World Academy of Science, Engineering and Technology
International Journal of Mathematical and Computational Sciences
Vol:1, No:8, 2007

On Analysis of Boundness Property for ECATNets by Using Rewriting Logic

Noura Boudiaf, and Allaoua Chaoui

***Abstract*—**To analyze the behavior of Petri nets, the accessibility graph and Model Checking are widely used. However, if the analyzed Petri net is unbounded then the accessibility graph becomes infinite and Model Checking can not be used even for small Petri nets. ECATNets [2] are a category of algebraic Petri nets. The main feature of ECATNets is their sound and complete semantics based on rewriting logic [8] and its language Maude [9]. ECATNets analysis may be done by using techniques of accessibility analysis and Model Checking defined in Maude. But, these two techniques supported by Maude do not work also with infinite-states systems. As a category of Petri nets, ECATNets can be unbounded and so infinite systems. In order to know if we can apply accessibility analysis and Model Checking of Maude to an ECATNet, we propose in this paper an algorithm allowing the detection if the ECATNet is bounded or not. Moreover, we propose a rewriting logic based tool implementing this algorithm. We show that the development of this tool using the Maude system is facilitated thanks to the reflectivity of the rewriting logic. Indeed, the self-interpretation of this logic allows us both the modelling of an ECATNet and acting on it.

***Keywords*—**ECATNets, Rewriting Logic, Maude, Finite-state Systems, Infinite-state Systems, Boundness Property Checking.

I. Introduction

THE development of provably error-free concurrent systems is still a challenge of system engineering. Modeling and analysis of concurrent systems by means of Petri nets is one of the well known approaches using formal methods. Two of well known analysis techniques of Petri nets are dynamic analysis and Model Checking. These two methods are largely used in the verification of different category of Petri nets. However, if the analyzed Petri net is unbounded then the reachability graph becomes infinite and Model Checking can not be used, even for small Petri nets.

ECATNets [2] are a category of algebraic Petri nets (APNs) based on a safe combination of algebraic abstract types and high level Petri nets. The semantic of ECATNets is defined in terms of rewriting logic [8], allowing us to built models by formal reasoning. As Petri nets, ECATNets provide a quickly understood formalism due to their simple construction and graphical depiction. Moreover, ECATNets have a strong theory and development tools based on powerful logic with sound and complete semantic. The integration of ECATNets in rewriting logic is very promising in terms of specification and verification of their properties. Rewriting logic provides to ECATNets a simple, intuitive, and practical textual version to analyze systems, without loosing the formal semantic. ECATNets analysis may be done by using techniques of accessibility analysis and Model Checking defined in Maude [3]. However, these two techniques supported by Maude do not work with infinite-states system. As a category of Petri nets, ECATNets can be unbounded and so infinite system. Consequently, study of boundeness property is important in our sense to decide the applicability of accessibility analysis and Model Checking for ECATNets.

The study of boundness property for different category of Petri nets is a known problem in the literature. Such studies aim in general to construct a coverability graph for Petri nets. Among these: The Pr/T-net reachability analysis tool PROD [12] implements several methods for efficient reachability analysis, PAPETRI [1] constructs reachability and coverability graphs for place/transition nets, colored nets, and algebraic Petri nets, CPN/AMI [5], DESIGN/CPN [10], and INA (Integrated Net Analyzer) [11] computes the coverability graph (for Place/Transition Nets (P/T) and colored Petri nets (CPN) with time and priorities) using the algorithm of KARP and MILLER. In the case where the net is bounded, the coverability graph corresponds to the usual reachability graph. In [4], FINKEL considers reachability graphs and coverability graphs as special cases of a more general structure, so-called ω-state graphs. Among all these state graphs, there exists a unique one which is minimal with respect to the number of nodes.

In our case, we will restrict ourselves to study boundeness property for ECATNets. We do not have as an objective in this paper the construction of the coverabilty graph but only we decide through a proposed algorithm if an ECATNet is bounded or not.

In this paper, we propose an algorithm and its rewriting logic based tool to check boundness property for ECATNets. First, we study unbounded places in ECATNets by giving some propositions and their proofs. Then, we extract conditions of unboundness of ECATNets. The algorithm computes the accessibility graph (finite) and checks at the same time the conditions of unbounded places in an ECATNet. If one of the unboundeness conditions is true, the algorithm stops computing and returns that the ECATNet is not bounded. Otherwise, if the algorithm finishes the

Noura Boudiaf is with University of Constantine, Algeria. (e-mail: boudiafn@yahoo.com).
Allaoua Chaoui is with University of Constantine, Algeria (e-mail: a_chaoui2001@yahoo.com).

construction of accessibility graph, then it returns that the ECATNet is bounded. After that, we present a tool based Maude that implements this algorithm. Such tool allows us to know the applicability of the accessibility analysis and the Model Checking of Maude for the ECATNets. The development of this tool is not very complicated thanks to the reflectivity of Maude language. Indeed, the self-interpretation of this logic allows us both the modelling of an ECATNet and acting on it.

The remainder of this paper is organized as follows: the section 2 is a general presentation of the ECATNets and their description in rewriting logic. Some proprieties about ECATNets including a study of unbounded places case are presented in section 3. Section 4 contains our proposed algorithm which detects if an ECATNet is bounded or not. In section 5, we introduce briefly the concept of the meta-computation in Maude. In section 6, we describe our algorithm's implementation in Maude system. In section 7, we give an example of an ECATNet and its description in Maude meta-level representation. Section 8 contains the application of the tool on the example. Finally, the section 9 concludes the paper.

II. ECATNets

ECATNets [2] are a kind of net/data model combining the strengths of Petri nets with those of abstract data types. Places are marked with multi-sets of algebraic terms. Input arcs of each transition t, i.e. (p, t), are labeled by two inscriptions IC(p, t) (Input Conditions) and DT(p, t) (Destroyed Tokens), output arcs of each transition t, i.e. (t, p'), are labeled by CT(t, p') (Created Tokens), and finally each transition t is labeled by TC(t) (Transition Conditions) (see figure 1). IC(p, t) specifies the enabling condition of the transition t, DT(p, t) specifies the tokens (a multi-set) which have to be removed from p when t is fired, CT(t, p') specifies the tokens which have to be added to p' when t is fired. Finally, TC(t) represents a boolean term which specifies an additional enabling condition for the transition t. The current ECATNets' state is given by the union of terms having the following form (p, M(p)). As an example, the distributed state s of a net having one transition t and one input place p marked by the multi-set a $\oplus$ b $\oplus$ c, and an empty output place p', is given by the following multi-set : s = (p, a $\oplus$ b $\oplus$ c).

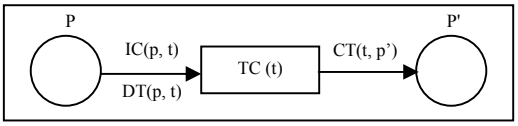

Fig. 1 A generic ECATNet

A transition t is enabled when various conditions are simultaneously true. The first condition is that every IC(p, t) for each input place p is enabled. The second condition is that TC(t) is true. Finally, the addition of CT(t, p') to each output place p' must not result in p' exceeding its capacity when this capacity is finite. When t is fired, DT(p, t) is removed (positive case) from the input place p and simultaneously CT(t, p') is added to the output place p'. Let's note that in the non-positive case, we remove the common elements between DT(p, t) and M(p). Transition firing and its conditions are formally expressed by rewrite rules. A rewrite rule is a structure of the form "t: u $\rightarrow$ v if boolexp"; where u and v are respectively the left and the righthand sides of the rule, t is the transition associated with this rule and boolexp is a Boolean term. Precisely u and v are multi-sets of pairs of the form (p, $[m]_\oplus$), where p is a place of the net, $[m]_\oplus$ a multi-set of algebraic terms, and the multi-set union on these terms, when the terms are considered as singletons. The multi-set union on the pairs (p, $[m]_\oplus$) will be denoted by $\otimes$. $[x]_\otimes$ denotes the equivalence class of x, w.r.t. the ACI (Associativity, Commutativity, Identity = ϕ_M) axioms for $\otimes$. An ECATNet state is itself represented by a multi-set of such pairs where a place p is found at least once if it's not empty. Now, we recall the forms of the rewrite rules (i.e., the meta-rules) to associate with the transitions of a given ECATNet.

IC(p,t) is of the form $[m]_\oplus$

Case 1. $[IC(p, t)]_\oplus = [DT(p, t)]_\oplus$
The form of the rule is then given by:
$t : (p, [IC(p, t)]_\oplus) \rightarrow (p', [CT(t, p')]_\oplus)$
where t is the involved transition, p its input place, and p' its output place.
Case 2. $[IC(p, t)]_\oplus \cap [DT(p, t)]_\oplus = \phi_M$
This situation corresponds to checking that IC(p, t) is included in M(p) and, in the positive case, removing DT(p, t) from M(p). In the case where DT(p, t) is not included in M(p), we have to remove the elements which are common to these two multi-sets. The form of the rule is given by:
$t : (p, [IC(p, t)]_\oplus) \otimes (p, [DT(p, t)]_\oplus \cap [M(p)]_\oplus) \rightarrow (p, [IC(p, t)]_\oplus) \otimes (p', [CT(t, p')]_\oplus)$
Case 3. $[IC(p, t)]_\oplus \cap [DT(p, t)]_\oplus \neq \phi_M$
This situation corresponds to the most general case. It may however be solved in an elegant way by remarking that it could be brought to the two already treated cases. This is achieved by replacing the transition falling into this case by two transitions which, when fired concurrently, give the same global effect as our transition. In reality, this replacement shows how ECATNets allow specifying a given situation at two levels of abstraction. The forms of the axioms associated with the extensions are, w.r.t. the explanation already given, evident and thus not commented.

IC(p, t) is of the form ~$[m]_\oplus$
The form of the rule is given by:
$t : (p, [DT(p, t)]_\oplus \cap [M(p)]_\oplus) \rightarrow (p', [CT(t, p')]_\oplus)$
if $([IC(p, t)]_\oplus \setminus ([IC(p, t)]_\oplus \cap [M(p)]_\oplus)) = \phi_M \rightarrow$ [false]
IC(p, t) = empty
The form of the rule is given by:
$t: (p,[DT(p, t)]_\oplus \cap [M(p)]_\oplus) \rightarrow (p',[CT(t, p')]_\oplus)$ if $[M(p)]_\oplus \rightarrow \phi_M$
When the place capacity C(p) is finite, the conditional part of the rewrite rule will include the following component:

$[CT(p, t)]_\oplus \oplus [M(p)]_\oplus \cap [C(p)]_\oplus \rightarrow [CT(p, t)]_\oplus \oplus [M(p)]_\oplus$ (**Cap**)

In the case where there is a transition condition TC(t), the conditional part of our rewrite rule must contain the following component: TC(t) → [true].

III. Study of Unbounded Places

The development of an algorithm that detects cases of unbounded places in an ECATNet is a delicate problem. It is about an unbounded place when the number of algebraic terms in this place increases infinitely. The study of case of an unbounded place comes back to study the monotony property. In an ECATNet, this property depends strongly on assignments of algebraic term variables that label arcs joining places and transitions. We separate three cases of ECATNets : simple ECATNet (without conditions of transitions and ECATNet's places with infinite capacity), ECATNet with conditions of transitions and places with infinite capacity, ECATNet without conditions of transitions and places with infinite capacity, and of course the general case. In the case of an ECATNet with places with infinite capacity, the property of the monotony is respected. In the case of the places with finite capacity, the monotony may be respected or not. We focus in our study on the case when IC(p, t) is of the form $[m]_{\oplus}$ and we exclude from the two other cases (IC(p, t) is of the form $\tilde{}[m]_{\oplus}$, and IC(p, t) = empty).

A. ECATNet's Places with Infinite Capacity

We give in following some propositions and their proofs. We focus in these propositions on the ECATNet of the first case ($[IC(p, t)]_{\oplus} = [DT(p, t)]_{\oplus}$). We obtained the same result for the two other cases ($[IC(p, t)]_{\oplus} \cap [DT(p, t)]_{\oplus} = \phi_M$ and $[IC(p, t)] \cap [DT(p, t)] \neq \phi_M$).

B. Absence of Transitions Conditions

Proposition 1. Let M , M' two markings and S a sequence of transitions, if $M \xrightarrow{S}$ and $M \subseteq M'$ then $M' \xrightarrow{S}$

Proof 1. We make call to the proof by recurrence. For $S = t$ (one transition), if t is enabled at M , then we have :

$\forall p \in P \quad IC(p,t) \subseteq M(p)$ or $DT(p,t) \subseteq M(p)$

$\forall p \in P$ if $M(p) \subseteq M'(p)$ then

$IC(p,t) \subseteq M'(p)$ or $DT(p,t) \subseteq M'(p)$

Consequently, t is enabled at M' . Let's assume that this property is verified for $S = t_1..t_k$ and we prove that it is for $S = t_1..t_k t_{k+1}$. We have:

$M \xrightarrow{t_1..t_k} M_k \xrightarrow{t_{k+1}} M_{k+1}$

By supposition, $M \subseteq M'$ then $M' \xrightarrow{t_1..t_k} M'_k$

Now, is t_{k+1} enabled at M'_k ?

We have $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 ... \xrightarrow{t_k} M_k$

$$\forall p \in P \quad M_1(p) = (M(p) \setminus DT(p,t_1)) \otimes CT(p,t_1) \quad (1)$$

If $IC(p,t_1) \subseteq M(p)$ and $DT(p,t_1) \subseteq M(p)$

Such that \ and $\otimes$ are subtraction and union of multi-sets. While $DT(p,t_1) \subseteq M(p)$ then, without risks in multi-sets, we can write :

$$M_1(p) = (M(p) \otimes CT(p,t_1)) \setminus DT(p,t_1) \quad (2)$$

$$M_k(p) = (M_{k-1}(p) \otimes CT(p,t_{k-1})) \setminus DT(p,t_{k-1}) \quad (3)$$

then :

$$M_k(p) = (M_{k-2}(p) \otimes CT(p,t_{k-2})) \setminus DT(p,t_{k-2})) \otimes CT(p,t_{k-1}) \setminus DT(p,t_{k-1}) \quad (4)$$

we can write :

$$M_k(p) = ((M_{k-2}(p) \otimes CT(p,t_{k-2}) \otimes CT(p,t_{k-1})) \setminus (DT(p,t_{k-2}) \otimes DT(p,t_{k-1})) \quad (5)$$

consequently, we can write :

$$M_k(p) = ((M(p) \otimes (\overset{k}{\underset{l=1}{\otimes}} CT(p,t_l))) \setminus (\overset{k}{\underset{l=1}{\otimes}} DT(p,t_l)) \quad (6)$$

moreover, we have also :

$$M'_k(p) = ((M'(p) \otimes (\overset{k}{\underset{l=1}{\otimes}} CT(p,t_l))) \setminus (\overset{k}{\underset{l=1}{\otimes}} DT(p,t_l)) \quad (7)$$

if $M_k(p) \subseteq M'_k(p)$ then

$$M(p) \otimes (\overset{k}{\underset{l=1}{\otimes}} CT(p,t_l)) \subseteq M'(p) \otimes (\overset{k}{\underset{l=1}{\otimes}} CT(p,t_l)) \quad (8)$$

and then

$$M(p) \otimes (\overset{k}{\underset{l=1}{\otimes}} CT(p,t_l))) \setminus (\overset{k}{\underset{l=1}{\otimes}} DT(p,t_l)) \subseteq M'(p) \otimes (\overset{k}{\underset{l=1}{\otimes}} CT(p,t_l)) \setminus (\overset{k}{\underset{l=1}{\otimes}} DT(p,t_l)) \quad (9)$$

because

$$\overset{k}{\underset{l=1}{\otimes}} DT(p,t_l) \subseteq M(p) \otimes (\overset{k}{\underset{l=1}{\otimes}} CT(p,t_l)) \quad (10)$$

that is to say $M_k(p) \subseteq M'_k(p)$, if t_{k+1} is enabled at M_k then it is enabled at M'_k. If $M \xrightarrow{t_1..t_{k+1}}$ and $M \subseteq M'$ then $M' \xrightarrow{t_1..t_{k+1}}$. It means that the property of monotony is verified.

Proposition 2. If $M \xrightarrow{s} M_1$ and $M \subseteq M_1$ then p place, such that $M(p) \subset M_1(p)$ is an unbounded place.

Proof 2. We have in this case :

$M \xrightarrow{s} M_1 ... \xrightarrow{s} M_k$ When k offers toward the infinite with $M \subseteq M_1$ $M_1 \subseteq M_2$ and $M_{k-1} \subseteq M_k$

For $p \in P$ if $M(p) \subset M_1(p)$ then

$$\exists m(p) \neq \phi \quad M_1(p) = m(p) \oplus M(p) \quad (11)$$

(m is non-empty multi-set). On the other hand

$$M_2(p) = ((M_1(p) \otimes (\overset{k}{\underset{l=1}{\otimes}} CT(p,t_l))) \setminus (\overset{k}{\underset{l=1}{\otimes}} DT(p,t_l)) \quad (12)$$

consequently,

$$M_2(p) = ((m(p) \otimes M(p) \otimes (\overset{k}{\underset{l=1}{\otimes}} CT(p,t_l))) \setminus (\overset{k}{\underset{l=1}{\otimes}} DT(p,t_l)) \quad (13)$$

without risk, we write

$$M_2(p) = m(p) \otimes ((M(p) \otimes (\overset{k}{\underset{l=1}{\otimes}} CT(p,t_l))) \setminus (\overset{k}{\underset{l=1}{\otimes}} DT(p,t_l)) \quad (14)$$

this means that

$$M_2(p) = m(p) \otimes M_1(p) \quad (15)$$

or

$$M_2(p) = m(p) \otimes m(p) \otimes M(p) \quad (16)$$

we have

$$m(p) \subset m(p) \otimes m(p) \quad (17)$$

and then

$$M(p) \subset M_1(p) \subset M_2(p) \quad (18)$$

by recurrence, we will have

$$M(p) \subset M_1(p) \subset M_2(p) \subset ... \subset M_k(p) \quad (19)$$

or

$$M_k(p) = \underbrace{m(p) \otimes ... \otimes m(p)}_{k \quad times} \otimes M(p) \quad (20)$$

That is to say, that if k offers toward the infinite, then the number of the algebraic terms in place p increases toward the infinite.

Interpretation 2. For one transition *t*, we have :

$$\forall p \in {}^\bullet t \quad M'(p) = M(p) \setminus DT(p,t) \quad (21)$$

if $IC(p,t) \subseteq M(p)$

$$\forall p \in t^\bullet \quad M'(p) = M(p) \otimes CT(p,t) \quad (22)$$

For $p \in t^\bullet \quad M(p) \subseteq M'(p)$ then

$$M(p) = M'(p) \otimes CT(p,t) \quad (23)$$

It is achieved some either circumstances.

$p \in {}^\bullet t \quad M(p) \subseteq M'(p)$

then $M(p) \subseteq M'(p) \setminus DT(p,t)$

It is possible, only in two cases :

- $DT(p,t) = \phi$, we sensitize without withdrawing
- input place is always output place $DT(p,t) \subseteq CT(p,t)$

We add more algebraic terms than we have just withdrawn.

C. Presence of Transitions Conditions

The presence of a condition for a transition does not make any problems with regard to the preservation of monotony. A transition condition is true if the values which make it true are inside the input places of the transition of this condition. By increasing the multi-sets of terms in these places, these values always exist and the condition is always true. That wants to say, if a transition is enabled since a marking M , it is always a since M' such that $M \subseteq M'$.

D. ECATNet's Places with Finite Capacity

We distinguish two cases. We discuss them in the following propositions:

Proposition 3. If $M_1 \xrightarrow{S} M_2$ and $M_1 \subseteq M_2$ and for every finite place p, $M_1(p) = M_2(p)$, then every infinite place p' such $M_1(p') = M_2(p')$ is an unbounded place.

Proof 3. For simplicity, we only take into consideration $S = t$ (one transition). Let's consider that $P = IP \cup FP$ (IP : Places with infinite capacities, FP : Places with finite capacities)

if t is enabled at M_1, then we have :

$\forall p \in P \quad IC(p,t) \subseteq M_1(p)$ and $DT(p,t) \subseteq M_1(p)$

$\forall p \in P \quad$ if $\quad M_1(p) \subseteq M_2(p)$

then $\quad IC(p,t) \subseteq M_2(p) \quad$ and $\quad DT(p,t) \subseteq M_2(p)$

on the other hand, $\forall p \in FP : M_1(p) \subseteq M_2(p)$ if t doesn't change the marking in finite places because if *t* deletes tokens from a finite place, *t* put the same tokens in this place or *t* is just independent from this place. Then *t* is also enabled at M_2.

$$M_2(p) = (M_1(p) \setminus DT(p,t)) \otimes CT(p,t) \quad (24)$$

because

$$M_1(p) \subseteq (M_1(p) \setminus DT(p,t)) \otimes CT(p,t) \quad (25)$$

then $CT(p,t) \setminus DT(p,t)$ is a positive multi-set. In this case, we can write:

$$M_2(p) = M_1(p) \otimes (CT(p,t) \setminus DT(p,t)) \quad (26)$$

because

$$\forall p \in FP : M_1(p) = M_2(p) \quad (27)$$

we conclude that

$$\forall p \in FP : CT(p,t) \setminus DT(p,t) = \phi \quad (28)$$

we can continue in this way and we get

$$M_3(p) = M_2(p) \otimes (CT(p,t) \setminus DT(p,t)) \quad (29)$$

$$M_3(p) = (M_1(p) \otimes (CT(p,t) \setminus DT(p,t))) \otimes (CT(p,t) \setminus DT(p,t)) \quad (30)$$

$$M_3(p) = M_1(p) \overset{2}{\underset{i=1}{\otimes}} (CT(p,t) \setminus DT(p,t)) \quad (31)$$

t is always enabled at M_3. By recurrence we get :

$$M_k(p) = M_1(p) \overset{k}{\underset{i=1}{\otimes}} (CT(p,t) \setminus DT(p,t)) \quad (32)$$

we put

$$CT(p,t) \setminus DT(p,t) = m(p) \quad (33)$$

and

$$M_k(p) = M_1(p) \overset{k}{\underset{i=1}{\otimes}} (m(p)) \quad (34)$$

we can say $M_1 \xrightarrow{t^*}$

for a place $p \in IP$ with $M_1(p) \subset M_2(p)$, $m(p) \neq \phi$. So, if k goes toward the infinite, then the number of the algebraic terms in place p increases toward the infinite.

Proposition 4. if $M \xrightarrow{S} M'$ and $M \subseteq M'$ and the first transition in S in not enabled since M'. If it exists S' such that $M' \xrightarrow{S'} M''$ S' stops when S become enabled. If we have the following case :

$M \xrightarrow{S} M' \xrightarrow{S'} M'' \xrightarrow{S} M'''$ and if $M' \subseteq M'''$ and $M''(p) \subseteq M(p)$ for each place p with bounded capacity yielding S disabled at M', then every infinite place p' such $M'(p') \subset M'''(p')$ is an unbounded place.

Interpretation 4. The only reason that makes the first transition of S disabled is the problem of the overtaking of certain places capacities. S' makes lower the algebraic terms in places suffering of overflow. For it, S becomes again enabled, then we get M'''. Then, we get the infinity in certain places with infinite capacities.

IV. An Algorithm to Detect Unbounded Places for ECATNets

In this section, we present our algorithm for checking if a given ECATNet is bounded or not. The basic idea of the algorithm consists in computing the accessibility graph (finite) of the ECATNet and checking at the same time the conditions of unbounded places. If one of the unboundeness conditions is true, the algorithm stops computing and returns that ECATNet is not bounded. Otherwise, if the algorithm finishes the accessibility graph, then it returns that the ECATNet is bounded.

First, we define some functions that will be used in the framework of our algorithm:

SubMarking(m, m') : returns true if m is included in m'.

StSubMarking(m, m') : returns true if m is a sub-marking of m' and m ≠ m' (m is strictly sub-marking of m').

ReachableMarking(m, l) : returns a result marking of firing rewriting rule l at m. If ReachableMarking(m, l) = mt, then the rule l is not enabled at m.

GetSubMarkingConcerningPlace(m, p) : gives a marking that is sub-marking of m concerning the place p.

In the following algorithm, we consider **(C1)** and **(C2)** the conditions of the presence of unbounded places in an ECATNet. **(C1)** is the condition that we find it after *if* and **(C2)** is the condition we find it after the *or*.

Algorithm Boundeness Property Decision for ECATNets

Input : N : ECATNet without inhibitor arcs, P: set of places of N, P = FP∪IP, FP : set of finite places, IP : set of infinite places, FP∩IP = ϕ, L : set of transitions (rewriting rules) of N, m_0 : the initial marking.

Output : Decision if an ECATNet is bounded or not

Method : **var** Decision : (Bounded, UnBounded) := Bounded;

1. The root is labeled by the initial marking m_0
2. A marking m doesn't have a successor if and only if:
-for each rewriting rule l, ReachableMarking(m, l) = ϕ
-it exists on the path of m_0 to m another marking m' = m
3. if the two conditions are not verified, let m'' be the marking such $m \xrightarrow{l} m''$ then

for (every rule l, where ReachableMarking(m, l) ≠ ϕ) **&** (Decision = Bounded) **do**

if (∃ m' : marking on the path of m_0 until m **&** m' is a sub-marking of ReachableMarking(m, l)
& ∀ p ∈FP:
GetSubMarkingConcerningPlace(ReachableMarking(m, l), p)=
GetSubMarkingConcerningPlace(m',p)) **(C1)**

or

(∃ m' : marking on the path of m_0 until m **&** m'
is strictly sub-marking of ReachableMarking(m, l)
& ($\exists l_i$ (i =I, 2)∈ L **&** $l_1 \neq l_2$ on the path from m' until m such that
& $\exists m_1, m_2$ two markings on the path from m' until m
& $m' \xrightarrow{l_1 Sl} m_1$ and $m_1 \xrightarrow{l_2 S'} m_2$
and $m_2 \xrightarrow{l_1 Sl}$ ReachableMarking(m, l)
& SubMarking(m', m1) = true
& ReachableMarking(m1, l_1) = ϕ
& ∀ p ∈ FP: GetSubMarkingConcerningPlace(m2, p)
is sub-marking of GetSubMarkingConcerningPlace(m', p) **(C2)**

then Decision := UnBounded;

else m'' = ReachableMarking(m, l);

return (Decision);

V. Meta-Level Computation in Maude

Maude provides a platform getting easy implementation of ECATNets' tool. Meta-level description is one of services provided by Maude. This service permits describing a module in meta-level. This module becomes an input to another module. We will use meta-level representation in Maude to describe an ECATNet and act on it. The syntax of meta-level representation is different from ordinary representation in Maude. Term and module in the meta-level are called meta-term and meta-module respectively. Meta-term is considered as term of a generic type called Term. A Meta-module is considered as a term of generic type called Module. To manipulate a module in meta-level representation, Maude provides a module called META-LEVEL. This module encapsulates some services called descent functions. A descent function performs reduction and rewriting of meta-term, according to the equations and rules in the corresponding meta-module, at the meta-level.

Function metaApply. metaApply is the process of applying a rule of a system module to a term:

```
sort ResultTriple ResultTriple? .
subsort ResultTriple < ResultTriple? .
op {_,_,_} : Term Type Substitution –> ResultTriple [ctor] .
op failure : –> ResultTriple? [ctor] .
op metaApply : Module Term Qid Substitution Nat –> ResultTriple? .
```

The first four parameters are representations in meta-level of a module, a term in a module, some rules label in the module and a set of assignments (possibly empty) defining a

partial substitution for variables being in this rule. The last parameter is a natural number. Given a natural N as a fifth parameter, metaApply function returns the $(N+1)^{th}$ result of application of every substitution. In our application, we don't need any substitution, than we take the empty substitution none. In this case, we take 0 as last parameter. For more details about the two last parameters see [3]. This function returns a triple formed by result term, type of this term and substitution, otherwise metaApply returns 'failure'.

Function getTerm. We apply getTerm function on metaApply to extract only the resulting term:

```
op getTerm : ResultTriple –> Term .
```

VI. Implementation of the Algorithm in Maude

In the framework of this work, we used as platform Maude in its version 2.0.1 under Windows-XP. The development of this application is not very complicated thanks to the meta-computation concept in Maude. Many details are adjusted by the ECATNets description in Maude. For instance, a transition can have a condition that will be integrated in the rule. The call of metaApply function allows to evaluate the condition and to free our application to deal with this detail. Let's consider textual version of ECATNets in Maude. First, we present a generic module that describes basic operations for ECATNet:

```
fmod GENERIC-ECATNET is
  sorts Place Marking GenericTerm .
  op mt : -> Marking .
  op <_;_> : Place GenericTerm -> Marking .
  op _._ : Marking Marking -> Marking [assoc comm id: mt] .
endfm
```

As illustrated in this code, mt is the empty marking implementing ϕ_M. Respecting some syntactical constraints in Maude language, we define the operation "<_;_>" which permits the construction of elementary marking. The two underlines indicate the positions of the operation's parameters. The first parameter of this operation is a place and the second one is an algebraic term (marking) in this place.

A. Reachability Graph Representation.

A triple < T ; L ; T' > means that the rewriting of T (T for Term) is T' by using the rule L. This is equivalent to say that the firing of the transition represented by the rule L at the marking represented by T gives the marking represented by T'. The accessibility graph will be represented by a list of this kind of triples.

B. Application's Functions

Many functions are developed in functional programming paradigm to implement the above algorithm. In this section, we describe in detail how we realized some of these functions. We present also some basic sorts.

We define a sort BoundnessData containing two constants Bounded and UnBounded. An element of the sort Decision is a couple of an element ListOfTriple (accessibility graph under construction) and an element of sort BoundnessData. The operation 1st which is applied on a pair of sort Decision returns the first element of this pair that is of sort ListOfTriple.

```
sorts BoundnessData Decision .
ops Bounded UnBounded : -> BoundnessData .
op _;_ : ListOfTriple BoundnessData -> Decision .
op 1st : Decision -> ListOfTriple .
eq 1st((LT ; Bd)) = LT .
op 2nd : Decision -> BoundnessData .
eq 2nd((LT ; Bd)) = Bd .
```

The function ReachableMarking(M, T, L) returns a successor marking of the whole T by applying L, otherwise, the function returns the empty marking mt. M is a module representing ECATNet system:

```
op ReachableMarking : Module Term Qid -> Term .
eq ReachableMarking(M, T, L) =
if metaApply(M, T, L, none, 0) =/= failure
 then if getTerm(metaApply(M, T, L, none, 0)) == 'mt.Marking
        then 'mt.Marking
        else  getTerm(metaApply(M, T, L, none, 0)))    fi
 else mt   fi .
```

The function AccessibleMark(M, T, L), with T a term to compute its successor by firing rule L, this function returns, in success case, a triple of the form < T ; L ; RT > such that RT is the successor marking of T after firing L:

```
op AccessibleMark : Module Term Qid -> Triple .
eq AccessibleMark(M, T, L) = MediumAccessibleMark(M, T,
GetAllSubTerms(T), L) .
```

This function calls the function MediumAccessibleMark(M, T, GetAllSubTerms(T), L). The function GetAllSubTerms(T) returns a stack of all sub-terms of T. This is necessary because metaApply(M, T, L, none, 0) gives a result term if and only if the whole term T matches exactly the left hand side of a rule L. For a super term of T and which is different from it, this function doesn't give a result term. In this case, we proceed to the decomposition of the term by extracting all its sub-terms components. Then, we apply to each sub-term top(S) the ReachableMarking(M, top(S), L) function for rule L. The sub-term component that is a left part of this rule is subtracted from its super-term. The result RT of the subtraction is added to the result term of ReachableMarking(M, top(S), L).

```
op MediumAccessibleMark : Module Term Stack Qid -> Triple .
eq MediumAccessibleMark(M, T, S, L) =
if S == emptystack
then errorltt
else  if ReachableMarking(M, top(S), L) == mt
       then  MediumAccessibleMark(M, T, pop(S), L)
       else  if StackCompareEqTerms(T, top(S)) == true

then if
CapacityCheckingInPlaces(ReachableMarking(M, T, L)) == true
      then < T ; L ; ReachableMarking(M, T, L) >
      else MediumAccessibleMark(M, T, pop(S), L)
      fi
else if
```

```
CapacityCheckingInPlaces(TermAddition(TermSubstractionExt(T,
top(S)),  ReachableMarking(M, top(S), L))) == true

    then < T ; L ; TermAddition(TermSubstractionExt(T, top(S)),
                  ReachableMarking(M, top(S), L)) >

    else MediumAccessibleMark(M, T, pop(S), L)
  fi  fi  fi  fi .
```

The main function of our application is Decidability-Decision(M, T0). This function is an interface with the user:

```
op Decidability-Decision : Module Term -> BoundnessData .
eq Decidability-Decision(M, T0) = 2nd(AllAccessibleMark(M, T0,
T0, emptystack, emptystack, GetRulesName(M), T0, emptyt, IPs,
FPs, GetRulesName(M))) .
```

Decidability-Decision(M, T0) calls and initializes parameters of AllAccessibleMark(M, T0, S, S1, S', LS, APath, LT, IP, FP, LS1) function. Let's note that FP is the set of places with finite capacity and S is a stack containing terms markings to be treated. When we obtain successors markings of top(S), we put it first in S1. If S becomes empty, we pass to deal with markings in S1 and so we put the content of S1 in S. S' is a stack containing markings that are dealt with. This is important in order to avoid looking for successors marking for those that are already treated. LS is a list containing initially all labels of module's rules and LS1 is a list always containing all labels of module's rules (GetRulesName(M)). APath is a stack of term lists, when each term list is a path. The first term of this list (path) is the initial marking and the last one is a marking waiting to compute its accessible markings. LT contains the coverability graph created until this moment :

```
op AllAccessibleMark : Module Term Stack Stack Stack List Stack
ListOfTriple List List List -> Decision .
eq AllAccessibleMark(M, T0, S, S1, S', LS, APath, LT, IP, FP, LS1)
=
if S == emptystack
 then if S1 == emptystack
       then emptyt ; Bounded
 else AllAccessibleMark(M, T0, S1, emptystack, S', LS1, APath,
      LT, IP, FP, LS1)
 fi
 else  if LS == emptyList
       then AllAccessibleMark(M, T0, pop(S), S1, push(top(S), S'),
            LS1, APath, LT, IP, FP, LS1)
       else if findTermInStack(top(S), S') == true
            then AllAccessibleMark(M, T0, pop(S), S1, S', LS1,
                 APath, LT, IP, FP, LS1)
            else  if  AccessibleMark(M, top(S), head(LS)) ==
                      errorltt
                  then AllAccessibleMark(M, T0, S, S1, S',
                       tail(LS), APath, LT, IP, FP, LS1)
                  else

if (SubMarkingFoundOnPath(M, T0, top(S),
   3rd(AccessibleMark(M, top(S), head(LS))), APath, head(LS))
   =/= nil
  and  ConstantMarksInFPlaces(M,
       SubMarkingFoundOnPath(M, T0, top(S),
       3rd(AccessibleMark(M, top(S), head(LS))),
       APath, head(LS)),
       3rd(AccessibleMark(M, top(S), head(LS))), FP) == true)

 or (SubMarkingFoundOnPath(M, T0, top(S),
    3rd(AccessibleMark(M, top(S), head(LS))), APath, head(LS))
    =/= nil
    and TwoMarkingsFoundRef(M, SubMarkingFoundOnPath(M,
          T0, top(S), 3rd(AccessibleMark(M, top(S),
          head(LS))), APath, head(LS)),  top(S), LT, APath,
          head(LS), FP) == true)

then emptyt ; UnBounded

else ConcatListOfTriple(AccessibleMark(M, top(S), head(LS)),

1st(AllAccessibleMark(M, T0, S,
 DeleteDupInStackExt(TrListToStack(AccessibleMark(M, top(S),
head(LS)), S1)),  S', tail(LS), PathSCreation(top(S),
GetRTerms(AccessibleMark(M, top(S), head(LS))), APath),
ConcatListOfTriple(AccessibleMark(M, top(S),
head(LS)), LT),   IP, FP, LS1))) ;

2nd(AllAccessibleMark(M, T0, S,
DeleteDupInStackExt(TrListToStack(AccessibleMark(M, top(S),
head(LS)), S1)), S', tail(LS), PathSCreation(top(S),
GetRTerms(AccessibleMark(M, top(S), head(LS))), APath),
ConcatListOfTriple(AccessibleMark(M, top(S), head(LS)), LT),
IP, FP, LS1))
 fi  fi  fi  fi  fi .
```

We compute successor marking of top(S) with each rule head(LS) in LS. Each time LS becomes empty, we reinitialize its content with LS1 to continue accessible markings' computation for another marking. If S and S1 are empty, there is no marking to compute its successors. This marks the end of the computation. If S is empty and not S1, we put the contents of S1 in S, this permits to compute the accessible markings from those in S1. It is necessary the reinitialization of LS by LS1.

In the case when S isn't empty (i.e, we have markings to compute their successors), we need to verify some conditions before computing the successors of top(S). First, we check if LS isn't empty and if top(S) doesn't exist in S'. Because if LS is empty, this means that we have already computed all accessible markings of top(S), so we discard it to S', and if top(S) exists in S', so top(S) is already treated before.

If such conditions are checked, we can proceed to compute accessible marking of top(S) with head(LS) rule. For that, we call AccessibleMark(M, top(S), head(LS)). This function insures that head(LS) rule is enabled or not at top(S) marking. The term errorltt indicates that head(LS) is not enabled at top(S). In this case, we discard this rule and we continue to see if there are others rules (tail(LS)) enabled at top(S). If head(LS) is enabled at top(S), we check if conditions of unbounded places are valid or not. This is expressed by the condition (T' =/= nil and ConstantMarksInFPlaces(M, T', RT, head(LS), FP) == true). The first condition (T' =/= nil) implements the condition **(C1)** of the algorithm and the second one implements **(C2)**. If the conditions are true, we stop computing reachability graph and we return UnBounded. Let's note that 3rd(AccessibleMark(M, top(S), head(LS))) returns the third element in the triple (result term of firing

head(LS) at top(S)). TrListToStack(AccessibleMark(M, top(S), T', head(LS), IP), S1) puts in S1 the third element in the triple AccessibleMark(M, top(S), head(LS), IP), and DeleteDupInStackExt eliminates any duplication in the result stack. PathSCreation(top(S), GetRTerms(AccessibleMark(M, top(S), head(LS), IP)), APath) puts the new created accessible marking in its appropriate place in APath. Finally, ConcatListOfTriple(AccessibleMark(M, top(S), head(LS)), LT) allows adding new created triple AccessibleMark(M, top(S), head(LS), IP) to existent coverability graph (LT).

VII. EXAMPLE

The subject of this section is the application of the proposed tool on a simple industrial case. This example is presented in [6] and it is described by using ECATNets formalism in [7]. We take this description with some modifications. This example presents an infinite-state real system.

A. Example Presentation

The example is about a cell of production that manufactures forged pieces of metal with the help of a press. This cell is composed of a table A that serves to feed the cell by raw pieces, of a robot of handling, a press and a table B that serves to the storage of forged pieces. The robot includes two arms, disposed at right angles on one same horizontal plan, interdependent of one same axis of rotation and without vertical mobility possibility. The figure 2 represents the spatial disposition of elements of the cell. The robot can seize a raw piece of the table A and to put down it in the press with the help of the arm 1. It can also seize a forged piece of the press and can put down it on the table of storage B with the help of the arm 2. In short, the robot can do two movements of rotation. The first allows it to pass from its initial position to its secondary one. This movement permits the robot to deposit a raw piece in the press and possibly the one of a forged piece on the table of storage B. The second allows it to pass from its secondary position towards its initial position and to continue the cycle of rotation.

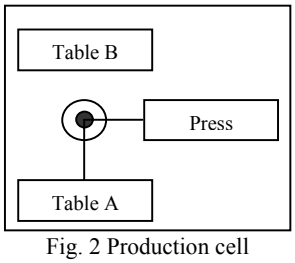

Fig. 2 Production cell

B. ECATNets Model of the Example

Figure 3 represents the ECATNets model of production cell. The symbol ϕ is used to denote the empty multi-set in arcs inscriptions. Please note that *r* denotes 'raw' and *f* denotes 'forge'. If the inscriptions IC(p, t) and DT(p, t) are equals, then we only present IC(p, t) on the arc (p, t). The rewriting rules of the system will be presented in the following section directly in Maude.

ECATNets Places.

Ta : table A ; set, possibly empty, of raw pieces.
Tb : table B ; set, possibly empty, of raw pieces.
Ar1 : arm 1 of robot ; at most a raw piece.
Ar2 : arm 2 of robot ; at most a forge piece.
Pr : press ; at most a raw piece or a forge piece.
PosI : initial spatial position of robot ; it is marked "ok" if it is the current position of robot.
PosS : secondary spatial position of robot ; it is marked "ok" if it is the current position of robot.
EA : this place is added for testing if the two arms of robot are empty.

ECATNets Transitions.
T1 : taking of a raw piece by the arm 1 of the robot.
T2 : taking of a forge piece by the arm 2 of the robot.
D1 : deposit of a raw piece in the press.
D2 : deposit of a forge piece on the table B.
TS1, TS2 : rotation of the robot from its initial position towards its secondary position.
TI : rotation of the robot from its secondary position towards its initial position.
F : forge of the raw piece introduced in the press.
E : deposit of a raw piece on the table A.
R : removing forge pieces from the table B.

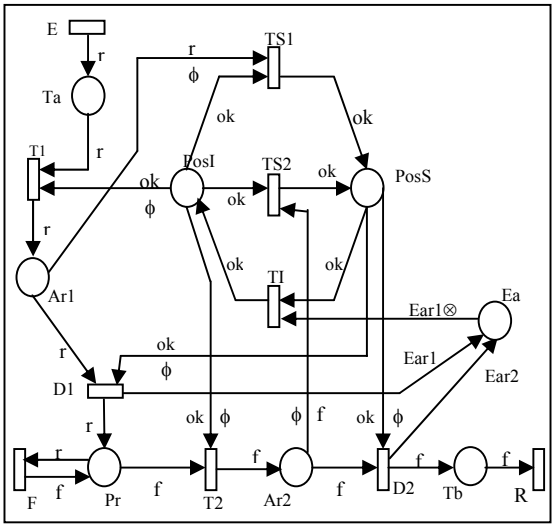

Fig. 3 ECATNet model of the cell

C. Meta-level Representation of the Example in Maude

The user is not obliged to write (his/her) ECATNet in a meta-representation. (He/she) can write it in the common mode, and then (he/she) uses the function of Maude upModule which allows transforming the representation of a module to its meta-representation. The passing in the other direction also is possible, thanks to the function downModule. For more of clarity, we preferred to give the module describing the previous ECATNet in its meta-representation. In the module META-LEVEL-ROBOT-ECATNET-SYSTEM, the META-ROBOT module is defined as a constant of type Module and its contents are described by means of an equation. 'GENERIC-ECATNET is the description in meta-level of module GENERIC-ECATNET

described previously. For more simplicity, we only present some rewriting rules describing Robot behavior.

```
mod META-LEVEL-ROBOT-ECATNET-SYSTEM is
....
op META-ROBOT : -> Module .
eq META-ROBOT = (mod 'META-ROBOT is
protecting ‘GENERIC-ECATNET .  protecting 'INT .

sorts 'Cointype ; 'RPosType ; 'EmptyArmType  .
subsort 'Cointype < 'GenericTerm .
subsort 'RPosType < 'GenericTerm .
subsort 'EmptyArmType < 'GenericTerm .
subsort 'Place < 'Marking .

op 'ok : nil -> 'RPosType [ctor] .
op 'Ear1 : nil -> 'EmptyArmType [ctor] .
op 'Ear2 : nil -> 'EmptyArmType [ctor] .
op 'raw : nil -> 'Cointype [ctor] .
op 'forge : nil -> 'Cointype [ctor] .

op 'Ta : nil -> 'Place [ctor] . op 'Tb : nil -> 'Place [ctor] .
op 'Pr : nil -> 'Place [ctor] . op 'Ar1 : nil -> 'Place [ctor] .
op 'Ar2 : nil -> 'Place [ctor] . op 'PosI : nil -> 'Place [ctor] .
op 'PosS : nil -> 'Place [ctor] . op 'Ea : nil -> 'Place [ctor] .
none none

rl 'm:Marking => '_._['<_;_>['Ta.Place, 'raw.Cointype], 'm:Marking]
[label('E)]  .

rl  '_._['<_;_>['PosS.Place,  'ok.RPosType],  '<_;_>['Ar2.Place,
'forge.Cointype]]  =>  '_._['<_;_>['Tb.Place,  'forge.Cointype],
'_._['<_;_>['Ea.Place,  'Ear2.EmptyArmType],  '<_;_>['PosS.Place,
'ok.RPosType]]] [label('D2)]  .

rl '<_;_>['Tb.Place, 'forge.Cointype] => 'mt.Marking [label('R)]  .
...
endm) .  endm
```

VIII. Application of the Tool on the Example

To apply the tool on the example, we call the main function of the application Decidability-Decision(META-ROBOT, '_._['<_;_>['PosI.Place, 'ok.RPosType], '<_;_>['Ea.Place, 'Ear2.EmptyArmType]]). The example is about an unbounded ECATNet. It’s clear that the transition ‘E is always enabled and so ‘Ta is an unbounded place. Consequently, the application of the tool on this example returns Unbounded as result (figure 4).

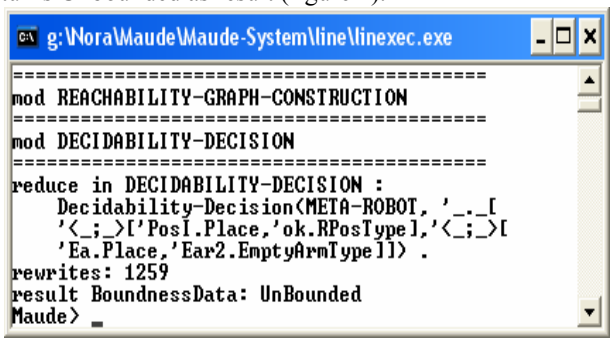

Fig. 4 Application of the boundness property checker on the ECATNet example

IX. Conclusion

In this paper, we proposed an algorithm and its Maude based tool to check boundness property for ECATNets. Such algorithm is motivated by the fact that analysis techniques like reachability analysis and Model Checking of Maude cannot deal with infinite-state model including unbounded ECATNets. The tool aims to inform us if our ECATNet is bounded or not. In this case, we can deduce if we can apply or not the accessibility analysis and the Model Checking of Maude on this ECATNet. The development of this tool is not very complicated thanks to the reflectivity of Maude language and the integration of the ECATNets formalism in this language.

References

[1] G. Berthelot, C. Johnen, and L. Petrucci. “PAPETRI : environment for the analysis of Petri nets”. Volume 3 of Series in Discrete Mathematics and Theoretical Computer Science (DIMACS), p. 43-55. American Mathematical Society, 1992.

[2] M. Bettaz, M. Maouche. “How to specify Non Determinism and True Concurrency with Algebraic Term Nets”. Volume 655 of LNCS, Spring-Verlag, 1993, pp. 11-30.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. “The Maude 2.0 System”. In Proc. Rewriting Techniques and Applications (RTA), Volume 2706 of LNCS, Spring-Verlag , 2003, pp. 76-87.

[4] A. Finkel. “The Minimal Coverability Graph for Petri Nets”. In: Rozenberg, G.: Volume 674 of LNCS,; Advances in Petri Nets 1993, Springer-Verlag, 1993, pp. 210-243.

[5] C. Girault and P. Estraillier, “CPN-AMI”. MASI Lab, University Paris VI, France.

[6] T. Lindner. "Formal Development of Reactive Systems : Case Study Production Cell". Volume 891 of LNCS, Spring-Verlag, 1995, pp. 7-15.

[7] M. Maouche, M. Bettaz, G. Berthelot and L. Petrucci. “Du vrai Parallélisme dans les Réseaux Algébriques et de son Application dans les Systèmes de Production”. Conférence Francophone de Modélisation et Simulation (MOSIM’97), Hermes, 1997, pp. 417-424.

[8] J. Meseguer. “Rewriting Logic as a Semantic Framework of Concurrency: a Progress Report”. Seventh International Conference on Concurrency Theory (CONCUR'96), Volume 1119 of LNCS, Springer Verlag, 1996, pp. 331-372.

[9] J. Meseguer. “Rewriting logic and Maude: a Wide-Spectrum Semantic Framework for Object-based Distributed Systems”. In S. Smith and C.L. Talcott, editors, Formal Methods for Open Object-based Distributed Systems, (FMOODS’2000), 2000, pp. 89-117.

[10] V. Pinci and L. Zand. “DESIGN/CPN”. USA, 1993.

[11] S. Roch and P.H. Strake. “INA (Integrated Net Analyzer) : Version 2.2”. Manual, Humboldt-Universität zu Berlin Institut für Informatik, Lehrstuhl für Automaten- und Systemtheorie, 1999.

[12] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. “PROD reference manual”. Technical Report B13, Helsinki University of Technology, Digital Systems Labora tory, Espoo, Finland, August 1995.