Distributed Symbolic Model Checking for μ–calculus

Orna Grumberg[1], Tamir Heyman[1,2], and Assaf Schuster[1]

[1] Computer Science Department, Technion, Haifa, Israel
[2] IBM Haifa Research Laboratories, Haifa, Israel

Abstract. In this paper we propose a distributed symbolic algorithm for model checking of propositional μ–calculus formulas. μ-calculus is a powerful formalism and many problems like (fair) CTL and LTL model checking can be solved using the μ–calculus model checking. Previous works on distributed symbolic model checking were restricted to reachability analysis and safety properties. This work thus signi£cantly extends the scope of properties that can be veri£ed 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 thus can be implemented on huge distributed clusters of computing nodes. In this way, the memory modules of the computing nodes collaborate to create a very large store, thus enables 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.

1 Introduction

In the early 1980's, model checking procedures have been suggested [5, 15, 12], which could handle systems with few thousands states. In the early 1990's, symbolic model checking methods have been introduced. These methods, based on Binary Decision Diagrams (BDDs) [2], could verify systems with 10^{20} states and more [4]. This progress has made model checking applicable to industrial designs of medium size. Signi£cant efforts have been made since to £ght the *state explosion problem*. But the need in verifying larger systems grows faster than the capacity of any newly developed method.

Recently, a new promising method for increasing the memory capacity was introduced. The method uses the collective pool of memory modules in a network of processes. In [10], distributed symbolic reachability analysis has been performed, for £nding the set of all states reachable from the initial states. In [1], a distributed symbolic on-the-¤y algorithm has been applied in order to model check properties written as regular expression. Experimental results show that distributed methods can achieve an average memory scale-up of 300 on 500 processes. Consequently, they £nd errors that were not found by sequential tools.

This paper extends the scope of properties that can be veri£ed for large designs, by presenting a distributed symbolic model checking for the μ-calculus. The *μ-calculus* is a powerful formalism for expressing properties of transition systems using £xpoint operators. Many veri£cation procedures can be solved by translating them into μ–calculus model checking[4]. Such problems 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 [9, 16, 18, 7, 13]. In this work we parallelize a simple sequential algorithm, as presented in [6]. The 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. At no point in the distributed algorithm a whole set is stored by a single process.

Distributed computation might be subtle 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 to carry this operation locally by each process. 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 it is (distributively) decided 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 £nd the set of all predecessor 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 owner.

Keeping the memory requirements low is done 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, commonly 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, £rst the two sets should be re-sliced according to the same partition. Only then the processes apply conjunction to their individual slices.

Distributing the sets of states is only one facet of the problem. The transition relation also strongly in¤uences 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 £nal result is of tractable size, its intermediate results might explode the memory. We propose a scalable distributed method for the pre-image computation, including partitioning of the transition relation.

2 Preliminaries

2.1 The Propositional μ–Calculus

Below we de£ne the propositional μ–calculus [11]. 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 de£ned 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 £nite set of states; $R \subseteq St \times St$ is the transition relation, and $L : St \rightarrow 2^{AP}$ is the labelling function that maps each state to the set of atomic propositions true in that state.

In order to de£ne 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 de£ned recursively as follows (where M is omitted when clear from the context).

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

Tarski [17] showed that least and greatest £xpoints always exist if τ is monotone. If τ is also continuous, then the least and greatest £xpoints of τ can be computed by $\cup_i \tau^i(False)$ and $\cap_i \tau^i(True)$, respectively. In [6] it is shown that if M is £nite then any monotone τ is also continuous.

In this paper we consider only monotone formulas. Since we consider only £nite transition systems, they are also continuous. The function `fixpt` on the right-hand-side of Figure 1 describes an algorithm for computing the least or greatest £xpoint, depending on the initialization of Q_{val}. If the parameter I is $False$ then the least £x-point is computed. Otherwise, if $I = True$, then the greatest £xpoint 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 £nds the set of states in M that satisfy f. Figure 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 `ev` function.

2.2 Elements of Distributed Symbolic Model Checking

Our distributed algorithm involves several basic elements that were developed in [10]. For completeness, we brie¤y mention these elements in this subsection.

intermediate results, are represented by BDDs. the algorithm execution, the sets of states obtained are partitioned among the processes. A set of *window functions* is used to de£ne the partitioning, determining the slice that is stored (we say: *owned*) by each process.

```
1  function ev(f, e)
2   case
3     f = p:      res = {s | p ∈ L(s)}
4     f = Q:      res = e(Q)
5     f = ¬g:     res = ¬ev(g, e)
6     f = g1∨g2:  res = ev(g1, e)∨ev(g2, e)
7     f = g1∧g2:  res = ev(g1, e)∧ev(g2, e)
8     f = EXg:    res = {s | ∃t[sRt ∧ t ∈ev(g, e)]}
9     f = μQ.g:   res = fixpt(Q, g, e, False)
10    f = νQ.g:   res = fixpt(Q, g, e, True)
11  endcase
12  return(res)
13 end function
```

```
1 function fixpt(Q, g, e, I)
2  Q_val = I
3  repeat
4    Q_old = Q_val
5    Q_val = ev(g, e[Q ← Q_old])
6  until (Q_val = Q_old)
7  return Q_val
8  end function
```

Fig. 1. Pseudo–code for sequential μ–calculus model checking

De£nition 1. *[Complete set of window functions] 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 for every $1 \leq i, j \leq k$, $i \neq j$, $W_i \wedge W_j = 0$ and $\bigvee_{i=1}^{k} W_i = 1$.*

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

We use the *slicing algorithm*, as described in [10] to get a set of window functions. The objective of this algorithm is to distribute a given set evenly among the nodes. Its input is a set of states, and its output is a set of window functions. These functions slices the input set into subsets that are approximately of the same size.

Maintaining balanced memory requirement by the processes is done by means of a *memory balance* algorithm, as described in [10]. When this algorithm is applied at an already sliced set of states, a new partitioning is computed, one that will balance the size of the subsets. The new partitioning is computed by pairing large slice of the set with small one and re-slicing their union. This algorithm de£nes a new set of window functions that will be used to produce further intermediate results.

During the memory balance algorithm, as well as during other parts of the distributed model checking algorithm, BDDs are shipped between the processes. The communication uses a compact and universal BDD representation, as described in [10]. Different variable order is allowed in the different processes.

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 2.1. 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 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 imbalance sets of states which are produced during the computation. Distributed termination algorithm is used to announce global termination. In the rest of this section, we describe elements used by this algorithm.

3.1 Switching From Initial to Distributed Computation

When the initial phase terminates, several subformulas have already been evaluated and the sets of states associated with them are 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. 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, when partitioning the sets, there are two factors involved: the required storage space for the sets, and the space needed to manipulate them. In order to keep the £rst 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, observe that the memory used in performing an operation is proportional to the size of the set it is applied to, thus the part of each set in each process should be small.

In model checking, the most acute peaks in memory requirement usually occur while operations are 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. It is important to note that as a result the slicing criterion may differ for different sets.

We use a slicing algorithm[10] described generally in Section 2.2. In order to slice all the sets that where already evaluated at the point of phase switching, slicing is applied to each one of them.

While the slicing algorithm works it 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 $InitSet$ that for each formula f, holds the subset of the set of states satisfying f and owned by this process. Formally, for each process id, $InitSet(f) = f \wedge W_{id}$. The distributed phase will start by sending the tables $InitEval$ and $InitSet$ and the list of slices W_i to all the processes.

3.2 The Distributed Phase

The distributed version of the model checking algorithm for the μ–calculus is given in Figure 2. While the sequential algorithm £nds the set of states in a given model that satisfy a formula of the μ–calculus logic, in the distributed algorithm each process £nds 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 £nds the set of states $\mathtt{ev}(f, e) \wedge W_{id}$.

In fact, a weaker property is required in order to guarantee the correctness of the algorithm. We only need to know that when evaluating a formula f, every state satisfying

f is collected by at least one of the processes. For ef£ciency, however, we require in addition that every state is collected by exactly one process.

Given a formula f the algorithm £rst 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 signi£cantly larger than the others, then the effectiveness of the distributed system is destroyed. To avoid this situation, whenever a new set of states is created a memory balance procedure is invoked to keep a balanced memory requirement by 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.

Each process in the distributed algorithm evaluates each subformula f as follow (see Figure 2):

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 are evaluated, the environment must have a value for Q (computed in a previous step).

A subformula of the form $\neg g$: evaluated by £rst 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 all other processes of 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 actually send each other only states that owned by the receiver. This reduces communication.

A subformula of the form $g_1 \vee g_2$: evaluated by £rst 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. 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 £rst 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 the overall communication is reduced to only two rounds.

A subformula of the form $\mathbf{EX}g$: evaluated by £rst 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. 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 £xpoint and greatest £xpoint, respectively): evaluated using a special function `fixpt` that iterates until a £xpoint is found. The computations for the formulas differ only in the initialization which is $False$ for $\mu Q.g$ and the current window functions for $\nu Q.g$.

```
 1 function pev(f, e)
 2   case
 3     InitEval(f) :   return(InitSet(f))
 4     f = p :         res = {s | p ∈ L(s)} ∧ W_id
 5     f = Q :         return (e(Q))
 6     f = ¬g :        res =exchnot(pev(g, e))
 7     f = g1 ∨ g2 :   res =exch(pev(g1, e)∨pev(g2, e))
 8     f = g1 ∧ g2 :   res1 =pev(g1, e)  res2 =pev(g2, e)
 9                     res =exch(res1)∧exch(res2)
10     f =EXg :        res =exch({s | ∃t[sRt ∧ t ∈pev(g, e)]})
11     f = μQ.g :      res =fixpt(Q, g, e, False)
12     f = νQ.g :      res =fixpt(Q, g, e, W_id)
13   endcase
14   ldBlnc(res) /* balances W; updates res accordingly */
15   return(res)
16 end function
```

```
1 function fixpt(Q, g, e, init)
2    Q_val = init
3    repeat
4      Q_old = Q_val
5      Q_val = pev(g, e[Q ← Q_old])
6    until (parterm(exch(Q_val) =exch(Q_old)))
7    return Q_val
8 end function
```

```
1 function exch(S)
2    res = S ∧ W_id
3    for each process i ≠ id
4      sendto(i, S ∧ W_i)
5    for each process i ≠ id
6      res = res∨ receivefrom(i)
7    return res
8 end function
```

```
1 function exchnot(S)
2 res = (¬S) ∧ W_id
3 for each process i ≠ id
4   sendto(i, (¬S) ∧ W_i)
5 for each process i ≠ id
6   res = res∧ receivefrom(i)
7 return res
8 end function
```

Fig. 2. Pseudo–code for a process id in the distributed model checking

3.3 Sources of Scalability

The ef£ciency of a parallelization approach is determined by the ratio between computation complexity, normalized by computation speed, and communication complexity,

normalized by communication bandwidth. In our parallel model checking algorithm, this ratio (excluding normalization, which is dependent on the underlying platform) can be estimated by observing that peak memory requirement for a single μ-calculus operation of a symbolic computation is a lower bound on the computation complexity of this operation. On average, in the distributed setup, the size of BDD structures that are sent (received) by a process is a fraction of its BDD manager size at the end of the operation (after memory balance). Thus, roughly speaking, for a single operation computation, peak memory utilization bounds from below the computation complexity, whereas the size of the BDD manager represents the communication complexity. General wisdom holds that the ratio between peak and manager sizes reaches 2 or 3 orders of magnitudes, which, for current computing platforms is suf£cient to keep the processor and communication subsystems equally busy. Indeed, our experiments with previous parallel symbolic computations in a distributed setup consisting of a slow network con£rmed the ef£ciency of this approach [10, 1].

Scalability of a parallel system is the ability to include more processes in order to handle larger inputs of higher complexity. Linear scalability is used to describe a parallel system that does not loose performance while scaling up. Recall that the volume of communication performed by a single process in our algorithm during a single operation, may be represented on average by a fraction of its BDD manager size at the end of the operation. Also, the corresponding peak memory that is used by the process during that operation is bounded by the size of its memory module (otherwise the operation over¤ows). By the above mentioned ratio between the sizes of the peak and the BDD manager, the manager size (in between operations) is also bounded. Thus, using our effective slicing procedure, the local BDD manager size does not increase when the system is scaled up globally in order to check larger models using more processes. Thus, the ratio between computation and communication for each process does not vary substantially when the system scales up, implying almost linear scalability of our distributed model checking algorithm.

Finally, we note that a higher ratio of peak to BDD manager sizes, which may result from a larger transition system in larger models, will enhance the scalability of our parallel approach. Since the size of memory module limits the peak size, a higher ratio implies smaller BDD manager, which, in turn, implies lower communication volumes. Thus, when the checked models grow, the method may exhibit super-linear scalability.

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 satisfying this formula. 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 de£nition.

De£nition 2. *[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 it owns. The set of window functions do not change.
Let f be a μ–calculus formula, e_{id} be the environment in process id. $\texttt{pev}_{id}(f, e_{id})$ denotes the set of states returned by procedure `pev`, when run by process id on f and e_{id}.
Theorem 1 de£nes the relationship between the outputs of the sequential and the distributed algorithms.

Theorem 1 (Correctness). *Let f be a μ–calculus formula, e be a well partitioned environment by $e_1, \ldots e_k$, e' be the environment when $\texttt{ev}(f, e)$ terminates and for all $i = 1, \ldots, k$, e'_i be the environment when $\texttt{pev}_i(f, e_i)$ terminates. Then, e' is well partitioned by $e'_1, \ldots e'_k$ and $\texttt{ev}(f, e) = \bigvee_{i=1}^{k} \texttt{pev}_i(f, e_i)$.*

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' is well partitioned by $e'_1, \ldots e'_k$. Due to lack of space we only consider several of the more interesting cases.
Base: $f = p$ for $p \in AP$ – Immediate.
Induction:
$\underline{f = Q}$, where $Q \in VAR$ is a relational variable: $\bigvee_{i=1}^{k} \texttt{pev}_i(Q, e_i) = \bigvee_{i=1}^{k} e_i(Q)$. Since e is well partitioned, $e(Q) = \bigvee_{i=1}^{k} e_i(Q)$, which is equal to $\texttt{ev}(f, e)$.
$\underline{f = \neg g}$: $\texttt{pev}_{id}(\neg g, e_{id})$ £rst applies $\texttt{pev}_{id}(g, e_{id})$ which results with S_{id}. It then runs the procedure $\texttt{exchnot}(S_{id})$ that 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 `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 $S_i = \texttt{pev}_i(g, e_i)$, $\neg \bigvee_{j=1}^{k} S_j = \neg \bigvee_{j=1}^{k} \texttt{pev}_i(g, e_i)$, which by the induction hypothesis is identical to $\neg\ \texttt{ev}(g, e)$. This, in turn, is identical to $\texttt{ev}(\neg g, e)$. Applying `ldBlnc` at the end of `pev`, repartitions the subsets between the processes, however, their disjunction remains the same. Thus, $\texttt{ev}(\neg g, e) = \bigvee_{i=1}^{k} \texttt{pev}_i(\neg g, e_i)$.
$\underline{f = g_1 \vee g_2}$: $\texttt{pev}_{id}(f, e_{id})$ £rst computes $\texttt{pev}_{id}(g_1, e_{id}) \vee \texttt{pev}_{id}(g_2, e_{id})$. At the end of this computation, the global set is:

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

By the induction hypothesis, this is identical to $\texttt{ev}(g_1, e) \vee \texttt{ev}(g_2, e)$ which is identical to $\texttt{ev}(g_1 \vee g_2, e)$. Applying the procedures `exch` and `ldBlnc` change the partition of the sets among the processes, but not the global set.
$f = \mu Q.g$, a least £xpoint formula: As in previous cases, we would like to prove that $\bigvee_{i=1}^{k} \texttt{pev}_i(\mu Q.g, e_i) = \texttt{ev}(\mu Q.g, e)$. 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. 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 parameters are $True$, then `parterm` returns $True$ to all processes. Otherwise, it returns $False$ to all processes.

Lemma 1. *Let Q^j, be the value of Q_{val} in iteration j of the sequential £xpoint algorithm. Similarly, let Q^j_{id} be the value of Q_{val} in iteration j of the distributed £xpoint algorithm in process id. Q^0 is the initialization of the sequential algorithm; Q^0_{id} is the initialization of the distributed algorithm. Then, • In every iteration, e is well partitioned by $e_1, \ldots, e_k$. • For every j: $Q^j = \bigvee_{i=1}^{k} Q^j_i$. • If the sequential `fixpt` algorithm terminates after i_0 iterations then so does the distributed £xpoint algorithm.*

Proof: We prove the lemma by induction on the number j of iterations in the loop of the sequential function `fixpt`.
Base: $j = 0$:
• At iteration 0, e is well partitioned based on the induction hypothesis of Theorem 1.
• In case $f = \mu Q.g$, the initialization of the sequential algorithm, as well as the distributed algorithm is $False$. Hence, $Q^0 = Q^0_{id} = False$ which implies $Q^0 = \bigvee_{i=1}^{k} Q^0_i$.
• Both algorithms perform at least one iteration, so they do not terminate at iteration 0.
Induction: Assume Lemma 1 holds for iteration j. We prove it for iteration $j + 1$.
• 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. In the sequential algorithm e may be changed in two ways: $e(Q)$ is assigned a new value Q^j, or a recursive call to `ev` may change e. Similarly, in the distributed algorithm two changes may occur: $e_{id}(Q)$ is assigned a new value Q^j_{id}, or a recursive call to $\texttt{pev}_{id}$ may change e_{id}.

By the induction hypothesis of Lemma 1 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 $j + 1$, `ev` in now invoked with an environment that is well partitioned by the environments $\texttt{pev}_{id}$ is invoked with. The induction hypothesis of Theorem 1 therefore guarantees that e' is well partitioned by $e'_1, \ldots, e'_k$.
• $Q^{j+1} = \texttt{ev}(g, e[Q \leftarrow Q^j])$ (line 5 of the sequential algorithm) and $Q^{j+1}_{id} = \texttt{pev}_{id}(g, e[Q \leftarrow Q^j_{id}])$ (line 5 of the distributed algorithm).

By the first bullet above, $e[Q \leftarrow Q^j]$ is well partitioned. Thus, the induction hypothesis of Theorem 1 is applicable and implies that $\texttt{ev}(g, e[Q \leftarrow Q^j]) = \bigvee_{i=1}^{k} \texttt{pev}_i(g, e[Q \leftarrow Q_i^j])$. Hence, $Q^{j+1} = \bigvee_{i=1}^{k} Q_i^{j+1}$.

• 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, $\texttt{exch}(Q_{id}^j) = \texttt{exch}(Q_{id}^{j+1})$ and therefore `parterm` returns $True$ to all processes.

Let $W_1, \ldots, W_k$ be the current window functions. By the second bullet above, $Q^j = \bigvee_{i=1}^{k} Q_i^j$ and $Q^{j+1} = \bigvee_{i=1}^{k} Q_i^{j+1}$.

$$\forall id[\texttt{exch}(Q_{id}^j) = \texttt{exch}(Q_{id}^{j+1})] \;\Leftrightarrow\; \forall id[\bigvee_{i=1}^{k} Q_i^j \wedge W_{id} \;=\; \bigvee_{i=1}^{k} Q_i^{j+1} \wedge W_{id}] \;\Leftrightarrow$$

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

The last equality is implied by the previous one since the window functions are complete. This complete the proof of the lemma and also the proof of the theorem. Q.E.D.

The above theorem can be extended to state that when all procedures $\texttt{pev}_{id}(f, e_{id})$ terminate, the subsets owned by each of the processes are disjoint. This is important in order to avoid duplication of work. However, it is not necessary for the correctness of the model checking algorithm.

5 Scalable Distributed Pre-image Computation

The main goal of our distributed algorithm is to reduce the memory requirement. 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 **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 with the sizes of the transition relation R and the set S. 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 pre-image, but not of R. In order to make our method scalable for very large models, we need to reduce the size of the transition relation as well.

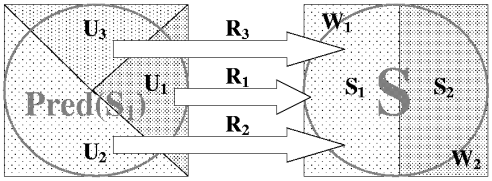
The transition relation consists of pairs of states. We distinguish between the source states and the target states by refer to the latter as St'. Thus, $R \subseteq St \times St'$.

A reduction of the second parameter of R, St', can be achieved by applying the well-known restriction operator [8]: Prior to any application of pre-image, a process that owns a slice S_i of S reduces its copy of R by restricting St' to S_i. This reduction is *dynamic* since pre-image operations are applied to different sets during model checking.

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 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 as well.

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 groups of m processes each. All processes in the same group have the same W_i, and hence own the same $S_i = S \cap W_i$. However, each process in the group has a different U_j. Process (i, j) with W_i and U_j computes pre-image of S_i by $pred_j(S_i) = \exists s'[R_j(s, s') \wedge S_i(s')]$. Since $U_1, \ldots, U_m$ is a complete set of window functions, $\bigvee_{j=1}^{m} pred_j(S_i) = pred(S_i)$. Thus, the group with window function W_i computes the same set as process i in the algorithm of Section 3.

Once the computation is completed, procedure `exch` is applied to exchange non-owned states (according to W_i). Procedure `ldBlnc` is used to update the W_i window functions in order to balance the memory load. Both procedures are de£ned as before. However, when `ldBlnc` changes the window functions, all members in each of the groups should agree on the new window function.



The Figure above demonstrates a pre-image computation using 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.

The method suggested in this section applies slicing to the full transition relation in case it can be held in memory, but is too large to enable a successful completion of the pre-image operation. However, often the transition relation is given *partitioned*, i.e., given as a set of small relations N_l, each de£ning the value of variable v_l in the next states. The size of the partitioned transition relation is usually small, therefore can be constructed by one process and then be sliced using the algorithm suggested in [14]. In this case the model checking is done directly with the partitioned transition relation [3].

5.1 Distributed Construction of the Sliced Full Transition Relation

The full transition relation R is a conjunction of all N_l. Here we consider cases where either R or its construction cannot £t 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. Each process starts constructing by gradually conjuncting partitions N_l, until a threshold is reached. The current (partial) transition relation is then partitioned among the processes, using the slicing algorithm. Each process continues to conjunct the partitions that have not been handled yet, until all partitions are conjuncted. During conjunction, further slicing or balancing are applied so that the £nal slices will be balanced.

Acknowledgement: We would like to thank Ken McMillan for his time and patient and for his help in choosing a notation for the μ–calculus model checking algorithm.

References

1. 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)*, Austin, Texas, November 2000.
2. R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
3. 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.
4. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.
5. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic veri£cation of £nite-state concurrent systems using temporal logic speci£cations. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, January 1983.
6. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT press, December 1999.
7. R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27:725–747, 1990.
8. O. Coudert, J. C. Madre, and C. Berthet. Verifying of Synchronous Sequential Machines Based on Symbolic Execution. In J. Sifakis, editor, *Workshop on Automatic Veri£cation Methods for Finite State Systems*, pages 365–373. Springer-Verlag, Grenoble, France, 1989.
9. E. A. Emerson and C.-L. Lei. Ef£cient 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.
10. 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 Veri£cation*. Springer-Verlag, June 2000.
11. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27, 1983.
12. O. Lichtenstein and A. Pnueli. Checking that £nite state concurrent programs satisfy their linear speci£cation. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, January 1985.
13. 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 Veri£cation, LNCS 818*, pages 338–350. Springer-Verlag, 1994.
14. 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.
15. J.P. Quielle and J. Sifakis. Speci£cation and veri£cation of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.
16. 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.
17. A. Tarski. A lattice-theoretical £xpoint theorem and its applications. *Paci£c J. Math*, 1955.
18. G. Winskel. Model checking in the modal ν-calculus. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, 1989.