Towards Distributed Verification of Petri Nets Properties

M.C. Boukala
LSI, Computer Science department, USTHB
BP 32 El-Alia
Algiers, ALGERIA
boukala@lsi-usthb.dz

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

Abstract

The use of distributed or parallel processing gained interest in the recent years to fight the state space explosion problem. Many industrial systems are described with large models, and the state space being even larger, it does not fit completely into the memory of a single computer.
In this approach several computers connected over a network cooperate. The state space is then partitionned among these computers, and each of them contributes to the verification by considering its own subspace.
In this paper, we address the verification of basic behavioural properties: reachability, liveness and home state and their distributed analysis. In particular, the verification of the latter properties requires the generation of the full state space and the computation of its terminal strongly connected components. Here, we propose to use a distributed Tarjan algorithm to perform this computation.
The performance of distributed verification depends on several criteria, e.g. load balancing of the partitionned state space, but also more importantly on a good partitioning. Therefore, choosing an adequate hash function to assign nodes to processors is important.

Keywords: Petri nets, verification, distributed algorithms

1. INTRODUCTION

Systems developped nowadays are both more and more complex and critical. When addressing the design of such systems, it is necessary to ensure reliability, by verifying the system properties. The main approach to verification consists in the generation and analysis of the state space. Some properties such as reachability or deadlocks can be verified on-the-fly, and only require information on states reachable from the initial marking. On the contrary, liveness and home states are elaborate properties which require a full generation of the state space including not only nodes but also arcs. Even for small models, the size of the state space may be too huge to fit in the memory of a single computer.

To cope with the state space explosion problem, several reduction techniques have been proposed: on-the-fly verification checks properties during the state space construction ; sweep-line construction [Sch04] considers a progress measure, and discards states that will not be encountered further in the construction ; modular verification [CP00] ; partial order reductions [Val92, NG97] ; symmetries [AHI98, CFJ93] ; using symbolic or compact representations like BDDs and Kronecker algebra.
Recently, several works addressed distributed verification, many of them focussing on distributed

LTL model-checking [BLS01, BO03, LS99, LN01]. These works are mainly concerned with detecting accepting cycles which correspond to a faulty behaviour.

In this work, we focus on checking usual properties as liveness and home states. These properties are important and complex. They require the full generation of the state space. The verification of the both properties is based on the computation of the terminal strongly connected components (SCC). We proposed a distributed version of the Tarjan algorithm to compute them. The terminal SCC computation is also used to verify many other properties, e.g. the verification of the accepting execution in the street automata [MJ96].

The paper is organised as follows. Basic notions and notations are introduced in section 2: the Petri nets model is described and the necessary background is recalled. In section 3, we present distributed algorithms to generate a partitionned state space. Section 4 describes the most frequently used properties, i.e. reachability, deadlocks, home states and liveness. The distributed algorithms to check these properties, and in particular liveness and home state properties are presented in section 5. A prototype tool implementing these algorithms was designed, and experimental results are presented in section 6. There, a good partitionning of the state space is discussed for two classical problems, i.e. the philosophers and the distributed data base. Section 7 concludes the papers and discusses future work.

2. PRELIMINARIES

The state space generation is the construction of the basic model on which is used for verification. This model is usually a transitions system, composed of a set of nodes which represent the states of the system and a set of edges, which represent transitions between the states.

Systems or programs behaviour are usualy described using high-level description languages such as Petri nets or process algebras, then the state space is generated from the high-level model.

Petri nets [Mur89] are often used to describe and study concurrent systems. It allows for both a graphical representation and a formal (mathematical) semantics.

A Petri net is a directed bipartite graph with two kinds of nodes, places and transistions, and where arcs are either from a place to a transition or from a transition to a place. The graphical representation of places is ellipses, that of transitions, rectangles. Arcs are labelled with weights. Labels for weight 1 are usually omitted. A marking assigns to each place a non-negative integer If a marking assigns to place p a non-negative integer k, we say that p is marked with k tokens, they are represented by k black dots within place p. A marking is denoted by M, a m-vector, where m is the total number of places. The p^{th} component of M, denoted by $M(p)$, is the number of tokens in place p. Formally a Petri net is defined as follows:

Definition 1 (Petri net) *A* Petri net *is a triple,* $\mathcal{N} = \langle P, T, W \rangle$ *where:*

- P *is a finite set of* places *;*
- T *is a finite set of* transitions *such that* $P \cap T = \emptyset$ *and* $P \cup T \neq \emptyset$ *;*
- $W : (P \times T) \cup (T \times P) \longrightarrow \mathbb{N}$ *is a weight function,* $W(p,t)$ *(resp.* $W(t,p)$*) gives the weight of the arc from* p *to* t *(resp. from* t *to* p*).*

Usually, an initial marking is associated with the Petri net:

Definition 2 (Marked Petri net) *A* marked Petri net $(\mathcal{N}, M_0)$*, is a Petri net* $\mathcal{N}$ *with an initial marking* $M_0 : P \longrightarrow \mathbb{N}$*. The initial marking of a place* $p \in P$ *is* $M_0(p)$*.*

From now on, we assume $\mathcal{N} = \langle P, T, W \rangle$ is a Petri net and $M : P \longrightarrow \mathbb{N}$ a marking of $\mathcal{N}$.

The dynamic behaviour of a discrete system is given by its states and their changes. Markings evolve according to the following transition (firing) rule:

Definition 3 (Enabling) *A transition $t \in T$ is* enabled *in marking M iff $\forall p \in P : M(p) \geq W(p,t)$. This is denoted by $M[t\rangle$.*

Definition 4 (Firing rule) *A transition $t \in T$ enabled in marking M can* fire*, leading to a new marking M' such that:*

$$\forall p \in P : M'(p) = M(p) - W(p,t) + W(t,p)$$

This is denoted by $M[t\rangle M'$.

The firing rule is extended to sequences of transitions. If a marking M_n can be obtained from a marking M by firing a sequence of transitions, it is said to be reachable:

Definition 5 (Reachable marking) *A marking M_n is* reachable *from a marking M iff:*

$$\exists \sigma \in T^*, \sigma = t_1 t_2 \dots t_n, \exists M_1, M_2 \dots M_n : M[t_1\rangle M_1[t_2\rangle M_2 \dots [t_n\rangle M_n$$

The firing of sequence σ is then denoted by $M[\sigma\rangle M_n$. The set of reachable markings *from marking M is denoted by $[M\rangle$.*

For a Petri net initially marked by M_0, the set of reachable markings is $[M_0\rangle$.

Using the firing rule, we can construct the reachability graph which is a directed graph with markings as vertices and where an arc between vertices M_1 and M_2 is labelled by the transition t leading from M_1 to M_2.

Definition 6 (Reachability graph) *Let $(\mathcal{N}, M_0)$ a marked Petri net. Its* reachability graph $G(\mathcal{N}, M_0) = (V, A)$ *such that:*

- $V = [M_0\rangle$ *is the set of vertices ;*
- $A = \{(M_1, t, M_2) \in V \times T \times V \mid M_1[t\rangle M_2\}$ *is the set of arcs, labelled by the transitions names.*

Algorithm 1 gives the basic construction of the reachability graph of a marked Petri net $(\mathcal{N}, M_0)$. It uses a set $Waiting$ of nodes already created but not processed yet. Function `Node` creates a node in the graph and adds it to $Waiting$, if it does not already exist. Function `Arc` creates a new arc in the graph.

Algorithm 1: Reachability Graph $G(\mathcal{N}, M_0)$

input : a marked Petri net $(\mathcal{N}, M_0)$
output: its reachability graph $G(\mathcal{N}, M_0)$
begin
  $Waiting \leftarrow \emptyset$
  `/* Initial state */`
  `Node` (M_0)
  `/* Process states in` $Waiting$ `*/`
  while $Waiting \neq \emptyset$ **do**
    `Choose` $M \in Waiting$
    forall the $t \in T$ *such that* $M[t\rangle M'$ **do**
      `Node` (M')
      `Arc` (M, t, M')
    endfall
  endw
  $Waiting \leftarrow Waiting \setminus \{M\}$
end

3. DISTRIBUTED GENERATION OF THE STATE SPACE

Several works have developed distributed tools generating and exploring the states space on a cluster of workstations [GMS01, KP04, AAC87]. Partitionning this set of states over the different stations is done using a hash function $h : V \longrightarrow S$ where:

- V is the set of all states (vertices of the reachability graph) ;
- $S = \{1, 2, \cdots, n\}$ is a set of stations (or processes) identifiers.

A marking M will then be stored and processed by process $h(M)$.

Each process i constructs and stores a part ((V_i, A_i)) of the state space such that:

- $V_i = \{M \in V \mid h(M) = i\}$;
- $A_i = \{(M, t, M') \in A \mid si \in V_i\}$.

There are two kinds of arcs: *local arcs*, linking nodes which belong to the same station ; and *traversal arcs*, linking two states belonging to two different stations, as sketched in figure 1. This latter kind of transitions requires transmission of messages between stations.

A good hash function not only insures a load balancing of states but also minimises the number of traversal arcs.

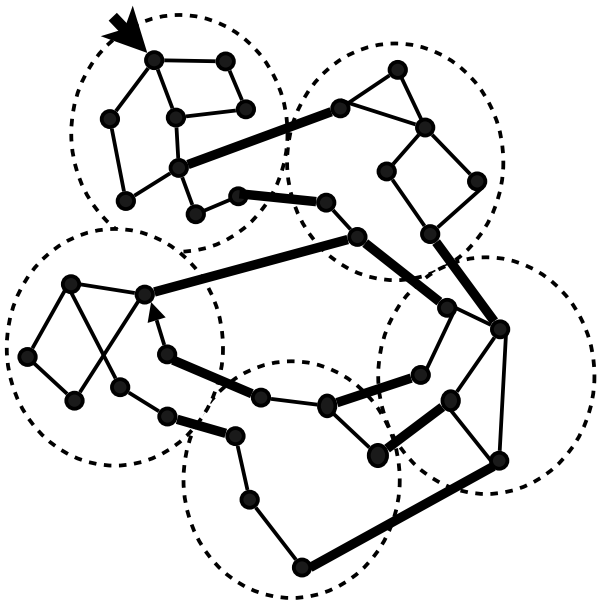

FIGURE 1: Partitionning state space on stations of the net

Several approaches achieve distributed state space generation:

In the first, there is two types of processes: a *generator process* which computes the states and transmits them to *storage processes*, using the hash function.

In the second approach consists of a *coordinator process* and *worker processes*. The coordinator initiates the generation by sending the initial state M_0 to the process $i = h(M_0)$.

There also exists a similar approach but without coordinator, in which the coordinator role is achieved by one of the workers, selected e.g. by election. Nevertheless, some problems then arise, such as termination, and are more complex to handle.

We chose to follow the second approach, as in [KP04]. It has the advantage of allowing parallel computations (as compared to the first approach) and easing implementation.

When a worker process i receives a state M it stores it in its local memory and pursues the generation of successor nodes. For a successor M', if $h(M') = i$, then M' is stored locally as well, otherwise M' is sent to the process $h(M')$. The worker process i executes distributed generation of algorithm 2. It is adapted from algorithm 1 to handle assignement of states to processes.

Algorithm 2: Distributed Generation()

```
begin
    /* Process states in Waiting */
    while Waiting ≠ ∅ do
        Choose M ∈ Waiting
        forall the t ∈ T such that M[t⟩M' do
            if h(M') ≠ i then
                /* New state M' does not belong to this process */
                Send (h(M'), M')
            else
                /* M' belongs to process i */
                Node (M')
            endif
            Arc (M, t, M')
        endfall
    endw
    Waiting ← Waiting \ {M}
end
```

The communications between processes are asynchronous. Thus, when receiving a message, the process is preempted and a handler function `MessageHandler ()`. Algorithm 3 indicates the operations executed when a state is received.

Algorithm 3: Message Handler()

```
begin
    ⋮
    if Message.Type = STATE then
        /* The message received contains a state to be processed */
            (Message.State)
        DistributedGeneration ()
    endif
    ⋮
end
```

$Message$ is a structure variable, which is composed by the items: $Message.Type$ and $Message.State$, respectively corresponding to the message type, and the state or marking transmitted in the message.

As in [KP04], among message types are those ensuring the termination of the distributed algorithm. They are all taken car of by the same procedure `MessageHandler ()`. The termination occurs when all the processes are terminated and no message is in transit.

4. PROPERTIES

A major issue is the analysis of concurrent systems properties. The reachability graph is the basic model on which most verifications are built on. Two types of properties can be studied:

- Behavioural properties, which depend on initial marking, and the reachability graph is often used to do this kind of analysis.

- Structural properties which are independent of the initial marking.

In this work we focus on basic behavioural properties: reachability, deadlocks, liveness, home state, their distributed analysis. The distributed algorithms presented here are traversal algorithms [Tel01], and some of the properties require paths, hence we need to store the arcs as well as the nodes. To do so, we explore the graph in a depth-first manner.

4.1. Reachability

The reachability problem for Petri nets consists in proving that a given marking M is in $[M_0\rangle$. This property is often used to check whether a faulty state M is reachable, e.g. an elevator moving while the door is open. It is convenient to look for erroneous states.

In some applications, one may be interested in the markings of a subset of places and not care about the others places in the net. This leads to a submarking reachability problem.

Reachability is known to be decidable, although it takes at least exponential space (and time) to verify in the general case. Determining whether a given state M is reachable from the initial marking can be done in parallel, each *worker* process searching through its own local set of states. It M is reachable, the worker with state M returns `true` to the coordinator, and the remaining *workers* return `false`. Otherwise (M not reachable), all workers return `false`. The coordinator can then report the result. This can be generalised to determine in parallel whether a state satisfying a given state predicate ϕ is reachable. In that case, several workers may return `true`, i.e. all those having in their part of the reachability graph a state M satisfying the predicate ϕ. If states are distributed evenly among the workers, then we should expect a linear speed up, compared to the usual mono-processor algorithm.

4.2. Deadlocks

A *deadlock* is often a non-suitable property, which happens when there is no further possible evolution of the system.

Definition 7 (Deadlock) *A marking M is a* deadlock *iff:* $\forall t \in T : \neg M[t\rangle$

The verification of this property consists in determining whether the reachability graph contains a marking M without successor. To verify in parallel the deadlock property, every *worker* process may look in its own reachability graph for such markings. As in the reachability analysis algorithm, the workers return the result of their search to the coordinator, which reports the global result, and a linear speedup can also be expected.

4.3. Home States

A home state is a marking that can always be reached in the future.

Definition 8 (Home state) *A marking M is a* home state *iff:* $\forall M' \in [M_0\rangle : M \in [M'\rangle$

A marked Petri net (N, M_0) is said to be *reversible* if M_0 is a home state. Thus, in a reversible net one can always get back to the initial marking.

4.4. Liveness

A transition is said to be *live* if it can always be fired in the future. We now give the definition for a set of transitions.

Definition 9 (Liveness) *Let $X \subseteq T$ be a set of transitions. Set X is* live *iff:*

$$\forall M \in [M_0\rangle : \exists M' \in [M\rangle, \exists t \in X : M'[t\rangle$$

This property is extended to a net: a marked Petri net (N, M_0) is said to be *live* if all of its transitions $t \in T$ are live. I.e., no matter which marking has been reached from M_0, it is possible to ultimatly fire any transition of the net by progressing through some further firing sequence. This means that a live Petri net guarantees deadlock-freeness.

The verification of such a property in practice comes down to verifying that the transition t belongs to all terminal strongly connected components (terminal SCCs, for short) of the Petri net reachability graph. Recall that a strongly connected component is said to be *terminal* if no vertex within this SCC has a successor in another SCC.

5. TOWARDS DISTRIBUTED VERIFICATION

Liveness and home state are desirable properties for many systems. Their verification is based on computation of the SCCs of the reachability graph.

In a distributed framework, as nodes are split among several stations, the SCCs calculus has to be achieved in a distributed fashion as well. This is a crucial step towards verification, which will then easily follow from the definitions.

Several algorithms compute the SCCs of a graph, e.g. :

1. a simple algorithm based on marking successors and predecessors, starting from a given node, to build the SCC it belongs to.
2. Tarjan's algorithm [Tar72], which is well-known for its efficiency.

Tarjan's algorithm has linear time computation, and requires only the links from the edges to the immediate successors to be stored.

The first algorithm, despite its simplicity, needs more memory usage, as the links to successors and predecessors are necessary.

Tarjan's algorithm [Tar72] executes iteratively the following actions:

- a numbering of the states in a depth first manner ;
- a computation of attachment points ;
- determination of the SCCs.

We then propose to distribute Tarjan's algorithm to compute the SCCs over a distributed state space.

5.1. Vertices numbering

The first step consists in numbering the states in a depth first order. It can be done as follows:

- the coordinator process sends the initial state to the process determined by the hash function $h(s_0)$.
- The process which receives the initial state starts the numbering.
- When a process p assigns a number to a state s, it also numbers each of its successors s', belonging to process p, if it is not numbered yet, in a depth first order. If a successor s' belongs to another process q then process p sends the pair (s, Num) to process q. Then, process q resumes the numbering from state s'.
- When a process q receives a state s from a process p, then if s is not numbered yet, it associates the received number Num with state s ($Num(s) \leftarrow Num$). Otherwise, s was already processed by q which then sends a *backtrack* message to process p.
- When the numbering of a subtree whose origin is received from the distant process p is terminated, a *backtrack* message is sent to p, so that it resumes its numbering.

Algorithm 4 achieves the `Numbering` of states by process p_i, upon reception of a state s from another process p_j. Note that the newly received state is added to the stack by `MessageHandler`, described in algorithm 5.

This algorithm follows a depth-first strategy for the generation of the state space. States that should be numbered are stored at the top of a stack (variable $Stack$), and are removed from there when processed. The stack also stores $backtrack$ information so that a process knows when it has finished its numbering and signals it to the process that had sent a state for processing.

Algorithm 4: Numbering()

```
begin
    NotEnd ← true
    while ¬empty(Stack) ∧ NotEnd do
        if top(Stack) ≠ backtrack then
            /* There is a state to number */
            s ← top(Stack)
            pop(Stack)
            if h(s) = i then
                /* It belongs to process i */
                s.Num ← Num
                Num ← Num + 1
                forall the s' such that ∃t ∈ T : s[t⟩s' do
                    push(s', Stack)
                endfall
            else
                /* The state belongs to another process */
                Send(h(s), (s, Num))
            endif
        else
            /* There is no node to number, hence backtrack */
            Send(p_j, backtrack)
            NotEnd ← false
        endif
    endw
end
```

Function `MessageHandler` takes care of the reception of states and *backtrack* messages, as in algorithm 5.

5.2. Attachment point computation

In Tarjan's algorithm, the attachment point of a state s, $at(s)$, represents the first state with the minimal number which can be reached from s. The attachment point is equal to the minimum of:

- $Num(s)$
- $Num(s')$, if s' is a successor of s and $Num(s') < Num(s)$
- $at(s')$ if s' is a successor of s and $Num(s') > Num(s)$

In the first two cases, the attachment point can be computed as soon as the numbering process starts. However, for the last case the attachment points are computed in a backtrack order, a specific stack is then used to contain the pairs (s, s') such that $Num(s) > Num(s')$ in order to compute the attachment point of s when the attachment point of $Num(s')$ will be known.

When the process of numbering is finished, if the pair (s, s') is at the top of the stack, we can be sure that the attachment point of s' is known, and thus we can compute the attachment point of s.

In the distributed case, we have two possibilities:

Algorithm 5: Message Handler()

```
begin
    ⋮
    if Message.Type = STATENUM then
        /* The message received contains a state to be numbered */
        s ← Message.State
        if s.Num ≠ 0 then
            /* The state has already be numbered */
            Send (p_j, backtrack)
        else
            /* The state has not been numbered yet */
            Num ← Message.Num
            push ((p_j, backtrack), Stack)
            push (s, (Stack))
            Numbering ()
        endif
    endif
    if Message.Type = backtrack then
        /* The process to which a node has been sent has finished */
                ()
    endif
    ⋮
end
```

1. s and s' are on the same station, then the update of the attachment point of s is done locally, and this can be done in parallel on the different stations.
2. if s belongs to process p and s' to process q, then both p and q have the pair (s, s') in their local stacks. Process p waits for the attachment point of s' when it pops the pair (s, s'), and process q sends to p the attachment point of s', when its pops the pair (s, s').

The processes compute in parallel the attachment points when the extremities are locally stored, they synchronise to compute them when the extremities do not belong to same process.

The next step consists in determining the edge candidate to be the origin of a SCC, this edge is such that :

$$Num(s) = \max\{Num(s') \mid Num(s') = at(s')\}$$

Each worker process starts by searching in its own edges the one which verify the previous condition, and sends them to the coordinator process which determines the one corresponding to the maximum. The subtree whose root correspond to the maximum constitutes a SCC.

This operation of numbering, attachment points computation and determination of a SCC is repeated until we obtain all the SCCs of the graph, and among this SCCs we distinguish those that are terminal.

6. EXPERIMENTATIONS

These algorithms were implemented for distributed Petri nets analysis, within a prototype tool. Given a Petri net, the reachability graph is partitionned on a cluster of stations, the SCCs computed and the liveness property is verified. In this section, we give the results obtained for two examples: the dining philosophers and the disributed database managers problems. The tests were carried on a cluster composed of 5 stations (Pentium IV with 512 Mbytes of memory) connected via Ethernet.

6.1. The dining philosophers problem

The Dining philosophers problem, introduced by Dijkstra, concerns the problem of ressource allocation between processes. The problem consists in a certain number of philosophers sitting around a circular table. There is a fork between each pair of neighbouring philosophers. Each philosopher spends his life alternatively thinking and eating, 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 partionned into sets S_1, $S_2 \ldots$ such that S_i should not contain all the states corresponding to i philosophers that are eating. Indeed, these states are not linked by any transition, they constitute a stable, so it is more interesting to store them on different stations. In the contrary, a state where there are i eating philosophers is linked only with sttes 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. So we tailored a hash function which tries to store them on the same station. The results obtained are given in table 1. The last column indicates the time for the computation with a single process (i.e. not distributed).

Nb Phil	Nb States	Nb Trans (N1)	Nb Trav. Trans. (N2)	N1/N2	Nb Mess. (N3)	N3/N2	Time (s)	1 proc Time
5	11	30	16	1.88	152	9.5	$<$0.01	$<$0.01
10	123	680	200	3.40	1,144	5.72	$<$0.01	0.01
15	1,364	11,310	2,260	5.00	11,535	5.10	0.06	0.12
20	15,127	167,240	25,084	6.67	125,815	5.02	0.87	3.85
25	167,761	2,318,400	278,206	8.33	1,391,691	5.00	44.14	521.50

TABLE 1: Philosophers Problem

We can note, from this table, that the number of traversal transitions compared to the total number of transitions, given by the ratio N1/N2, grows to reach more then 8, which means that there is one travesal transition for eight local transitions. This is a reasonable ratio which allows for limiting the number of messages. The number of messages is approximately 5 messages for each traversal transition, as indicated in the table by the ratio N3/N2.

6.2. Distributed database managers problem

The distributed database consists in n differents sites, each of which contains a copy of a database. This copy is handled by a local database manager. Each manager is allowed to make an update of its own copy, but must then send a message to all other managers, so that they perform the same update on their own copy. A database manager can be in three different states: $Inactive$, $Waiting$ (for acknowledgements) or $Performing$ (an update). When a manager, s decides to make an update he sends messages to all other managers, changes its state from $Inactive$ to $Waiting$ and waits until all other managers acknowledge the update. When the other managers receive a message, their state changes from $Inactive$ to $Performing$. Then, the database manager sends an $Acknowledgment$ to indicate, saying that the update is finished and its state changes from $Performing$ back to $Inactive$. The first manager (that started the whole updating process) may receive all the acknowledgements and its state changes from $Waiting$ back to $Inactive$. After that, the operations can start again with another update request.

A good hash function is easily found since a single manager initiates the work. Then, we can store the states where the manager i has done so, on station number ($i \mod n$). The results of the experiments with the database managers are given in table 2.

Despite the important increase of the size of the state space, the number of travesal transitions and the number of messages grows only slightly.

Nb Managers	Nb States	Nb Trans.	Nb Trav. Trans.	Nb Mess.	Time (s)	1 proc Time
6	1,460	4,878	14	208	$<$0.01	$<$0.01
7	5,105	20,433	15	224	0.27	0.01
8	17,498	81,688	16	245	1.33	0.12
9	59,051	314,955	19	282	2.68	3.85
10	196,832	1,181,010	22	293	131.73	521.5

TABLE 2: Distributed Database Problem

7. CONCLUSION

In this paper, we proposed steps towards distributed analysis of Petri net models. Based on the construction framework of [KP04], we introduced a distributed vesion of Tarjan's algorithm ([Tar72]) that allows for strongly connected components. Using these algorithms, it is possible to verify standard Petri nets properties, such as reachability, deadlocks, home states and liveness, in a distributed manner.

The main advantage of such an approach is the possibility to distribute the state space among a set of machines, thus limiting the drawbacks of the state space explosion problem.

Our algorithms were implemented in a prototype tool and some experiments on standard examples were carried on. The results obtained are satisfactory.

However, more work has to be achieved. We have in particular seen the importance of having a good hash function to ensure a good distribution of states among the set of processes. Finding such a hash function is not trivial. We are also interested in widening the analysis scope to temporal logic formulae.

REFERENCES

[[AAC87]] S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability anlysis for protocol verification environments. In P. Varaiya and H. Kurzhanski, editors, *Discrete Event Systems: Models and Application*, volume 103 of *LNCIS*, pages 40–56. Springer-Verlag, 1987.

[[AHI98]] K. Ajami, S. Haddad, and J.-M. Ilié. Exploiting symmetry in linear temporal model checking: One step beyond. In Springer-Verlag, editor, *Proc. 4th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 52–67, 1998.

[[BLS01]] J. Barnat, L.Brim, and J. Strîbrnà. Distributed LTL model checking in SPIN. In *Proc. 8th International SPIN workshop on Model Checking Software (SPIN'2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 200–216. Springer-Verlag, 2001.

[[BO03]] S. Blom and S. Orzan. Distributed state space minimization. *Electronic Notes in Theoretical Computer Science*, 80:1–15, 2003.

[[CFJ93]] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Proc. 5th Int. Conf. Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 450–462. Springer-Verlag, 1993.

[[CP00]] S. Christensen and L. Petrucci. Modular analysis of Petri nets. *The Computer Journal*, 43(3):224–242, 2000.

[[GMS01]] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proc. 8th International SPIN workshop on Model Checking Software (SPIN'2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 217–234. Springer-Verlag, 2001.

[[KP04]] L. Kristensen and L. Petrucci. An approach to distributed state exploration for coloured Petri nets. In *Proc. 25th Int. Application and Theory of Petri Nets (ICATPN'2004)*, volume 3099 of *Lecture Notes in Computer Science*, pages 474–483. Springer-Verlag, 2004.

[[LN01]] M. Leucker and T. Noll. Truth/SLC — A Parallel Verification Platform for Concurrent Systems. In *Proc. 13th Int. Conf. on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 255–259. Springer-Verlag, 2001.

[[LS99]] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. 5th and 6th International SPIN workshops on Model Checking Software (SPIN'1999)*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1999.

[[MJ96]] M.R.Henzinger and J.A.Telle. Faster algrithms for the nonemptiness of streett automata and for communication protocol pruning. In R.Karlson and A.lingas, editors, *Algorithm Theory: SWAT'96*, volume 1097, pages 16–27. Springer-Verlag, Berlin, LNCS, 1996.

[[Mur89]] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[[NG97]] R. Nalumasu and G. Gopalakrishnan. A new partial order reduction algorithm for concurrent systems. In *Proc. Int. Conf. Hardware Description Languages and their Applications (CHDL'97)*. Chapman & Hall, 1997.

[[Sch04]] K. Schmidt. Automated generation of a progress measure for the sweep-line method. In *Proc. 10th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 192–204. Springer-Verlag, 2004.

[[Tar72]] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[[Tel01]] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2001.

[[Val92]] A. Valmari. A stubborn attack on state explosion. *Formal Methods in Systems Design*, 1(4):297–322, 1992.