Adapting and Implementing a Reduction Rule to ECATNets by Using Rewriting Logic

N. Boudiaf, and K. Bensaber

Abstract—**ECATNets are a category of algebraic Petri nets. The main feature of ECATNets is, their sound and complete semantics are based on rewriting logic. In this paper, we study the application of reduction rules to ECATNets in order to cope with state explosion problem. We adopted a reduction rule defined for algebraic Petri nets by K. Schmidt in 1997. The integration of ECATNets in the rewriting logic and its language Maude is very promising in terms of implementation of this reduction rule thanks to the concept of the reflectivity of rewriting logic: the self-interpretation of this logic allows us the modeling of an ECATNet and acting on it. We used Maude system to implement and execute such reduction rule adapted for ECATNets. We show how the representation of ECATNets in Maude, helps us to simplify the development by disregarding several details. Maude offers many functions to deal with the meta-level which makes our implementation easy. We show also in this paper how reduction rules help us to get reduced ECATNets smaller than original ones and faster for analyzing some properties.**

Index Terms—**Algebraic Petri Nets, Reduction Rules, ECATNets, Rewriting Logic, Maude, Verification.**

I. Introduction

ECATNets are a category of algebraic Petri nets (APNs) based on a safe combination of algebraic abstract types and high level Petri nets [1]. The semantic of ECATNets is defined in terms of rewriting logic [7], allowing us to build 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 and its language Maude [8] 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.

Analysis techniques supported by Petri nets in general, and ECATNets in particular, are usually very expensive, and any proposed reduction rule helping to cope with state explosion problem is interesting. In [9], Schmidt has presented some reduction rules for APNs bringing some solutions to cope with this problem. For instance, the application of some reduction rules can decrease the steps needed for creating reachability graph or replacing an unbounded net by a bounded one. Each reduction rule preserves a specific set of APNs properties.

In his paper, Schmidt has presented in total nine reduction rules. Compared to others, some of them have different implementation's needs. In this sense, we have proposed in [2] tools for adapting and implementing three reduction rules defined by Schmidt to ECATNets which are: 'places without input transitions', 'parallel transitions' and 'equivalent places'. These three reduction rules present similar implementation's requirements. These rules are different from those presented in [3]. In [3], we presented our implementation of reduction rules: 'transitions without input places', 'parallel places' and 'loop transition'. As extension of the work presented in [2] and [3], we describe in this paper our adaptation of the reduction rule: 'side condition places' to ECATNets. This reduction rule presents different and more complicated implementation's needs compared to those described in the two previous papers. Moreover, Maude based tool that implements this rule is described in this paper. We will also informally describe the adaptation of the reduction rule 'free choice decisions' to the ECATNet. Through a simple example, we will show how both reduction rules 'side condition places' and 'free choice decisions' help us to get reduced nets smaller than original nets and easier for analyzing some properties.

The integration of ECATNets in the rewriting logic and its language Maude is very promising in terms of implementation of these reduction rules thanks to the concept of the reflectivity of rewriting logic: a module could be an input data (parameter) to another module [4]. The input module is described in the meta-level. This concept of reflectivity allows us to describe an ECATNet as a module in meta-level and this module becomes an input to our tool implementing reduction rules. Moreover, Maude offers many services and functions to deal with the meta-level which makes our implementation easy.

The remainder of this paper is organized as follows: In section 2, we give a brief definition of APNs and an overview about reduction rule 'side condition place' for APNs defined in [9]. Rewriting logic and Maude language is introduced shortly in section 3. The section 4 is a general introduction of ECATNets and their description in rewriting logic. In section 5, we present the meta-level computation concept in Maude. We explain in this section some descent functions of Maude used in the framework of this paper. The adaptation in rewriting logic of the reduction rule to the ECATNets is presented in section 6 and its implementation in Maude is presented in section 7. An example of an ECATNet and its

Manuscript received March 6, 2009; revised January 12, 2012.

N. Boudiaf is with the University of Oum El Bouaghi, Oum El Bouaghi 04000 Algeria (phone: (+213) 032 42 42 12; fax: (+213) 032 42 10 36; e-mail: boudiafn@gmail.com).

K. Bensaber is with Eurofunk Kappacher GmbH Company, Salzburg, Austria (e-mail: kbensaber@yahoo.de).

description in Maude are given in the section 8. In this section, we show also how we apply the reductions rules and the developed tool on the example. In section 9, we explain in details the benefits of using reduction rules. Finally, section 10 concludes the paper.

II. Algebraic Petri Nets

A. Definition (Algebraic Petri Nets)

A tuple AN = [D; P, T, F; ψ, ξ, λ; m_0] is an algebraic Petri net **iff** :

- D = [Σ, E] is a specification with Σ = [S, Ω]:
  - S is q set of sorts;
  - Ω is a set of operations symbols;
  - E is a set of equations;
- [P, T, F] is a net;
- ψ : P → S is s sort assignment;
- ξ assigns a set of Σ variables ξ(t) to each transition t ∈ T;
- λ is the arc inscription such that for f =[p, t] or f = [t, p] in F, λ(f) is a multi–term over $T_{\Omega,\psi(p)}(\xi(t))$;
- m_0 is the initial marking, it assigns a finite multi–term over $T_{\Omega,\psi(p)}$ to every p ∈ P.

B. Reduction Rules for APNs

In this section, we present the reduction rule 'side conditions places'. Because APNs are polymorph, they can be interpreted according to different models of the specification. Schmidt describes the application condition of every rule for term algebra, standard algebra and skeleton algebra. In our implementation, we take into account only standard algebra. For more details about reduction rules, see [9]. In the following, we consider N as the net before reduction rule application and N' the net after reduction. pF and Fp are respectively the set of output transitions and the set of input transitions of a place p.

Reduction Rule: Side Condition Places. If all transitions check only a place without changing its marking, then this place can be removed. After applying this rule interleaved events could become concurrent without changing the interleaving semantics of the net.

Application Condition. There is a place p such that for all transitions t : λ[p, t] ≡I λ[t, p].

Application. For every injective assignment of an occurrence of a term in m_0(p) to λ[t, p] and every unifier σ for the set of pairs of assigned terms, insert a transition t_σ where ξ(t_σ) = var(σ), and for all p' ∈ P \ {p}, λ([p', t_σ]) = λ([p', t])σ and λ([t_σ, p']) = λ([t, p'])σ.
Remove p, pF and all arcs connected to them.

Properties.

1. All removed transitions are dead in N;
2. p is bounded in N;
3. All properties and formulas concerning non removed nodes hold in N **iff** they hold in N';

Benefit. The size of the graph does not change at all, but one component of the marking vector is removed and the enabledness check for transitions becomes faster. Moreover, interleaved transitions become concurrent**.** This makes this rule interesting for partial order verification methods.

III. Rewriting Logic Review

In rewriting logic, each concurrent system is represented by a rewrite theory ℜ = (Σ, E, L, R). Its static structure is described by the signature (Σ, E), whereas its dynamic structure is described by the set of labelled rewrite rules R, which are applied modulo the equation E. An important feature of the rewriting logic is that a rewrite theory ℜ = (Σ, E, L, R) can be viewed as an executable specification of the concurrent system that it formalizes. This section reminds the basic definitions of the rewriting logic.

A labelled rewrite theory ℜ is a 4-tuple ℜ = (Σ, E, L, R) where (Σ, E) is a signature; Σ is the sorts set and operators and E is a set of Σ-equations. The signature (Σ, E) is an equational theory which describes the particular algebraic structure of the states of a system (multi-set, binary tree, etc.) which are distributed according to this same structure. $R \subseteq L \times (T_{\Sigma,E}(X))^2$ is the set of pairs whose first component is a label and the second is a pair of E-equivalence classes of terms, with X = {x_1,…,x_n,…}a countable infinite set of variables. The elements of R are called conditional rewrite rules. They describe the elementary and local transitions in a concurrent system. Each rewrite rule corresponds to an action being able to occur, simultaneously, with other actions. The rewriting will operate on equivalence classes of terms, modulo the set of equations E. For a rewrite rule (r, ([t],[t']), ($[u_1]$,$[v_1]$),….,($[u_k]$,$[v_k]$)) we use the notation r: [t]→[t'] if $[u_1]\rightarrow[v_1] \wedge \ldots \wedge [u_k]\rightarrow[v_k]$, where [t] represents the equivalence class of the term t. A rule r expresses that the equivalence class containing the term t is changed to the equivalence class containing the term t' if the conditional part of the rule, $[u_1]\rightarrow[v_1] \wedge \ldots \wedge [u_k]\rightarrow[v_k]$, is verified.

Given a labeled rewrite theory ℜ, we say that ℜ entails a sequent r: [t]→[t'], or that r: [t]→[t'] is a (concurrent) ℜ-rewrite and write ℜ |- r : [t] → [t'] **iff** [t]→ [t'] is derivable from the rules in ℜ by a finite application of the deduction rules (reflexivity, transitivity, congruence, and replacement) of rewriting logic.

A rewrite theory is a static description of a concurrent system. Its semantics is defined by a mathematical model which describes its behaviour. The model for a given labelled rewriting theory ℜ = (Σ, E, L, R) is a category $\tau_\Re(X)$ whose objects (states) are equivalence classes of terms [t]∈$T_{\Sigma,E}(X)$ and whose morphisms (transitions) are equivalence classes of proof-terms representing proofs in rewriting deduction .

- **Reflexivity.** For every [t] ∈ $T_{\Sigma,E}(X)$: $\dfrac{}{[t]\rightarrow[t]}$
- **Congruence.** For every $f \in \Sigma_n$, n ∈ N : $\dfrac{[t_1]\rightarrow[t_1'] \quad \ldots \quad [t_n]\rightarrow[t_n']}{[f(t_1,\ldots,t_n)]\rightarrow[f(t_1',\ldots,t_n')]}$
- **Replacement.** For every rewriting rule $r:[t(x_1,\ldots,x_n)]\rightarrow[t'(x_1,\ldots,x_n)]$ in R,

$\frac{[w_1] \to [w_1'] \quad ... \quad [w_n] \to [w_n']}{[t(\bar{w}/\bar{x})] \to [t'(\bar{w}'/\bar{x})]}$, such that $t(\bar{w}/\bar{x})$ indicates the simultaneous substitution of w_i for x_i in t.

Transitivity. $\frac{[t_1] \to [t_2] \quad [t_2] \to [t_3]}{[t_1] \to [t_3]}$

A. Maude Language

Maude is a specification and programming language based on rewriting logic. Maude is simple, expressive and efficient. It is rather simple to program with Maude, considering that it belongs to the declarative programming languages. It is possible to describe using Maude different types of applications, from prototyping ones to high concurrent applications. Maude is a competitive language in terms of execution and simulation with imperative programming languages [4]. Three types of modules are, in fact, defined in Maude. The *functional* modules, the *system* modules and the *object-oriented* modules which can, in fact, be reduced to system modules. However, they offer explicitly the advantages of the object paradigm. Because we do not need *object-oriented* modules in this work, we will introduce only functional and system modules.

Functional Modules. The functional modules define data types and related operations, which are based on equations theory. By using equations as simplification rules, each expression called term could be evaluated to its reduced form called canonical representation. All the equal terms by means of equations form an equivalence class. The canonical form represents all the terms of the same equivalence class. The set of all the equivalence classes of the ground (i.e, variable-free) terms constitutes a denotational model for a functional module (initial algebra). Equations in a functional module are oriented. They are used from left to right and the final result of the simplification of an initial term is unique independently of the order in which these equations are applied. In addition to equations, this type of modules supports membership's axioms. These axioms impose constraints so that a term is of a particular type if a certain condition is satisfied. This condition is a conjunction of equations and unconditional tests of memberships.

System Modules. The system modules define the dynamic behavior of a system. This type of module augments the functional modules by the introduction of rewriting rules. This type of module offers a maximum degree of concurrency. A system module describes a "rewriting theory" which includes kinds, operations and three types of statements: equations, memberships and rewriting rules. These three types of statements can be conditional. A rewriting rule specifies a "local concurrent transition" which can proceed in a system. The execution of such transition, specified by the rule, can take place when the left part of a rule matches to a portion of the global state of the system and the condition of the rule is valid.

IV. ECATNETS

Let be Spec = (Σ, E) an algebraic specification of an abstracted data type given by the user. MTΣ,E(X) represents the free commutative monoid of the terms TΣ,E(X) (the ∑-algebra of the equivalence classes of the ∑-terms with variables in X, modulo the equations E) endowed with the internal operator ⊕ and having ∅ as the identity element. CATdas(E, X) is the structure of equivalence classes formed from the multi-sets of MTΣ,E(X) modulo the associative, commutative and identity axioms for the operator ⊕. An ECATNet [1] is a structure ECATNet = (P, T, sort, IC, DT, CT, Cap, TC) where:

- P is a finite set of places; T is a finite set of transitions;
- sort : P → S is a function that associates a sort to each place;
- IC (Input Condition): P × T → CATdas(E, X)~ where CATdas(E, X)~ is the set of the following elements: $CATdas(E,X)_{sort(p)} \cup \{empty\} \cup \{[m]\sim / [m] \in CATdas(E,X)_{sort(p)}\}$;
- DT (Destroyed Tokens): P × T → CATdas(E, X);
- CT (Created Tokens): P × T → CATdas(E, X);
- Cap (Capacity): P × T → CATdas(E,∅) is a partial function that for every p ∈ domain(Cap), associates a capacity Cap(p) ∈ CATdas(E,∅);
- TC (Transition Condition): T → $CATdas(E, X)_{bool}$ is a function such that for every t ∈ T, TC(t)∈ $CATdas(E, X(t))_{bool}$ where X(t) is called the transition context. A graphical representation of a generic ECATNet is given in (Fig. 1). The expressions IC(p, t), DT(p, t), CT(p', t) and TC(t) are multi-sets of equivalence classes of ∑-terms with ⊕ being the multi-set union operator which is associative, commutative and has the term ∅ (the empty multi-set) as the identity element. IC(p, t) specifies the condition on the input place marking for the enabling of the transition t. DT(p, t) and CT(p', t) specify, respectively, the multi-set of tokens to be removed from the marking of p and the multi-set of tokens to be created in the output place p', when t is fired. TC(t) is a Boolean term which specifies an additional enabling condition for the transition t. TC(t) may contain variables occurring in IC(p, t), DT(p, t) and CT(p', t). When TC(t) is omitted, the default value is the term True.

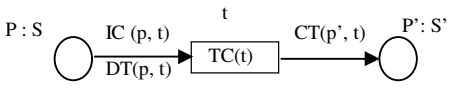

Fig. 1. A generic ECATNet

A transition t is fireable when several conditions are satisfied simultaneously:

- The first condition is that every IC(p, t) is satisfied for each input place p of the transition t:

If IC(p, t) is of the form [m] (positive case) then [m] must be included in the input place marking M(p). (if m = ∅, then IC(p, t) is always true)

If IC(p, t) is of the form [m]~ (negative case) then [m] must not be included in M(p).

If IC(p, t) = empty then M(p) has to be empty (i.e. M(p) = ∅).

- The second condition is that TC(t) is true.

Finally, the addition of the tokens CT(p', t) to the output place p' of t must not result in p' exceeding its capacity when this capacity is finite. When t is fired, DT(p, t) is removed from the input place p and simultaneously CT(p', t) is added to the output place p'. In the case where DT(p, t) is not included in M(p) (i.e. IC(p, t) ≠ DT(p, t)) we remove the elements which are common between these two multi-sets. For the sake of clarity, in the graphical representation of ECATNets, we note in italic the term IC(p, t) for a given transition t.

ECATNets Distributed State. Let P ={p_1, … p_n} the set of places of an ECATNet N. Each place p_i of N is marked with a finite multi-set of algebraic terms of sort Multiset. The constant ∅ represent the empty multi-set and ⊕ is the corresponding multi-set union operator, which is associative, commutative and has the constant ∅ as the identity element (ACI axioms). The marking of an ECATNet N has typically a structure of a multi-set made up of the marking of each place of N [10]. This last can be viewed as a term of the form <p_1 ; Mp_1> ⊗...⊗ <p_n ; Mp_n> of sort Marking endowed with a multi-set union operator denoted ⊗, for which the ACI axioms are also specified with the empty marking ∅ as the identity element. The marking of an ECATNet is axiomatized by the following equational theories. The theory MARKING-OPERATION defined in [6], specifies the multi-set difference, the multi-set cardinality, the multi-set inclusion and the multi-set intersection operators.

```
fmod PLACE-MARKING is
sorts  Place  Token Multiset Place-marking .
subsort Token < Multiset .
op Ems:-> Multiset.
------ the constant implementing the empty multi-set
op _⊕_ : Multiset  Multiset -> Multiset [assoc comm id: ∅].
op <_;_> : Place Multiset -> Place-marking .
endfm
fmod MARKING is pr PLACE-MARKING.
------ pr for protecting
sort Marking .  subsort Place-marking < Marking .
op Em :-> Marking.
------ The constant implementing the empty marking ∅
op _⊗_: Marking Marking -> Marking [assoc comm id:∅].
endfm
fmod MARKING-OPERATION is
Pr MARKING . vars mts mts1 mts2 : Multiset .
vars t1 t2 : Token .
------ Operations on multi-sets ------------------
op _Inclu_ : Multiset Multiset -> Bool .
op Nbr : Multiset -> Nat .
op _Supp_ : Multiset Multiset -> Multiset .
op _Inter_ : Multiset Multiset -> Multiset .
------ Difference ---------------------------------
eq mts1 Supp Ems = mts1 .
eq Ems Supp mts1 = Ems .
eq (t1 ⊕ mts1) Supp (t1 ⊕ mts2)= mts1 Supp mts2 .
eq (t1 ⊕ mts1) Supp t1 =  mts1 .
eq (t2 ⊕ mts1) Supp t1 =  t2 ⊕ (mts1 Supp t1)[ owise] .
------ Cardinality  -------------------------------
eq Nbr(Ems) = 0 . eq Nbr(t1) = s(0) .
eq Nbr(t1 ⊕ mts1) = s(Nbr(mts1)) .
------ Inclusion ----------------------------------
ceq mts1 Inclu Ems = true if mts1 == Ems.
ceq mts1 Inclu Ems = false if mts1 =/= Ems.
eq Ems Inclu mts1 = true.
eq t1 Inclu t2 ⊕ mts1= true if t1 == t2.
ceq t1 Inclu t2 ⊕ mts1 = t1 Inclu mts1 if t1 =/= t2 .
eq (t1 ⊕ mts1) Inclu (t1 ⊕ mts2) = mts1 Inclu mts2 .
eq (t1 ⊕ mts1) Inclu (t2 ⊕ mts2) = false [owise] .
------ Intersection  ------------------------------
eq mts1 Inter Ems = Ems .
eq (t1 ⊕ mts1) Inter (t1 ⊕ mts2) = t1 ⊕ (mts1 Inter mts2).
eq (t1 ⊕ mts1) Inter (t2 ⊕ mts2) = mts1 Inter mts2 [owise] .
endfm
```

ECATNets Rewrite Rules. The behaviour of the ECATNets ([1], [6]) is expressed by a set of rewrite rules describing the local effect of each transition and a set of deduction rules allowing to deduce an ECATNet marking from a previous one, after steps of concurrent rewritings. The form of these rules is rather complex if we consider all features of ECATNets such as places capacities, transitions conditions, the negative case and the fact that when a transition t is fired (in the case where DT(p, t) is not included in M(p)) we remove from its input place p, the common elements between DT(p, t) and M(p). In the following, we propose a general technique for the representation of the rewrite rules expressing the behaviour of ECATNets transitions taking into account all the mentioned features and extending that one already proposed for contextual ECATNets. A transition t which has p_1, p_2 and p_3 as input places and p_4 as an output place, such that IC(p_1, t) = [α1], IC(p_2, t)=[α2]˜ and IC(p_3, t) = empty, is formalized by a rewrite rule of the following general form (given in Maude syntax, where Mp_1, Mp_2 , Mp_3 and Mp_4 are variables of sort Multiset):

crl [t] : <p_1 ; Mp_1> ⊗ <p_2 ; Mp_2> ⊗ <p_3 ; Mp_3> ⊗ <p_4 ; Mp_4> → <p_1 ; Mp_1 **Supp** (Mp_1 **Inter** DT(p_1, t)) > ⊗ <p_2 ; Mp_2 **Supp** (Mp_2 **Inter** DT(p_2 ; t)) > ⊗ <p_4 ; CT(p_4, t) ⊕ Mp_4>
if ((α1 **Inclu** Mp_1) and not (α2 **Inclu** Mp_2) and (Mp_3 == ∅) and **Nbr** (CT(p_4, t) ⊕ Mp_4) ≤ Cap (p_4)) .

The test on the capacity can be omitted, if Cap(p_4) is infinite. If the transition is associated to a local condition TC(t), the rewrite rule will contain the additional component [TC(t)→true]. The application of the splitting and/or recombining axioms on places marking (i.e. ∀p∈ P, where P is the set of an ECATNet places, <p ; m_1 ⊕ m_2> ≡<p ; m_1> ⊗ <p ; m_2>) are restricted in order to obtain coherent execution results under Maude engine. In the case when for each transition t of an ECATNet, $[IC(p, t)]_{\oplus}$ = $[DT(p, t)]_{\oplus}$, the rewrite rule formalizing the behaviour of t can be simplified as follows : < p, $[IC(p, t)]_{\oplus}$ > → < p', $[CT(p', t)]_{\oplus}$ >. Note that the splitting and/or recombining axioms can be applied in this case which allows detecting the maximum of parallelism in ECATNets computations by judiciously splitting and/or recombining different multi-sets equivalence class of terms.

V. Meta-Level Computation in Maude

Maude provides a plate-form getting easy implementation for ECATNets' tools. 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 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.

A. *Descent Functions in Maude*

Function metaMatch. The process of matching two terms at the top is reified by a built-in function metaMatch:
op metaMatch : Module Term Term Condition Nat _ > Substitution? .

The operation metaMatch(R, t, t', Cond, n) tries to match at the top terms t and t' in the module R in such way that the resulting substitution satisfies the condition Cond. The natural number is used to indicate possible matches. In success case, this function returns substitution as result. In failure case, it returns noMatch. From theoretical point of view, consider that T_Σ : is a Σ–algebra of ground terms in the signature Σ and $T_\Sigma(X)$ is a term algebra where terms may have variables in a set of variables X. Given a term $t \in T_\Sigma(X)$, corresponding to the lefthand side of an oriented equation, and a subject ground term $t' \in T_\Sigma(X)$, we say that t matches t' if there is a substitution σ such that $\sigma(t) \equiv t'$, that is, σ(t) and t' are syntactically equal terms.

Selectors Functions. A selector function allows extracting a part of the code. For example, selector function getRls takes meta-representation of a module and returns meta-representation of rules:
op getRls : Module –> RuleSet .

Operators used to meta-represent sets of rules are:
sorts Rule RuleSet . subsort Rule < RuleSet .
op rl_=>_[_] : Term Term AttrSet –> Rule [ctor] .
op crl_=>_if_[_] . : Term Term Condition AttrSet –> rule [ctor] .
op none : -> RuleSet [ctor] .
op __ : RuleSet RuleSet –> RuleSet [ctor assoc comm id: none] .

VI. Discussion and Adaptation of Reduction Rule 'Side Condition Places' to ECATNets

ECATNets are described in meta-level (as data) and may be manipulated by another module. We can apply Maude descent functions on it. In sequel, M is the name of a module describing an ECATNet system and T0 is an initial marking. Let's consider that all places of M have an infinite capacity. Let R be the set of all rewrites rules of M. We consider p an input place (resp. output place) for r if p appears in the lefthand (resp. righthand) side of the rule (transition) r. We need to define some functions used in the context of our adaptation of reduction rules to ECATNet. The function RuleLeft(r) (resp. RuleRight(r)) returns the term in lefthand (resp. righthand) side of a rule r. The function RuleCondition(Rl) returns the condition part of the rule Rl. The function GetSubTermConcerningPlace(T, p) returns the SubTerm of T that concerns only the place p. So, for a term T = < p ; T1 > .. < p ; Tn > . T' (T' is independent from p), this function returns the term < p ; T1 > .. < p ; Tn >.

Application Condition. There is a place p such that for all transitions (rewriting rules) Rl, we have:
EquivalentTerms(GetSubTermConcerningPlace(RuleLeft(Rl), p), GetSubTermConcerningPlace(RuleRight(Rl), p), RuleCondition(Rl)) == true. **[a]**

Application. For every injective assignment of an occurrence of a term in GetSubTermConcerningPlace(T0, p) to GetSubTermConcerningPlace(RuleRight (Rl), p) and every unifier sub for the set of pairs of assigned terms, insert a transition Rl_{sub} and for all $p' \in P \setminus \{p\}$,
GetSubTermConcerningPlace(RuleLeft(Rl_{sub}), p') =
GetSubTermConcerningPlace(RuleLeft(Rl_{suv}), p')
and GetSubTermConcerningPlace(RuleRight(Rl_{sub}), p') =
GetSubTermConcerningPlace(RuleRight(Rl_{suv}), p') **[b]**

Remove p and all terms < p ; T > for $p \in P$ figuring in rewrite rules of M.

VII. Implementation of Reduction Rule 'Side Condition Places'

For simplicity reasons, we present some principal functions of the application which should be sufficient to explain all the implementation. We first explain how to implement the EquivalentTerms function. After that, we give details about the implementation of 'side condition place' reduction rule.

A. *Implementation of the Function EquivalentTerms*

In a module called COMPLETENESS-DECISION, we propose the description of this function. The condition (metaMatch(M, T1, T2, CD, 0) =/= noMatch) decides that the term T2 is the term T1 after doing a certain substitution of its variables if a certain condition CD is true.

```
op EquivalentTerms: Module Term Term Condition -> Bool .
eq EquivalentTerms(M, T1, T2, CD) =
if metaMatch(M, T1, T2, CD, 0) =/= noMatch
   and  metaMatch(M, T2, T1, CD, 0) =/= noMatch
then true
else if metaMatch(M, T1, T2, CD, 0) =/= noMatch
        or  metaMatch(M, T2, T1, CD, 0) =/= noMatch
      then EquivalentTerms1(M, GetTermsInPlace(T1),
           GetTermsInPlace(T2))
      else false  fi fi .
```

However, the condition (metaMatch(M, T1, T2, CD, 0) =/= noMatch) is not sufficient to implement all the signification of EquivalentTerms. As described in [9], if we take, for instance, T1 = < p ; a > . < p ; b > (ground term with a and b are constant of sort S) and T2 = < p ; x > . < p ; succ(x) > (x is a variable of

sort S), we have in this case metaMatch(M, T1, T2, CD, 0) == noMatch. However, if a and b are the only elements of the sort S with a = succ(b) and b = succ(a), then the application condition of reduction rule is verified because the multi-set composed of x and succ(x) is always equal to the multi-set composed of a and b, for every substitution. In this case, T1 may be considered equivalent than T2. We can consider also T1 equivalent to T2 (we explain this in the next sub-section). We can say that T1 is equivalent to T2.

For this reason, we propose the function EquivalentTerms1 that deals with this case. This function takes three parameters, the first one is of sort Module and the two others are of sort TermList. In the expression EquivalentTerms1 (M, GetTermsInPlace(T1), GetTermsInPlace(T2)), the function GetTermsInPlace(T) extracts terms figuring in the places of T, i.e, if T = < p1 ; t1 > .. < pn ; tn > such that GetTermsInPlace returns the list of terms t1,..,tn.

The global behaviour of the function EquivalentTerms1 is to consider for each sort S, a sub-list of terms (GTL) in GetTermsInPlace(T1) being all of the same sort S and a sub-list of terms (TL) in GetTermsInPlace(T2) of the same sort S. These two sub-lists must have same lengths. GTL contains only ground (variable-free) and TL must contain any terms, else the condition of the reduction rule isn't valid.

The function EquivalentTerms1verifies the set of terms of GTL (without repetition) constitutes all elements of sort S (property 1). Moreover, this function looks for all possible substitutions for ti (i=1,.., n). Variables in common between terms must have same values. After that, the function shows that for each substitution, GTL and TL are identical in terms of their contents (property 2). If it exist a substitution of all variables of terms of TL, such that GTL and TL are different in terms of their contents, then the condition of the rule is not verified and EquivalentTerms1(M, GTL, TL) returns false.

Let GTL be the list containing terms gt1,..,gtn and TL the list containing t1,..,tn. Let's note that after showing gt1,..,gtn are the alone elements of S, then if we have a substitution for each term ti,(i=1,..,n), it is, surely, equal to one of terms gtj (j =1,..,n).

The implementation in Maude of the checking of the property 2 is possible now thanks to the 'Completeness Checker' tool developed by Hendrix in [5]. This tool allows the checking of the property 'Sufficient completeness'. This property indicates that the canonical form of any ground is based only on 'constructor' operations. The declaration of the function checkCompleteness in Maude is as follows :

```
op checkCompleteness : Module MembAxSet EquationSet
MembAxSet -> ProofObligationSet .
```

The first parameter of this function represents a module, the second one represents subjects, the third one represents equations and the fourth parameter represents constructor memberships.

The function checkCompleteness(M, Subjects, Eqs, Mbs) discards all equations and constructor membership that do not unify with a subject, but if a subject has a non-conditional matching, this function removes this it. If all subjects have a matching, then this function returns that there is no proof obligation (none.ProofObligationSet).

To use this function in our definition of EquivalentTerms1, we proceed first to create a new module New-M containing the set of terms of sort S as the alone elements of S. Let gt1,..,gtn be the terms of sort S, the module containing the definition of its terms as the alone elements of sort S, must contain all definitions of operations and constructor memberships in the following manner :

- If gti (i=1,..,n) is a constant (not-parameterized operation), then we declare :

```
op gti : -> S .
```

- If gti (i=1,..,n) is not a constant (parameterized operation), then gti is of the form f(r1,..,rm). For the moment, we accept only the case where r1,..,rm are not-parameterized constants. The declaration of ti is of the following form :

```
op f : S1 .. Sm -> S .
cmb f(x1:S1, …, xm :Sm) : S if (x1:S1 == r1.S1) and … and
(xm:Sm == rm.Sm) .
```

To check the previous property, we define the function SortCompleteness as follows:

```
op SortCompleteness : Module Module Type -> Bool .
eq SortCompleteness(New-M, M, S) =
if checkCompleteness(New-M, patterns(M, S),
  getsEqs(New-M),   patterns(New-M, S)) ==
  (none).ProofObligationSet  then true else false fi .
```

The function patterns(M, S) looks for ground terms that constitute initial algebra for the sort S. In infinity case, this function returns 'model' terms covering all elements of the sort S. The function SortCompleteness searches for each term of sort S in M, if it exist a term of the same sort in New-M, in such way it does not exist a matching between the terms. If it exists a ground term in M and for each term in New-M, it does not exist a matching between the two terms, then the condition is not valid. In this case, defined elements in New-M are not the alone elements of S.

In the following, we present in details the implementation of the function EquivalentTerms1 and some functions called by it. EquivalentTerms1(M, TL1, TL2) extracts all sorts of elements of TL1 and calls MoreGeneralTerms2(M, TL1, TL2, GetSortsOfTerms(M, (TL1, TL2))). The function GetSortsOfTerms(M, (TL1, TL2) returns sorts of all elements in TL1 and TL2:

```
op EquivalentTerms1 : Module TermList TermList -> Bool .
eq EquivalentTerms1(M, nil, nil) = true .
eq EquivalentTerms1(M, nil, (T2, TL2)) = false .
eq EquivalentTerms1(M, TL1, TL2) =
 if length(TL1) == length(TL2)
  then EquivalentTermsComp2(M, TL1, TL2,
        GetSortsOfTerms(M, (TL1, TL2)))
  else false fi .
```

The function EquivalentTermsComp2(M, TL1, TL2, (S TpL)) extracts a sub-list of TL1 and a sub-List of TL2. Terms of these two sub-list must be of the same sort S. This function calls EquivalentTermLists(M, GetTermsOfSort(M, TL1, S), GetTermsOfSort(M, TL1, S)) :

```
op EquivalentTermsComp2 : Module TermList TermList
TypeList -> Bool .
eq EquivalentTermsComp2(M, TL1, TL2, nil) = true .
eq EquivalentTermsComp2(M, TL1, TL2, (S TpL)) =
if EquivalentTermLists(M, GetTermsOfSort(M, TL1, S),
   GetTermsOfSort(M, TL2, S)) == true
 then EquivalentTermsComp2(M, TL1, TL2, TpL)
 else false fi .
```

The function EquivalentTermLists (M, (T1, TL1), (T2, TL2)) removes from TL1 any non ground term that has its equivalent terms in TL2 and calls EquivalentTermLists1. The function EquivalentVars(M, TL1, TL2) returns some terms of TL1 that have equivalents in TL2. The function SameTypeTest(TL1) returns true if all elements of TL1 are of the same type.

```
op EquivalentTermLists : Module TermList TermList -> Bool
.
eq EquivalentTermLists(M, (T1, TL1), (T2, TL2)) =
if (length(TL1) == length(TL2))
   and  (SameTypeTest(TL1) == true)
   and (SameTypeTest(TL2) == true)
   and (getType(T1) == getType(T2))
 then EquivalentTermLists1(M,
       TermListSub(TL1, EquivalentVars(M, TL1, TL2)),
       TermListSub(TL2, EquivalentVars(M, TL2, TL1)),
       getType(T1)) else false fi .
```

EquivalentTermLists1(M, TL1, TL2, S) checks first that one list contains ground terms and the other one contains any terms. This function checks the property 1 described above and calls MoreGeneralTermLists2(M, TL2, TL1) :

```
op MoreGeneralTermLists1 : Module TermList TermList
Sort -> Bool .
eq MoreGeneralTermLists1(M, TL1, TL2, S) =
if  (GroundTermsTest(TL1) == true
   and VarsTest(TL2) == true)
  then if SortCompleteness(M, TermListNoDup(M, TL1), S)
          == true
        then MoreGeneralTermLists2(M, TL2, TL1)
        else false  fi  else false fi .

op EquivalentTermLists1 : Module TermList TermList Sort
-> Bool .
eq EquivalentTermLists1(M, TL1, TL2, S) =
if  (GroundTermsTest(TL1) == true
    and NotGroundTermsTest(TL2) == true)
 then if SortCompleteness(M, TermListNoDup(M, TL1), S)
         == true
       then EquivalentTermLists2(M, TL2, TL1) else false fi
       else if (NotGroundTermsTest(TL1) == true
             and GroundTermsTest(TL2) == true)
            then if SortCompleteness(M,
                    TermListNoDup(M, TL2), S) == true
                 then EquivalentTermLists2(M, TL1, TL2)
                 else false fi
        else false fi  fi .
```

EquivalentTermLists2(M, TL, GTL) extracts all possible substitutions for variables appearing in TL according to values of GTL by calling the function ValidSubsForTList(M, TL, GTL) :

```
op EquivalentTermLists2 : Module TermList
GroundTermList -> Bool .
eq EquivalentTermLists2(M, TL, GTL) =
EquivalentTermLists3(TL, GTL, ValidSubsForTList(M, TL,
GTL)) .
```

EquivalentTermLists3(TL, GTL, (Sub ;; Subs)) checks the property 2 described above. For each substitution Sub, the function SubstitutionTermList(TL, Sub) returns a new list after replacing variables of TL by values according to the contents of Sub. EqualTermList(SubstitutionTermList(TL, Sub), GTL) returns true if the obtained new list (SubstitutionTermList(TL, Sub)) is equal to the list GTL. If this function returns false, the function EquivalentTermLists3(TL, GTL, (Sub ;; Subs)) returns false, else it continues this test for another substitution in Subs. In the case of expiring all substitutions (emptySubList), EquivalentTermLists3(TL, GTL, emptySubList) returns true.

```
op EquivalentTermLists3 : TermList GroundTermList
SubList -> Bool .
eq EquivalentTermLists3(TL, GTL, emptySubList) = true .
eq EquivalentTermLists3(TL, GTL, Sub) =
if EqualTermList(SubstitutionTermList(TL, Sub), GTL) ==
true  then true else false fi .
eq EquivalentTermLists3(TL, GTL, (Sub ;; Subs)) =
 if EqualTermList(SubstitutionTermList(TL, Sub), GTL)
    == true
 then MoreGeneralTermLists3(TL, GTL, Subs)  else false fi .
```

B. Implementation of Reduction Rule 'Side Condition Places'

The module implementing reduction rule 'side condition place' is called SIDE-CONDITION-PLACES-REDUCTION, we present and explain some of its principal functions:

```
mod SIDE-CONDITION-PLACES-REDUCTION  is
…
var M : Module . var P : Term . vars Rl Rl1 Rl2 : Rule .
vars Rls Rls1 Rls2 : RuleSet . vars T T0 T1 T2 : Term .
var TL : TermList . var GTL : GroundTermList .
var CD : Condition . var L : List .  var Sub : Substitution .
var Subs : SubList .
```

Module-AfterSide-Condition-Places-Reduction(M, T0) is the main function of the application, in the code of this function we find the function AddTransitionsInModule(M, Add-Sub-Rules(M, T0, Get-All-Side-Condition-Places(M))) which creates new transitions according to every side condition place detected. After adding such transitions we call the function DeletePlacesAndConnectedArcsInModule to

delete all side conditions places and all arcs connected to them.

```
op Module-AfterSide-Condition-Places-Reduction : Module Term -> Module .
eq Module-AfterSide-Condition-Places-Reduction(M, T0) =
DeletePlacesAndConnectedArcsInModule(AddTransitionsIn Module(M,
Add-Sub-Rules(M, T0, Get-All-Side-Condition-Places(M))),
Get-All-Side-Condition-Places(M)) .
```

The function Get-All-Side-Condition-Places(M) returns a list of all side condition places, this function calls GetPlacesOfModule(M) to extract all places of the module and pass this list of places as parameter to the function All-Side-Condition-Places(M, GetPlacesOfModule(M)).

```
op Get-All-Side-Condition-Places : Module -> List .
eq Get-All-Side-Condition-Places(M) =
All-Side-Condition-Places(M, GetPlacesOfModule(M)) .
```

The function All-Side-Condition-Places(M, L) checks if every place of the ECATNet M, is a side condition place or not. If a place head(L) is side condition place (SideConditionPlace(M, head(L)) returns true in this case), so it is added to the list returned by this function otherwise it is discarded.

```
op All-Side-Condition-Places : Module List -> List .
eq All-Side-Condition-Places(M, L) =
if L == emptyList then emptyList
else if SideConditionPlace(M, head(L)) == true
      then head(L) . All-Side-Condition-Places(M, tail(L))
      else All-Side-Condition-Places(M, tail(L)) fi fi .
```

The function SideConditionPlace(M, P) extracts all rules of the module (call of getRls(M)) and passes them as parameter to VerifySideConditionPlace(M, P, getRls(M)) which returns true if P is a side condition place and false otherwise. This function checks for the place P and every rule rl in the module if the application condition if satisfied or not.

```
op SideConditionPlace : Module Term -> Bool .
eq SideConditionPlace(M, P) =
VerifySideConditionPlace(M, P, getRls(M)) .
op VerifySideConditionPlace : Module Term RuleSet -> Bool .
eq VerifySideConditionPlace(M, P, none) = true .
eq VerifySideConditionPlace(M, P, Rl Rls) =
if EquivalentTerms(M,
GetSubTermConcerningPlace(RuleLeft(Rl), P),
  GetSubTermConcerningPlace(RuleRight(Rl), P),
RuleCondition(Rl)) == true
 then VerifySideConditionPlace(M, P, Rls) else false fi .
```

The function Add-Sub-Rules(M, T0, L) creates rules for all side condition places in the list L by calling Add-Sub-Rules-1(M, T0, head(L)) which creates rules for every place head(L).

```
op Add-Sub-Rules : Module Term List -> RuleSet .
eq Add-Sub-Rules(M, T0, L) =
 if L == emptyList then none
 else Add-Sub-Rules-1(M, T0, head(L)) Add-Sub-Rules(M, T0, tail(L)) fi .
op Add-Sub-Rules-1 : Module Term Term -> RuleSet .
eq Add-Sub-Rules-1(M, T0, P) =
Add-Sub-Rules-2(M, T0, P,
Get-Side-Condition-Place-Rules(M, P) ) .
op Add-Sub-Rules-2 : Module Term Term RuleSet -> RuleSet .
eq Add-Sub-Rules-2(M, T0, P, none) = none .
eq Add-Sub-Rules-2(M, T0, P, Rl Rls) =
Add-Sub-Rules-3(M, T0, P, Rl) Add-Sub-Rules-2(M, T0, P, Rls) .
op Add-Sub-Rules-3 : Module Term Term Rule -> RuleSet .
eq Add-Sub-Rules-3(M, T0, P, Rl) =
Create-Rules-For-Subs-Except-Place(M, P, Rl,
Get-Injective-Assignment(M, P, Rl, T0)) .
```

The function Get-Injective-Assignment(M, P, Rl, T0) returns an injective assignment of an occurrence of a term in GetTermsInPlace(GetSubTermConcerningPlace(T0)) (tokens in P), to GetTermsInPlace(GetSubTermConcerningPlace(RuleLeft(Rl), P)):

```
op Get-Injective-Assignment : Module Term Rule GroundTerm -> SubList .
eq Get-Injective-Assignment(M, P, Rl, T0) =
ValidSubsForTList(M, GetTermsInPlace(
GetSubTermConcerningPlace(RuleLeft(Rl), P)),
GetTermsInPlace(GetSubTermConcerningPlace(T0))) .
```

Get-Side-Condition-Place-Rules(M, P) extracts the rewriting rules that verifying the condition application of the reduction rule **[a]** :

```
op Get-Side-Condition-Place-Rules : Module Term -> RuleSet .
eq Get-Side-Condition-Place-Rules(M, P) =
Get-Side-Condition-Place-Rules-1(M, P, getRls(M)) .
op Get-Side-Condition-Place-Rules-1 : Module Term RuleSet -> RuleSet .
…
```

Create-Rules-For-Subs-Except-Place (M, P, Rl, Subs) and the all functions called by this function are developed to create Rl_{sub} that satisfies **[b]**:

```
op Create-Rules-For-Subs-Except-Place : Module Term Rule SubList -> RuleSet .
eq Create-Rules-For-Subs-Except-Place (M, P, Rl, Subs) =
Create-Rules-For-Subs(DeletePlacesDefInModule(M, P), Rl, Subs) .

op Create-Rules-For-Subs : Module Rule SubList -> RuleSet .
eq Create-Rules-For-Subs(M, Rl, Subs) =
if Subs =/= emptySubList then Create-Rule-For-Sub(M, Rl,
   Sub-head(Subs))
  Create-Rules-For-Subs(M, Rl, Sub-tail(Subs)) else none fi .
…
endm
```

VIII. EXAMPLE

A. *Presentation of the Example*

Figure 2 represents the ECATNet model of the dining philosopher problem. If inscriptions IC(p, t) and DT(p, t) are equals, then we present only IC(p, t) on the arc (p, t). Rewriting rules of this system will be presented directly in Maude.

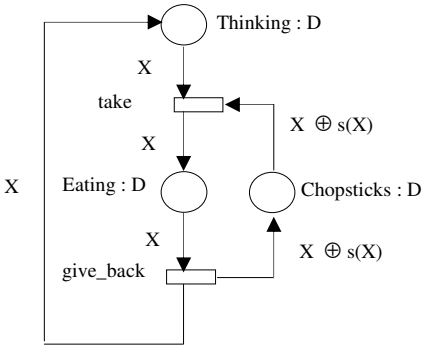

Fig. 2. ECATNet Model of Philosopher.

Let's note that this example does not contain for the moment any 'side condition place'. To make the tool implementing 'side condition place' reduction rule applicable on this example, we have to reduce it by using another reduction rule 'free choice decisions' which is not detailed in this paper. But, we need to explain the adaptation of this reduction rule to ECATNet in informal way, and through the example of philosophers.

B. *Application of Reduction Rules on the Example*

In this section, we introduce in informal way, and through the example of philosophers, how to adapt the reduction rule 'free choice decisions' to ECATNet.

Reduction Rule: Free Choice Decisions. Let's note that the transition take puts any value X of the domain D, and give_back needs to be fired any value of the same domain D (the term X covers all elements of the domain of Eating). The place Eating is empty at the initial marking. We can delete the place Eating and merge the transitions take (t_i) and give_back (t_j) to get one transition take_give_back (t_{ij}). The input of the new transition is the input of the transition take and the output of the new transition is the output of the transition give_back as depicted in the figure 3.

Properties.
1. t_i is live in N **iff** there is a j such that t_{ij} is live in N'
2. p in unbounded in N **iff** a place in (pF)F is unbounded in N'
3. Every formula which does not concern p and (pF)F holds in N if it holds in N'

Benefits. One of the benefits of this rule is that a sequence of two transitions of the original net is replaced by a single transition in the reduced net. Consequently the number of states of the accessibility graph decreases.

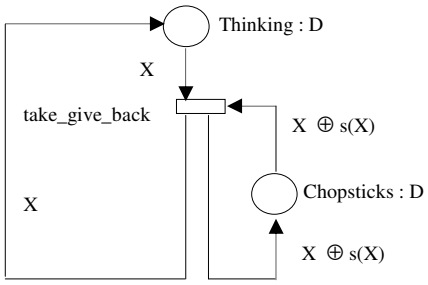

Fig. 3. Reduction of the Philosopher net after applying 'free choice decisions' reduction rule.

The two places in this obtained ECATNet are 'side condition places'. If we eliminate the place Chopsticks, so we get the reduced net in the figure 4.

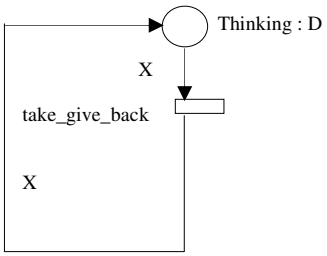

Fig. 4. Reduction of the reduced Philosopher net after applying 'side condition place' and eliminating chopsticks.

The reduced net of Fig. 4 still has another 'side condition place', which is Thinking, by eliminating this place, we get the empty net. In the sequel, we give abbreviated names to original ECATNet and to every ECATNet obtained after a reduction. The table 1 resumes ECATNets' naming.

TABLE I
NAMING OF OBTAINED ECATNETS

ECATNet	Definition
N	Original ECATNet
N_1	Reduced ECATNet after applying 'free choice decisions' rule
N_2	Reduced ECATNet after applying 'side condition place' rule on N_1 and eliminating Chopsticks
N_3	Reduced ECATNet after applying 'side condition place' rule on N_1 and eliminating Thinking
N_4	Reduced ECATNet after applying 'side condition place' rule on N_1 and eliminating both Chopsticks and Thinking

C. *Implementation of the Example*

Now, we will present the module describing this ECATNet in Maude meta-level. Of course, the user is not obliged to write his/her ECATNet in a meta-representation. He/she can write it in the ordinary mode, and he/she uses the function upModule of Maude which allows transforming the representation of a module to its meta-representation. The transformation in the other direction is possible also thanks to the function downModule. But just for well explain our work, we have preferred presenting the module describing the previous ECATNet in its meta-representation. In a module called

META-LEVEL-PHILOSOPHER-ECATNET-SYSTEM, the module META-PHILOSOPHER is defined as a constant of Module type and its content is described using an equation. 'META-ECATNET is the description in meta-level of the module containing basic operations of ECATNet. Because of the simplicity of the example, we do not need all multi-sets operations describe above, so the module META-ECATNET will contain a minimum of operations manipulating marking. We define in Maude only three sorts Place, Token and Marking. For syntactic reason, we define the operation '_._' to implement the operation described above ⊗. For simplicity reason, we have not defined an operation implementing ⊕. The operation '_._' which implements ⊗ is sufficient while basing on the concept of decomposition. For example, (p, a ⊕ b) can be written as < p ; a > . < p ; b >. The module META-DOMAIN contains the definition of the data type D which defines the algebraic terms that we find in the places.

```
fmod
META-LEVEL-PHILOSOPHER-ECATNET-SYSTEM is
pr LIST-OF-TRIPLE . op META-ECATNET : -> Module .
op META-DOMAIN : -> Module .
op META-PHILOSOPHER : -> Module .
------------------------------------------------------------------------
eq META-ECATNET = (fmod 'META-ECATNET is nil
sorts 'Place ; 'Token ; 'Marking .
none op 'Em : nil -> 'Marking [none] .
op '<_;_> : 'Place 'Token -> 'Marking [none] .
op '_._ : 'Marking 'Marking -> 'Marking [assoc comm id : 'Em.
Marking] .
none none endfm) .
------------------------------------------------------------------------
eq META-DOMAIN = (fmod 'META-DOMAIN is nil
sorts 'D . none
op '1 : nil -> 'D [none] . op '2  : nil -> 'D [none] .
op '3 : nil -> 'D [none] .  op 'S_ : 'D -> 'D [none] .
none eq 'S_['1.D] = '2.D [none] . eq 'S_['2.D] = '3.D [none] .
eq 'S_['3.D] = '1.D [none] .
none endfm) .
------------------------------------------------------------------------
eq META-PHILOSOPHER =
(mod 'META-PHILOSOPHER is
including META-ECATNET . including 'META-DOMAIN .
none subsort 'D <  'Token .
-------- Places -------
op 'Thinking : nil -> 'Place [none] .
op 'Chopsticks : nil -> 'Place [none] .  none none
-------- Rewriting Rules --------
rl '_._['<_;_>['Thinking.Place, 'X:D],
 '_._['<_;_>['Chopsticks.Place, 'X:D],
'<_;_>['Chopsticks.Place, 'S_['X:D]]]]
=> '_._['<_;_>['Thinking.Place, 'X:D],
 '_._['<_;_>['Chopsticks.Place, 'X:D],
'<_;_>['Chopsticks.Place, 'S_['X:D]]]]
[label('takegiveback)] .
endm) . endfm
```

D. *Application of the Tool on the Example*

In the framework of this work, we used as platform the version 2.3 of Maude under Linux. For the application of the tool on the example, we can call any function. For instance, to know what are side condition places, we use the command 'red' (red for reduce) to call the function Get-All-Condition-Places(META-PHILOSOPHER) which returns in this case a list containing the two places 'Thinking ' and 'Chopsticks'. To apply the reduction rule 'side condition place', we have to call the principal function of the application with a chosen initial marking:

```
red Module-AfterSide-Condition-Places-Reduction(
META-PHILOSOPHER,
'_._['<_;_>['Thinking.Place, '1.D],
 '_._['<_;_>['Thinking.Place, '2.D] ,
'_._['<_;_>['Thinking.Place, '3.D],
 '_._['<_;_>['Chopsticks.Place, '1.D],
'_._['<_;_>['Chopsticks.Place, '2.D ],
 '<_;_>['Chopsticks.Place, '3.D ] ] ] ] ] ]) .
```

As illustrated in figure 5, the tool eliminates the two transitions (rewriting rule) in the ECATNet META-PHILOSOPHER after applying the reduction rule 'side condition place'. The new obtained module does not contain any rewriting rule (none).

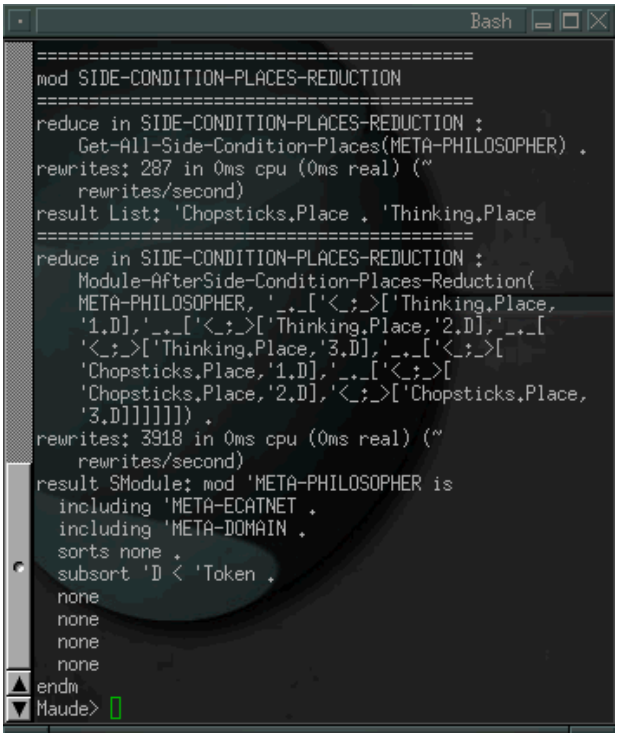

Fig. 5. Reduction of the Module META-PHILOSOPHER after applying 'side condition places' reduction rule.

IX. Benefits of the Reduction Rules

Reduction helps us to get smaller nets in order to detect easily and quickly some properties that are preserved by reduction rules like accessibility graph construction, boundness, liveness and LTL formulas. Let's show the impact of the reduction rules on the analysis of these properties through the previous example. Just for simplicity, we used in the sequel the Maude tools (accessibility analyser, LTL

Model Checker, etc.) under Windows system which give the same result as same Maude tools under Linux.

Accessibility Graph Construction. The sequence of the two transitions take and give_back of the net N is replaced by a single transition take_give_back in the reduced net N'. Consequently the number of states of a reachability graph decreases from 20 states (from 0 to 19) as depicted in the figure 6, to only 1 state as described in the figure 7.

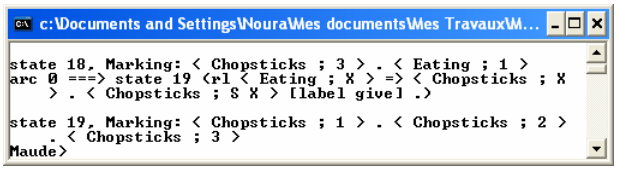

Fig. 6. Part of the accessibility graph of the net N (**before** applying 'free choice decisions' reduction rule)

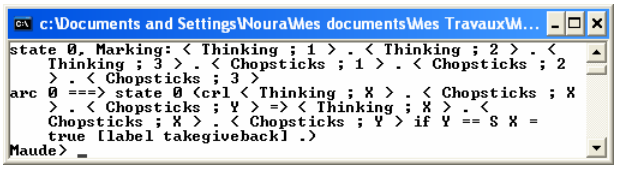

Fig. 7. Part of the accessibility graph of the net N' (**after** applying 'free choice decisions' reduction rule)

Let's note that we construct an accessibility graph in Maude by executing the command search as follows:
search in PHILOSOPHER : Initial-Marking =>* M:Marking .
show search graph .

The command search looks for all states that are accessible from the initial marking and next it shows the entire graph (show search graph).

Boundeness.
- According to the property 2 of 'free choice decisions' reduction rule, we conclude that Eating is unbounded in N **iff** Thinking or Chopsticks are unbounded in N_1, which is equivalent to say: Eating is bounded in N **iff** Thinking and Chopsticks are bounded in N_1.
- According to the property 2 of 'side condition place' rule, we conclude that Thinking and Chopsticks are both bounded in N_1.

Consequently, all three places are bounded.

Liveness.
- According to the property 1 of 'free choice decisions' reduction rule we have: take and give_back are live in N **iff** take_give_back is live in N_1.
- According to the property 3 of 'side condition place' rule, we have: take_give_back is live in N' **iff** take_give_back is live in N_2.

It is easy to detect that take_give_back is live in the reduced net N_2. Thus, the transition take and give_back are live in N.

LTL Model Checking.
- According to the property 3 of 'free choice decisions' reduction rule we have: each formula which concerns the place Chopsticks holds in N **iff** it holds in N_1;
- According to the property 3 of 'side condition place' rule, we have: each formula which concerns the place Chopsticks holds in N_1 **iff** it holds in N_3;

So, let's define a LTL property that concerns only the place Chopsticks and we check its correctness for the three nets N, N_1 and N_3. After that, we use LTL Model Checker of Maude to compare the number of rewriting steps needed to check the property for each net. First, we define a predicate:

```
op Term-In-Place : D Place -> Prop .
ceq M |=  Term-In-Place(X, P) = true
if Find-Term-In-Place(X, P, M) ==  true .
```

Such that, the function Find-Term-In-Place(X, P, M) returns true when the algebraic term X is in the place P in the marking M. In this case, the predicate Term-In-Place(X, P) is correct. Now, let's consider the following property:

```
op MUTUAL-EXCLUSION : Place -> Prop .
eq MUTUAL-EXCLUSION(P) =
( [] ( (~ Term-In-Place(1, P) /\ ~ Term-In-Place(2, P) ) =>
Term-In-Place(3, P) ) )   ****** [c]
/\ ( [] ( (~ Term-In-Place(2, P) /\ ~ Term-In-Place(3, P) ) =>
Term-In-Place(1, P)  ) )
/\ ( [] ( (~ Term-In-Place(3, P) /\ ~ Term-In-Place(1, P) ) =>
Term-In-Place(2, P) ) ) .
```

The first line of the property (**[c]**) means: always, if the terms 1 and 2 are not in Chopsticks (philosopher 1 is eating) then the term 3 is in Chopsticks (philosophers 2 and 3 must wait). It is a mutual exclusion property.

Finally, we call Maude LTL Model Checker to check this property for the three nets N, N_1 and N_3 as follows:

```
red modelCheck(Initial-Marking,
                MUTUAL-EXCLUSION(Chopsticks) ) .
```

We get the results in the table 2. Fig. 8 shows the result of the Maude Model Checking of the property for the net N_3.

TABLE II
REWRITING STEPS REQUIRED TO CHECK THE PROPERTY FOR THE THREE NETS

ECATNet	Rewriting steps
N	657
N_1	158
N_3	128

We note how reduction rules decreased significantly the number of rewriting steps (and so the time) required to check the property.

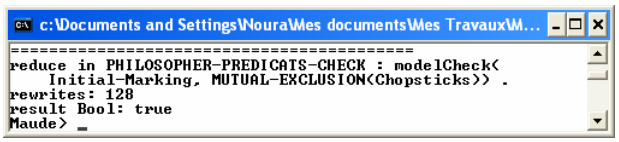

Fig. 8. Model Checking of the property in the case of the net N_3

X. CONCLUSION

In this paper, we have shown the adaptation of two reduction rules for ECATNets. These two rules were defined in [9] for APNs. First, we described in informal way how we adapted the reduction 'free choice decisions' to ECATNet. But, we gave in formal way the adaptation and the implementation of the reduction rule 'side condition place' for ECATNet by using rewriting logic. Of course, Maude is considered that it is the most appropriate language to develop

such tools implementing reduction rules for ECATNets. In addition of the integration of ECATNet in Maude, this last is a language of specification equipped with a safe and complete semantics and it is a programming language with a platform, allowing us to implement and validate ECATNets' properties. Maude has a large battery of many tools like: simulator, accessibility analysis, Model Checking, etc.

We showed also in this paper, the benefits we gained when we use reduction rules. These last reduce the size of the original ECATNet to get small reduced ECATNet and make easier and faster the reasoning about some preserved properties like accessibility graph construction, boundness, liveness and LTL formulas. This consequence was defended through the example of philosophers.

REFERENCES

[1] 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.

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

[3] N. Boudiaf, K. Barkaoui, A. Chaoui, "Implémentation des Règles de Réduction des ECATNets dans Maude," *Proceedings of Mosim'06 Conference*, Rabat, Maroc, 2006, pp. 1505–1514.

[4] M. Clavel et aL., *Maude Manual (Version 2.3)*. Internal report, SRI International, 2007.

[5] J. Hendrix, M. Clavel and J. Meseguer, "A Sufficient Completeness Reasoning Tool for Partial Specifications," Volume 3467 of LNCS, Spring-Verlag, 2005, pp. 165–174.

[6] A. Hicheur, K. Barkaoui and N. Boudiaf, "Modeling Workflows with Recursive ECATNets," *PN&WM 2006, IEEE post-proceedings*, 2006, pp. 7–14.

[7] J. Meseguer, "Rewriting Logic as a Semantic Framework of Concurrency: a Progress Report," Volume 1119 of LNCS, Springer Verlag, 1996, pp. 331–372.

[8] J. Meseguer, "Rewriting logic and Maude: a Wide-Spectrum Semantic Framework for Object-based Distributed Systems," *(FMOODS'2000), In S. Smith and C.L. Talcott, editors*, 2000, pp. 89–117.

[9] K. Schmidt, "Applying Reduction Rules to Algebraic Petri Nets," *TKK Monoistamo; Otaniemi1997*, ISSN 0783 5396, 1997.

[10] N. Zeghib, K. Barkaoui and M. Bettaz , "Contextual ECATNets semantics in terms of conditional rewriting logic," *in Proc. 4th ACS/IEEE Int. Conf. on Computer Systems and Application*, UAE, 2006, pp. 936– 943.

Noura Boudiaf is an Associate Professor of Computer Science at the Department of Computer Science of the University of Oum El-Bouaghi in Algeria. She received Ph.D. in Computer Science from the University of Constantine in Algeria. Her main areas of interest include Object-oriented Software Engineering, Multi-agents Systems, Petri net Analysis Methods, and Formal Methods.

Kamel Bensaber is a Software Consultant in Eurofunk Kappacher GmbH Company, Salzburg, Austria. He received the B.S. degree in Computer Science at Constantine University and respectively the M.S and Ph.D. in Language Sciences (Oral Communication) from Stendhal University, Grenoble, France in 1998. His main areas of interest include Object-oriented modelling combined with meta-data approach, Complex Information Systems, Prototyping, Information retrieval, Digital Libraries, Speech databases.