Distributed Symbolic Model Checking

Tamir Heyman

Distributed Symbolic Model Checking

Research Thesis

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Tamir Heyman

Submitted to the senate of The Technion - Israel Institute of Technology

TEVET, 5764 Haifa December 2003

The research thesis was done under the supervision of Prof. Orna Grumberg and Prof. Assaf Schuster, in the Faculty of Computer Science, the Technion - Israel Institute of Technology.

The generous financial help of the Technion is gratefully acknowledged.

To my best friend since high school, my partner in life, the mother of my girl, Yafit my wife, for been on my side for so many years, through so many hard and good times. For the many good decisions you helped to make. You are my sunshine.

To Orna and Assaf, for been the most dedicated supervisors anyone could hope for. For letting me choose directions and milestones. For holding my hand and standing behind me all the way through. For working so hard, so long, so late with so much humor to meet such aggressive milestones. I thank you from my bottom of my heart.

Special thanks to my brother Amnon Heyman, for his help in debugging and designing sophisticated C++ code and for his availability around the clock.

Many thanks Karen Yorav, my old friend, and to Ran Wolff, my room-mate, for being supportive and giving so much wise advice.

Many thanks are due to Sharon Kessler for her time, patience and help with many technical English texts.

Contents

List of Figures

List of Tables

Abstract

This work presents a scalable method for parallelizing symbolic reachability analysis on a distributed memory environment of workstations. We have developed an adaptive partitioning algorithm that significantly reduces space requirements. The memory balance is maintained by dynamically repartitioning the state space throughout the computation. A compact BDD representation allows coordination by shipping BDDs from one machine to another. This representation allows for different variable orders in the sending and receiving processes. The algorithm uses a distributed termination protocol, with none of the memory modules preserving a complete image of the set of reachable states. No external storage is used on the disk. Rather, we make use of the network, which is much faster.

In order to implement a better algorithm we had to develop the Division system. The Division system is a generic distributed system for research on distributed model checking. Our preliminary experimental results show that the algorithm is indeed work-efficient. Although the goal of this research is to check larger models, the results also indicate the potential to obtain high speedups, because communication overhead is very small.

Division enables the development of a novel distributed symbolic algorithm for reachability analysis that can effectively exploit, "as needed", a large number of machines working in parallel. The novelty of the algorithm is in its dynamic allocation and reallocation of processes to tasks and in its mechanism for recovery, from local state explosion. As a result, the algorithm is *work efficient*: it utilizes only those resources that are actually needed. In addition, its high adaptability makes it suitable for exploiting the resources of very large distributed, non-dedicated environments. Thus, it has the potential of verifying very large systems.

We combine a scheme for on-the-fly model checking for safety properties with a scheme for scalable reachability analysis. We suggest an efficient, BDD-based algorithm for a distributed construction of a counterexample. The extra memory requirement for counterexample generation is evenly distributed among the processes by a memory balancing procedure. At no point during the computation does the memory of a single process contain all the data. This enhances scalability. Collaboration between the parallel processes during counterexample generation reduces memory utilization for the backward step.

We further propose a distributed symbolic algorithm for model checking of propositional μ–calculus formulas. μ-calculus is a powerful formalism and μ–calculus model checking can solve many problems, including, for example, verification of (fair) CTL and LTL properties. This work thus significantly extends the scope of properties that can be verified distributively, enabling us to specify sophisticated properties for very large designs.

The algorithm distributively evaluates subformulas. It results in sets of states which are evenly distributed among the processes. We show that this algorithm is scalable and therefore can be implemented on huge distributed clusters of computing nodes. The memory modules of the computing nodes collaborate to create a very large memory space, thus enabling the checking of much larger designs. We formally prove the correctness of the parallel algorithm. We complement the distribution of the state sets by showing how to distribute the transition relation.

Chapter 1

Introduction

Hardware designs are becoming larger and more sophisticated, while their development cycle is getting shorter. At the same time, VLSI technology is improving rapidly. However, the technology used to test that a product implements the specification cannot keep up with the rapid growth in hardware complexity. This is because testing is done with simulations, which cannot cover all the states in a big design. This gap between design size and verification capability is known as the verification crisis. As opposed to verification by simulation, formal verification methods have been more successful in proving that a given design implements its specification. Yet formal verification methods have also been limited in their ability to deal with large designs.

One such formal verification method widely used in the industry is model checking. A model checking algorithm gets a model and a specification written as a temporal formula. If the model satisfies the formula, it returns ‘true’; otherwise it returns ‘false’. When the algorithm returns ‘false’ it also provides a counterexample that demonstrates why the model does not satisfy the formula. The counterexample feature is vital to the debugging of the system.

In the early 1980s, procedures for model checking that were capable of handling systems consisting of a few thousands of states were proposed [25, 58, 48]. Model checking tools have successfully uncovered subtle errors in small-sized complex designs. It is the large memory requirements of these tools, known as the state explosion problem, that limits their applicability to large designs. This is their main drawback.

1.1 Distributed Symbolic Reachability Analysis

Reachability analysis is a key component, and a dominant one, in model checking. In fact, for most safety properties, model checking can be reduced to reachability analysis [9]. Thus, for safety properties, verification is possible if reachability analysis is possible.

Many approaches to reducing the memory requirements of model checking tools have been investigated. One of the most successful approaches is *symbolic model checking* [17], in which computation is done over sets of states. Nowadays, many model checkers represent these sets using binary decision diagrams (BDDs) [15]. The symbolic approach has made model checking applicable to industrial designs of medium size.

Current model checking tools can verify systems with hundreds of variables using BDD-based methods [17, 52] and falsify systems with thousands of variables using SAT-based methods [13]. A recent comparison [2] shows that each of the BDD-based and SAT-based methods is superior to the other for certain types of problems. But it is widely understood that the capability of model checking tools must be extended further. Typically, BDD-based model checking tools suffer from high space requirements while SAT-based tools suffer from high time requirements. The goal of this work is to overcome the space problem of BDD-based model checkers.

We presents a basic algorithm for distributed symbolic reachability analysis. The state space on which the reachability analysis is performed is partitioned into *slices*, where each slice is *owned* by one process. The processes perform a standard Breadth First Search (BFS) algorithm on their own slices. However, this BFS algorithm can discover states that do not belong to the slice the process owns. Such states are called *non-owned* states. When non-owned states are discovered, they are sent to the process that owns them. As a result, a process only requires memory for storing the reachable states it owns, and for computing the set of immediate successors for these states. The experimental results show that communication is not the bottleneck.

1.2 Division

In order to improve the algorithm, we developed the Division system. The Division system is a generic distributed system developed for research of distributed model checking. Division is an event driven system. An event can be the arrival of an object or method at a process. Division's processes can exchange any object including objects that have BDDs. Furthermore, this generalization adds almost no computation and communication overhead.

The system includes a simple and small interface to an external sequential model checker (e.g., NuSMV). This allows Division to work with several external sequential model checkers, each implementing the same interface. Furthermore, the system can potentially benefit from the external model checker's sequential optimizations.

Division provides the functionality required for distributed symbolic computation. It supports transmitting BDDs, slicing them, and invoking a process with a sequential model checker. Transmitting BDDs is very efficient and the size of objects that include BDDs are translated to messages with smaller size than the original object.

The Division user is a researcher in the distributed model checking field, who wants to implement a new algorithm. Division was designed to minimize the changes that take place in the system when a new algorithm is implemented. The layered design of Division restricts these changes to a specific layer at a time.

1.3 A Work-efficient Distributed Algorithm for Reachability Analysis

We suggest a new distributed algorithm which overcomes the drawbacks of the previous, basic one. The algorithm uses two types of processes: coordinators and workers. Each worker can be either active or free. The algorithm works iteratively. It is initialized with one active worker that runs a symbolic reachability algorithm, starting from the set of initial states. While the algorithm is running, workers are allocated and freed, as needed. At any iteration, each of the active workers applies image computation and then sends those states it does not own to their owners. Therefore, we will refer to these as a worker's non-owned states.

Our algorithm is designed to overcome the problem of memory overflow that is likely to occur during image computation and the exchange operation. For image computation we use a new BDD operation that resembles ordinary image computation, except that it stops if the intermediate results create memory overflow. In this case, the BDD representing the intermediate results is partitioned into k slices. One slice is left with the overflowed worker and the others are distributed to $k-1$ free colleagues. k is called the *splitting degree*. It is a parameter of the new algorithm and is usually small (often $k = 2$). Since the BDD is huge, the slicing is very effective. Once the BDD is split, each worker resumes the computation of (its part of) the image *from the point at which it stopped*. However, each worker now works on a smaller BDD. If state explosion occurs during the exchange procedure, then the BDD is split for sharing with $k-1$ free colleagues. Exchanging of non-owned states then proceeds according to the new ownership.

The new algorithm enables the slicing procedure to split according to set of new states, the set of reachable states or intermediate results, depending on what caused the memory overflow. Since the chosen BDDs are large, slicing is always very effective. Furthermore, slicing affects the performance of the new algorithm much less than it affects the basic one because, in the case of a high work load at one of the coworkers, the new algorithm can simply split again. Therefore, the new algorithm can use a more heuristic and faster slicing algorithm in order to reduce the slicing overhead. These features provide the new algorithm with strength and flexibility.

It may also happen that the memory requirement of a worker decreases below a certain threshold (the size of a BDD may decrease even if it represents a larger set of states). In that case, several workers with small memory requirements are combined and all but one become free.

It is important to note that splitting occurs only "as needed," when a worker actually has a memory overflow. Thus the algorithm is *work efficient*: it exploits to the maximum the resources of the active workers before allocating additional ones. This efficiency allows, for a given network, computing reachability of (i.e., verifying) larger systems. Moreover, our algorithm can effectively exploit any network size. Thus, the larger the available network, the larger the systems that can be verified.

1.4 On-the-Fly Model Checking

Another approach for dealing with the state explosion problem is *on-the-fly model checking*. In this approach parts of the model are developed whenever the need arises. The check is usually guided by an automaton that *monitors* the behavior of the system in order to detect erroneous behaviors. A lot of work is saved because the algorithm stops as soon as an erroneous behavior is detected. On-the-fly algorithms [32, 57, 12] usually use a depth-first search (DFS) to traverse the state space. Therefore they do not work well with BDD-based methods.

In [9], on-the-fly model checking of regular expressions has been reduced to reachability analysis. As a result, it can be implemented with BDDs. Furthermore, this method uses a monitoring automaton whose size is linear in the length of the formula, whereas other methods use automata of exponential size. The method can handle a large class of safety properties, including RCTL(subset of CTL that can be expressed in regular expressions).

We extend the basic reachability analysis algorithm to on-the-fly model checking of regular expressions. Our method includes a distributed algorithm that employs several processes for counterexample generation: the entire set of states is never held in a single process. Therefore, this extension is scalable and can handle large designs.

Producing the counterexample requires additional storage of sets of states during reachability analysis, one set for each step. In the distributed algorithm each process stores only part of each set. In order to balance the parts of the sets across the processes, we apply a slicing function that defines for each process the parts of the set it should store. The parts a process stores may belong to different parts of the state space. This makes the distributed counterexample generation somewhat tricky: we need to track the steps backwards while switching different slices and maintaining the memory requirement at a low level.

1.5 Distributed Symbolic μ-calculus

We extend the specification language even further. It presents a distributed symbolic model checking algorithm for the μ-calculus. The *μ-calculus* is a powerful formalism for expressing properties of transition systems by means of least and greatest fixpoint operators. Many temporal and modal logics can be encoded by the μ–calculus. Moreover, model checking algorithms for μ–calculus work particularly well with BDDs. This is due to the fact that the μ–calculus model checking is based on set manipulations, for which BDDs are particularly suitable.

Burch et al. showed that many verification problems can be solved by translating them into μ–calculus model checking problems [17]. One such problem is CTL with fairness (fair CTL) model checking. Fairness is essential for many aspects of modeling and specification. It is used, for instance, in describing the environment in which a system executes. It is also used for excluding unrealistic behaviors which have been added to the model due to abstraction. Other problems that can be solved by translating them into μ-calculus model checking problems include LTL model checking, bisimulation equivalence, and language containment of ω-regular automata [17].

Many sequential algorithms for μ-calculus model checking have been suggested [34, 65, 69, 29, 49]. The basic algorithm works bottom-up through the formula, evaluating each subformula based on the evaluation of its own subformulas. A formula is interpreted as the set of states in which it is true. Thus, for each μ–calculus operation, the algorithm receives a set (or sets) of states and returns a new set of states.

The distributed algorithm follows the same lines as the sequential one, except that each process runs its own copy of the algorithm and each set of states is stored distributively among the processes. Every process *owns* a slice of the set, so that the disjunction of all slices contains the whole set. An operation is now performed on a set (or sets) of slices and returns a set of slices. As with previous algorithms none of the processes stores a whole set. Therefore, this extension is scalable and can handle large designs.

1.6 Network of Workstations (NOW)

There are several platforms that allow the implementation of distributed algorithms. We used a network of workstations (NOW) as our computing resource. A NOW is a computing system that uses an entire workstation as its building blocks. These building blocks are interconnected by a local area network. In this system, a distributed algorithm can utilize both the computation power and memory resources of the workstations in the network. Moreover, this computing resource uses an existing infrastructure.

A NOW system is attractive as a memory resource because of its memory capacity, access time and bandwidth. The current access time of a standard network is 1000 times faster than disk access time but 1000 times slower than main memory access time. In the future [3], network access time is expected to further exceed disk access time while maintaining its present access time in comparison to the main memory. Currently the network bandwidth is 1 Gbit per second for a standard network.

The algorithms that are presented here are scalable. By utilizing the scalability of a NOW system, they become scalable solutions capable of handling large designs.

1.7 Thesis Organization

Chapter 2 compares and contrasts related works and discusses these works in relation to ours. Chapter **??** presents the principle of distributed symbolic model checking as well as a basic algorithm for distributed symbolic reachability analysis. Chapter 4 presents the Division system, which was developed as an infrastructure for the implementation of much more sophisticated algorithms. Chapter 5 presents a new approach for distributed symbolic model checking. The new approach is work efficient and splits the BDDs only ”as needed.” Moreover, it includes a new BDD operator for image computation that allows the distributed algorithm to overcome memory overflows. Chapter 6 extends the distributed symbolic reachability analysis to distributed on-the-fly symbolic model checking of regular expressions. Chapter 7 extends the specification language even further. It presents a distributed symbolic model checking algorithm for the μ-calculus. Finally, Chapter 8 presents our conclusions.

Chapter 2

Related Work

2.1 Reachability Analysis with Partitioned BDDs

Narayan et al. [56] present a partitioned ROBDD (PROBDD). Their work presents a new data structure to store and manipulate Boolean functions. The new data structure consists of k partitions, each of which is responsible for (*owns*) part of the work. They use the concept of a window function, which is also mentioned in [45]. A Boolean function f is sliced by a set of window functions $w_1, \ldots, w_k$. The set of window functions is complete, i.e., $\vee_{i=1}^{k} w_i = 1$. Each slice f_i is equal to $f \wedge w_i$ and f is represented by $\{(w_i, f_i) \mid 1 \leq i \leq k\}$. For some cases it is more efficient to choose disjoint windows, i.e., $\forall 1 \leq i \neq j \leq k . w_i \wedge w_j = 0$.

In their work it is assumed that the set of window functions is the same for all functions. It is further assumed that the ith partition in all functions has the same variable ordering. Under these restrictions, given two Boolean functions f and g and a set of window functions $w_1, \ldots, w_k$, the following holds. (a) The PROBDD representing $\overline{f}$ (i.e., NOT f) is $\{(w_i, w_i \wedge \overline{f_i}) \mid 1 \leq i \leq k\}$. (b) The PROBDD representing $f \odot g$ where $\odot$ represents any binary operation between f and g is $\{(w_i, w_i \wedge (f_i \odot g_i)) \mid 1 \leq i \leq k\}$.

There are two differences between their work and ours. The most important is that theirs is sequential. Therefore, their algorithm requires additional developments before it can be applied to distributed systems. Furthermore, our work is more general because we allow different sets of window functions for each function in order to maintain the efficiency of the original slicing.

Narayan et al. [55] show a way to compute reachability analysis using PROBDD sequentially. They make the following observation: Given a set of window functions $w_1 \ldots w_k$ that slices a set of states R into $R_i = R \wedge w_i,\ for 1 \leq i \leq k$, the image of R_i under T_{ii}, where $T_{ii} = w_i(s) \wedge w_i(s') \wedge T(s, s')$ lies completely within the partition i. Similarly, the image of R_i under T_{ij}, where $T_{ij} = w_i(s) \wedge w_j(s') \wedge T(s, s')$ lies completely within the partition j. Multiple steps of image computation are performed on each R_i under T_{ii}. In accordance with the above observation, these steps add states only within partition i. Once a fixed point is reached within a partition i, the transition relation T_{ij} is used to compute the states for each additional partition j.

Since this work uses a single machine, exchanging non-owned states at every step is too expensive. This is the reason why the authors choose to perform the image computation on owned states locally until a fixed point is reached, and only then to compute the non-owned states. The image computation for non-owned states is then performed for each partition separately. This computation is optimized to produce only the non-owned states for a single partition. However, the computation includes all the states that are reachable from the local partition in any number of steps. In our method each process owns part of the state space according to a set of window functions. Each process knows the window function of all other processes. Each process computes all the states that are reachable from the local partition in a single step. Then the non-owned states are sliced according to the set of window functions and are sent to their owners. Therefore, the two methods are not comparable.

The number of steps in the basic sequential algorithm is equal to the number of distributed steps in our algorithm. The number of steps in their approach is greater than in the basic sequential algorithm for two reasons: First, once a partition has reached a fixed-point, $k-1$ more steps are needed in order to compute, for each one of the $k-1$ remaining partitions, the non-owned states. Second, since synchronization between different partitions is done only after both partitions reach a fixed-point, the worst case scenario is one in which there is one order of magnitude more steps than a regular sequential algorithm.

In [55] the authors comment that they believe their algorithm can be parallelized. This, however, would have required additional developments. In order to exploit the full power of the parallel machinery, it was necessary to adapt the BFS for asynchronous computation. We had to coordinate and minimize communications, avoid unnecessary blocking, and employ a distributed termination detection scheme.

Cabodi et al. [21] present a way to compute reachability analysis using a different method of BDD partitioning. The computation is partitioned into three levels. At the top level, the frontier is broken into several subsets, each smaller than a given threshold. The image computation is then applied to each subset. After image computation is completed, the big sets are re-sliced and the small sets are merged.

The authors show that an image computation can be broken down internally into two additional levels: the image computation level and the conjunction steps within the image computation. In the first level the current state and the transition relations are decomposed using the same variable. The image of the original set is the disjunction of images of each subset. The disjunction is obtained by using the appropriate decomposed transition relation. The second level is suitable for partitioned transition relations. In this situation the image computation is obtained through several micro-steps. The authors claim, but do not prove, that the first level can be applied in a micro-step.

Their approach differs from ours in each of the three decomposition levels. Since their decomposition at the top level creates duplication of states, several partitions may still include the same state after the image computation is complete. This duplication reduces the effectiveness of their decomposition, whereas we proved in Chapter 7 that in our approach the subsets remain disjoint.

Their next level of decomposition, the image computation level, partitions the current states and the transition relation using one of the current variables. In our basic algorithm we partition the set using only the current states and use the restrict operator on the transition relation. Using this operator is equal to or better than partitioning. In Chapter 7 we present the Sliced Transition Relation, obtained from slicing the transition relation not only according to the current states, but according to the next states as well.

The conjunction steps level is extended in our work in two ways. First, we prove the validity of splitting the micro-step. Second, we split the micro-step in order to find a better way of dividing the work among processes by splitting the BDDs according to the complexity of the image computation. This is in contrast to Cabodi et al. who split the micro-step only for small image computation that does not justify their first level of internal decomposition.

Cabodi et al. [22] present a faster heuristic method for finding a BDD variable to slice a Boolean function. They first estimate, for each variable, the number of nodes that appear in the cofactor, and then select the variable to slice accordingly. They traverse the function once and estimate the number of nodes in the cofactor of each variable. Given a variable v from the support of a function f, the sizes of the two cofactors, f_v and $f_{\overline{v}}$, are estimated. The estimation is based on two sets of nodes: $Lvl(f, v)$, the set of nodes marked with v, and $Rch(f, v, \phi)$, the set of nodes that has a path to a node that is connected by ϕ edges (with $\phi \in 0, 1$) to a node in $Lvl(f, v)$. The authors observe that $Rch(f, v, 0)$ is a potential overestimation of nodes outside the cofactor. Since the nodes in $Lvf(v)$ disappear in f_v and $f_{\overline{v}}$, they are excluded from the resulting estimations of $\mid f_v \mid$ and $\mid f_{\overline{v}} \mid$. The estimated size of f_v is $\mid f \mid - \mid Lvl(v) \mid - \mid Rch(v, 0) \mid$ and the estimated size of $f_{\overline{v}}$ is $\mid f \mid - \mid Lvl(v) \mid - \mid Rch(v, 1) \mid$.

Ferare et al. [36] present a different way to compute reachability analysis using a partitioned BDD. A different heuristic is used to traverse the function once and estimate the number of nodes that appear in the cofactor of each of the variables. For a BDD that represents a function F, each node in that BDD is the root of a sub-DAG represents a sub-function f. For each such BDD node representing a function f, they estimate the number of nodes in the sub-DAG of f and mark this as $c[f]$. Clearly, $c[f] \leq c[f_0] + c[f_1] + 1$, and equality is obtained when there is no sharing between f_0 and f_1. An exact $c[f]$ may be calculated as $c[f] \leq c'[f_0] + c'[f_1] + 1$, where $c'[f_0]$ and $c[f_1]$ are the number of unvisited nodes encountered during the traversal of f_0 and f_1, respectively. If the traversal starts with f_0, the value of $c'[f_0]$ is an exact estimation while $c'[f_1]$ is an underestimation and vice versa.

In order to compute the $c[f]$ for all f in F, two traversals are performed: one in which the 0-edge is expanded first and one in which the 1-edge is expanded first. In each traversal $c[f]$ is updated to be the maximum value of both. Once the computation of $c[f]$ is complete, a heuristic cost is assigned to each variable using a heuristic cost function, and the N variables with the minimal cost are selected. Then, a much more accurate check is performed to select the best variable.

An additional contribution of Ferare's paper is the use of a priority queue to select the next partition to be searched. The paper suggests that the best priority strategy is to select the smallest remaining partition. The algorithm starts with one set of states and computes the set of successors. If the set of successors is bigger than a given threshold, it is split into several parts. All parts are then put into the priority queue. The part with the highest priority is then extracted from the queue and the set of its successors is computed.

In each step one part is extracted from the queue. Only the successors that have not yet been developed are put back into the queue. When there are no more new states, no new parts are put back into the queue, and the computation terminates.

Another work on partitioning BDDs was done by Matsuura et al. [51]. This work show how to compress the presentation of a shared BDD (SBDD), also known as a ”multi-rooted BDD,” by partitioning. The SBDD represents a set of functions in the same BDD manager; hence, the variable order is the same in each function. Note that in many cases a better variable order can be found for each function separately. The work includes an algorithm that divides a set of functions into two subsets, called bi-partition SBDDs. Each subset then uses a different BDD manager and may use a different BDD order. An optimal algorithm is required to check all the different orders and partitions; hence it is not practical for large BDDs. The algorithm, which is heuristic and uses dynamic reorder [61], finds a bi-partitioned SBDD. The algorithm first finds the smallest possible size of each function, by applying dynamic reorder only for this function. Then it compares the minimal size of each function to its size in the current BDD order. Next, it divides the set of functions into two subsets: the functions whose minimal size is similar to their size in the current BDD order, and the others. Afterwards, the algorithm applies dynamic reorder on each subset. Finally, the algorithm consider replacing two functions and it may in fact decide not to replace them. It select one function in each subset, and check what is the size of the same be-partitioned SBDD except this function is in the other subset.

Their partitioning is focused on hardware synthesis, where there are several large functions. Therefore, their algorithm requires additional developments before it can be applied to verification, where only two large functions exist. Furthermore, they present only the partitioning method but do not explain how to manipulate the partitioned functions.

2.2 Explicit State Model Checking

Stern and Dill present a way to parallelize the explicit murφ verifier [64]. They present an asynchronous algorithm for reachability analysis. The algorithm includes a termination detection procedure and a counterexamples procedure for cases when the algorithm finds a bug. Their algorithm is intended to achieve speedup and indeed it does. The impact of the communication infrastructure on the algorithm was analyzed. This analysis included the impact of delays in communicating a small message, the time consumed in sending a message, and the minimum time between two messages. Their experiments show that the algorithm is insensitive to slow communication. This can be explained by the overlap between the latency and the communication.

Each process has a unique ID but runs the same code. A hash function maps each state to one of the processes. The randomness of the hash function provides random load balance. Probability methods are used to prove that there is a high chance for very even distribution of the state space over the nodes. A process first computes the set of successors from one of its states. Next, it performs symmetry reduction, and only then sends the states to their owners.

The termination detection algorithm starts when the master sends a message to each of the processes telling them to report their status. Each process reports the number of messages it sent, the number of messages it received, and the number of states it has to develop. The master compares the sum of sends to the sum of receives: if they are equal and none of the processes has undeveloped states, it terminates the algorithm.

The main difference between this algorithm and ours is that it is explicit while ours is symbolic. The explicit algorithm uses the fact that each of the states is manipulated separately to divide the work among the processes. The experimental results show that the randomness of the hash function indeed distributes the reachable states evenly, but there is no way to guarantee this in general. Our slicing algorithm uses sets of states represented as BDDs and therefore cannot use a hash function. However, our method continually slices processes that have a large amount of work. In this way, the work load is balanced dynamically.

2.3 Parallel Algorithms for Constructing BDDs

Kimura and Clarke present a parallel algorithm for constructing binary decision diagrams [46]. They consider a BDD as an automaton and a BDD operation as the automaton product construction that is followed by a minimization phase. The sequential product construction phase creates new states. A new state is a pair of states, each of which belongs to an automaton. After a new state is created, it is connected by an edge to its predecessors. When the product construction ends, the minimization phase starts and all redundant states are deleted.

The parallel algorithm, implemented on a shared memory machine, starts with a single process. The process starts the construction phase and creates a new state; then it delivers its successors to other processes. When the other processes complete the construction and minimization phases, this process starts the minimization phase and completes the construction. This method imposes a total order on the processes because they are invoked in layer by layer order.

Ranjan et al. use a network of workstations to implement BDDs [59]. They assign one or more layers, depending on their size, to each process. The process that owns the upper and lower layers owns more layers than the process that owns the middle layers. This work defines a generalized address for a BDD node that is obtained from its BDD variable ID and its address in the computer that owns this node. The use of a variable ID as part of the address makes it possible to identify, without any communication, which machine this nodes belongs to.

A BDD operation starts with the process that owns the top layer. This process starts in the apply phase. When this process reaches a BDD node that belongs to a layer lower than its own lowest layer, it creates a request for that node and puts it in the appropriate request queue. When the process receives the results for all the requests, it starts the reduction phase. In the reduction phase the process sends nodes to processes that own layers above the highest layer the process owns. The reduction phase continues until the process sends all the nodes it does not own. The assignment of BDD sections imposes a total order on the processes.

Since the computations are carried out one process at a time, only the memory resources of the workstations are exploited. Moreover, the variable indices are statically distributed over several processors. One disadvantage of this approach is that if the number of nodes in certain levels grows very large, an uneven distribution of nodes may result.

Both Kimura et al. and Ranjan et al. work at the BDD level. The disadvantage is that because each BDD operation involves communication, scalability is limited by communication overhead. Moreover, since BDD accesses are pseudo random, applying BDD operations requires many random accesses, thus aggravating communication. The more machines in the system, the higher the ratio between remote and local accesses. In contrast, all accesses of a BDD operation in our method are local, except at the end of each forward step. Furthermore, the volume of communication in our method does not depend on the internal BDD data structure. In particular, it does not depend on the pseudo-random process of accessing BDD nodes.

2.4 Sequential Optimization

The distributed approach to model checking is orthogonal to other efforts in improving the capacity of current model checking algorithms. Many techniques have been developed in order to reduce the size of the model for sequential algorithms. These techniques are called *model reduction* techniques, and they include abstraction [26], symmetry [27, 33], partial order [54, 68], static analysis [18, 19], and manual reductions [6].

The BDD variable order affects the BDD size significantly. Many efforts have been made to find a variable order that reduces memory requirements [61, 1, 20, 44, 53, 62, 63, 67].

Since finding the best variable order is an NP-hard problem, these approaches are heuristics. They all apply a dynamic reordering procedure during the model checking computation. When the reordering procedure is invoked, it usually reduces the BDD size.

Chapter 3

Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits

3.1 Introduction

Reachability analysis is a key component, and a dominant one, in model checking. In fact, for most safety properties, model checking can be reduced to reachability analysis [9]. Thus, for safety properties, verification is possible if reachability analysis is possible.

This chapter presents a scalable parallel algorithm for performing reachability analysis on very large circuits [42]. Our algorithm relies on partitioning the state space and it actually parallelizes the computation of the different tasks. Our method parallelizes symbolic reachability analysis on a network of processes with disjoint memory that communicate via message passing. Its main characteristics are as follows:

- An *adaptive* cost function, used in partitioning the memory representation, results in k *small* and *balanced* slices with *small duplication*. This significantly increases the size of the overall state space that can be handled by the reachability analysis.
- Balance is initially obtained by iteratively partitioning the largest slice into two smaller ones, until k slices are obtained. Balance is maintained through the entire computation by pairing large slices with small ones and re-slicing their union. Re-slicing of different pairs is performed in parallel.
- Termination of the reachability analysis is detected, with none of the processes preserving the complete image of the reachable states. Balancing is important in order to avoid overflow in one of the processes while other have free memory.
- A compact and efficient BDD representation is used to allow shipping BDDs between processes by means of store, send and restore operations. Furthermore, the processes do not have to use the same BDD variable order when passing BDDs among themselves.

- No external storage is used such as disk. Rather, we make limited use of the network, which is much faster.

We now describe our method in greater detail. The state space on which the reachability analysis is performed is partitioned into *slices*, where each slice is *owned* by one process. The processes perform a standard Breadth First Search (BFS) algorithm on their own slices. However, this BFS algorithm can discover states that do not belong to the slice that the process owns. Such states are called *non-owned* states. When non-owned states are discovered, they are sent to the process that owns them. As a result, a process only requires memory for storing the reachable states it owns, and for computing the set of immediate successors for these states. The experimental results in Section 6.4 show that communication is not the bottleneck. We can thus conclude that the number of non-owned states found by a process is usually small.

Computation on a single slice of a set generally requires less memory than computation on the whole set. The *memory reduction*, which is the ratio between the original set size and the largest slice, determines the success of the partitioning. Thus, large memory reduction enables the reachability analysis of models larger than those which can be analyzed using sequential methods. Furthermore, applying parallel computation reduces execution time.

Effective slicing should significantly increase the size of the overall state space that can be handled. Obtaining an effective slicing is, however, no simple matter, since low memory requirements of BDDs are based on *sharing* among their parts. Our slicing procedure is therefore designed to avoid as much duplication as possible in the partitioned slices. This is achieved by means of an *adaptive* cost function. This function chooses slices with small duplication. At the same time, it avoids trivial partitioning in which one part is of size 0 and the other contains the whole set. Experimental results show that our procedure results in significantly better slicing than can be obtained by fixed cost functions (e.g., [21, 55]).

Memory balance is also instrumental in making parallel computation more effective. The balance obtained by the initial slicing may be destroyed as more reachable states are found by each process. Therefore, balancing is dynamically applied throughout the computation by means of a procedure that keeps the memory requirement more or less uniform for the different processes.

Our method requires passing BDDs between processes, both for sending non-owned states to their owners and for balancing. For this purpose, we developed a compact and efficient BDD representation as a buffer of bytes. This representation allows for different variable orders in the sending and receiving processes.

We implemented our technique on a loosely-connected distributed environment of workstations, embedded it in a powerful model checker called RuleBase [8], and tested it by performing reachability analysis on a set of large benchmark circuits. Compared to execution on a single machine with 256MB memory, the parallel execution on 32 machines with 256MB memory uses much less space, and covers more steps when the analysis overflows. Our slicing algorithm achieves a linear memory reduction factor which is maintained throughout the analysis by the memory balancing protocol. By a linear memory reduction factor we mean that when the number of slices increases the size of each slice decreases linearly. The timing breakdown shows that the communication is not a bottleneck with our approach, even when the network is relatively slow.

3.2 Parallel Reachability Analysis

The set of reachable states is usually computed by applying a Breadth First Search (BFS), starting from the set of initial states. In general, two sets of nodes have to be maintained during the reachability analysis:

1. The set of nodes discovered so far, called `reachable`. This set becomes the set of reachable states when the exploration ends.
2. The set of reached but not yet developed nodes, called `new`.

The right side of Figure 3.1 gives the pseudo–code of the sequential BFS algorithm.

```
mySlice = receive(fromSingle)
reachable = receive(fromSingle)
new = receive(fromSingle)
while (Termination(new)==0) {
    next = nextStateImage(new)
    next = sendReceiveAll(next)
    next = next $\cap$ mySlice
    new = next $\setminus$ reachable
    reachable = reachable $\cup$ next
}
```

(a) BFS by one process

```
reachable = new = initialStates
while (new $\neq \emptyset$) {
    next = nextStateImage(new)
    new = next $\setminus$ reachable
    reachable = reachable $\cup$ next
}
```

(b) Sequential BFS

Figure 3.1: Breadth First Search

The parallel algorithm is composed of an initial sequential stage and a parallel stage, depicted on the left side of Figure 3.1. In the sequential stage, the reachable states are computed on a single node as long as memory requirements are below a certain threshold. When the threshold is reached, the algorithm that will be described in Section 3.3 slices the state space into k slices. Then it initiates k processes. Each process is informed of the slice it owns and of the slices owned by each of the other processes. The process receives its own slice and proceeds to compute the reachable states for that slice in iterative BFS steps

During a single step, each process computes the `next` set of states that are reached directly from the states in its `new` set. The procedure sendReceiveAll runs in parallel by all process to exchange none owned states. The `next` set contains owned as well as non-owned states. `sendReceiveAll` runs in parallel by all processes to exchange none owned states. Each process splits its `next` set according to the k slices and sends the non-owned states to their corresponding owners. At the same time, the process receives the states it owns from the other processes.

The reachability analysis procedure for one process is presented on the left side of Figure 3.1. Lines 1-3 describe the setup stage: the process is notified as to which slice it owns; it also receives the initial sets of states it needs to compute from. The rest of the procedure is an iterative computation that repeats until distributed termination is detected. The main difference between the two procedures in Figure 3.1 is the modification of the `next` set in lines 6-7 as the result of communication with the other processes.

The parallel stage requires an extra process called the *coordinator*. This process coordinates the communication between the processes, including exchange of states, dynamic memory balance, and distributed termination detection. However, the information does not go through the coordinator, but is exchanged directly between the processes.

In order to exchange non-owned states, each process sends to the coordinator the list of processes with which it needs to communicate. The coordinator matches pairs of processes and instructs them to communicate. The pairs exchange states in parallel and then wait for the coordinator to match them with other processes. Matching continues until all communication requests are fulfilled. A process that ends its interaction may continue to the next step, without waiting for the rest of the processes to complete their interaction.

3.2.1 Memory Balancing

One of the objectives of slicing is to distribute the memory requirements equally among the nodes. Initial slicing of the state space is based on the known reachable set at the beginning of the parallel stage. This slicing may become inadequate as more states are discovered during reachability analysis. Therefore the memory requirements of the processes are monitored at each step; whenever they become unbalanced, a balance procedure is executed. The coordinator matches those processes that have a very large memory requirement with processes that have a very small one. Each pair of processes then re-slices the union of its two slices, to obtain a better balanced slicing. The pair uses the same procedure that is used to slice the whole state space (described in Section 3.3), with $k = 2$. After the balance procedure is completed, the pair informs the other processes of the new slicing.

3.2.2 Termination Detection

In the sequential BFS algorithm, termination is detected when there are no more undeveloped states (i.e., `new` is empty). In the parallel algorithm, each process can only detect when `new` is empty in its slice. However, a process may eventually receive new states even if at some step its `new` set is temporarily empty.

The parallel termination detection procedure starts after the processes exchange all non-owned states. Each process reports to the coordinator whether its `new` set is empty or non-empty. If all the processes report an empty `new` set, the coordinator concludes that termination has been reached and reports this to all the processes.

3.3 Boolean Function Slicing

Model checking uses several sets of states, as well as a transition relation between these sets. In symbolic computation these sets are represented as Boolean functions. This representation allows the handling of huge sets of states; unfortunately it may be very large by itself. We can, however, partition a large set into smaller subsets whose union is the whole set. Each partition, or slice, should require less memory. Furthermore, the subsets should be disjoint. Disjoint subsets will allow us to avoid duplication of work during reachability analysis. Since sets are represented as Boolean functions, slicing is defined for this kind of functions.

Definition 1: [Boolean function slicing] [56] Given a Boolean function $f : B^n \rightarrow B$, and an integer k, a Boolean function slicing $\chi(f, k)$ of f is a set of k function pairs, $\chi(f, k) = \{(S_1, f_1), \ldots, (S_k, f_k)\}$, which satisfy the following conditions:

1. S_i and f_i are Boolean functions, for $1 \leq i \leq k$
2. $S_1 \vee S_2 \vee \ldots \vee S_k \equiv 1$
3. $S_i \wedge S_j \equiv 0$, for $i \neq j$
4. $f_i \equiv S_i \wedge f$, for $1 \leq i \leq k$

The S_i functions that define the partitioning of the state space are referred to as *slices*. The reduction in memory requirements depends on which set of slices $S_1, \ldots, S_k$ is chosen.

The low memory requirements of BDD trees are based on sharing among their subtrees (see Figure 3.2 for a BDD example). Each subtree of the BDD is stored only once even if it is used many times (node 4 in Figure 3.2, for example). After slicing, this is no longer true. A subtree of the BDD that is used by two slices and belongs to two owners is duplicated in both owners. It is the slicing itself that causes the duplication. As the duplication increases, so does the total amount of work, and further slicing is thus required in order to reduce the slice size. Since the number of possible slices equals the number of nodes in the system, we are limited in the number of slices that we can obtain. Consequently, we must try to reduce the size of the slices as much as possible. An effective slicing should therefore reduce the slice size as well as prevent duplication.

Given a model with n variables, there is a superexponential number of ways to slice a set of states in to two: 2^{2^n}. Therefore, a heuristic approach is needed. One such possible approach is to find a slicing that minimizes $MAX(|f_1|, \ldots, |f_k|)$. However, based on the above argument and on our experiments in Section 3.6, a better approach is to slice in a way that also minimizes the sharing of BDD nodes among the k functions $f_1, \ldots, f_k$.

3.3.1 Slicing a Function in Two: `SelectVar`

Our slicing algorithm, `SelectVar`, slices a Boolean function (a BDD) into two by assigning a value to a BDD variable. The algorithm receives a BDD, f, and a threshold, δ. It

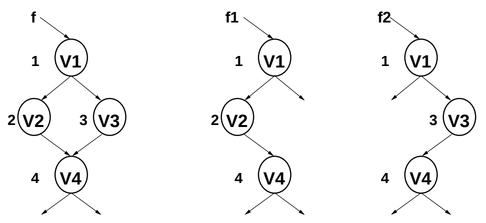


Figure 3.2: Slicing f into f_1 and f_2 using v_1

selects one of the BDD variables v and slices f into $f_v = f \wedge v$ and $f_{\overline{v}} = f \wedge \overline{v}$. Figure 3.2 shows an example of this kind of slicing, in which variable v is used to slice the function f into f_1 and f_2.

The cost of such a slicing is defined as:

Definition 2: [Cost(f, v, α):] $\alpha * \frac{MAX(|f_v|,|f_{\overline{v}}|)}{|f|} + (1-\alpha) * \frac{|f_v|+|f_{\overline{v}}|}{|f|}$

The $\frac{MAX(|f_v|,|f_{\overline{v}}|)}{|f|}$ factor gives an approximate measure to the reduction achieved by the partition. The $\frac{|f_v|+|f_{\overline{v}}|}{|f|}$ factor gives an approximate measure of the number of shared BDD nodes in f_v and $f_{\overline{v}}$ (e.g., node 4 in Figure 3.2), and therefore reflects the *duplication* in the partition.

The cost function depends on the value of α, where $0 \leq \alpha \leq 1$. An $\alpha = 0$ means that the cost function completely ignores the reduction factor, while $\alpha = 1$ means that the cost function completely ignores the duplication factor. Our algorithm uses a novel approach in which α is adaptive and its value changes in each application of the slicing algorithm. The algorithm accomplishes two important goals: **(1)** the size of each slice is below a given threshold δ, and **(2)** there is as little duplication as possible.

Initially, the algorithm only attempts to find a BDD variable that minimizing the duplication factor ($\alpha = 0$), while still reducing the memory requirements below the threshold (i.e., $\max(|f_1|, |f_2|) \leq |f| - \delta$). If such a slicing does not exist, the algorithm increases α gradually, allowing a gradual increase in duplication until $\max(|f_1|, |f_2|) \leq |f| - \delta$ is reached or $\alpha = 1$.

The threshold is used to guarantee that the partition is not trivial, i.e., it is not the case that $|f_1| \approx |f|$ and $|f_2| \approx 0$. If the largest slice is approximately $|f| - \delta$ and the duplication is small, the other slice will be approximately of size δ. This partition is not trivial, but it is unbalanced. If the largest slice is less than $|f| - \delta$ and the duplication is small, the partition will be more balanced.

The pseudo–code for the algorithm `SelectVar`(f, δ) is given in Figure 3.3. We set `STEP` $= \min(0.1, \frac{1}{k})$ and $\delta = \frac{|f|}{k}$, where k is the number of overall slices we want to obtain.

Note that even though our algorithm may compute the cost functions for many different α, $|f \wedge v|$ and $|f \wedge \overline{v}|$ are computed only once for each variable v. Therefore, computation time is not increased. Furthermore, the computation of $|f \wedge v|$ and $|f \wedge \overline{v}|$ for different variables v is done in parallel, with different computers computing the values for the different

```
$\alpha$ = $\Delta\alpha$ = STEP
$BestVar$ = the variable $v$ with minimal $cost(f,v,\alpha)$
while $((\max(|f \wedge BestVar|, |f \wedge \overline{BestVar}|) > |f| - \delta) \wedge (\alpha \leq 1))$
     $\alpha = \alpha + \Delta\alpha$
     $BestVar$ = the variable $v$ with minimal $cost(f,v,\alpha)$
return BestVar
```

Figure 3.3: Pseudo–code for the algorithm `SelectVar`(f, δ).

variables. The values are then sent to yet another computer that determines the variable with the minimal cost. Also note, that the algorithm searches for a good partitioning in the current BDD order. Each process is then invokes dynamic reorder to take the advantage of the partitioning. Rather than gradually increasing α, it is possible to find the best α using binary search. For a model with a large number of BDD variables and large k, this improvement is essential to the efficiency of our method.

Our slicing procedure is different from those of [56, 21] in that we use adaptive α, and place a marked emphasis on obtaining small duplication. Since cost functions are computed in parallel, we can allow them to be computed more precisely, thus achieving a more finely tuned slicing. The comparison to fixed α as suggested in [56, 21] is given in Table 3.2.

3.3.2 Slicing a Function into k Slices

Recall that `SelectVar` may result in two unbalanced slices that are approximately of sizes $|f| - \delta$ and δ. Unbalanced slices render our method less effective. In order to avoid this problem, we repeatedly slice the largest slice until k slices are obtained. In this way we obtain a balanced partition.

3.4 Optimizing the `SelectVar` Procedure

`SelectVar`(f, δ) selects a state variable v and uses it to split a set f into two sets: $f \wedge v$ and $f \wedge \overline{v}$. Recall that the algorithm attempts to satisfy two conditions: **(1)** that the size of both resulting sets is at most $|f| - \delta$, and **(2)** that the duplication is minimized.

The efficiency of this algorithm and its success in meeting the above conditions are crucial factors in the efficiency of the whole scheme, especially when the number of processes increases. In this section we discuss several possible improvements in the algorithm.

Our main observation is that the split of f which best meets the required conditions might not be achieved using a single variable. Indeed, it was suggested earlier that the algorithm can achieve better results by the choice of a general function g which determines two sets: $f \wedge g$ and $f \wedge \overline{g}$ [56]. However, since the number of candidates for g is exponential, it would be too time consuming to try them all. In the rest of this section we develop heuristics which help to choose a "good" g while maintaining a reasonable complexity for the choice.

We construct g iteratively as follows. Suppose that for a certain step we already have g'. We now choose a state variable v and compute the smallest cost $Cost(f, g, \alpha)$, using definition 2, of all the functions of the form $g = g' \text{ op v}$. There are 16 possibilities, 8 of which are symmetrical, since we use both g and $\overline{g}$. The option $g = 0$ means that one slice is the empty set and the other is the whole set, therefore it does not help slicing. The option $g = g'$ is the same as the current slicing. The option $g = v$ has already been checked when g' was only one variable. We are left, therefore, with the following options: $g' \wedge v$, $\overline{g'} \wedge v$, $g' \bigotimes v$, $g' \wedge \overline{v}$, $g' \vee v$. Definition 3 defines the general functions g_i, using a base function g' and a state variable v.

Definition 3: [general function $g_i(g', v)$] Let g' be a base function and v a state variable. Then the following are general functions: $g_1 = g' \wedge v$, $g_2 = \overline{g'} \wedge v$, $g_3 = g' \bigotimes v$, $g_4 = g' \wedge \overline{v}$, $g_5 = g' \vee v$

Figure 3.4 presents the pseudo–code for the algorithm `BestGeneral`$(f, g', \alpha, SliceSet)$. This algorithm chooses the best splitter from the set of $SliceSet$ and one of the general functions with g'. The set of states to be sliced is f, the base function is g' and the set of potential slicers, initially the set of all variables, to be used is $SliceSet$.

```
BestGeneral$(f, g', \alpha, SliceSet)$
$MinCost = \infty$
forall $v \in SliceSet$
  forall $i \in \{1..5\}$
    if $(MinCost > Cost(f, g_i, \alpha))$
      $MinCost = Cost(f, g_i, \alpha)$
      $g = g_i$
return $g$
```

Figure 3.4: Pseudo–code for the algorithm `BestGeneral`$(f, g', \alpha, SliceSet)$.

The complexity of `BestGeneral`, assuming that we try all state variables, can be as high as five times the number of state variables times the cost of a BDD operation. We attempt to lower the complexity of this procedure by selecting variables that are very likely to provide a good result.

In what follows, we describe how the above observation can be utilized by means of a set of heuristics that we found to be effective in selecting such a general function (see Figure 3.2). There are several configurable parameters which appear in the description of the heuristics; currently, these parameters are set in our implementation by trial-and-error.

Optimization 1. The general construction of a splitting function g, as described above, is called by `SelectVar`(f, δ) in several cases. The construction is not, however, called directly. When a splitting variable $BestVar$ is found by `SelectVar`, we first call `ImprovingSplit`. `ImprovingSplit`$(f, BestVar, SliceSet, \alpha)$ then applies `BestGeneral` using $BestVar$ as a base function with the last value of α, in an attempt to further improve the cost. In this case `BestGeneral` is applied with the set of all potential slicers. The pseudo–code for the algorithm `ImprovingSplit` is given in Figure 3.5.

```
ImprovingSplit($f, BestVar, SliceSet, \alpha$)
$g =$ $BestVar$
Loop:
  $g' = g$
  $g =$ BestGeneral ($f, g', \alpha, SliceSet$)
until $Cost(f, g', \alpha) \leq Cost(f, g, \alpha)$
return $g'$
```

Figure 3.5: Pseudo–code for the algorithm `ImprovingSplit`$(f, BestVar, SliceSet, \alpha)$.

`ImprovingSplit` is also called to apply the general construction when a splitting variable which meets the conditions cannot be found, and the algorithm is forced to increase α. In these cases, the base function for the construction, namely `BestVar`, is the variable with the lowest cost so far. This may be viewed as an attempt by `SelectVar` to find a function which meets the required conditions while α (and thus the duplication) is still small.

When α is very small, it may be misleading to choose slicing functions according to the cost. This is because, for very small α, the trivial slicing into empty part and original function is common. Obviously, this choice ensures that there will be no progress from the point of view of `SelectVar`. We thus attempt to reduce the size of the slices, even if we have to pay the price of a somewhat increased duplication. This brings us to the next optimization.

Optimization 2. (Only very small α) We choose the best splitting variable so far, and iteratively add more variables according to the general construction of the slicing function. This time, however, we first select those variables that really do decrease the size of the slices that were designated useful by Definition 4 below. Only then, out of those variables selected, do we choose the one for which the slicing function achieves a minimal cost. The pseudo–code for the algorithm `LowDuplication`$(f, BestVar, SliceSet, \alpha)$ is given in Figure 3.6.

Definition 4: [useful variable (f, g', v)] Let g' be a base function and v a state variable. Then v is useful if and only if:

$$\bigvee_{i=1}^{5} (\max(|f \wedge g'|, |f \wedge \overline{g'}|) > \max(|f \wedge g_i(g', v)|, |f \wedge \overline{g_i(g', v)}|))$$

While the previous optimizations attempt to create a general construction by a base function and a state variable, it is possible to use the same construction with two functions. Definition 3 is used with the modification that v is a Boolean function. The third optimization adds function with high potential to becoming good slicers to the list $SliceSet$.

```
LowDuplication($f, BestVar, SliceSet, \alpha$)
$g =$ $BestVar$
Loop:
  $g' = g$
  $g =$ BestGeneral $(f, g', \alpha, \{v \mid v \in SliceSet \wedge useful(f, g', v)\})$
until $Cost(f, g', \alpha) \leq Cost(f, g, \alpha)$
return $g'$
```

Figure 3.6: Pseudo–code for the algorithm `LowDuplication`$(f, BestVar, SliceSet, \alpha)$.

Optimization 3. We choose a small number l of the best variables found so far `BestVar`$_1$, …, `BestVar`$_l$, and send them as inputs to `BestGeneral`. This time we use each variable only once: we start with the best one, `BestVar`$_1$, add the second best, `BestVar`$_2$, etc. If any of the $l-1$ resulting functions meet the conditions – we are done. Otherwise, they are added to the existing list $SliceSet$. This may increase the input to the next iteration of `SelectVar` by $l-1$ functions. These functions have a high potential for becoming good slicers. The pseudo–code for the algorithm `UsingBestVars`$(f, BestVar_1, \ldots, BestVar_l, \alpha, \delta, SliceSet)$ is given in Figure 3.7.

```
UsingBestVars($f, BestVar_1, \ldots, BestVar_l, \alpha, \delta, SliceSet$)
$g' =$ $BestVar_1$
for $i = 2 \ldots l$ do
  $g =$ BestGeneral $(f, g', \alpha, \{BestVar_i\})$
  if $(\max(|f \wedge g|, |f \wedge \overline{g}|) \leq |f| - \delta)$ return $g$
  $SliceSet = SliceSet \cup \{g\}$
  $g' = g$
```

Figure 3.7: Pseudo–code for the algorithm `UsingBestVars`$(f, BestVar_1, \ldots, BestVar_l, \alpha, \delta, SliceSet)$
.

The pseudo–code for the algorithm `SelectVar` that uses all three optimizations is given in Figure 3.8.

3.5 Efficient Transfer of BDDs

As described in Section 6.3.1, processes periodically exchange BDDs during reachability analysis. In order to exchange BDDs three issues should be addressed. The address in BDD pointers refers to the sender local memory and does not have any meaning in the receiver local memory. The sender and the receiver may have different BDD order therefore

```
SelectVar($f, \delta$)
$\alpha$ = $\Delta\alpha$ = STEP
$SliceSet$ = all state variables
$BestVar$ = the slicer $v \in SliceSet$ with minimal $cost(f, v, \alpha)$
$BestVar$ = LowDuplication($f, BestVar, SliceSet, \alpha$)
while $((\max(|f \wedge BestVar|, |f \wedge \overline{BestVar}|) > |f| - \delta) \wedge (\alpha \leq 1))$
   $\alpha = \alpha + \Delta\alpha$
   $BestVar$ = the slicer $v \in SliceSet$ with minimal $cost(f, v, \alpha)$
   $BestVar$ = ImprovingSplit ($f, BestVar, SliceSet, \alpha$)
   $BestVar$ = UsingBestVars ($f, BestVar_1, \ldots, BestVar_l, \alpha, \delta, SliceSet$)
return BestVar
```

Figure 3.8: Pseudo–code for the algorithm `SelectVar`(f, δ), with the optimizations.

a function may has different BDD representation. During computation large BDD trees are shifted and may overload the communication.

The BDD data structure is a graph in which each node has two pointers to other nodes. In order to send such a graph to other processes, one should make the pointers point to the same nodes, even though the nodes addresses may be different. This is a well known in remote procedure calls world, and there exist algorithms for generating messages to move data including graph data structures [14]. However, in our case the sender and receiver may have different BDD orders. In this case the graph created at the receiver side is different from the original graph in the sender side.

Moving of data across the network is slower than other process operations. Reducing the size of the transferred data is especially important in preventing communication from becoming the bottleneck of our method.

Below we describe our solution to the problems listed above. Two utility functions are used: `bdd2msg` translates a BDD into compact `msg` data, and `msg2bdd` translates the `msg` data back to a BDD after the transfer has taken place. The purpose of `bdd2msg` is to serialize the BDD structure in order to make it suitable for raw buffer transfer.

BDD nodes represent a Boolean function f recursively. The functions 0 and 1 are represented by two special BDDs, called ZERO and ONE. Other functions are represented by a node that contains a variable identification x, and two pointers, `leftPtr` and `rightPtr`, pointing to two other BDD nodes that represent $f_{\overline{x}}$ and f_x, respectively. The function f is expressed according to the Shannon expansion: $\overline{x}f_{\overline{x}} + xf_x$.

The reduction achieved is 50% of the size of the original BDD. There are two causes for this reduction: the removal of the “next” pointer (used for garbage collection) and the fact that the number of nodes in a `msg` is limited to a predefined buffer size, which is smaller than the address space. As a results, the pointers `leftPtr` and `rightPtr` are smaller than pointers to addresses in memory.

We enumerate the variables and use the variable number instead of its name in order to save space. The `msg` data is a sequence of records. Each `msg` record has four fields: an index for that record (symbolic pointer), denoted as `Sid`; a variable's number, denoted as `Xid`; an `Sid` for the record's left successor, and one for its right successor. The index field indicates the record's location in the `msg` data. The records ZERO and ONE have special indexes.

`bdd2msg` traverses the nodes of BDD f in Depth–First Search (DFS) post order. It creates the corresponding `msg` records from the leaves upwards. Every time it creates a new `msg` record, it increments an index, which serves as the `Sid` for that record. The relationship between the actual pointer of the BDD node pointer and the corresponding `Sid` is recorded in a dictionary. Thus, when a pointer to a BDD node is encountered for the second time, the pointer is replaced by the `Sid` taken from the dictionary. The records are created in DFS post order. This ensures that every time a new record needs to be created, the `Sid`s for its left and right successors are already in the dictionary.

`msg2bdd` traverses the `msg` records sequentially from beginning to end. It creates the corresponding BDD nodes one by one as it traverses the data. Since the created BDD might have a different variable order than the original, it is not possible to use the pointers in the `msg` record "as is". Instead, the Shannon expansion is used to create the corresponding BDD node from the record. Thus, the `msg` data does not have to be in the same order as in the original BDD. The relationship between the `Sid` and the corresponding pointer of the BDD node is again recorded in a dictionary. Thus, when an `Sid` of a record is encountered as a left or right successor of another record, the `Sid` of the successor can be replaced by the BDD pointer in the dictionary. The organization of `msg` ensures that every time a new node needs to be created, the pointers for its left and right successors are already in the dictionary.

Remark: The transferred BDD f is compressed by the restrict operator described in [31]. This further reduces the size of the BDD. In order to obtain the compression, non-owned states can be treated as "don't-cares". The restrict operator changes f by adding or removing non-owned states such that the BDD size of the restricted f is smaller. The senders use the restrict operator using states non-owned by the receiver as "don't-cares". The receiving process ensures that the BDD is intersected with its slice after the reconstruction.

3.6 Experimental Results

In this section we report initial performance results achieved using our approach. We implemented our partitioned BDD and embedded it in an enhanced version of McMillan's SMV [52] at the IBM Haifa Research Laboratory [8].

Our parallel testbed included 32 RS6000 machines, each consisting of a 225MHz PowerPC processor and 256MB memory. The communication between the nodes consisted of a 100Mbps token ring. The nodes were non-dedicated; they were mainly employee workstations, which were often in use (along with the network) at the same time that we ran our experiments.

After each invocation of the garbage collector, the total size of the BDD manager was examined. If the size of the BDD manager at a certain machine exceeded 4M nodes, computation was terminated on this machine, as well as on all other machines taking part in the parallel computation, and “overflow” was declared. If overflow occurs during step t (beginning at step 0), then the maximal size of the set of reachable states is obtained at the end of step $t - 1$.

We conducted our experiments using five of the largest circuits we found in the ISCAS89 and addendum’93 benchmarks. Two additional large examples (BIQ and ARB) are components in the IBM Gigahertz processor. The characteristics of the seven circuits are given in Table 3.1.

3.6.1 Slicing Results

The success of our slicing algorithm is a crucial factor in the efficiency of the parallel execution. This success is indicated by two parameters of the obtained partition: the *duplication*, which is the ratio between the overall size of the slices and the original reachable size, and the *memory reduction*, which is the ratio between the original reachable size and the largest slice in the partition.

Table 3.2 presents the slicing results of reachable sets for four slicing methods. The figure gives results for 16, 32, 65, and 130 slices. These results are included to show phenomena which appear only towards large numbers of slices. The slicing algorithms are invoked when the size of the reachable set exceeds a threshold of 100,000 BDD nodes.

The first method selects the slicing function variable which achieves the greatest memory reduction. In the `SelectVar` algorithm, this corresponds to choosing $\alpha = 1$. The second method is the same as that used in Cabodi et al. [21]. This corresponds to choosing the splitting variable with the a fixed α, as suggested by Cabodi et al.[21]. The third method is the one presented in Section 3.3, which adapts α to select the partition with minimal duplication. The fourth method includes the optimizations described in Section 3.4, and splitting is carried out using the general function described in that section.

The table shows that the average improvements in the memory reduction factor obtained by our optimizations (adapting α and choosing a general splitting function), relate to fixed α are 25%, 22%, 18%, and 10% for slicing into 130, 65, 32, and 16 parts, respectively. These optimizations are of greater significance as the number of slices increases. Thus, opting for lower duplication when the number of slices grows is the key to better slicing, and this choice served as the guideline for our optimizations.

An average memory reduction factor of more than 55 was achieved over our benchmark suite when slicing into 130 slices. When the number of slices increases, a larger threshold should be chosen. This is because the threshold per slice, $100,000/130 = 750$ in our experiments, may be too small. On the other hand, efficiency dictates an earlier split when the bottleneck is the complexity of the slicing algorithm or the resources required by the initial sequential stage.

High level slicing results for three of the circuits, and for an additional component in the IBM Gigahertz processor GXI with 298 variables, are given in Table 3.3. Since the threshold was increased to 275,000, the slicing took place at a later step. The slicing method is the adaptive alpha without optimizations. An average memory reduction factor of more than 220 was achieved over our benchmark suite when slicing into 512 slices. This shows that when the circuits and the threshold are large enough, better slicing can be achieved even without the computationally costly optimizations.

The adaptive alpha algorithm chooses a value for alpha in every slicing. Figure 3.9 presents the value of alpha as determined at each slicing, when slicing to 32 slices. Figure 3.10 presents the average of values over all the examples. On average, the first 15 slices used a rather low value of alpha; the latter slices used a much larger alpha. This is consistent with our desire that low duplication be the dominant factor at the beginning of the slicing. After the first 15 slices are produced, we want each one to be sliced into two small parts regardless of duplication. Indeed, a high value of alpha causes memory reduction to dominate, as was explained in Definition 2.

During the memory balance procedure, the slicing function is used to reslice the unified state space of pairs of processes. The statistics for the values of alpha, as chosen by the adaptive slicing function, are given in Table 3.4. In most cases, a low value for alpha is selected.

3.6.2 Space Reduction Using Parallel Reachability Analysis

Figures 3.11 to 3.17 show the results for memory use, giving the reachable size and peak usage for every step. The state space is first sliced into a number of slices that equals the number of machines. Then, during memory balance, the initial slices are resliced. Each of the graphs compares memory use in the single machine execution to that of the parallel system.

The graphs show that scalability is obtained due to the performance of the slicing algorithm, which achieves a good memory reduction. In the parallel execution, the circuits which do overflow always cover more steps safely when the level of parallelism increases. Figure 3.16 shows the analysis process for circuit BIQ, which safely reaches step 30 with 32 machines, as opposed to step 19 with the single machine execution.

3.6.3 Timing and Communication

Table 3.5 gives the timing breakdown for reachability analysis on the benchmark suite. This table provides information regarding the ratio of computation (the *compute* column in the figure) to communication (the *exchange* column) and memory balancing (the *balance* column) in our scheme. The traversal is synchronous; hence it is interesting to note the amount of time the nodes were idle while waiting for the other nodes. The overall time spent on exchange and balance is an upper bound on the idle time. The table shows that the overall picture is fairly balanced. In other words, communication is not a bottleneck in our algorithm, despite the fact that we use a relatively slow network.

Table 3.6 shows the computation time for increasing levels of parallelism. The results show that for sufficiently large circuits, speedups increase as parallelism increases.

Circuit	#	max reachable		max new		peak		fixed point		gc
	vars	size	step	size	step	size	step	time	steps	time
prolog	117	402K	5	578K	5	1,283K	6	1,452	9	193
s1269	55	98K	3	128K	3	3,055K	5	1,601	10	296
s3330	172	839K	6	2,107K	6	4,250K	7	20,055	9	1,017
s5378	198	524K	25	157K	35	4,058K	19	136,309	44	10,163
s1423	91	3,831K	13	3,691K	13	11,413K	14	3,863	ov(14)	714
BIQ	187	853K	19	798K	19	4,468K	20	12,390	ov(20)	2,234
ARB	116	1,193K	6	1,183K	6	8,893K	7	4,112	ov(7)	788

Table 3.1: Characteristics of our benchmark suite, taken from ISCAS89 and addendum'93, and from the IBM Gigahertz processor.
All sizes are given in BDD nodes and all times in seconds. **Max reachable** is the maximal (over the steps) set of nodes already reached. **Max new** is the maximal (over the steps) set of nodes reached but not yet developed. Note that **new** may be larger than **reachable** (at any step), since the joint BDD representation of the current step **new** and the previous step **reachable** may decrease in size. The peak is the maximal size at any point during a step. In order to mask the effect of garbage collection (gc) scheduling decisions, the peak is measured after every gc invocation. Fixed point is the number of steps/time it takes to get to fixed point. Ov(x) means memory overflow at step x. The time was measured using an RS6000 machine, consisting of a 225MHz PowerPC processor with 256MB memory.

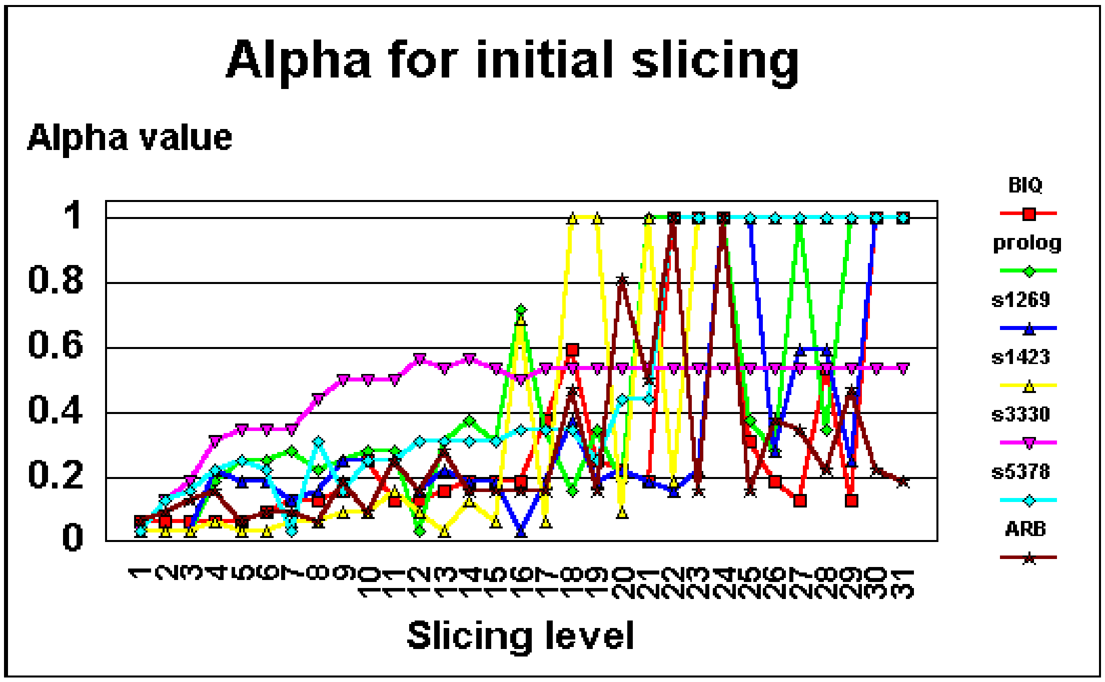


Figure 3.9: Values of α that are used when slicing into 32 parts.
The slicing procedure is called 31 times, each using a different value of α.

circuit/method	16 slices		32 slices		65 slices		130 slices	
	mem	dup	mem	dup	mem	dup	mem	dup
prolog, 4th step, reachable size is 199,961 BDD nodes								
$\alpha = 1$	8.33	1.38	14.75	1.65	23.26	2.09	37.79	2.69
fixed α	8.33	1.38	14.98	1.65	23.26	2.09	37.76	2.60
adaptive α	10.23	1.34	16.82	1.44	28.54	1.84	43.23	2.34
adaptive α + optimizations	9.89	1.22	16.66	1.44	30.01	1.66	48.87	2.04
s1269, 3rd step, reachable size is 100,170 BDD nodes								
$\alpha = 1$	12.75	1.05	22.45	1.21	36.84	1.42	60.75	1.74
fixed α	12.75	1.04	22.33	1.19	36.16	1.38	61.47	1.66
adaptive α	12.75	1.04	21.40	1.20	36.04	1.38	62.53	1.62
adaptive α + optimizations	12.75	1.02	22.33	1.18	38.53	1.27	70.42	1.41
s3330, 4th step, reachable size is 233,952 BDD nodes								
$\alpha = 1$	6.18	2.17	9.23	2.86	13.46	3.93	19.83	5.35
fixed α	6.18	2.15	9.11	2.82	13.46	3.83	19.83	5.20
adaptive α	6.02	2.16	8.92	2.82	14.00	3.68	19.94	4.72
adaptive α + optimizations	6.17	1.92	9.81	2.40	19.81	3.16	24.16	4.16
s1423, 10th step, reachable size is 175,631 BDD nodes								
$\alpha = 1$	9.87	1.24	16.65	1.37	30.51	1.51	54.59	1.74
fixed α	9.87	1.24	16.65	1.34	31.97	1.47	55.60	1.69
adaptive α	10.64	1.13	19.21	1.21	35.88	1.36	64.76	1.50
adaptive α + optimizations	11.60	1.10	18.24	1.18	38.36	1.23	70.68	1.35
s5378, 7th step, reachable size is 177,105 BDD nodes								
$\alpha = 1$	5.37	2.13	8.96	2.72	13.83	3.43	22.84	4.39
fixed α	5.88	1.95	8.96	2.72	13.83	3.43	25.29	3.84
adaptive α	7.71	1.74	11.81	2.04	19.43	2.63	28.46	3.58
adaptive α + optimizations	8.63	1.56	12.14	2.03	19.94	2.51	30.70	3.26
BIQ, 14th step, reachable size is 203,019 BDD nodes								
$\alpha = 1$	8.06	1.48	13.33	1.67	24.43	1.93	41.12	2.26
fixed α	8.54	1.44	13.60	1.60	26.87	1.86	43.35	2.15
adaptive α	8.54	1.37	16.33	1.50	29.36	1.69	50.00	1.97
adaptive α + optimizations	8.79	1.27	20.11	1.21	37.65	1.24	65.50	1.38
ARB, 5th step, reachable size is 177,105 BDD nodes								
$\alpha = 1$	10.43	1.35	17.15	1.50	28.78	1.72	48.46	1.97
fixed α	9.99	1.31	16.15	1.45	28.77	1.63	51.76	1.83
adaptive α	10.03	1.28	17.46	1.34	32.61	1.46	58.31	1.66
adaptive α + optimizations	10.72	1.08	19.64	1.17	37.03	1.26	71.21	1.27

Table 3.2: Partitioning results measured by two parameters:
the duplication **dup**, which is the ratio between the overall size of the slices and the original reachable size, and the memory reduction **mem**, which is the ratio between the original reachable size and the largest slice in the partition. Slicing threshold = 100,000 BDD nodes.

circuit	size	64 slices		128 slices		256 slices		512 slices	
		mem	dup	mem	dup	mem	dup	mem	dup
s1423	511,802	38.51	1.18	76.81	1.21	140.81	1.39	230.41	1.58
s5378	275,876	54.85	0.91	97.52	1.10	146.28	1.43	195.04	2.12
BIQ	501,752	31.82	1.51	43.14	1.79	86.29	2.21	143.82	2.51
GXI	1,561,983	42.88	1.05	83.96	1.81	166.40	1.13	312.88	1.24

Table 3.3: Partitioning results measured by two parameters:
the duplication (dup), which is the ratio between the overall size of the slices and the original reachable size, and the memory reduction (mem), which is the ratio between the original reachable size and the largest slice in the partition. Slicing threshold = 275,000 BDD nodes.

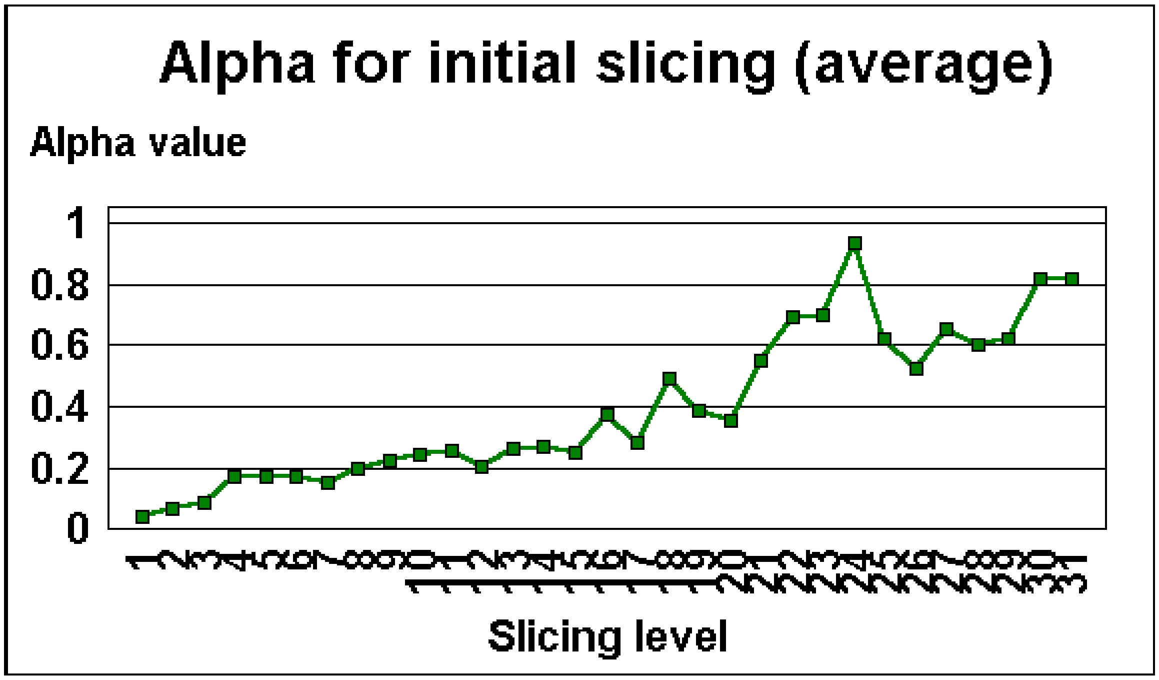
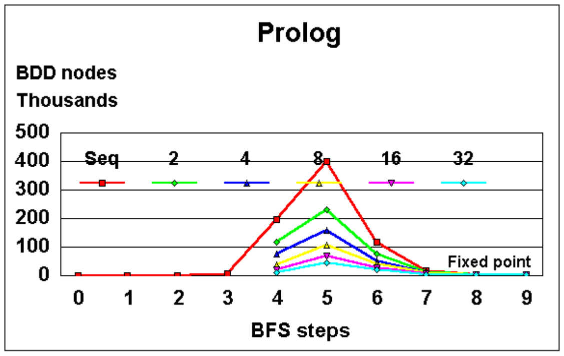


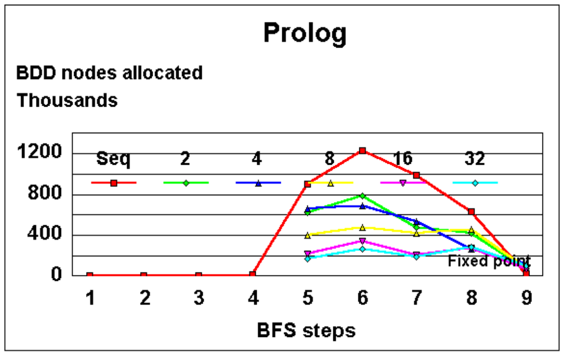
Figure 3.10: Average values of α used by the slicing process.
Since the first 15 applications are more concerned with duplication, the resulting α is relatively small. Latter applications are more concerned with memory reduction; hence, a larger α is chosen.

Circuit	# Balance	Ave	Stdev	Min	Max
prolog	82	0.165	0.116	0.1	0.8
s1269	102	0.194	0.183	0.1	1.0
s3330	32	0.133	0.058	0.1	0.4
s5378	329	0.142	0.103	0.1	0.4
s1423	83	0.138	0.068	0.1	0.4
BIQ	186	0.109	0.040	0.1	0.4
ARB	44	0.119	0.040	0.1	0.2

Table 3.4: Statistics for values of alpha by the memory balance algorithm for parallel execution on 32 machines.
# Balance gives the total number of memory balance calls during execution by all processes. **Ave** gives the average value of α. **Stdev** gives the standard deviation of the values. **Min** is the minimum value and **Max** is the maximum value.

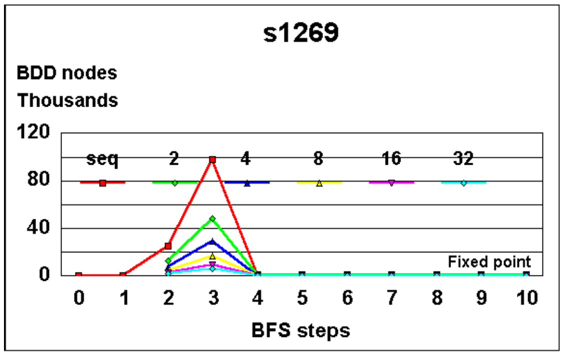


(a) **Size of reachable states set**

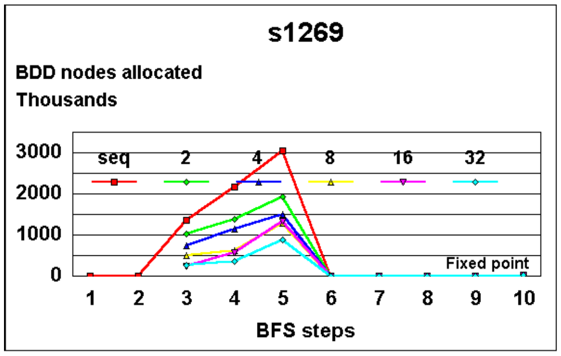


(b) **Nodes allocated (peak)**

Figure 3.11: Memory utilizations by 1,2,4,8,16,32 nodes, during reachability analysis of prolog.

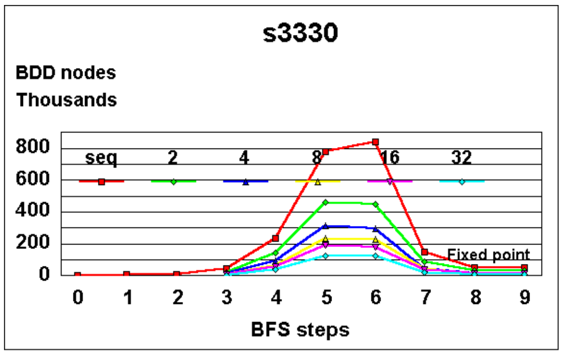


(a) **Size of reachable states set**

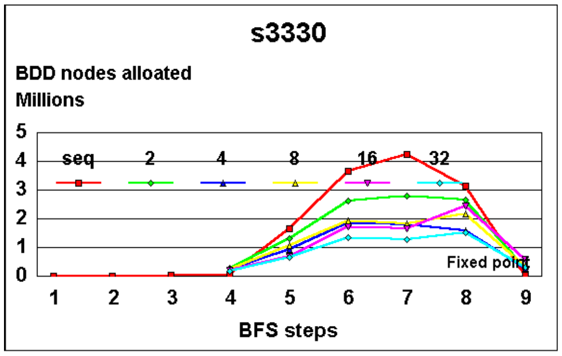


(b) **Nodes allocated (peak)**

Figure 3.12: Memory utilizations by 1,2,4,8,16,32 nodes, during reachability analysis of s1269.

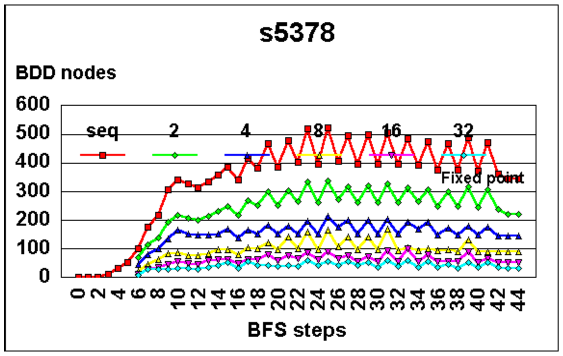


(a) **Size of reachable states set**

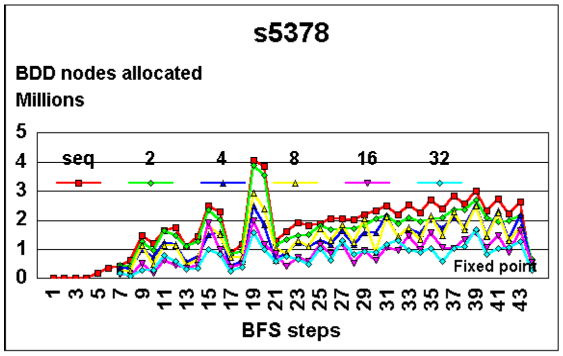


(b) **Nodes allocated (peak)**

Figure 3.13: Memory utilizations by 1,2,4,8,16,32 nodes, during reachability analysis of s3330.

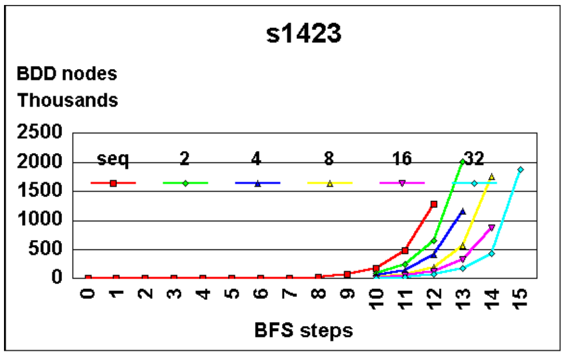


(a) **Size of reachable states set**

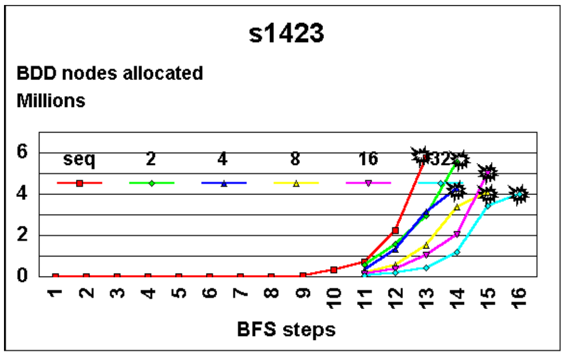


(b) **Nodes allocated (peak)**

Figure 3.14: Memory utilizations by 1,2,4,8,16,32 nodes, during reachability analysis of s5378.

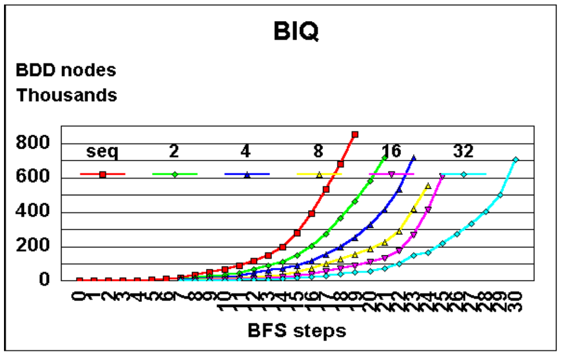


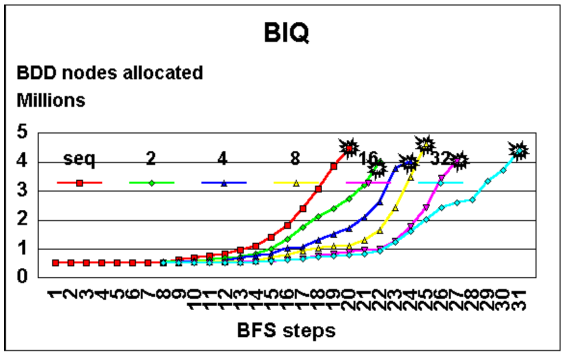
(a) **Size of reachable states set**



(b) **Nodes allocated (peak)**

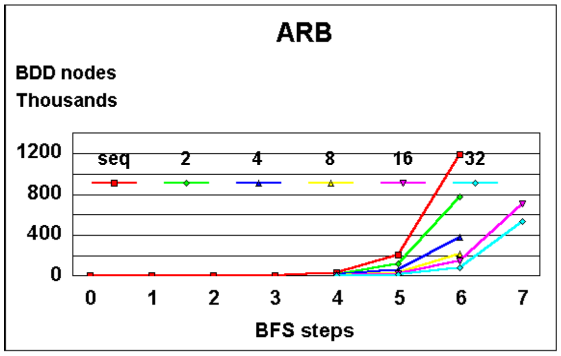
Figure 3.15: Memory utilizations by 1,2,4,8,16,32 nodes, during reachability analysis of s1423.

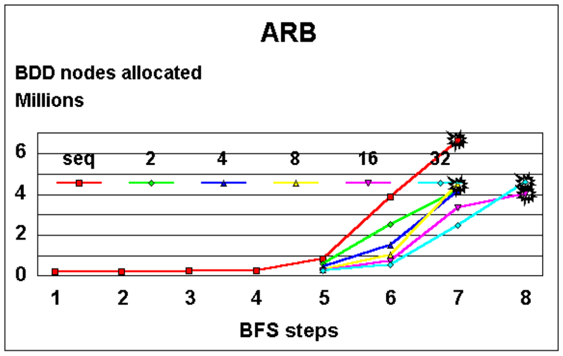




(a) **Size of reachable states set** (b) **Nodes allocated (peak)**

Figure 3.16: Memory utilizations by 1,2,4,8,16,32 nodes, during reachability analysis of BIQ.





(a) **Size of reachable states set** (b) **Nodes allocated (peak)**

Figure 3.17: Memory utilizations by 1,2,4,8,16,32 nodes, during reachability analysis of ARB.

Circuit	steps	total	seq. stage	slicing	compute	exchange non-owned	balance	gc
prolog	9	5,031	9	1,712	438	2,330	628	225
s1269	10	5,212	12	530	613	4,388	369	100
s3330	9	14,989	34	1,603	8,406	4,881	1,192	538
s5378	44	25,045	427	3,371	9,977	6,006	6,158	1,730
s1423	(12)14	12,039	144	2,221	1,697	1,721	6,036	1,043
BIQ	(20)30	50,012	74	724	25,102	7,939	16,156	5,641
ARB	(6)7	8,744	13	793	1,995	2,114	3,753	567

Table 3.5: Timing data (seconds) for parallel execution on 32 machines. Each of the measures is the worst sample over all the machines. The **steps** count shows that when overflow occurred in our system, it happened at a later stage than in the single-machine experiment (shown in brackets). The **sequential** stage shows the time it took to get to the threshold. The **slicing** gives the time it took a single node to compute the initial slicing. The total parallel is the **total** time over all steps, including **computing**, **exchanging non-owned** states, memory **balancing**, and **garbage collection** time. Note that the total time is the maxima over sums and not the sum over maxima. Note also that communication time is counted only in the exchange non-owned and balancing columns.

Circuit	seq.	2	4	8	16	32
prolog	1,452	2,410	3,789	4,682	4,078	5,023
s1269	1,608	997	1,656	1,919	2,327	5,327
s3330	20,055	20,443	10,934	10,220	12,786	14,989
s5378	136,309	53,727	50,993	61,977	24,506	25,158

Table 3.6: Timing data (seconds) for parallel execution on 1,2,4,8,16,32 machines. Each of the measures is the worst sample over all the machines. The **seq.** column gives the time it took a single node to get to fixed point. The remaining of the columns show the time it took 2,4,8,16,32 nodes to get to fixed point.

Chapter 4

Division

4.1 Introduction

The Division system is a generic distributed system developed for research of distributed model checking. The Division system includes several layers, each having a simple small interface. The system uses an external sequential model checker and can potentially benefit from its optimizations. Division is an event-driven system. An event can be the arrival of an object or method at a process. Division's processes can exchange any object including objects that have BDDs. Furthermore, this generalization adds almost no computation and communication overhead.

Division's prospective user is a researcher in the distributed model checking field who wants to implement a new algorithm, and this chapter, which describes the Division system, can be used as a reference manual. Division was designed to minimize the changes that take place in the system when a new algorithm is implemented. The layered design of Division restricts these changes to a specific layer at a time. The system assists in collecting the experimental results.

4.2 System Design

This section describes each one of the layers as shown in Figure 4.1.

The lowest layer, called the *infrastructure layer*, provides the functionality required from the infrastructure and hides any implementation details of the sequential model checker in use. This layer provides infrastructures for sequential symbolic computation and for sequential administration. It includes a simple and small interface to an external sequential model checker (e.g., NuSMV). This allows Division to work with several external sequential model checkers, each implementing the same interface.

The second layer, called the *distributed computation infrastructure layer*, provides the functionality required for distributed computation. This layer provides interfaces for communication, distributed output, transferring objects (not including BDDs), and invoking processes based on their roles. This layer requires a distributed file system and an interface for

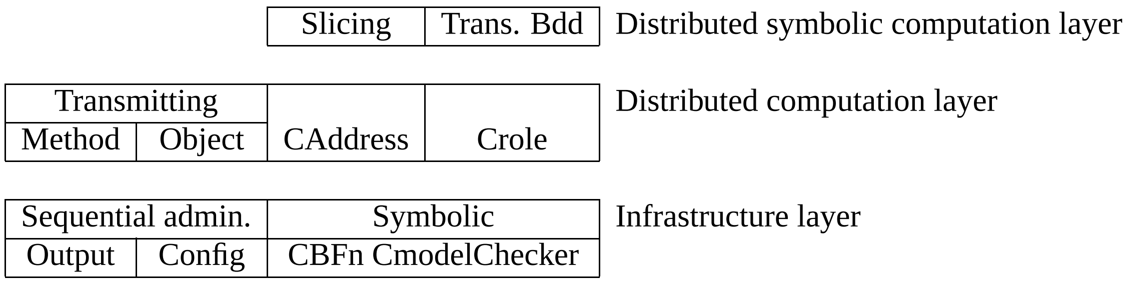


Figure 4.1: Division layers

communication.

The third layer, called the *distributed symbolic computation layer*, provides the functionality required for distributed symbolic computation. This layer supports transmitting BDDs, slicing them, and invoking a process with a sequential model checker.

4.2.1 Infrastructure Layer

This layer provides sequential symbolic functionality and sequential administration. The interface for symbolic computation includes two handler classes. One handler class represents a Boolean function (CBFn), and another handler class represents a model checker interface (CModelChecker). The interface for sequential administration consists of two mechanisms: a mechanism for configuration and a mechanism for output control.

The CBFn supplies a uniform interface for a Boolean function which is independent of the model checker in use. This interface includes the methods required for reachability analysis: getting the initial set, set manipulations, image computation, and storing a set in or retrieving it from a message buffer. This class assumes that there will be no calls to it until the CModelChecker class is invoked and the sequential model checker is initialized.

The CModelChecker supplies a uniform interface independent of the model checker in use. It has special method *Init(. . .)* for initializing the sequential model checker. During initialization the sequential model checker reads the program and translates it into BDDs. Once the initialization is completed the sequential model checker is ready for the class CBFn calls. The CModelChecker class has methods to reorder the BDD and force garbage collection.

The configuration mechanism includes a class called CConfig. This class uses a data member: this data member is a set of pairs, each containing a parameter name and a default value. Upon initialization, this class reads a configuration file and updates the parameter values. Whenever a set of parameters is required, a new class, which inherits from CConfig, is declared. The new class includes member functions that check the parameters and return their values.

The output control mechanism enables the user to collect, to filter, and to present the processes' output. The process puts a *message* in a *channel*. The channel is connected to a view that is presented to the user by a *watch*. The channel has runtime severity values, and only messages with at least that severity are allowed through the channel. The user can create multiple channels to refine the output. Views are the way the data is displayed or stored, e.g., in a file. Watches can be used to collect and organize messages in a readable way.

4.2.2 Distributed Computation Layer

This layer supports attaching an address to each process, invoking the processes with different roles, transmitting objects and transmitting methods. This layer implements the class CAddress that defines a unique address for each process. A distributed algorithm includes several types of processes, each having a different role. This layer includes the class CRole, which holds a set of pairs of addresses and roles. When a process starts to run, it uses the class CRole to get its role name, and takes action accordingly.

Transmitting objects and transmitting methods requires storing an object in a message (serialization) and retrieving the object from it. Division uses a message buffer that has methods for reading and writing elementary types. Each object that needs to be serialized has a unique code, a *store* method, and a *retrieve* method. The object store's method puts the values of the data members in a message buffer. In order to send an object or a method, the sender puts the object in a message buffer and attaches an object code in front. Then, it sends this buffer to the receiver.

The mechanism used by Division to retrieve an object consists of two stages: retrieving the object from a message, and executing a handler for this object. The object's retrieve method sets the data members according to the values in the message buffer. The receiver uses the object code to select the retrieve method that creates a new object from the message. Then the receiver executes the handler for this object code on the new object.

4.2.3 Distributed Symbolic Computation Layer

This layer supports transmitting BDDs, slicing them, and invoking a process with a sequential model checker. Division includes a mechanism for storing and retrieving BDDs, thus enabling them to be transmitted. The slicing mechanism allows many different slicing algorithms to be implemented by means of several building blocks that can be selected easily by the user.

Division uses a special message buffer, which the sequential model checker can use to store and retrieve BDDs. The CBFn store method uses the sequential model checker to place the contents of its BDD in the message buffer. The CBFn retrieve method uses the sequential model checker to get the contents of its BDD from the message buffer. In order to send a CBFn object, the sender first uses the store method to put the object's contents in a buffer and then it sends the buffer. The receiver gets the buffer and uses the CBFn retrieve method to create a new CBFn object.

The slicing mechanism includes the class CSlice, which has a set of states and a window function. This class includes a method called *split*, which can be configured to use any of the splitting algorithms. Split selects the best way to slice a given set and creates several new CSlices in accordance with the splitting degree.

A process that requires a sequential model checker uses the handler class CModelChecker. The process uses the Init(. . .) method to initialize the model checker. Then it uses the class CRole to get its role name, and takes action accordingly.

4.3 Using Division

Division is a platform for research, and as such, it includes a mechanism for running many configurations of the same example and producing graphical reports. The current distribution uses NuSMV as the external model checker, but it can be replaced by any other symbolic model checker that implements the same interface.

4.3.1 Installing Division

Division requires an external model checker, communication infrastructure, and a Unix operating system or one of its flavors (e.g., Linux). The external model checker should provide the functionality required by the interface for symbolic computation. The communication infrastructure should support MPI runs.

To install Division, download the tar file from `www.cs.technion.ac.il\tamirh` and follow the instructions in the README file. In order to ensure portability, all the sources of Division, as well as the external model checker, need to be compiled. There are several simple examples that can be run to test the installation.

4.3.2 General Support for Measuring Performance

Division includes general support for measuring the performance of a distributed algorithm. There are two types of measurements: those performed at one point in time are called statics and those performed over a period of time are called dynamics. Division uses a special process called *checker* that collects measurements from all the processes and builds reports summarizing them. In addition, we developed a methodology for running the same algorithm on different examples, using a different configuration each time.

Each process in Division can send static data to the checker asynchronously. Each process runs timers that can be reset, suspended, and resumed. Timers are used to collect dynamic data. When a process resets a timer, the timer sends its value to the checker and then clears the value. Each measure contains the name of the parameter and the measured value.

The checker process maintains several reports. Each report is an organized collection of measurements and contains a method called *print*. Print puts all the data inside a message, which can then be placed in a file. The checker adds a measurement it received to the appropriate report. The report then updates its output using the print method. The print method may place the data in such a way that it can be used by programs like GnuPlot to generate more sophisticated outputs.

Since the algorithm is going to be run on different examples using a different configuration each time, each example has a separate directory. Each directory includes a subdirectory for each of the configurations. A subdirectory includes a configuration file and, after a run, it contains all the output reported. Thus the user can collect experimental results in an organized way.

4.4 Future Development

Division version 1.0 was used to implement the work efficient distributed algorithm for reachability analysis in Chapter 5. Several important features are being considered for future versions. Communication in Division is currently based on MPI [35] version 1.0. An important feature, not supported in this version, is the addition and removal of processes after the system has started to run. An additional feature that has not yet been implemented is a checkpoint restart mechanism for fault tolerance. Finally, in order to speed up the distributed algorithm, a fast splitting algorithm based on[36, 22] is required as a building block.

Chapter 5

A Work-Efficient Distributed Algorithm for Reachability Analysis

5.1 Introduction

The algorithm in chapter 3 has several drawbacks for systems with very large number of processes. First, it immediately splits to as many slices as the number of processes in the network and does not release them until it terminates. Thus, it occupies all processes in the network all the time, regardless of actual need. Second, slicing is often inefficient because it partitions a relatively small BDD into many small slices. The more processes in the system, the less efficient the slicing is, which renders the algorithm non-scalable. Third, it does not provide a means to overcome the memory overflow that occurs during an image computation or an exchange operation. It is well known that intermediate results of image computation may be orders of magnitude larger than its initial and resulting BDDs. Similarly, during an exchange operation the memory of a process may overflow as a result of the BDDs it receives. Unfortunately, even when there are under-utilized processes, there is no way to recover from such overflows since load balancing is available only at the end of iterations. Finally, balancing is applied only to the sets R. However, the size of intermediate results in image computation depends on N and is often much larger than R. Thus, load balancing does not handle the dominant factors of memory overflow.

In this chapter we suggest a new algorithm which overcomes the drawbacks of the previous one [40]. The algorithm uses two types of processes: coordinators and workers. Each worker can be either active or free. The algorithm works iteratively. It is initialized with one active worker that runs a symbolic reachability algorithm, starting from the set of initial states. During its run, workers are allocated and freed, as needed. At every iteration, each of the active workers applies image computation and then sends those states it does not own to their owners. Therefore, we will refer to these as a worker's non-owned states.

Since memory overflow is likely to occur during the image computation and the exchange operation, our algorithm is designed to overcome these problems. For image computation we use a new BDD operation that resembles ordinary image computation, except that it stops if the intermediate results create memory overflow. In this case, the BDD representing the intermediate results is partitioned into k slices. One slice is left with the overflowed worker and the others are distributed to $k-1$ free colleagues. k is called the *splitting degree*. It is a parameter of the new algorithm and is usually small (often $k = 2$). Since the BDD is huge, the slicing is very effective. Once the BDD is split, each worker resumes the computation of (its part of) the image *from the point at which it stopped*. However, each worker now works on a smaller BDD. If state explosion occurs during the exchange procedure, then $R \cup N$ is split in to $k - 1$ free colleagues. Exchanging of non-owned states then proceeds according to the new ownership.

The new algorithm enables the slicing procedure to split according to R, N, or intermediate results, depending on what caused the memory overflow. Since the chosen BDDs are large, slicing is always very effective. Furthermore, slicing affects the performance of the new algorithm much less than it affects the one from [42] because, in the case of a high work load at one of the co-workers, the new algorithm can simply split again. These features provide the new algorithm with strength and flexibility, and allow to reduce the slicing complexity.

It may also happen that the memory requirement of a worker decreased below a certain threshold (the size of a BDD decrease even if it represents a larger set of states). In that case, several workers with small memory requirements are combined and all but one become free.

It is important to note that splitting occurs only "as needed", when a worker actually has a memory overflow. Thus the algorithm is *work-efficient*: it exploits to the maximum the resources of the active workers before allocating additional ones. This efficiency allows, for a given network, computing reachability (i.e., verifying) of larger systems. Moreover, our algorithm can effectively exploit any network size. Thus, the larger the available network, the larger the systems that can be verified.

We have implemented our algorithm in Division, a generic platform for the study of distributed symbolic model checking [38]. Division requires a model checker as an external module. We used NuSMV [23] for this purpose: a re-implementation of McMillan's SMV [52].

Unfortunately, using NuSMV implied that we could not directly compare the results of [42] to the results of this work. The experiments in [42] were conducted using the high-performance RuleBase [8] model checker that was not available to us in this work. The two tools are not comparable as many of the RuleBase optimizations are not implemented in NuSMV.

Our parallel testbed included 25 dual process PC machines. The nodes communicated via a fast Ethernet connection. We conducted our experiments using four of the largest circuits from the ISCAS89 and addendum'93 benchmarks.

With our distributed algorithm, we can compute larger models than we can compute with a single machine using the same model checker. In all the examples the new algorithm using the less sophisticated model checker (NuSMV) would be sufficient to compute the same models and reach the same BFS step as in [42].

5.2 The Worker Algorithm

Our distributed algorithm uses a set of window functions [21, 56] to partition the state space among all workers in the network. Each worker *owns* the states in one of the window functions and computes the reachable states in this window.

Figure 5.1 presents a high-level view of the workers algorithm. Essentially, the algorithm performs a reachability task. The algorithm starts with only one worker that owns the entire state space, while the rest of the workers are free. If a worker runs out of memory (*memory overflow*), it distributes parts of its work among a few free workers.

The worker repeatedly computes images and sends its non-owned states to their owners until termination is detected (namely, a fixed-point is reached). While iterating, if the workload of the worker becomes too small, it participates in a collect_small procedure.

There are two points at which a worker may run out of memory (*memory overflow*): during the image computation and during the exchange of non-owned states. Upon memory overflow, the worker splits the states it owns into two parts: one that will be processed at the current worker and another to be processed at another worker. As a result, the states belonging to the new worker become non-owned and are sent out to the new worker.

```
function reach_task()
 1 Loop until termination()
 2  Image() if overflow, split and use new workers
 3  Exchange() if overflow, split and use new workers
 4  Collect_small()
 5 return owned states
```

Figure 5.1: High–level pseudo–code for a worker

Let us describe the algorithm for the workers in greater detail, as shown in Figure 5.2. The reachability task includes a set of reachable states R and a set of reachable states that are not yet developed, N. For brevity, we omit in this section the worker subscript id from R_{id} and N_{id}, as well as the window function w_{id}. The set R is included in a window function w. The sets R and N, as well as the window function w, may change during the algorithm's execution.

In the **Image** procedure, the worker computes the set of states that can be reached in one step from N and stores the result in a new N. However, if during image computation the memory overflows, the worker splits w and updates R and N accordingly, as described below.

In the **Exchange** procedure the worker uses w to define the part of the state space it "owns". It sends out the non-owned states ($N \setminus w$) to their owners and receives its owned states that were found by other workers.

Finally, if only a small amount of work remains, the worker joins the **Collect_small** procedure. The collect_small procedure adds up the tasks of several workers, each of which has only a small amount of work. This is done by joining together the parts of the state space owned by those workers and assigning the unified ownership to one of them. The others become "free" ($w = \emptyset$) and return to the pool of free workers.

In the Image procedure, the image is computed using a new BDD operation,
The Image procedure is using a new BDD operation, `boundedImage`$(N, Max, Failed)$. This operation is different from traditional image computation in that it stops the local computation in case of a memory overflow (i.e., the number of BDD nodes exceeds Max). Upon overflow, the `Image` procedure calls the `Split` procedure, which repartitions the ownership of the worker and updates R, w and N accordingly.

In the Exchange procedure, the worker first requests and receives from the `ex_coor` process the up-to-date list of window functions owned by the other workers. The worker then sends the `ex_coor` the list of workers to whom it wishes to send non-owned states. Then, in the Exchange_loop procedure, the `ex_coor` schedules the worker for state exchange with other workers.

In the Exchange_loop procedure the worker is scheduled by the `ex_coor` to exchange non-owned states with colleagues that either found states owned by the worker or own states that were found by the worker. The worker continues to receive exchange commands from the `ex_coor` until it gets a $< done >$ command when there are no more pending exchanges. If the worker's memory overflows during the exchange procedure and the worker fails to receive more owned states, it notifies the `ex_coor` and calls the Split procedure to reduce its ownership.

If the worker in the Collect_Small procedure has enough work, it exits immediately. Otherwise, the worker notifies the `small_coor` about the sizes of its N and R sets. In reply, it receives one of three commands and proceeds accordingly: $< End >$ commands it to exit the Collect_Small; $< Non\_owner, p_{clg} >$ commands it to deliver its ownership and owned states to a colleague worker p_{clg}, waits for the `ex_coor` to acknowledge the update of its window functions (performed by p_{clg}), and then return to the pool; $< Owner, p_{clg} >$ commands it to take over the ownership and states of another worker p_{clg} and report the new ownership to the `ex_coor`.

The Split procedure starts by requesting from the `pool_mgr` $k-1$ new workers (which, together with the overflowed worker, makes it a k-way split). If Split is called from Exchange, then the window function w of the overflowed worker is split into k new window functions $\{W_i'\}$, such that $\{W_i' \cap R\}$ have approximately the same sizes. If Split is called from Image, then two sets of k new window functions are computed, as follows. If R is big enough, then, as in the previous case, a set of window functions $\{W_i'\}$ is computed such that the sizes of $\{W_i' \cap R\}$ are approximately the same. Otherwise, if R is too small, one of the workers gets all of w while the others remain empty. In any case, the ith new window function W_i' determines, for the ith worker, its new window w_i. In addition, w is split again into another set of window functions $\{Nw_i'\}$, this time making $\{Nw_i' \cap N\}$ equal in size. After the new window functions are computed, the overflowed worker sends the corresponding states to its new colleagues.

The reason for computing two different partitions when Image overflows is that $\{Nw_i'\}$ attempts to balance the current image computation, while $\{W_i'\}$ attempts to balance the memory requirement in the full reachability process. In section 5.4 we further discuss the

```
function reach_task($R, w, N,$ method)
  if method $=$ "$exchange$"
    goto Exchange_loop($R, w, N$)
  Loop_forever
    Image($R, w, N$)
    Exchange($R, w, N$)
    if (termination()) return $R$
    $N$=$N \setminus R$
    $R$=$R \cup N$
    Collect_small($R, w, N$)
    if($w = \emptyset$)
      send <$to\_pool, id$> to ex_coor
      return to pool

procedure Exchange($R, w, N$)
  <$\{w_i\}$>=receive from ex_coor
  send <$\{p_i\}$> to ex_coor
  Exchange_loop($R, w, N$)

procedure Collect_Small($R, w, N$)
  While($| N | + | R | < Min$)
    send <($| N |, | R |$)> to small_coor
    <$action$>=receive from small_coor
    if $acction =< End >$ return
    if $action =< Non\_owner, p_{clg} >$
      send <$R, w, N$> to $p_{clg}$
      $R$=$w$=$N$=$\emptyset$
      <"$release$">=receive from ex_coor
      return
    if $action =< Owner, p_{clg} >$
      <$R', w', N'$>=receive from $p_{clg}$
      $R$=$R \cup R'$; $w$=$w \cup w'$; $N$=$N \cup N'$
      send <$w, p_{id}, p_{clg}$> to ex_coor

procedure Image($R, w, N$)
  $N$ = boundedImage($N, Max, Failed$)
  While($Failed$)
    Split($R, w, N,$ "$Image$")
    $N$ = boundedImage($N, Max, Failed$)

procedure Exchange_loop($R, w, N$)
  loop until $< done >$ received from ex_coor
    <$p_{clg}, w_{clg}$>=receive from ex_coor
    send <$N \cap w_{clg}$> to $p_{clg}$
    <$N'$>=receive from $p_{clg}$
    $overflow$ = $N'$ is too large
    send <$overflow$> to $p_{clg}$
    send <<$status$>=receive from $p_{clg}$> to ex_coor
    if ($overflow$) Split($R, w, N,$ "$Exchange$")
    else
      $N$=$N \cup N'$
      send <"$done$"> to ex_coor

procedure Split($R, w, N,$ method)
  <$\{p_2 \ldots p_k\}$>=receive from pool_mgr
  if(method $=$ "$exchange$"
    $\{W'_i\}$=$\{Nw'_i\}$=Slice($R \cup N, k$)
  else
    if($| R |$ big enough
      $\{W'_i\}$=Slice($R, k$)
    else
      $\{W'_i\}$=$\emptyset, i \in 2 \ldots k$;  $W'_1$=$w$
    $\{Nw'_i\}$=Slice($N, k$)
  $\forall i \in 2 \ldots k$:
  send <$R \cap W'_i, w \cap W'_i, N \cap Nw'_i,$method> to $p_i$
  $R$=$R \cap W'_1$; $w$=$w \cap W'_1$, $N$=$N \cap Nw'_1$
  send <$\{i \in 1..k \mid w \cap W'_i\}$> to ex_coor
```

Figure 5.2: Pseudo–code for a worker in the distributed reachability computation

optimization of the partitioning process.

In the case that R is ”too small” or even empty, the new colleagues are simply helping the overflowed worker with a single image computation. Once the image is computed, all states produced by the helpers are non-owned and will be sent to other workers that own them. From our experience, this case is not uncommon; it occurs when the peak memory requirement during image computation is much larger than R.

As mentioned in the introduction, an important advantage of our algorithm over previous works is that it calls the Slice function only when the memory overflows, and with k much smaller than the total number of workers. This makes slicing much more effective in producing even splits of the input sets of states.

We remark that the Slice procedure itself is no different from the slicing mechanisms described in [42]. Thus, in this chapter, we use it as a black box and focus on the distributed algorithm itself.

5.3 The Coordinators

There are three coordinators in the algorithm: The `ex_coor` coordinator responsible for the exchange procedure. The `small_coor` coordinator collects as many under-utilized workers as possible. The `pool_mgr` coordinator keeps track of free workers. The coordinators does not take part in the computation and does not hold set of states. Instead they coordinate the workers communications. In a large distributed system they may need to be distributed as well.

5.3.1 The `ex_coor`

The `ex_coor` coordinator holds the current set of window functions and coordinates the exchange of non-owned states between workers. In order to hold a consistent view of the current set of window functions, the `ex_coor` is notified immediately on every split or merge of windows. It takes the following actions on incoming event notifications:

- When a worker requests an exchange it first *registers* at the `ex_coor`. The `ex_coor` replies with the up-to-date set of window functions and receives in return the set of colleagues the worker wants to communicate with.

- When a worker splits, the `ex_coor` updates the set of window functions. If the splitting worker is already registered for exchange states, the `ex_coor` notifies all the workers that have asked to send it states that they should send the states to the new set of workers, according to the new set of window functions.

- When workers perform *Collect_Small* and join their ownerships, the `ex_coor` updates the set of window functions. If there are workers registered for exchanging states with the joining workers, the `ex_coor` redirects them to the new owner. When the `ex_coor` completes to update the set of window functions it sends $< release >$ command to the worker that becomes non-owner.

- When a worker completes the exchange of non-owned states with another worker, the coordinator marks it as available for another round of exchange states.
- When a worker asks to re-launch an exchange because the colleague overflowed and had to split while they were interacting, the `ex_coor` adds this request to the list of exchange requests.

5.3.2 The `small_coor`

The `small_coor` coordinator collaborates with `ex_coor` to prevent deadlocks and to collect as many under-utilized workers as possible. The `small_coor` receives registration requests from workers that completed the exchange phase and are left with a very low load (very small $R \cup N$). The first registrant is blocked until more of them arrive. When there are several registrants the `small_coor` instructs them to merge.

5.3.3 The `pool_mgr`

The `pool_mgr` coordinator keeps track of free workers. During initialization, the `pool_mgr` marks all but one worker as free. When a worker invokes the Split procedure, it sends a request to the `pool_mgr` for $k-1$ free workers (where k is the splitting degree). The `pool_mgr` replies with a list of $k-1$ worker ids and removes them from the free list. Throughout the algorithm, when a worker becomes free, i.e., when its ownership becomes empty, it returns to the `pool_mgr` and is added to the free list for later assignments.

If at the time free workers are requested from the `pool_mgr`, the free list happens to be empty or is shorter than $k-1$, the `pool_mgr` announces a "worker overflow" and stops the execution globally.

5.4 Optimizing the Splitting in Image Computation Overflow

Our algorithm is based on the assumption that in case of a memory overflow during image computation, splitting the window of the overflowing worker enables the completion of the computation using more workers. The current splitting method strives to effectively slice the set N on which the image is computed (see Chapter **??**). However, since the computation is symbolic, reducing the size of the subsets does not guarantee a corresponding reduction in the image size. Furthermore, it guarantees even less for the size of the intermediate results that commonly dictate the *peak memory requirement* during the image computation. Our experience shows that even when the size of the parts is the same, the size of the peaks may differ greatly. Thus, while one of the slices may have no problem in completing the image computation, another may overflow again.

Another problem with the current splitting method is the time penalty for memory overflow. When the image computation overflows and the set N is split, the work that was

```
function ex_coor
$Ws[0]$=$full$; $Comm\_list$={}
Loop-forever
 if $\exists p_i, p_j$ s.t. $(p_i, p_j) \in Comm\_list \wedge p_i$ ready $\wedge p_j$ ready
   mark $p_i$ busy; send <$p_i, Ws[p_i]$> to $p_j$
   mark $p_j$ busy; send <$p_j, Ws[p_j]$> to $p_i$
 <$cmd$>=receive from $any\ worker$
 if $cmd =< “done”, p_{id} >$ mark $p_{id}$ ready
 if $cmd =< “resend”, p_{id}, p_{colleague} >$ add $(p_{id}, p_{colleague})$ to $Comm\_list$
 if $cmd =< “exchange”, p_{id} >$
   send <$Ws$> to $p_{id}$
   <$comm$>=receive from $p_{id}$
   add $comm$ to $Comm\_list$
 if $cmd =< “collect-small”, p_{id}, w_{id}, p_i >$
   $Ws[p_{id}]$=$w_{id}$
   $\forall p_j$ s.t. $(p_j, p_i) \in Comm\_list$
     replace $(p_j, p_i)$ with $(p_j, p_{id})$ in $Comm\_list$
     remove $p_{id}$ from list of active workers
     send <$dismissed$> to $p_i$
 if $cmd =< “split”, p_{id}, NewWs$=$\{(p_i, w_i)\} >$
   $\forall (p_i, w_i) \in NewWs$:   $Ws[p_i]$=$w_i$
   $\forall p_j, p_i$ s.t. $(p_j, p_{id}) \in Comm\_list \wedge (p_i, w_i) \in NewWs$ add $(p_j, p_i)$ to $Comm\_list$
   if $p_{id}$ already registered
     $\forall (p_i, w_i) \in NewWs$ register $p_i$; mark $p_i$ as ready
 if all $\{p_i\}$ already registered
   $\forall p_i$ s.t. (ready $p_i$) $\wedge$ (not $p_i \in Comm\_list$):  send <$“done”$> to $p_i$
 if all $\{p_i\}$ registered and $Comm\_list$ is empty
   allow registrations from next image step
```

Figure 5.3: Pseudo–code for the ex_coor.

Terminology: An *active* worker is such that is not in the pool of free workers. An active worker that completes current image computation *registers* at the ex_coor for exchanges of non-owned states. From the point of registration and until there are no more states to exchange, the worker is either *busy* in sending and receiving states to a certain colleague, or it is *ready* waiting for a colleague to register or to become ready.

invested in the current image step is lost, and the work is repeated all over again. In fact, in the case of several subsequent memory overflows, the work is repeated again and again. Notice that the ratio between the peak memory requirement in the image computation and the set N is commonly two or three orders of magnitude. Thus, memory overflow commonly occurs when a big part of the image computation has already been done locally, and all this work must be repeated. Since the image computation takes most of the time in our distributed algorithm, the repeated work slows down the algorithm substantially.

The solution to the above two problems is simply to split the intermediate results and not the set N. After the splitting, the parts of the intermediate results are distributed among the new workers, so computing the image for each of them continues from the point of the overflow. In this way there is no time penalty for overflow except for the splitting computation (which is of somewhat higher complexity than before). Of course, communicating the intermediate results requires a much higher bandwidth. However, network bandwidth and communication delay turn out to be minor factors as compared with the time spent in the image computation, even with our standard Fast Ethernet.

In terms of memory requirements this solution has two advantages. First, splitting is applied to a much larger set, which makes it a lot easier to split effectively. Second, splitting is applied much closer to the peak, which makes it more efficient in reducing the peak memory requirements of the resulting parts.

The optimized algorithm uses a partitioned transition relation. The full transition relation is a conjunction of all partitions:

$$T(V, V') = T_1(V, V') \wedge T_2(V, V') \wedge \ldots \wedge T_n(V, V'),$$

and an image computation thus becomes

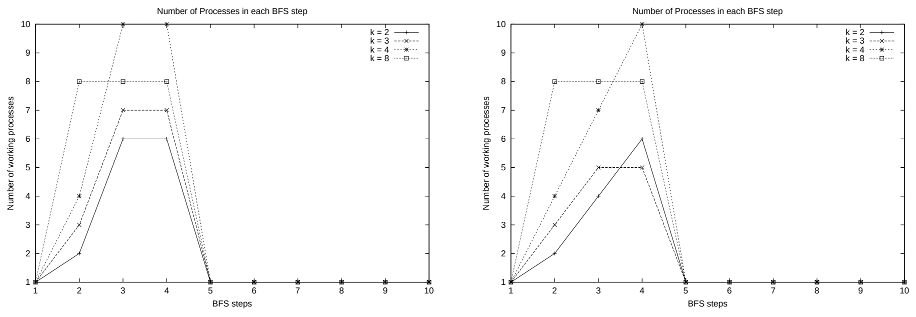
$$S'(V') = \exists V[S(V) \wedge T_1(V, V') \wedge T_2(V, V') \wedge \ldots \wedge T_n(V, V')].$$

The technique for image computation suggested by Burch et al. [16] is to iteratively conjoining the partitions, and to quantify-out variables as soon as further steps do not depend on them. The order in which $T_i(V, V')$ are conjoined is very important to the efficiency of this technique [37]. For the sake of simplicity, let us assume the order is given such that T_1 is the first to conjunct, then T_2, until T_n. Let D_i be the set of variables on which $T_i(V, V')$ depend. Let $E_i = D_i - \bigcup_{m=i+1}^{n} D_m$. A symbolic step is carried out iteratively as follows:

$$\begin{array}{rcl} S_1(V, V') & = & \exists E_1[T_1(V, V') \wedge S(V)] \\ S_2(V, V') & = & \exists E_2[T_2(V, V') \wedge S_1(V, V')] \\ & \vdots & \\ S'(V') & = & \exists E_n[T_n(V, V') \wedge S_{n-1}(V, V')]. \end{array}$$

If overflow occurs during step $0 < j < n$, we look for a set of window functions $w_1 \ldots w_k$ such that $\bigvee_{i=1}^{k} w_i = 1$. The ith worker will get $S_j(V, V') \wedge w_i$. We can now rewrite the $j+1$ step as follows:

$$S_{j+1}(V, V') = \exists E_{j+1}[\bigvee_{i=1}^{k} T_{j+1}(V, V') \wedge S_j(V, V') \wedge w_i].$$



(a) **with no optimization** (b) **optimized**

Figure 5.4: Number of workers required in each BFS step of s1269. Overflow is declared for worker memory utilization exceeding $6M$ BDD nodes.

Since the existential quantification is distributive over disjunction, the above expression is equal to:

$$S_{j+1}(V,V') = \bigvee_{i=1}^{k} \exists E_{j+1}[T_{j+1}(V,V') \wedge S_j(V,V') \wedge w_i].$$

Therefore, the disjunction of the $j+1$th step assigned to each worker is equal to the step done without splitting.

The algorithm uses a new BDD operation: BoundInc$(S(V,V'), \{T_i(V,V')\}, Max)$, where $S(V,V')$ is the function from which the image computation continues, $\{T_i(V,V')\}$ is the set of partitions that were not yet used, and Max is the threshold for overflow during image computation. In the beginning of the algorithm, $S(V,V')$ is the set of states whose image is to be computed in this step, and $\{T_i(V,V')\}$ are all the partitions. If the algorithm overflows, BoundInc returns in $S(V,V')$ the last intermediate result computed prior to the overflow, and in $\{T_i(V,V')\}$ the rest of the partitions that have not been used. If the algorithm completes the image computation, $S(V,V')$ equals the next set of states, and an empty list of partitions is returned.

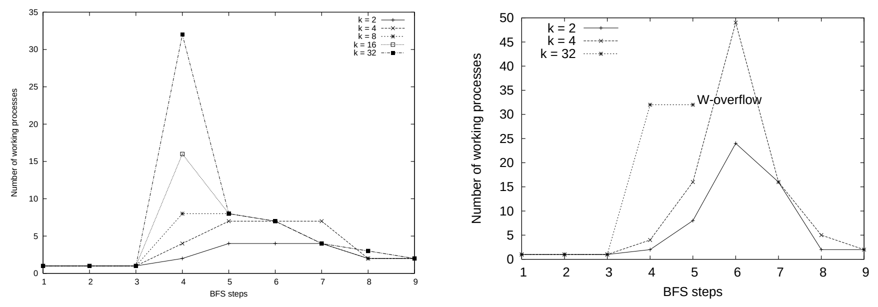
Figure 5.4 illustrates the benefit of using the optimized algorithm for the circuit s1269. Figure 5.4(a) provides the number of workers required in each step for various splitting degrees. For instance, for splitting degree $k=2$, six workers are needed in order to complete Step 3. Figure 5.4(b) shows that this step requires only four workers when using the optimization described in this section. In all other steps and splitting degrees the number of workers required by the optimized algorithm was always less than or equal to the non-optimized version.

5.5 Experimental Results

Our parallel testbed included 25 PC machines, each consisting of dual 1.7GHz Pentium 4 processors with 1GB memory. The communication between the nodes consisted of a Fast

Circuit	#vars	peak		fixed point	
		size	step	time	steps
prolog	117	2.6M	5	2,431	9
s1269	55	16M	5	5,053	10
s3330	172	16M $>$	Ov(3)	-	Ov(3)
s1423	88	16M $>$	Ov(13)	-	Ov(13)

Table 5.1: Benchmark suite characteristics.
The peak is the maximal memory requirement at any point during an image step. Fixed point is the number of image steps and the time (seconds) it takes to get to the fixed point. Ov(m) denotes memory overflow at step m.



(a) **prolog** $Max = $ 1M nodes allocated

(b) **S3330** $Max = $ 7M nodes allocated

Figure 5.5: Number of workers in each BFS step.
Overflow is declared for worker memory utilization exceeding Max BDD nodes.
W-overflow halts the computation when more than 60 workers are required.

Ethernet. We conducted our experiments using four of the largest circuits from the ISCAS89 benchmarks. The characteristics of the circuits are given in Table 5.1.

5.5.1 Number of Workers for Reachability Analysis

Since the memory required by each worker is bounded by a given threshold, we only care about the number of active workers at each iteration. Figures 5.5(a), 5.5(b), 5.6 and 5.4 give the number of workers required at any step of the analysis, and the threshold that was used. The figures prove that using a lower splitting degree is more work efficient, namely, the computation can be carried using fewer resources with a lower splitting degree. This is explained by the fact that when the splitting degree is high, new workers may join in even when the computation can do without them: the computation proceeds with workers that may be under-utilized (but not sufficiently so to be collected by the Collect_Small process).

In steps $1, 2, 3$ in Figure 5.5(a) only one worker is needed. In step 4, this worker needs help in order to complete the image computation. Dividing the work into two is sufficient,

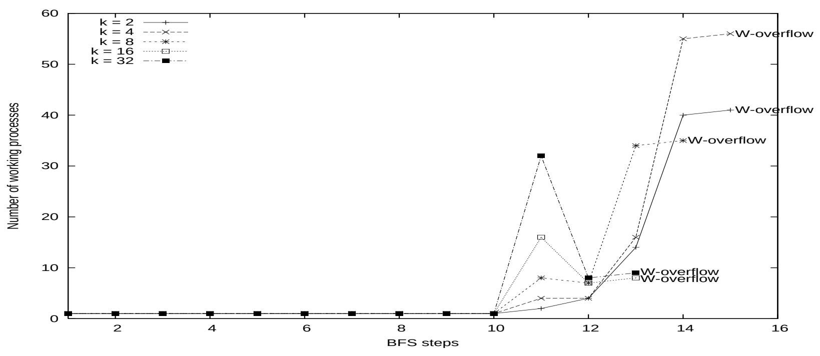


Figure 5.6: Number of workers in each BFS step of s1423.
Overflow is declared for worker memory utilization exceeding $6M$ BDD nodes. **W-overflow** is where more than 60 workers required.

but when the splitting degree is higher we occupy more workers without actually needing them. In steps 8 and 9 the image computation requires less memory and the size of the sets R and N requires less workers. Indeed the number of workers decreases as a result of the Collect_Small procedure.

Figure 5.5(b) shows that the distributed system can complete the reachability analysis, whereas a single machine overflows.

5.5.2 Timing and Communication

We have performed some initial studies regarding the timing and breakdown of running our distributed system. The results show several very clear findings and trends that we now briefly discuss.

First, communication overhead is minor. Our experiments show that the time to reach local overflow is much higher than the time required to dump the contents of memory into the network. Although this finding should be re-evaluated when our system is further optimized (see below), it seems strong enough to sustain. If the system scales up to include more workers, the communication time might grow as a result of more non-owned states that are found. Nevertheless, we expect the computation time to remain dominant because the communication volume for every worker at any split or exchange operation is bounded by the size of the RAM of that worker. We remark that technology trends predict much faster commodity networks (even when compared to the larger expected RAMs) very soon.

Second, splitting is a major element in the computation. It can count up to dozens of percentage points of the computation time, and these numbers grow rapidly when the system scales up. Others have previously addressed the splitting complexity [22]; we intend to speed up the splitting module in our future work.

Third, the fact that the reachability computation is synchronized in a step-by-step fashion has a major impact on the computation time. The problem is that at the end of a step all computing workers wait for the slowest one, who may be slicing and re-slicing several times during the step (remember that slicing is slow!). However, despite its synchronized operation, the new algorithm is very flexible. We believe that it can become the basis for a truly non-synchronized variant.

One interesting phenomena that was not masked by the inefficiencies above is a tradeoff between being work efficient and obtaining speedups. While the best hardware utilization is achieved with splitting degree of 2, the fastest computation times are obtained using somewhat higher splitting degrees (e.g., $k = 8$ for Prolog). Thus, a splitting degree higher that 2 may become instrumental in cases that the speedup is more important than RAM utilization.

Chapter 6

Scalable Distributed On-the-Fly Symbolic Model Checking

6.1 Introduction

A model checking algorithm takes a model and a specification written as a temporal formula. If the model satisfies the formula, the algorithm returns ‘true’;otherwise it returns ‘false’ and provides a counterexample demonstrating why the model does not satisfy the formula. The counterexample feature is vital to the debugging of the system.

A powerful approach to reducing the memory requirements of model checking is *on-the-fly model checking*, in which parts of the model are developed whenever the need arises. The check is usually guided by an automaton that *monitors* the behavior of the system in order to detect errors and stop the evaluation as soon as an error is found. Several on-the-fly algorithms [32, 57, 12] for CTL* use a depth-first search (DFS) traversal of the state space. Since BDD-based methods work efficiently on sets of states, we use an on-the-fly algorithm suggested by Beer et al. [9]. This algorithm uses breadth-first search (BFS) for traversal of the state space. It model checks specifications given as regular expressions describing “bad” (unwanted) behaviors. Note the difference from regular model checking in which the specification formula describes the good behaviors. In this method, a regular expression is translated into an automaton, using the standard algorithm [43]. The acceptance state of the automaton indicates an error state in the model for the given specification. The automaton and the model are then multiplied. Finally, a BFS is used for reachability analysis. The BFS stops as soon as an error state is detected. Industrial temporal languages such as Sugar [7] and ForSpec [4] employ regular expressions. See Section 6.5 for a detailed description of model checking regular expressions on-the-fly.

In this chapter we combine the approaches of [9] and Chapter 3, obtaining a distributed symbolic on-the-fly model checking method that can handle very large designs [10]. Our method includes a distributed algorithm that employs several processes for counterexample generation: the entire set of states is never held in a single process.

Producing the counterexample requires additional storage of sets of states during reachability analysis, one set for each step. In the distributed algorithm each process stores only part of each set. In order to balance the parts of the sets across the processes, we apply a slicing function that defines for each process the parts of the set it should store. The parts a process stores may belong to different parts of the state space. This makes the distributed counterexample generation somewhat tricky: we need to track the steps backwards while switching different slices and maintaining the memory requirement at a low level.

We implemented our method inside the high-performance verification tool RuleBase [8], developed by the IBM Haifa Research Lab. We used a distributed, non-dedicated, slow network system of 32 standard workstations. The performance results show that our method scales well. Large examples that could not fit into the memory of a single machine terminate using the parallel system. The parallel system appears to be balanced with respect to memory utilization. Furthermore, communication over the network does not become a bottleneck.

We were also able to show that the distributed algorithm is more effective for on-the-fly model checking that includes counterexample generation than it is for reachability analysis. There are two main reasons for this. First, the counterexample generation procedure requires that sets of states be saved, and this consumes more space. The parallel system, however, enables the effective splitting and balancing of this additional space. This enhances scalability. Second, the parallel system, even when failing to complete reachability to the fixpoint, is usually able to proceed for several steps beyond the point reached by a single machine. This improves the chances that our on-the-fly model checking will find an error state during these steps.

6.2 The Sequential On-the-Fly Algorithm

In this section we describe the main characteristics of the sequential on-the-fly model checking algorithm, for safety properties, presented in [9]. This algorithm is the basis for our distributed method.

Given a system model M and a regular expression φ describing "bad" behavior, the corresponding automaton $\mathcal{A}$ is constructed and combined with M. $\mathcal{A}$ monitors the behavior of M. If it detects an erroneous behavior, an error flag is set. $\mathcal{A}$ then enters a special state and stays there forever. We call a state that satisfies the error flag an *error state*. Thus, M does not contains any bad behaviors that satisfies φ if and only if the combination of M and $\mathcal{A}$ (that is, $M \times \mathcal{A}$) does not reach an error state. In order to check that M satisfies φ, we run a reachability analysis on $M \times \mathcal{A}$ that constantly checks whether an error state has been encountered. The algorithm traverses the (combined) model using a breadth–first search (BFS). Starting from the set of initial states, it constructs a *doughnut* at each iteration. This doughnut is the set of new states found in that iteration. The doughnuts are kept for later use in the generation of the counterexample. Keeping the doughnuts increases the space requirements of this algorithm, and they exceed those of (pure) reachability analysis.

The model checking algorithm terminates successfully if all reachable states have been traversed and no error state has been found. If at any stage an error state is encountered, the model checking algorithm stops and the generation of a counterexample begins.

A counterexample is a sequence of states that starts with an initial state and ends with an error state. It is generated backwards. The algorithm begins with an error state and selects a state from among its predecessors. Then the generation continues, following the doughnuts that were produced and stored by the reachability analysis algorithm. All these selected states are saved in the order in which they were found. Counterexample generation terminates when the doughnut of the initial states is reached. At this point the selected states comprise a complete counterexample sequence.

Figure 6.1 presents the sequential algorithm for on-the-fly model checking, including the counterexample generation procedure. The algorithm differs from simple BFS in three ways: it evaluates the formula while computing the set of reachable states; it saves the sets of states for the counterexample generation; if it reaches an error state, it constructs a counterexample. The counterexample generation procedure is based on the one in [24]. Lines 1–9 describe the model checking phase. At each iteration i, the set of `new` states that have not yet been reached is kept in doughnut S_i.

The algorithm terminates if either no new states are found ($\texttt{new} = \emptyset$), in which case it announces success, or if an `error` state is found ($\texttt{new} \cap \texttt{error} \neq \emptyset$), in which case it announces failure.

In lines 16–22, the counterexample $\texttt{Ce}_0, \dots \texttt{Ce}_k$ is generated. The counterexample is of length $k + 1$ (line 14), since an error state was first found in the k-th iteration. We choose $\texttt{Ce}_k \in S_k$ from among the error states reached. Having already chosen a state $\texttt{Ce}_i \in S_i$, we compute the set of `bad` states by finding the set of predecessors for $\texttt{Ce}_i$: $\texttt{pred}(\texttt{Ce}_i)$. We then intersect it with the doughnut S_{i-1} (line 19). Since each state in S_i is a successor of some state in S_{i-1}, the set `bad` will not be empty. We now choose $\texttt{Ce}_{i-1}$ from the set of `bad` states. The generation of the counterexample continues until $\texttt{Ce}_0$ is chosen.

6.3 Distributed Algorithm

The distributed algorithm for on-the-fly model checking consists of two phases:

- The model checking phase
- The counterexample generation phase

6.3.1 Distributed Model Checking

In the distributed algorithm, an initial sequential stage precedes the distributed stage. The reachable states are first computed on a single process. When a certain memory requirement threshold is reached, the state space is partitioned into k slices, whose union is the whole state space. This partition, or slicing, should require less memory. Furthermore, the subsets should be disjoint. Disjoint subsets will allow us to avoid duplication of work during reachability analysis. The slicing algorithm [41, 55, 22] selects a variable and uses it to slice a set into two disjoint subsets. Using the slicing algorithm k times results in k subsets that are distributed to k processes. This ends the sequential stage.

```
reachable = new = initialStates
i = 0
while ((new $\neq \emptyset$)&&(new $\cap$ error = $\emptyset$)) {
  $S_i$ = new
  i = i+1
  next = nextStateImage(new)
  new = next $\setminus$ reachable
  reachable = reachable $\cup$ next
}
if (new $= \emptyset$) {
  print ``formula is true in the model''
  return
}
k = i
print ``formula is false in the model''
bad = new $\cap$ error
while (i>=0) {
  Ce$_i$ = choose one state from bad
  if (i>0) bad=pred(Ce$_i$)$\cap S_{i-1}$
  i = i-1
}
print ``counterexample is:''  Ce$_0 \cdots$Ce$_k$
```

Figure 6.1: Sequential algorithm for on-the-fly model checking, including counterexample generation

The distributed stage begins with each process being informed of the slice it *owns*, and of the slices owned by each of the other processes (which are *non-owned* by this process). The process receives its own slice and proceeds to compute the reachable states for that slice in iterative BFS steps. At each such step, the set of new states is kept in a doughnut.

Each process computes the set `next` of states that are reached directly from the states in its `new` set. The `next` set contains owned as well as non-owned states. Each process splits its `next` set according to the k slices and sends the non-owned states to their corresponding owners. At the same time, the process receives the set of states it owns from the other processes.

The model checking phase for one process P_j is given in lines 1–13 of Figure 6.2. Lines 1–3 describe the setup stage where the process receives the slice it owns and the initial sets of states it needs to compute from. Lines 5–17 describe the iterative computation.

Distributed termination detection (line 5) is used to determine when this phase should end. *All* processes should end at this phase if one of two conditions holds: none of the processes found a new state or one of them found an error state. In the first case, the specification has been proven correct and the algorithm terminates. In the second case the specification is false, and all processes proceed to the counterexample generation phase. In order to distinguish between the two cases, the termination detection procedure is used (line 14) with the error parameter equal 0.

Several points distinguish distributed model checking from sequential model checking. When distributed model checking is used,

- the set `next` is modified (lines 9–10) through communication with the other processes and is restricted to include only owned states;
- distributed termination detection is applied;
- for each doughnut i, each process P_j stores the slice of the doughnut $S_{(i,j)}$ it owns.

Our distributed algorithm is made particularly effective by the *memory balancing* procedure, which maintains approximately equal memory requirements across the processes during the entire computation. This is accomplished by pairing large slices with small ones and reslicing their union in a balanced way. As a result, a process owns (and stores) different slices of the doughnuts in different iterations. Therefore, in some iteration, a process may own a state that does not have any predecessors stored in the slices of the doughnuts it owned previously. The distributed generation of a (correct) counterexample is nonetheless guaranteed by the following property, which is true by construction:

$$S_i = \bigcup_j S_{(i,j)}, \tag{6.1}$$

where S_i is the doughnut computed by the sequential algorithm at iteration i.

6.3.2 Distributed Counterexample Generation

To generate a counterexample, our algorithm uses the doughnut slices that are stored in the memory of the processes. The distributed counterexample generation algorithm consists of *local phases* and *coordination phases*. In the local phase, all processes run in parallel. Each process takes the counterexample generated so far, denoted by the suffix $\texttt{Ce}_i \ldots \texttt{Ce}_k$. It then executes the sequential algorithm for counterexample generation, adding the additional states $\texttt{Ce}_{i-1}$,$\texttt{Ce}_{i-2}$,... until it can proceed no further. A process may get stuck after producing a counterexample with suffix $\texttt{Ce}_i \ldots \texttt{Ce}_k$ if it cannot find a predecessor for $\texttt{Ce}_i$ in its own slice of the (i-1)th doughnut. However, by property (6.1) and by the fact that each element in S_i has a predecessor in S_{i-1}, there must be a process that has such a predecessor for $\texttt{Ce}_i$.

In the coordination phase, the process that produces the largest suffix is selected and used to reinitiate the local phase in all processes. If this suffix is complete (i.e., it contains all states $\texttt{Ce}_0 \ldots \texttt{Ce}_k$), the process simply prints its counterexample and all processes terminate. Otherwise, the process broadcasts its suffix, together with its iteration number, to all other processes. Each process updates its data accordingly and reinitiates the local phase from that point. The algorithm continues until a complete suffix is found.

Lines 18–35 of Figure 6.2 describe the algorithm. Lines 22–26 contain the local phase, while lines 27–35 contain the coordination phase. The algorithm uses the following three variables:

- `myId`, which is the index of the process (`myId`=j for process P_j);
- `minIte`, the smallest iteration number, chosen at the start of the coordination phase;
- `minProc`, the smallest index among the processes with the smallest iteration number.

6.3.3 Reducing Peak Memory Requirement

In order to generate the counterexample, the sets `bad` = `pred`($\texttt{Ce}_i$)$\cap S_{(i,j)}$ must be computed. This is done by intersecting the doughnut slice $S_{(i,j)}$ with the set of predecessors of the state $\texttt{Ce}_i$ (lines 24, 35). The BDDs for $\texttt{Ce}_i$ and `bad` are usually small. However, a very large peak in memory use may be caused by intermediate BDDs obtained during the computation of `bad`. This phenomenon can be viewed in example GXI (Figure 6.7), where a significant increase in memory use causes the parallel system to overflow during the computation of the counterexample.

Changing the order of operations can, however, produce smaller intermediate BDDs. This, in turn, reduces the peak memory requirement. In the new order, we first restrict the transition relation of our model to the doughnut slice $S_{(i,j)}$ and only then use it to compute `pred`($\texttt{Ce}_i$). Since our implementation is based on the partitioned transition relation [16], we actually restrict each one of the partitions to the doughnut slice.

To increase precision, we define the operations we perform by means of Boolean functions (represented as BDDs). Assume that our model consists of a set of Boolean variables V. The Boolean function $TR(V, V')$ represents the transition relation of the model, where V and V' represent the current and next state, respectively.

```
mySlice = receive(fromSingle)
reachable = receive(fromSingle)
new = receive(fromSingle)
i = 0
while (Termination(new,error)==0) {
  S$_{(i,j)}$ = new
  i = i+1
  next = nextStateImage(new)
  next = sendReceiveAll(next)
  next = next $\cap$ mySlice
  new = next $\setminus$ reachable
  reachable = reachable $\cup$ next
}
if (Termination(new,0)==1) {
  print ``formula is true in the model''
  return
}
k = i
print ``formula is false in the model''
bad = new $\cap$ error
while (i>=0) {
  while ((i>=0) &&(bad $\neq \emptyset$)) {
    Ce$_i$ = choose one state from bad
    if (i>0) bad=pred(Ce$_i$)$\cap$ S$_{(i-1,j)}$
    i = i-1
  }
  (minIte,minProc)=MinIteFromAll(i,myId)
  i = minIte
  if (i<0) {
    if (myId == minProc)
      print ``counterexample is:''  Ce$_0 \cdots$Ce$_k$
    return
  }
  Ce$_{i+1} \cdots$Ce$_k$=broadcast(minProc,Ce$_{i+1} \cdots$Ce$_k$)
  bad=pred(Ce$_{i+1}$)$\cap$ S$_{(i,j)}$
}
```

Figure 6.2: Process P_j in the distributed algorithm for on-the-fly model checking, including the generation of a counterexample.

Let $\texttt{Ce}_i(V)$ be the Boolean function for the singleton set consisting of the state $\texttt{Ce}_i$, and let $S_{(i,j)}(V)$ be the Boolean function for the slice $S_{(i,j)}$. Then the computation of `bad` at the j'th process can be described by the expression

$$\exists V' \, [\, TR(V, V') \wedge Ce_i(V') \,] \;\wedge\; S_{(i,j)}(V). \tag{6.2}$$

Our transition relation consists of partitions $PTR_n(V, V')$ such that $TR(V, V') = \bigwedge_n PTR_n(V, V')$. Consequently, the previous expression can be rewritten as

$$\exists V' \, [\, \bigwedge_n PTR_n(V, V') \wedge Ce_i(V') \,] \;\wedge\; S_{(i,j)}(V). \tag{6.3}$$

Since $S_{(i,j)}(V)$ does not depend on V', it can be moved into the scope of the quantifier, resulting in an equivalent expression:

$$\exists V' [\bigwedge_n \left(PTR_n(V, V') \wedge S_{(i,j)}(V) \right) \wedge Ce_i(V')]. \tag{6.4}$$

This expression describes the computation at the j'th process. First, each partition of the transition relation is restricted to the doughnut slice $S_{(i,j)}$, and then the predecessors of Ce_i are computed.

This computation can be made more efficient by using the *simplify-assuming* technique [31]. Let $f : E \longrightarrow \{0, 1\}$ be a Boolean function over some domain E. If we are concerned only with the value of f over some subset D of E, then we may reduce the BDD size for f. This can be done by finding another function f' which agrees with f on D and can have any value on elements not in D.

Formally, given a function $f : E \longrightarrow \{0, 1\}$ and an assumption $D \subseteq E$, we say that a function $f' : E \longrightarrow \{0, 1\}$ *simplifies f assuming D* if it satisfies

$$f' \wedge D = f \wedge D. \tag{6.5}$$

We denote such an f' by $f|_D$.

The algorithm given by [31] guarantees that the BDD size of f' is equal to or smaller than the BDD size of f. We use this technique to reduce the size of each partition in the transition relation. The reduced partition sizes decrease the memory requirement during the computation of the expression in (6.4). Instead of intersecting each $PTR_n(V, V')$ with $S_{(i,j)}(V)$, we simplify $PTR_n(V, V')$ assuming $S_{(i,j)}(V)$ and intersect the result with $S_{(i,j)}(V)$. Since simplify-assuming satisfies (6.5), the expression in (6.4) is equivalent to

$$\exists V' [\bigwedge_n \left(PTR_n(V, V')|_{S_{(i,j)}(V)} \wedge S_{(i,j)}(V) \right) \wedge Ce_i(V')]. \tag{6.6}$$

Since $S_{(i,j)}(V)$ does not depend on V', it can be moved outside of the scope of the quantifier, resulting in an equivalent expression:

$$\exists V' [\bigwedge_n \left(PTR_n(V, V')|_{S_{(i,j)}(V)} \right) \wedge Ce_i(V')] \wedge S_{(i,j)}(V). \tag{6.7}$$

The improvement described above uses precise information in order to restrict the partitions of the transition relation. This requires computing a different restriction for each doughnut in each step of the counterexample generation. We suggest a different method of restriction, which is computed only once for each process. Process P_j simplifies $PTR_n(V, V')$ assuming U_j, where $U_j = \cup_i S_{(i,j)}$ is the union of all the doughnut slices owned by P_j. Since $S_{(i,j)} \subseteq U_j$, the expression in (6.4) is equivalent to

$$\exists V' [\bigwedge_n \left(PTR_n(V, V')|_{U_j(V)} \right) \wedge Ce_i(V')] \wedge S_{(i,j)}(V). \tag{6.8}$$

Note that $PTR_n(V, V')|_{U_j(V)}$ is computed only once at the beginning of the counterexample generation process.

We next suggest an orthogonal improvement that exploits the fact that we compute the set of predecessors of a singleton ($Ce_i(V')$, which contains only one state, Ce_i). We replace the intersection of $PTR_n(V, V')$ and $Ce_i(V')$ by substituting the state Ce_i for V' in $PTR_n(V, V')$. The existential quantifier is then redundant and can be removed. Modified thus, equation (6.3) can first be rewritten as

$$\exists V' \, [\bigwedge_n \, (\, PTR_n(V, Ce_i) \,) \,] \, \wedge S_{(i,j)}(V). \tag{6.9}$$

The existential quantifier is then redundant and can be removed to obtain

$$\bigwedge_n \, (\, PTR_n(V, Ce_i) \,) \, \wedge S_{(i,j)}(V). \tag{6.10}$$

Combining the above optimization with simplify-assuming, we can compute (6.3) as

$$S_{(i,j)}(V) \wedge \bigwedge_n \left(PTR_n(V, Ce_i)|_{U_j(V)} \right). \tag{6.11}$$

Again, $PTR_n(V, V')|_{U_j(V)}$ is computed only once. At step i of the counterexample generation procedure, P_j assigns Ce_i to each of $PTR_n(V, V')|_{U_j(V)}$ and then intersects $S_{(i,j)}(V)$ with them.

Experimental results show that all of the suggested optimizations significantly reduce memory requirements. Compare, for instance, the results in Figure 6.7, where example GXI is run without any optimization, to the results in Figure 6.8 where it runs with the optimization in expression 6.10. On the other hand, we found the optimization in expression 6.11 to have no significant advantage over the one in expression 6.10.

6.4 Experimental Results

In this section we report on the performance evaluation of our approach. We implemented our method inside the high-performance verification tool RuleBase [8], which is based on McMillan's SMV [52] and was developed by the IBM Haifa Research lab. Our parallel testbed includes 32 RS6000 machines, each consisting of a 225 MHz PowerPC processor and

512 MB memory. The communication between the nodes consists of a 100 Megabit/second token ring.

We selected large circuits to show that the parallel system can find errors that the sequential algorithm cannot find. This is because the sequential algorithm uses more memory than is available in a single machine. We experimented with two of the largest circuits we found in the benchmarks: ISCAS89 +addendum'93. In order to test the counterexample generation, we used common properties that are often tested when verifying hardware designs. We also used two large examples, BIQ and GXI, which are components of IBM's Gigahertz processor. We used the original properties for these examples. These properties are explained in Table 6.1. We mapped properties to automata using the IBM implementation as described in [9]. Characteristics of the circuits and the automata are given in Table 6.2.

6.4.1 Space Reduction Using the Distributed Algorithm

This section presents the results for on-the-fly model checking of the benchmark suite using our 32 machine test-bed. Figures 6.3 to 6.9 summarize memory utilization, giving the peak memory consumption for every step. Each of the graphs compares the memory utilization in the single-machine execution to that of the parallel system. For the parallel system we give the highest (peak) memory utilization in any of the machines.

We give examples for four models and six properties. Two properties are checked for the BIQ and S1423 models: one that overflows on a single machine, and another that completes the computation even when only a single machine is used.

As Figure 6.3 shows, an overflow occurs at cycle 15 while the algorithm searches for an error state in BIQ using a single machine. The overflow occurs because the counterexample generation (CE) phase requires that the doughnuts be saved, and this consumes a lot of memory. In contrast, the parallel algorithm does not overflow. It finds the error state in BIQ at cycle 17.

At the cycle where the error state is found, we see a drop in memory utilization. This drop is due to the fact that the CE phase will begin in the next step, making state exchange and load balancing unnecessary. State exchange and load balancing may contribute significantly to the observed peak in memory requirement. This effect is particularly strong in BIQ spec1, s1423 spec1, and s5378 spec1 (Figure 6.5, Figure 6.3 and Figure 6.9).

Figure 6.4 shows another drop in memory utilization, at the first counterexample cycle. This drop, which is characteristic of many examples, is caused by two factors. First, the transition relation computations during a backward step are usually simpler than those performed during a forward step, and they require less memory. This is due to the relative simplicity of the relation consisting of a single origin (the last state in the CE found so far). Second, the set of reachable states can be released since it is not needed for counterexample generation.

The parallel algorithm finds an error state in cycle 14 of S1423, as depicted in Figure 6.5. In this case, finding the error state on-the-fly is essential because, even with our parallel system, we were not able to complete reachability analysis on this example.

Figure 6.7 demonstrates why our optimizations are necessary. In this example, an overflow occurs during a step backwards from a single state using the original transition relation. The overflow occurs because we are using the partitioned transition relation, in which backward steps are much harder to perform than forward. We can avoid this problem by using substitution of the singleton (as describe in Section 6.3.3) instead of quantification over the partitioned transition relation. Figure 6.8 demonstrates the effect of this method on our example.

6.4.2 Timing and Communication in the Distributed Algorithm

Table 6.3 gives the timing breakdown for on-the-fly model checking on our benchmark suite. The parallel reachability stage takes most of the computation time. As shown in [41], communication does not become a bottleneck at this stage.

6.5 Regular Expressions in Symbolic Model Checking

When specifying a formula in temporal logic, one describes what should *hold* in the model. Another way to specify a property is to describe what should *never hold* in the model, that is, to describe the set of *bad* computations rather than the good ones. A nice way to describe a set of finite bad computations is by means of *regular expressions* (RE), as follows: Let W be a finite set of symbols (in our case, signal names in the model under test). The alphabet Σ, over which the regular expressions are defined, is the set of all Boolean expressions over W.

As an example, consider a model with two signals: req and ack, and consider a property specifying that every req must be followed by an ack in the next cycle. Σ in this case consists of all 16 possible Boolean functions ($true$, $false$, req, $\neg req$, $req \vee ack$, etc.). A description of the bad computations of this property would state that sequences with req holding in one state and ack *not* holding in the next state are illegal.

Using regular expressions, we get the following:

$$(true*)(req)(\neg ack)$$

In order to check a given model M against a RE specification r, one has to build the corresponding automaton A_r [43] and check that any word in $L(A_r)$ is not a prefix of a computation path in M.

Using RuleBase, we perform this check in the following way: First, we translate A_r into a corresponding non-deterministic finite state machine F_r in the input language of SMV, with final states $q_1, ..., q_n$. We then model-check the CTL formula

$$AG(\neg q_1 \wedge, ..., \wedge \neg q_n)$$

against the model $M \times F_r$.

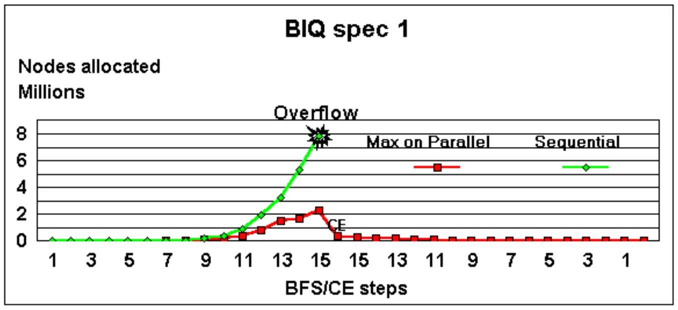


Figure 6.3: Memory utilization during on-the-fly model checking of BIQ (spec 1)

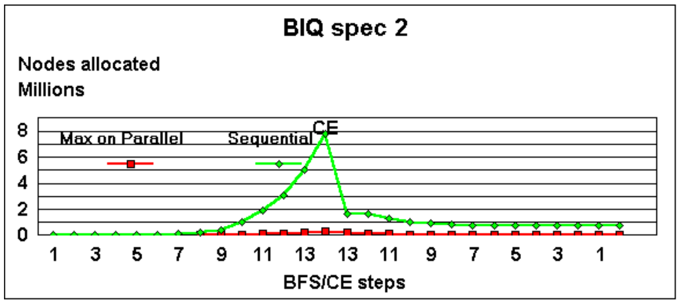


Figure 6.4: Memory utilization during on-the-fly model checking of BIQ (spec 2)

Note that the formula to be checked is of the form $AG(p)$, where p is a Boolean formula. It can thus be checked on-the-fly [50, 9], saving a lot of time and space. Model checking of regular expressions is more efficient in most cases than model checking of CTL formulas [9].

The expressive power of the regular expressions we have described above differs from that of temporal logics. In [9], Beer et al. present an algorithm for translating a subset of CTL formulas to RE specifications. The subset of CTL which can be translated to RE is called $RCTL$.

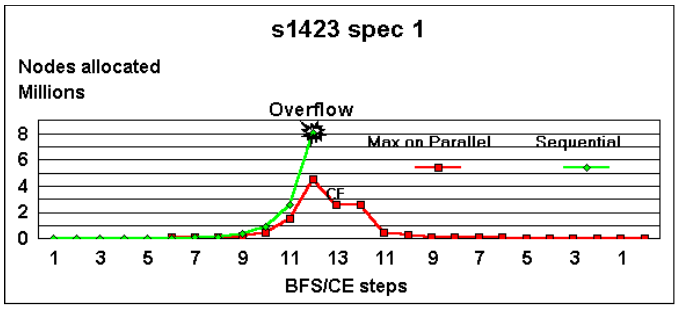


Figure 6.5: Memory utilization during on-the-fly model checking of s1423 (spec 1)

BIQ spec 1
If the $writePtr$ points to P and the value on the bus is D, then four cycles after the next time a $read$ from P occurs, the value going out should be D. Sugar: $\{[*], (writePtr(0..3) = P(0..3))\&(dataIn(0) = D(0)),$ $goto(readPtr(0..3) = P(0..3))\}(AX[4](dataOut(0) = D(0)))$
BIQ spec 2
If the $writePtr$ points to P and the value on the bus is D, then two cycles after the next time a $read$ from P occurs, the value going out should be D. Sugar: $\{[*], (writePtr(0..3) = P(0..3))\&(dataIn(0) = D(0)),$ $goto(readPtr(0..3) = P(0..3))\}(AX[2](dataOut(0) = D(0)))$
s1423 spec 1
If $G729$ and $G726$ are true, then $G726$ is true ten cycles later. Sugar: $AG(G729\&G726 \rightarrow AX[10](G726))$
s1423 spec 2
If $G729$ and $G726$ are true, then $G726$ is true seven cycles later. Sugar: $AG(G729\&G726 \rightarrow AX[7](G726))$
GXI spec 1
If $start$ is true and the address is A, then if two cycles later a rejection occurs, then between 2 to 32 cycle later, start should hold again, with address equals A. Sugar: $\{[*], START\&ADDR(0..2) = A, true, reject\}$ $(ABF[2..32](START\&ADDR(0..2) = A))$
s5378 spec 1
If $n3104gat$ is true, then starting six cycle later, $n3106gat$ should hold before $n3104gat$ holds. Sugar: $AG(n3104gat \rightarrow AX[6](n3106gat$ before $n3104gat))$

Table 6.1: The specifications in Sugar with explanations.

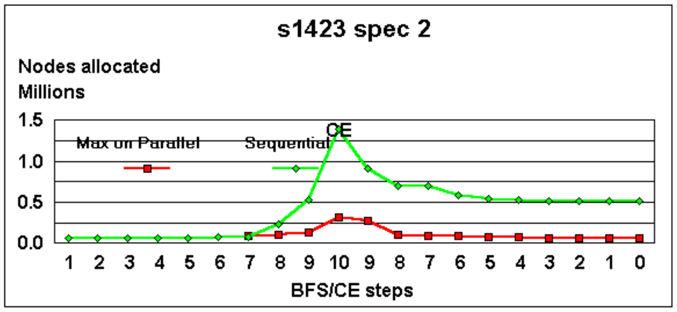


Figure 6.6: Memory utilization during on-the-fly model checking of s1423 (spec 2)

Circuit	#vars + sat	peak		spec check		
		size	step	time	steps	CE
BIQ						
spec 1	102 + 5	5.85M	15	15,059	Ov(15)	
spec 2	102 + 5	5.33M	14	3,811	15	95
s1423						
spec 1	91 + 4	8.64M	12	2,024	Ov(12)	
spec 2	91 + 3	1.54M	10	625	11	58
GXI						
spec 1	292 + 6	8.14M	44	16,222	Ov(44)	
s5378						
spec 1	188 + 4	9.66M	6	4,440	Ov(6)	

Table 6.2: Characteristics of our benchmark suite
#vars gives the number of variables in the model and in the sat(ellite). Sizes are given in million BDD nodes, and all times in seconds. The peak is the maximal memory requirement at any point during a step. In order to mask the effect of garbage collection scheduling decisions, the peak is measured after gc invocations. Spec check is the number of steps it takes to find an error state, and the time it takes to generate a counterexample (CE). Ov(x) designates memory overflow during step x. All measurements were taken using an RS6000 machine consisting of a 225 MHz PowerPC processor with 512 MB memory.

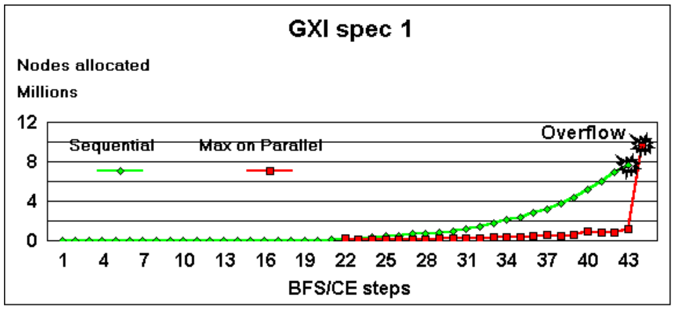


Figure 6.7: Memory utilization during on-the-fly model checking of GXI (spec 1), using quantification. An overflow occurs during counterexample generation.

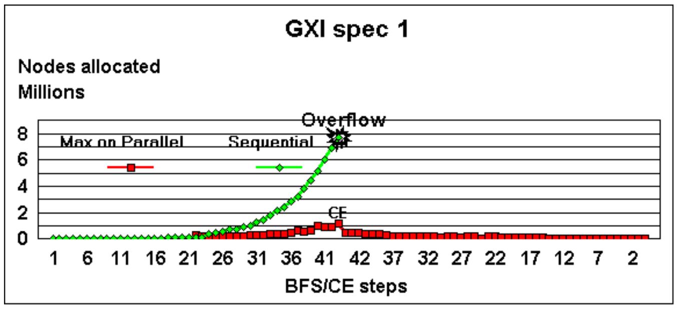


Figure 6.8: Memory utilization during on-the-fly model checking of GXI (spec 1), using substitution

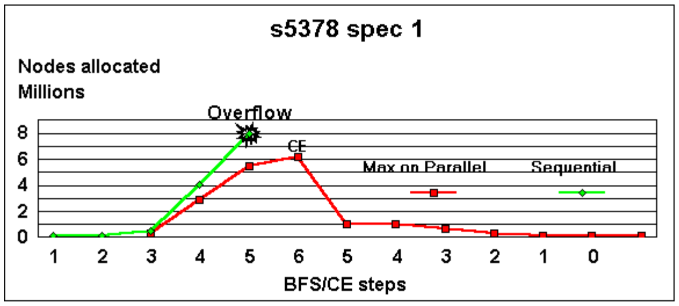


Figure 6.9: Memory utilization during on-the-fly model checking of s5378 (spec 1)

Circuit spec	steps	total	Reachability seq	Reachability par	Spec check eval	Spec check CE
BIQ 1	17(15)	1,957	174	1,804	31	74
BIQ 2	15	921	184	731	24	52
s1423 1	14(12)	16,032	13	15,911	35	117
s1423 2	11	521	116	337	10	102
GXI 1	45(44)	8,468	1,866	6,570	26	138
s5378 1	7(6)	12,873	384	11,509	69	105

Table 6.3: Timing data (seconds) for parallel execution on 32$\times$512MB machines. Each of the measures is the worst sample over all the machines. The steps count shows that the parallel system always reaches a point beyond that at which a single machine overflows (this point is given in brackets). Total is the total time over all steps, including the sequential stage, parallel reachability stage and counterexample generation time. Note that the total time is the maxima over sums and not the sum over maxima. Seq(uential) is the time it took to reach the threshold at which the parallel stage started. Par(allel) is the parallel reachability analysis time. Eval(uation) is the total time it took to evaluate, at each step, whether one of the processes found an error state. (Note that in the sequential stage, Eval is a single BDD operation, while in the parallel stage it also requires global interaction over the network). CE is the time it took to generate the counterexample.

Chapter 7

Distributed Symbolic μ-calculus

7.1 Introduction

This chapter extends the scope of properties that can be verified for large designs. It presents a distributed symbolic model checking algorithm for the μ-calculus, which is a powerful formalism for expressing properties of transition systems using least and greatest fixpoint operators. Many verification procedures can be solved by translating them into μ–calculus model checking[17] problems. Such verification procedures include (fair) CTL model checking, LTL model checking, bisimulation equivalence, and language containment of ω-regular automata.

Many algorithms for μ-calculus model checking have been suggested [34, 65, 69, 29, 49]. In this work we parallelize a simple sequential algorithm [28]. The algorithm works bottom-up through the formula, evaluating each subformula based on the value of its own subformulas. A formula is interpreted as the set of states in which it is true. Thus, for each μ–calculus operation, the algorithm receives a set (or sets) of states and returns a new set of states.

The distributed algorithm follows the same lines as the sequential one, except that each process runs its own copy of the algorithm and each set of states is stored distributively among the processes [39]. Every process *owns* a slice of the set, so that the disjunction of all slices contains the whole set. An operation is now performed on a set (or sets) of slices and returns a set of slices. At no point in the distributed algorithm is a whole set is stored by a single process.

The intuitive solution for a distributed computation might prove to be deceptive for some operations. For instance, in order to evaluate a formula of the form $\neg g$, the set of states satisfying g should be complemented. It is impossible for a single process to carry out this operation locally. Rather, each process sends the other processes the states they own, which are not in g “to the best of its knowledge.” If none of the processes “knows” that a state is in g, then the state is (distributively) determined to be in $\neg g$.

While performing an operation, a process may obtain states that are not owned by it. For instance, when evaluating the formula $\mathbf{EX}f$, a process will find the set of all predecessors of states in its slice for f. However, some of these predecessors may belong to the slice of another process. Therefore, the procedure `exch` is executed (in parallel) by all processes, and each process sends its non-owned states to their respective owners.

Memory requirements are kept low through frequent calls to a memory balancing procedure. It ensures that each set is partitioned evenly among the processes. This ensures that the memory requirements, which are usually proportional to the size of the manipulated set, are evenly distributed among the processes. However, this also requires different slicing functions for different sets. As a result, we may need to apply an operation to two sets that are sliced according to different partitions. In the case of *conjunction*, for instance, the two sets should first be re-sliced according to the same partition. Only then do the processes apply conjunction to their individual slices. Narayan et al. [56] show how to perform negation, conjunction, and disjunction under the assumption that the set of window functions does not change. However, if the set does not change, the memory requirement will be unbalanced as explained. This will render the distributed system ineffective.

Distributing the sets of states is only one facet of the problem. The transition relation also strongly influences the memory peaks that appear during the computation of pre-image (**EX**) operations. The pre-image operation has one of the highest memory requirements in model checking. Even when its final result is of tractable size, its intermediate results might cause memory overflow. We propose a scalable distributed method for the pre-image computation, including slicing of the transition relation.

7.2 Preliminaries

7.2.1 The Propositional μ–Calculus

Below we define the propositional μ–calculus [47]. We will not distinguish between a set of states and the Boolean function that characterizes this set. By abuse of notation we will apply both set operations and Boolean operations on sets and Boolean functions. Let AP be a set of atomic propositions and let $VAR = \{Q, Q_1, Q_2, \ldots\}$ be a set of relational variables. The μ–calculus formulas are defined as follows:

- if $p \in AP$, then p is a formula;
- a relational variable $Q \in VAR$ is a formula;
- if f and g are formulas, then $\neg f$,$f \wedge g$,$f \vee g$, $\mathbf{EX}\ f$ are formulas;
- if $Q \in VAR$ and f is a formula, then $\mu Q.f$ and $\nu Q.f$ are formulas.

μ–calculus consists of the set of *closed* formulas in which every relational variable Q is within the scope of μQ or νQ.

Formulas of the μ–calculus are interpreted with respect to a *transition system* $M = (St, R, L)$, where St is a nonempty and finite set of states, $R \subseteq St \times St$ is the transition relation, and $L : St \to 2^{AP}$ is the labelling function that maps each state to the set of atomic propositions true in that state.

In order to define the semantics of μ–calculus formulas, we use an *environment* $e : VAR \rightarrow 2^{St}$, which associates with each relational variable a set of states from M.

Given a transition system M and an environment e, the semantics of a formula f, denoted $[[f]]_M e$, is the set of states in which f is true. We denote by $e[Q \leftarrow W]$ a new environment that is the same as e except that $e[Q \leftarrow W](Q) = W$. The set $[[f]]_M e$ is defined recursively as follows (where M is omitted when clear from the context).

- $[[p]]e = \{s \mid p \in L(s)\}$
- $[[g_1 \wedge g_2]]e = [[g_1]]e \cap [[g_2]]e$
- $[[Q]]e = e(Q)$
- $[[g_1 \vee g_2]]e = [[g_1]]e \cup [[g_2]]e$
- $[[\neg g]]e = St \setminus [[g]]e$
- $[[\mathbf{EX}g]]e = \{s \mid \exists t\ [(s,t) \in R$ and $t \in [[g]]e]\ \}$
- $[[\mu Q.g]]e$, $[[\nu Q.g]]e$ are the least and greatest fixpoints, respectively, of the predicate transformer $\tau : 2^{St} \rightarrow 2^{St}$ defined by: $\tau(W) = [[g]]e[Q \leftarrow W]$

Tarski [66] showed that least and greatest fixpoints always exist if τ is monotonic. If τ is also continuous, then the least and greatest fixpoints of τ can be computed by $\cup_{i \in N} \tau^i(False)$ and $\cap_{i \in N} \tau^i(True)$, respectively. In [28] it is shown that if M is finite then any monotonic τ is also continuous.

In this chapter we consider only monotonic formulas. Since the only transition systems we consider are finite, they are also continuous. The function `fixpt` in Figure 7.2 describes an algorithm for computing the least or greatest fixpoint, depending on the initialization of Q_{val}. If the parameter $init$ is $False$, the least fixpoint is computed. Otherwise, if $init = True$, the greatest fixpoint is computed.

Given a transition system M, an environment e, and a formula f of the μ–calculus, the *model checking* algorithm for μ–calculus finds the set of states in M that satisfy f. Figure 7.1 presents a sequential recursive algorithm for evaluating μ–calculus formulas. For closed μ–calculus formulas, the initial environment is irrelevant. The necessary environments are constructed during recursive applications of the `eval` function.

```
function eval($f,e$)
 1  case
 2    $f$= $p$:        $res$= $\{s \mid p \in L(s)\}$
 3    $f$= $Q$:        $res$= $e(Q)$
 4    $f$= $\neg g$:       $res$= $\neg$eval($g,e$)
 5    $f$= $g_1 \vee g_2$:  $res$= eval($g_1,e$)$\vee$eval($g_2,e$)
 6    $f$= $g_1 \wedge g_2$:  $res$= eval($g_1,e$)$\wedge$eval($g_2,e$)
 7    $f$= $EXg$:      $res$= $\{s \mid sRt \wedge t \in$eval($g,e$)$\}$
 8    $f$= $\mu Q.g$:    $res$= fixpt($Q,g,e,False$)
 9    $f$= $\nu Q.g$:    $res$= fixpt($Q,g,e,True$)
10  endcase
11  return($res$)
end function
```

Figure 7.1: Pseudo–code for sequential μ–calculus model checking

```
function fixpt($Q, g, e, init$)
1 $Q_{val}$= $init$
2 repeat
3   $Q_{old}$= $Q_{val}$
4   $Q_{val}$= eval($g, e[Q \leftarrow Q_{val}]$)
5 until ($Q_{val} = Q_{old}$ )
6 return $Q_{val}$
end function
```

Figure 7.2: Pseudo–code for computing fixpoint

7.2.2 Elements of Distributed Symbolic Model Checking

Our distributed algorithm includes several basic elements that were developed in Chapter **??**. For completeness, we give a brief overview of these elements in this subsection.

Sets of states in the transition system, as well as the intermediate results, are represented by BDDs. At any point during the algorithm's execution, the sets of states obtained are partitioned among the processes. A set of *window functions* is used to define the partitioning, determining the slice that is *owned* by each process.

Definition 5: [Complete set of window functions [56, 22]] A window function is a Boolean function that characterizes a subset of the state space. A set of window functions $W_1, \ldots, W_k$ is complete if and only if $\bigvee_{i=1}^{k} W_i = 1$.

Unless otherwise stated, we assume that all sets of window functions are complete.

We use the *slicing algorithm*, as described in Chapter **??**, to get a set of window functions.

Maintaining an equal load while the intermediate results are being stored is essential for the scalability of the parallel algorithm. The equal load is maintained throughout the algorithm by means of the *memory balance* procedure in Chapter **??**.

More formally, the `ldBlnc` procedure is a parallel algorithm, as follows. Let $W_1, \ldots, W_k$ be a set of window functions, and res be a set of states, so that process i owns the subset $res_i = res \wedge W_i$. When `ldBlnc` terminates, a new set of window functions $W'_1, \ldots, W'_k$ is produced, and process i owns $res'_i = res \wedge W'_i$.

During the memory balance procedure, as well as during other parts of the distributed model checking algorithm, BDDs are shipped between the processes. A compact and universal BDD representation is used, as described in Chapter **??**, for the communication. To send a local BDD structure, the process first converts it to the universal representation, then sends it to a different process which converts the universal representation back to its local BDD structure. Different variable order is allowed in the different processes. The size of the universal representation is independent of local variable ordering, and it is linear in the BDD size. Converting a universal represented BDD into the receiver BDD structure (according to the local variable order) may sometimes involve higher complexity (up to exponential in certain cases).

7.3 Distributed Model Checking for μ–Calculus.

The general idea of the distributed algorithm is as follows. The algorithm consists of two phases. The initial phase starts as the sequential algorithm, described in Section 7.2. It terminates when the memory requirement reaches a given threshold. At this point, the distributed phase begins. In order to distribute the work among the processes, the state space is partitioned into several parts, using a slicing procedure. Throughout the distributed phase, each process *owns* one part of the state space for every set of states associated with a certain subformula. When a computation of a subformula produces states owned by other processes, these states are sent out to the respective processes. A memory balancing mechanism is used to repartition imbalanced sets of states produced during the computation. A distributed termination algorithm is used to announce global termination. In the rest of this section we describe elements used by this algorithm.

7.3.1 Switching to the Distributed Phase

When the initial phase terminates, several subformulas have already been evaluated and the sets of states associated with them have been stored. In order to start the distributed phase, we slice the sets of states found so far and distribute the slices among the processes.

Each set of states is represented by a BDD and its size is measured by the number of BDD nodes. In each process all sets are managed by the same BDD manager, where parts of the BDDs that are used by several sets are shared and stored only once. Thus, two factors affect the partitioning of the sets: the required storage space for the sets, and the space needed to manipulate them. In order to keep the first factor small, it is best to partition the sets so that the space used by the BDD manager for all sets in each process is small. To keep the second factor small, each part of each set in each process should also be kept small. This is possible because the memory used in performing an operation is proportional to the size of the set it is applied to.

In model checking, the most acute peaks in memory requirement usually occur while operations are being performed. Thus, it is more important to reduce the second factor. Indeed, rather than minimizing the total size of each process, our algorithm slices each set in a way that reduces the size of its parts. As a result, the slicing criterion may differ for different sets. We use a slicing algorithm[42] described generally in Section 7.2.2. Slicing is applied to each one of the sets that has already been evaluated when phase switching occurs.

The slicing algorithm updates two tables: $InitEval$ and $InitSet$. $InitEval$ keeps track of which sets have been evaluated by the initial phase of the algorithm. $InitEval(f)$ is $True$ if and only if f has been evaluated by the initial algorithm. Each process id has the table $InitSet$, which for each formula f such that $InitEval(f) = True$, holds the subset of the set of states satisfying f and owned by this process. Formally, for each process id, and for each formula f, if $InitEval(f) = True$ then $InitSet(f) = f \wedge W_{id}$. The distributed phase will start by sending the tables $InitEval$ and $InitSet$, as well as the list of slices W_i, to all the processes.

7.3.2 The Distributed Phase

The distributed version of the model checking algorithm for the μ–calculus is given in Figure 7.3. While the sequential algorithm finds the set of states that satisfy, in a given model, a formula of the μ–calculus logic, each process in the distributed algorithm finds the part of this set that the process owns. Intuitively, the distributed algorithm works as follows: given a set of slices W_i, a formula f, and an environment e, the process id finds the set of states $\texttt{eval}(f, e) \wedge W_{id}$.

In fact, a weaker property is required in order to guarantee the correctness of the algorithm. It is enough to know that when evaluating a formula f, every state satisfying f is collected by at least one of the processes. For efficiency, however, we require in addition that every state be collected by exactly one process.

Given a formula f, the algorithm first checks if the initial phase has already evaluated it by checking if $InitEval(f) = True$. If so, it uses the result stored in $InitSet(f)$. Otherwise, it evaluates the formula recursively. Each recursive application associates a set of states with some subformula.

Preserving the work load is an inherent problem in distributed computation. If the memory requirement in one of the processes is significantly larger than in the others, the effectiveness of the distributed system is disrupted. To avoid this situation, a memory balance procedure is invoked whenever a new set of states is created, in order to maintain a balanced memory requirement for the new set. The memory balance procedure changes the slices W_i and updates the parts of the new set in each of the processes accordingly. Old sets are kept unchanged. Since each set is balanced, so is the overall memory requirement.

Each process in the distributed algorithm evaluates each subformula f as follows (see Figure 7.3):

A propositional formula $p \in AP$: evaluated by collecting all the states s that satisfy two conditions: p is in the labelling $L(s)$ of s and, in addition, s is owned by this process.

A relational variable Q: evaluated using the local environment of the process. Since only closed μ–calculus formulas, every variable is in the scope of μ or ν, are evaluated, the environment must have a value for Q (computed in a previous step).

A subformula of the form $\neg g$: evaluated by first evaluating g, and then using the special function `exchnot`. Given a set of states S and a partition $S_1, \ldots, S_k$ of S, each process i runs the procedure `exchnot` on S_i. The process reports to all the other processes about the states that do not belong to S "as far as it knows." Since each state in S belongs to some process, if none of the processes knows that s is in S, then s is in $\neg S$.

Since each process holds only the states of $\neg S$ that it owns, the processes only send states that are owned by the receiver. This reduces communication.

A subformula of the form $g_1 \vee g_2$: evaluated by first evaluating g_1 and g_2, possibly with different slicing functions. This means that a process can hold a part of g_1 with respect to one slicing and a part of g_2 with respect to another slicing. Nevertheless, since each state of g_1 and of g_2 belongs to one of the processes, each state of $g_1 \vee g_2$ now belongs to one of the processes as well. Applying the function `exch` results in a correct distribution of the states among the processes, according to the current slicing.

A subformula of the form $g_1 \wedge g_2$ can be translated, using De Morgan's laws, to $\neg(\neg g_1 \vee \neg g_2)$. However, evaluating the translated formula requires four communication phases (via `exch` and `exchnot`). Instead, such a formula is evaluated by first evaluating g_1 and g_2. As in the previous case, they might be evaluated with respect to different window functions. Here, however, the slicing of the two formulas should agree before a conjunction can be applied. This is achieved by applying `exch` twice, thus reducing the overall communication to only two rounds.

A subformula of the form **EX**g: evaluated by first evaluating g and then computing the pre-image using the transition relation R. Since every state of g belongs to one of the processes, every state of the pre-image also belongs to one of the processes . In fact, a state may be computed by more than one process if it is obtained as a pre-image of two parts. Applying `exch` completes the evaluation correctly.

Subformulas of the form $\mu Q.g$ and $\nu Q.g$ (the least and greatest fixpoints, respectively): evaluated using a special function `fixpt` that iterates until a fixpoint is found. The computations for the formulas differ only in the initialization, which is $False$ for $\mu Q.g$ and is the current window function for $\nu Q.g$. The `fixpt` function uses a distribution termination detection procedure, `parterm`, to check whether a fixpoint has been reached. Each process calls `parterm` with a Boolean value. The process reports true if and only if a fixpoint has been reached "as far as it knows." The fixpoint is evaluated by applying `exch` on both the last and current value of Q and comparing the parts that the process owns. Since each state belongs to some process, a fixpoint is reached if none of the processes gets a new state during the last iteration.

7.4 Correctness

In this section we prove the correctness of the distributed algorithm, assuming the sequential algorithm is correct. The sequential algorithm evaluates a formula by computing the set of states that satisfy it. In the distributed algorithm every such set is partitioned among the processes. The union over all the partitions for a given subformula is called the *global set*. In the proof we show that, for every μ–calculus formula, the set of states computed by the sequential algorithm is identical to the global set computed by the distributed algorithm. Note that the global set is never actually computed and is introduced only for the sake of the correctness proof. In the proof that follows we need the following definition.

Definition 6: [Well-partitioned environment] An environment e is well partitioned by parts $e_1, \ldots, e_k$ if and only if, for every $Q \in VAR$, $e(Q) = \bigvee_{i=1}^{k} e_i(Q)$.

The procedures `exch` are applied by all processes with a set of non-disjoint subsets S_i that cover a set res. Given a set of window functions, the procedures exchange non-owned parts so that at termination each process has all the states from res that it owns. The set of window functions does not change. Lemma 1 defines the relationship between the output of the procedure `exch` and the current set of window functions.

```
function peval($f, e$)
  case
    $InitEval(f)$:   return($InitSet(f)$)
    $f$= $p$:         $res$= $\{s \mid p \in L(s)\} \wedge W_{id}$
    $f$= $Q$:         return ($e(Q)$)
    $f$= $\neg g$:       $res$= exchnot(peval($g, e$))
    $f$= $g_1 \vee g_2$:   $res$= exch(peval($g_1, e$)$\vee$peval($g_2, e$))
    $f$= $g_1 \wedge g_2$:   $res_1$= peval($g_1, e$) $res_2$= peval($g_2, e$)
                     $res$= exch($res_1$)$\wedge$exch($res_2$)
    $f$= $\mathbf{EX}g$:      $res$= exch($\{s \mid \exists t[sRt \wedge t \in$peval($g, e$)]$\}$)
    $f$= $\mu Q.g$:      $res$= fixpt($Q, g, e, False$)
    $f$= $\nu Q.g$:      $res$= fixpt($Q, g, e, W_{id}$)
  endcase
  ldBlnc(res) /* balances W; updates res accordingly */
  return(res)
end function

function fixpt($Q, g, e, init$)
  $Q_{val}$= $init$
  repeat
    $Q_{old}$= $Q_{val}$
    $Q_{val}$= peval($g, e[Q \leftarrow Q_{old}]$)
  until (parterm(exch($Q_{val}$)=exch($Q_{old}$)))
  return $Q_{val}$
end function

function exch($S$)
  res= $S \wedge W_{id}$
  for each process $i \neq id$
    sendto($i, S \wedge W_i$)
  for each process $i \neq id$
    $res$= $res \vee$ receivefrom($i$)
  return $res$
end function

function exchnot($S$)
$res$= $(\neg S) \wedge W_{id}$
for each process $i \neq id$
  sendto($i, (\neg S) \wedge W_i$)
for each process $i \neq id$
  $res$= $res \wedge$ receivefrom($i$)
return $res$
end function
```

Figure 7.3: Pseudo–code for a process id in the distributed model checking

Lemma 1 *[`exch` procedure] Let $W_1, \ldots, W_k$ be a set of window functions and res be a set of states. Assume that each process id runs procedure `exch` with subset S_{id}, where $\bigvee_{i=1}^{k} S_i = res$. Then the set of window functions does not change and, after all procedures terminate, each process id has $res_{id} = res \wedge W_{id} = \bigvee_{i=1}^{k} S_i \wedge W_{id}$.*

Proof: At termination of procedure `exch`, process id has the following set:

$$res_{id} = (S_{id} \wedge W_{id}) \vee \bigvee_{j \neq id} (S_j \wedge W_{id}) \;=\; \bigvee_{i=1}^{k} S_i \wedge W_{id} \;=\; res \wedge W_{id}.$$

Q.E.D.
Let f be a μ–calculus formula and e_{id} be the environment in process id.
$\texttt{peval}_{id}(f, e_{id})$ denotes the set of states returned by procedure `peval`, when run by process id on f and e_{id}.
Theorem 1 defines the relationship between the outputs of the sequential and the distributed algorithms.

Theorem 1 (Correctness) *Let f be a μ–calculus formula, $W_1 \ldots W_k$ be a complete set of window functions, and $W'_1 \ldots W'_k$ be the set of window functions when `eval`(f, e) terminates. In addition, let e be a well–partitioned environment by $e_1, \ldots e_k$, and e' be the environment when `eval`(f, e) terminates. Furthermore, for all $i = 1, \ldots, k$, let e'_i be the environment when $\texttt{peval}_i(f, e_i)$ terminates. Then e' is well partitioned by $e'_1, \ldots e'_k$, $W'_1 \ldots W'_k$ is a complete set of window functions, and `eval`(f, e) = $\bigvee_{i=1}^{k} \texttt{peval}_i(f, e_i)$.*

It follows trivially from Theorem 1 that the disjunction of all the parts of a set evaluated by the processes for a function f is equal to the entire set evaluated by the sequential algorithm.

Proof: We prove the theorem by induction on the structure of f. In all but the last two cases of the induction step the environments do not change, and therefore e' is well partitioned by $e'_1, \ldots e'_k$.

The set of window functions is modified by applying `ldBlnc` at the end of `peval`. The procedure `ldBlnc` repartitions the subsets between the processes. However, their disjunction remains the same. Therefore, $W'_1 \ldots W'_k$ is a complete set of window functions.
Base: $f = p$ for $p \in AP$ $\bigvee_{i=1}^{k} \texttt{peval}_i(f, e_i) = \bigvee_{i=1}^{k} (\{s \mid p \in L(s)\} \wedge W_i) = \{s \mid p \in L(s)\} \wedge \bigvee_{i=1}^{k} W_i$.
Since $\bigvee_{i=1}^{k} W_i = 1$ (the set of window functions is complete), the above expression is equal to $\{s \mid p \in L(s)\}$, which is exactly $\texttt{eval}(f, e)$.

Induction:

1. $\underline{f = Q}$, where $Q \in VAR$ is a relational variable: $\bigvee_{i=1}^{k} \texttt{peval}_i(Q, e_i) = \bigvee_{i=1}^{k} e_i(Q)$. Since e is well partitioned, $e(Q) = \bigvee_{i=1}^{k} e_i(Q)$, which is equal to $\texttt{eval}(f, e)$.

2. $\underline{f = \neg g}$: $\texttt{peval}_{id}$($\neg g, e_{id}$) first applies $\texttt{peval}_{id}$(g, e_{id}), which results in S_{id}. It then runs the procedure $\texttt{exchnot}$(S_{id}), which returns the result res_{id}.

$$res_{id} = ((\neg S_{id}) \wedge W_{id}) \wedge \bigwedge_{j \neq id} ((\neg S_j) \wedge W_{id}) \ = \ \bigwedge_{j=1}^{k} ((\neg S_j) \wedge W_{id}).$$

When $\texttt{exchnot}$ terminates in all processes, the global set computed by all processes is (recall that $\bigvee_{i=1}^{k} W_i = 1$):

$$\bigvee_{i=1}^{k} \left(\bigwedge_{j=1}^{k} ((\neg S_j) \wedge W_i) \right) \ = \ \bigwedge_{j=1}^{k} (\neg S_j) \wedge \bigvee_{i=1}^{k} W_i \ = \ \bigwedge_{j=1}^{k} (\neg S_j) \ = \ \neg \bigvee_{j=1}^{k} S_j.$$

Since $S_i = \texttt{peval}_i(g, e_i)$, $\neg \bigvee_{j=1}^{k} S_j = \neg \bigvee_{j=1}^{k} \texttt{peval}_i(g, e_i)$, which by the induction hypothesis is identical to $\neg$ $\texttt{eval}$(g, e). This, in turn, is identical to $\texttt{eval}$($\neg g, e$). Thus, $\texttt{eval}$($\neg g, e$)= $\bigvee_{i=1}^{k} \texttt{peval}_i$($\neg g, e_i$).

3. $\underline{f = g_1 \vee g_2}$: $\texttt{peval}_{id}(g_1 \vee g_2, e_{id})$ first computes $\texttt{peval}_{id}(g_1, e_{id}) \vee$ $\texttt{peval}_{id}(g_2, e_{id})$. At the end of this computation, the global set is:

$$\bigvee_{i=1}^{k} (\texttt{peval}_i(g_1, e_i) \vee \texttt{peval}_i(g_2, e_i)) = \bigvee_{i=1}^{k} \texttt{peval}_i(g_1, e_i) \vee \bigvee_{i=1}^{k} \texttt{peval}_i(g_2, e_i).$$

By the induction hypothesis, this is identical to $\texttt{eval}$(g_1, e) $\vee$ $\texttt{eval}$(g_2, e), which is identical to $\texttt{eval}$($g_1 \vee g_2, e$). Applying the procedures $\texttt{exch}$ and $\texttt{ldBlnc}$ changes the partition of the sets among the processes, but not the global set.

4. $\underline{f = g_1 \wedge g_2}$: $\texttt{peval}_{id}(g_1 \wedge g_2, e_{id})$ first computes the two sets $res_1^{id} =$ $\texttt{peval}_{id}(g_1, e_{id})$ and $res_2^{id} = \texttt{peval}_{id}(g_2, e_{id})$, then applies $\texttt{exch}$ to each of them, and finally conjuncts the results. Note that no $\texttt{ldBlnc}$ is invoked between the two applications of $\texttt{exch}$. Therefore, both use the same window functions. Let $W_1, \ldots, W_k$ be those window functions. Then the global set is

$$\bigvee_{i=1}^{k} res_i = \bigvee_{i=1}^{k} (\texttt{exch}(res_1^i) \wedge \texttt{exch}(res_2^i)) \ =$$

$$\bigvee_{i=1}^{k} \left((W_i \wedge \bigvee_{j=1}^{k} res_1^j) \ \wedge \ (W_i \wedge \bigvee_{j=1}^{k} res_2^j) \right).$$

By the induction hypothesis, $\bigvee_{j=1}^{k} res_1^j = \texttt{eval}(g_1, e)$ and $\bigvee_{j=1}^{k} res_2^j = \texttt{eval}(g_2, e)$. Thus,

$$\bigvee_{i=1}^{k} res_i \ = \ \bigvee_{i=1}^{k} (\texttt{eval}(g_1, e) \ \wedge \ \texttt{eval}(g_2, e) \wedge W_i) \ =$$

$$\texttt{eval}(g_1 \wedge g_2, e) \wedge \bigvee_{i=1}^{k} W_i \;=\; \texttt{eval}(g_1 \wedge g_2, e).$$

Applying `ldBlnc` does not change the global set; thus $\bigvee_{i=1}^{k} \texttt{peval}_i(g_1 \wedge g_2, e_i) = \texttt{eval}(g_1 \wedge g_2, e)$.

5. $\underline{f = \textbf{EX}\ g}$: $\texttt{peval}_{id}(\textbf{EX}g, e_{id})$ evaluates the set of all predecessors of states in $\texttt{peval}_{id}(g, e_{id})$, using the transition relation R. The global set of all predecessors s can be represented by the formula $\bigvee_{i=1}^{k} \exists t[(s,t) \in R \wedge t \in \texttt{peval}_i(g, e_i)]$. The global set computed at this stage is:
$$\bigvee_{i=1}^{k} \exists t \left[(s,t) \in R \wedge t \in \texttt{peval}_i(g, e_i)\right].$$
Since disjunction and existential quantification are commutative, the above formula is identical to
$$\exists t \left[\bigvee_{i=1}^{k} (s,t) \in R \wedge t \in \texttt{peval}_i(g, e_i)\right] = \exists t \left[(s,t) \in R \wedge t \in \bigvee_{i=1}^{k} \texttt{peval}_i(g, e_i)\right].$$
By the induction hypothesis, $\bigvee_{i=1}^{k} \texttt{peval}_i(g, e_i) = \texttt{eval}(g, e)$. Thus, the global set is identical to
$$\exists t \left[(s,t) \in R \wedge t \in \texttt{eval}(g, e)\right] \;=\; \texttt{eval}(\ \textbf{EX}\ g, e).$$
Since the procedures `exch` and `ldBlnc` do not change the global set, $\bigvee_{i=1}^{k} \texttt{peval}_i(\textbf{EX}g, e_i)$ = $\texttt{eval}(\textbf{EX}g, e)$.

6. $\underline{f = \mu Q.g}$, a least fixpoint formula: $\texttt{peval}_{id}(\mu Q.g, e_{id})$ evaluates the least fixpoint formula by calling $\texttt{fixpt}_{id}(Q, g, e_{id}, False)$). Similarly, the sequential algorithm, $\texttt{eval}(\mu Q.g, e)$, evaluates the least fixpoint formula by calling the sequential function $\texttt{fixpt}(Q, g, e, False)$). As in previous cases, we would like to prove that $\bigvee_{i=1}^{k} \texttt{peval}_i(\mu Q.g, e_i)$ =
$\texttt{eval}(\mu Q.g, e)$. Since `ldBlnc` does not change the correctness of this claim, we only need to prove that $\bigvee_{i=1}^{k} \texttt{fixpt}_i(Q, g, e_i, False)$) =
$\texttt{fixpt}(Q, g, e, False)$). In addition, we need to show that the environment remains well partitioned when the computation terminates. The following lemma proves stronger requirements. It shows that at every iteration, the results of the sequential algorithm are identical to the global results of the distributed algorithm and that both algorithms terminate at the same iteration. This guarantees that the results at termination match. The lemma also proves that the environment is well partitioned at every iteration. The lemma uses the following property of procedure `parterm`.

Property 1: Procedure `parterm` is invoked by each of the processes with a Boolean parameter. If all processes send $True$, then `parterm` returns $True$ to all processes. Otherwise, it returns $False$ to all processes.

Lemma 2 *Let Q^j be the value of Q_{val} in iteration j of the sequential fixpoint algorithm. Similarly, let Q^j_{id} be the value of Q_{val} in iteration j of the distributed fixpoint algorithm in process id. Q^0 is the initialization of the sequential algorithm; Q^0_{id} is the initialization of the distributed algorithm. Then,*

- *(a) At every iteration, e is well partitioned by $e_1, \ldots, e_k$.*
- *(b) For every j: $Q^j = \bigvee_{i=1}^{k} Q^j_i$.*
- *(c) If the sequential `fixpt` algorithm terminates after i_0 iterations, then so does the distributed `fixpt` algorithm.*

Proof: We prove the lemma by induction on the number j of iterations in the loop of the sequential function `fixpt`.

Base: $j = 0$:

- (a) At iteration 0, e is well partitioned, according to the induction hypothesis of Theorem 1.
- (b) In the case that $f = \mu Q.g$, the value of both the sequential and the distributed algorithm at initialization is $False$. Hence, $Q^0 = Q^0_{id} = False$, which implies $Q^0 = \bigvee_{i=1}^{k} Q^0_i$.
- (c) Since both algorithms perform at least one iteration, they will not terminate at iteration 0.

Induction: Assume Lemma 2 holds for iteration j. We prove it for iteration $j + 1$.

- (a) Let e', $e'_1, \ldots, e'_k$ be the environments at the end of iteration $j + 1$, and assume that e is well partitioned by $e_1, \ldots, e_k$ at the end of iteration j. The only changes to the environments in iteration $j + 1$ may occur in line 5 of the distributed and sequential algorithms. Changes may occur for two reasons: $e(Q)$ may be assigned a new value Q^j, or a recursive call to `eval` may change e. Similarly, in the distributed algorithm, two changes may occur: $e_{id}(Q)$ may be assigned a new value Q^j_{id}, or a recursive call to $\texttt{peval}_{id}$ may change e_{id}.

  By the induction hypothesis of Lemma 2 we know that $Q^j = \bigvee_{i=1}^{k} Q^j_i$. Hence, $e[Q \leftarrow Q^j](Q) = \bigvee_{i=1}^{k} e_i[Q \leftarrow Q^j_i](Q)$. Since no other change has been made to the environments, and since e is well partitioned, we conclude that $e[Q \leftarrow Q^j]$ is well partitioned by $e_1[Q \leftarrow Q^j_1], \ldots, e_k[Q \leftarrow Q^j_k]$.

  In iteration $j + 1$, `eval` is now invoked with an environment that is well partitioned by the environments $\texttt{peval}_{id}$ is invoked with. The induction hypothesis of Theorem 1 therefore guarantees that e' is well partitioned by $e'_1, \ldots, e'_k$.

(b) Q^{j+1} = $\texttt{eval}(g, e[Q \leftarrow Q^j])$ (line 5 of the sequential algorithm) and Q^{j+1}_{id} = $\texttt{peval}_{id}(g, e[Q \leftarrow Q^j_{id}])$ (line 5 of the distributed algorithm).

By item (a), $e[Q \leftarrow Q^j]$ is well partitioned. Thus, the induction hypothesis of Theorem 1 is applicable and implies that

$$\texttt{eval}(g, e[Q \leftarrow Q^j]) = \bigvee_{i=1}^{k} \texttt{peval}_i(g, e[Q \leftarrow Q^j_i]).$$

Hence, $Q^{j+1} = \bigvee_{i=1}^{k} Q^{j+1}_i$.

(c) The sequential `fixpt` procedure terminates at iteration $j+1$ if $Q^j = Q^{j+1}$. We prove that this holds if and only if for every process id, $\texttt{exch}(Q^j_{id})$ = $\texttt{exch}(Q^{j+1}_{id})$, and therefore `parterm` returns $True$ to all processes.

Let $W_1, \ldots, W_k$ be the current window functions. By item (b), $Q^j = \bigvee_{i=1}^{k} Q^j_i$ and $Q^{j+1} = \bigvee_{i=1}^{k} Q^{j+1}_i$.

$$\forall id[\texttt{exch}(Q^j_{id}) = \texttt{exch}(Q^{j+1}_{id})] \;\Leftrightarrow$$

$$\forall id[\bigvee_{i=1}^{k} Q^j_i \wedge W_{id} \;=\; \bigvee_{i=1}^{k} Q^{j+1}_i \wedge W_{id}] \;\Leftrightarrow$$

$$\forall id[Q^j \wedge W_{id} \;=\; Q^{j+1} \wedge W_{id}] \;\Leftrightarrow\; Q^j \;=\; Q^{j+1}.$$

The last equality is implied by the previous one because the window functions are complete. This completes the proof of the lemma. Q.E.D.

7. $\underline{f = \nu Q.g}$, a greatest fixpoint formula: The proof for this case is almost identical to the previous one. The only change should be made to the definition of Q^0, Q^0_i in the statement of the lemma, so that $Q^0 = True$ and $Q^0_i = W_i$. The proof of second bullet in the base case should be changed accordingly. This completes the proof. Q.E.D.

7.4.1 The Processes Own Disjoint Subsets

Theorem 1 can be extended to state that when all procedures $\texttt{peval}_{id}(f, e_{id})$ terminate, the subsets owned by each of the processes are disjoint. This is important in order to avoid duplication of work. A set of window functions that defines disjoint ownership is presented in the following definition:

Definition 7: [Disjoint Set of Window Functions] A set of window functions $W_1, \ldots, W_k$ is disjoint if and only if, for every $1 \leq t, l \leq k$, $t \neq l$, $W_t \wedge W_l = 0$.

The distributed algorithm uses the `exchange` procedure to store disjoint subsets of each set. The following lemma specifies this property:

Lemma 3 *[`exch` procedure makes disjoint parts] Let $W_1, \dots, W_k$ be a set of disjoint window functions and S be a set of states. Assume that each process id runs procedure `exch` with a subset S_{id}. Then at termination of the procedures in all processes, for every $1 \le t, l \le k$, $t \ne l$, $\texttt{exch}(S_t) \wedge \texttt{exch}(S_l) = 0$.*

Proof: By Lemma 1, at termination of procedure `exch`, for every $1 \le t, l \le k$, $t \ne l$,$res_t \wedge res_l = (\bigvee_{j=1}^{k} S_j \wedge W_t) \wedge (\bigvee_{j=1}^{k} S_j \wedge W_l)$. Since W_i is a set of disjoint window functions, the last expression equals 0. Q.E.D.

We now show that, for every μ–calculus formula, the subsets computed by the distributed algorithm are disjoint. In the proof that follows we need the following definition.

Definition 8: [Disjoint Environment] Environment parts $e_1, \dots, e_k$ are disjoint if and only if, for every $Q \in VAR$, for every $1 \le t, l \le k$, $t \ne l$, $e_t(Q) \wedge e_l(Q) = 0$.

Theorem 2 proves that given a disjoint set of window functions, the distributed algorithm returns disjoint results.

Theorem 2 (The Processes Own Disjoint Subsets) *Let f be a μ–calculus formula, $W_1 \dots W_k$ be a disjoint set of window functions, and $W'_1 \dots W'_k$ be the set of window functions when `eval`(f, e) terminates. In addition, let $e_1, \dots e_k$ be disjoint environment parts, and for all $i = 1, \dots, k$, let e'_i be the environment when $\texttt{peval}_i(f, e_i)$ terminates. Then $e'_1, \dots e'_k$ are disjoint environment parts, $W'_1 \dots W'_k$ is a disjoint set of window functions, and for every $1 \le t, l \le k$, $t \ne l$,*

$$\texttt{peval}_t(f, e_t) \wedge \texttt{peval}_l(f, e_l) = 0.$$

Proof: We prove the theorem by induction on the structure of f. In all but the last two cases of the induction step the environments are not changed and therefore $e'_1, \dots e'_k$ are disjoint.

The set of window functions is modified by applying `ldBlnc` at the end of `peval`. The procedure `ldBlnc` repartitions the subsets between the processes. However, the set of window functions remains disjoint. Therefore, $W'_1 \dots W'_k$ is a disjoint set of window functions.

Base: $f = p$ for $p \in AP$ for every $1 \le t, l \le k, t \ne l$, $\texttt{peval}_t(f, e_t) \wedge \texttt{peval}_l(f, e_l) = \{s \mid p \in L(s)\} \wedge W_t \wedge \{s \mid p \in L(s)\} \wedge W_l$.

Since for every $1 \le t, l \le k$, $t \ne l$, $W_t \wedge W_l = 0$ (the set of window functions is disjoint), the above expression is equal to 0.

Induction step:

1. $\underline{f = Q}$, where $Q \in VAR$ is a relational variable: for every $1 \le t, l \le k$, $t \ne l$,$\texttt{peval}_t(f, e_t) \wedge \texttt{peval}_l(f, e_l) = e_t(Q) \wedge e_l(Q)$. Since $e_1, \dots, e_k$ are disjoint, the last expression equals 0.

2. $\underline{f = \neg g}$: `peval`$_{id}(\neg g, e_{id})$ first applies `peval`$_{id}(g, e_{id})$, which results in S_{id}. It then runs the procedure `exchnot`(S_{id}), which returns the result res_{id}.

$$res_{id} = ((\neg S_{id}) \wedge W_{id}) \wedge \bigwedge_{j \neq id} ((\neg S_j) \wedge W_{id}) \;=\; \bigwedge_{j=1}^{k} ((\neg S_j) \wedge W_{id}).$$

Therefore, for every $1 \leq t, l \leq k$, $t \neq l$,`peval`$_t(f, e_t) \wedge$ `peval`$_l(f, e_l) = res_t \wedge res_l =$

$$\bigwedge_{j=1}^{k} ((\neg S_j) \wedge W_t) \wedge \bigwedge_{j=1}^{k} ((\neg S_j) \wedge W_l).$$

Since $W_t \wedge W_l = 0$, the above expression is equal to 0. Applying `ldBlnc` at the end of `peval` repartitions the subsets between the processes; however, the subsets remain disjoint. Thus, for every $1 \leq t, l \leq k$,$t \neq l$,`peval`$_t(f, e_t) \wedge$ `peval`$_l(f, e_l) = 0$.

3. $\underline{f = g_1 \vee g_2}$: `peval`$_{id}(g_1 \vee g_2, e_{id})$ first computes the disjunction of `peval`$_{id}(g_1, e_{id})$ and `peval`$_{id}(g_2, e_{id})$, which results in S_{id}. Then it runs the procedure `exch`(S_{id}). Therefore, for every $1 \leq t, l \leq k$, $t \neq l$, `peval`$_t(f, e_t) \wedge$`peval`$_l(f, e_l)$ =`exch`(S_t)$\wedge$`exch`(S_l). By the induction hypothesis, the window functions used by `exch` are disjoint. Therefore we can apply Lemma 3, which ensures that the last expression equals 0.

4. $\underline{f = g_1 \wedge g_2}$: `peval`$_{id}(g_1 \wedge g_2, e_{id})$ first computes the two sets $res_1^{id} =$ `peval`$_{id}(g_1, e_{id})$ and $res_2^{id} =$ `peval`$_{id}(g_2, e_{id})$. It then applies `exch` to each set and conjuncts the results. Therefore, for every $1 \leq t, l \leq k$, $t \neq l$, `peval`$_t(f, e_t) \wedge$ `peval`$_l(f, e_l) =$ `exch`(res_1^t)$\wedge$ `exch`(res_2^t)$\wedge$ `exch`(res_1^l)$\wedge$ `exch`(res_2^l). Lemma 3 ensures that the last expression equals 0.

5. $\underline{f = \textbf{EX}\, g}$: `peval`$_{id}(\textbf{EX} g, e_{id})$ evaluates the set of all predecessors of states in `peval`$_{id}(g, e_{id})$, which results in S_{id}. It then runs the procedure `exch`(S_{id}). Therefore, for every $1 \leq t, l \leq k$, $t \neq l$,`peval`$_t(f, e_t) \wedge$ `peval`$_l(f, e_l)$ =`exch`(S_t)$\wedge$`exch`(S_l). Lemma 3 ensures that the last expression equals 0.

6. $\underline{f = \mu Q.g}$, a least fixpoint formula: `peval`$_{id}(\mu Q.g, e_{id})$ evaluates the least fixpoint formula by calling `fixpt`$_{id}(Q, g, e_{id}, False)$). As in previous cases, we would like to prove that for every $1 \leq t, l \leq k$, $t \neq l$,`peval`$_t(f, e_t) \wedge$ `peval`$_l(f, e_l) = 0$. Since `ldBlnc` does not change the correctness of this claim, we only need to prove that for every $1 \leq t, l \leq k$, $t \neq l$,`fixpt`$_t(Q, g, e_t, False)$)$\wedge$ `fixpt`$_l(Q, g, e_l, False)$) $= 0$. In addition, we need to show that the environment remains disjoint when the computation terminates. The following lemma proves stronger requirements. It shows that at every iteration, the results and the environment parts are disjoint. This guarantees that at termination they are disjoint as well.

Lemma 4 *Let Q_{id}^j be the value of Q_{val} in iteration j of the fixpoint algorithm in process id. Q_{id}^0 is the value of Q_{val} at initialization. Then,*

(a) *At every iteration, $e_1, \ldots, e_k$ are disjoint.*

(b) *For every j,$1 \leq t, l \leq k$, $t \neq l$, $Q_t^j \wedge Q_l^j = 0$.*

Proof: We prove the lemma by induction on the number j of iterations in the loop of the function `fixpt`.

Base: $j = 0$:

(a) At iteration 0, $e_1, \ldots, e_k$ are disjoint, according to the induction hypothesis of Theorem 2.

(b) In case $f = \mu Q.g$, the initialization of the distributed algorithm is $False$. Hence, for every $1 \leq t, l \leq k$, $t \neq l$, $Q_t^0 = Q_l^0 = 0$, which implies $Q_t^0 \wedge Q_l^0 = 0$.

Induction step: Assume Lemma 4 holds for iteration j. We prove it for iteration $j+1$.

(a) Let $e'_1, \ldots, e'_k$ be the environments at the end of iteration $j + 1$, and assume that $e_1, \ldots, e_k$ are disjoint at the end of iteration j. The only changes to the environments in iteration $j + 1$ may occur in line 5 of the algorithms. Changes may occur for two reasons: $e_{id}(Q)$ may be assigned a new value Q_{id}^j, or a recursive call to $\texttt{peval}_{id}$ may change e_{id}.

By the induction hypothesis of Lemma 4 we know that for every $1 \leq t, l \leq k$, $t \neq l$, $Q_t^j \wedge Q_l^j = 0$. Hence, for every $1 \leq t, l \leq k$, $t \neq l$, $e_t[Q \leftarrow Q_t^j](Q) \wedge e_l[Q \leftarrow Q_l^j](Q) = 0$. Since no other change has been made to the environments, and since $e_1, \ldots, e_k$ are disjoint, we conclude that for every $1 \leq t, l \leq k$, $t \neq l$, $e_t[Q \leftarrow Q_t^{j+1}](Q) \wedge e_l[Q \leftarrow Q_l^{j+1}](Q) = 0$.

In iteration $j + 1$, $\texttt{peval}_{id}$ is now invoked with a disjoint environment. The induction hypothesis of Theorem 2 therefore guarantees that $e'_1, \ldots, e'_k$ are disjoint.

(b) Q_{id}^{j+1} = $\texttt{peval}_{id}(g, e[Q \leftarrow Q_{id}^j])$ (line 5 of the distributed algorithm).

By item (a), $e_{id}[Q \leftarrow Q_{id}^j]$ are disjoint. Thus, the induction hypothesis of Theorem 2 is applicable and implies that for every $1 \leq t, l \leq k$, $t \neq l$, $\texttt{peval}_t(g, e[Q \leftarrow Q_t^j]) \wedge \texttt{peval}_l(g, e[Q \leftarrow Q_l^j]) = 0$. Hence, for every $1 \leq t, l \leq k$, $t \neq l$, $Q_t^{j+1} \wedge Q_l^{j+1} = 0$.

This completes the proof of the lemma Q.E.D.

7. $\underline{f = \nu Q.g}$, a greatest fixpoint formula: The proof for this case is almost identical to the previous one. The only change should be made to the definition of Q_i^0 in the statement of the lemma, so that $Q_i^0 = W_i$. The proof of the second bullet in the base case should be changed accordingly. This completes the proof. Q.E.D.

7.5 Scalable Distributed Pre-image Computation

The main goal of our distributed algorithm is to reduce the memory requirement of the symbolic model checking operations. In symbolic model checking, pre-image is one of the operations with the highest memory requirement. Given a set of states S, pre-image computes $pred(S)$ (also denoted by $\mathbf{EX}\ S$ in μ-calculus), which is the set of all predecessors of states in S. The pre-image operation can be described by the formula $pred(S) = \exists s'[R(s,s') \wedge S(s')]$. It is easy to see that the memory requirement of this operation grows as the sizes of the transition relation R and the set S grow. Furthermore, intermediate results sometimes exceed the memory capacity even when $pred(S)$ can be held in memory.

Our distributed algorithm reduces memory requirements by slicing each of the computed sets of states. This takes care of the S parameter of a pre-image computation, but not of the R parameter. In order to make our method scalable for very large models, we need to reduce the size of the transition relation as well.

The transition relation consists of pairs of states. We distinguish between the source states and the target states by referring to the latter as St'. Thus, $R \subseteq St \times St'$.

A reduction of the second parameter of R, St', can be achieved by applying the well-known restriction operator [30]: Prior to any application of the pre-image computation, a process that owns a slice S_i of S reduces its copy of R by restricting St' to S_i. Since pre-image operations are applied to different sets during model checking, this reduction is *dynamic*.

We further reduce R by adding a *static* slicing of St according to (possibly different) window functions $U_1, \ldots, U_m$. The slicing algorithm of Section 7.2.2 can be used to produce $U_1, \ldots, U_m$, so that R is partitioned to m slices of similar size. Each slice R_j is a subset of $(St \cap U_j) \times St'$. Since R does not change during the computation, $U_1, \ldots, U_m$ do not change either.

Having k window functions $W_1, \ldots, W_k$ for S and m window functions $U_1, \ldots, U_m$ for R, we use $k \times m$ processes. All processes $(i,1), (i,2), \ldots, (i,m)$ have the same W_i and hence own the same $S_i = S \wedge W_i$. However, these processes have a different U_l. Process (i,l) with W_i and U_l computes the pre-image of S_i by $pred_j(S_i) = \exists s'[R_l(s,s') \wedge S_i(s')]$.

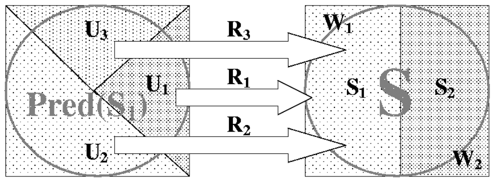


Figure 7.5 above demonstrates a pre-image computation using a sliced transition relation with $k = 2$ and $m = 3$. Given a set S sliced into S_1, S_2 according to W_1, W_2 respectively, the pre-image of S_1 is computed by three processes. Each process uses a different slice of the transition relation, R_1, R_2 and R_3, according to U_1, U_2 and U_3.

7.5.1 Model Checking Algorithm with Sliced Transition Relation

The algorithm `pareval`str(f, e) is similar to `peval`, but uses a sliced transition relation. Formulas not in the form of **EX**g do not use the transition relation. The algorithm works the same way as `peval` does on these formulas, using one process $(i, 1)$ for each window function W_i. The `exch` algorithm and the `ldBlnc` algorithm work only with the relevant processes $(1, 1)$,$(2, 1)$,. . .,$(k, 1)$.

A formula in the form **EX**g is evaluated by first using the processes $(1, 1)$,$(2, 1)$,. . .,$(k, 1)$ to evaluate g. Then each process $(i, 1)$ broadcasts its copy of g_i to the processes $(i, 2)$,. . .,(i, m). Each process (i, l) computes the pre-image of g_i using R_l. Finally, the processes use the algorithm `exchstr` (given in Figure 7.4) to complete the evaluation and update the processes $(1, 1)$,$(2, 1)$,. . .,$(k, 1)$.

```
functionexchstr($S, < uId, wId >$)
1   for all $1 \leq i \leq k$
2     sendto($< i, 1 >, S \wedge W_i$)
3   if $uId \neq 1$ return 0
    /* $uId = 1$ */
4   $res$= $\emptyset$
5   for all $1 \leq l \leq m$
6     for all $1 \leq i \leq k$
7       $res$= $res \vee$receivefrom($< i, l >$)
8   return $res$
end function
```

Figure 7.4: Pseudo–code for exchanging non-owned states after pre-image computation using the sliced transition relation

The method suggested in this section applies slicing to the full transition relation if it can be held in memory but is too big to enable a successful completion of the pre-image operation. However, the given transition relation is often *partitioned,* i.e., it is given as a set of small relations N_l, each defining the value of variable v_l in the next states. The size of the partitioned transition relation is usually small; therefore it can be constructed by one process and then sliced using the algorithm suggested in [55]. In this case, model checking is done directly with the partitioned transition relation [16].

7.5.2 Distributed Construction of the Sliced Full Transition Relation

In this section we consider cases in which the full transition relation R is a conjunction of all N_l. We consider cases where either the size of R or intermediate results during its construction cannot fit into the memory of a single process.

Our goal is to construct slices R_j of R, with none of the processes ever holding R. One process starts the construction by computing the conjunction of partitions N_l gradually, until a threshold is reached. The current (partial) transition relation is then sliced among the processes, using the slicing algorithm. Each process continues to conjunct the partitions that have not yet been handled, until all partitions are conjuncted. During the conjunction, further slicing or balancing are applied so that the final slices are balanced.

7.5.3 Correctness of the Algorithm with a Sliced Transition Relation

In this section we prove the correctness of the distributed algorithm `parevalstr`. Theorem 3 proves that the output of the distributed algorithm `parevalstr` and the output of the distributed algorithm `peval` are equal. In the proof that follows we need the following definition.

Definition 9: [Sliced Transition Relation] A transition relation R corresponds to a sliced transition relation $R_1, \ldots, R_m$ if and only if for every $1 \leq l \leq m$, $R_l = R \wedge U_l$, where $U_1, \ldots, U_m$ is a complete set of window functions.

Theorem 3 (Correctness with Sliced Transition Relation) *Let f be a μ–calculus formula and let R be a transition with the corresponding sliced transition relation $R_1, \ldots, R_m$. In addition, let $e_1, \ldots e_k$ be a distributed environment, e'_i be the environment when* $\texttt{peval}_i(f, e_i)$ *terminates, and e''_i be the environment when* $\texttt{parevalstr}_{i,1}(f, e_i)$ *terminates. Then, $e'_i = e''_i$ and* $\texttt{peval}_i(f, e_i)$*=* $\texttt{parevalstr}_{i,1}(f, e_i)$*.*

From Theorem 3 and Theorem 1 we can conclude that the union over the parts evaluated by all processes for a function f is equal to the entire set evaluated by the sequential algorithm.

Proof: We prove the theorem by induction on the structure of f.
$\texttt{parevalstr}_{i,1}(f, e_i)$ works the same way as $\texttt{peval}_i(f, e_i)$ does for all formulas except those of the form **EX**g. Therefore it is enough to prove the theorem only for formulas in the form **EX**g.
Base: $f = p$ for $p \in AP$. Immediate, since not **EX**g.
Induction:
$f =$ **EX** g: $\texttt{parevalstr}_{i,l}$(**EX**$g, e_i$) evaluates the set of all predecessors of states in $\texttt{parevalstr}_{i,1}(g, e_i)$, using the transition relation R_i. The set of all predecessors $s_{i,l}$ can be represented by the formula $\exists t[(s,t) \in R_i \wedge t \in \texttt{parevalstr}_{i,1}(g, e_i)]$. Then each process runs $\texttt{exchstr}(s_{i,l}, i, l)$ and places the results in $s'_{i,l}$. The result in processes $(wId, 1)$ is as follows:

$$s'_{wId,1} = \bigvee_{l=1}^{m} \bigvee_{i=1}^{k} s_{i,l} \wedge w_i$$

The above formula is therefore identical to:

$$w_i \wedge \bigvee_{l=1}^{m} \bigvee_{i=1}^{k} \exists t \left[(s,t) \in R_l \wedge t \in \texttt{parevalstr}_{i,1}(g, e_i)\right].$$

Since disjunction and existential quantification are commutative, the above formula is identical to

$$w_i \wedge \exists t \left[\bigvee_{i=1}^{k} (s,t) \in (\bigvee_{l=1}^{m} R_l) \wedge t \in \texttt{parevalstr}_{i,1}(g, e_i)\right].$$

Since R_l are sliced transition relations, the above formula is identical to:

$$w_i \wedge \exists t \left[(s,t) \in R \wedge t \in \bigvee_{i=1}^{k} \texttt{parevalstr}_{i,1}(g, e_i) \right].$$

By the induction hypothesis, $\texttt{parevalstr}_{i,1}(g, e_i) = \texttt{peval}_i(g, e_i)$. Thus, the set returned by process $(i, 1)$ is identical to

$$w_i \wedge \exists t \left[(s,t) \in R \wedge t \in \bigvee_{i=1}^{k} \texttt{peval}_i(g, e_i) \right].$$

The last expression is identical to:

$$w_i \wedge \texttt{peval}_i(\ \mathbf{EX}\ g, e_i).$$

Lemma 1 ensures that the set returned by procedure $\texttt{exch}(\texttt{peval}_i(\mathbf{EX}g, e_i))$ is identical to the above formula, and thus $\texttt{parevalstr}_{i,1}(\mathbf{EX}g, e_i) = \texttt{peval}_i(\mathbf{EX}g, e_i)$.
This completes the proof. Q.E.D.

7.6 Scalability

A distributed algorithm is scalable if it remains effective for large problems when running on a large number of nodes. The main factors that influence scalability are the memory requirement of the algorithm at each node and the communication volume. If the memory requirement at each node decreases as the number of nodes grows, the algorithm can probably handle larger problems by using a large number of nodes.

Our experience in previous work [42, 11] indicates that the bandwidth of the current standard network allows systems with a few dozen nodes to work effectively, and communication does not become a bottleneck. A very large network will need to handle larger communication volume.

There are two sources for the memory requirements of the algorithm: the memory required from each node to store the sets and the memory required to compute the image of a single set. Since each set is distributed evenly among the nodes by the `ldBlnc` procedure, the memory requirement from each node is expected to be balanced. Therefore, the memory required by each node is expected to decrease when the number of nodes increases.

The memory requirement for computing the image of a set depends on the set size. Since computation is applied to a balanced set, the size of each subset decreases linearly to the number of nodes. Therefore, the memory requirement for the computation is expected to decrease when the number of nodes increases.

The algorithm works bottom up through the formula, evaluating each subformula based on the value of its own subformulas. It evaluates each subformula using a number of nodes that work in parallel. However, the evaluation is synchronized by the call to the `ldBlnc`, `exch` and `exchnot` procedures. The evaluation takes a constant number of operations for all the operators except fixpoint. Lemma 2 proves that the distributed algorithm takes the same number of steps for fixpoint operators as the sequential. Therefore, we conclude that the complexity of the distributed algorithm is the same as in the sequential case. The complexity of evaluating a formula depends only on the number of alternations d of the least and greatest fixpoints [34]. A sequential [34, 49] algorithm requires n^d steps where n is the number of states in the transition system.

Our algorithm requires several standard machines, each consisting of local processors and local memory. The communication between the machines consists of a standard ethernet. The algorithm can be implemented using the MPI standard [35]. Therefore, it does not require any special architecture.

Chapter 8

Conclusions and Future Work

This work presents a distributed algorithm for symbolic reachability analysis that improves significantly on previous works. Its adaptability to any network size and its high utilization of network resources make it suitable for solving very large verification problems.

The experimental environment that is used to evaluate our new algorithm currently consists of NuSMV and the newly introduced Division system. Division is a new platform for distributed symbolic model checking research, featuring a high-level generic interface to "external" model checkers.

Currently, Division is in the final stages of interfacing with Intel's high-performance model checker Forest. We thus expect our results to improve substantially and to become better in the near future. In addition, the algorithm exhibits additional promising features, to be studied in depth in our future work.

1. Different variable orders for different workers may lead to high – even super linear – gains in reductions in memory requirement. However, transforming a BDD from one order to another may result in a memory overflow. With the new algorithm, if an overflow occurs while exchanging BDDs the exchange is halted and the owned state space is split. Then the exchange can resume.

2. We expect the new algorithm to execute on very large, possibly non-dedicated networks. Further study is required, however, because, in such unstable environments, machines may frequently crash and recover.

3. The algorithm can run on a network of heterogeneous machines, by setting different thresholds and parameters for different machines according to their characteristics.

4. The work efficient algorithm requires less synchronization than the one in Chapter **??**. It is a first step towards a fully asynchronous algorithm, mandatory for obtaining high speedups.

5. If the number of processes in the network is very large the coordinators can be implemented distributively by several processes.

6. Resuming image computation after splitting, from the point at which it stopped, is a novel technique that saves time as well as space. We believe that this technique will prove to be useful also for non distributed implementations in which the computation of reachable states is partitioned (e.g. [36, 5, 60]).

Bibliography

[1] A. Aziz, S. Tasiran, and R.K. Brayton. BDD Variable Ordering for Interacting Finite State Machines. In *31st ACM/IEEE Design Automation Conference (DAC)*, San Diego, CA, June 1994. San Diego Convention Center.

[2] N. Amla, R. Kurshan, K. McMillan, and Medel R. K. Experimental Analysis of Different Techniques for Bounded Model Checking. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03), LNCS*, Warsaw, Poland, 2003.

[3] T. E. Anderson, D.E Culler, and D. A. Patterson. A Case for NOW:Network of Workstations. Technical report, 1994.

[4] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. Springer-Verlag, 2002.

[5] S. Barner and O. Grumberg. Combining symmetry reduction and upper-approximation for symbolic model checking. In *Proc. of the 14th International Conference on Computer Aided Verification, LNCS*, 2002.

[6] J. Baumgartner and T. Heyman. An Overview and Application of Model Reduction Techniques in Formal Verification. In *IEEE International Perfomance, Computing, and Communications Conference*, pages 165–171, 1998.

[7] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The Temporal Logic Sugar. In *Proc. of the 13th International Conference on Computer Aided Verification*. Springer-Verlag, June 2001.

[8] I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: An Industry-Oriented Formal Verification Tool. In *33rd Design Automation Conference*, pages 655–660, 1996.

[9] I. Beer, S. Ben-David, and A. Landver. On-the-Fly Model Checking of RCTL Formulas. In *Proc. of the 10th International Conference on Computer Aided Verification, LNCS 818*, pages 184–194, 1998.

[10] S. Ben-David, O. Grumberg, T. Heyman, and A. Schuster. Scalable Distributed On-the-Fly Symbolic Model Checking. *Software Tools for Technology Transfer*, 4(4):496–504, November 2003.

[11] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable Distributed On-the-Fly Symbolic Model Checking. In *Third International Conference on Formal methods in Computer-Aided Design (FMCAD'00), LNCS*, Austin, Texas, November 2000.

[12] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient On-the-Fly Model Checking for CTL*. In *Proc. of the Conference on Logic in Computer Science (LNCS'95)*, June 1995.

[13] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking Without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems,* 5^{th} *International Conference, TACAS'99, LNCS 1579*, 1999.

[14] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. In *Proceedings of the ACM Symposium on Operating System Principles*, page 3, Bretton Woods, NH, 1983. Association for Computing Machinery.

[15] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[16] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the 1991 International Conference on Very Large Scale Integration*, August 1991.

[17] 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):142–171, June 1992. Special Issue: Selections from 1990 IEEE Symposium on Logic in Computer Science.

[18] Olaf Burkart and Bernhard Steffen. Model Checking for Context-free Processes. LNCS 630, pages 123–137. Springer, 1992.

[19] Olaf Burkart and Bernhard Steffen. Pushdown Processes: Parallel Composition and Model Checking. LNCS 836, pages 98–113. Springer, 1994.

[20] Kenneth M. Butler, Don E. Ross, Rohit Kapur, and M. Ray Mercer. Heuristics to Compute Variable Orderings for Efficient Manipulation of Ordered Binary Decision Diagrams. In ACM-SIGDA; IEEE, editor, *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 417–420, San Francisco, CA, June 1991. ACM Press.

[21] G. Cabodi, P. Camurati, and S. Quer. Improved Reachability Analysis of Large FSM. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 354–360. IEEE Computer Society Press, June 1996.

[22] G. Cabodi, P. Camurati, and S. Quer. Improving the Efficient of BDD-Bsaed Operators by Means of Partitioning. *IEEE Transactions on Computer-Aided Design*, pages 545–556, May 1999.

[23] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99), LNCS 1633*, pages 495–499, Trento, Italy, 1999.

[24] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *32rd Design Automation Conference*, pages 655–660, 1995.

[25] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, January 1983.

[26] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16, 5:1512–1542, September 1994.

[27] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In C. Courcoubetis, editor, *Proc. of the Fifth International Conference on Computer Aided Verification, LNCS 697*, Crete, 1993.

[28] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT press, December 1999.

[29] R. Cleaveland. Tableau-Based Model Checking in the Propositional μ-calculus. *Acta Informatica*, 27:725–747, 1990.

[30] O. Coudert, J. C. Madre, and C. Berthet. Verifying of Synchronous Sequential Machines Based on Symbolic Execution. In J. Sifakis, editor, *Workshop on Automatic Verification Methods for Finite State Systems*, pages 365–373. Springer-Verlag, Grenoble, France, 1989.

[31] Olivier Coudert, Jean C. Madre, and Christian Berthet. Verifying Temporal Properties of Sequential Machines without Building Their State Diagrams. In R. Kurshan and E. M. Clarke, editors, *Workshop on Computer Aided Verification, DIMACS, LNCS 531*, pages 23–32. Springer-Verlag, New Brunswick, NJ, June 1990.

[32] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1:275–288, 1992.

[33] E. Emerson and A. P. Sistla. Symmetry and Model Checking. In C. Courcoubetis, editor, *Proc. of the Fifth International Conference on Computer Aided Verification, LNCS 697*, Crete, 1993.

[34] E. A. Emerson and C.-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1986.

[35] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[36] R. Fraer, G. Kamhi, B. Ziv, M.Y. Vardi, and L. Fix. Prioritized Traversal: Efficient Reachability Analysis for Verification and Falsification. In *Proc. of the 12th International Conference on Computer Aided Verification, LNCS*, 2000.

[37] D. Geist and I. Beer. Efficient Model Checking by Automated Ordering of Transition Relation Partitions. In *Proc. of the Sixth International Conference on Computer Aided Verification, LNCS 818*, pages 299–310, 1994.

[38] O. Grumberg, A. Heyman, T. Heyman, and A. Schuster. Division System: A General Platform for Distributed Symbolic Model Checking Research, 2003. http://www.cs.technion.ac.il/Labs/dsl/projects/division_web/division.htm.

[39] O. Grumberg, T. Heyman, and A. Schuster. Distributed Model Checking for μ-calculus. In *Proc. of the 13th International Conference on Computer Aided Verification, LNCS*, 2001.

[40] O. Grumberg, T. Heyman, and A. Schuster. A Work-Efficient Distributed Algorithm for Reachability Analysis. In *Proc. of the 15th International Conference on Computer Aided Verification, LNCS*, 2003.

[41] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits. In *Proc. of the 12th International Conference on Computer Aided Verification, LNCS*, 2000.

[42] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits. *Formal Methods in System Design*, 21(2):317–338, November 2002.

[43] J.E. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesely, 1979.

[44] J. Jain, W. Adams, and M. Fujita. Sampling schemes for computing variable orderings. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 631–638. IEEE Computer Society Press, 1998.

[45] J. Jain, J. Bitner, J.A. Abraham, and D.S. Fussel. Functional Partitioning for Verification and Related Problems. In *Proc. Brown/MIT VLSI Conference*, pages 210–226, March 1992.

[46] S. Kimura and E.M. Clarke. Parallel Algorithms for Constructing Binary Decision Diagrams. In *Proc. of the International Conference on Computer Design*, pages 220–223, 1990.

[47] D. Kozen. Results on the propositional μ-calculus. *TCS*, 27, 1983.

[48] O. Lichtenstein and A. Pnueli. Checking that Finite State Concurrent Programs Satisfy their Linear Specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, January 1985.

[49] D. Long, A. Browne, E. Clark, S. jha, and W. Marrero. An Improved Algorithm for the Evaluation of Fixpoint Expressions. In *Proc. of the Sixth International Conference on Computer Aided Verification, LNCS 818*, pages 338–350, 1994.

[50] D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Carnegie Mellon University, 1993.

[51] M. Matsuura, T. SASAO, J. T. Butler, and Y. Iguchi. Bi-Partition of Shared Binary Decision Diagrams. *IEICE Transaction fundamentals*, E85-A(12):2693–2700, 2002.

[52] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[53] M. R. Mercer, R. Kapur, and D. E. Ross. Functional approaches to generating orderings for efficient symbolic representations. In *Proceedings of the 29th Conference on Design Automation*, pages 624–627, Los Alamitos, CA, USA, June 1992. IEEE Computer Society Press.

[54] H. Miller and S. Katz. Saving Space by Fully Exploiting Invisible Transitions. In *Proc. of the 8th International Conference on Computer Aided Verification,LNCS*, pages 336–347, 1996.

[55] A. Narayan, A. Isles, J. Jain, R. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability Analysis Using Partitioned-ROBDDs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 388–393. IEEE Computer Society Press, June 1997.

[56] A. Narayan, J. Jain, M. Fujita, and A. L. Sangiovanni-Vincentelli. Partitioned-ROBDDs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 547–554. IEEE Computer Society Press, June 1996.

[57] Doron Peled. Combining Partial Order Reductions with On-The-Fly Model Checking. In *Proc. of the Sixth International Conference on Computer Aided Verification, LNCS 818*, pages 377–390, 1994.

[58] J.P. Quielle and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.

[59] R. K. Ranjan, J. V. Sanghavi, R. K. Brayton, and A. Sangiovanni-Vincentelli. Binary Decision Diagrams on Network of Workstations. In *IEEE International Conference on Computer Design*, October 1996.

[60] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 154–158, November 1995.

[61] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *Intl. Conf. on Computer Aided Design*, Santa Clara, Ca., November 1993.

[62] S. Panda and F. Somenzi. Who are the Variables in Your Neighbourhood. In *IEEE /ACM International Conference on CAD*, pages 74–77, San Jose, California, 1995. ACM/IEEE, IEEE Computer Society Press.

[63] S. Panda, F. Somenzi, and B.F. Plessier. Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams. In *IEEE /ACM International Conference on CAD*, pages 628–631, San Jose, California, November 1994. ACM/IEEE, IEEE Computer Society Press.

[64] Ulrich Stern and David L. Dill. Parallelizing the Murphy Verifier. In *Proc. of the 9th International Conference on Computer Aided Verification, LNCS 1254*, pages 256–267, 1997.

[65] C. Stirling and D. J. Walker. Local Model Checking in the Model Mu-Calculus. In *Proc. of the 1989 Int. Joint Conf. on Theory and Practice of Software Development*, 1989.

[66] A. Tarski. A lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific J. Math*, 5:285–309, 1955.

[67] H. Toutai, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincetelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 130–133. IEEE Computer Society Press, 1990.

[68] A. Valmari. A stubborn attack on the state explosion problem. In *Auto/Autograph*, New Brunswick, 1990.

[69] G. Winskel. Model Checking in the Modal μ-calculus. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, 1989.