Distributed Symbolic Model Checking for μ–calculus*

Orna Grumberg, Tamir Heyman, Assaf Schuster

Computer Science Department, Technion, Haifa, Israel

July 17, 2003

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 μ–calculus model checking can solve many problems, including, for example, verification of (fair) CTL and LTL properties. Previous works on distributed symbolic model checking were restricted to reachability analysis and safety properties. This work thus significantly extends the scope of properties that can be verified distributively, enabling us to use them for very large designs.

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

*This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant number 111/01-2).

1 Introduction

In the early 1980s, model checking procedures were suggested [6, 19, 15] which could handle systems with a few thousand states. In the early 1990s, symbolic model checking methods were 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. Significant efforts have been made since to fight the *state explosion problem*. But the need to verify larger systems is growing faster than the capacity of any newly developed method.

Recently, a new promising method to fight the state explosion problem was introduced. The method uses the collective pool of memory modules in a network of processes. Distributed symbolic reachability analysis is used to find the set of all states reachable from the initial states [13]. A distributed symbolic on-the-fly algorithm was applied in order to model check properties written as regular expressions [1]. Experimental results show that distributed methods can reduce the average memory requirement 300 times using 500 processes. Consequently, distributed methods find errors that were not found by sequential tools.

This paper extends the scope of properties that can be verified for large designs. It presents a distributed symbolic model checking algorithm for the μ-calculus, which is a powerful formalism for expressing properties of transition systems using least and greatest fixpoint operators. Many verification procedures can be solved by translating them into μ–calculus model checking[4] problems. Such verification procedures include (fair) CTL model checking, LTL model checking, bisimulation equivalence, and language containment of ω-regular automata.

Many algorithms for μ-calculus model checking have been suggested [10, 20, 22, 8, 16]. In this work we parallelize a simple sequential algorithm [7]. The algorithm works bottom-up through the formula, evaluating each subformula based on the value of its own subformulas. A formula is interpreted as the set of states in which it is true. Thus, for each μ–calculus operation, the algorithm receives a set (or sets) of states and returns a new set of states.

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

by a single process.

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

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

Memory requirements are kept low through frequent calls to a memory balancing procedure. It ensures that each set is partitioned evenly among the processes. This ensures that the memory requirements, which are usually proportional to the size of the manipulated set, are evenly distributed among the processes. However, this also requires different slicing functions for different sets. As a result, we may need to apply an operation to two sets that are sliced according to different partitions. In the case of *conjunction*, for instance, the two sets should first be re-sliced according to the same partition. Only then do the processes apply conjunction to their individual slices. Narayan et al. [18] show how to preform negation, conjunction, and disjunction under the assumption that the set of window functions does not change. However, if the set does not change, the memory requirement will be unbalanced as explained. This will render the distributed system ineffective.

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

The rest of this paper is organized as follows: In Section 2 we briefly review the propositional μ–calculus logic and its model checking algorithm. We also briefly review the distributed symbolic model checking elements that were developed in [13]. In Section 3 we describe our distributed model checking algorithm for μ–calculus. In Section 4 the correctness of our algo-

rithm is proved, and in Section 5 an enhancement for pre-image computation is described. We conclude in Section 7.

2 Preliminaries

2.1 The Propositional μ–Calculus

Below we define the propositional μ–calculus [14]. We will not distinguish between a set of states and the Boolean function that characterizes this set. By abuse of notation we will apply both set operations and Boolean operations on sets and Boolean functions. Let AP be a set of atomic propositions and let $VAR = \{Q, Q_1, Q_2, \ldots\}$ be a set of relational variables. The μ–calculus formulas are defined as follows:

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

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

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

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

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

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

Tarski [21] showed that least and greatest fixpoints always exist if τ is monotonic. If τ is also continuous, then the least and greatest fixpoints of τ can be computed by $\cup_{i \in N} \tau^i(False)$ and $\cap_{i \in N} \tau^i(True)$, respectively. In [7] it is shown that if M is finite then any monotonic τ is also continuous.

In this paper we consider only monotonic formulas. Since the only transition systems we consider are finite, they are also continuous. The function `fixpt` in Figure 2 describes an algorithm for computing the least or greatest fixpoint, depending on the initialization of Q_{val}. If the parameter *init* is $False$, the least fixpoint is computed. Otherwise, if $init = True$, the greatest fixpoint is computed.

Given a transition system M, an environment e, and a formula f of the μ–calculus, the *model checking* algorithm for μ–calculus finds the set of states in M that satisfy f. Figure 1 presents a sequential recursive algorithm for evaluating μ–calculus formulas. For closed μ–calculus formulas, the initial environment is irrelevant. The necessary environments are constructed during recursive applications of the `eval` function.

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

Figure 1: Pseudo–code for sequential μ–calculus model checking

2.2 Elements of Distributed Symbolic Model Checking

Our distributed algorithm includes several basic elements that were developed in [12]. For completeness, we give a brief overview of these elements in this subsection.

Sets of states in the transition system, as well as the intermediate results, are represented by BDDs. At any point during the algorithm's execution,

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

Figure 2: Pseudo–code for computing fixpoint

the sets of states obtained are partitioned among the processes. A set of *window functions* is used to define the partitioning, determining the slice that is stored (we say: *owned*) by each process.

Definition 1: [Complete set of window functions [18, 5]] A window function is a Boolean function that characterizes a subset of the state space. A set of window functions $W_1, \ldots, W_k$ is complete if and only if $\bigvee_{i=1}^{k} W_i = 1$.

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

We use the *slicing algorithm*, as described in [12], 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 slice the input set into approximately equal subsets.

The slicing algorithm uses the `SelectVar` algorithm, which slices a Boolean function (a BDD) into two by assigning a BDD variable. The `SelectVar` algorithm receives a BDD, f, and a threshold, δ. It selects one of the BDD variables v and slices f into $f_v = f \wedge v$ and $f_{\overline{v}} = f \wedge \overline{v}$.

The cost of such a slicing is defined as:

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

The $\frac{MAX(|f_v|,|f_{\overline{v}}|)}{|f|}$ factor gives an approximate measure to the reduction achieved by the partition. The $\frac{|f_v|+|f_{\overline{v}}|}{|f|}$ factor gives an approximate measure of the number of shared BDD nodes in f_v and $f_{\overline{v}}$ and therefore reflects the *duplication* in the partition. The cost function depends on the value of α, where $0 \leq \alpha \leq 1$. $\alpha = 0$ means that the cost function completely ignores the reduction factor, while $\alpha = 1$ means that the cost function completely ignores the duplication factor.

Initially, the algorithm only attempts to find a BDD variable that will minimize the duplication factor ($\alpha = 0$), while still reducing the memory requirements below the threshold (i.e., $\max(|f_1|, |f_2|) \leq |f| - \delta$). If such a slicing does not exist, the algorithm increases α gradually, allowing a gradual increase in duplication until $\max(|f_1|, |f_2|) \leq |f| - \delta$ is reached. Note that even though our algorithm may compute the cost functions for many different α, $|f \wedge v|$ and $|f \wedge \overline{v}|$ are computed only once for each variable v.

Maintaining an equal load while the intermediate results are being stored is essential for the scalability of the parallel algorithm. The equal load is maintained throughout the algorithm by means of a *memory balance* procedure [12]. This procedure matches those processes that have a large memory requirement with processes that have a small one. Each pair of processes then re-slices the union of its two window functions to obtain a better balanced slicing. The pair uses the same procedure that is used to slice the whole state space. Re-slicing of different pairs is performed in parallel. A process with a huge memory requirement may be matched several times with processes that have a small one. This algorithm defines a new set of window functions that will be used to produce further intermediate results. Following the computation of the new set of window functions, the set of states is distributed accordingly.

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

During the memory balance procedure, as well as during other parts of the distributed model checking algorithm, BDDs are shipped between the processes. A compact and universal BDD representation is used, as described in [12], for the communication. To send a local BDD structure, the process first converts it to the universal representation, then sends it to a different process which converts the universal representation back to its local BDD structure. Different variable order is allowed in the different processes. The size of the universal representation is independent of local variable ordering, and it is linear in the BDD size. Converting a universal represented BDD into the receiver BDD structure (according to the local variable order) may sometimes involve higher complexity (up to exponential in certain cases).

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

3.1 Switching to the Distributed Phase

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

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

In model checking, the most acute peaks in memory requirement usually occur while operations are being performed. Thus, it is more important to reduce the second factor. Indeed, rather than minimizing the total size of each process, our algorithm slices each set in a way that reduces the size of its parts. As a result, the slicing criterion may differ for different sets. We use a slicing algorithm[13] described generally in Section 2.2. Slicing is applied to each one of the sets that has already been evaluated when phase

switching occurs.

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

3.2 The Distributed Phase

The distributed version of the model checking algorithm for the μ–calculus is given in Figure 3. While the sequential algorithm finds the set of states that satisfy, in a given model, a formula of the μ–calculus logic, each process in the distributed algorithm finds the part of this set that the process owns. Intuitively, the distributed algorithm works as follows: given a set of slices W_i, a formula f, and an environment e, the process id finds the set of states $\texttt{eval}(f, e) \wedge W_{id}$.

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

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

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

Each process in the distributed algorithm evaluates each subformula f as follows (see Figure 3):

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

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

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

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

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

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

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

Subformulas of the form $\mu Q.g$ and $\nu Q.g$ (the least and greatest fixpoints, respectively): evaluated using a special function `fixpt` that iterates until a fixpoint is found. The computations for the formulas differ only in the initialization, which is $False$ for $\mu Q.g$ and is the current window function for

$\nu Q.g$. The `fixpt` function uses a distribution termination detection procedure, `parterm`, to check whether a fixpoint has been reached. Each process calls `parterm` with a Boolean value. The process reports true if and only if a fixpoint has been reached "as far as it knows." The fixpoint is evaluated by applying `exch` on both the last and current value of Q and comparing the parts that the process owns. Since each state belongs to some process, a fixpoint is reached if none of the processes gets a new state during the last iteration.

4 Correctness

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

Definition 3: [Well-partitioned environment] An environment e is well partitioned by parts $e_1, \ldots, e_k$ if and only if, for every $Q \in VAR$, $e(Q) = \bigvee_{i=1}^{k} e_i(Q)$.

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

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

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

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

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

Figure 3: Pseudo–code for a process id in the distributed model checking

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

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

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

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

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

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

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

Induction:

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

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

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

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

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

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

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

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

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

4. $\underline{f = g_1 \wedge g_2}$: $\texttt{peval}_{id}(g_1 \wedge g_2, e_{id})$ first computes the two sets $res_1^{id} = \texttt{peval}_{id}(g_1, e_{id})$ and $res_2^{id} = \texttt{peval}_{id}(g_2, e_{id})$, then applies `exch` to each of them, and finally conjuncts the results. Note that no `ldBlnc` is invoked between the two applications of `exch`. Therefore, both use the

same window functions. Let $W_1, \ldots, W_k$ be those window functions. Then the global set is

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

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

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

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

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

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

5. $\underline{f = \mathbf{EX}\ g}$: $\texttt{peval}_{id}(\mathbf{EX}g, e_{id})$ evaluates the set of all predecessors of states in $\texttt{peval}_{id}(g, e_{id})$, using the transition relation R. The global set of all predecessors s can be represented by the formula $\bigvee_{i=1}^{k} \exists t[(s,t) \in R \wedge t \in \texttt{peval}_i(g, e_i)]$. The global set computed at this stage is:

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

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

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

By the induction hypothesis, $\bigvee_{i=1}^{k} \texttt{peval}_i(g, e_i) = \texttt{eval}(g, e)$. Thus, the global set is identical to

$$\exists t \left[(s,t) \in R \wedge t \in \texttt{eval}(g, e)\right] \;=\; \texttt{eval}(\ \mathbf{EX}\ g, e).$$

Since the procedures `exch` and `ldBlnc` do not change the global set, $\bigvee_{i=1}^{k} \texttt{peval}_i(\mathbf{EX}g, e_i) = \texttt{eval}(\mathbf{EX}g, e)$.

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

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

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

   (a) At every iteration, e is well partitioned by $e_1, \ldots, e_k$.

   (b) For every j: $Q^j = \bigvee_{i=1}^{k} Q^j_i$.

   (c) If the sequential `fixpt` algorithm terminates after i_0 iterations, then so does the distributed `fixpt` algorithm.

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

Base: $j = 0$:

(a) At iteration 0, e is well partitioned, according to the induction hypothesis of Theorem 1.

(b) In the case that $f = \mu Q.g$, the value of both the sequential and the distributed algorithm at initialization is $False$. Hence, $Q^0 = Q^0_{id} = False$, which implies $Q^0 = \bigvee_{i=1}^{k} Q^0_i$.

(c) Since both algorithms perform at least one iteration, they will not terminate at iteration 0.

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

(a) Let e', $e'_1, \ldots, e'_k$ be the environments at the end of iteration $j+1$, and assume that e is well partitioned by $e_1, \ldots, e_k$ at the end of iteration j. The only changes to the environments in iteration $j+1$ may occur in line 5 of the distributed and sequential algorithms. Changes may occur for two reasons: $e(Q)$ may be assigned a new value Q^j, or a recursive call to `eval` may change e. Similarly, in the distributed algorithm, two changes may occur: $e_{id}(Q)$ may be assigned a new value Q^j_{id}, or a recursive call to $\texttt{peval}_{id}$ may change e_{id}.
By the induction hypothesis of Lemma 2 we know that $Q^j = \bigvee_{i=1}^{k} Q^j_i$. Hence, $e[Q \leftarrow Q^j](Q) = \bigvee_{i=1}^{k} e_i[Q \leftarrow Q^j_i](Q)$. Since no other change has been made to the environments, and since e is well partitioned, we conclude that $e[Q \leftarrow Q^j]$ is well partitioned by $e_1[Q \leftarrow Q^j_1], \ldots, e_k[Q \leftarrow Q^j_k]$.
In iteration $j + 1$, `eval` is now invoked with an environment that is well partitioned by the environments $\texttt{peval}_{id}$ is invoked with. The induction hypothesis of Theorem 1 therefore guarantees that e' is well partitioned by $e'_1, \ldots, e'_k$.

(b) $Q^{j+1} = \texttt{eval}(g, e[Q \leftarrow Q^j])$ (line 5 of the sequential algorithm) and $Q^{j+1}_{id} =$
$\texttt{peval}_{id}(g, e[Q \leftarrow Q^j_{id}])$ (line 5 of the distributed algorithm).
By item (a), $e[Q \leftarrow Q^j]$ is well partitioned. Thus, the induction hypothesis of Theorem 1 is applicable and implies that

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

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

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

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

$$\forall id[\texttt{exch}(Q_{id}^j) = \texttt{exch}(Q_{id}^{j+1})] \quad \Leftrightarrow$$

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

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

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

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

4.1 The Processes Own Disjoint Subsets

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

Definition 4: [Disjoint Set of Window Functions] A set of window functions $W_1, \ldots, W_k$ is disjoint if and only if, for every $1 \leq t, l \leq k$, $t \neq l$, $W_t \wedge W_l = 0$.

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

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

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

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

Definition 5: [Disjoint Environment] Environment parts $e_1, \ldots, e_k$ are disjoint if and only if, for every $Q \in VAR$, for every $1 \leq t, l \leq k$, $t \neq l$, $e_t(Q) \wedge e_l(Q) = 0$.

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

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

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

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

The set of window functions is modified by applying `ldBlnc` at the end of `peval`. The procedure `ldBlnc` repartitions the subsets between the processes. However, the set of window functions remains disjoint. Therefore, $W'_1 \ldots W'_k$ is a disjoint set of window functions.
Base: $f = p$ for $p \in AP$ for every $1 \leq t, l \leq k, t \neq l$, $\texttt{peval}_t(f, e_t) \wedge \texttt{peval}_l(f, e_l) = \{s \mid p \in L(s)\} \wedge W_t \wedge \{s \mid p \in L(s)\} \wedge W_l$.
Since for every $1 \leq t, l \leq k$, $t \neq l$, $W_t \wedge W_l = 0$ (the set of window functions

is disjoint), the above expression is equal to 0.

Induction step:

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

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

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

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

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

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

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

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

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

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

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

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

(b) For every j,$1 \leq t, l \leq k$, $t \neq l$, $Q^j_t \wedge Q^j_l = 0$.

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

Base: $j = 0$:

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

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

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

(a) Let $e'_1, \ldots, e'_k$ be the environments at the end of iteration $j + 1$, and assume that $e_1, \ldots, e_k$ are disjoint at the end of iteration j.

The only changes to the environments in iteration $j+1$ may occur in line 5 of the algorithms. Changes may occur for two reasons: $e_{id}(Q)$ may be assigned a new value Q_{id}^j, or a recursive call to $\texttt{peval}_{id}$ may change e_{id}.

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

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

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

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

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

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

5 Scalable Distributed Pre-image Computation

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

S grow. Furthermore, intermediate results sometimes exceed the memory capacity even when $pred(S)$ can be held in memory.

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

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

A reduction of the second parameter of R, St', can be achieved by applying the well-known restriction operator [9]: Prior to any application of the pre-image computation, a process that owns a slice S_i of S reduces its copy of R by restricting St' to S_i. Since pre-image operations are applied to different sets during model checking, this reduction is *dynamic*.

We further reduce R by adding a *static* slicing of St according to (possibly different) window functions $U_1, \ldots, U_m$. The slicing algorithm of Section 2.2 can be used to produce $U_1, \ldots, U_m$, so that R is partitioned to m slices of similar size. Each slice R_j is a subset of $(St \cap U_j) \times St'$. Since R does not change during the computation, $U_1, \ldots, U_m$ do not change either.

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

Figure 5 above demonstrates a pre-image computation using a sliced transition relation with $k = 2$ and $m = 3$. Given a set S sliced into S_1, S_2 according to W_1, W_2 respectively, the pre-image of S_1 is computed by three processes. Each process uses a different slice of the transition relation, R_1, R_2 and R_3, according to U_1, U_2 and U_3.

5.1 Model Checking Algorithm with Sliced Transition Relation

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

A formula in the form $\mathbf{EX}g$ is evaluated by first using the processes $(1, 1)$,$(2, 1)$,...,$(k, 1)$ to evaluate g. Then each process $(i, 1)$ broadcasts its copy of g_i to the processes $(i, 2)$,...,(i, m). Each process (i, l) computes the pre-image of g_i using R_l. Finally, the processes use the algorithm `exchstr` (given in Figure 4) to complete the evaluation and update the processes $(1, 1)$,$(2, 1)$,...,$(k, 1)$.

```
functionexchstr(S,(uId,wId))
1   for all 1 ≤ i ≤ k
2     sendto((i,1), S ∧ W_i)
3   if uId ≠ 1 return 0
    /* uId = 1 */
4   res= ∅
5   for all 1 ≤ l ≤ m
6     for all 1 ≤ i ≤ k
7       res= res∨receivefrom((i,l))
8   return res
end function
```

Figure 4: Pseudo–code for exchanging non-owned states after pre-image computation using the sliced transition relation

The method suggested in this section applies slicing to the full transition relation if it can be held in memory but is too big to enable a successful completion of the pre-image operation. However, the given transition relation is often *partitioned*, i.e., it is given as a set of small relations N_l, each defining the value of variable v_l in the next states. The size of the partitioned transition relation is usually small; therefore it can be constructed by one process and then sliced using the algorithm suggested in [17]. In this case, model checking is done directly with the partitioned transition relation [3].

5.2 Distributed Construction of the Sliced Full Transition Relation

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

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

5.3 Correctness of the Algorithm with a Sliced Transition Relation

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

Definition 6: [Sliced Transition Relation] A transition relation R corresponds to a sliced transition relation $R_1, \ldots, R_m$ if and only if for every $1 \leq l \leq m$, $R_l = R \wedge U_l$, where $U_1, \ldots, U_m$ is a complete set of window functions.

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

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

Proof: We prove the theorem by induction on the structure of f.
$\texttt{parevalstr}_{i,1}(f, e_i)$ works the same way as $\texttt{peval}_i(f, e_i)$ does for all for-

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

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

The above formula is therefore identical to:

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

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

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

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

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

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

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

The last expression is identical to:

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

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

6 Scalability

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

Our experience in previous work [13, 1] indicates that the bandwidth of the current standard network allows systems with a few dozen nodes to work effectively, and communication does not become a bottleneck. A very large network will need to handle larger communication volume.

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

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

The algorithm works bottom up through the formula, evaluating each subformula based on the value of its own subformulas. It evaluates each subformula using a number of nodes that work in parallel. However, the evaluation is synchronized by the call to the `ldBlnc`, `exch` and `exchnot` procedures. The evaluation takes a constant number of operations for all the operators except fixpoint. Lemma 2 proves that the distributed algorithm takes the same number of steps for fixpoint operators as the sequential. Therefore, we conclude that the complexity of the distributed algorithm is the same as in the sequential case. The complexity of evaluating a formula depends only on the number of alternations d of the least and greatest fixpoints [10]. A sequential [10, 16] algorithm requires n^d steps where n is the number of states in the transition system.

Our algorithm requires several standard machines, each consisting of local processors and local memory. The communication between the machines consists of a standard ethernet. The algorithm can be implemented using the MPI standard [11]. Therefore, it does not require any special architecture.

7 Conclusion

This paper presents a framework for distributed symbolic model checking. It includes a scalable distributed symbolic model checking algorithm for μ–calculus. It suggests using a sliced transition relation for image computation of very large transition systems. Many other model checking algorithms for subsets of μ–calculus can use this framework. Future work should address such implementation issues as selecting window functions, selecting order of communication during the exchange procedure, and balancing memory utilization without forcing synchronization.

Acknowledgement: We would like to thank Ken McMillan for his time, patience and 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), LNCS*, 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):142–171, June 1992. Special Issue: Selections from 1990 IEEE Symposium on Logic in Computer Science.

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

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

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

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

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

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

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

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

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

[14] D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27, 1983.

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

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

[17] 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.

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

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

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

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

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