Distributed and Predictable Software Model Checking

Nuno P. Lopes[1] and Andrey Rybalchenko[2]

[1] INESC-ID / IST, TU Lisbon
[2] Technische Universität München

Abstract. We present a predicate abstraction and refinement-based algorithm for software verification that is designed for the distributed execution on compute nodes that communicate via message passing, as found in today's compute clusters. A successful adaptation of predicate abstraction and refinement from sequential to distributed setting needs to address challenges imposed by the inherent non-determinism present in distributed computing environments. In fact, our experiments show that up to an order of magnitude variation of the running time is common when a naive distribution scheme is applied, often resulting in significantly worse running time than the non-distributed version. We present an algorithm that overcomes this pitfall by making deterministic the counterexample selection in spite of the distribution, and still efficiently exploits distributed computational resources. We demonstrate that our distributed software verification algorithm is practical by an experimental evaluation on a set of difficult benchmark problems from the transportation domain.

1 Introduction

There has been a recent rapid growth of the amount of computing resources clustered in data centers together with their increasing availability for wide access. Compute clusters can be found in academic institutions, and today it is even possible to rent computers from major providers. A typical cluster offers a large number of compute nodes interconnected via a high throughput and low latency network. The number of available CPUs can be up to one hundred and more, however usually they do not share any memory, i.e., the communication between compute nodes is via message passing over the network.

Such clusters open unprecedented opportunities for amplifying the scalability of software model checking tools. Software model checking [12] can be abstractly viewed as a tree construction, where nodes correspond to sets of program states and edges are labeled by program statements. The goal is to compute a set of nodes that contains all states that the program can reach, and then check if error states were included. Computation of child nodes is an expensive task that requires invocation of a decision procedure. This tree construction procedure has a promising potential for distribution. However, to the best of our knowledge, such computing infrastructure has not yet been utilized for software model checking.

In this paper we present a distributed version of a software model checking algorithm that is designed for the distributed execution on compute nodes that communicate via message passing. Our algorithm implements a prominent approach to software verification that is based on predicate abstraction and its counterexample-guided refinement [2, 3, 6, 8]. Based on preliminary experiments, we decided to use a centralized approach with a single master node and a set of worker nodes. The master node keeps the reachability tree and a queue of nodes whose successors need to be computed by the workers. In addition, each worker node maintains a partial reachability tree that is used to locally check if a computed set of program states has been already reached. Although theoretically the master node is a bottleneck of our approach, our preliminary experiments indicated that in practice neither CPU nor network usage reach their limits.

A successful adaptation of a predicate abstraction-based model checking algorithm from sequential to distributed setting needs to address challenges imposed by the inherent non-determinism present in distributed computing environments. Since the success of predicate abstraction-based verifiers crucially depends on the choice of counterexamples used to discover predicates, any deviation from an intended program exploration strategy, and hence a counterexample selection strategy, e.g., BFS, can significantly impact the verifier running time.

Message queues, different link latencies and CPU speeds make counterexample selection non-deterministic, i.e., different verification runs can discover different series of counterexamples. Such lack of predictability can have a twofold negative impact. First, the verification time can increase due to a suboptimal exploration strategy. In fact, our experiments with a naive distribution scheme summarized by the table in Fig. 1 show that up to an order of magnitude variation of the running time can occur.[3] Due to the random counterexample choice, the running time of the naive distributed algorithm can be up to two times slower than the sequential algorithm. Second, the outcome of the verification is practically impossible to reproduce. (Note however that computed proofs can still be checked, since the proof validity is independent of how it was computed.)

In this paper we present an algorithm that addresses the predictability requirement. Our algorithm overcomes this efficiency pitfall by making deterministic the counterexample selection in spite of the distribution. While achieving the desired deterministic behavior at the level of abstraction refinement, the algorithm does not impose any synchronization during the reachability computation, which is necessary to achieve the full utilization of available resources.

We implemented our distributed algorithm as an extension of the model checker ARMC [17] by using the DAHL distribution framework [14]. We demonstrate that our distributed software verification algorithm is practical by an experimental evaluation on a set of difficult benchmark problems from the transportation domain. For the evaluation we used a compute cluster with 40 CPUs on commodity workstations. In the experiments we observed a linear scalability in the number of CPUs with the factor 0.4–0.5, that is, with 40 CPUs our imple-

[3] The naive approach distributes the computation of reachable states among workers without any attempt to control the counterexample discovery.

Test	Slowdown			
	5 nodes	10 nodes	20 nodes	40 nodes
larger_scale1_lb	1.03	1.79	2.63	3.29
larger_scale1_ub	–	–	–	–
scale1_lb	1.16	1.36	1.22	1.80
scale1_ub	1.63	33.21	13.90	1.04
timing	1.13	1.04	1.00	1.01
gasburner	1.92	2.08	1.71	2.18
rtall_tcs	1.48	1.37	1.29	2.94

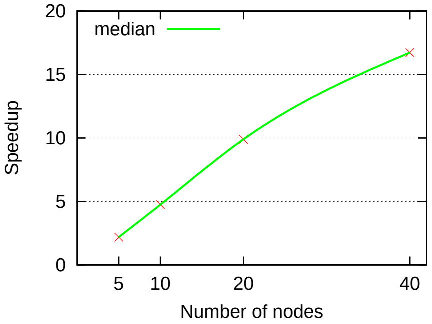

Fig. 1. On the left, we show the slowdown – the ratio between the best and the worst running time – of a naive distributed algorithm computed for three runs for each example. In larger_scale1_ub, the naive distribution did not succeed due to reaching the memory limit of 2 GB caused by a very large number of predicates collected in several days of running time. On the right, the summary of our experimental evaluation is presented. The graph presents the correlation of speedup – ratio between the running time in the sequential and distributed settings – and the number of compute nodes.

mentation is up to 20 times faster (the median value over all benchmarks) than the sequential one, see Figure 1 and Section 5 for more details. We also observed that our algorithm reduces the variation of running time across different model checker runs to negligible 1–3% (in contrast to 150–3300% as witnessed by the table of Figure 1 for the naive distribution).

In summary, this paper makes the following contributions:

- To the best of our knowledge, the first distributed model checking algorithm based on predicate abstraction and its counterexample guided refinement that offers predictable execution behavior and linear scalability;
- A practical implementation of the proposed algorithm;
- An experimental evaluation of the algorithm on a network of workstations demonstrating the effectiveness of the approach.

Related work Previous efforts for utilizing distributed computation environments were focused on symbolic and explicit model checking algorithms, [5, 9, 13, 20]. Overcoming memory limitations of a single workstation was their primary objective, yet some implementations also delivered increase of performance. For example, the parallel Murϕ verifier [20] achieves near-linear speedups with up to 60 machines. The algorithm distributes the state space across the machines using a (static) hash function. The algorithm of [5] also shows near-linear speedups with up to ten machines. None of these algorithms needed to address the determinacy issues arising due to iterative abstraction refinement.

Jha [11] proposes d-IRA, a parallel version of refinement based model checking for hybrid systems, and shows near-linear speedups when executed on four CPUs (on the same machine). Our algorithm is similar to d-IRA in its usage of a master-slave architecture and counterexample generation entirely performed by the master. The crucial difference lies in the granularity of distribution. d-IRA lets each worker compute a relaxation of the full system for a single counterexample, while our algorithm distributes each abstract reachability computation.

Holzmann et al. [10] exploit the availability of multiple CPUs in the Swarm tool that runs several instances of a model checker in parallel with different state space exploration strategies. Each instance runs reachability computation sequentially, while our algorithm targets its distribution.

Venet and Brat [21] propose an algorithm for distributed pointer analysis that was shown to scale up to four CPUs. Similarly to our algorithm, the proposed algorithm uses a master node to distribute the global state and assign the work pieces. The main difference lies in the distribution scheme, since it uses work pieces that are individual C source files, while our algorithm distributes at the level of individual statements.

Monniaux [16] presents an algorithm for distributed abstract interpretation with widening, and its evaluation on five CPUs. The algorithm uses a master node to distribute the work pieces that are created at branch or dispatch points in the source code. The algorithm of [16] makes distributed abstract interpretation deterministic by requiring the associativity and commutativity of the join operator. In contrast, our algorithm achieves determinism through a deterministic counterexample selection scheme.

Prabhu et al. [18] present a 0CFA specific algorithm that exploits the parallelism and computation power of GPUs, achieving speedups up to 72x.

2 Example

We illustrate our distributed algorithm on a simple example program. First, we show how the choice of counterexamples for the predicate discovery affects the overall execution of the reachability computation. Then, we demonstrate what our algorithm does to make the counterexample selection process deterministic.

Consider the program in Figure 2 for which we want to check the validity of the assertion in the last line. On the right we present the corresponding control-flow graph, in which $\ell_{\mathcal{I}}$ is the initial location and $\ell_{\mathcal{E}}$ is the error location. The edges are annotated with transition relations, i.e., logical formulas over the program variables and their primed versions that represent the effect of executing the program statement. Our goal is to prove that the error location $\ell_{\mathcal{E}}$ is not reachable. Intuitively, the assertion validity can be proved by keeping track of the following two observations, so-called predicates, about the program: $x \geq 1$ and $y \geq 1$.

Now we consider how this program is verified by a distributed verification algorithm. The algorithms keeps track of a fixed set of predicates over variables and traces how the predicate validity changes when the program state is modified by executing program statements. For the sake of illustration we omit the details of how the reachability trees are constructed and focus on counterexamples, which are paths through a reachability tree that lead to the error locations, and their use for predicate discovery.

```
if (x > 0) {
    y = x;
} else if (x == 0) {
    y = 2;
} else {
    y = 1;
    y = y + 1;
}
assert(y >= 1);
```

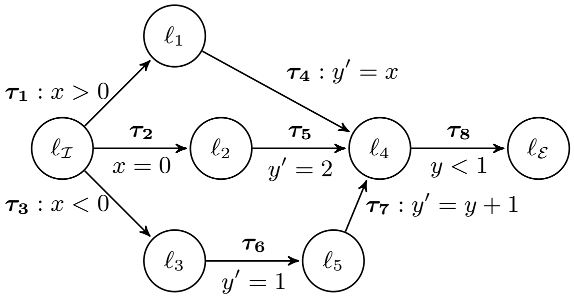

Fig. 2. An example C program and its control-flow graph. To simplify the figure, we omit update expressions $x' = x$ and $y' = y$ in the transition relations.

Naive distribution In the case of naive distribution we do not impose any constraints on the counterexample selection. First, we attempt to prove the program correct without keeping track of any predicates. In this case we could discover any of the three counterexamples $\pi_1 = \tau_1\tau_4\tau_8$, $\pi_2 = \tau_2\tau_5\tau_8$, and $\pi_3 = \tau_3\tau_6\tau_7\tau_8$, since any path through the program that leads to the assertion appears to be feasible when we omit any reasoning about the values of program variables. By applying a predicate discovery procedure on these counterexamples, e.g., the one based on interpolation [7, 19], we obtain the following set of predicates, respectively: $\mathcal{P}_1 = \{x \geq 1, y \geq 1\}$, $\mathcal{P}_2 = \{y \geq 2\}$, and $\mathcal{P}_3 = \{y \geq 1, y \geq 2\}$.

Now we continue by considering each possible counterexample, i.e., we simulate how the verification proceeds if π_1, π_2, or π_3 was discovered and provided predicates. For each of the scenarios we now assume that the algorithm keeps track of the discovered set of predicates $\mathcal{P}_1$, $\mathcal{P}_2$, or $\mathcal{P}_3$ respectively.

In the first case, no further counterexample is found, which means that the program is proved correct using $\mathcal{P}_1$. For the cases 2 and 3, the algorithm can discover either $\pi_{2.1} = \tau_1\tau_4\tau_8$ or $\pi_{2.2} = \tau_3\tau_6\tau_7\tau_8$, and $\pi_{3.1} = \tau_1\tau_4\tau_8$, respectively. For these paths the predicate discovery yields the following sets of predicates, respectively: $\mathcal{P}_{2.1} = \{y \geq 2, x \geq 1, y \geq 1\}$, $\mathcal{P}_{2.2} = \{y \geq 2, y \geq 1\}$, and $\mathcal{P}_{3.1} = \{y \geq 1, y \geq 2, x \geq 1\}$.

Taking the sets of predicates $\mathcal{P}_{2.1}$ or $\mathcal{P}_{3.1}$ into consideration leads to the completion of the correctness proof in the respective cases. For the scenario that discovered the predicates $\mathcal{P}_{2.2}$, another refinement iteration is needed, which discovers and analyzes the counterexample $\pi_{2.2.1} = \tau_1\tau_4\tau_8$ producing the predicates $\mathcal{P}_{2.2.1} = \{y \geq 2, y \geq 1, x \geq 1\}$.

In summary, we have the following four scenarios of executing the naive distribution, which are determined by the discovered sequences of counterexamples: π_1, $(\pi_2, \pi_{2.1})$, $(\pi_2, \pi_{2.2}, \pi_{2.2.1})$, and $(\pi_3, \pi_{3.1})$.

Our simple example indicates that depending on the choice of counterexamples to perform refinement, which is non-deterministic in a naive distribution, the verification algorithm can take one, two, or three iterations, which results in a significant variation of the execution time. For complex, large program, the difference in behavior can be significantly bigger.

Predictable distribution Now we demonstrate how our deterministic counterexample selection can be enforced. First, we define a total ordering $\prec$ on counterexample paths, which can be the lexicographic ordering where individual transitions are compared by considering their names. Assume that we define $\tau_1 \prec \cdots \prec \tau_8$, which yields for example that $\tau_1\tau_4\tau_8 \prec \tau_2\tau_3\tau_8$.

Now we can use the ordering to select the shortest minimal discovered counterexample. However, when committing to the selection of a candidate counterexample we need to take into consideration the possibility that a smaller counterexample – currently under construction in the distributed environment – can be discovered later on. This means that we need to wait until the reachability tree construction reaches the depth of the candidate.

For example, assume that the counterexamples π_2 and π_3 are already discovered and π_2 is the candidate for selection, since π_2 is shorter than π_3. We still need to wait until the reachability tree construction reaches the depth 3, which is the length of π_2. Finally, π_1 will be discovered and since $\pi_1 \prec \pi_2$, it will be selected for the predicate discovery.

Thus, the deterministic selection approach leads to the predictable execution of the verification algorithm while leaving unconstrained the order on which the reachability tree is explored. The induced cost of waiting for the completion of the tree construction up to the depth determined by the candidate counterexample does not lead to a significant overhead in practice, as our experiments indicate.

In the remaining sections we provide a detailed description of our distributed verification algorithm, which is based on the idea illustrated in this section.

3 Preliminaries

In this section we define programs and computations, and provide a brief description of predicate abstraction-based approach to program verification together with a counterexample-guided abstraction refinement procedure.

Programs and computations We assume an abstract representation of programs by transition systems [15]. A *program* $P = (\Sigma, s_{\mathcal{I}}, \mathcal{T}, s_{\mathcal{E}})$ is given by a set of program *states* Σ, an *initial* state $s_{\mathcal{I}} \in \Sigma$, a set of *transitions* $\mathcal{T}$, and an *error* state $s_{\mathcal{E}} \in \Sigma$. Each transition $\tau \in \mathcal{T}$ has a corresponding *transition relation* $\rho_\tau \subseteq \Sigma \times \Sigma$. The error state $s_{\mathcal{E}}$ is used to represent assertion statements commonly present in programming languages. Each failed assertion leads to $s_{\mathcal{E}}$.

A *computation* of P is a sequence of states $s_1, s_2, \ldots$ such that s_1 is the initial state, i.e., $s_1 = s_{\mathcal{I}}$, and there is a transition $\tau \in \mathcal{T}$ between each pair of consecutive states s and s', i.e., $(s, s') \in \rho_\tau$. A state s is *reachable* if it appears in some computation. The program is *safe* if the error state is *not* reachable.

A *path* is a sequence of transitions. Let $\circ$ be the *relational composition* function for binary relation over states, i.e., for $X, Y \subseteq \Sigma \times \Sigma$ we have $X \circ Y = \{(s, s') \mid \exists s'' \in \Sigma : (s, s'') \in X \wedge (s'', s') \in Y\}$. Then, a path relation ρ_π is a relational composition of transition relations along the path, i.e., for $\pi = \tau_1 \ldots \tau_n$ we have $\rho_\pi = \rho_{\tau_1} \circ \cdots \circ \rho_{\tau_n}$. A path is *feasible* if its path relation is not empty.

Predicate abstraction Our goal is to verify if a given program is safe. To achieve this, we need to consider all reachable states and check if the error state appears among them. The set of all reachable states can be computed iteratively using the function $\mathit{post} : (\mathcal{T} \times 2^{\Sigma}) \to 2^{\Sigma}$ such that $\mathit{post}(\tau, S) = \{s' \mid \exists s \in S : (s, s') \in \rho_\tau\}$. Its least fixed point above $\{s_\mathcal{I}\}$ is the set of reachable states, i.e.,

$$s \text{ is reachable if and only if } s \in \mathit{lfp}(\lambda S. \bigcup_{\tau \in \mathcal{T}} \mathit{post}(\tau, S), \{s_\mathcal{I}\}) \ .$$

The exact computation of the set of reachable states is a undecidable problem, however for the verification purposes a sufficiently close *abstraction* is enough. The framework of abstract interpretation [4] provides a formal foundation for the approximate, yet sound abstraction of reachable states, where abstraction is defined as an *over-approximation.* Given an abstraction function $\alpha : 2^{\Sigma} \to 2^{\Sigma}$ such that $\forall S \subseteq \Sigma : S \subseteq \alpha(S)$, we construct an *abstraction* $\mathit{post}^\#$ of *post* as follows: $\mathit{post}^\#(\tau, S) = \alpha(\mathit{post}(\tau, S))$. Our abstraction puts together and operates on sets of program states. We call such sets *abstract states* and let $\Sigma^\# = 2^\Sigma$ be the set of all abstract states. The least fixed point of $\mathit{post}^\#$ above the abstraction of the initial state is an over-approximation of the reachable states, i.e.,

$$\mathit{lfp}(\lambda S. \bigcup_{\tau \in \mathcal{T}} \mathit{post}^\#(\tau, S), \alpha(\{s_\mathcal{I}\})) \ \supseteq \ \mathit{lfp}(\lambda S. \bigcup_{\tau \in \mathcal{T}} \mathit{post}(\tau, S), \{s_\mathcal{I}\}) \ .$$

If the error state is not included in the over-approximation then the program is safe, that is, we obtain a sound method for verifying program safety.

The abstraction function α can be constructed automatically from a given set of basic building blocks, called *predicates*, where a predicate represents a set of program states. Given a set of predicates $\mathcal{P} = \{P_1, \dots, P_n\}$, where $P_i \subseteq \Sigma$, and a *theorem prover* that can decide validity of subset inclusion between sets of states represented in a logical language, we obtain an implementation of an abstraction function $\alpha^\mathcal{P} : 2^\Sigma \to 2^\Sigma$ as follows: $\alpha^\mathcal{P}(S) = \cap\{P \in \mathcal{P} \mid S \subseteq P\}$.

Abstraction refinement In order to verify program safety using predicate abstraction, we need to supply a set of predicates. Predicates can be derived in a goal-oriented way by using the counterexample-guided abstraction refinement approach [3]. The crux of this approach to predicate discovery lies in leveraging *spurious counterexamples*, which are program paths that expose the coarseness of the abstraction function determined by the currently used set of predicates.

A path $\pi = \tau_1 \dots \tau_n$ is a spurious counterexample if the abstract reachability computation along the path leads to the error states, i.e.,

$$s_\mathcal{E} \in \mathit{post}^\#(\tau_n, \mathit{post}^\#(\tau_{n-1}, \dots \mathit{post}^\#(\tau_1, \alpha^\mathcal{P}(\{s_\mathcal{I}\})))\ ,$$

but the actual, not abstracted path does not lead to the error state, i.e., $(s_\mathcal{I}, s_\mathcal{E}) \notin \rho_\pi$. By analyzing a spurious counterexample using automated reasoning techniques, e.g., proofs [8] and interpolation [7], we extract a set of new predicates that *excludes* the spurious counterexample. Let $\textsc{Refine} : \mathcal{T}^+ \to 2^\Sigma$

be a refinement function that extracts predicates from spurious counterexamples. Then, after adding the extracted set of predicates to the abstraction function and obtaining $\alpha^{\mathcal{P} \cup \textsc{Refine}(\pi)}$, the error state is no longer reachable via abstract reachability computation along π.

4 Predictable distributed verification algorithm

This section presents our distributed algorithm for the predicate abstraction-based verification of program safety together with counterexample guided abstraction refinement. Our algorithm is parametrized. Its first parameter is the function Refine, which takes a spurious counterexample and returns a set of predicates that, when used by the abstraction function, leads to the elimination of the counterexample during the abstract reachability computation. The second parameter is a total ordering on program paths $\prec$, which can be defined by the lexicographic extension of a total order on program transitions' names.

We designed our algorithm for the execution in a centralized environment that consists of a master node and a set of worker nodes, where there is a bidirectional communication link between the master and each of the workers. We assume that messages are received in the same order they were sent with respect to a single sender. This assumption greatly simplifies the algorithm and does not impose any practical restriction, since it can be established by using the TCP transport protocol which is ubiquitously available.

Our algorithm consists of four building blocks given by the procedures MainMaster, MainWorker, AddNode, and the function SelectCounterexample. The master node executes the procedure MainMaster as the main event processing loop, which in turn can invoke AddNode and SelectCounterexample. Each worker executes MainWorker, which in turn can execute AddNode. The procedure AddNode takes as the first input parameter a flag indicating whether AddNode is executed on the master or on a worker.

We describe each building block of our algorithm in details, while referring to the program in Figure 2 as a running example.

Procedure MainMaster Figure 3 shows the implementation of MainMaster. This procedure maintains the central reachability tree by keeping the nodes whose successors need to be computed in an outgoing queue, which is processed by workers, and putting the computed successors to the tree.

The master starts by adding the abstraction of the initial program state to the queue (line 4), and then waits (while idling at line 7) for workers to join the system and ask for work.

The master handles three kinds of events. "`EventJoin` w" indicates that a worker node w wants to join the system, "`EventAskForNode` w" is sent by a worker w if it asks for work, and "`EventReachedNode` n" delivers a computed successor to the master.

When a worker w wants to join the system, it sends a `EventJoin` w event to the known address of the master node. Upon reception, the master registers the worker (line 9) and sends the current set of predicates to the worker (line 10).

```
procedure MainMaster
input
  $P = (\Sigma, s_{\mathcal{I}}, \mathcal{T}, s_{\mathcal{E}})$ - program
vars
  $\mathcal{P} \subseteq 2^{\Sigma}$ - finite set of predicates over states
  $Reach \subseteq \Sigma^{\#}$ - reachable abstract states
  $Parent \subseteq \Sigma^{\#} \times \mathcal{T} \times \Sigma^{\#}$ - parent relation between abstract states
  $Depth : \Sigma^{\#} \to \mathbb{N}$ - depth of abstract states
  $DepthBound \in \mathbb{N} \cup \{\infty\}$ - minimal depth of reachable error abstract state
  $Queue \in (\Sigma^{\#})^{*}$ - queue of abstract states
  $Workers, IdleWorkers$ - worker nodes and their idle subset
begin
  $Reach := \{\alpha^{\mathcal{P}}(\{s_{\mathcal{I}}\})\}$
  $Depth := \lambda x.0$
  $DepthBound := \infty$
  $Queue := \alpha^{\mathcal{P}}(\{s_{\mathcal{I}}\})$
  $\mathcal{P} := Parent := Workers := IdleWorkers := \emptyset$
  repeat
    match InputEvent() with
     | EventJoin $w$ ->
       $Workers := \{w\} \cup Workers$
       send ($w$, EventPreds $\mathcal{P}$)
     | EventAskForNode $w$ ->
       if $Queue$ is not empty then
         $n :=$ take from $Queue$
         send ($w$, EventDoNode $(n, Depth(n))$)
       else
         $IdleWorkers := \{w\} \cup IdleWorkers$
         if $IdleWorkers = Workers$ then
           if $DepthBound = \infty$ then
             return "program $P$ is safe"
           else
             $\pi :=$ SelectCounterexample()
             if $\rho_{\pi} = \emptyset$ then
               $\mathcal{P} :=$ Refine$(\pi) \cup \mathcal{P}$
               for each $w' \in Workers$ do
                 send ($w'$, EventPreds $\mathcal{P}$)
               send ($w$, EventDoNode $(\alpha^{\mathcal{P}}(\{s_{\mathcal{I}}\}), 0)$)
               $Reach := \{\alpha^{\mathcal{P}}(\{s_{\mathcal{I}}\})\}$
               $Depth := \lambda x.0$
               $DepthBound := \infty$
               $Parent := Queue := \emptyset$
             else
               return "counterexample $\pi$"
     | EventReachedNode$(n, d, m, \tau)$ ->
       AddNode("master", $n$, $d$, $m$, $\tau$)
       if $n \in s_{\mathcal{E}} \wedge Depth(n) < DepthBound$ then
         $DepthBound := Depth(n)$
end.
```

Fig. 3. Procedure MainMaster.

The master handles workers requests for work according to two scenarios depending on the work availability, either (lines 12–14) or (lines 15–32). If there is work available, then the master sends a piece of work to the worker. Otherwise, the worker is queued in the list of idle workers (line 16). If all the registered workers become idle, then it means the reachability computation has terminated, either by exploring the full tree or reaching the depth bound. The program is correct if no counterexamples have been found upon termination (lines 18–19). Otherwise, the master selects a counterexample by calling the function SelectCounterexample and checks its feasibility (lines 21–22). If the counterexample is not feasible, then it is analyzed by applying Refine and the discovered predicates are broadcasted to all workers (lines 23–25). Finally, we restart the reachability computation by initializing the relevant data structures.

When a new abstract state is computed by a worker and the worker determines that the state might have not been reached before, the state is sent to the master, where the final decision if the state should be added to the tree is made (line 34). If the added abstract state contains the error state of the program and its depth is less than the current depth bound, then the algorithm adjusts the counterexample depth bound (lines 35–36).

Procedure MainWorker We present the implementation of MainWorker in Figure 4.

When a worker is started, first it joins the system by sending a join message to the master (line 3), and then asks for a piece of work (line 4). Afterwards, the worker processes the events received from the master (lines 5–15).

Workers handle two kinds of events. "`EventPreds` $\mathcal{P}$" is received before the abstract reachability computation begins. This event is used to update the set of predicates, as well as to reset the relevant data structures (lines 7–10). The other event kind processed by workers is "`EventDoNode` n", which comes from the master and contains an abstract state whose immediate successors should be computed by the worker. Upon receiving such an event, the worker applies the one-step abstract reachability operator on the received abstract state, adds the result to its local reachability tree (lines 12–14), and, if a computed abstract state appears to have not been discovered previously, sends it to the master. Finally, the worker asks for a new piece of work.

Procedure AddNode See Figure 5 for the implementation of the procedure AddNode. This procedure is used to add an abstract state to the abstract reachability tree and is responsible for discarding those abstract states that represent program states already appearing in the tree.

AddNode starts by pruning nodes that are subsumed by the abstract state that we are trying to add (line 1). The checks for the depth are part of the implementation to enforce deterministic execution. So, we only allow a node to prune other if its depth is smaller. As an optimization, if the abstract state that we are trying to add is already present in the tree at the same depth, then we only update the parent relation, as the recomputation of the node is unnecessary (lines 2–3). Finally, if the abstract state is not subsumed by any other one present

```
   procedure MainWorker
   input
      $P = (\Sigma, s_{\mathcal{I}}, \mathcal{T}, s_{\mathcal{E}})$ - program
      $self$ - this node
      $master$ - master node
   vars
      $\mathcal{P} \subseteq 2^{\Sigma}$ - finite set of predicates over states
      $Reach \subseteq \Sigma^{\#}$ - reachable abstract states
      $Parent \subseteq \Sigma^{\#} \times \mathcal{T} \times \Sigma^{\#}$ - parent relation between abstract states
      $Depth : \Sigma^{\#} \rightarrow \mathbb{N}$ - depth of abstract states
   begin
1     $Depth$ := $\lambda x.0$
2     $\mathcal{P}$ := $Parent$ := $Reach$ := $\emptyset$
3     send ($master$, EventJoin $self$)
4     send ($master$, EventAskForNode $self$)
5     repeat
6         match InputEvent() with
7          | EventPreds $\mathcal{P}'$ ->
8            $\mathcal{P}$ := $\mathcal{P}'$
9            $Depth$ := $\lambda x.0$
10           $Parent$ := $Reach$ := $\emptyset$
11         | EventDoNode $(m, d)$ ->
12           for each $\tau \in \mathcal{T}$ do
13               $n$ := $\alpha^{\mathcal{P}}(post(\tau, m))$
14               AddNode("worker", $n$, $d + 1$, $m$, $\tau$)
15           send ($master$, EventAskForNode $self$)
   end
```

Fig. 4. Procedure MainWorker.

in the tree (again, with a lower depth), then it is added to the tree (lines 4–7). The last step of AddNode involves putting the recently added node to the work queue. As the work queue is maintained by the master, the worker has to send the abstract state to the master for further processing (line 15), while the master directly adds it to its queue (line 10–14). If there is an idle worker, the master immediately sends the state to such a worker for processing (lines 10–12).

Function SelectCounterexample Figure 6 presents the implementation of the function SelectCounterexample. This function deterministically selects one counterexample from the reachability tree.

SelectCounterexample generates all the counterexample paths, and then selects the minimal path according to total order $\prec$. It works in a breadth-first way and traverses the reachability tree backwards from the reachable error states.

First, SelectCounterexample initializes the paths list with the reachable error states (line 1) and then expands these paths iteratively by following the parent relation (lines 4–7). Then, when all shortest counterexample paths are found (i.e., when the start state is first reached), the loop terminates (lines 8–9).

```
procedure AddNode
input
    $mode \in \{$“master”, “worker”$\}$ - execution mode
    $n \in \Sigma^{\#}$ - node to add
    $d \in \mathbb{N}$ - depth of node $n$
    $m \in Reach$ - parent of node $n$
    $\tau \in \mathcal{T}$ - transition from $m$ to $n$
begin
    $Reach := Reach \setminus \{n' \in Reach \mid n' = n \wedge Depth(n') > d \vee$
                                   $n' \subset n \wedge Depth(n') \geq d\}$
    if $n \in Reach \wedge Depth(n) = d$ then
        $Parent := \{(m, \tau, n)\} \cup Parent$
    else if $\forall n' \in Reach : \neg(n \subseteq n' \wedge d \geq Depth(n'))$ then
        $Reach := \{n\} \cup Reach$
        $Parent := \{(m, \tau, n)\} \cup Parent$
        $Depth := Depth[n \mapsto d]$
        match $mode$ with
         | “master” ->
            if $IdleWorkers \neq \emptyset$ then
                $w :=$ pick and remove one from $IdleWorkers$
                send ($w$, EventDoNode ($n, d$))
            else
                $Queue :=$ add $n$ to $Queue$
         | “worker” -> send ($master$, EventReachedNode ($n, d, m, \tau$))
end
```

Fig. 5. Procedure AddNode.

The set of paths in *ErrorPaths* is then guaranteed to include counterexample paths of equal, shortest length. Finally, the minimal counterexample wrt. $\prec$ is selected from the list and returned to the caller.

Correctness In order to state the correctness of our algorithm, we present the following statements.

Invariant 1 *A node n with depth d cannot be subsumed by a larger node n' with depth d' if $d < d'$.*

Lemma 1 SelectCounterexample *is deterministic.*

Proof. (Sketch) As the algorithm uses a total ordering function and it works only with data that is independent of the execution (transition identifiers), the algorithm is deterministic.

Lemma 2 *Workers are deterministic.*

Proof. (Sketch) First, assuming no messages are lost, every worker updates its set of predicates when an iteration begins. Second, the computation done by a

```
function SelectCounterexample
vars
    $PathsSofar, ErrorPaths \subseteq \Sigma^{\#} \times \mathcal{T}^*$ - partial counterexamples with start state
begin
    $PathsSofar := \{(n, \epsilon) \mid n \in Reach \wedge s_{\mathcal{E}} \in n\}$
    $ErrorPaths := \emptyset$
    do
        for each $(n, \pi) \in PathsSofar$ do
            $PathsSofar := PathsSofar \setminus \{(n, \pi)\}$
            for each $(m, \tau, n) \in Parent$ such that $m \in Reach$ do
                $PathsSofar := \{(m, \tau \cdot \pi)\} \cup PathsSofar$
        $ErrorPaths := \{(n, \pi) \in PathsSofar \mid s_{\mathcal{I}} \in n\}$
    while $ErrorPaths = \emptyset$
    return $\min(\prec, ErrorPaths)$
end
```

Fig. 6. Function SelectCounterexample.

worker is itself deterministic, which depends only on the input state and the predicates set.

Theorem 1 *Every execution of the* MainMaster *algorithm on a given program computes the same fixpoint in the same number of iterations.*

Proof. (Sketch) First, the AddNode procedure ensures that every subsumed node in the tree gets pruned and every non-subsumed node gets added (w.r.t. Invariant 1), disregarding the order that the input is given. Second, MainMaster ensures that all non-subsumed nodes up to the lowest height are computed. Therefore all shortest counterexamples are found in an iteration. Finally, by Lemmas 1 and 2 and the previous observations, we have that the algorithm picks counterexamples deterministically.

5 Experiments

In this section we describe our experiments with an implementation of our distributed algorithm as an extension of the model checker ARMC [17].

Benchmarks For the experimental evaluation we used a set of benchmarks from the transportation domain [1] and a standard hybrid system example, gasburner. Our benchmarks are automata-theoretic models compiled from complex specifications that describe communication, timing and data manipulation aspects and are represented using a combination of CSP, Duration Calculus, and Object-Z. Table 1 provides some details about the size of the benchmarks.

As a baseline for our evaluation we used the results obtained by executing the sequential model checker ARMC. As Table 2 demonstrates, our benchmarks are difficult (for the sequential algorithm) and require verification time in order of hours.

Test	Program size		# Predicates	Max. length of counterexamples	Refinement iterations
	# Variables	# Transitions			
larger_scale1_lb	16	6127	154	15	32
larger_scale1_ub	16	6127	217	25	46
scale1_lb	15	3337	98	15	20
scale1_ub	15	3337	182	25	37
timing	47	99093	17	17	14
gasburner	19	3124	209	41	64
rtall_tcs	20	18757	50	15	30

Table 1. Size of benchmark programs and details their verification in sequential setting.

Test	Running time, hours				Speedup		
	Sequential	Distributed					
		10 nodes	20 nodes	40 nodes	10 nodes	20 nodes	40 nodes
larger_scale1_lb	1.8	2.9	1.4	0.7	0.62	1.33	2.66
larger_scale1_ub	50.9	9.8	4.6	2.3	5.18	11.06	22.13
scale1_lb	0.3	0.2	0.1	0.1	1.15	2.37	4.24
scale1_ub	5.9	2.7	1.3	0.6	2.17	4.67	9.24
timing	0.7	0.1	0.1	0.04	5.64	10.95	16.73
gasburner	5.5	1.1	0.6	0.3	4.75	9.90	18.35
rtall_tcs	1.0	0.1	0.1	0.04	9.44	19.13	28.76
Median					4.75	9.90	16.73

Table 2. Running time, in hours, for the sequential (BFS) and distributed cases for 10, 20, and 40 nodes, together with the speedup.

Setup The tests were run on a cluster of AMD Opteron 252 (2.6 Ghz) machines, with 3 GB of RAM, 64 KB of L1 caches, and 1 MB of L2 cache each, running Linux kernel version 2.6.24. The computers were connected through a gigabit LAN with an average round-trip time (RTT) of 0.14 ms.

Our implementation of the distributed model checker was compiled using the SICStus Prolog compiler 4.0.5 and the DAHL distribution framework [14].

We verified our benchmarks in five configurations using 5, 10, 20, and 40 compute nodes. Each of the experiments was executed three times to investigate the predictability of our distributed algorithm.

Results Our experiments show encouraging results, see Table 2. We observe a reduction in order of magnitude of the running time, which decreased hours to minutes. The speedup – ratio between the running times in the sequential and distributed setting – is on par with the number of utilized compute nodes and grows linearly with the number of worker nodes.

Besides the overall positive outcome of the running time evaluation, the experiments also show that our implementation can still be improved. We observe that the efficiency of our algorithm – the ratio between the speedup and the number of compute nodes – is between 40 and 50%, which leaves significant space for improvement. We also observed that the difference between running times of the same example across multiple runs is below 1–3%, which indicates the predictability of our algorithm.

References

1. Automatic Verification and Analysis of Complex Systems (AVACS). http://www.avacs.org.
2. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI*, 2001.
3. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, 2000.
4. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, 1977.
5. H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. In *SPIN*, 2001.
6. S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, 1997.
7. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from Proofs. In *POPL*, 2004.
8. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL*, 2002.
9. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits. In *CAV*, 2000.
10. G. J. Holzmann, R. Joshi, and A. Groce. Tackling Large Verification Problems with the Swarm Tool. In *SPIN*, 2008.
11. S. K. Jha. d-IRA: A Distributed Reachability Algorithm for Analysis of Linear Hybrid Automata. In *HSCC*, 2008.
12. R. Jhala and R. Majumdar. Software Model Checking. *ACM Computing Surveys*, 41(4), Oct. 2009.
13. F. Lerda and R. Sisto. Distributed-Memory Model Checking with SPIN. In *SPIN*, 1999.
14. N. P. Lopes, J. A. Navarro, A. Rybalchenko, and A. Singh. Applying Prolog to Develop Distributed Systems. *Theory and Practice of Logic Programming*, 10(4–6):691–707, July 2010.
15. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer, 1995.
16. D. Monniaux. The Parallel Implementation of the Astrée Static Analyzer. In *APLAS*, 2005.
17. A. Podelski and A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *PADL*, 2007.
18. T. Prabhu, S. Ramalingam, M. Might, and M. Hall. EigenCFA: Accelerating flow analysis with GPUs. In *POPL*, 2011.
19. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. In *VMCAI*, 2007.
20. U. Stern and D. L. Dill. Parallelizing the Murϕ Verifier. *Formal Methods in System Design*, 18(2):117–129, Mar. 2001.
21. A. Venet and G. Brat. Precise and Efficient Static Array Bound Checking for Large Embedded C Programs. In *PLDI*, 2004.