Distributed model-checking and counterexample search for CTL logic

Mohand Cherif Boukala*
LSI, Computer Science Department, USTHB,
BP 32 El-Alia, Algiers, Algeria
E-mail: boukala@lsi-usthb.dz
*Corresponding author

Laure Petrucci
LIPN, CNRS UMR 7030, Université Paris XIII,
99, avenue Jean-Baptiste Clément,
F-93430 Villetaneuse, France
E-mail: Laure.Petrucci@lipn.univ-paris13.fr

Abstract: In this paper, we propose a distributed algorithm for CTL model-checking and a counterexample search whenever the CTL formula is not satisfied. The distributed approach is used in order to cope with the state space explosion problem. A cluster of workstations performs collaborative verification over a partitioned state space. Thus, every process involved in the distributed verification performs the labelling procedure on its own partial state space, and uses the parse tree of the CTL formula to evaluate sub-formulas and delay the synchronisations so as to minimise idle time. A counterexample search consists in a distributed construction of the tree-like corresponding to the failure executions. Some experiments have been carried out to evaluate the efficiency of this approach.

Keywords: verification; CTL; distributed model checking; counterexamples.

Reference to this paper should be made as follows: Boukala, M.C. and Petrucci, L. (xxxx) 'Distributed model-checking and counterexample search for CTL logic', Int. J. Critical Computer-Based Systems, Vol. X, No. Y, pp.000–000.

Biographical notes:
1 Introduction

The design of critical systems requires checking properties to ensure a high-level of reliability. Formal specification and verification is a means towards that goal. Model-checking is a successful verification method based on reachability analysis (McMillan, 1998; Bérard et al., 2001; Burch et al., 1992; Holzmann, 2003; Clarke et al., 1986). However, it often encounters the state space explosion problem when dealing with large and complex systems.

Methods to cope with this problem use compact representations and symbolic model-checking (Bryant, 1992), symmetries (Bérard et al., 2001), on-the-fly verification (Bérard et al., 2001),... Another approach consists in taking advantage of a distributed environment. The idea is to increase the computational power and more specifically a large available memory by using a cluster of computers. The use of networks of computers can provide the resources required to achieve verification of industrial case studies.

The important feature of distributed algorithms is to solve a given problem by distributing the data among the participating stations with a small amount of coordination. One of the main issues in distributing model-checking algorithms is the partitioning of the state space among the individual computers. Distributed algorithms for state space generation from high-level models, such as Petri nets, using clusters or parallel computers, have recently been proposed (e.g., Kumar and Mercer, 2004; Garavel et al., 2001; Ciardo et al., 1998).

In the same context, several works proposed to distribute the model-checking of linear temporal logic (LTL) (Lerda and Sisto, 1999; Barnat et al., 2003a, 2003b; Brim et al., 2003, 2004), consisting in a distributed search of accepting cycles in Büchi automata. Distributed and parallel model checking of branching time logic (CTL) (Brim et al., 2002; Bell and Haverkort, 2005; Bourahla, 2005), and modal $\mu$-calculus (Bolling et al., 2002) were also studied. In Boukala and Petrucci (2007), we presented a distributed verification of reachability, liveness and home state properties.

Here, we focus on enumerative distributed model-checking for CTL logic, performed in a backward manner as in Bell and Haverkort (2005). We propose a distributed search of a counterexample whenever the formula is not satisfied, based on the construction of the tree-like structure (Clarke et al., 2002) corresponding to the sub-graph where the negation of the formula holds. The paradigm used is distributed computing based on message passing on a MIMD architecture with distributed memory.

The paper is organised as follows. In Section 2, we shortly recall CTL basics. Then, Section 3 describes the distributed setting we use for computing and storing state spaces.
Section 4 presents the approach for CTL model-checking in this distributed framework. The counterexample search in such a distributed environment is presented in Section 5. An implementation of our algorithms led to experimental results discussed in Section 6.

2 The computation tree logic CTL

CTL was introduced by Clarke et al. (1986), and is commonly defined as a branching time logic. It is based on models where at each moment there may be several different possible futures. The syntax of CTL formulas is given in Definition 1, where the most elementary expressions are atomic propositions. The set of atomic propositions is denoted by \( AP \) with elements \( p, q, r \ldots \)

**Definition 1 (CTL syntax):** The set of CTL formulas is inductively defined as follows:

- \( p \in AP \) is a CTL formula, where \( AP \) is a set of atomic propositions.
- If \( \varphi \) and \( \psi \) are CTL formulas, then: \( \neg \varphi \), \( \varphi \land \psi \) and \( \varphi \lor \psi \) are CTL formulas.
- If \( \varphi \) and \( \psi \) are CTL formulas, then: \( EX \psi \), \( E(\psi U \phi) \) and \( A(\psi U \phi) \) are CTL formulas.

\( E \) and \( A \) are the existential and universal path operators. They express that a property is valid for some path and for all paths, respectively. \( X \) and \( U \) are the linear temporal operators, called respectively Next and Until operators. They must be in the scope of an existential or universal path operator.

The interpretation of the CTL logic formulas is defined over a Kripke structure:

**Definition 2 (Kripke structure):** A Kripke structure is a tuple \( M = (S, T, \lambda, s_0) \) where:

- \( S \) is a non-empty set of states
- \( T \subseteq S \times S \) associates each state \( s \in S \) with its possible successors
- \( \lambda : S \rightarrow 2^{AP} \), associates with each state \( s \in S \) the set of atomic propositions \( \lambda(s) \) valid in \( s \)
- \( s_0 \in S \) is the initial state.

The properties search in this structure will involve paths:

**Definition 3 (Paths):** A path is an infinite sequence of states \( \sigma = s_0s_1s_2 \cdots \in S^\omega \) such that \( \forall i \geq 0; (s_i, s_{i+1}) \in T \).

The \((i+1)\)th element of \( \sigma \) is denoted by \( \sigma[i] \).

The set of paths starting in state \( s \) is \( P_M(s) = \{ \sigma \in S^\omega | \sigma[0] = s \} \).

The semantics of CTL formulas is then defined as follows.

**Definition 4 (CTL semantics):** Let \( p \in AP \) be an atomic proposition, \( M = (S, T, \lambda, s_0) \) a Kripke structure, \( s \in S \), and \( \varphi, \psi \) CTL-formulas. The satisfaction relation \( \models \) is inductively defined by:

- \( s \models p \iff p \in \lambda(s) \)
- \( s \models \neg \varphi \iff \neg(s \models \varphi) \)
The interpretations for atomic propositions, negation and conjunction are as usual. \( EX \varphi \) is valid in state \( s \) if and only if there exists some path \( \sigma \) starting in \( s \) such that in the next state of this path (state \( \sigma[1] \)), the property holds.

\( A[\varphi \cup \psi] \) is valid in state \( s \) if and only if every path starting in \( s \) has an initial finite prefix such that \( \psi \) holds in the last state of this prefix and \( \varphi \) holds at all other states along the prefix.

\( E[\varphi \cup \psi] \) is valid in \( s \) is similar but only requires the existence of a path.

It is said that \( M \models \varphi \) iff \( s_0 \models \varphi \).

Some abbreviations are also often defined and used in the literature: EF, AF, EG, AG and AX, but they can be expressed through the operators defined above.

A model-checking algorithm for CTL logic consists in computing, for a given formula \( \varphi \) and a model \( M \), the subset: \( \text{Sat}(\varphi) = \{ s \in S \mid s \models \varphi \} \) in an iterative way.

The problem of checking if \( s \models \varphi \) is then reduced to checking if \( s \in \text{Sat}(\varphi) \). \text{Sat} is constructed inductively using the formula parse tree (see e.g., Clarke et al., 1986; Inggs and Barringer, 2006).

### 3 Distributed state space exploration

Among the approaches to render verification amenable, distributed state space exploration consists in splitting the state space on different machines so as to take advantage of their memory.

The partitioning of state space is a crucial issue. Compact states encodings, load balancing policies (Kumar and Mercer, 2004; Lerda and Sisto, 1999; Garavel et al., 2001), and similar approaches (Caselli et al., 1994; Ciardo et al., 1998) yield very good speedups.

The idea underlying a distributed algorithm for state-space generation is to use multiple processes to perform the exploration concurrently on distinct parts of the state space. Let us assume there are \( N \) machines connected by a network and communicating by message passing. To distribute the states among the different machines, we use a hash function \( h : S \rightarrow \{0, \ldots, N - 1\} \). Then \( h(s) \) is the owner of state \( s \), i.e., the process responsible for storing \( s \) and exploring its successors. This defines a partition of \( S \) into \( N \) sets: \( S_i = \{ s \mid h(s) = i \} \) for \( i = 0, \ldots, N - 1 \).

To be efficient, the hash function should provide a uniform partitioning and minimum cross arcs. A cross arc is an arc where the successor \( s' \) of state \( s \) does not belong to the same station as state \( s \). A large number of cross arcs necessarily leads to additional communications.

As in Kristensen and Petrucci (2004), we used two types of processes: A coordinator and \( N \) workers processes. The coordinator, which is a very light process, can run on the same station as any worker process. It initiates the generation, by sending the initial state
Distributed model-checking and counterexample search for CTL logic

$s_0$ to the worker process $h(n_0)$, and detects termination of the overall computation. Workers execute the same code, a worker process $i$ is executed by station $i$, to generate the subset of states $S_i$ owned by this station from the input high-level formalism (e.g., Petri nets). In the following, we describe only the algorithms executed by the workers processes, for both state space generation and verification. The generation algorithm executed by a machine $i$ (workeri) is briefly described in Algorithm 1.

Algorithm 1 DistGeneration($i$)

```plaintext
1 begin
2 input: A Petri net $N$
3 /* Process states in Waiting */
4 while Waiting $\neq \emptyset$ do
5     Choose $s \in$ Waiting
6     $S_i \leftarrow S_i \cup \{s\}$
7     for all the $s'$ such that $s' \in \text{succ}(s)$ do
8         /* $s'$ is a successor of $s$ */
9         if $h(s') \neq i$ then
10            /* New state $s'$ does not belong to this process */
11               Send ($h(s')$, $s'$)
12         else
13             /* $s'$ belongs to process $i$ */
14               Node ($s'$)
15               Arc ($s$, $s'$)
16     Waiting $\leftarrow$ Waiting \ $\{M\}$
17 end
```

In the algorithm, Waiting contains the states which are not explored yet. At each iteration an element $s$ of Waiting is removed from the set and its successors are computed. Node($s'$) adds local successors to Waiting if they have not been explored yet, while distant successors are sent to their owner process. Arc($s$, $s'$) adds the arc $(s, s')$ to the set of arcs.

Arcs can be stored using either a successors or a predecessors list, and every successor is marked by its owner process identification, in particular for distant successors.

The communications are asynchronous, thus, when receiving a message, the processes are preempted and the handler function MessHandler() is executed. The excerpt of this function, shown in Algorithm 2, indicates the operations executed when a state is received by station $i$.

The main procedure basically contains a waiting loop which stops when a termination message is received from the coordinator process. The DistGeneration($i$) procedure is invoked when the process receives its first marking. Messages vehicle states (markings) or other information ensuring the termination of the distributed algorithm. They are all
taken care of by the same procedure \texttt{MessageHandler}(i). The termination occurs when all processes have finished handling their states, and no message is in transit.

\begin{algorithm}
\begin{algorithmic}[1]
\STATE \textbf{begin}
\STATE ... \\
\STATE \textbf{case} \texttt{Message.Type} = \texttt{STATE} \\
\STATE \hspace{1em} \texttt{s} \leftarrow \texttt{Message.state} \\
\STATE \hspace{1em} \texttt{V}_i \leftarrow \texttt{V}_i \cup \{\texttt{s}\} \\
\STATE \hspace{1em} \texttt{DistGeneration}(i) \\
\STATE ... \\
\STATE \textbf{end}
\end{algorithmic}
\end{algorithm}

4 Distributed model checking of CTL formulas

In this section, we present algorithms to verify CTL formulas on a distributed state space. We consider the same architecture as for the state space generation, based on a coordinator and workers processes. We suppose that the state space is entirely partitioned over all the stations of the cluster, each station (process) \( i \) owns the set of states \( S_i = \{s \mid h(s) = i\} \), and that each process previously constructed the parse tree of the CTL formula (Clarke et al., 1986).

Each process computes the set of local states \( S_{\text{Sat}}(\varphi) \) which satisfy formula \( \varphi \), such that:

\[
S_{\text{Sat}}(\varphi) = \bigcup_{i=0}^{N-1} S_{\text{Sat}}(i,\varphi).
\]

Propositional sub-formulas, atomic propositions, negation and conjunction can be verified locally by each process. However, when the CTL formulas includes the temporal operators \( \text{EX}, \text{AX}, \text{EU} \) or \( \text{AU} \), communications between processes to check the non-local neighbouring states become necessary. We consider only the operators \( \text{EX}, \text{EU} \) and \( \text{AU} \) which cover all the CTL formulas.

To compute \( S_{\text{Sat}}(\varphi) \) for a formula \( \varphi \) containing sub-formulas, we proceed inductively by labelling with \( \varphi \) the states satisfying \( \varphi \). We assume that the sets of local states satisfying these sub-formulas are already computed (labelled). This is always the case when the parse tree is used.

4.1 Propositional CTL formulas

When \( \varphi \) is a propositional CTL formula, all the processes compute independently their set of local states \( S_{\text{Sat}}(\varphi) \) which satisfy \( \varphi \) using Algorithm 3.

Since there is no need for communications to perform the distributed checking of logical formulas, we can expect a linear speedup.
Algorithm 3  \textit{Sat}(\varphi)

\begin{align*}
&\textbf{begin} \\
&\quad \text{Sat}(\varphi) \leftarrow \emptyset \\
&\quad \text{\textbf{switch} } \varphi \text{ \textbf{do}} \\
&\quad \quad \text{\textbf{case} } p: \\
&\quad \quad \quad \text{\textbf{for each} } s \in S_i \text{ \textbf{do}} \\
&\quad \quad \quad \quad \text{if } p \in \lambda(s) \text{ then Sat}(\varphi) \leftarrow \text{Sat}(\varphi) \cup \{s\} \\
&\quad \quad \text{\textbf{case} } \neg \varphi_1: \\
&\quad \quad \quad \text{\textbf{for each} } s \in S_i \text{ \textbf{do}} \\
&\quad \quad \quad \quad \text{if } s \notin \text{Sat}(\varphi_1) \text{ then Sat}(\varphi) \leftarrow \text{Sat}(\varphi) \cup \{s\} \\
&\quad \quad \quad \text{\textbf{case} } \varphi_1 \land \varphi_2: \\
&\quad \quad \quad \text{\textbf{for each} } s \in S_i \text{ \textbf{do}} \\
&\quad \quad \quad \quad \text{if } s \in \text{Sat}(\varphi_1) \land \text{Sat}(\varphi_2) \text{ then Sat}(\varphi) \leftarrow \text{Sat}(\varphi) \cup \{s\} \\
&\quad \end{align*}

4.2 \textit{neXt} CTL formulas: \(\psi = \text{EX} \varphi\) and \(\psi = \text{AX} \varphi\)

Assume that the local set \text{Sat}(\varphi) is known. To compute the set of states satisfying \(\psi = \text{EX} \varphi\), each machine executes Algorithm 4.

Algorithm 4  \textit{Sat}(\psi = \text{EX} \varphi) 

\begin{align*}
&\textbf{begin} \\
&\quad \text{Sat}(\psi) \leftarrow \emptyset \\
&\quad \text{\textbf{for each} } s \in \text{Sat}(\varphi) \text{ \textbf{do}} \\
&\quad \quad \text{\textbf{for each} } s' \in \text{predecessor}(s) \text{ \textbf{do}} \\
&\quad \quad \quad \text{if } h(s') = i \text{ then} \\
&\quad \quad \quad \quad \text{Sat}(\psi) \leftarrow \text{Sat}(\psi) \cup \{s'\} \\
&\quad \quad \quad \text{else} \\
&\quad \quad \quad \quad \text{Send } (h(s'), < \text{Sat}_{\text{EX}}, \psi, s'>) \\
&\quad \end{align*}

Every process \(i\) adds each local predecessor \(s'\) of the states which satisfy \(\varphi\) to \text{Sat}(\varphi), and sends messages to the processes owning distant predecessors to do so. When a process \(j\) receives a state \(s\), it adds it to \text{Sat}(\varphi) (see 7, lines 3 to 4). There is no need for the process \(j\) to reply to process \(i\).
Since we proceed in a backward manner, it is not necessary to wait until the sets $\text{Sat}_j(\phi)$ are known in other processes $j \neq i$. Thus, the processes can work independently and in a parallel way. Hence, we can also expect a linear speedup.

The satisfaction of the formulas $AX\phi$ can be obtained by checking $\neg EX\neg \phi$.

### 4.3 Exist until formulas: $\psi = E(\phi_1 \cup \phi_2)$

To evaluate the CTL formula $\psi = E(\phi_1 \cup \phi_2)$, we assume that $\text{Sat}_i(\phi_1)$ and $\text{Sat}_i(\phi_2)$ are known for each station $i$. This means that processes must wait for these sets to be known in all processes.

To compute the set $\text{Sat}_i(\psi)$, each process executes Algorithm 5.

**Algorithm 5** $\text{Sat}_i(\psi = E(\phi_1 \cup \phi_2))$

1. begin
2. $\text{Sat}_i(\psi) \leftarrow \text{Sat}_i(\phi_2)$
3. $\text{Snew}_i \leftarrow \text{Sat}_i(\phi_2)$
4. while $\text{Snew}_i \neq \emptyset$ do
   5. Choose $s \in \text{Snew}_i$
   6. $\text{Snew}_i \leftarrow \text{Snew}_i \setminus \{s\}$
   7. for each Predecessor $s'$ of $s$ do
      8. if $h(s') = i$ then
         9. if $s' \in \text{Sat}_i(\phi_1) \setminus \text{Sat}_i(\psi)$ then
            10. $\text{Snew}_i \leftarrow \text{Snew}_i \cup \{s'\}$
            11. $\text{Sat}_i(\psi) \leftarrow \text{Sat}_i(\psi) \cup \{s'\}$
         else
            12. Send $(h(s'), <\text{Sat}_i, \psi, s'>)$
   end
5. end

A set $\text{Snew}_i$ is used to contain the new states which satisfy $\psi$. Initially $\text{Snew}_i$ contains $\text{Sat}_i(\phi_2)$. At each iteration a state $s$ is removed from $\text{Snew}_i$, and each of its predecessors $s' \notin \text{Sat}_i(\psi)$ satisfying $\phi_1$, then satisfies $\psi$, is then included in $\text{Snew}_i$ to check whether its predecessors also satisfy $\phi_1$. A message is sent to the owner processes of distant predecessors. When a process $j$ receives a such message, it adds the state to $\text{Snew}_j$ and to $\text{Sat}_j(\psi)$ (see 7 lines 8 to 13).

### 4.4 All until CTL formulas: $\psi = A(\phi_1 \cup \phi_2)$

Similar ideas are used to compute the set of states which satisfy AU formulas. $\text{Sat}_i(\phi_1)$ and $\text{Sat}_i(\phi_2)$ being known for all processes, $\text{Sat}_i(\psi)$ is computed for each process $i$.

For each element $s \in \text{Snew}_i$, this algorithm still uses the same verification techniques. At each iteration state $s$ is removed from $\text{Snew}_i$, and for each of its predecessors $s' \notin \text{Sat}_i(\psi)$ satisfying $\phi_1$, the number of successors satisfying $\psi$ $\text{HoldSuccNb}(s')$ is
incremented; $s'$ is added to both $\text{Sat}(\psi)$ and $\text{Snew}_i$ whenever $\text{HoldSuccNb}(s')$ is equal to its out-degree. It then satisfies $\phi_1$, is included in $\text{Snew}_i$ to check whether its predecessors also satisfy $\phi_1$. Synchronisations are necessary while the processes evaluate the sub-formulas. Thus, they wait to synchronise with other processes. The synchronisations happen only before the $\textit{until}$ sub-formulas verification. This allows for minimising the idle time of processes.

**Algorithm 6 $\text{Sat}(\psi = A(\phi_1 \cup \phi_2))$**

1. begin
2. $\text{Sat}(\psi) \leftarrow \text{Sat}(\phi_2)$
3. $\forall s \in S_i, \text{HoldSuccNb}(s) \leftarrow 0$
4. $\text{Snew}_i \leftarrow \text{Sat}(\phi_2)$
5. for each $s \in \text{Snew}_i$ do
6.     $\text{Snew}_i \leftarrow \text{Snew}_i \setminus \{s'\}$
7.     for each $s' \in \text{Predecessor}(s)$ do
8.         if $h(s') = i$ then
9.             if $s' \in (\text{Sat}(\phi_1) \setminus \text{Sat}(\psi))$ then
10.                $\text{HoldSuccNb}(s') \leftarrow +$
11.                if $\text{HoldSuccNb}(s') = \text{OutDegree}(s')$ then
12.                    $\text{Snew}_i \leftarrow \text{Snew}_i \cup \{s'\}$
13.                    $\text{Sat}(\psi) \leftarrow \text{Sat}(\psi) \cup \{s'\}$
14.          else
15.            Send $(h(s'), <\text{Sat}_{\text{id}}, \psi, s'>)$
16.     end
17. end

4.5 Specific messages

Communications are required when successors and predecessors are not handled by the same process. Hence, function $\text{MessHandler}()$ is extended, as described in Algorithm 7, so as to handle the messages concerning the formulas evaluation.

The termination of the verification of a formula $\psi$ occurs when the initial state is added to $\text{Sat}(\psi)$ or when all processes end the labelling of their own states. In the latter case, the formula is not satisfied, and we propose a distributed search of the counterexample.
Algorithm 7 MessageHandler(i)

begin
... case Message.Type = SatEX
Sat(ψ) ← Sat(ψ) ∪ \{Message.state\}
case Message.Type = SatAX
s ← Message.state
HoldSuccNb(s) + +
if HoldSuccNb(s) = OutDeg(s) then
Sat(ψ) ← Sat(ψ) ∪ \{s\}
case Message.Type = SatEU
s ← Message.state
if ((s ∉ Sat(ψ) ∧ s ∈ Sat(ϕ1)) then
Snewi ← Snewi ∪ \{s\}
Sat(ψ) ← Sat(ψ) ∪ \{s\}
case Message.Type = SatAU
s ← Message.state
if (s ∉ Sat(ψ) ∧ s ∈ Sat(ϕ1)) then
HoldSuccNb(s) + +
if HoldSuccNb(s) = OutDegree(s) then
Snewi ← Snewi ∪ \{s\}
Sat(ψ) ← Sat(ψ) ∪ \{s\}
end

5 Distributed counterexample search

The model-checking algorithm either terminates with success, if the property holds, or with failure otherwise. In this latter case, a counterexample provides the designer with the reasons for the failure and can be helpful to find subtle errors in the system. A large class of temporal logics admit counterexamples as simple tree-like structures (Clarke et al., 2002). It consists in a structure which shows that the negation of the formula is true.

Here, we propose to determine a counterexample from the distributed state space, allowing to use explicit state enumeration techniques on very large models, by exploiting resources of multiple stations of a cluster.

In this section, we focus on determining the counterexample for a formula using the temporal operators $AU$ and $EU$ since they are the most challenging cases.
The counterexample search is done in a depth first manner, starting from the initial state \( s_0 \).

It is easy to determine the counterexample in the case of a formula \( \psi = AX\varphi \) when it does not hold, there exists a successor \( s \) of \( s_0 \) such that \( s \not\models \varphi \). Thus, the counterexample is the sub-graph containing only \( s_0 \) and its successors not satisfying \( \varphi \).

However, for the both formulas: \( A(\varphi_1 \cup \varphi_2) \) formula and \( E(\varphi_1 \cup \varphi_2) \), the corresponding counterexample consists in all the paths \( \pi = s_0s_1 \ldots \) such that:

\[
\exists i \geq 0 \land s_i \models (\neg\varphi_1 \land \neg\varphi_2) \land \forall j (0 \leq j < i \Rightarrow s_j \not\models (\varphi_1 \land \neg\varphi_2))
\]

or

\[
\exists i \geq 0 \land \exists j (j > 0 \land s_i = s_{i+j} \land \forall k (0 \leq k < j \Rightarrow s_k \models (\varphi_1 \land \neg\varphi_2)))
\]

The first condition means that \( \pi \) contains a succession of states \( s_0, s_1, \ldots, s_{i-1} \) where \( \varphi_1 \) holds, and in \( s_i \) neither \( \varphi_1 \) nor \( \varphi_2 \) hold.

In the second, the path \( \pi \) contains a cycle where \( \varphi_1 \) holds but not \( \varphi_2 \). It is then possible to execute the cycle infinitely often, thus never reaching a state where \( \varphi_2 \) holds.

To exhibit the counterexample (Algorithm 8), each process uses a stack, \( \text{Waiting} \), containing the initial state \( s_0 \) in the process \( h(s_0) \).

\( \pi \) will contain the local part of the path corresponding to a counterexample. The elements of \( \pi \) are labelled using a global counter allowing the coordinator to reorder the entire counterexample.

**Algorithm 8** CounterExpleSearch()

```plaintext
begin
  break ← false
  repeat
    s ← pop (Waiting)
    if s.type = Backtrack then
      Send (s.sender, < Backtrack >)
      break ← true
    else
      NewSucc ← false
      push (s, \pi) /* Adds state s to the top of \pi */
      for each s' ∈ successors(s) do
        if h(s') = i then
          if s' ∈ \pi
            then
              Send (Coordinator, \pi)
          else
            if s' ∉ Sat(\varphi_1 \lor \varphi_2) then
              Send (Coordinator, \pi)
```
if $s' \not\in \text{Sat}(\varphi_1 \wedge \neg \varphi_2)$ then
  push ($s'$, Waiting)
  NewSucc $\leftarrow$ true

else
  Send ($h(s')$, $s'$, CExample)
  break $\leftarrow$ true
  \text{This process will stop and process $h(s')$ will continue the CE search}

if successors ($s$) = 0 then
  Send (Coordinator, $\pi$)
  if $\neg$ NewSucc then pop ($\pi$) /* removes the state on the Top of $\pi$ */

until ((Waiting $= 0$) $\lor$ break)

Note: Send the part of the path or cycle constituting the counterexample to the coordinator which reorders the states to display the entire counterexample.

The Waiting stack can contain either states or backtrack tags. The states are those in which $\varphi_1$ holds but not $\varphi_2$. The backtrack tags are used when the backtrack leads to a distant predecessor.

At each iteration, the state $s$ at the top of the stack Waiting is extracted and added to $\pi$, and each local successor $s'$ of $s$ can be in one of the following cases:

- $s'$ satisfies $\varphi_1$ but not $\varphi_2$. Then $s'$ is added to the stack Waiting extending the path which could be a counterexample.
- $s'$ belongs to $\pi$. This means that $\pi$ contains a cycle whose states satisfy $\varphi_1$ but not $\varphi_2$. $\pi$ is then a counterexample for the formula $\psi$.
- $s'$ does not satisfy $\varphi_1$ nor $\varphi_2$. Then $\pi$ constitutes a counterexample.

For the distant successors, messages are sent to their owners to handle them. If $s$ has no successor, or no successor satisfying either $\varphi_1$ or $\varphi_2$, then $\pi$ also constitutes a counterexample (lines 18 and 26).

When the top of the stack contains a backtrack tag, this means that the next state to pop is distant, a message is sent to the owner to continue processing.

The process $h(s_0)$ starts the counterexample search as described in Algorithm 9 when it receives a CounterExample message.

Cycle detection can be performed using depth-first exploration. However, if the exploration is carried out in a parallel way, some cycles might be overlooked. Hence, when a process sends a state, it remains idle until it receives a Backtrack message or receives another state to be visited.
Distributed model-checking and counterexample search for CTL logic

Algorithm 9  MessageHandler()

1  begin
2    ...
3    case Message.Type = CExample
4        push (Waiting, Backtrack)
5        push (Waiting, Message.State)
6        CounterExempleSearch()
7    case Message.Type = Backtrack
8        CounterExempleSearch()
9    ...
10   end

6  Implementation and experimental results

The previous algorithms were implemented within a prototype tool. Given a Petri net, the reachability graph is generated on a cluster of stations, and then a CTL formula is verified. In this section, we give the results obtained for the dining philosophers problems. Other problems like the distributed database managers were also studied. Whenever the formula is not satisfied, the sub-graph corresponding to the counterexample is entirely computed, and reported to the user. The tests were carried on a cluster composed of 12 stations (Pentium IV with 512 Mbytes of memory).

6.1  The dining philosophers

The dining philosophers problem, introduced by Dijkstra, concerns the problem of resources allocation between processes. Several philosophers sit around a circular table. There is a fork between each pair of neighbouring philosophers. Each philosopher spends his life alternating thinking and eating phases, and may arbitrarily decide to use either the fork to his left or the one to his right, but needs both of them to eat.

The states of the dining philosophers problem can be partitioned into sets $S_1$, $S_2$, $\ldots$ such that $S_i$ does not contain all the states corresponding to $i$ eating philosophers. Indeed, these states are not linked by any transition, they constitute a stable, so it is more interesting to store them on different stations. On the contrary, a state where there are $i$ eating philosophers is only linked to states where there are $i - 1$ and $i + 1$ eating philosophers, such that there are respectively $i - 1$ and $i$ eating philosophers common to both states. Hence, our hash function tries to store them on the same station.

We can note that the number of cross arcs compared to the total number of transitions, given by the ratio $N_1 / N_2$, grows to reach more then 8, which means that there is one traversal transition for more then eight local transitions. The ratio $N_1 / N_2$ shows that the number of messages is approximately 2 messages for each traversal transition. A superlinear speedup is obtained, as shown in the last two columns. This is possible, using a local hash table, since each process explores its own states.
We analysed several formulas which do not hold, e.g., two thinking philosophers can
start eating at the same time,
\[ E[(\text{thinking}(i) \land \text{thinking}(j)) \lor (\text{eating}(i) \land \text{eating}(j))] \]
or two neighbours eating at same time,
\[ EF(\text{eating}(i) \land \text{eating}(i+1 \mod n)) \]
The analysis of these formulas always leads to a failure, and a counterexample search.
The second formula gives only cycles involving the entire state space.
The results obtained for the first formula are given in Tables 1 and 2 where the last
column indicates the computation time with a single process (i.e., not distributed).

### Table 1  Distributed generation

<table>
<thead>
<tr>
<th>Nb Phil</th>
<th>Nb states</th>
<th>Nb trans ((N_1))</th>
<th>Nb cross. arcs ((N_2))</th>
<th>(N_1 / N_2)</th>
<th>Nb Mess. ((N_3))</th>
<th>(N_3 / N_2)</th>
<th>Time G. (sec)</th>
<th>I proc time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>11</td>
<td>30</td>
<td>18</td>
<td>1.67</td>
<td>36</td>
<td>2.00</td>
<td>&lt; 0.01</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>10</td>
<td>123</td>
<td>680</td>
<td>242</td>
<td>2.81</td>
<td>494</td>
<td>2.03</td>
<td>&lt; 0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>15</td>
<td>1,364</td>
<td>11,310</td>
<td>2,731</td>
<td>4.14</td>
<td>5477</td>
<td>2.01</td>
<td>0.06</td>
<td>0.38</td>
</tr>
<tr>
<td>20</td>
<td>15,127</td>
<td>167,240</td>
<td>30,236</td>
<td>5.53</td>
<td>60,493</td>
<td>2.00</td>
<td>2.52</td>
<td>13.48</td>
</tr>
<tr>
<td>25</td>
<td>167,761</td>
<td>2,318,400</td>
<td>315,718</td>
<td>7.34</td>
<td>631,587</td>
<td>2.00</td>
<td>32.58</td>
<td>216.09</td>
</tr>
<tr>
<td>30</td>
<td>1,728,813</td>
<td>28,686,031</td>
<td>3,572,821</td>
<td>8.03</td>
<td>7,156,116</td>
<td>2.00</td>
<td>7,547.45</td>
<td>&gt;&gt;b</td>
</tr>
</tbody>
</table>

Note: >>b in the table means that more than 4 Mbytes of memory are required and the
element could not be processed on a single machine.

### Table 2  Distributed verification and counterexample search

<table>
<thead>
<tr>
<th>Nb Phil</th>
<th>Verif. time (sec)</th>
<th>I proc verif time</th>
<th>Tot. Nb of CE</th>
<th>Paths CE</th>
<th>Mess. Nb 1st CE</th>
<th>Tot. Nb Mess</th>
<th>Max cpu time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>&lt; 0.01</td>
<td>&lt; 0.01</td>
<td>11</td>
<td>4</td>
<td>17</td>
<td>49</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>10</td>
<td>&lt; 0.01</td>
<td>0.01</td>
<td>218</td>
<td>42</td>
<td>24</td>
<td>480</td>
<td>0.01</td>
</tr>
<tr>
<td>15</td>
<td>0.03</td>
<td>0.18</td>
<td>3,502</td>
<td>267</td>
<td>31</td>
<td>5,265</td>
<td>0.12</td>
</tr>
<tr>
<td>20</td>
<td>0.18</td>
<td>0.77</td>
<td>49,917</td>
<td>5,168</td>
<td>49</td>
<td>52,221</td>
<td>3.46</td>
</tr>
<tr>
<td>25</td>
<td>0.84</td>
<td>4.33</td>
<td>868,345</td>
<td>95,375</td>
<td>68</td>
<td>512,614</td>
<td>39.52</td>
</tr>
<tr>
<td>30</td>
<td>5.15</td>
<td>&gt;&gt;b</td>
<td>10,043,424</td>
<td>190,951</td>
<td>182</td>
<td>5,204,315</td>
<td>149.96</td>
</tr>
</tbody>
</table>

Note: >>b in the table means that more than 4 Mbytes of memory are required and the
element could not be processed on a single machine.

The first columns in Table 2 give results concerning formulas verification containing the
EU operator, respectively the maximum CPU time for each process and the CPU time of
a single process verification. Here also, we have an interesting speedup.

The second part of columns concerns the distributed counterexample search. Its first
column gives the total number of counterexamples, the second column the number of
counterexamples which are paths (others are cycles), followed by the number of
messages needed to the distributed search of the first counterexample and the total
number of messages to find all the counterexamples. In the last column is the maximum
CPU time for each process to find all counterexamples.
7 Conclusions

In this paper, we proposed algorithms for CTL distributed model-checking. First, the state space is generated and partitioned over a cluster of stations; this step is achieved by set of machines which collaborate to explore the entire state space. Then, the same set of machines is used to achieve a distributed model-checking of CTL formulas. We have adapted the sequential algorithms used to verify the CTL formulas to the distributed case. This allows for verifying large and more complicated systems using explicit representation of the states. A counterexample, based on a partitioned search, is given whenever the CTL formula is not satisfied. Several tests were performed over a cluster of workstations, a super-linear speed up is obtained for large size problems, also allowing verification of examples not amenable on a single machine.

References


