Automatic Translation of Ada-ECATNet Using Rewriting Logic

N. Boudiaf

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 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 and its programming language Maude. Rewriting logic is considered as one of very powerful logics in terms of description, verification and programming of concurrent systems We proposed previously a method for translating Ada-95 tasking programs to ECATNets formalism (Ada-ECATNet) and we showed 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. We showed also previously how the ECATNet formalism offers to Ada many validation and verification tools like simulation, Model Checking, accessibility analysis and static analysis. In this paper, we describe the implementation of our translation of the Ada programs into ECATNets.

Keywords—Ada tasking, Analysis, Automatic Translation, ECATNets, Maude, Rewriting Logic.

I. INTRODUCTION

ONE of the most attractive features of the Ada programming language is the tasking, which allows 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 [1] 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-

N. Boudiaf is with the University of Oum El Bouaghi, Algeria (e-mail: boudiafn@gmail.com).

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 [2], reduction rules [4], [5], 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.

Previously, we showed in [6] how ECATNets formalism presents a very compact representation for Ada program. In [6], we present some refinement rules which allow reduction during translation step. Such translation minimizes effectively the number of program states. 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 reduction rules defined by Schmidt [16] are adapted and implemented to ECATNets in [4], [5]. This double reduction allows a meaningful decrease of the complexity of state-space analysis. But, in the works based on simple Petri nets or CPNs like, Quasar tool developed in [9], authors translate Ada programs first to ordinary Petri nets or CPNs and they reduce obtained Ada-nets after. So, only one reduction is possible for Ada-nets.

In this paper, we describe the implementation of our translation of the Ada programs to ECATNets. Considering the complexity of Ada, this implementation touches only a subset of the concepts of this language. Our efforts are concentrated on the concepts relating to concurrency and the definition of task. Our objective is to show the feasibility of an automatic translation based-Maude of an Ada program towards ECATNet. We show in this work the validation of the translation of some basic concepts of concurrency as the rendez-vous, etc. The Ada-ECATNet translator is based on the integration of the three traditional phases of a compiler: lexical analysis, syntactic analysis and generation of ECATNet code. The language Maude is used to implement such translator.

To simplify the use of our Ada-ECATNet translator, we developed also a small application as an interface between

user and Maude system for a better execution of this translator. The user can use this application to create or open his Ada program; he can ask the application for lexical analysis or ECATNet code generation by only clicking on the appropriate command. In this paper we showed how this tool woks.

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 in informal way with the help of an example. In section 4, the main phases of our Ada-ECATNet translator are described. Technical aspect of the application is presented in section 5. In section 6, we showed how we apply the translator on a simple example of Ada program to get ECATNet code. Finally, we conclude the paper in the section 7.

II. ECATNETS

ECATNets [1] 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 : $s = (p, a \oplus b \oplus c)$.
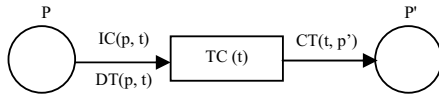

Fig. 1 A generic ECATNet

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, the common elements between DT(p, t) and M(p) are removed. 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. Now the forms of the rewrite rules (i.e., the meta-rules) to associate with the transitions of a given ECATNet are recalled.

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), the elements which are common to these two multi-sets have to be removed. 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:
$t : (p, [DT(p, t)]_\oplus \cap [M(p)]_\oplus) \rightarrow (p', [CT(t, p')]_\oplus)$
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. SOME GUIDELINES OF TRANSLATION FROM ADA TO ECATNET THROUGH AN EXAMPLE

Most concepts of Ada translation to ECATNets are defined in [3]. For lack of space reason, here we give 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. A sort 'Producer' to represent task type producer is defined. In this case, a producer task is an algebraic term constant 'Pr' of sort 'Producer'. A n-tuple algebraic term composed of algebraic terms that represent 'task' and its 'local variables' is used. The translation of entry Write gives us the ECATNet of the 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. There is a condition isempty(q) == false for the transition TaskSelectWrite. For the translation of a protected type, a place is created to contain a 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 the 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, Pr and Ch are integrated 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.
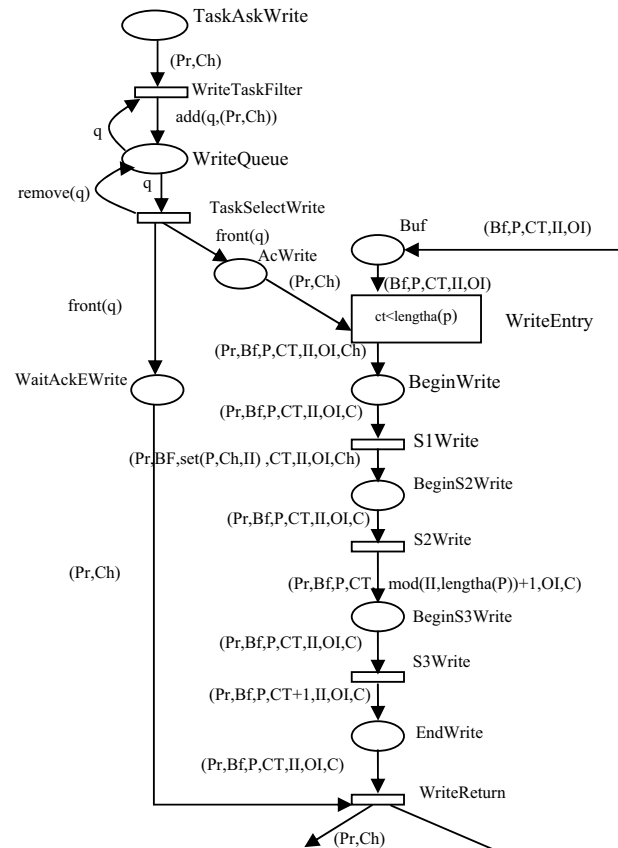

Fig. 2 Representation of entry Write of Buffer type by ECATNets

C. Mapping the Obtained Ada-ECATNet to Maude

Among kinds of modules defined in Maude, there are 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. The operation "<_;_>" is defined to permit 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. An operation to implement the operation ⊕ is not defined. 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), can be written as (p, a) ⊗ (p, b) ⊗ (p, c). Now, a part of module implementing the ECATNet buffer is presented: 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. Data types like Queue of this ECATNet are described in a hierarchy of functional modules when, Queue is declared 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
```

The application of rules defined in [6] on entry Write of Buffer type gives a compact representation in figure 3. In Maude program, the rules WriteTaskFilter and WriteTaskSelect are kept without any change. But, the remaining five transitions are merged 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) .

IV. ADA-ECATNET TRANSLATOR

The Ada-ECATNet translator is developed in the same way as any other compiler. Classical known phases are proposed : a lexical analysis, a syntactic analysis and a phase of code generation. Let us note that if the Ada program does not contain errors, our application returns the ECATNet equivalent code, in the presence of mistakes in Ada program, our application returns the constant 'ErrorUple'. Thereafter, the realization's details of this translator's phases are explained. Figure 4, describes a view on the different steps of the Ada-ECATNet translator.
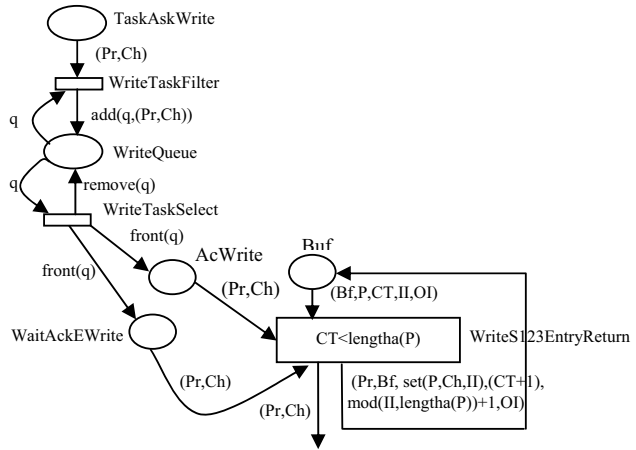

Fig. 3 Compact representation of entry Write of Buffer type after applying refinement rules
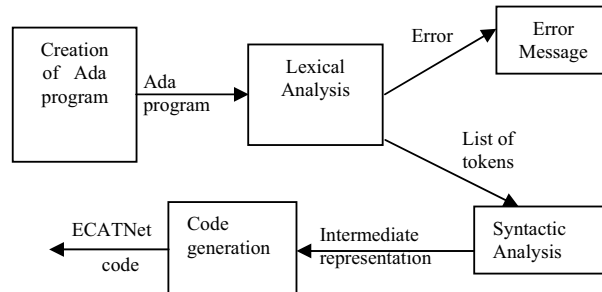

Fig. 4 Methodical view on Ada-ECATNet translator

A. Lexical analysis

This phase takes Ada program as input and generates all the lexemes constituting this program. This phase transforms the Ada program to a list of its lexemes in form of list of strings.

B. Syntactic analysis

This step transforms the input list of Ada program lexemes into an intermediate representation to facilitate us the generation of the ECATNet code. For the development of this phase, certain types of specific data are created. This phase takes as input a data type 'Uple' composed of a list 'List' and a stack 'Stack'. The list contains the strings indicating the lexemes of the program Ada (operator, identifier, keyword, etc).

Data types. The stack contains information on the Ada program necessary for the generation of equivalent ECATNet code. Each time the parser evolves in the list, it collects information on the Ada program and put it in the stack. The basic element of the stack is a data type called 'Code'. This last contains also strings. The following operation allows the construction of an 'Uple':

op (_;_) : List Stack -> Uple .

If the list to be analyzed contains a mistaken string (the Ada program is erroneous), the analyzer returns 'ErrorUple':

op ErrorUple : -> Uple .

The operation 1st extracts the first parameter from (_;_), which is a list and the function 2nd extracts the second parameter which is of sort 'Stack'. Now, the operations for the construction of a code (element of sort 'Code') and the construction of an empty code:

op _,_ : Code Code -> Code [assoc id : nullCode] .
op nullCode : -> Code .

Two constants BeginCode and EndCode are defined of the type 'Code'. These two constants are used to delimit in a stack, the necessary information concerning a specific syntactic construction of a Ada program. The type 'Tuple' is defined to support the token consisted of the identifier of the task and its variables. The operation _, _ allows building extensible tuple:

sorts Elt Tuple . subsort Elt < Tuple . op _,_ : Elt Tuple -> Tuple .

Also another sort 'ExtTuple' is defined which to be composed of several 'Tuple'. This sort is useful in the cases of passage of the parameters at the time of the call of an entry or of procedure. According to our Ada-ECATNet translation, when the task calls an entry, the tuple representing the state of the task enters the place where it is the queue of the entry. In this case, there is no mechanism to distinguish the current parameters used in the call. Then, in order to not lose of such information at the time of the call, the tuple is rewritten in a term of kind 'ExTuple'. This term is composed of two tuples: the one which represents the state of the task and the other contains the current parameters to pass later at the time of the concretization of the rendez-vous between the two tasks.

Syntactic Analysis Functions. Now, we explain some functions of syntactic analysis. It is about the functions concerning the analysis of the statements. Initially, the production rule of the assignment statement is:

assignment_statement ::=   variable_name **:=** expression;

Intuitively, after the analysis of this statement, the name of the variable and the contents of the expression must be saved. That is to say assignment-statement-Analysis (L; (Id, Ex)) is the function which analyzes the assignment statement and save the elements figuring in the assignment statement. Id is the left part of the assignment and Ex is the right part of the assignment:

op assignment-statement-Analysis : Uple -> Uple .
eq assignment-statement-Analysis(L ; (Id, Ex)) =
if IsIdentifier(head(L)) == true
   and head(tail(L)) == ":="
   and expression-Analysis(tail(tail(L)) ; Ex) =/= ErrorUple
   and head(1st(expression-Analysis(tail(tail(L)) ; Ex))) == ";"
then tail(1st(expression-Analysis(tail(tail(L)) ; Ex))) ;
      (head(L), 2nd(expression-Analysis(tail(tail(L)) ; Ex)))
else ErrorUple fi .

In this code, four conditions are defined:
- IsIdentifier(head(L)) == true : this condition is true if head (L) is an identifier.
- head(tail(L)) == ":=" : this condition is true if the element which is next this identifier is equal to ": =".

- expression-Analysis(tail(tail(L)) ; Ex) : this condition is correct if the elements which are in the list next the identifier and ": =" constitutes a correct expression.
- head(1st(expression-Analysis(tail(tail(L)) ; Ex))) == ";" : this last condition is true if the element which is in the list next the elements constituting the expression is equal to "; ".

If the four conditions are valid, the function assignment-statement-Analysis (L; (Id, Ex)) returns an 'Uple' tail (1st (expression-Analysis (tail (tail (L)) ; Ex))) ; (head (L), 2nd (expression-Analysis (tail (tail (L)) ; Ex))). This last is consisted of the remainder of the list to analyze: tail (1st (expression-Analysis (tail (tail (L)) ; Ex))) after eliminating the elements described above concerning the assignment, and a code (head (L), 2nd (expression-Analysis (tail (tail (L)) ; Ex))) containing the necessary information for the generation of the ECATNet code for this assignment later. This code contains the right part of the assignment head(L) and its left part 2nd (expression-Analysis (tail (tail (L)) ; Ex))). This part, itself is returned by the function expression-Analysis (tail (tail (L)) ; Ex)) who is responsible for the analysis of the expressions.

The function assignment-statement-Analysis (L; (Id, Ex)) is called during the analysis by the simple-statement-Analysis function (L; (Kind, Id, IdL, Ex)) which analyzes the Ada code generated by simple_statement. Four parameters for the code are needed: Kind, Id, IdL and Ex. Initially, the production rules of simple_statement are:

simple_statement ::= null_statement | assignment_statement |
                     exit_statement | return_statement
                     | entry_call_statement | abort_statement

Let's give in detail also entry_call_statement:
entry_call_statement ::= entry_name [actual_parameter_part]**;**
That is to say entry-call-statement-Analysis (L; (Id, IdL)) the function which analyzes this instruction. It returns a code composed of name of the entry Id and a list of the actual parameters IdL. The last parameter Kind is used to save the type of the instruction, the following code is a part of the simple-statement-Analysis function (L; (Kind, Id, IdL, Ex)), (Ex is condition returned by exit-statement-Analysis) :

op simple-statement-Analysis : Uple -> Uple .
eq simple-statement-Analysis(L ; (Kind, Id, IdL, Ex)) =
if IsIdentifier(head(L)) == true
then if assignment-statement-Analysis(L ; (Id, Ex))
        =/= ErrorUple
     then 1st(assignment-statement-Analysis(L ; (Id, Ex))) ;
         ("assg", 1stC(2nd(assignment-statement-Analysis(L ;
          (Id, Ex)))),
          empty, 2ndC(2nd(assignment-statement-Analysis(L ;
          (Id, Ex)))))
   else …
**** Analysis of the other types of statements
fi .

In this code, if head (L) is an identifier, then we test if assignment-statement-Analysis (L; (Id, Ex)) =/= ErrorUple is valid. If this condition is true, so it is about an assignment

statement in the code. The result returned in this case by the function is composed by the remainder of the list of entry after skipping the elements of the assignment: 1st (assignment-statement-Analysis (L; (Id, Ex))). The code is consisted of four elements: such as the first element "assg" indicate the kind of the statement which is the assignment in this case, the second element is the left part of the assignment, the third one is empty (it is independent of the assignment, but it is not empty for other instructions). The fourth element is the right part of the assignment.

C. Code Generation

This phase takes the previous intermediate representation and generates the ECATNet code equivalent to the Ada program. The generated code is in the form of a hierarchy of the functional modules and system modules. Functional modules implement data types. System module implements the concurrent behavior of the tasks and their communications. The system module imports the last functional module in the hierarchy of the functional modules. A part of code which is used to generate the ECATNet code relating to the assignment statement is explained. In our file concerning the generation of the code, the following variables are declared:

vars Id RuleOrder Place1 : String . vars L : List . var SK : Stack .

The following function Create-RulesFor-assg-Stmts (L, SK, Id, RuleOrder, Place1) generates the ECATNet code of an assignment statement. Such as L is the list containing the identifier and the local variables of the task, SK is the stack containing the part of code relating to the assignment for which, a code will be generated. Id is the name of the unit (task, package, entry,…), RuleOrder is the order of next rewriting rule to be generated. Place1 saves the order of the next place to be generated. Create-RulesFor-assg-Stmts (L, SK, Id, RuleOrder, Place1) starts by removing BeginCode and EndCode by calling StackSubstraction (pop (SK), EndCode). Then, it calls Create-RulesFor-assg-Stmts-1 (L, StackSubstraction (pop (SK), EndCode), Id, RuleOrder, Place1). StackSubstraction (pop (SK), EndCode) allows returning in this case a code:

op Create-RulesFor-assg-Stmts :
    List Stack String String String -> String .

eq Create-RulesFor-assg-Stmts(L, SK, Id, RuleOrder, Place1) = Create-RulesFor-assg-Stmts-1(L, StackSubstraction(pop(SK), EndCode), Id, RuleOrder, Place1) .

This code returned is Cd which is composed of four parameters saving the parts of the assignment. The function Create-Tuple (L) transforms this list with a tuple. If L = a1. a2… .an, then the tuple obtained is form (a1, a2,… ,an). The function ReplaceEltinList (2ndC (Cd), 4rthC (Cd), L) allows replacing the occurrence of the variable 2ndC (Cd) in the list L by the expression 4rthC (Cd) which is the third part of the assignment. The variables of the task are stored in a list L saved in the code:

op Create-RulesFor-assg-Stmts-1 :
    List Stack String String String -> String .

eq Create-RulesFor-assg-Stmts-1(L, Cd, Id, RuleOrder, Place1) =
" rl [" + NewRule(RuleOrder, Id) + "] :
< " + NewPlace(Place1, Id) + " ; " + Create-Tuple(L) + " > "
+ " => " + " < " + NewPlace(SuccNumber(Place1), Id) + " ; "
+ Create-Tuple(ReplaceEltinList(2ndC(Cd), 4rthC(Cd), L))
+ " > . " .

Now, we explain another function Create-RulesFor-entry-call-Stmt(L, SK, Id, RuleOrder, Place1) which is used to generate the ECATNet code of an entry call statement. Such as SK is the stack containing the part of code relating to the call of the entry for which, a code will be generated. L, Id, RuleOrder and Place1 have the same significance as in the function Create-RulesFor-assg-Stmts(L, SK, Id, RuleOrder, Place1). In the code created by Create-RulesFor-entry-call-Stmt(L, SK, Id, RuleOrder, Place1), CreateTaskAsk-Place (entry-call-name(SK)) creates a name of place containing the name of the called entry. In fact, the function entry-call-name(SK) returns the name of the called entry. The first built rewriting rule, allows the token of the task which is Create-Tuple (L) to put itself in the place of the entry entry-call-name (SK). This place which is called CreateTaskA sk-Place (entry-call-name(SK)) is also in the code of the suitable entry and is used to receive the tasks which call this entry. The function entry-call-list(SK) returns the current parameters passed to the called entry:

op Create-RulesFor-entry-call-Stmt :
List Stack String String String -> String .
eq Create-RulesFor-entry-call-Stmt(L, SK, Id, RuleOrder, Place1) =
" rl [" + NewRule(RuleOrder, Id) + "] : "
+ " < " + NewPlace(Place1, Id) + " ; " + Create-Tuple(L)
+ " > "
+ " => " + " < " + CreateTaskAsk-Place(entry-call-name(SK)) + " ; "
+ "(" + Create-Tuple(L) + " ; "
+ Create-Tuple(entry-call-list(SK)) + ") > . "
+ " rl [" + NewRule(SuccNumber(RuleOrder), Id) + "] : "
+ " < " + CreateReturnFromCall-Place(entry-call-name(SK))
+ " ; " + "(" + Create-Tuple(L) + " ; "
+ Create-Tuple(entry-call-list(SK)) + ") > "
+ " => " + " < " + NewPlace(SuccNumber(Place1), Id)
+ " ; " + Create-Tuple(L) + " > . " .

The second rewriting rule built by this function represents the return of the entry call. In the code of the entry, a place CreateReturnFromCall-Place (entry-call-name (SK)) is created. This rewriting rule allows to the task to leave this place towards a new place to follow its activity. The function entry-call-name(SK) returns the name of the called entry. The function entry-call-list(SK) gets back the current parameters to the called entry.

V. TECHNICAL ASPECT

Let's note that Maude system has only textual version. So, instead to use directly Maude system to execute the translator, we developed a small application as interface between the user and Maude system. This application is developed with Delphi language to help user to use easily and in better way our translator. The application provides to the user classical edition of text files to create his Ada program with 'File' and 'Edit' options. 'Help' option contains some explanations

about this tool. Moreover, two options are proposed: 'Lexical Analysis' and 'Code Generation' in the main menu. 'Lexical analysis' contains two options: 'List of Tokens Creation' and 'List of Tokens in Maude'. The first option allows the calling of the lexical analyser and displaying all tokens of the Ada program. The second one allows the transforming of this list to a list of strings and creating a module containing this list as constant to be treated by the code generator later. 'Code Generation' option contains two options too. The first one is 'ECATNet Code generation' which allows the calling of the translator written in Maude and generating a system module equivalent to the Ada program. The second option which is 'ECATNet Complete Code' allows the extraction of the initial marking equivalent to the initial state of the Ada program. So, this option generates the ECATNet code and adds a rewriting command with initial marking as parameter. The obtained code in this case is ready to be simulated under Maude system. In figure 5, we present a part of Ada-ECATNet based-Maude translator.
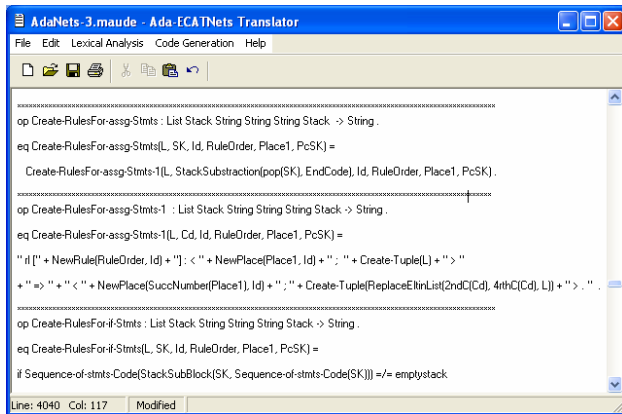
Fig. 5. Part of Ada-ECATNet based-Maude translator

A. Ada-ECATNet Translator Files

Let's go back to the Ada-ECATNet translator; we note that a hierarchy of functional modules to implement in Maude this translator are developed. For the simplicity reason, these modules are distributed on some files. This file LexAdaNets.maude contains lexical analyser. The file AdaNetsData-1.maude contains a functional module which defines some data types to be imported by the generated code, for instance, arrays. The file AdaNets.maude includes all data types used by the application. Finally, files AdaNets-1.maude, AdaNets-2.maude and AdaNets-3.maude contain functions to generate ECATNet code to an equivalent Ada program. The first file contains modules to generate ECATNet code for Ada basic statements, the file AdaNets-2.maude contains modules to generate ECATNet code for statements related to task concept and AdaNets-3.maude contains modules to generate ECATNet code for general code (package).

VI. EXAMPLE

In this section, we show two things, the first one how the phases of our translator act on the previous example and the second one the execution of the example under our application. Let's note that some syntactic modifications are done on the example. For instance, a package BufferPackage to contain tasks and protected type Buffer is created. Consider an Ada code composed of only one instruction:
Buffer.Write (Char);

The lexical analyzer transforms this code to an 'Uple' composed of a list of strings and an empty stack:
"Write" . "(" . "Char" . ")" . ";" ; emptystack

The syntactic analysis transforms this call of an entry with a representation in the form of stack containing three elements BeginCode, ("entry call", "Write", "Char") and EndCode. The two constants BeginCode and EndCode are used to delimit the code of the entry call. The code in the middle contains three strings: "entry call", "Write" and "Char".

In fact, the first string allows indicating that this code concerns the entry call. The second string contains the name of the entry called and the third string can be a list of the parameters of the called entry. The result obtained after the analysis of this instruction is the 'Uple' composed by the list of entry which is empty in our case, and the representation in form of stack of the entry call: empty ; push(BeginCode, push(("entry call", "Write", "Char"), EndCode))

The syntactic analysis generates an intermediate representation in the form of stack (returned by the function which analyzes a sequence of instructions. The latter contains only one instruction in our example:
empty ; push(BeginCode, push("sequence of statements",
      push(BeginCode, push("entry call", "Write", "Char",
         "", EndCode))))

Only a small portion of code generated by our application is presented. The generator of code creates a declaration of variable var Generic-producer: producer. It allows writing only one ECATNet code for all the producers. The rewriting rule producer-Rule-4 translates a part of the call of the Write entry. This call is expressed as putting the tuple (Generic-producer, Char) in the place Task-Ask-Write-Place. Let us note that the places used in this part of code are created before their use like the code expressing the Write entry.

After the execution of this rewriting rule, the token (Generic-producer, Char) is in the Write entry which is not explained here for simplicity. Normally, at the end of the execution of this entry, this token is found in a place named Return-From-Call-Entry-Write-Place. The rewriting rule producer-Rule-5 takes again the token and puts it in a place named producer-Place-4; so, the task can continue its execution:
var Generic-producer : producer .
rl [producer-Rule-4] :
< producer-Place-3 ; (Generic-producer, Char) >
=> < Task-Ask- Write-Place ; ((Generic-producer, Char) ;
    (Char)) > .
rl [producer-Rule-5] : < Return-From-Call-Entry- Write-Place ;
((Generic-producer, Char) ; (Char)) >
=> < producer-Place-4 ; (Generic-producer, Char) > .

After showing how our translator acts on the Ada program

to translate it to an ECATNet code, let's explain how the user can use the application 'Ada-ECATNets Translator' to get an ECATNet code equivalent to an Ada program. After opening the application, the user can create his Ada program or just opened it if it exists. The figure 6 presents a part of the Ada program example opened in our application:

Fig. 6. Part of the Ada program example

After calling the lexical analyser to identify every token in the program in Ada, the application creates a list of strings in Maude. Figure 7 describes a functional module created by our application to contain the Ada program as a list of strings (tokens) in Maude.

Fig .7. Part of the Maude module containing Ada program tokens

If we execute of the command 'ECATNet Code generation' in 'Code Generation' option, then the ECATNet code is obtained after translating the Ada program example. To get this ECATNet code, the application calls Maude system to execute the translator written in Maude and saved in the file AdaNets-3.maude. In the figure 8, there is a part of the ECATNet code of translated Ada program Buffer. This part is about the beginning of some created modules of data and the principal module translating the package containing tasks and

the protected type buffer. In the figure 9, a part of the ECATNet code is given, but this time, the part of the code is about some rewriting rules translating some statements of the Write entry.

Fig. 8. Part of ECATNet code of translated Ada program Buffer

Fig. 9. Part of ECATNet code of translated Ada program Buffer

VII. CONCLUSION

In this paper, we presented our approach in the implementation of Ada-ECATNet translator. We explained the various phases of this translator and we detailed the data types used in these phases. The translator was developed with the help of Maude language. Within the framework of the implementation of this translator, the functional mode of the Maude language is used. Also a small application is developed as an interface between user and Maude system for a better execution of this translator. The user can use this application to create or open his Ada program, he can ask the application for lexical analysis or ECATNet code generation by only clicking on a command.

This automatic translator gives us an environment to Ada programs to be analysed by using ECATNets based on rewriting logic and Maude language tools. It allows us to take advantage of the battery of verification tools of ECATNets formalism to check the correction of Ada programs. First ECATNets offer a very compact representation and double reduction to Ada program; one reduction can be done during the translation step and the other one after the translation of Ada code to ECATNet. On another hand, ECATNet formalism offer many kinds of validation and verification tools like: simulation, Model Checking, accessibility analysis and static analysis.

REFERENCES

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

[2] 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.

[3] N. Boudiaf, A. Chaoui, "Towards Automated Analysis of Ada-95 Tasking Behavior By Using ECATNets", in *Proc. Conference ISIIT'04*, Jordan, 2004.

[4] N. Boudiaf, K. Barkaoui, A. Chaoui, "Implémentation Des Règles de Réduction des ECATNets dans Maude", in *Proc. Conference Mosim'06*, Rabat, Maroc, 2006, pp. 1505-514.

[5] N. Boudiaf, K. Barkaoui and A. Chaoui, "Applying Reduction Rules to ECATNets", in *Proc. AVIS'06 Workshop (Co-located with the conferences ETAPS'06)*, Vienna, Austria, 2006.

[6] N. Boudiaf, A. Chaoui, "Double Reduction of Ada-ECATNet Representation Using Rewriting Logic", *Enformatika Journal (Transactions on Engineering, Computing and Technology)*, Vol. 15, ISSN 1305-5313, pp. 278-284, October 2006.

[7] E. Bruneton and J-F. Pradat-Peyre, "Automatic Verification of Concurrent Ada Programs", in *Proc. Reliable Software Technologies-Ada-Europe*, 1999.

[8] M. Clavel and aL, "Maude Manual (Version 2.2)", Internal report, SRI International, December 2005.

[9] S. Evangelista, C. Kaiser, J. F. Pradat-Peyre, and P. Rousseau, "Quasar: a new tool for analyzing concurrent programs". *in Proc. 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", in *Proc. Comput. Lang. 25(2)*, 1999, pp. 55-87.

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

[12] J. Meseguer, "Rewriting Logic as a Semantic Framework of Concurrency: a Progress Report", in *Proc. Seventh International Conference on Concurrency Theory*, Vol. 1119 of LNCS, Springer Verlag, 1996, pp. 331-372.

[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, in Proc. 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, 1997.