1. INTRODUCTION

Recently, formal verification is used for software intervening with critical systems, to avoid some fatal, dramatic and expensive errors [1,2].

Verification is usually performed by use of various formal methods that allow the precise description of systems and their desired properties, as well as reasoning about them. The systems are modeled in a language with clear mathematically defined syntax and semantics, like logics, automata and process algebra.

One of the most automated tool in the field of formal verification is Model Checking [3], where a finite state space of a system is constructed (generation phase) and the desired correctness properties are checked against it (verification phase). Generation phase consists on exploring all possible reached states from the initial state.

Actually, a large part of real-life systems are made up of many interacting components, for which state space can be huge and eventually cannot fit in memory or even be, it will be too large for being explored entirely. This leads to the so called state space explosion problem, which is known to be the most limiting and decreasing factor of this technique.

Several approaches have been developed to cope with this issue, such as symbolic state space encoding using binary decision diagram (BDD)[4,5], other method have taken benefits from some equivalence relations to reduce the state space of the model, like: partial order reduction [6], bisimulation relation [7], alpha-equivalence relation [8,9] and aggregation [10].

Currently, an approach based on the use of hardware resources and exploits the parallel and distributed computations, is variously investigated [11, 12, 13, 14], all these approaches consist on the use of a cluster or a network of workstations and each machine explores a part of the state space. However, they bear differences on the memory architecture used, either shared or distributed one, the data structure selected to store states, and the way sates are distributed over machines of the network.

In this current paper, our contributions are of interest for the formal verification domain and more especially for the distributed Model Checking. In its first step, Model Checking consists on generating the full state space: starting from the initial state, compute all its successors, repeat the process for all the newly discovered states until achieving exploration; there is no more state to explore.

Inspired from the file sharing service in Peer to Peer systems [15], we propose a new algorithm for distributed state space generation based on distributed hash table as a data structure for storing the set of states generated locally by the correspondent processor and for performing a quick look-up of states that have already been explored and the more important is to avoid exploring any state more than once.

So that, we aim at better construct the full state space of systems to enhance the problem of combinatorial state space explosion and achieve a good load balancing between processors of the network. Besides, this approach is very interesting for the community of graphs, since it handles the generation of graphs in a distributed environment on the fly while generating it.

The remainder of this paper is organized as follow; Section 2 introduces the context of our work and gives some preliminaries definitions: Petri nets, Distributed Hash Tables, Maximality semantic. Section 3 presents the basic proposed algorithm for the on the fly distributed generation of MLTS. In Section 4, we provide our measurements and experimental results. Finally, Section 5 draws conclusion and some future plans.

Keywords: Maximality based Labeled Transition System; Distributed Algorithm; State Space Explosion; DHT; Petri Nets.
2. PRELIMINARIES

2.1. PETRI NETS RELATED DEFINITIONS

- A Petri net is a tuple \((S,T,W)\) where \(S\) is the set of places, \(T\) is the set of transitions such that \(S \cap T = \emptyset\), and \(W : (S \times T \cup (T \times S)) \rightarrow \mathbb{N} = \{0,1,2,...\}\) is the weight function. Graphically, transitions of \(T\) are represented by rectangles, places of \(S\) by circles and weight function by arrows associated with their weights. We suppose that all nets are finite, i.e. \(|S \cup T| \in \mathbb{N}\).

- For \(x \in S \cup T\) the pre-set \(*x\) is defined by \(*x = \{y \in S \cup T \mid W(y,x) \neq 0\}\) and the post-set \(x^*\) is defined by \(x^* = \{y \in S \cup T \mid W(y,x) \neq 0\}\).

- The marking of a Petri net \((S,T,W)\) is defined as a function \(M : S \rightarrow \mathbb{N}\). A marking is generally represented graphically by putting tokens in places.

- The transition rule stipulates that a transition \(t\) is enabled by \(M\) iff \(M(s) \geq W(s,t)\) for all \(s \in S\). The firing of a transition \(t\) will produce a new marking \(M'\) defined by \(M'(s) = M(s) - W(s,t) + W(t,s)\) for all \(s \in S\). The occurrence of \(t\) is denoted by \(M(t)\).

- Two transitions \(t_1\) and \(t_2\) (not necessarily distinct) are concurrently enabled by a marking \(M\) iff \(M(s) \geq W(s,t_i)\) for all \(s \in S\).

- A marked Petri net \((S,T,W,M_0)\) is a Petri net \((S,T,W)\) with an initial marking \(M_0\).

- An alphabet \(A\) is a finite set; we suppose that \(\tau \notin A\) (\(\tau\) will indicate invisible action, or silent action).

- The labeling of a Petri net \(N = (S,T,W)\) is a function \(\lambda : T \rightarrow A \cup \{\tau\}\). If \(\lambda(t) \notin A\) then \(t\) is said to be observable or external; at the opposite, \(t\) is silent or internal.

- \(\Sigma = (S,T,W,M_0)\) is a labeled system iff \((S,T,W)\) is a marked Petri net and \(\lambda\) is a labeling function of \((S,T,W)\).

- An action \(a \in A\) of a system \(\Sigma = (S,T,W,M_0)\) is auto-concurrent in a marking \(M\) iff \(M\) concurrently enables two observable transitions \(t_1\) and \(t_2\) (not necessarily distinct) such that \(\lambda(t_1) = \lambda(t_2) = a\).

- A sequence \(a = M_{\alpha}M_{\beta}...\) is an occurrence sequence iff \(M_{\alpha}t(M)\) for \(\alpha \geq i\). A sequence \(t_1t_2...\) is a transition sequence starting with \(M\) iff there is an occurrence sequence \(M_{\alpha}t_1M_{\beta}t_2...\). If a finite sequence \(t_1t_2...t_n\) leads from \(M\) to \(M'\) we write \(M(t_1t_2...t_n)M'\). The set of reachable markings of a marked Petri net \((S,T,W,M_0)\) is defined as \([M_0] = \{M \mid t_1t_2...t_n : M_0 = M_0 t_1t_2...t_nM\}\).

2.2. MAXIMALITY BASED LABELED TRANSITION SYSTEM

Let \(\Sigma\) be a countable set of events names, maximality based labeled transition system of support \(\Sigma\) is a quintuplet: \((\Sigma, \lambda, \mu, \psi, \xi)\) with:

\[=(S,T,\alpha,\beta)\] is a transition system such that:

- \(S\) is the countable set of states in which the system can be found.
- \(T\) is the countable set of transitions indicating the change of system states.
- \(\alpha\) and \(\beta\) are two applications of \(T\) in \(S\) such that for all transition \(t\) we have: \(\alpha(t)\) is the origin of the transition and \(\beta(t)\) its goal.
- \((\lambda, \mu)\) is a transition system labeled by an alphabet \(A\).
- \(\psi : S \rightarrow \mathbb{N}\) which associates to each state the finite set of maximal event names present in this state.
- \(\mu : T \rightarrow \mathbb{N}\) is a function which associates to each transition the finite set of event names corresponding to actions that have already begun their execution and of which the end of execution enables this transition.

\[\xi : T \rightarrow M; t : \text{a function which associates to its transition an event name identifying its occurrence. Such that for any transition } t, \mu(t) = \psi(a(t)) - \mu(t) \text{ and } \psi(\beta(t)) = (\psi(a(t)) - \mu(t)) \cup \{\xi(t)\}.\]

2.3. OPERATIONAL MAXIMALITY SEMANTICS FOR PETRI NETS

In this section, we introduce a notations and functions for the definition of marking graph associated to a labeled system in a maximality-based approach.

**Definition 1** Let \(N = (S, T, W)\) be a Petri net, the marking of \(N\) is a function \(M : S \rightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N}\). Among others, the marking \(M(s)\) of a place \(s \in S\) is a pair \((\mathcal{F}T, \mathcal{B}T)\) such that \(\mathcal{F}T \subseteq \mathbb{N}\) and \(\mathcal{B}T \subseteq \mathbb{N}\) denote respectively the number of free tokens and the set (possibly empty) of bound tokens in the place \(s\). In what follows, a Petri net with a marking will be called configuration. \([M(s)]\) denote the total number of tokens in a place \(s\). If \(M(s) = (\mathcal{F}T, \mathcal{B}T)\) such that \(\mathcal{B}T = \{n_1x_1,...,n_mx_m\}\) then \([M(s)] = \mathcal{F}T + |\mathcal{B}T|\) with \(|\mathcal{B}T| = \sum_{i=1}^{m} n_i\) is the cardinal of the bound tokens set in \(s\).

**Definition 2** Let \((S, T, W)\) be a Petri net with a marking \(M\):

- The set of maximal event names in \(M\) is the set of all event names identifying bound tokens in the marking \(M\). Formally, the function \(\psi\) will be used to calculate this set, it can be defined as \(\psi(M) = \bigcup_{s \in \mathcal{S} = \{1,2,...\}} \bigcup_{x \in \mathcal{X} = \{x_1,x_2,...\}} M(s) = \{n_1, n_2, ..., n_m\}\).
- \(N \in \mathcal{A}_\Sigma\) be a non-empty finite set of event names, makefree \((N, M)\) is defined recursively by:

\[\text{makefree} (\{x_1, x_2, ..., x_n\}, M) = \text{makefree} (\{x_2, ..., x_n\}, \text{makefree} (\{x_1\}, M)),\]
makefree \((fx, M) = M'\) such that for all \(s \in S\), if \(M(s) = (F_1, T_1, B_1)\) then

* If there is \((n, t, x) \in B_T\) then \(M'(s) = (F_1 + n, B_T - \{(n, t, x)\})\) (Conversion of \(n\) free tokens).
* Otherwise, \(M'(s) = M(s)\).

- Let \(t\) be a transition of \(T\); \(t\) is said to be enabled by the marking \(M\) iff \(|M(s)| > W(s, t)\) for all \(s \in S\). The set of all transitions enabled by the marking \(M\) will be noted enabled \((M)\).

- The marking \(M\) is said to be minimal for the firing of the transition \(t\) iff \(|M(s)| = W(s, t)\) for all \(s \in S\).

- Let \(M_1\) and \(M_2\) be two markings of the Petri net \((S, T, W)\). \(M_1 \neq M_2\) iff \(\forall s \in S\), if \(M_1(s) = (F_1, T_1, B_1)\) and \(M_2(s) = (F_2, T_2, B_2)\) then \(F_1 \neq F_2\) and \(T_1 \neq T_2\) and \(B_1 \neq B_2\) such that the relation \(\neq\) is extended to bound tokens sets as follows:

\[
F_1 \neq F_2 \iff \forall (n_1, t, x) \in B_1, \exists (n_2, t, x) \in B_2\text{ such that }n_1 \neq n_2.
\]

- Let \(M_1\) and \(M_2\) be two markings of the Petri net \((S, T, W)\) such that \(M_1 \neq M_2\). The difference \(M_1 - M_2\) is a marking \(M_1 - M_2 = M_3\) such that for all \(s \in S\), if \(M_3(s) = (F_3, T_3, B_3)\) and \(M_2(s) = (F_2, T_2, B_2)\) then \(M_3(s) = (F_3 - F_2, T_3 - T_2, B_3 - B_2)\) with

\[
F_3 - F_2 = F_3 - F_2, T_3 - T_2 = T_3 - T_2, \text{ and } B_3 - B_2 = B_3 - B_2\text{ such that }n_1 \neq n_2\text{ then }n_1 - n_2, t, x = B_3 - B_2.
\]

- \(\text{Min}(M, t) = \{M' | M' \neq M\}\) and \(M'\) is minimal for the firing of \(t\).

- Let \(\lambda\) be a set. The function get: \(2^{\lambda} - \emptyset \rightarrow \lambda\) is a function which satisfies get(E) \(\subseteq E\) for any \(E \in 2^{\lambda}\).

- Given a marking \(M\), a transition \(t\) and an event name \(x \in \psi(M)\) occur \((t, x, M) = M'\) such that \(\forall s \in S\), if \(M(s)\) = \((F_s, T_s, B_s)\) then \(M'(s) = (F_s, T_s, B_s)\) with \(F_s = F_s \cup \{(W(t, s) - t, x)\}\) if \(W(t, s) \neq \emptyset\) and \(B_s = B_s\) otherwise. Hence, \(M'\) is the resultant marking from the addition of tokens bound to \(t\) to the marking \(M\).

### 2.4. DISTRIBUTED HASH TABLES

Peer-to-peer systems can tie together millions of edges hosts to provide efficient and fault-tolerance infrastructures services, such as search, storage, caching, load-balancing and routing. In the early 2000, the popularity of peer-to-peer file sharing systems such as Napster and Gnutella [16] leads to grow research on structured overlay networks and then to invent the structured distributed hash tables: Chord [17].

Distributed hash tables (DHT) are a powerful building block for highly scalable decentralized systems, they are commonly used in peer to peer file sharing network such as PChord [18]. The DHTs route requests over a structured overlay network \(t\) the node responsible for a given key. The basic approach of many DHTs is as follows:

- Participating computers [stations in the network] each have a node ID in the key space. [key, value] pairs are stored on nodes with IDs close to the key for some notion of closeness. Finally, a node-ID-based routing algorithm lets anyone efficiently locate servers near any given target key.

In the general approach of DHT, we assign 160-bit opaque IDs to nodes and provide a lookup algorithm that locates successively ‘closer’ nodes to any desired ID, converging to the lookup target in algorithmically many steps.

Distributed hash tables are applied to a variety of distributed application such as file systems, databases, reputation engines. In recent years, DHTs have increasingly been adapted and deployed for production use in industry. For example, Amazon developed the Dynamo DHT, and facebook developed Cassandra [19], which is also used by Twitter, Cisco WebEx, and others.

Currently DHT are used to carry out load balancing algorithms for peer to peer systems [20].

### 3. DISTRIBUTED STATE SPACE GENERATION BASED DISTRIBUTED HASH TABLES

#### 3.1. THE DISTRIBUTED HASH TABLES IN DISTRIBUTED GRAPH GENERATION

The reason behind introducing the distributed hash table in the process of graph generation was to reduce the combinatorial state space explosion and build an efficient model checking while getting advantages of distribution features. For that, using a single shared memory to store the data will decrease the algorithm performances. Thus, our idea was to associate each processor with a local hash table (memory space) where states are tied with their generator processor identity, so, each processor will have its own hash table where states generated by other processor will be stored with all the necessary information.

A distributed hash table is a table that stores the relation correlating each state with the processor that has generated it. A distributed hash table can be considered as an associative array of size \(n\), where pairs of \(\text{<key, value>}\) are stored.

In our implementation, keys represent states which are given by their configurations, while the values indicate IDs of the processors that generate a given states, and a list \(L\) of the possible external transitions that relies the key state with other states in other nodes (Figure 1).

As a consequence, DHT forms an efficient data structure that can outperform other data structure by ensuring the following operations:

- A quick look-up of states that have already been generated.
- Obtaining some essential information about a give state, such as, knowing its generator processor, and getting IDs of processors having
external transitions with it, this will be very important in the process of verification of properties.

- Quick and easy insertion of newly discovered states.

```
<table>
<thead>
<tr>
<th>Conf1</th>
<th>ID_P0</th>
<th>ID_P1</th>
<th>ID_P4</th>
<th>ID_Px</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conf2</td>
<td>ID_Ps</td>
<td>ID_P0</td>
<td>ID_P7</td>
<td>ID_P3</td>
</tr>
</tbody>
</table>
```

*Figure 1. The Distributed Hash Table structure.*

The operations performed on the DHTs are:

- Create_DHTvalue (conf, ID_P)
- Check (); search in the first column, if a state has already been inserted.
- Insert (new_DHTvalue, DHT)
- Update (DHT), by adding cross-transitions in L.

Each operation is atomic, and all nodes (processors) see the same coherent image of all existing hash tables. Initially, all DHTs are empty and initialized with the value (0, 0).

To insert states and look for values inside the DHT, a partition function (which will be defined in the next subsection) is used. The partition function takes as input a state and returns the identity of the processor who will store it in its private DHT.

For each newly discovered state, we create a pair DHT value (new_conf, ID_P). The insertion operation is performed by checking whether the value turned by the partition function correspond to the same ID_P or to another.

- If Attr_f(new_conf) = ID_P, then, we check whether the state has already been added in the DHT, if the state is a fresh one, we add the DHT value in DHT of processor ID_P. Otherwise (the state exist in DHT), we link the new_conf with the old one (create an internal transition).
- If Attr_f(new_conf) = ID_P’ then we know that new_conf will be added and saved in the DHT of ID_P’, for that, a send message containing the essential information is performed.

To check if a state s is in the DHT of ID_P’, we test if the Attr_f(s) return the value ID_P’, if this is the case, we check if s is in the local DHT. If Attr_f(s) ≠ ID_P’, we perform the lookup-up on the DHT of the value turned by the partition function.

Thus, the operation of checking is limited in one processor (the one turned by the partition function which do not make collisions), so that, we reduce the number of control messages (sent and received).

### 3.2. THE PARTITION FUNCTION

The usual approach in distributed reachability analysis is to partition the state space using a static function [9, 11, 21] which is based on a hash function, that depend on the state representation. This function must unambiguously assign each state to one of the network nodes. So, choosing a good partition scheme will affect the algorithm performances, in term of load balancing and collisions between states.

Then, the problem is how to distribute a graph over a network of workstations while preserving a good load balancing over workers.

**Definition**

Let MLTS = (Ω, λ, µ, θ, ψ) be an MLTS and Attr_f : S → {0, ..., N−1} a total function called partition function. A partitioning of MLTS under Attr_f is tuple = (M0, ..., Mn−1) such that i ∈ {0 ... n−1} Mi = fragmentMLTS( {s ∈ S | Attr_f(s) = i} ).This function takes a state, which is represented in our case by a configuration, and returns the identifier of a node.

In our approach, we choose a partition function of the form

\[
\text{Attr}_f(s) = f(s, P) \mod N
\]

Where P is prime number and f(s; P) is the hash function that computes the remainder modulo P of the integer value of the state s.

This definition is somehow similar to the one given in [9, 11]. However, in [9, 11] authors rely only on the partition function to distribute states over nodes, that leads to a bad load balancing. In addition, this static scheme suffers from the high communication overhead [22]; since states are discovered, number of cross-transition (transitions that require inter-node communication) grows rapidly because of the randomly assignment of states over nodes.

To address this problem, we have conceive an approach based on the use of both distributed hash tables and partition function to store all reachable states, while maintaining a local generation of states. This leads to achieving a good load balancing (since states are evenly divided over processors in the initial step), and makes a great way to control the amount of cross-transitions, so then, the explicit communication overhead.

### 3.3. THE PROPOSED ALGORITHM

**Framework assumptions**

The target framework of the developed algorithm is a failure free distributed system (no node failure, no message loss). We assume that processes communicate by invoking two message passing primitives: SEND (message, destination_node) and RECEIVE (message), SEND is non blocking, RECEIVE is blocking and return a Boolean answer indicating whether a message has been received or not.
There are three types of messages: DHT_insert, Notify, and DHT_value. The first one being used for sending states to the appropriate node that will store it in its local DHT. Notify and DHT_value are used for the work-sharing technique.

We assume that each machine of the network maintain a private hash table (as defined above) where states (generated by other processors) are stored, and a stack (NT_config) of non-explored states.

In this article, we choose the formal description technique Petri Nets (place/transition). This description will be translated to the subjacent model, presented by a graph (or automate) of non-explored states. The MLTS contain eventually with certain abstraction, all the possible reachable states.

**Description of the approach**

We consider a network of N machines numbered from 0 to N–1 and a Petri net given by its initial configuration, the computation is started by the initiator machine having the index h(initial_config).

As mentioned before, our approach makes use of distributed hash table as a global look-up table, to test whether a state may have already been founded or not and then, to decide using the partition function where to store it.

Furthermore, we can recovery which processor had generated a given state to make the necessary updates (create new DHT value). To this end, we make sure that no redundant states will appear in the full graph.

As we perform on a distributed architecture, load balancing between workers is an important factor to be respected, to overcome this issue, we devised an algorithm named Maximal based Labeled transition system MLTS (we refer the readers to [23] for further information). The MLTS contain eventually with certain abstraction, all the possible reachable states.

**Algorithm Worker**

```plaintext
Algorithm Worker

Variables:
NT_config: the list of configurations not yet explored
S_i: the list of state the MLTS
T_i: the list of transition the MLTS

While (NT_config ≠ Ø) (N_msg_send < 0) Do
conf := pop(NT_config);
Exhaustive development /* SOS of Maximal */
For each t in Transition Do /*t is a Petri net transition*/
    If enabled(t) Then
        For each new configuration new_conf Do
            Create_DHT value (new_conf, i);
            Create internal transition (conf, new_conf);

        Endfor
Endwhile
```

**Algorithm Reciever**

```plaintext
Algorithm Reciever

RECEIVE (msg);
Case msg
"DHT_insert", DHTvalue >:;
    N_msg_send, --;
    check if conf exist in DHT,;
    If conf exist Then
        Update (DHT, \* add ID_P to L \*
        Create cross-transition;
    Else
        Insert (DHT_value, DHT);
    Endif
Endfor
```

Each processor runs two threads. The first, thread Worker (Figure 2) is busy with creating a new DHT value (pair) for each new state, then, keep it or send it to the processor where it should be inserted (ID processor is determined using the partition function Attr_f()).

When all generated states are visited (NT_config = Ø), a notification message is sent to the right neighbor of process i (indexed by (i+1) mod N) in order to minimize the number of idle processor and maximize the processors occupancy.

The other thread handles the incoming messages (Figure 3). A request for inserting a state (DHT_value) in a hash table is handled by looking up inside the DHT, if not found, adding it to the table, or if found, make update the transitions. The Receiver thread may terminate when all workers announced that they have no more state to send and all requests have been treated.

**Algorithm Receiver**

```plaintext
Algorithm Receiver

RECEIVE (msg);
Case msg
"DHT_insert", DHTvalue >:
    N_msg_send, --;
    check if conf exist in DHT,;
    If conf exist Then
        Update (DHT, \* add ID_P to L \*
        Create cross-transition;
    Else
        Insert (DHT_value, DHT);
    Endif
Endfor
```

```plaintext
Algorithm Worker

Variables:
NT_config: the list of configurations not yet explored
S_i: the list of state the MLTS
T_i: the list of transition the MLTS

While (NT_config ≠ Ø) (N_msg_send < 0) Do
    conf := pop(NT_config);
    Exhaustive development /* SOS of Maximal */
    For each t in Transition Do /*t is a Petri net transition*/
        If enabled(t) Then
            For each new configuration new_conf Do
                Create_DHT value (new_conf, i);
                Create internal transition (conf, new_conf);
```

**Figure 2. Algorithm performed by thread Worker.**

Each processor runs two threads. The first, thread Worker (Figure 2) is busy with creating a new DHT value (pair) for each new state, then, keep it or send it to the processor where it should be inserted (ID processor is determined using the partition function Attr_f()).

When all generated states are visited (NT_config = Ø), a notification message is sent to the right neighbor of process i (indexed by (i+1) mod N) in order to minimize the number of idle processor and maximize the processors occupancy.

The other thread handles the incoming messages (Figure 3). A request for inserting a state (DHT_value) in a hash table is handled by looking up inside the DHT, if not found, adding it to the table, or if found, make update the transitions. The Receiver thread may terminate when all workers announced that they have no more state to send and all requests have been treated.

**Algorithm Receiver**

```plaintext
Algorithm Receiver

RECEIVE (msg);
Case msg
"DHT_insert", DHTvalue >:
    N_msg_send, --;
    check if conf exist in DHT,;
    If conf exist Then
        Update (DHT, \* add ID_P to L \*
        Create cross-transition;
    Else
        Insert (DHT_value, DHT);
    Endif
Endfor
```

**Algorithm Worker**

```plaintext
Algorithm Worker

Variables:
NT_config: the list of configurations not yet explored
S_i: the list of state the MLTS
T_i: the list of transition the MLTS

While (NT_config ≠ Ø) (N_msg_send < 0) Do
    conf := pop(NT_config);
    Exhaustive development /* SOS of Maximal */
    For each t in Transition Do /*t is a Petri net transition*/
        If enabled(t) Then
            For each new configuration new_conf Do
                Create_DHT value (new_conf, i);
                Create internal transition (conf, new_conf);
```
Add_All (Conf_stack, NT_config);
START Worker;

Figure 3. Algorithm performed by thread Receiver.

4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

In order to investigate the feasibility and effectiveness of the proposed approach, we have enriched the environment (D-STEM-PNet) developed previously in [10] by the distributed generation of MLTS based distributed hash tables. This environment has a graphical editor to draw and edit Petri nets “Figure 4” and a result viewer “Figure. 5” that produces a dot file type [24].

The proposed algorithm has been encoded in java programming language, and implemented on a network of 3 machines with 16 GBytes of RAM under windows XP, each machine have 2 processors with 4 cores of calculus. Machines are connected with 100 Mbps Ethernet. For the communication we have used jade environment [25] so each node is presented by an agent.

For experiments we run a simulation of 10 nodes, Figure 6 shows the dispersion of states over the nodes. Compared to figure 7 that shows the dispersion given by applying a static hash function like in [9, 11], we can clearly see that our proposed outperform the one based on a hash function.

Case study:

In order to illustrate the interest of the proposed approach, we study in this section an example of processes synchronization, namely ticket reservation system illustrated in “Fig. 4”.

We can also see the rate of both external and internal transitions (figure 7), it is obvious that a good connectivity was achieved, our method minimize the number of cross-transition, and so on, minimize the number of message on the network.
Finally, figure 8, shows the total inactivity time of each processor. We can see that the work sharing technique used in our approach give good results.

5. CONCLUSION

In this paper, we presented a method to increase the problem of state space explosion, by introducing a new data structure. The approach is derived from the file sharing system. It implies creating a Distributed Hash Tables to maintain the distribution of states while performing the generation on the fly.

This approach optimizes the nodes load balanced and minimize the rate of inactivity of each processor. We automated this approach by developing a Java tool, which will be integrated in the FOCOVE environment [26]. The approach shows promising results that can be optimized in the future by exploiting the computational power of recent multi-core architecture. Furthermore, we attempt to modify the program to deal with the system scalability while taking account of reliability, load, administration and heterogeneity.

REFERENCES


