model is Lamport's mutual exclusion algorithm for five processes:

```
1  model lamport:
2
3  var b : array[4] of int;
4      x, y : int;
5      N : int;
6
7  process P:
8      var i = pid, j : int;
9      state NCS:
10         trans b[i] = 1; goto q1
11     state q1:
12         trans x = i; goto q2
13     state q2:
14         trans guard (y != 255) b[i] = 0; goto q22
15         trans guard (y = 255) goto p
16     state q22:
17         trans guard (y = 255) goto NCS
18     state p:
19         trans y = i; goto q3
20     state q3:
21         trans guard (x = i) goto CS
22         trans guard (x != i) b[i] = 0; j = 0; goto q4
23     state q4:
24         trans guard (j < N and b[j] = 0) j = j + 1; goto q4
25         trans guard (j = N) qoto q5
26     state q5:
27         trans guard (y = i) goto CS
28         trans guard (y = 255) qoto NCS
29     state CS:
30         trans y = 255; goto e1
31     state e1:
32         trans b[i] = 0; goto NCS
33 end;
34
35 init: new P; new P; new P; new P; new P; end;
36
37 end.
```

A.2 Model DP15

This is the model of fifteen dining philosophers:

```
1  model Dining:
2
3  var fork: array[15] of int;
4      N : int;
5      eating : int;
6
7  process phil:
8      var i = pid, j = (i + 1) % N : int;
9      state think:
10         trans guard (fork[i] = 0) fork[i] = 1; goto one
11     state one:
12         trans guard (fork[j] = 0) fork[j] = 1; eating++; goto eat
13     state eat:
14         trans fork[i] = 0; eating--; goto finish
15     state finish:
16         trans fork[j] = 0; goto think
17 end;
18
19 init:
20     new phil; new phil; new phil; new phil; new phil;
21     new phil; new phil; new phil; new phil; new phil;
22     new phil; new phil; new phil; new phil; new phil;
23 end;
24
25 end.
```

Appendix B

Additional figures

B.1 Average degrees

Model	*Degree*
AD6, AD8	1.5
BA1	1.8
BA4	2.6
LF5, HA15	3.0
BA5, BA6, LP5	3.4
BA7, LF7	3.5
LP6	3.6
LP7	4.2
FI6	4.3
EL2	5.8
EL3	7.2
DP15	10.0

Table B.1: The selected ESML models and their average degrees (branching factors).

Bibliography

[1] S. C. Allmaaier, S. Dalibor, and D. Kreishe. Parallel graph generation algorithms for shared and distributed memory machines. In *Proceedings of the Parallel Computing Conference ParCo'97*, Bonn, Germany, September 1997. (24)

[2] H. R. Andersen. An introduction to binary decision diagrams. Technical report, 1997. Course notes on the WWW. (4, 22)

[3] J. Barnat, L. Brim, and I. Černá. Property driven distribution of nested DFS. In *Proceedings of the 3rd International Workshop on Verification and Computational Logic*, pages 1–10. University of Southampton, October 2002. (24, 25)

[4] J. Barnat, L. Brim, I. Černá, and P. Šimeček. DiVinE - the distributed verification environment. In *Proceedings of the 4th International Workshop on Parallel and Distributed Methods in Verification (PDMC'05)*, pages 89–94, Lisboa, Portugal, July 2005. (63)

[5] J. Barnat, L. Brim, and J. Stíbrná. Distributed LTL model-checking in SPIN. In *Proceedings of the 8th International SPIN Workshop on Model Checking Software*, Lecture Notes in Computer Science #2057, pages 200–216. Springer-Verlag, May 2001. (23)

[6] H. Barringer, M. Fischer, and G. Gough. Fair SMG and linear time model checking. In *In Proceedings of the Internation Workshop on Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science #407, pages 133–150. Springer-Verlag, June 1989. (8)

[7] G. Behrmann. A performance study of distributed timed automata reachability analysis. *Electronic Notes in Theoretical Computer Science*, 68(4):486–502, October 2002. (2, 25, 26)

[8] G. Behrmann, T. Hune, and F. W. Vaandrager. Distributing timed model checking - how the search order matters. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, Lecture Notes in Computer Science #1855, pages 216–231. Springer-Verlag, July 2000. (2, 26)

[9] V. Braberman, A. Olivero, and F. Schapachnik. Zeus: A distributed timed model-checker based on Kronos. *Electronic Notes in Theoretical Computer Science*, 68(4):503–522, October 2002. (2, 26, 27)

[10] V. Braberman, A. Olivero, and F. Schapachnik. On-the-fly workload prediction and redistribution in the distributed timed model checker Zeus. *Electronic Notes in Theoretical Computer Science*, 128(3):3–18, September 2004. (2, 27)

[11] L. Brim, I. Černá, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In *Proceedings of the 21st International Conference on the Foundations of Software Technology and Theoretical Computer Science (FSITCS'01)*, Lecture Notes in Computer Science #2245, pages 96–107. Springer-Verlag, December 2001. (23)

[12] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986. (4, 22)

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992. (4, 10)

[14] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state space generation of discrete-state stochastic models. *INFORMS Journal on Computing*, 10(1):82–93, January 1998. (14)

[15] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs: Workshop*, Lecture Notes in Computer Science #131, pages 52–71. Springer-Verlag, May 1981. (4)

[16] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. In *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV'90)*, Lecture Notes in Computer Science #531, pages 233–242. Springer-Verlag, June 1990. (8, 9)

[17] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28(7):1–12, July 1993. (13)

[18] A. David, G. Behrmann, K. G. Larsen, and W. Yi. A tool architecture for the next generation of UPPAAL. In *10th Anniversary Colloquium of the International Institute for Software Technology of The United Nations University*, Lecture Notes in Computer Science #2757, pages 352–366. Springer-Verlag, March 2002. (25, 39)

[19] C. Daws, A. Olivero, S. Tripakis, , and S. Yovine. The tool Kronos. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 208–219. Springer-Verlag, 1996. (26)

[20] D. L. Dill and U. Stern. A new scheme for memory-efficient probabilistic verification. In *IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, pages 333–348. Chapman & Hall, October 1996. (10)

[21] D. L. Dill and U. Stern. Parallelizing the murphi verifier. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV'97)*, Lecture Notes in Computer Science #1254, pages 256–278. Springer-Verlag, June 1997. (11, 22, 25, 26, 41, 94)

[22] D. L. Dill and U. Stern. Using magnetic disk instead of main memory in the Murphi verifier. In *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV'98)*, Lecture Notes in Computer Science #1427, pages 172–183. Springer-Verlag, 1998. (10)

[23] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier/The MIT Press, 1990. (7)

[24] M. V. Espada, S. Orzan, and J. van de Pol. A state space distribution policy based on abstract interpretation. *Electronic Notes in Theoretical Computer Science*, 128(3):35–45, April 2005. (2, 27)

[25] H. Garavel. OPEN/CÆSER: An open software architecture for verification, simulation, and testing. In *Proceedings of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, Lecture Notes in Computer Science #1384, pages 64–84. HAL - CCSd - CNRS, 1998. (23)

[26] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proceedings of the 8th International SPIN Workshop on Model Checking Software*, Lecture Notes in Computer Science #2057, pages 217–234. Springer-Verlag, 2001. (23, 41)

[27] J. Geldenhuys. Efficiency issues in the design of a model checker. Master's thesis, University of Stellenbosch, November 1999. (19, 73)

[28] J. Geldenhuys. State caching reconsidered. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software*, Lecture Notes in Computer Science #2989, pages 23–39. Springer-Verlag, April 2004. (17)

[29] J. Geldenhuys and P. J. A. de Villiers. Runtime-efficient state compaction in SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, Lecture Notes in Computer Science #1680, pages 12–21. Springer-Verlag, July 1999. (10, 22, 75, 88)

[30] J. Geldenhuys, C. P. Inggs, and H. Hansen. An asynchronous distributed nested search algorithm for LTL model checking. 2007. (2, 23, 31, 33, 45)

[31] J. Geldenhuys and A. Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. *Theoretical Computer Science*, 345(1):60–82, November 2007. (9, 17)

[32] P. Godefroid. *Partial-order Methods for the Verification of Concurrent Systems: an Approach to the State-explosion Problem.* Lecture Notes in Computer Science #1032. Springer-Verlag, 1996. (9)

[33] T. Heyman, D. Geist, O. Grumberg, and A. Shuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, Lecture Notes in Computer Science #1855, pages 20–35. Springer-Verlag, July 2000. (25)

[34] G. J. Holtzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proceedings of the 3rd International SPIN Workshop*, April 1997. (10)

[35] G. J. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64:2413–2434, 1985. (17, 21)

[36] G. J. Holzmann. Automated protocol validation in Argos: assertion proving and scatter searching. *IEEE Transactions on Software Engineering*, 13(6):683–696, June 1987. (17)

[37] G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *Proceedings of the 7th IFIP WG6.1 International Conference on Protocol Specification, Testing and Verification*, pages 339–344. North-Holland Publishing Co., Amsterdam, May 1987. (10, 21)

[38] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, September 2003. (5, 6, 9, 10, 17, 21)

[39] S. Iyer, J. Jain, D. Sahoo, and E. A. Emerson. Under-approximation heuristics for grid-based bounded model checking. *Electronic Notes in Theoretical Computer Science*, 135(2):31–46, July 2005. (11)

[40] C. Joubert. Distributed model checking:from abstract algorithms to concrete implementations. *Electronic Notes in Theoretical Computer Science*, 89(1):114–127, September 2003. (18)

[41] P. Krčál. Distributed explicit bounded LTL model checking. *Electronic Notes in Theoretical Computer Science*, 89(1):33–50, September 2003. (2, 24, 45)

[42] R. Kumar and E. G. Mercer. Load balancing parallel explicit state model checking. *Electonic Notes in Theoretical Computer Science*, 128(3):19–34, April 2005. (25, 26)

[43] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, Lecture Notes in Computer Science #1680, pages 22–39. Springer-Verlag, September 1999. (2, 11, 26)

[44] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993. (1)

[45] J. Lewi and B. Vergauwen. A linear local model checking algorithm for CTL. In *In Proceedings of the 4th International Conference on Concurrency Theories*, Lecture Notes in Computer Science #715, pages 447–461. Springer-Verlag, August 1993. (8)

[46] D. M. Nicol and G. Ciardo. Automated parallelization of discrete state-space generation. Technical report, Darthmouth College, January 1997. (24)

[47] R. Pelánek. Typical structural properties of state spaces. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software*, Lecture Notes in Computer Science #2989, pages 5–22. Springer-Verlag, April 2004. (73)

[48] D. A. Peled. All from one, one for all: on model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science #697, pages 409–423. Springer-Verlag, June 1993. (9)

[49] A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the 18th IEEE Symposium on the Foundation of Computer Science*, pages 46–57. IEEE Computer Society Press, October 1977. (7)

[50] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, Lecture Notes in Computer Science #137, pages 337–351. Springer-Verlag, April 1982. (4)

[51] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, June 1985. (17)

[52] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. (9)

[53] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, December 1992. (9)

[54] M. Y. Vardi and P. Wolper. An automatic-theoretic approach to automatic program verification. In *In Proceedings of the 1st IEEE Symposium on Logic in Computer Science*, pages 332–344. IEEE Computer Society, June 1986. (8)