Achieving High Speedups in Distributed Reachability Analysis through Asynchronous Computation

Research Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Sciences

Nili Ifergan

Submitted to the Senate of the Technion - Israel Institute of Technology

Tamuz, 5765 Haifa July 2005

The Research Thesis Was Done Under The Supervision of Prof. Orna Grumberg and Asoc. Prof. Assaf Schuster in the Department of Computer Science.

The generous financial help of the Technion is gratefully acknowledged.

Table of Contents

Table of Figures

1 Abstract

This thesis presents a novel BDD-based distributed algorithm for reachability analysis which is completely asynchronous. Previous BDD-based distributed schemes are synchronous: they consist of interleaved rounds of computation and communication, in which the fastest machine (or one which is lightly loaded) must wait for the slowest one at the end of each round.

We make two major contributions. First, the algorithm performs image computation and message transfer concurrently, employing non-blocking protocols in several layers of the communication and the computation infrastructures. As a result, regardless of the scale and type of the underlying platform, the maximal amount of resources can be utilized efficiently. Second, the algorithm incorporates an adaptive mechanism which splits the workload, taking into account the availability of free computational power. In this way, the computation can progress more quickly because, when more CPUs are available to join the computation, less work is assigned to each of them. Less load implies additional important benefits, such as better locality of reference, less overhead in compaction activities (such as reorder), and faster and better workload splitting.

We implemented the new approach by extending a symbolic model checker from Intel. The effectiveness of the resulting scheme is demonstrated on a number of large industrial designs as well as public benchmark circuits, all known to be hard for reachability analysis. Our results show that the asynchronous algorithm enables efficient utilization of higher levels of parallelism. High speedups are reported, up to an order of magnitude, when computing reachability for models with higher memory requirements than was previously possible.

2 Introduction

This work presents a novel BDD-based asynchronous distributed algorithm for reachability analysis. Our research focuses on obtaining high speedups while computing reachability for models with high memory requirements. We achieve this goal by designing an asynchronous algorithm which incorporates mechanisms to increase process utilization. The effectiveness of the algorithm is demonstrated on a number of large circuits, which show significant performance improvement.

Model checking [12] is a technique for verifying the correctness of finite-state systems with respect to temporal logic properties. Reachability analysis is a main component of model checking. Most temporal safety properties can easily be checked by reachability analysis [4]. Furthermore, liveness property checking can be efficiently translated into safety property checking [5].

Despite numerous improvements in model checking techniques over recent years, the so-called state explosion problem remains their main obstacle. In the case of industrial-scale models, time becomes a crucial issue as well. Existing BDD-based symbolic verification algorithms are typically limited by memory resources, while SAT-based verification algorithms are limited by time resources. Despite recent attempts to use SAT-based algorithms for full verification (pure SAT in [26, 10, 24, 18] and SAT with BDDs in [20, 19, 21]), it is still widely acknowledged that the strength of SAT-based algorithms lies primarily in falsification, while BDD-based model checking continues to be the de facto standard for verifying properties (see surveys in [28] and [6]). The goal of this work is verification, rather than falsification, of large systems. Therefore, we based our techniques on BDDs.

The use of distributed processing to increase the speedup and capacity of model checking has recently begun to generate interest [8, 29, 25, 3, 27, 22, 17]. Distributed algorithms that achieve these goals do so by exploiting the cumulative computational power and memory of a cluster of computers. In general, distributed model checking algorithms can be classified into two categories: explicit state representation based [29, 25, 3, 27] and symbolic (BDD-based) state representation based [22, 17]. Explicit algorithms use the fact that each state is manipulated separately in an attempt to divide the work evenly among processes; given a state, a hash-function identifies the process to which the state was assigned. The use of hash-functions is not applicable in symbolic algorithms which manipulate *sets* of states, represented as BDDs. In contrast to sets of explicit states, there is no direct correlation between the size of a BDD and the number of states it represents. Instead, the workload can be balanced by partitioning a BDD into two smaller BDDs (each representing a subset of the states) which are subsequently given to two different processes.

The symbolic work-efficient distributed synchronous algorithm presented in [17] is the algorithm that is closest to ours. In [17], as well as in our algorithm, processes (called workers) join and leave the computation dynamically. Each worker *owns* a part of the state space and is responsible for finding the reachable states in it. A worker splits its workload when its memory overflows, in which case it passes some of its owned states to a free worker.

Unlike the algorithm proposed in this work, the one in [17] works in synchronized iterations. At any iteration, each of the workers applies image computation and then waits for the others to complete the current iteration. Only then do all workers send the non-owned states discovered by them to their corresponding owners.

The method in [17] has several drawbacks. First, the synchronized iterations result in unnecessary and sometimes lengthy idle time for "fast" processes. Second, the synchronization phase is time-consuming, especially when the number of processes is high. Consequently, processes split as infrequently as possible in an attempt to reduce the overhead caused by synchronization. This leads to the third drawback: processes underutilize the given computational power, since available free processes are not used until there is absolutely no other choice but to join them in. These drawbacks make the algorithm insufficiently adaptive to the checked system and the underlying parallel environment. Furthermore, the combined effect of these drawbacks worsens with two factors: the size of the parallel environment and the presence of heterogeneous resources in it (as are commonly found today in non-dedicated grid and large-scale systems). These drawbacks limit the scalability of the algorithm and make it slow down substantially.

In order to exploit the full power of the parallel machinery and achieve scalability, it was necessary to design a new algorithm which is asynchronous in nature. We had to change the overall scheme to allow concurrency of computation and communication, to provide non-blocking protocols in several layers of the communication and the computation infrastructures, and to develop an asynchronous distributed termination detection scheme for a dynamic system in which processes join and leave the computation. In contrast to the approach presented in [17], the new algorithm does not synchronize the iterations among processes. Each process carries on the image computations at its own pace. The sending and receiving of states is carried out "in the background," with no coordination whatsoever. In this way, image computation and non-owned state exchange become *concurrent* operations.

Our algorithm is aimed at obtaining high speedup while fully utilizing the available computational power. To this end, when the number of free processes is relatively high the splitting rate is increased. This mechanism imposes *adap-*

tive early splitting to split a process even if its memory does not overflow. This approach ensures that free computational power will be utilized in full. In addition to using more processes, splitting the workload before memory overflows means that processes will handle smaller BDDs. This turned out to be a critical contribution to the speedup achieved by the new approach because a smaller BDD is easier to manipulate (improved locality of reference, faster image computation, faster and less frequent reorders, faster slicing, etc.).

In the asynchronous approach, when a process completes an iteration it carries on to the next one without waiting for the others. Consequently, splitting the workload with new processes is an efficient method for speeding up the computation since the overhead in adding more workers is negligible. However, this approach poses a huge challenge from the viewpoint of parallel software engineering. Given that the state space partition varies dynamically and that the communication is asynchronous, messages containing states may reach the wrong processes. By the time a message containing states is sent and received, the designated process may cease to own some or all of these states due to change of ownerships. Our algorithm overcomes this problem by incorporating a *distributed forwarding mechanism* that avoids synchronization but still assures that these states will eventually reach their owners. In addition, we developed a new method for opening messages containing packed BDDs which saves local buffer space and avoids redundant work: the mechanism ensures that only the relevant part of the BDD in the message is opened at every process visited by the message.

Distributed termination detection presents another challenge: although a certain process may reach a fixpoint, there may be states owned by this process that were discovered (or, are yet to be discovered) by others and are on their way to this process (in the form of BDDs packed in messages). The two-phase Dijkstra [13] termination detection algorithm is an efficient solution in such cases. However, we had to face yet another algorithmic complication that was not addressed by Dijkstra: the number of processes in the computation can vary dynamically and cannot be estimated or bounded in advance. We found no solution to this problem in the distributed computing literature. Thus, we had to develop the solution ourselves as an extension of the Dijkstra algorithm.

3 The Distributed Asynchronous Approach

We begin by describing the sequential symbolic (BDD-based) reachability algorithm. The pseudo-code is given in Figure 1. The set of reachable states is computed sequentially by applying Breadth-First Search(BFS) starting from the set of initial states S_0. The search is preformed by means of *image computa-*

tion which, given a set of states, computes a set containing their successors. In general, two sets of states have to be maintained during reachability analysis:

1) The set of reachable states discovered so far, called R. This set becomes the set of reachable states when the exploration ends.
2) The set of reached but not yet developed states, called N. These states are developed in each iteration by applying image computation on N.

```
Reachability(S_0)
1) R = N = S_0
2) while (N ≠ φ)
3)     N = Image(N)
4)     N = N \ R
5)     R = R ∪ N
6) return R
```

Fig. 1. Sequential Reachability Analysis

The distributed reachability algorithm relies on the notion of Boolean function slicing [1]. The state space is partitioned into *slices*, where each slice is *owned* by one process. A set, $w_1 \dots w_k$, of Boolean functions called *window functions* defines for each process the slice it owns. Windows are represented as BDDs. The set of window functions is complete and disjoint, that is, $\vee_{i=1}^{k} w_i = 1$ and $\forall i \neq j : w_i \wedge w_j = 0$, respectively. States that do not belong to the slice owned by a process are called *non-owned* states for this process.

As noted earlier, reachability analysis is usually carried out by means of a BFS exploration of the state space. Both the sequential algorithm (previously described in Figure 1) and the distributed synchronous algorithm (as in [22, 17]) use this technique: in iteration i, image computation is applied to the set of states, N, which are reachable in i steps (and no fewer than i steps) from the set of initial states. Thus, when iteration i completes, all the states which are reachable in at most $i + 1$ steps have already been discovered. While in a sequential search a single process develops the states in N, in a distributed search N is developed by a number of processes, according to the state space partition. In the latter, at the end of each iteration, the processes synchronize on a barrier ,i.e., wait until all processes complete the current iteration. Only then do the processes exchange their recently discovered non-owned states and continue to the next iteration.

However, reachability analysis need not be performed in such a manner. Note that reachability analysis would be correct even if, in iteration i, not all the states which are reachable in i steps are developed, as long as they will be developed in a future iteration. Thus, when a process completes iteration i, it does not have to wait until the other processes complete it. It can continue in

the image computation on the newly discovered states and receive owned states discovered by other processes at a later time. This is the key idea behind the asynchronous approach.

Like [22, 17], our algorithm uses two types of processes: workers and coordinators. Each active worker *owns* a slice of the state space and is responsible for discovering the reachable states within its slice. The algorithm 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. The algorithm works iteratively. At each iteration, each active worker computes an image and exchanges non-owned states, until a fixpoint is reached and termination is detected. During image computation, each worker computes the new set of states that can be reached in one step from the owned part of N of which it is aware. The new computed set contains owned as well as non-owned states. During the exchange operation, each worker sends the non-owned states to their corresponding owners. The novelty of our algorithm is that the iterations are not synchronized among workers. In addition, image computation and state exchange become concurrent operations.

Image computation and the receiving of owned states from other workers are critical points in which memory overflow may occur. In either case, the computation stops and the worker splits its ownership into k parts. One slice is left with the overflowed worker and $k-1$ parts are distributed to $k-1$ free workers. While in a distributed synchronous algorithm splitting occurs only when memory overflows, in our approach splitting is also used to obtain speedups. In particular, a worker will *split-on-timeout* if it has been working for t minutes without splitting and free workers are available. In addition, if the memory requirements of several workers decrease below a certain threshold, they merge their ownership and all but one become free.

The concurrency between image computation and state exchange is made possible by the asynchronous sending and receiving of states. Non-owned states are transformed into BDD messages. The sending of the BDD message is done in the background, by the operating system. As a result, a worker who sends a BDD message to a colleague is not blocked until the BDD message is actually sent or received. Similarly, a worker need not immediately process a BDD message that it receives. Received BDD messages are accumulated in a buffer $InBuff$. The worker can retrieve them whenever it chooses. We assume that the operating system enables sending and receiving of messages in such a non-blocking manner. The worker retrieves BDD messages from $InBuff$ during image computation, transforms them to BDDs, and stores them in a set called $OpenBuff$ until the current image operation is completed. To summarize, a

worker has to maintain three sets of states as well as one buffer during the distributed asynchronous reachability analysis:

1) A set of reachable states R.
2) A set of reachable states that were not yet developed N.
3) A buffer $InBuff$ containing unopened BDD messages received asynchronously by the operating system from other workers.
4) A temporary set of states $OpenBuff$. These states are obtained by opening BDD messages that were retrieved from $InBuff$.

In addition, our algorithm uses three coordinators: the `exch_coord`, which holds the current set of owned windows and is notified on every split or merge. The `exch_coord` is also responsible for termination detection; the `pool_mgr`, which keeps track of free workers; and the `small_coord`, which merges the work of underutilized workers. Following is an explanation of how we handle BDD messages. The algorithm itself will be explained in detail in Sections 5, 6 and 7.

4 Forwarding and Sending of BDD Messages

Workers often exchange non-owned states during reachability analysis. BDDs are translated into and from messages as described in [17]. This method reduces the size of the original BDD by 50% when forming a BDD message. Thus, BDD messages are transferred across the net quite efficiently. Moreover, recall that there is no exchange phase in which the processes send BDD messages all at once; messages are sent and received asynchronously during the computation. In addition, received BDD messages are opened during image computation and pending messages do not accumulate. As a result, the communication overhead is negligible and the memory required to store BDD messages that are waiting to be sent or opened is relatively small. These observations held in all the experiments we conducted.

As mentioned earlier, the asynchronous nature of our algorithm along with the fact that the state partition changes dynamically during the distributed computation imply that messages of non-owned states may reach the wrong worker (some or all of the states in the BDD message do not belong to the worker), and must be forwarded to their owner(s). Therefore, we attach a window to each BDD message. We refer to a BDD message as a pair $\langle T, w \rangle$, where T is the message containing the BDD and w is the attached window. Before worker P_i sends worker P_j a BDD message, it receives from the `exch_coord` the window w'_j which P_j owned when it last updated the `exch_coord` . This is the window P_i *assumes* P_j owns. As illustrated in Figure 2(a), when P_i sends a message to P_j

it attaches the window w'_j it assumes P_j owns. If P_j is required to forward this message to worker P_k with an assumed window w'_k, it will change the window to $w'_j \wedge w'_k$ before doing so (see Figure 2(b)). Note that our algorithm guarantees that all non-owned states will eventually reach their owners (see Section 8, Theorem 1). Thus, a BDD message may be forwarded more than once but only a finite number of times.

Fig. 2. (a)P_i sends P_j a BDD message with an assumed window w'_j (b) P_j forwards a BDD message to P_k with an assumed window w'_k

The Open_Buffer procedure, described in Figure 3, retrieves BDD messages from $InBuff$. Recall that those messages are received asynchronously into $InBuff$ by the operating system. When a worker retrieves BDD messages from $InBuff$, it requests and receives from the `exch_coord` the updated list of window functions owned by the workers. Next, it asynchronously forwards each BDD message to each worker whose window's intersection with the message window is non-empty. Then it opens the BDD message.

```
procedure Open_Buffer(w, OpenBuff)
1)  {⟨T, w'⟩} ← get waiting BDD messages from InBuff
2)  {⟨P_j, w_j⟩} ← receive from exch_coor all window functions
3)  foreach (⟨T, w'⟩)
4)    foreach ((j ≠ my_id) ∧ (w' ∩ w_j ≠ ∅))
5)      send ⟨T, w' ∩ w_j⟩ to P_j
6)    Res=Selective_Opening(T, w' ∩ w, Failed)
7)    if Failed = TRUE
8)      return BDD message to InBuff
9)      Split(R, w, N, OpenBuff)
10)   else OpenBuff = OpenBuff ∪ Res
```

Fig. 3. Open_Buffer algorithm performed by worker with id my_id and window w

We use a new method, called *selective opening*, for opening BDD messages. This method extracts from a BDD message only the states that are under a given window, without transforming the entire message to BDD form. Thus, redundant work is avoided. Before explaining how the selective opening is implemented, we first describe how a BDD is represented in a message form as suggested in [17].

BDD nodes represent a boolean function f recursively. The functions 0 and 1 are represented by special BDDs called ZERO and ONE, respectively. Other functions are represented by a node that contains a variable identification x, and two pointers `leftPtr` and `rightPtr`, that point to two other BDD nodes that represent $f_{\bar{x}}$ and f_x respectively. The function f is expresses based on the Shannon expression: $\bar{x}f_{\bar{x}} + xf_x$. A BDD in a message form is a sequence of records. Each record has four fields: an index for that record (a symbolic pointer), denoted as `Sid`. The variable id of the record denoted as `Xid`. An `Sid` for the left son, and an `Sid` for the right son. The index field indicates the record location in the message. The records ZERO and ONE have special indices.

A BDD message is obtained by traversing the nodes of a BDD f in Depth First Search (DFS) order. The corresponding records are created from the leaves upwards. Every time a new message record is created, a special index is incremented. This index serves as the `Sid` for that record. The relationship between the actual BDD node and the corresponding `Sid` is recorded in a dictionary. Since the records are created in a DFS order, every time a new record needs to be created the `Sid` for its left and right sons are already in the dictionary. The reverse operation, i.e., transforming a message back to a BDD form is performed as follows. The message records are traversed from start to end while creating the corresponding BDD nodes one by one using a Shanon expression. The relationship between the `Sid` and the corresponding pointer of the BDD node is again recorded in a dictionary. Because of the way the message is organized, every time a new node need to be created its function can be calculated using the pointers of its left and right sons which are already in the dictionary.

The idea in the selective opening method is to replace in the dictionary the node which represents the ONE leaf with the pointer to the BDD which represents the window. Thus, the resulting BDD is an intersection of the original BDD with the given window. In our algorithm, the selective opening method is used to extract only the owned stated, i.e., states which are under the window of the message intersected with the window of the worker. In particular, if the intersection between the window of the message and the window of the worker is empty, the message will not be opened at all. The worker holds the states extracted from the BDD message in $OpenBuff$. This buffer contains only owned states.

Though the selective opening method only extracts the required states, the operation may fail due to memory overflow. In this case, the worker splits its ownership and thereby reduces its workload. Note that BDD messages pending in $InBuff$ do not need special handling due to the split; the next time the worker calls the Open_Buffer procedure and retrieves a pending BDD message,

it forwards it according to the updated state partition given by the `exch_coord` and extracts the owned states according to its new window.

5 The Worker Algorithm

A high level description of the algorithm performed by a worker with ID my_id is shown in Figure 4. We will first describe each procedure in general, and then in detail.

```
procedure Reach_Task (R, w, N, OpenBuff)
1)  loop forever
2)    Bounded_Image(R, w, N, OpenBuff)
3)    Exchange(OpenBuff)
4)    if (Terminate() = TRUE)
5)      return R
6)    Collect_Small(R, w, N)
7)    if (w = ∅)
8)      send ⟨'to_pool', my_id⟩ to pool_mgr
9)      return to pool but keep forwarding BDD messages (by calling Open_Buffer)
```

Fig. 4. Pseudo-code for a worker in the asynchronous distributed reachability computation

During the Bounded_Image procedure, a worker computes the set of states that can be reached in one step from N, and stores the result in N. During the computation, the worker also calls the Open_Buffer procedure and extracts owned states into $OpenBuff$ (N and R will be updated with those states only in the Exchange procedure). If memory overflows during image computation or during the opening of a buffer, the worker splits w and updates N, R and $OpenBuff$ according to the new window. The same holds true if a split timeout occurs.

During the Exchange procedure the worker sends out the non-owned states ($N \setminus w$) to their assumed owners and updates N, R with new states accumulated in $OpenBuff$ (new states are states that do not appear in R).

If only a small amount of work remains, i.e., N and R are very small, the worker applies the Collect_Small procedure. The Collect_Small procedure merges the work of several workers into one task by merging their windows. As a result, one worker is assigned the unified ownership (merge as owner) and the rest become "free" ($w = \oslash$, merge as non-owners) and return to the pool of free workers.

After performing Collect_Small, the worker checks in the Reach_Task procedure if its window is empty and it needs to join the pool of free workers. The window of a worker can be empty if it merged as non-owner in the Collect_Small procedure, or if it joined the computation with an empty window (this will be

discussed later). Such workers, which participated in the computation once and then joined the pool of free workers, are called *freed*. The set of freed workers is contained in the set of free workers.

Freed workers may still receive misrouted BDD messages and thus need to forward them. For example, before worker Pi is freed, another worker may send it a message containing states that were owned by Pi. Should this message reach P_i after it was freed, P_i must then forward the message to the current owner(s) of these states. Methods for avoiding this situation will be discussed later. Note that if freed workers are required to forward BDD messages, they must participate in the termination algorithm. Following is a detailed description of each procedure.

```
procedure Bounded_Image(R, w, N, OpenBuff)
1)  Completed = FALSE
2)  while Completed = FALSE
3)     Bounded_Image_Step(R, w, N, Max, Failed, Completed)
4)     SplitOnTime ← (t < SplitTimer) ∧ (query ⟨pool_mgr, "free workers"⟩ = TRUE)
5)     if ((Failed = TRUE) ∨ (SplitOnTime = TRUE) )
6)        Split(R, w, N, OpenBuff)
7)    Open_Buffer(w, OpenBuff)
```

Fig. 5. Pseudo-code for a worker in the Bounded_Image procedure

The **Bounded_Image Procedure** is described in Figure 5. The image is computed by means of $Bounded\_Image\_Step$ operations, which are repeated until the computation is complete. This algorithm uses a *partitioned transition relation*. Each partition defines the transition for one variable. The conjunction of all partitions gives the transition of all variables. Each Bounded_Image_Step applies one more partition and adds it to the intermediate result. The steps are bounded in the sense that the Bounded_Image_Step procedure receives as an argument the maximal amount of memory that it may use. If it exceeds this limit, the procedure stops and $Failed$ becomes true.

The technique for computing an image using the partitioned transition relation was suggested by Burch et al. [9] and was used for the synchronous distributed algorithm in [17]. Using bounded steps to compute the image allows memory consumption to be monitored and the computation stopped if there is memory overflow. Also explained in [17] is how the partitioned transition relation helps to avoid repeating an overflowed computation from the beginning: each worker resumes the computation of its part of the image from the point at which it stopped and does not repeat the bounded steps that were completed in the overflowed worker.

Our asynchronous algorithm exploits the partitioned computation even further. During image computation, between each bounded step, we retrieve pending BDD messages from $InBuff$, forward them if necessary, and extract owned

states into $OpenBuff$. By doing so, we free the system buffer which contained the messages and produce asynchronous send operations, if forwarding is needed. Note that R and N are updated with $OpenBuff$ only after the current image computation is completed. In addition, during image computation, the worker can perform early split according to the progress of its computation and availability of free workers in the pool. We chose to implement the early splitting mechanism as follows: a worker will *split-on-timeout* according to the $SplitTimer$ value and the availability of free workers.

```
procedure Exchange(R, N, OpenBuff)
1) {⟨P_j, w_j⟩} ← receive from exch_coor all window functions
2) foreach (j ≠ my_id)
3)    send ⟨N ∩ w_j, w_j⟩ to P_j
4)    N = N \ w_j
6) N = N ∪ OpenBuff
7) N = N \ R
8) R = R ∪ N
9) OpenBuff = ⊘
```

Fig. 6. Pseudo-code for a worker in the Exchange procedure

The **Exchange Procedure** is described in Figure 6. First, the worker requests and receives from the `exch_coord` the list of window functions owned by the workers. Then it uses this list to asynchronously send recently discovered non-owned states to the other workers. The only guarantee we have is that the window list is complete, disjoint and that the window owned by the requesting worker is equal to its window in the list. Recall that the state space partition changes dynamically during the algorithm, and it is possible that this list is not up to date anymore. However, the forwarding mechanism guarantees that non-owned states will eventually reach their owners (Section 8, Theorem 1). After sending the non-owned states, the worker updates N, R with states accumulated in $OpenBuff$ and recalculates N, R.

Collect_Small Procedure. The pseudo-code for the Collect_Small procedure is described in Figure 7. If the worker has enough work, it exits immediately. Otherwise, it informs the `small_coord` the sizes of its N and R sets. Note that when the worker enters the Collect_Small procedure its $OpenBuff$ is empty ($OpenBuff$ was set to be empty in the Exchange procedure). The worker then waits for a reply from the `small_coord`. It can receive one of three commands and proceed as following:

– $\langle `end' \rangle$, means that the coordinator could not find any other worker to merge with this worker or that the worker is not small enough. Upon receiving this command the worker exits the Collect_Small procedure.

```
procedure Collect_Small (R, w, N)
1)  while |N| + |R| < Min
2)    send ⟨|N| + |R|⟩ to small_coord
3)    ⟨action⟩ ← receive from small_coord
4)    if action = ⟨'end'⟩
5)      return
6)    if action = ⟨'merge_as_non_owner', P_collegue⟩
7)      send ⟨R, w, N, InBuff⟩ to P_colleague
8)      R = N = w = InBuff = ∅
9)      ⟨release⟩ ← receive from exch_coord
10)     return
11)   if action = ⟨'merge_as_owner', P_colleague⟩
12)     ⟨R', w', N', InBuff'⟩ ← receive from P_colleague
13)     R = R ∪ R'; w = w ∪ w'; N = N ∪ N'; InBuff = InBuff ∪ InBuff'
14)     send ⟨'collect_small', w, my_id, colleague_id⟩ to exch_coord
15)     ⟨release⟩ ← receive from exch_coord
```

Fig. 7. Pseudo-code for a worker in the Collect_Small procedure

- $\langle$'$merge\_as\_non\_owner'$, $P_{colleague}\rangle$ commands the worker to deliver its ownership and owned states to a colleague worker $P_{colleague}$. The worker changes its window to be empty and waits for an acknowledgment from the `exch_coord` that the window was updated by $P_{colleague}$ and by the `exch_coord`. Only then it can exit the procedure.
- $\langle$'$merge\_as\_owner'$, $P_{colleague}\rangle$ commands it to take over the ownership and states of another worker $P_{colleague}$, and report the new ownership to `exch_coord`. The worker waits for an acknowledgment from the `exch_coord` that the window was updated before existing the procedure.

A worker which merges as non-owner is freed. As mentioned before, freed workers keep forwarding BDD messages. However, this can be avoided. A freed worker can stop forwarding BDD messages if all the messages that were sent to it have arrived and no new messages will be sent to it. First, if all the other workers have already requested and received a set of windows that does not include this freed worker, then no new messages will be sent to it. In addition, to ensure that all the already sent BDD messages have arrived, we can either bound the arrival time of a BDD message or run a termination-like algorithm. The termination algorithm will be discussed later.

Split Procedure. The pseudo-code for the Split procedure is described in Figure 8. This procedure starts by asking the `pool_mgr` for $k-1$ free workers. Split is called during the Bounded_Image procedure due to memory overflow that occurred in Bounded_Image_Step or while opening a BDD message. A worker may also call the Split procedure during the Bounded_Image procedure in case a split-on-timeout occurred and there are available free workers.

```
procedure Split(R, w, N, OpenBuff)
1)  if (N is big enough)
2)    {NewNw_i} ← Slice(N, k)
3)    If (max(R, OpenBuff) is big enough
4)      {NewW_i} ← Slice(max(R, OpenBuff), k)
5)    else
6)      NewW_1 ← w; ∀i ∈ {2...k} : {NewW_i} ← ∅
7)  else
8)    NewNw_1 ← w; ∀i ∈ {2...k} : {NewNw_i} ← ∅
9)    NewW_i ← Slice(max(R, OpenBuff), k)
10) foreach i ∈ {2...k}
11)   send ⟨R ∩ NewW_i, OpenBuff ∩ NewW_i, w ∩ NewW_i, N ∩ NewNw_i⟩ to P_i
12) R = R ∩ NewW_1; OpenBuff = OpenBuff ∩ NewW_1
13) w = w ∩ NewW_1; N = N ∩ NewNw_1
14) send ⟨'split', my_id, {(P_i, w ∩ NewW_i)|i ∈ {1...k}}⟩ to exch_coord
15) ⟨release⟩ ← receive from exch_coord
16) reset SplitTimer
```

Fig. 8. Pseudo-code for a worker in the Split procedure

In all the above cases, two sets of k window functions are computed. We compute two different sets of partitions. The partition of N (the set of windows $\{NewNw_i\}$) is done in an attempt to balance its size among the workers which will continue developing it. Notice that N may be containing intermediate results and not actual states since the image computation may be stopped and need to be continued. The partition of R and $OpenBuff$ (the set of windows $\{NewW_i\}$) is done in an attempt to balance the BDDs representing real states which were already discovered. Notice that the workers are essentially functioning as storage area by holding these states until termination is detected.

We compute the two sets of partitions as follows. If N is big enough, first, w is split into a set of window functions $\{NewNw_i\}$ according to N. Then, w is split into another set of window functions $\{NewW_i\}$. If either R or $OpenBuff$ is big enough we split w into $\{NewW_i\}$ according to the bigger one; Otherwise, if they are too small or empty, $\{NewW_i\}$ will be empty for all other workers except for $NewW_1$ which will be equal to w. In that case, the new colleagues which have an empty window are called *Helpers*. Helpers are simply assisting the overflowed worker with a single image computation. Once the image computation is completed, all the Helpers will send the states they produced to their owners and join the pool of free workers in the Reach_Task procedure. In our experiments, we observed that the creation of helpers is a common occurrence.

In case N is too small the worker keeps all of it (meaning that $NewW_1$ is equal to w and all other $\{NewW_i\}$ are empty) and split R and $OpenBuff$ according to the bigger one.

The two sets of k window functions are computed using the $Slice$ procedure. The $Slice$ procedure, when given a BDD and a splitting degree k, computes a set of k windows that partition the BDD to k parts. The slices do not have to be equally sized. Our algorithm can run in a heterogenic environment of computers, with different computational power and memory size. Therefore, if the `pool_mgr` is able to provide technical information about the free workers, the slicing algorithm can adjust the slices accordingly.

After computing the partitions of $R, OpenBuff$ and N, the splitting worker sends each worker its new window and its part of $R, OpenBuff$ and N. It also updates the `exch_coord` with the new windows the workers own and waits for an acknowledgement indicating that the `exch_coord` is updated. At the end of the procedure, the $SplitTimer$ variable is reset. This timer measures the time elapsed from the last split. It is used to determine whether a worker should split its workload (see Bounded_Image procedure, Figure 5)

6 Asynchronous Termination Detection

Our termination detection algorithm is an extension of the two-phase Dijkstra [13] termination detection algorithm. Dijkstra's algorithm assumes a fixed number of processes and synchronous communication. In our extension, the communication is asynchronous and processes may join and leave the computation.

The presented termination detection algorithm has two phases: the first phase during which the `exch_coord` receives $want\_term$ requests from all the active and freed workers, and the second phase, during which the `exch_coord` queries all the workers that participated in the previous phase as to whether they regret the termination. After receiving all responses, it decides whether to terminate or reset termination and notifies the workers of its decision. The `exch_coord` is discussed in Section 7.

Each worker detects termination locally and notifies the `exch_coord` when it wants to terminate. Upon receiving a regret query, the worker answers as to whether it regrets its request. The next message the worker will receive from the `exch_coord` will command it to terminate or reset termination. Note that the communication described above is asynchronous and does not block the workers.

The pseudo-code for the Terminate function performed by a worker is given in Figure 9. A worker enters this function after completing each image step and sending the non-owned states. It decides whether to terminate or not according to the return value of the function. The termination status of a worker can be one of the following: no_term, if it does not want to terminate; $want\_term$, if

it wants to terminate; $regret\_term$, if its status was $want\_term$ when it discovered that it still has work to do; $terminate$, if it should terminate. The initial termination status is no_term.

```
 function Terminate()
 1) if (TerminationStatus = 'terminate')
 2)    return (TRUE)
 3) if (N = ∅ ∧ 'there are no pending BDD messages in InBuff' ∧ 'all async' sends are complete')
 4)    if (TerminationStatus = 'no_term')
 5)       TerminationStatus = 'want_term'
 6)       send exch_coord ⟨TerminationStatus, my_id⟩
 7)       return FALSE
 8) else if (TerminationStatus = 'want_term')
 9)          TerminationStatus = 'regret_term'
10) ⟨action⟩ ← receive from exch_coord if any
11) if action = 'regret_termination_query'
12)    send ⟨'regret_status', TeminationStatus, my_id⟩
13) if action = 'reset_term'
14)    TerminationStatus = 'no_term'
15) if action = 'terminate'
16)    TerminationStatus = 'terminate'
17)    return TRUE
18) return FALSE
```

Fig. 9. Pseudo-code for a worker in the Terminate procedure

Upon entering the Terminate function the worker checks whether all of the following three conditions hold:

- It does not have any new states to develop ($N = \oslash$)
- it does not have any pending BDD messages in $InBuff$
- all its asynchronous send operations have been completed

We will clarify the last condition. If a worker receives a BDD message, the sender will not consider the send operation complete until it receives an acknowledgement from this worker. Without acknowledgement, there could be a BDD message that was sent but not received, and no worker would know of its existence. Note that the acknowledgement is sent and received asynchronously.

If the termination status is no_term and all conditions hold, the termination status is changed to $want\_term$. The worker will notify the `exch_coord` that it wants to terminate and exit the function (with return value false). If the termination status is $want\_term$ and one of the conditions does not hold, it may have more work to do. Thus, the termination status is changed to $regret\_term$. If the worker has a pending command from the `exch_coord`, it acts accordingly. It can be prompted to send its termination status, or else to set it to either no_term or $terminate$. The return value of the function is true, only if the worker changed its status to $terminate$.

7 The Coordinators

7.1 The `exch_coord`

The `exch_coord` holds the set of window functions and coordinates the termination detection algorithm. The set of window functions that the `exch_coord` holds is disjoint and complete; the `exch_coord` is notified on every split or merge of windows (a worker will not change its window until it receives a 'release' which indicates the `exch_coord` is updated with the change). Note that even though the `exch_coord` is updated on every window change, BDD message forwarding may still occur because of the asynchronous nature of the algorithm (see Section 4 for further details).

Figure 10 describes a high-level pseudo-code for the algorithm performed by the `exch_coord`. The `exch_coord` maintains a set of window functions Ws, where $Ws[P_i]$ holds the window owned by P_i. The `exch_coord` also maintains two lists: a list of active workers, $ActiveWL$, and a list of freed workers, $FreedWL$. The algorithm is initiated with one active worker, P_0, which owns the entire state space. Thus, $Ws[0]$ is initialized to one, the active workers list to $\{0\}$ and the freed workers list to $\oslash$. The `exch_coord` receives notifications from workers and acts accordingly: when a worker splits, the `exch_coord` updates the set of window functions, removes the new workers from the freed workers list (note that the freed workers list does not necessarily contain those workers) and adds them to the active workers list; when workers perform Collect_Small and join their ownership, the `exch_coord` updates the set of window functions and moves the freed worker from the active workers list to the freed workers list. The `exch_coord` is also responsible for termination detection.

```
   function Exch_Coord()
 1) Ws[0] = one
 2) ActiveWL = {0}; FreedWL = ⊘
 3) Loop-forever
 4)    ⟨cmd⟩ = receive from any worker
 5)    if cmd = ⟨'collect_small', P_id, w_id, P_i⟩
 6)       Ws[P_id] = w_id
 7)       ActiveWL = ActiveWL \ P_i
 8)       FreedWL = FreedWL ∪ P_i
 9)       send ⟨'release'⟩ to P_i and to P_id
10)    if cmd = ⟨'split', P_id, NewWs = {(p_i, w_i)}⟩
11)       foreach (p_i, w_i) ∈ NewWs
12)          Ws[P_i] = w_i
13)          ActiveWL = ActiveWL ∪ P_i
14)          FreedWL = FreedWL \ P_i
15)       send ⟨'release'⟩ to P_id
16)    TerminationDetection(cmd)
```

Fig. 10. High-level pseudo-code for the `exch_coord`

The exch_coord detects termination according to the TerminationDetection function (Figure 11(a)). The $TPhase$ variable indicates the termination phase and can have one of the following values: no_term, which means that no termination request has yet been received; $want\_term$, where the exch_coord collects termination requests; $regret\_term$, where the exch_coord collects regret termination responses.The initial value of $TPhase$ is no_term. In addition, the exch_coord holds the following three lists: the $WantTL$ list, which is used in the $want\_term$ phase and contains all the active and freed workers that have **not** sent a termination request; the $RegretQueryL$ list, which is used in the $regret\_term$ phase and contains all the workers that have **not** sent a regret response; and the $ResetOrTermL$ list, which contains all the workers that will be notified of the termination decision when the $regret\_term$ phase ends. The $CancelTerm$ variable holds the termination decision (true if the termination should be cancelled and false otherwise).

The phase changes are triggered by commands received from the workers. The exch_coord can receive one of four commands and proceed accordingly. The $want\_term$ phase begins upon receiving a $want\_term$ request. Then the $WantTL$ is assigned the value of all active and freed workers. During this phase, the exch_coord receives $want\_term$ requests from all the workers in this list. Each worker that sends a request is removed from the $WantTL$ list and added to the $RegretQueryL$. When the $WantTL$ list becomes empty, the $regret\_term$ phase begins and all the workers in the $RegretQueryL$ are sent a regret query (see MoveToRegretPhaseIfNeeded procedure, Figure 11(b)). During this phase, those workers send a response to the query (their regret status). Each worker that sends a response is removed from the list $RegretQueryL$ and added to the $ResetOrTermL$. Only when the $RegretQueryL$ becomes empty are the workers in the $ResetOrTermL$ sent the decision as to whether or not to terminate (see ResetOrTerminateIfNeeded procedure, Figure 11(c)). The exch_coord decides not to terminate if one of the workers regretted the termination or if split or merge occurred. In the latter case, the exch_coord also updates the appropriate lists, as follows.

As noted earlier, the termination algorithm supports a dynamic number of processes, i.e., workers may join and leave the computation. Workers join the computation only if a split occurs. In this case, the exch_coord receives a split notification, $\langle 'split', P_{id}, NewW_s = \{(p_i, w_i)\}\rangle$, which means that worker P_{id} split and several workers P_i joined the computation. The exch_coord acts according to the termination phase. If the termination phase is no_term, the termination status remains unchanged. Otherwise, the $CancelTerm$ is set to true. If the termination phase is $want\_term$, the exch_coord adds the new workers to the $WantTL$. However, if the termination phase is $regret\_term$,

```
procedure TerminationDetection(cmd)
 1) Initialization:
 2)   CancelTerm = FALSE
 3)   RegretQueryL = ⊘
 4)   TPhase = 'no_term'
 5) if cmd = ⟨'want_term', P_i⟩
 6)   if TPhase = 'no_term'
 7)     WantTL = ActiveWL ∪ FreedWL
 8)     TPhase = 'want_term'
 9)   if TPhase = 'regret_term' (P_i is a split colleague)
10)     send ⟨'regret_termination_query'⟩ to P_i
11)   RegretQueryL = RegretQueryL ∪ {P_i}
12)   MoveToRegretPhaseIfNeeded(P_i)
13) if cmd = ⟨'regret_status', stat, P_i⟩
14)   CancelTerm = CancelTerm ∨ (stat = 'regret_term')
15)   ResetOrTermL = ResetOrTermL ∪ {P_i}
16)   ResetOrTerminateIfNeeded(P_i)
17) if (cmd = ⟨'split', P_id, {(P_i, w_i)}⟩ ∧ TPhase ≠ 'no_term')
18)   CancelTerm = TRUE
19)   if TPhase = 'want_term'
20)     add new workers to to WantTL
21) if (cmd = ⟨'collect_small', P_id, w_id, P_i⟩ ∧ TPhase ≠ 'no_term')
22)   CancelTerm = TRUE
```

(a) TerminationDetection procedure

```
procedure MoveToRegretPhaseIfNeeded(P_i)
1) WantTL = WantTL \ {P_i}
2) if (WantTL = ⊘ ∧ TPhase = 'want_term')
3)   TPhase = 'regret_phase'
4)   ∀P_j ∈ RegretQueryL :
5)     send ⟨'regret_termination_query'⟩ to P_j
```

(b) MoveToRegretPhaseIfNeeded Auxiliary procedure

```
procedure ResetOrTerminateIfNeeded(P_i)
1) RegretQueryL = RegretQueryL \ {P_i}
2) if (RegretQueryL = ⊘ ∧ CancelTerm = FALSE)
3)   ∀P_j ∈ ResetOrTermL : send ⟨'terminate'⟩ to P_j
4) if (RegretQueryL = ⊘ ∧ CancelTerm = TRUE)
5)   ∀P_j ∈ ResetOrTermL : send ⟨'reset_term'⟩ to P_j
6)   ResetOrTermL = ⊘;  CancelTerm = FALSE
7)   TPhase = 'no_term'
```

(c) ResetOrTerminateIfNeeded Auxiliary procedure

Fig. 11. (a) Pseudo-code for the `exch_coord` in the TerminationDetection procedure (b) Auxiliary procedure MoveToRegretPhaseIfNeeded (c) Auxiliary procedure ResetOrTerminateIfNeeded

those workers are not added to any list. Nevertheless, it is possible that one of the new workers will send a $want\_term$ request while the termination phase is

$regret\_term$. In this case, the worker will be sent a regret query and be added to the $RegretQueryL$ (see Figure 11(a), lines 9,10).

A worker can leave the computation only if Collect_Small occurs. In this case, the `exch_coord` receives a collect_small notification, $\langle `collect\_small', P_{id}, w_{id}, P_i \rangle$, which means that worker P_i was freed. If the termination phase is no_term, the termination status remains unchanged. Otherwise, $CancelTerm$ is set to true. Since freed workers still participate in the termination algorithm, none of the list is updated. However, the presented termination algorithm can be easily modified to handle the case in which freed workers are not obliged to participate in the termination detection. In that case, the $FreedWL$ is redundant and the algorithm should be modified as follows. When the `exch_coord` receives a collect_small notification the freed worker should be removed from all the lists in which it appears (according to the current termination phase). Then, the `exch_coord` should check whether a phase move is needed (by invoking MoveToRegretPhaseIfNeeded or ResetOrTerminateIfNeeded).

7.2 The `small_coord`

The small_coord collects as many under-utilized workers as possible. It receives merge requests from under-utilized (small) workers. The small_coord stops a small worker for a predefined time; if timeout occurs and no other small worker has arrived in the meantime, it releases the worker. If a small worker arrives while another is waiting, it matches the two for merging.

7.3 The `pool_mgr`

The pool_mgr keeps track of free workers. During initialization it marks all workers as free, except for one. When a worker becomes free, it returns to the pool. When a worker splits, it sends the `pool_mgr` a request for $k-1$ free workers. The `pool_mgr` sends in reply a list of $k-1$ *ID*s of free workers, which are then removed from the pool. If the `pool_mgr` is asked for $k-1$ workers and there are not enough free workers in the pool, it stops the execution globally and announces "worker overflow." The `pool_mgr` also responds to queries as to whether there is high availability of free workers.

8 Proof of Correctness

Assumption 1 *Messages are not lost*

Definition 1. *[slice failure] If the Slice procedure is unable to partition a BDD to k parts, each smaller than the original BDD, then the execution stops globally*

Definition 2. *[workers overflow] If the* `pool_mgr` *is asked for k free workers and there are not enough free workers in the pool, then the execution stops globally*

From now on, when we say that the algorithm terminates we mean that the `exch_coord` detects termination and that there is no slice failure or workers overflow.

Lemma 1. *Every state is developed only once*

Proof: Immediate from the fact that the state space partition is kept disjoint and that each worker only develops the states that are under its window. □

Lemma 2. *The state space partition does not change forever*

Proof: The state space partition can change only when a worker splits or when workers merge their ownership in the Collect_Small procedure. Note that if the number of splits is finite, the number of mergers is finite as well. Therefore, it is enough to prove that the number of splits is finite.

By way of contradiction, suppose there is an infinite number of splits. After a workers splits, the first operation it performs is a Bounded_Image_Step. Then it can split again without completing the bounded image step due to memory overflow. It can also complete the bounded image step and split due to memory overflow while opening a buffer or split-on-timeout. Recall that the number of reachable states is finite and each state is developed only once. Therefore, from some point in time, no new states are being developed. Since each image computation operation consists of a finite number of Bounded_Image_Steps to be executed, we can conclude that there exists a Bounded_Image_Step which is being executed with fail indefinitely. Since a slice failure did not occur, the Slice procedure which is called on every split, partitioned each BDD to k parts (where k is the splitting degree), such that each slice is smaller than the original BDD. Workers split an infinite number of times and therefore the size of the BDD reduces infinitely. This is not possible. □

Lemma 3. *If a worker receives a BDD message after sending a negative regret response, then one of the other workers sent or will send a positive regret response.*

Proof: By way of contradiction. Assume worker P_i received a BDD message after sending a negative regret response. Further assume all the other workers sent or will send a negative regret response as well. Notice that at the point in time which P_i sent a negative regret response all other workers must have sent a $want\_term$ request to the `exch_coord` (otherwise the `exch_coord` would not send a regret query). As a result, we can conclude the following:

I All the messages each worker sent before sending a $want\_term$ request were received and acknowledged (if there are unacknowledged messages a worker will not send a $want\_term$ request).

II None of the workers sent or received a message before sending the (negative) regret response (otherwise it would send a positive regret response).

Therefore each worker which sent a BDD message after sending its $want\_term$ request did so after sending a negative regret response. Let us look at the first worker which sent a message after sending a $want\_term$ request. Since it was the first worker, it did not receive any message before sending the message and after sending the $want\_term$ request and had no pending messages. Moreover since it sent a $want\_term$ request its N and $OpenBuff$ sets were empty. According to the algorithm a worker which has an empty N, an empty $OpenBuff$ and does not have any pending messages will not send a message. This contradicts the hypothesis. □

Definition 3. *The termination detection algorithm performed by the* `exch_coord` *is said to have* ***initiated*** *when the* `exch_coord` *receives a '$want\_term$' request when its termination phase is 'no_term'*

Lemma 4. *If a split occurred after the termination detection algorithm initiated, the termination decision would be to reset termination*

Proof: Let us look at the first split (by a worker P_i) that occurred after the termination detection algorithm initiated and show that the termination decision would be to reset termination. There are several options:

- If the worker split before it sent its termination request, then the `exch_coord` will receive the split notification before its termination request and therefore would decide to cancel termination (line 18, TerminationDetection procedure, Figure 11(a)).
- If the worker split after it sent its termination request but before it sent its (negative) regret response, then the `exch_coord` will receive the split notification before its regret response and therefore would decide to cancel termination (line 18, TerminationDetection procedure, Figure 11(a)).
- If the worker split after it sent its negative regret response we will show that it is not possible that all other workers also sent a negative regret response. Since P_i sent a negative regret response its N was empty and there were no pending messages in $InBuff$. A split can only occur while opening a BDD message or while computing image. Therefore, the worker must have received a BDD message before splitting (and after sending a negative regret response). According to Lemma 3 one of the other workers will send or have already sent a positive regret response.

□

Lemma 5. *The algorithm does not terminate if there are BDD messages that were sent but not received*

Proof: By way of contradiction, assume that there is a message that was sent but not received, however, the exch_coord decides to terminate the algorithm. According to the termination algorithm performed by the exch_coord (Figure 11) all the active and freed workers must have sent the `exch_coord` a **negative** regret response. Furthermore, according to Lemma 4 no new workers joint the computation after the termination algorithm was initiated and therefore all the workers participate in the termination detection algorithm. Assume there is a message B sent by worker P_i to worker P_j which was not yet received by P_j.

We know that both P_i and P_j had sent a negative regret response to the `exch_coord`. We will show that according to the termination algorithm described in Figure 9, if P_i did not receive an acknowledgement for B, then it is not possible that the termination decision would be to terminate:

- If P_i sent the message before changing its status to '$want\_term'$ then the condition in line 3 would never hold since not all the asynchronous send operations will be complete. Therefore, P_i would never send a termination request. This contradicts the hypothesis.
- If P_i sent the message after changing its status to '$want\_term'$ and before sending the regret response then the next time it enters the Terminate procedure, the condition in line 8 would hold and P_i would have a positive regret response (regret its termination request). This also contradicts the hypothesis.
- If P_i sent the message after sending the negative regret response then it must have been as a result of a message B' it received from a worker P_k after it sent the regret response (since before that he had nothing to send). According to Lemma 3 one of the other workers will send or have already sent a positive regret response and therefore the termination decision would be not to terminate.

□

Theorem 1. *Every BDD message will eventually reach its owner*

Proof: According to Lemma 5 the algorithm will not terminate if there is a BDD messages that was sent but was not received. According to Assumption 1 messages are not lost in the communication level, i.e., a sent message will reach its destination with a finite period of time. Following Lemma 5 and our assumption, a BDD message will not reach its owner only if it is being forwarded forever.

Notice that every worker P_i which receives a BDD message $\langle T, w\rangle$ will only forward the part of the BDD message which does not belong to it, i.e., $\langle T, w \setminus w_i\rangle$. A BDD message will keep being forwarded to one or more workers as long as $w \setminus w_i \neq \oslash$. Moreover, if a BDD message $\langle T, w\rangle$ is being forwarded as $\langle T, w'\rangle$ its window can only reduce ($w' \subset w$, i.e., the set of states represented by w' is smaller than the set of states represented by w) or stay unchanged ($w' = w$). Since the window of the BDD message is finite (it represents a subset of the state space, which is finite), it can only reduce a finite number of times.

By the way of contradiction, suppose there is a BDD message B that from some point in time t_1 is being forwarded forever, i.e., its window never reduces. According to Lemma 2, the state space partition does not change forever. Therefore, there exist a time t_2 from which the `exch_coord` holds the final state space partition. Let us look at time t s.t. $t > t_1$ and $t > t_2$ in which B arrives to worker P_i. According to the Open_Buffer procedure described in Figure 3, P_i will receive the updated list of window functions from `exch_coord` and forward B to its owner P_j. Note that B can only be forwarded to one worker since the window of B does not reduce. Since P_j is the owner of B it will not forward B. This contradicts the hypothesis. □

Lemma 6. *The algorithm does not terminate if there are reachable states that were not developed*

Proof: We will prove that every state which is reachable from the set of initial states will be developed. Recall that the algorithm is initiated with one active worker which is given the set of initial states and owns the entire state space. A worker with a non empty N set will not send a termination request. However, since workers may split or merge their ownership and receive non-owned states during the computation we need to prove that termination is not detected before each worker updates its N with all the non-owned states which are under its window.

- Both during the Split and the Collect_Small procedures the state space partition is kept complete and disjoint. Every worker will receive its part of N and R and no states will be lost. Moreover, according to Lemma 4 if a worker split after the termination algorithm was initiated than termination will be cancelled. Similarly, if a worker merge after the termination algorithm was initiated than termination will ne cancelled. Moreover, in a merge operation states are not lost, the worker which mergers as owner updates its N and R sets with the ones of the other worker.
- According to Theorem 1, every BDD message will eventually reach its owner (before the algorithm terminates). Thus, each worker P_i will receive

all the BDD messages sent to it before the algorithm terminates. P_i will update N (and R) with the new states found in the opened BDD message (see Figure 6) and develop them. Moreover, assuming that P_i received all the BDD messages sent to it, termination will not be detected if N is not empty: according to the termination algorithm described in Figure 9, P_i will not change its termination status from 'no_term' to '$want\_term$' since the condition in line 3 would not hold. If N became non-empty only after P_i changed its status to '$want\_term$', then the condition in line 8 will hold and P_i will regret the termination.

□

Lemma 7. *Every state discovered by the algorithm is a reachable state*

Proof: Straightforward. The image computation starts from the set of initial states. Image computation is preformed only on states discovered in a previous image computation operation (by the current worker or by another worker). □

Theorem 2. *When termination is detected the set of states discovered by the algorithm is the set of reachable states*

Proof: Immediate from Lemma 6 and Lemma 7. □

Lemma 8. *Every worker with a $TerminationStatus$ which is equal to 'no_term' will eventually have a $TerminationStatus$ which is equal to '$want\_term$' so that it will never change to '$regret\_term$'.*

Proof: According to the termination algorithm described in Figure 9, line 3, it is enough to show that from some point in time $N = \oslash$, $InBuff$ is empty and all the asynchronous send operations were acknowledged.

According to Lemma 2, the state space partition does not change forever. Therefore, there is some point in time t_1 from which workers do not split or merge anymore.

Notice that since the number of reachable states is finite and each state is developed only once, the number of non-owned states discovered is also finite. Thus, the number of BDD messages each process sends is finite . Moreover, according to Theorem 1 every message is forwarded only a finite number of times. Therefore, from some point in time no messages will be received. Since each worker retrieves every BDD message which is in $InBuff$, there is some point in time $t_2 > t_1$ from which $InBuff$ will stay empty, for all workers. Since messages are not lost, there exists some point in time $t_3 > t_2$ from which all the previously sent messages are acknowledged, and no new messages are sent. Hence, from time t_3 and on, $InBuff$ is empty and all the asynchronous send operations are acknowledged, for all workers.

By way of contradiction, assume there exists a process P_i for which N never becomes empty. This implies that from time $t_4 > t_3$ worker P_i discovers an infinite number of states (since it does not split or merge and does not receive any BDD messages). This is not possible. □

Definition 4. *A set of workers is called "termination free" if all the workers in it are either in 'no_term' status or in '$want\_term'$ status but their termination request was not yet received by the* `exch_coord`

Lemma 9. *If all the active and freed workers are a termination free set before the* `exch_coord` *receives a $want\_term$ request of one of these workers, then the termination detection algorithm performed by the* `exch_coord` *will decide. In addition, the set of workers to which the* `exch_coord` *does not send its decision is a termination free set.*

Proof: We will prove that the $TPhase$ variable in the TerminationDetection procedure preformed by the `exch_coord` in Figure 11(a) will change to '$want\_term'$, then to $regret\_term$, and eventually the ResetOrTerminateIfNeeded procedure will be called when the $RegretQueryL$ is empty (i.e., the `exch_coord` will send its termination decision).

Upon receiving the '$want\_term'$ request the $TPhase$ is set to '$want\_term'$ and the $WantTL$ is set to the active workers list united with the freed workers list. First we show that this list will become empty.

All the workers in this list either sent a termination request which was not yet received, or are in 'no_term' status and thus according to Lemma 8 will eventually send a termination request. Therefore, according to line 1 in the MoveToRegretPhaseIfNeeded procedure all the workers will eventually be removed from this list. However, according to line 20 of the TerminationDetection procedure it is possible that a new worker will be added to this list. Since new workers that join the computation are in no_term status, according to Lemma 8 they will eventually send a termination request and thus be removed from the $WantTL$. To conclude, the $WantTL$ will become empty and all the workers which sent a termination request will be sent a regret query when the $TPhase$ is set to '$regret\_term'$, i.e., when the regret termination query phase begins. Notice that if there are active workers which are not in this list they are new workers that joint the computation and are either in no_term status or in $want\_term$ status but their termination request was not received while the `exch_coord` was in the $want\_term$ phase.

We next show that the regret phase ends, i.e., the $RegretQueryL$ becomes empty. Since all the workers in the regret query list were sent a regret query, and since messages are not lost, they will all respond to the query and thus

be removed from the $RegretQueryL$ according to line 16 of the Termination-Detection procedure. However, according to line 11 it is possible to receive a $want\_term$ request from a worker which was not in the $RegretQueryL$, this worker must be a new worker which joint the computation in a split operation since any other worker will not send another termination request. According to line 10, this worker will also be sent a regret query and thus it will send its regret response (and be removed from the $RegretQueryL$ list). To conclude, the $RegretQueryL$ will become empty and all the workers which responded to the regret query will be sent the termination decision. Notice that if there are active workers which were not in this list, they are new workers that joint the computation and are either in no_term status or in $want\_term$ status but their termination request was not received while the `exch_coord` was in the $regret\_term$ phase. □

Theorem 3. *The algorithm terminates*

Proof: According to the TerminationDetection procedure, the termination decision would be to terminate if all the workers do not regret the termination request and no split or merge were reported since the termination detection algorithm initiated. According to Lemma 2 there is a point in time t_1 from which there are no splits or merges. According to Lemma 9, since the algorithm is initiated with one worker in no_term status, there will be repeated initiations of the TerminationDetection procedure in all of which the `exch_coord` decides. In addition, when the termination algorithm initiates all the workers are a termination free set. According to Lemma 8 there is a time $t_2 > t_1$ from which all the workers do not regret their termination request. Let us look at an initiation of the termination algorithm which starts in time $t3 > t2$: since non of the workers will regret its decision and no splits or merges will occur, the termination decision would be to terminate. □

9 Experimental Results

We implemented our algorithm on top of Division [16], a generic platform for the study of distributed symbolic model checking which requires an external model checker. The algorithm was implemented in C and C++ and consisted of over 6,000 code lines. We used FORECAST [15] for this purpose. FORECAST is an industrial strength high-performance implementation of a BDD-based model checker developed at Intel, Haifa.

This section describes our preliminary experimental results on two ISCAS89 benchmarks (s1269, s3330) known to be hard for reachability analysis. Additional large-size examples are industrial designs taken from Intel. Our parallel testbed included a maximum of 28 PC machines, dual 2.4GHz Xeon processors

with 4GB memory. The communication between the nodes was via LAM MPI over fast Ethernet. We used daemon based communication, which allows true asynchronous message passing (i.e., the sending of messages progresses while the user's program is executing).

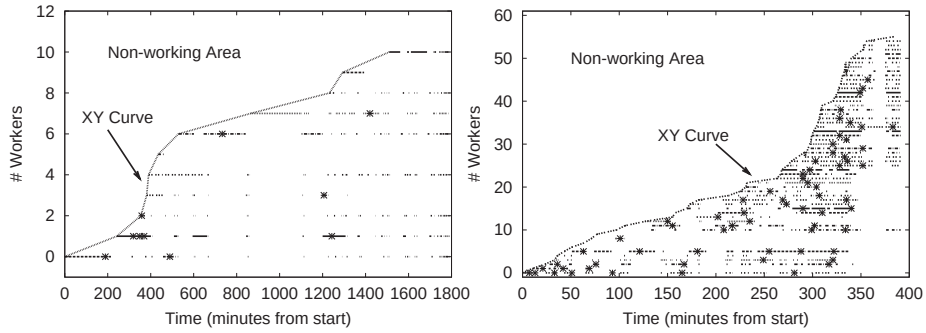
Our results are compared to FORECAST and to the work-efficient distributed synchronous implementation in [17]. The work-efficient implementation originally used NuSMV [11] as an external BDD-based model checker. For comparability, we replaced it with FORECAST. The work-efficient implementation which uses FORECAST will be referred to as FORECAST-D (Distributed FORECAST), and our prototype as FORECAST-AD (Asynchronous FORECAST-D). The splitting degree throughout the experiments was set to $k = 2$. We found this to be the optimal degree.

Circuit Name	# Vars	# Steps	Forecast		Forecast-D		Forecast-AD		
			Max. Step	Time(m)	Time(m)	# Workers	Time(m)	# Workers	Speedup (AD Vs. D)
s1269	55	9	9	45	50	12	15	6	3.3
s330	172	8	8	141	85	6	52	14	1.64
D_1	178	36	36	91	100	8	70	10	1.43
D_5	310	68	68	1112	897	5	150	18	5.98
D_6	328	94	94	81	101	5	76	3	1.3
Head_1_1	300	98	ovf(44)	-	9180	10	900	15	10.2
Head_2_0	276	85	ovf(44)	-	2784	4	390	55	7.14
Head_2_1	274	85	ovf(55)	-	1500	8	460	50	3.26
I1	138	139	ovf(102)	-	7178	18	2760	36	2.6

Fig. 12. A comparison between FORECAST, FORECAST-D and FORECAST-AD. If FORECAST was unable to complete an image step, we reported the overflowing step in parentheses. FORECAST-D and FORECAST-AD reached a fixpoint on all circuits. Column 10 shows the speedup when comparing FORECAST-AD and FORECAST-D run times.

Figure 12 clearly shows a significant speedup on all examples, up to an order of magnitude. When comparing FORECAST-D to FORECAST-AD, we were able to obtain a speedup even when the number of workers decreased. For instance, in the s1269 circuit, we obtained a speedup of 3.3 even though the number of workers decreased by a factor of 2. It can also be seen that the split-on-timeout mechanism in FORECAST-AD enables using more workers than in FORECAST-D. Using more workers clearly increases efficiency: for example in the Head_1_1 circuit, FORECAST-AD uses 1.5 times more workers, but the speedup is of an order of magnitude.

We analyzed worker utilization when using the split-on-timeout mechanism. Figure 13 provides utilization graphs for the Head_2_0 circuit, with this mechanism enabled and disabled. The Head_2_0 is a large circuit, difficult for reachability analysis. As can be seen in Figure 12, FORECAST is unable to reach a fixpoint on this circuit and overflows at step 44, while FORECAST-D requires over 46 hours to reach a fixpoint. Figure 13(a) clearly shows that when the split-



(a) Without Timeouts (b) With Timeouts

Fig. 13. FORECAST_AD worker utilization with and without the split-on-timeout mechanism in the Head_2_0 circuit. In each graph, the Y axis represents the worker's ID. The X axis represents the time(in minutes) from the beginning of the distributed computation. For each worker, each point indicates that it computed an image at that time; a sequence of points represents a time segment in which a worker computed an image; a sequence in which points do not appear represents a time segment in which a worker is idle (it does not have any new states to develop). An asterisk on the time line of a worker represents the point when it split. The XY curve connects times at which workers join the computation. This curve separates the ***working*** from the ***non-working*** area.

on-timeout mechanism is disabled, the workers are idle for much of the time. For instance, between 850 and 1100 minutes, only P_7 is working. This situation occurs when workers do not have any new states to develop and wait to receive new owned states. In this case, only when P_7 finds non-owned states and sends them to their corresponding owners are those workers utilized again. It is evident in Figure 13(b) that early splitting can significantly reduce such a phenomena. As can be seen, the phenomenon still exists, but on a much smaller scale, for instance between 360 and 380 minutes. In addition, when using split-on-timeouts, we are able to use more machines more quickly. In Figure 13(a) it takes 1600 minutes for 10 machines to come into use, whereas in Figure 13(b) this takes 70 minutes.

Circuit Name	# Vars	Forecast-AD				
		No Split on Timeouts (A)		Split on Timeouts (B)		Speedup
		Time(m)	# Workers	Time(m)	# Workers	(A Vs. B)
s330	172	120	8	52	14	2.3
D_5	310	617	14	150	18	4.1
Head_1_1	300	1140	4	900	15	1.3
Head_2_0	276	1793	11	390	55	4.6
Head_2_1	274	1200	5	460	50	2.6

Fig. 14. The split-on-timeout effect in FORECAST-AD. The "Speedup" column reports the speedup obtained when using the split-on-timeout mechanism.

Figure 13 also illustrates that when the number of workers increases, the relative size of the non-working area (the area above the XY curve) increases significantly. In the working area (the area below the XY curve), workers are dedicated to the distributed computation, whereas in non-working area, workers are in the pool and can be used for other computations. Thus the *effectiveness* of the mechanism, i.e, the relation between the speedup and the increase in the number of workers, should actually be measured with respect to the relative size of the working area. Figure 14 presents the speedup obtained on several circuits, when using the split-on-timeout mechanism.

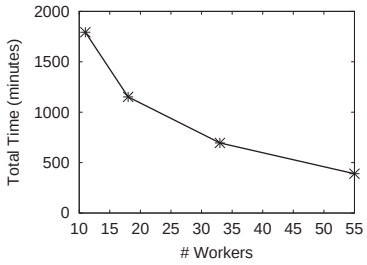


Fig. 15. The speedup obtained when increasing the number of workers on the Head_2_0 circuit (in FORECAST-AD). The X axis represents the time required to reach a fixpoint. The Y axis represents the maximal number of workers that participated in the computation. An asterisk on the (x, y) coordinate indicates that when the threshold of free workers is set to x, the reachability analysis ended after y minutes.

As can be seen in Figure 15, there is an almost linear correlation between the increase in computational power and the reduction in runtime on the Head_2_0 circuit. As the number of workers increases, the effectiveness decreases slightly. This can be explained by the fact that the relative size of the non-working area becomes larger as the number of workers increases (since we are not able to utilize free workers fast enough).

10 Related Work

Existing BDD-based symbolic verification algorithms are limited by memory resources, while SAT-based verification algorithms are limited by time resources. The strength of SAT-based verification algorithms lies primarily in falsification, while BDD-based symbolic model checking continues to be the de-facto standard for verifying properties [28].

Bounded Model Checking is a complementary technique to BDD-based model checking, introduced by Biere *e*t al. [7]. The basic idea in BMC is to search for counterexample in executions whose length is bounded by some integer k. Hence, when using BMC the user has to provide a bound on the number of cycles that should be explored, which implies that the method is incomplete if the

bound is not high enough. BMC has also the disadvantage of not being able to prove the absence of errors, in most realistic cases [6].

In recent years there have been several attempts to use SAT solvers for full verification, i.e., to perform unbounded reachability analysis (as oppose to the BMC technique in which reachability analysis is performed up to a predefined bound). These techniques can be classified into three categories. The first category includes techniques which have their root in BDD-based symbolic state space traversal. In these techniques the use of BDD have been partially or completely replaced with SAT solver. The second category consists of methods based on inductive reasoning. Inductive techniques are sound but usually incomplete. The third category of methods are iterative abstraction-refinement frameworks. In these methods, SAT-based BMC is used for abstraction or refinement. Other techniques are used for obtaining proofs on the smaller abstract models.

Pure SAT-Based methods for unbounded reachability analysis [26, 10, 24, 18] are based on All-SAT engines, i.e. a SAT solver is used to enumerate all solutions of a given formula. The All-SAT engine subsequently generates the set of newly reached states in a clausal form, until a fixpoint is reached. As opposed to BDD-based image computation which obtains all solutions at once, All-SAT engines enumerate solutions one by one; a problem occurs when the image set contains millions of solutions. In this case the enumeration and storage of solutions becomes impossible due to both time and space requirements.

The first purely SAT-based unbounded reachability algorithm was recently suggested by McMillan [26]. Similar approaches were suggested by [10, 24, 18]. These approaches differ from one another in the way state sets are represented, i.e, as CNF or as DNF formulas, and in the algorithms used to enlarge cubes, i.e., the method used to compress the formulas representing the set of reachable states.

In [20, 19, 21], the authors suggest a methodology which employs both SAT and BDD methods for image computation. In their approach, the transition relation is represented as a CNF formula, and state sets are represented using BDDs. BDD-based image computation is used to obtain all solutions bellow intermediate points in a SAT decision tree. By doing so, they avoid using an All-SAT engine which enumerates all solutions.

A different approach for combining SAT with decision diagrams was suggested by [2]. In the proposed algorithms for forward and backward reachability analysis the use of BDDs is replaced by SAT Solvers. *Reduced Boolean Circuits (RBCs)* are used to represent and perform manipulations on Boolean formulas. The fixpoint problem is formulated as a SAT problem by mapping the *RBCs* back to a formula. The produced formula is then given to an external SAT-solver.

There has been significate progress in the area of SAT-based full verification over the last decade. However, much remains to be done to make this technology pervasive in industrial designs. In particular, the experimental results presented for all the SAT-based techniques [10, 24, 18, 20, 19, 21, 2] were on small scale designs with very few latches on which a sequential BDD-based model checker can complete reachability analysis without any difficulty.

Distributed processing is another approach to increase the speedup and capacity of model checking. Distributed model checking algorithms can be classified into two categories: explicit state representation based [29, 25, 3] and symbolic state representation based [22, 17]. Stern and Dill present a way to parallelize the explicit Murφ verifier [29]. They present an asynchronous algorithm for reachability analysis. Their algorithm is intended to achieve speedup and indeed it does. 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. 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.

There are several differences between this algorithm and ours. The main difference is that this algorithm 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. Their 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. Another difference is that the number of processes in this algorithm is fixed while in our algorithm processes can join and leave the computation as needed.

In [25] the authors propose a way to distribute the SPIN [23] model checker. This approach is similar to the one proposed in the distributed Murφ verifier [29], however, it partitions the state space differently. Another method to distribute the SPIN [23] model checker was suggested in [3]. The distributed state space exploration in [25] is based on a nested depth-first search, whereas in [3] it is based on a non-nested depth-first search.

The closest work to ours is the symbolic work-efficient distributed synchronous algorithm for reachability analysis [17]. A comparison between our algorithm and the one in [17] as well as its drawbacks were discussed in the Introduction.

Distributed termination detection is a core problem of the theory of distributed computing. Our asynchronous termination detection algorithm is an extension of the two-phase Dijkstra [13, 14] termination detection algorithm. In our extension we handle the case where the number of workers varies dynamically (workers join and leave the computation) and the communication is

asynchronous whereas Dijkstra's algorithm requires synchronous communication and a fixed number of process. Dijkstra's algorithm is based on the *two-phase wave concept*, that is, a single process called the initiator sends out a probe, this probe is propagated to the other processes and then returned to the initiator. This wave algorithm is said to be *a centralized wave* since it is initiated by a single process.

Our termination detection algorithm can be initiated by a number of processes which send a termination request to the master. Only after all the processes send a termination request, does the master initiate a wave which queries as to whether anyone regrets its request. Our algorithm handles the asynchronous sending and receiving of BDD messages by an acknowledgement mechanism (as described in Section 6). The dynamic join and leave of processes requires the master to maintain a list of participating process for each of the termination phases, and update those lists when a process joins or leaves the computation. In addition, a join or leave of processes affects the termination decision and can prompt a phase move.

11 Conclusions and Future Work

This thesis presents a novel algorithm for distributed symbolic reachability analysis which is asynchronous in nature. We employed non-blocking protocols in several layers of the communication and the computation infrastructures: asynchronous sending and receiving of BDD messages (concurrency between image computation and state exchange), opening of messages between bounded image steps, a non-blocking distributed forwarding mechanism, non-synchronized iterations, and an asynchronous termination detection algorithm for a dynamic number of processes. Our dynamic approach tries to utilize contemporary non-dedicated large-scale computing platforms, such as Intel's Netbatch high-performance grid system, which controls all (tens of thousands) Intel servers around the world.

The experimental results show that our algorithm is able to compute reachability for models with high memory requirements while obtaining high speedups and utilizing the available computational power to its full extent.

Additional research should be conducted on better adapting the reorder mechanism to a distributed environment. One of the benefits of the distributed approach which we exploit is that each worker can perform reorder independently of other workers and thus find the best order for the BDD it holds. We did not elaborate on this matter since it is not the focus of the paper. Our adaptive early splitting approach not only better utilizes free workers but also results in processes handling smaller-sized BDDs, which are easier to manipulate. In particular, the reorders in small BDDs are faster and less frequent. Nevertheless,

the BDD package still spent a considerable time on reordering. We intend to explore the use of splitting as an alternative method for reordering.

References

1. M. Fujita A. Sangiovanni-Vincenteli A. A. Narayan, J. Jawahar. Partitioned-robdds. In *CAV*, pages 547–554, June 1996.
2. Parosh Aziz Abdulla, Per Bjesse, and Nikla Een. Symbolic Reachability Analysis Based on SAT-Solvers. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 411–425. Springer-Verlag, 2000.
3. J. Barnat, L. Brim, and J. Stribrna. Distributed LTL model-checking in SPIN. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 200–216. Springer-Verlag New York, Inc., 2001.
4. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *10th Computer Aided Verification*, pages 184–194, 1998.
5. A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *Proceedings of the 7th International ERCIM Workshop, FMICS02*, July 2002.
6. A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zue. *Bounded Model Checking*. Advances in Computers. Volume 58, Academic Press, 2003.
7. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.
8. V. A. Braberman, A. Olivero, and F. Schapachnik. Issues in distributed timed model checking: Building Zeus. *STTT*, 7(1):4–18, 2005.
9. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *Proceedings of Internation Conference on Very Large Integration, pp. 45-58, 1991*.
10. Pankaj Chauhan, Edmund M. Clarke, and Daniel Kroening. Using SAT based image computation for reachability analysis. Technical Report CMU-CS-03-151, Carnegie Mellon University, School of Computer Science, 2003.
11. 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.
12. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.
13. E. W. Dijkstra, W H. J. Feijen, and A. J.M Van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, pages 217–219, June 1983.
14. E. W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, pages 1–4, Aug. 1980.
15. R. Fraer, G. Kamhi, Z. Barukh, M.Y. Vardi, and L. Fix. Prioritized traversal: Efficient reachability analysis for verification and falsification. In *12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, Chicago, USA, July 2000.
16. 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.
17. O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. In *15th International Conference on Computer Aided Verification (CAV'03)*, LNCS, Bolder, Colorado, July 2003.
18. O. Grumberg, A. Schuster, and A. Yadgar. Memory Efficient All-Solutions SAT Solver and its Application for Reachability Analysis. In *Fifth International Conference on Formal methods in Computer-Aided Design (FMCAD'04)*, Austin, Texas, November 2004.

19. Aarti Gupta, Anubhav Gupta, Zijiang Yang, and Pranav Ashar. Dynamic detection and removal of inactive clauses in SAT with application in image computation. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 536–541. ACM Press, 2001.
20. Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. SAT-Based Image Computation with Application in Reachability Analysis. In *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, 2000.
21. Aarti Gupta, Zijiang Yang, Pranav Ashar, Lintao Zhang, and Sharad Malik. Partition-based decision heuristics for image computation using SAT and BDDs. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 286–292. IEEE Press, 2001.
22. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. A scalable parallel algorithm for reachability analysis of very large circuits. *Formal Methods in System Design*, 21(3):317 – 338, November 2002.
23. Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.
24. Hyeong-Ju Kang and In-Cheol Park. SAT-based unbounded symbolic model checking. In *DAC*, 2003.
25. Flavio Lerda and Riccardo Sisto. Distributed-Memory Model Checking with SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 22–39. Springer-Verlag, 1999.
26. Ken L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer Aided Verification*, 2002.
27. D. M. Nicol and G. Ciardo. Automated Parallelization of Discrete State-Space Generation. *J. Parallel Distrib. Comput.*, 47(2):153–167, 1997.
28. Mukul R. Prasad, Armin Biere, and Arti Gupta. A Survey of Recent Advances in SAT-Based Verification. To appear in STTT, 2005.
29. Ulrich Stern and David L. Dill. Parallelizing the murφ verifier. In *CAV*, pages 256–278, 1997.