World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:2, No:10, 2008

Double Reduction of Ada-ECATNet Representation using Rewriting Logic

Noura Boudiaf, and Allaoua Chaoui

Abstract**—**One major difficulty that faces developers of concurrent and distributed software is analysis for concurrency based faults like deadlocks. Petri nets are used extensively in the verification of correctness of concurrent programs. ECATNets [2] are a category of algebraic Petri nets based on a sound combination of algebraic abstract types and high-level Petri nets. ECATNets have 'sound' and 'complete' semantics because of their integration in rewriting logic [12] and its programming language Maude [13]. Rewriting logic is considered as one of very powerful logics in terms of description, verification and programming of concurrent systems. We proposed in [4] a method for translating Ada-95 tasking programs to ECATNets formalism (Ada-ECATNet). In this paper, we show that ECATNets formalism provides a more compact translation for Ada programs compared to the other approaches based on simple Petri nets or Colored Petri nets (CPNs). Such translation doesn't reduce only the size of program, but reduces also the number of program states. We show also, how this compact Ada-ECATNet may be reduced again by applying reduction rules on it. This double reduction of Ada-ECATNet permits a considerable minimization of the memory space and run time of corresponding Maude program.

Keywords**—**Ada tasking, ECATNets, Algebraic Petri Nets, Compact Representation, Analysis, Rewriting Logic, Maude.

I. Introduction

ONE of the most attractive features of the Ada programming language is the tasking, which permits concurrent execution within Ada programs [11]. The presence of concurrency greatly complicates analysis, testing and debugging of code. The expression of concurrency is achieved by the Ada tasking and rendez-vous. So, much effort is focused on these mechanisms. To do such analysis, we often find the utilization of Petri nets formalism [14], [15], [10]. The choice of this formalism for the verification of the Ada programs is reasonable, seen its strength to describe the dynamic behavior of concurrent program. Others preferred high-level Petri nets [7], [9] to analyze Ada programs. This choice is motivated by the strength of CPNs unlike ordinary Petri nets to describe both static and dynamic aspects of a system, which is a natural need to serve the analysis of the Ada programs in a satisfactory manner. On this path, we adopt the utilization of ECATNets [2] to translate an Ada concurrent program in order to verify it. As a kind of algebraic Petri nets, ECATNets bring more intuitive description for Ada-95 constructs. ECATNets are a category of algebraic nets based on a safe combination of algebraic abstract types and high-level Petri nets. In our sense, they present strength of expression enough for describing many concepts in Ada-95 and particularly the concept of task. The choice of ECATNets is motivated by their 'sound' and 'complete' semantics because of their integration in rewriting logic [12] and so its language Maude [13]. Moreover, ECATNets have already a strong battery of description and some analysis tools, such as static analysis [3], reduction rules [5], [6], reachability analysis and Model Checking of Maude; all are based on only one logic, the rewriting logic. Rewriting logic is considered as one of very powerful logics in terms of description, verification and programming of concurrent systems. The integration of ECATNets in rewriting logic allows them to benefit from Maude all development theories [8] and tools such as simulation, accessibility analysis and Model Checking techniques.

Intuitively, ECATNets formalism presents a very compact representation. Then, the ECATNets obtained as a result of translating an Ada program are relatively minimal and reduced. A concept of the Ada language can be comfortably translated to the ECATNet with a minimal number of places and transitions. Ada-ECATNet proposed in [4] and the reduced Ada-ECATNet are equivalent. In reduced Ada-ECATNet presented in this paper, we just 'skip' intermediate states that are not necessary for the verification of properties related to concurrency. In all existing approaches concerning the utilization of Petri nets (simple or high level) in the description and the verification of the Ada's programs, we notice that these works first aim to translate Ada-programs to Petri nets and then apply reduction rules on the obtained Ada-nets even in the [9]. Quasar tool developed in [9] is a complete environment for Ada-nets analysis by using CPNs. In this work, authors translate Ada programs first to CPNs and they reduce obtained Ada-nets after. But, in the present paper, the proposed reduction rules may be done during translation step. Such translation doesn't reduce just formally the size of program, but it minimizes effectively the number of program states. We can present two or many statements in a sequential bloc by using just one transition in ECATNets. This permits to reduce considerably the number of rewriting steps in the appropriated Maude program. So, the memory space and run time of Maude program are reduced. We will confirm such deduction through an example. We show how refinement rules reduce execution steps in case of simulation, reachability analysis and Model Checking. This proposed reduction is specific to Ada-ECATNet. Therefore, the obtained reduced Ada-ECATNet may be submitted to another reduction such that proposed for APNs. This is possible because we adapted

Noura Boudiaf is with Cedric-CNAM, Paris, France. (e-mail: boudiafn@yahoo.com).

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

and implemented reduction rules defined by Schmidt [16] to ECATNets in [5], [6]. This double reduction allows a meaningful decrease of the complexity of state-space analysis. In this direction, we will show how we apply on Ada-ECATNet, the reduction rule 'Parallel Places' adapted to ECATNets in [6]. In this paper, we propose some refinement rules to translate Ada-Statements to an ECATNet. In such a way, we present compactly many statements in one transition.

The rest of this paper is organized as follows. In section 2, we give a general description of ECATNets. In section 3, we present some proposed translation guidelines and application of our ideas through an example. In section 4, the main reduction rules are proposed. In section 5, we show how we apply on an example our proposed reduction rules as well the reduction rule 'parallel places'. Applications of some analysis methods as simulation, accessibility analysis and Model Checking on Ada-ECATNet are discussed in section 6. Results obtained by using our defined reduction rules are given in section 7. Finally, we conclude the paper in the section 8.

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 : $s = (p, a \oplus b \oplus c)$.

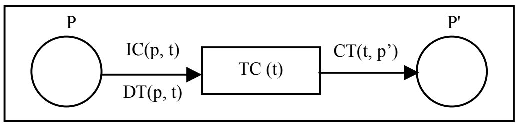


Fig. 1 A generic ECATNet

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 → 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. We now 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 $\sim[m]_{\oplus}$

The form of the rule is given by:

$t : (p, [DT(p, t)]_{\oplus} \cap [M(p)]_{\oplus}) \rightarrow (p', [CT(t, p')]_{\oplus})$
$\text{if } ([IC(p, t)]_{\oplus} \setminus ([IC(p,t)]_{\oplus} \cap [M(p)]_{\oplus})) = \phi_M \rightarrow [\text{false}]$

IC(p, t) = empty

The form of the rule is given by:

$t: (p,[DT(p,t)]_{\oplus} \cap [M(p)]_{\oplus}) \rightarrow (p',[CT(t,p')]_{\oplus}) \text{ if } [M(p)]_{\oplus} \rightarrow \phi_M$

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: $TC(t) \rightarrow [\text{true}]$.

III. Some Guidelines of Translation from Ada to ECATNet through an Example

Most concepts of Ada translation to ECATNets are defined in [4]. For lack of space reason, we give here just some ideas about the translation process through an example.

A. *Example Presentation*

The following segment of Ada program defines a buffer reached by producing and consuming task. Producing task might have the following structure:

```
task body Producer
Char : Character;
begin   loop … -- produce the next character Char
Buffer.Write(Char) ;   exit when Char = ASCII.EOT ;    end loop;
end Producer;
```

Buffer contains an internal Pool of the managed characters. This space has two indices, In_index denotes the place of the next input character and Out_Index denotes the place of the next output character.

```
protected Buffer is
entry Write(C :in character);    entry Read(C : out character);
private    Pool : Array[1..10] of character; Count : Natural := 0 ; In_Index,
Out_Index : positive := 1;
end Buffer;
protected body Buffer is
entry Write(C : in  character) when Count < Pool'length is
   begin Pool(In_Index) := C; In_Index := (In_Index mod Pool'length) + 1;
Count := Count + 1;  end Write;
…
end Buffer;
```

B. *Translation of the Ada Segment to ECATNets*

Types like character, positive, arrays and queues are translated to equivalent abstract data types in ECATNets. We define a sort 'Producer' to represent task type producer. In this case, a producer task is an algebraic term constant 'Pr' of sort 'Producer'. We use an n-tuple algebraic term composed of algebraic terms that represent 'task' and its 'local variables'. The translation of entry Write gives the ECATNet in figure 2, where: Pr: producing task, BF:Buffer, P: Pool, CT: count, II: In_Index, and IO: Out_Index. For this entry, we associate two places to manage the queue containing waiting tasks calling this entry. One place TaskAskWrite serves to manage the order of task arrival and it must have the maximal size of one task. This last must be transferred to the queue of the entry that is in the other place WriteQueue. The TaskAskWrite and AcWrite places have a maximal capacity of one token. We have a condition isempty(q) == false for the transition TaskSelectWrite. For the translation of a protected type, we create a place containing an n-uple composed of its variables (place Buf). The n-uple (Bf,P,CT,II,OI) waits in this place to be dealt by the entry Write or Read. If the token (Pr, Ch) is in AcWrite and the token (Bf,P,CT,II, OI) is in Buf, the rendez-vous can take place. The entry Write has a guard which is translated directly to the condition of the corresponding transition WriteEntry. When the rendez-vous takes place, the firing of the transition WriteEntry removes (Bf,P,CT,II,OI) and (Pr, Ch) from appropriate places. Removing (Bf,P,C,II,OI) from place Buf guarantees that another entry, procedure or a function can not be executed at the same time. So, another task can not execute entry Read while entry White is in evolution. When the rendez-vous takes place, we integrate Pr and Ch in the token representing Buffer. Ch gives its value to the variable C according to the mode '**in**' of parameters passing. A statement is translated to a transition. The transition S3Write translates the assignment statement Count := Count+1;. This transition transforms the token (Pr,Bf,P,CT,II,OI,C) to (Pr,Bf,P,CT+1,II,OI,C) where CT is replaced by CT+1.

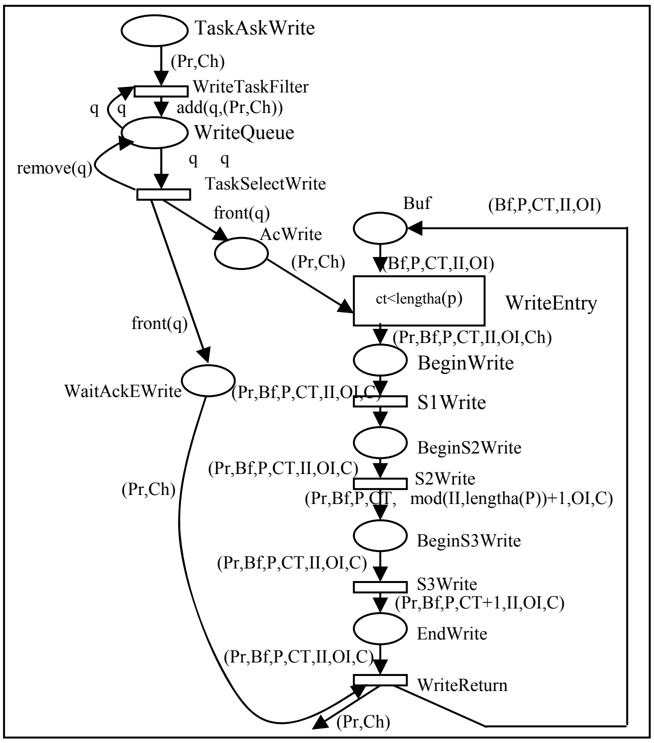


Fig. 2 Representation of entry Write of Buffer type by ECATNets

C. *Mapping the Obtained Ada-ECATNet to Maude*

Among kinds of modules defined in Maude, we find functional and system modules. Functional modules are used to define data types and functions on these types through theories of equations. System modules are used to define the dynamic behavior of a system. This kind of modules adds rewriting rules to the concepts defined by functional modules: sorts, subsorts, and equations. A maximal degree of concurrency is offered by this kind of modules. The following module is part of the developed code which is executable under Maude system.

```
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 an empty marking of a full ECATNet. We define the operation "<_;_>" which permits the construction of elementary marking. The two Underlines indicate the positions of operation's parameters. The first parameter of this operation is a place and the second one is an algebraic term (marking) in this place. We have not defined an operation to implement the operation ⊕. The operation "_._" which implements the operation ⊗ is sufficient while basing on the concept of decomposition. If a place contains many terms, for example (p, a ⊕ b ⊕ c), then we can write it as (p, a) ⊗ (p, b) ⊗ (p, c). Now, we give a part of module implementing the ECATNet buffer: BUFFER which calls BUFFER-DATA module. This last is a functional module calling all functional modules concerning descriptions of types used by system module BUFFER such as List, Queue, Array, Consumer and Producer. We describe data types like Queue of this ECATNet in a hierarchy of functional modules when we declare that Queue as sub-sort of GenericTerm to be able to have a Queue as second parameter of "<_;_>":

```
mod BUFFER is
protecting BUFFER-DATA .
...
ops TaskAskWrite WriteQueue AcWrite WaitAckEWrite BeginWrite
BeginS2Write BeginS3Write EndWrite : -> Place .  op Buf : -> Place .
var P : Array . var q : Queue . vars C Ch : EltArray .vars II OI CT : Int .
var CharL : List . var Pr : Producer .eq EOT = endoflist .
…  *** rules for Write
rl [WriteTaskFilter] : < TaskAskWrite ; (Pr , Ch) >
. < WriteQueue ; q >  => < WriteQueue ; addq(q, (Pr ,, Ch)) > .
…
endm
```

Note. For simplicity and to simulate the production of the next character Char in the ECATNets, we introduce InitialCharList place containing a list of characters. The producer takes each time a character from the list in this place and put it in the buffer. The consumer takes a character from the buffer and put it in the list in a defined place CharListResult.

IV. Reduction Rules

We have defined some refinement rules leading to an effective reduction of the Ada-ECATNet's size. But, we present in this section only two rules:

Rule 1. Concerning a sequence of assignment statements. First, we study the case of two statements. Then, we generalize the rule to many statements. For two assignment statements, $x{:}=e_1$, $y{:}=e_2$ represented by the ECATNet in figure 3 (a), we can obtain an ECATNet with only one transition instead of two, by replacing the occurrence of x by e_1 in the expression e_2 in $y{:}=e_2$. We put $y{:}= e_2[x/e_2]$. Then we have the new two statements $x := e_1$ and $y{:}= e_3$. These ones are represented with the ECATNet of figure 3 (b). In general, let I_1 the statements sequence $x_1{:}= e_1$, $x_2{:}= e_2$, …, $x_n{:}=e_n$, then we proceed as follow: In $x_2{:}=e_2$, we replace each occurrence of x_1 in e_2 by e_1. We obtain a new statements sequence I_2. Recursively, we take $x_i{:}=e_i$ statement and we replace in e_i each occurrence of x_i, …, x_{n-1} by the right hand sides of the appropriate assignments defined in I_{i-1} sequence. We obtain in this case a new statements sequence I_i. The operation terminates after getting the last assignment statements sequence I_n. This is the sequence which will be modeled by one transition with the help of an ECATNet.

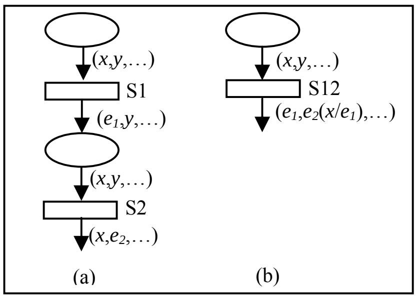


Fig. 3 Ada-ECATNets before (a) and after (b) applying rule

Rule 2. Concerning a sequence of assignment statements followed by a call of a procedure (or call of a function). Figure 4 (a) represents an ECATNet with two transitions: the first one represents an assignment statement and the other represents the call of a procedure. We can integrate them in one transition as pictured in figure 4 (b). Such integration is obtained by the refinement of the sequence of assignments in the way presented in rule 1. To deal with parameters, we do not put as parameters variables name but their equivalent right hand sides of assignment statements found in refined sequence.

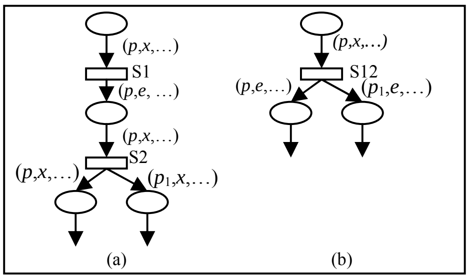


Fig. 4 Ada-ECATNet before (a) and after (b) applying

V. Application of Reduction Rules on the Example

In this section, we describe how we apply refinement rules and the reduction 'parallel places' on the obtained Ada-ECATNet.

A. Application of Ada-ECATNet Reduction Rules on the Example

The application of rules defined above on entry Write of Buffer type gives a compact representation in figure 5. In Maude program, we keep rules WriteTaskFilter and WriteTaskSelect without any change. But, we merged the remaining five transitions to only one transition:

```
crl [WriteS123EntryReturn] :  < Buf ; (BF , P , CT , II , OI) >
. < AcWrite ; (Pr ,, Ch) > . < WaitAckEWrite ; (Pr ,, Ch) >
 => < Buf ; (BF , set(P, Ch, II) , (CT + 1), ((II rem lengtha(P)) + 1) , OI) >
. < BeginS2Pr ; (Pr , Ch) >   if CT < lengtha(P) .
```

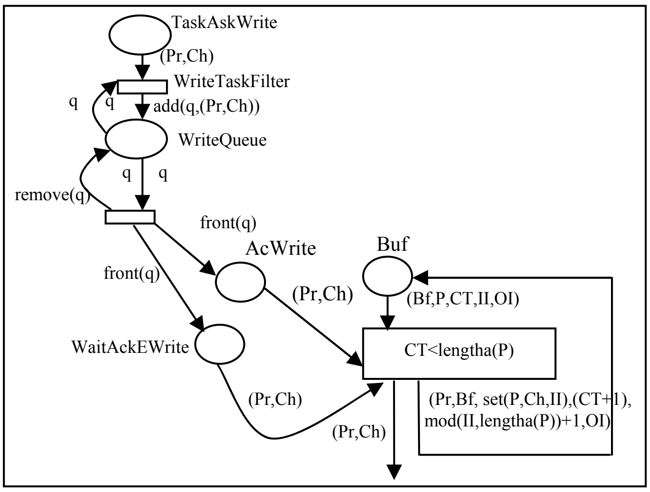


Fig. 5 Compact representation of entry Write of Buffer type after applying refinement rules

B. Application of APNs' Reduction Rules Adapted to ECATNets on the Example

After applying reduction rules proposed in this paper on the previous example, we note that the ECATNet given in Fig. 5 may be reduced again. We can apply reduction rule 'parallel places' adapted to ECATNets. Informally, two places are said parallel if they are linked in the same manner to all net's transitions. We can remove one with the smallest initial marking. We note that WaitAckEWrite and AcWrite are effectively parallel places. These two places are empty in initial marking. We can eliminate one of them. We delete the place WaitAckEWrite. We will see how this double reduction decreases the steps of some analysis phases.

VI. Ada-ECATNet Analysis

Before explaining how our proposed reduction rules have brought efficiency in the analysis of Ada-ECATNet, we present first analysis method of Maude system: reachability analysis and Model Checking.

A. Reachability Analysis

By using the command 'search' of Maude system, we can know if a certain state is accessible or not from initial state. In general, we know the final state which system must reach from a certain initial state. We want to know if a system reaches this final state or not. In non-deterministic systems, a simulation can not show if the system arrives to a specific final state because of the non-determination of the behaviour that we can not force it to follow a specific way. To know if there is a possible way in the system execution allowing it to reach this final state, we call the command 'search'. In our example, we want be sure that the previous state is reached by the initial state initila6. The following state allows launching the searching of all accessible state starting from initial marking until final marking. In the absence of a solution, Maude system displays 'no solution':

```
search in BUFFER : < InitialCharList ; 'r 'v 'b 'x 't 'y > . < BeginPr ; (Pr ,
AnyThing) > . < WriteQueue ; newq > . < BeginCs ; (Cs , AnyThing) > . <
ReadQueue ; newq > . < CharListResult ; empty > . < Buf ; (BF , newa , 0 , 1 ,
1) >  =>*  < EndPr ; Pr,endoflist > . < InitialCharList ; empty > . < EndCs  ;
Cs,endoflist > . < CharListResult ; 'y 't 'x 'b 'v 'r > . < ReadQueue ;  newq > . <
WriteQueue ; newq > . < Buf ; BF,store(store(store(store(store(newa, 't, 5), 'x,
4), 'b, 3),  endoflist, 2), 'y, 1),0,3,3 > .
```

This command allows obtaining accessibility graph of the ECATNet for the previous initial state. For that, we must precise a general final state which in the case of ECATNet is M:Marking. In this case, 'search' returns any accessible state from initial state because any state of an ECATNet is an instantiation of the state M:Marking.

```
search in BUFFER : < InitialCharList ; 'r 'v 'b 'x 't 'y > . < BeginPr ; (Pr ,
AnyThing) > . < WriteQueue ; newq > . < BeginCs ; (Cs , AnyThing) > . <
ReadQueue ; newq > . < CharListResult ; empty > . < Buf ; (BF , newa , 0 , 1 ,
1) >  =>* M:Marking .
```

To have all the accessibility graph, we have to write after this request, the following formula : show search graph .

Note. Accessibility analysis and Model Checking in Maude do not work with infinite-states system. But, in [1] authors show that after translating Ada programs to Petri nets, the obtained Ada-nets are finite-states. Consequently, we can apply these two techniques of Maude to analyze Ada-ECATNet.

B. Model Checking

In this section, we show the applicability of Maude Model Checker in the verification of an example of property about concurrency of Ada-ECATNet. When a task accesses to an entry of a protected type, another task can not access to any entry of this protected type. The task Pr (resp. Cs) is in the entry Write (resp. Read) if it is in one of possible places of this entry BeginWrite, BeginS2Write, BeginS3Write and EndWrite. First, we define some propositions like Pr-In-BeginWrite(Pr). This proposition is valid if Pr is in the place BeginWrite. For the producer Pr and consumer Cs and the initial state 'initial6', the valuation of this property is true :

```
red in BUFFER-CHECK : modelCheck(initial6, <> ((Pr-In-BeginWrite(Pr) ∨
Pr-In-BeginS2Write(Pr)  ∨ Pr-In-BeginS3Write(Pr) ∨  Pr-In-EndWrite(Pr) )
=>~(Cs-In-BeginRead(Cs)∨Cs-In-BeginS2Read(Cs)∨Cs-In-BeginS3Read(Cs)
∨ Cs-In-EndRead(Cs)))/\<>(Cs-In-BeginRead(Cs)  ∨ Cs-In-BeginS2Read(Cs)
∨ Cs-In-BeginS3Read(Cs) ∨ Cs-In-EndRead(Cs) =>~((Pr-In-BeginWrite(Pr) ∨
Pr-In-BeginS2Write(Pr)∨ Pr-In-BeginS3Write(Pr)∨ Pr-In-EndWrite(Pr))))) .
```

VII. Performance Valuation

To show how proposed rules have reduced in efficient way the size of Ada-ECATNet, we have applied simulation, Model Checking and reachability analysis under Maude system. In the sequel, we consider that:

Case1: Ada-ECATNet before applying any reduction rules.

Case2: Ada-ECATNet after applying reduction rules proposed in this paper.

Case3: Ada-ECATNet after applying reduction rules proposed in this paper and those proposed in [5], [6].

Let's note that Diff. in the following three tables is between Case1 and Case3.

Simulation. For an input InitialCharList containing every time from 6 to 10 characters, we have made a simulation for the three cases. For n (n = 6,..,10) the number of characters, we have calculated the number of rewriting steps required to get the final marking from the same initial marking for the three cases. We notice that the number of rewriting steps in Case1 is always more elevated than the one in Case2. Moreover, the gap between the two numbers rewriting steps in Case1 and Case2 increases every time the number of the characters InitialCharList increases. The difference between rewriting steps' numbers in Case2 and Case3 is small. The result of the simulation is presented in the comparative Table I.

TABLE I
SIMULATION: COMPARAISON OF RESULTS

	6 C.	7 C.	8 C.	9 C.	10 C.
Case1	1098	1298	1531	1797	1966
Case2	995	1180	1398	1649	1803
Case3	994	1179	1397	1648	1802
Diff.	**104**	**119**	**134**	**149**	**164**

Accessibility Analysis: We obtained interesting result when we have applied the reachability analysis tool of Maude to calculate the number of rewriting steps needed to construct reachability graph in the three cases. The reduction proposed in this paper has a great impact in decreasing the rewriting steps' number needed to construct the reachability graph. The application of 'parallel places' reduction rule decreases again this number. For instance when InitialCharList contains 10 characters, the construction of the reachability graph needed in Case2 is approximatively 44% less than the rewriting steps in the Case1. This difference is meaningful. The gap between Case3 and Case2 is small. As described above, the benefit of the reduction rules 'parallel places' consists to decrease steps in creating accessibility graph. Such benefit is realized in our situation between Case2 and Case3. The table III describes the evolution of rewriting steps number required in Case1, Case2 and Case3 to construct accessibility graph.

TABLE III
ACCESSIBILITY ANALYSIS: COMPARAISON OF RESULTS

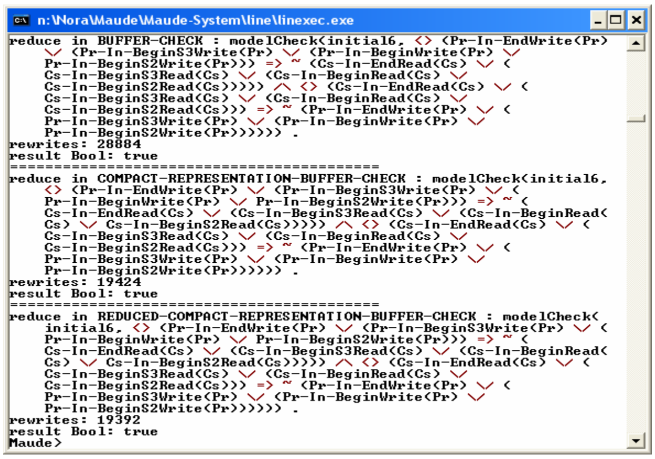
	6 C.	7 C.	8 C.	9 C.	10 C.
Case1	28622	36047	44119	52922	60747
Case2	17412	21517	25947	30702	34157
Case3	17380	21485	25915	30670	34125
Diff.	**11242**	**14562**	**18204**	**22252**	**26622**

Model Checking: We obtain the same result when we use Model Checker of Maude. For the input InitialCharList containing to every time from 6 to 10 characters, the table II resumes the evolution of rewriting steps' number required in Case1, Case2 and Case3 to check the correction of the property defined above.

TABLE II
MODEL CHECKING : COMPARAISON OF RESULTS

	6 C.	7 C.	8 C.	9 C.	10 C.
Case1	28884	36890	45089	53965	61890
Case2	19424	24425	29959	36030	41010
Case3	19392	24393	29927	35998	40978
Diff.	**9492**	**12497**	**15162**	**17967**	**20912**

A part of the verification of the property using Maude Model Checker is presented in Fig. 6. The reduction rules proposed in this paper decreases the number of rewriting steps from 28884 in Case1 to 19424 in Case2. The application of 'parallel place' reduction rule of APNs decreases it again to 19392 in Case3.



```
reduce in BUFFER-CHECK : modelCheck(initial6, <> (Pr-In-EndWrite(Pr)
    \/ (Pr-In-BeginS3Write(Pr) \/ (Pr-In-BeginWrite(Pr) \/
    Pr-In-BeginS2Write(Pr))) => ~ (Cs-In-EndRead(Cs) \/ (
    Cs-In-BeginS3Read(Cs) \/ (Cs-In-BeginRead(Cs) \/
    Cs-In-BeginS2Read(Cs))))) /\ <> (Cs-In-EndRead(Cs) \/ (
    Cs-In-BeginS3Read(Cs) \/ (Cs-In-BeginRead(Cs) \/
    Cs-In-BeginS2Read(Cs))) => ~ (Pr-In-EndWrite(Pr) \/ (
    Pr-In-BeginS3Write(Pr) \/ (Pr-In-BeginWrite(Pr) \/
    Pr-In-BeginS2Write(Pr)))))) .
rewrites: 28884
result Bool: true
========================================
reduce in COMPACT-REPRESENTATION-BUFFER-CHECK : modelCheck(initial6,
    <> (Pr-In-EndWrite(Pr) \/ (Pr-In-BeginS3Write(Pr) \/ (
    Pr-In-BeginWrite(Pr) \/ Pr-In-BeginS2Write(Pr))) => ~ (
    Cs-In-EndRead(Cs) \/ (Cs-In-BeginS3Read(Cs) \/ (Cs-In-BeginRead(
    Cs) \/ Cs-In-BeginS2Read(Cs))))) /\ <> (Cs-In-EndRead(Cs) \/ (
    Cs-In-BeginS3Read(Cs) \/ (Cs-In-BeginRead(Cs) \/
    Cs-In-BeginS2Read(Cs))) => ~ (Pr-In-EndWrite(Pr) \/ (
    Pr-In-BeginS3Write(Pr) \/ (Pr-In-BeginWrite(Pr) \/
    Pr-In-BeginS2Write(Pr)))))) .
rewrites: 19424
result Bool: true
========================================
reduce in REDUCED-COMPACT-REPRESENTATION-BUFFER-CHECK : modelCheck(
    initial6, <> (Pr-In-EndWrite(Pr) \/ (Pr-In-BeginS3Write(Pr) \/ (
    Pr-In-BeginWrite(Pr) \/ Pr-In-BeginS2Write(Pr))) => ~ (
    Cs-In-EndRead(Cs) \/ (Cs-In-BeginS3Read(Cs) \/ (Cs-In-BeginRead(
    Cs) \/ Cs-In-BeginS2Read(Cs))))) /\ <> (Cs-In-EndRead(Cs) \/ (
    Cs-In-BeginS3Read(Cs) \/ (Cs-In-BeginRead(Cs) \/
    Cs-In-BeginS2Read(Cs))) => ~ (Pr-In-EndWrite(Pr) \/ (
    Pr-In-BeginS3Write(Pr) \/ (Pr-In-BeginWrite(Pr) \/
    Pr-In-BeginS2Write(Pr)))))) .
rewrites: 19392
result Bool: true
Maude>
```

Fig. 6 Property verification using Maude Model Checking before and after applying reduction rules

VIII. CONCLUSION

ECATNets offer a compact representation of the Ada programs. In this paper, we proposed how to get a more compact Ada-ECATNet representation of Ada programs during the translation step. The reduction consists to compact these ECATNets by integrating several transitions (that represent some Ada sequential statements) in only one transition. Such operation reduces the size of the ECATNets, and minimizes the number of rewriting rules considerably in the equivalent Maude program. Therefore, the verification of properties of this program (Ada-ECATNet after reduction) takes less time than the initial program (Ada-ECATNet before reduction). The reduced Ada-ECATNet will have a small size in terms of transitions, places and arcs and so a small states number with regard to their once before reduction. Consequently the application of any verification tool becomes more efficient. We have experimented simulator, reachability analyzer and Model Checker to show what we gain. The obtained reduced Ada–ECATNet may be reduced again by applying reduction rules proposed for APNs. In this paper, we show through an example how it is possible to apply a reduction rule of APNs on Ada–ECATNet after applying refinement rule proposed in this paper. This double reduction decreases in efficient way the running times and the memory consumptions of some analysis methods as simulation, accessibility analysis and Model Checking.

REFERENCES

[1] K. Barkaoui and J-F Pradat-Peyre. "Verification in Concurrent Programming with Petri nets Structural Techniques". In Proceedings Third IEEE International High-Assurance Systems Engineering Symposium November 13 - 14, 1998 Washi, 1998.

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

[3] M. Bettaz, A. Chaoui, K. Barkkaoui. “On Finding Structural Deadlocks in ECATNets Using a Logic of Concurrency”. Journal on Computing and Information, Vol 2 No 1, pp. 495-506, 1996.

[4] N. Boudiaf, A. Chaoui, “Towards Automated Analysis of Ada-95 Tasking Behavior By Using ECATNets”. ISIIT’04, Jordan, 2004.

[5] N. Boudiaf, K. Barkaoui, Allaoua Chaoui. “Implémentation Des Règles de Réduction des ECATNets dans Maude”. Proceedings de la Conférence Mosim’06, 2-5 avril, Rabat, Maroc, pp. 1505-514, 2006.

[6] N. Boudiaf, K. Barkaoui and Allaoua Chaoui. "Applying Reduction Rules to ECATNets". Proceedings of AVIS'06 Workshop (Co-located with the conferences ETAPS'06), 1st April, Vienna, Austria, To appear in ENTCS, 2006.

[7] E. Bruneton and J-F. Pradat-Peyre. “Automatic Verification of Concurrent Ada Programs”, In Proceedings Reliable Software Technologies-Ada-Europe'99, 1999.

[8] M. Clavel and al., “The Maude 2.0 System”. In Proc. Rewriting Techniques and Applications, V. 2706 of LNCS, Spring-Verlag, 2003.

[9] S. Evangelista, C. Kaiser, J. F. Pradat-Peyre, and P. Rousseau. “Quasar: a new tool for analyzing concurrent programs”. In Ada-Europe 2003, LNCS. Springer-Verlag, 2003.

[10] Ravi K. Gedela, Sol M. Shatz and Haiping Xu. "Compositional Petri Net Models of Advanced Tasking in Ada-95". Comput. Lang. 25(2): 55-87, 1999.

[11] ISO/IEC 8652. “Information Technology – Programming Languages – Ada”, 1995.

[12] 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, p. 331-372, 1996.

[13] 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. Kluwer, 2000.

[14] T. Murata, B. Shenker, S. M. Shatz. ”Detection of Ada Static Deadlocks Using Petri Nets Invariants”. IEEE trans. Oo Software Engineering, vol. 15, No. 3, pp 314-326, 1989.

[15] S. M. Shatz, S. Tu, T. Murata, S. Duri. “An Application of Petri Net Reduction for Ada Tasking Deadlock Analysis”. IEEE Transactions on Parallel and Distributed Systems,1996.

[16] K. Schmidt. “Applying Reduction Rules to Algebraic Petri Nets”. TKK Monoistamo; Otaniemi 1997, ISSN 0783 5396, ISBN 951-22-3495-5, 1997.