CEUR-WS.org/Vol-2907/paper9.pdf

An Enhanced Rewriting Logic Based Semantics for High-Level Petri nets

Ammar Boucherit[1], Kamel Barkaoui[2], and Osman Hasan[3]

[1] Computer Science Department, University of El-Oued, Algeria
ammar-boucherit@univ-eloued.dz
[2] SYS: Équipe Systèmes sûrs, Cedric/CNAM, France
kamel.barkaoui@cnam.fr
[3] SEECS, National University of Sciences and Technology (NUST), Pakistan
osman.hasan@seecs.nust.edu.pk

Abstract. Petri nets and their numerous extensions (or subclasses) are one of the popular traditional formalisms for the specification and verification of concurrent systems. Furthermore, due to the expressivity of rewriting logic, Maude and its associated analysis tools have been adopted in many recent works for executing and analyzing Petri nets.
In this paper, we first present the existing semantics for the standard Petri nets. Then, we demonstrate the usefulness and the expressivity of the new enhanced semantics —that encodes the sets of tokens in a place by their cardinality — for such type of Petri nets. Thereafter, we show that such semantics can naturally express different variants of high-level Petri nets such as Petri nets with inhibitor arcs, variable arc weights, and coloured Petri nets. The distinguishing feature of this semantics is that it facilitates the checking of behavioral properties related to boundedness and liveness via the Maude LTL model checker.

Keywords: Maude · Petri nets · Rewriting logic.

1 Introduction

Petri nets were first introduced in 1962 [23], and are still one of the most effective, useful and reliable formalisms in practice for modeling, and analyzing discrete and dynamic systems [12]. The success of Petri nets is mainly due to some of their distinguishing features, such as the graphical notation, simple semantics, and the rich mathematical theory [24]. However, the analysis and simulation of complex systems modeled by Petri nets require user-friendly formal tools [11, 9, 30].
Maude [7] is an executable programming language that has been adopted in many recent works for specifying, executing and analyzing Petri nets [3, 25, 14, 16], thanks to the expressivity of rewriting logic [20] and its associated analysis

Copyright © 2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

tools. In particular, its support for module hierarchies, meta-level programming and parameterization allows designers to conceptually decompose the specification into parts (modules) [6, 27, 10]. This results practically in modularity, reusability, reduction of the development effort and time, as well as facilitating specification and verification of large and complex Petri net-based models.
In this context, there exists a rewriting logic-based semantics for Petri nets [20, 28] that uses the names of places to represent tokens and it has always been accepted as a simple way for representing token instances. However, this representation causes a difficulty in manipulating arcs multiplicities and the number of tokens within places. Moreover, this also gives the impression as if the marking change of a Petri net is a change made on tokens rather than a change of states of places. Moreover, the existing semantics makes the representation of priority or testing boundedness of a Petri net quite difficult even with some additional equations and major modifications. Therefore, having a counter may facilitate the manipulation of arcs multiplicities and avoid having to continuously count the number of tokens in a place.
In this paper, we propose an enhanced semantics based on a simple, intuitive and structured notation for Petri nets in rewriting logic. While tokens are repeated and not counted in the existing semantics, the number of tokens in a place is recorded in the proposed semantics. This leads to the easiness of specification of Petri net models unambiguously. It also accommodates inhibitor arcs, variable arc weights, and thereafter facilitates — since it uses an intentional and reduced notation — the simulation and analysis of both basic and high-level Petri nets by using Maude formal analysis tools.
The rest of the paper is organized as follows: Section 2 presents some necessary concept about Petri nets and rewriting logics. In Section 3, the existing rewriting logic semantics for Petri nets is described. Section 4 presents our proposed semantics, followed by some experimental results in Section 5 to show the relevance of the proposed enhancements. Finally, Section 6 concludes the paper with some perspectives for future work.

2 Preliminaries

2.1 Petri Nets

In this section, we briefly recall basic notions about Petri nets: both structural and dynamical parts of a Petri nets with the usual associated graphical representation and some of their relevant extensions related to our work. For this purpose, consider the following definition.

Definition 1 *A Petri net is defined as a 5-tuple* $\mathcal{N} = (P, T, Pre, Post, M_0)$ *where:*

- P*: a finite set of places with* $|P| = n$*,*
- T*: a finite set of transitions with* $|T| = m$ *and* $P \cap T = \emptyset$*,*
- Pre*:* $P \times T \longrightarrow \mathbb{N}$*, is the* Pre *incidence function*
- $Post$*:* $T \times P \longrightarrow \mathbb{N}$*, is the* $Post$ *incidence function*

– *M_0: $P \rightarrow \mathbb{N}$, function mapping a number of tokens to each $p \in P$ (so-called initial marking)*

The structural aspect of a Petri net (see, Figure 1(a)) is extremely simple and it basically consists of a set of places, transitions and directed arcs that connect places with transitions. The places (depicted by circles) represent system states or conditions and may hold a non-negative integer number of tokens, which are represented by black dots. The transitions (depicted by bars) represent the system state changes or events that may occur. The directed arcs (signified by arrows) define pre-conditions and/or post-conditions for each transition in terms of places.
The set of places linked to a transition t with arcs starting from places are called *input places* (or preset) is noted by $\bullet t = \{p|Pre(p,t) > 0\}$, and the set of places linked to a transition t with arcs starting from the transition are called *output places* (or postset) is noted by $t\bullet = \{p|Post(t,p) > 0\}$. A transition that does not have any input place is called a *source transition* and a transition that does not have any output place is called a *sink transition.* An arc may have a given integer value that defines its weight, i.e., the number of tokens that will be consumed or produced following this arc.
Note that $MPre$ (resp, $MPost$) is an $n \times m$ matrix that is commonly called the *pre* (resp, *post*) incidence matrix, where $MPre[i,j] = Pre(p_i,t_j)$ (resp, $MPost[i,j] = Post(t_j,p_i)$). In addition, $Pre(\bullet,t_j)$ and $Post(t_j,\bullet)$ denote all input and output arcs of a transition t_j with their weights, i.e., the j-th columns of the MPre and MPost matrices, respectively.
A function $M_f : P \times T \longrightarrow \mathbb{N}$ is called a marking function that assigns to each place of the Petri net a non-negative integer number of tokens. With the implicit ordering $(p_1,p_2,...,p_n)$ on the set of places P, a marking M — that describes the Petri net state — can be represented as a column vector M where $M[i]$ (i'th row) contains $M_f(p_i)$. Therefore, the dynamical aspect of a Petri net, starting from an arbitrary initial marking, is defined by the evolution of its marking, i.e., the sequence of markings generated by the set of the fired transitions. A transition t may be fired at a marking M if : $M[i] \geq Pre(p_i,t)$ $\{\forall\ p_i \in \bullet t\}$
Thereafter, firing of a transition t at marking M removes a number of tokens equals $Pre(p_i,t)$ from each of its input places and puts another number of tokens equals $Post(t,p_i)$ to each place of its output places. In other words, whenever a Petri net is marked with M, its new marking M' after firing a transition t, is defined as follows:

$$M' = M - Pre(\bullet,t) + Post(t,\bullet)$$

A Petri net is said to be "*k-bounded*" if the number of tokens in all of its places does not exceed a finite number k for any marking reachable from M_0. A Petri net is called safe if $k = 1$, i.e., for all reachable markings, no place of the Petri net has more than one token.
Petri nets have been extended by many researchers. Some of the relevant extensions related to our work are described below:

– One of the most widely used extensions is the introduction of inhibitor arcs [1]. Such arc is denoted graphically by an arc with a small circle attached to a transition (see, Figure 1(b)). The inhibitor arc reverses the logic of the enabling and firing rules, i.e., a transition will only be enabled if the input place contains less tokens than the weight of the inhibitor arc. In addition, tokens in the input place of inhibitor arcs are not consumed after firing. This extension notably increases the expressiveness by allowing a 'test to zero' and thus makes Petri nets as powerful as counter automata and Turing machines [18].
– The second extension, i.e., a Petri net with variable (dynamic) arc weights is proposed as a powerful modeling, analysis and simulation tool for complex dynamical systems [4]. In these Petri nets, the weight of an arc is a variable (dynamic) that is specified by the actual number of tokens in a place and thus depends on the current marking.
– The third extension is called coloured Petri nets (CPN) [15], which preserve the useful properties of standard Petri nets and enrich them with complex data structures (see, Figure 1(c)). The main characteristic that makes CPN models more compact and practical lies in the token definition. In the simple case, tokens have a simple data value (called token color) attached to them. Usually places contain tokens of one type that is called color set of the place.

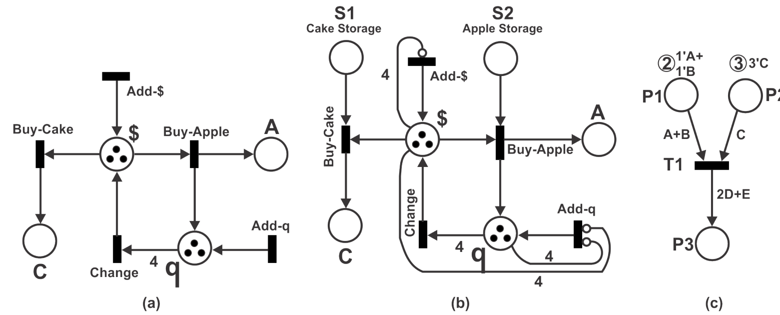

Fig. 1. Example of Petri nets

2.2 Rewriting Logic and Maude

Rewriting logic [19, 20] defines a simple, expressive and efficient logic for reasoning about concurrency and specifying concurrent systems. In fact, rewriting logic extends equational algebraic specifications with rewrite rules to deal with changes in concurrent systems [21]. However, Maude [7] is the executable specification language based on rewriting logic that we have used to implement our current prototypes. A concurrent system is specified by means of a rewrite theory as $\mathcal{R} = (\Sigma, \text{E}, \text{L}, \text{R})$. Its static structure is described by the equational

theory (Σ, E), whereas its dynamic behavior is described by the set of labelled conditional rewrite rules (L, R). As rewriting-based logic, systems evolution is emulated by matching and replacing parts of the system state according to the rewrite rules. Specifically, rewriting logic has proven to be a well-suited unifying framework for Petri nets [28] and a wide range of other concurrency models. Practically, a Maude specification consists principally of two types of modules:

– ***Functional modules*** (enclosed within `fmod ... endfm`) are used to describe the static aspect of the system. Such modules are based on membership equational logic to define data types (*sorts* and *subsorts*), operations on them (by means of equational theories) and constructor operators (that can have some equational attributes such as commutativity or associativity). The equational rewriting serves as a replacement of equals by equals from left to right, until the equivalent value is fully evaluated.
– ***System modules*** (enclosed within `mod ...endm`) are very general rewrite theories (`mod ...endm`) that may have equations (or import functional modules) in addition to rewrite rules, which can be conditional in order to define the dynamic part of the studied system.

Maude rewrite system offers a large number of powerful tools such as: an explicit-state LTL model checker, reachability tool and an inductive theorem prover to facilitate formally verifying systems.

3 Existing Rewriting Logic Semantics for Petri nets

The existing semantics for Petri nets is based on a number of previous works such as [29, 2, 17, 22]. The rewriting logic semantics for standard Petri nets was proposed in [20] and was then generalized for a wide range of Petri nets in [28]. To briefly illustrate this semantics, we consider the Petri net, given in Figure 1(a), describing the behaviour of the classic vending machine [5] that is used to buy cakes (place C in the Petri net) and apples (place A); a cake costs 1 dollar (place $) and an apple 3 quarters (place q). For simplicity, we assume that this machine only accepts dollars and it permits changing four quarters into a dollar.

3.1 Structural Aspects

The basic sorts needed to describe a Petri net are: ***place*** and ***marking***. A marking on a Petri net is viewed as a multiset[4] over its set of places, representing (a snap-shot of) a Petri net state and denoting the available tokens (resources) in each place. These elements are defined as sorts with a subsort relation as follows:

```
sorts Place Marking .    subsort Place < Marking .
```

[4] Mathematically, a multiset (or bag, or mset) is a set-like, unordered collection of elements in which elements are allowed to be repeated.

Then, the Petri net place's names are declared as operators as follows:

```
ops C A $ q :-> Place .
```

Thereafter, the current state (marking) of a Petri net can be defined with a finite multiset union operator as follows:

```
op null : -> Marking .
op _ _ : Marking Marking -> Marking [assoc comm id: null] .
```

According to this declaration, we can see that a marking (or partial marking) can be represented by an element of the finite multiset sort `Marking` and the union of two markings is a new marking. The empty marking is represented by the constant `null`. In addition, the attribute `"assoc"` (resp. `"comm"`) is used to declare that the operator is associative (resp. commutative). Finally, the initial marking may also be declared as an operator and then defined by an equation:

```
op initial :-> Marking .    eq initial = $ $ $ q q q .
```

In such a declaration, only the names of the places that hold tokens appear in the initial state with an occurrence equal to the number of tokens they contain.

3.2 Behavioral Aspects

The evolution of a Petri net is related to the transitions firing. For that, the specification of a transition consists of two multisets (termsets), where the first multiset (marking representing the transition *pre-set*) may be replaced with the second one (marking representing the transition *post-set*). Therefore, each transition t is described by a labelled rewrite rule with the following syntax:

```
rl [⟨Transition-Label⟩] : ⟨Termset-1⟩ => ⟨Termset-2⟩ .
```

In such a rule, the `Termset-1` (resp. `Termset-2`) contains only the set of input (resp. output) places of the corresponding transition t, where a place name is repeated in the left-hand or right-hand side of the rule, as many times as the weight of the arc linking the place to the transition t.

In that case, a transition specified by a rewrite rule can take place if its left-hand side (`Termset-1`) matches. Then, the sub-marking is transformed into the right-hand side (`Termset-2`) of such rule. The process of rewriting will start with a rewrite rule that matches its left-hand side in the initial marking, and stop if no rule matches anymore.

In addition, a sink (resp. source) transition is a special case and its corresponding rewrite rule remains the same and a variable (`M`, for example) of sort `Marking` is added to the `Termset-2` (resp. `Termset-1`) and also, used to completely replace the `Termset-1` (resp. `Termset-2`) since it has not a *pre-set* (resp. *post-set*).

Finally, by following the pre-described steps, we give the complete specification of the Petri net, presented in Figure 1 (a), as follows:

```
fmod PETRI-NET-SIGNATURE is
```

```
mod VENDING-MACHINE is
```

```
  sorts Place Marking .
  subsorts Place < Marking .
  op null : -> Marking .
  ops C A $ q : -> Place .
  op __ : Marking Marking -> Marking
              [assoc comm id: null] .
  op initial : -> Marking .
  eq initial = $ $ $ q q q .
endfm
```

```
  protecting PETRI-NET-SIGNATURE .
  var M : Marking .

  rl [add-$] : M => M $ .
  rl [add-q] : M => M q .
  rl [buy-C] : $ => C .
  rl [buy-A] : $ => A q .
  rl [change] : q q q q => $ .
endm
```

As we can see, in this specification, the static part (the signature) of the machine is given in a functional module `PETRI-NET-SIGNATURE`. This module has been imported in the system module `VENDING-MACHINE`, in which we add one rule for each transition to complete describing the dynamic part of the vending machine. Notice that the two rules `"[add-$]"` and `"[add-q]"` are used to describe the ***source*** transitions in the Petri net, and because of that, the rewriting in the module `VENDING-MACHINE` does not terminate.

4 Proposed Petri net Semantics Based on Rewriting Logic

In the semantics we propose here, our focus is on how to overcome the above-mentioned drawbacks in order to make the specification of Petri nets more natural. It has been found that, according to [26], there are various mathematical presentations of multisets. The one used in the existing semantics is "sequential", which represents a multiset as a sequence in which the multiplicity of an element equals the number of times the element occurs in the sequence. By contrast, we propose to use an alternative coherent style for presenting the multiset of the marking of a Petri net. More precisely, a multiset M can also be viewed as a set of tuples *(p,x)*, where p is the sequence identifier (element) and x is a function from the set P (set on which M is defined) to the set of non-negative integers, sending to each element p its multiplicity. For illustrating this further, we will use the Petri net given in Figure 1(a) in the subsequent sections.

4.1 Structural Aspects

Starting from the fact that places and tokens are two passive and distinguishable components in a Petri net and, therefore, each new proposed semantics for Petri nets has to clearly define and distinguish between these two primitive concepts. Thus, we propose to represent a token k that resides in place p as a tuple *(p, x)* (may also be called pair)[5] where p is a place identifier (label) belongs to P (set of places) and x is a variable ranging over non-negative integers that represents the number of occurrences of token k in place p.
Of course, the use of tuple-based notation may seem to be no more than a simple modification of the original representation. However, we argue that the use of

[5] A pair can only have two values — neither less nor more —. However, a tuple, has almost no semantic limitation on the number of values.

a counting-based notation is very beneficial. Intuitively, this notation facilitates the counting of tokens, drops the ambiguity between places and tokens and considerably enhances the description of long Petri net marking (very compact). In addition, this notation will help developers to naturally specify high level Petri nets.
Practically, a tuple is defined by enclosing two items in angle brackets, separated by a comma `<_,_>`. The first item is used to define a place by its name (sort : `PlaceName`) followed by the number of tokens[6] it holds (sort : `Int`)[7].
On the other hand, the marking[8] is consequently defined as a set of tuples. Mathematically, the elements of a set have no order among them; hence, tuples in a marking do not have any particular order. Thus, the basic sort `Marking` is used to define the marking of a Petri net. In addition, the operator `"__"` is also used to allow combining (union) two or more tuples and then produce a new set of tuples. The result may be a subset of tuples related to one transition (*pre-set* and *post-set*) or the whole set of tuples describing the global state of the Petri net (marking). The empty marking is represented by the constant `"null"`. Moreover, the attribute `"assoc"` (resp. `"comm"`) is used to declare that the operator is associative (resp. commutative). Therefore, the corresponding new signature is given as follows:

```
sorts PlaceName Place Marking .
subsort Place < Marking .
ops C A $ q :-> Placename .
op <_,_> : PlaceName Int -> Place [ctor] .
op null : -> Marking .
op _ _ : Marking Marking -> Marking [assoc comm id: null].
op initial :-> Marking .
eq initial = < $,3 > < q,3 > < C,0 > < A,0 > .
```

The first advantage of the proposed specification is that the test for the number of tokens in a place as well as the specification of inhibitor arcs is now possible. Moreover, the declaration of the initial marking in our proposal describes the overall Petri net, including all its places, and is not limited to the places holding tokens. Therefore, this declaration gives a clear snap-shot of the initial marking of the system and one can thereby know all the names of the Petri net places along with the tokens they hold.

4.2 Behavioral Aspects

Theoretically, the global state of a Petri net is generally represented by a marking M. Thereafter, when firing a transition t, the change in such a state of the

[6] For the sake of simplicity and since tokens are indistinguishable in basic Petri nets, we were only interested in their number in a place.
[7] We use the sort `Int` since the subtraction is not defined with sort `NAT`.
[8] It is noticed that the Petri net marking in the proposed semantics has the commutative monoidal structure since the set of tuples *(p,x)* is equipped with an associative binary operation (__) and an identity element.

Petri net occurs at the level of the set of input and output places by removing tokens from the former and adding tokens to the latter. Such evolution can be naturally specified in the rewriting logic by rewrite rules. In general, these rules are conditional and a rewrite rule has to be defined for each transition as follows:

```
crl [⟨Transition-Label⟩] : ⟨LHS⟩ => ⟨RHS⟩ if Cond .
```

During execution, Maude uses the whole set of tuples given in initial marking and whenever a subset (sub-marking) matches the `LHS` then that part can be replaced by `RHS` if the enabling condition[9] is verified. In fact, the `LHS` (resp. `RHS`) describes the state of both the input and output places before (resp. after) firing the transition t, and are defined as follows:

$$\texttt{LHS}, \texttt{RHS} = \text{set of tuples}(p, x), \text{ where } p \in \{\bullet t \cup t\bullet\}$$

In addition and according to the number of input places, a condition — defined by the expression `Cond` — can be either a single equation or a conjunction of equations using an associative binary conjunction connective such as: `/\` or `"and"`. Consequently, a rewrite rule could be somewhat larger, yet it is considerably more clear — in terms of presentation— than the one in the existing semantics. We now present the complete specification of the previous Petri net according to the proposed specification.

```
fmod NEW-PETRI-NET-SIGNATURE-1 is
  protecting INT .
  sorts PlaceName Place Marking .
  subsort Place < Marking .
  ops C A $ q : -> PlaceName .
  op <_,_> : PlaceName Int -> Place [ctor] .
  op __ : Marking Marking -> Marking [ctor assoc comm id: null] .
  ops null initial : -> Marking .
  eq initial = < $,3 > < q,3 > < C,0 > < A,0 > .
endfm
mod NEW-VENDING-MACHINE-1 is
  inc NEW-PETRI-NET-SIGNATURE-1 .
  vars x y z : Int .
  rl [add-$] : < $,x >  => < $,x + 1 > .
  rl [add-q] : < q,x >  => < q,x + 1 > .
  crl [buy-c] : < $,x > < C,y > => < $,x - 1 > < C,y + 1 > if (x > 0) .
  crl [buy-a] : < $,x > < A,y > < q,z > => < $,x - 1 > < A,y + 1 > < q,z + 1 >
                                                                  if (x > 0) .
  crl [change] : < $,x > < q,z > => < $,x + 1 > < q,z - 4 > if (z >= 4) .
endm
```

[9] The condition is not needed in the case of a source transition since it is unconditionally enabled and therefore, the corresponding rewrite rule will be unconditional.

5 Comparison with the Existing Semantics

In this section, we present the effectiveness of the proposed rewriting logic-based semantics for Petri nets. For that, some experimental comparison results have been presented in order to assess the advantages and evaluate of the performance of the proposed specification compared to the existing ones.

5.1 Compact Representation of Petri nets Marking

As the proposed semantics use tuples to describe a marking, there is a clear difference between a place and its tokens. Therefore, a marking will be very clear and the number of tokens in each place is shown without the need to manually count them. To show that, let's explore the behavior of our machine from the given initial marking. For that, we use the command `"rewrite"`[10] (abbreviated `"rew"`). For instance, consider the same command `"rew [5]"` (resp. `"rew [100]"`) to see and compare the presentation of the resulted marking — for both existing and enhanced semantics — after 5 (resp. 100) times of rule applications from the initial marking, which has 3 dollars and 3 quarters.

Existing Semantics	**Enhanced Semantics**

`Maude> rewrite [5] in VENDING-MACHINE : initial .`
`rewrites: 6 in 541371105387ms cpu (0ms real)`
`(0 rewrites/second)`
`result Marking: $ $ $ $ $ $ q q q`
	`Maude> rewrite [5] in NEW-VENDING-MACHINE-1 :`
	`initial .`
	`rewrites: 23 in 541371105385ms`
	`cpu (0ms real) (0 rewrites/second)`
	`result Marking: < q,3 > < $,0 > < C,2 > < A,2 >`

`Maude> rewrite [100] in VENDING-MACHINE : initial .`
`rewrites: 101 in 541371105435ms cpu (0ms real)`
`(0 rewrites/second)`
`result Marking: $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $`
`$ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $`
`$ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ q`
	`Maude> rewrite [100] in NEW-VENDING-MACHINE-1 :`
	`initial .`
	`rewrites: 481 in 541371105397ms cpu (0ms real)`
	`(0 rewrites/second)`
	`result Marking: < q,2 > < $,0 > < C,18 > < A,2 >`

As we can see, the execution results of the command `"rewrite"` in the existing semantics may become impractical and non-understandable, especially in the case of a marking with a considerable number of tokens.

5.2 Describing Inhibitor Arcs

In order to examine the ability of describing inhibitor arcs, we consider the second machine, presented in Figure 1(b), in which we imply users to not insert more than 4 dollars (resp. 4 quarters) in the coins box before buying something (resp. making change). However, the returned quarters after buying an apple are added without control.

In addition, we have added two places (`S1` and `S2`) to describe the vending machine capacity in delivering items (cake and apple). The capacity of the vending machine is assumed to be 50 items for both cakes and apples. This capacity is declared in the initial state and after delivering all items, the machine has to be refilled (this process is not considered here).

[10] It is necessary to determine the upper bound allowed for the number of rule applications when using such commands. Otherwise, infinity is assumed.

The Existing Semantics
In the existing semantics, the number of tokens in a place cannot be naturally obtained and, in order to get it, one can use an additional operator (enclosing the entire marking) such as the usual system-grabbing operator as follows:

```
op {_} : Marking -> PN .
```

Thereafter, one could similarly define a second operator as follows:

```
op number : PN Place -> Int .
```

that is simply defined as:

```
eq number({M M'},M) = 1 + number({M'},M) .
eq number({M'},M) = 0 [owise] .
```

With this new operator, one could describe transitions with inhibitor arcs. For instance, the rules `add-$` and `add-q` could now be written as follows:

```
crl [add-$] : {M} => {M $} if number({M},$) < 4 .
crl [add-q] :  {M} => {M q} if number({M},$) < 4 /\ number({M},q) < 4 .
```

The Proposed Semantics
The specification of Petri nets with inhibitor arcs is made naturally due to the proposed representation of places. Therefore, the specification of the machine presented in Figure 1(b) is given as follows:

```
fmod NEW-PETRI-NET-SIGNATURE-2 is
  protecting INT .
  sorts PlaceName Place Marking .       subsort Place < Marking .
  ops C A $ q S1 S2 : -> PlaceName .
  op <_,_> : PlaceName Int -> Place [ctor] .
  op __ : Marking Marking -> Marking [ctor assoc comm id: null] .
  ops null initial : -> Marking .
  eq initial = <$,3> <q,3> <C,0> <A,0> <S1,50> <S2,50> .
endfm

mod NEW-VENDING-MACHINE-2 is
  inc NEW-PETRI-NET-SIGNATURE-2 .
  vars x y z t : Int .
  crl [add-$] : < $,x > => < $,x + 1 > if (x < 4) .
  crl [add-q] : < $,x > < q,z > => < $,x > < q,z + 1 > if (z < 4) and (x < 4) .
  crl [buy-c] : < $,x > < C,y > < S1,z > => < $,x - 1 > < C,y + 1 > < S1,z - 1 >
                                                   if (x >= 1) and (z >= 1) .
  crl [buy-a] : < $,x > < A,y > < q,z > < S2,t > => < $,x - 1 > < A,y + 1 >
                          < q,z + 1 > < S2,t - 1 > if (x >= 1) and (t >= 1) .
  crl [change] : < $,x > < q,z > => < $,x + 1 > < q,z - 4 > if (z >= 4) .
endm
```

In this specification, the transitions `add-$` and `add-q` have been described with conditional rewrite rules with the necessary test for the number limit.

5.3 K-Bounded testing

The boundedness testing is possible since our system is terminating. In addition, we already know that the place $ is bounded with 17 because the studied system has a limited number (50) for both cakes and apples. Therefore, we would like to demonstrate the ability to check the correctness of such characteristic. To do so, one can use the reachability tool or Maude LTL model checker[11] to look for the existence of a marking where the place $ holds 18 tokens by using the following command.

```
Maude> search in NEW-VENDING-MACHINE-2 : initial =>* M < $,18 > .
No solution.
states: 714867  rewrites: 15490529 in 4281906278ms cpu (105312ms real)
(3 rewrites/second)
```

In this result, we can see that the Maude reachability tool did not find a solution, which means that the place $ will not have 18 tokens during the system evolution[12]. So, such place may be bounded and in order to confirm that, we have to check if there are some markings where the number of tokens in such place is equal to 17.

```
Maude> search in NEW-VENDING-MACHINE-2 : initial =>! M < $,17 > .
Solution 1 (state 714857)
states: 714864  rewrites: 15490426 in 4281906278ms cpu (109750ms real)
(3 rewrites/second)
M --> < q,1 > < C,50 > < A,50 > < S1,0 > < S2,0 >
Solution 2 (state 714863)
states: 714866  rewrites: 15490496 in 4281906278ms cpu (109750ms real)
(3 rewrites/second)
M --> < q,2 > < C,50 > < A,50 > < S1,0 > < S2,0 >
Solution 3 (state 714866)
states: 714867  rewrites: 15490529 in 4281906278ms cpu (109750ms real)
(3 rewrites/second)
M --> < q,0 > < C,50 > < A,50 > < S1,0 > < S2,0 >
No more solutions.
states: 714867  rewrites: 15490529 in 4281906278ms cpu (109750ms real)
(3 rewrites/second)
```

[11] Maude LTL model checker cannot be used for the existing semantics without the additional operator given in Section 5.2. In addition, it would be unsuitable to use the reachability tool with the existing semantic since the number of tokens have to be repeated as many as number of tokens to be searched.

[12] In this case, we have used the parameter " =>! " for *search* command in order to minimize (reduce) the set of solutions to the canonical final states, i.e., states that cannot be further rewritten. Otherwise, parameter " =>* " can be used (see [5] for more details).

As we can see, the place $ contains — and never holds more than — 17 tokens, and thus it is bounded with 17.
On the other hand, the LTL model-checker can also be used to check the previous property by defining the following modules:

```
mod VENDING-MACHINE-2-PREDS is
  protecting NEW-VENDING-MACHINE-2 .
  including SATISFACTION .
  subsort Marking < State .
  op Bound(_,_) : PlaceName Int -> Prop .
  var M : Marking .  var P : PlaceName .  vars x y : Int .
  ************** PLACE BOUNDEDNESS PROPERTY ***************
  ceq < P,x > M |= Bound(P,y) = true if x <= y .
endm
mod VENDING-MACHINE-2-CHECK is
  inc VENDING-MACHINE-2-PREDS .
  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .
  op Place-Bounded(_,_) : PlaceName Int -> Prop .
  op PN-Bounded(_,_) : PlaceName Int -> Prop .
  var P : PlaceName .
  var x : Int .
  eq Place-Bounded(P,x) = [](Bound(P,x)) .
endm
```

In the following, we give the boundedness verification results of places $ and q.

```
Maude> reduce in VENDING-MACHINE-2-CHECK : modelCheck(initial, Bounded($,17)) .
rewrites: 16920270 in 48121469397ms cpu (99937ms real) (~ rewrites/second)
result Bool: true
```

5.4 Petri nets With Variable Arc Weights

To better understand the notion of Petri nets with "variable arc weights", we consider the example, given in Figure 2(a), that is inspired by [4].

In this example, the transitions T1 and T2 are used to control and maintain the marking of the place P1 at a desired level. In this example, the marking shall not be less than 3. Therefore, the transition T1 will be fired when the marking of the place P1 is greater than the desired level and a token is added to the place P2. Thereafter, T2 will be enabled and then fired to remove the excess of tokens from the place P1 through the weight (M(P1)−3) of the arc (P1, T2). The

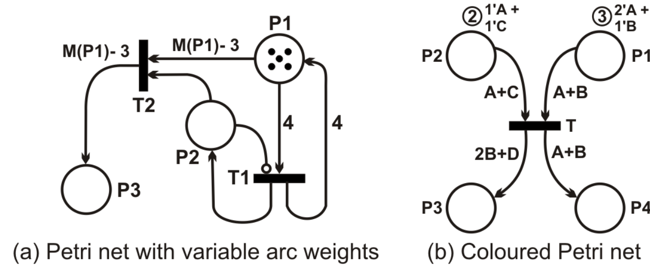

Fig. 2. Examples of Petri nets extensions

following code represents the initial marking and the rewrite rules describing the transitions T1 and T2 according to the proposed semantics as given in Section 4.

```
eq initial = <P1,5> <P2,0> <P3,0> .
crl [T1] : < P1,x > < P2,0 > => < P1,x > < P2,1 > if (x >= 4) .
crl [T2] : < P1,x > < P2,1 > < P3,y > => < P1,3 > < P2,0 > < P3,y + x - 3 >
                                                              if (x > 3) .
```

We should note that it is not possible to describe a Petri net with variable arc weights (i.e., produce or remove a variable number of tokens) in the existing semantics neither naturally nor with the additional operator, given in Section 5.2. Thus, for this purpose, one has to explore other alternatives (strategies, loops ...etc).

5.5 Coloured Petri nets

To show the extensibility of the proposed semantics for describing colored Petri nets, we consider the model shown in Figure 2(b). The given coloured Petri net has four places P1, P2, P3 and P4 and four colors of tokens that are: Color1 (A), Color2 (B), Color3 (C) and Color4 (D).
Of course, the specification of coloured petri nets is also based on tuples as seen in basic Petri nets, possibly defined with two generic parameters (sorts). The first parameter specifies the name of the place and the second determines the set of colors within this place. Such a specification is given as follows:

```
op <_,_> : PlaceName ColorSet -> Place .
```

For that, the set of color within a place need the following declaration:

```
sorts ColorId Color ColorSet .
subsort Color<ColorSet .
```

```
op _{_} : ColorId Int -> Color .
op _,_ : ColorSet ColorSet -> ColorSet .
```

Thereafter, the list of places and color names of the studied coloured Petri nets must be given. In our example, such declaration is:

```
ops P1 P2 P3 P4 : -> PlaceName .
ops A B C D : -> ColorId .
```

The complete specification of the structural aspects of our coloured Petri nets is given as follows:

```
fmod GENERIC-CPN-SIGNATURE is
  protecting INT .
  sorts ColorId Color ColorSet PlaceName Place Marking .
  subsort Color < ColorSet .
  subsort Place < Marking .
  ops A B C D : -> ColorId .
  ops P1 P2 P3 P4 : -> PlaceName .
  op _{_} : ColorId Int -> Color .
  op _ _ : ColorSet ColorSet -> ColorSet [comm]  .
  op <_,_> : PlaceName ColorSet -> Place .
  ops null initial : -> Marking .
  op _ _ : Marking Marking -> Marking [ctor assoc comm id: null] .
  eq initial = < P1,A{2} B{1} > < P2,A{1} C{1} > < P3,B{0} D{0} > < P4,
              A{0} B{0} > .
endfm
```

However, such generic specification for a coloured Petri net may be inconvenient since it does not preclude the user from constructing erroneous tuples (composed of place's name and colors) that do not belong to the coloured Petri net in study. Therefore, we propose a seconde well-formed signature for a coloured Petri net in which the places of the Petri net have to be decomposed into sets which share the same set of token colors. For instance, the given CPN has three color sets so that, P1 and P4 belong to the same color set (`"PlaceNameset1"`), P2 belongs to the color set (`"PlaceNameset2"`) and P3 belongs to the color set (`"PlaceNameset3"`).
Subsequently, we define each type of place (according to the colors of its tokens) in a separate operator in order to obtain an unambiguous presentation. The corresponding new signature is given as follows:

```
fmod CPN-SIGNATURE is
  protecting INT .
```

```
  sorts PlaceNameset1 PlaceNameset2 PlaceNameset3 .
  sorts Place Marking .
  subsort Place < Marking .
  ops P1 P4 : -> PlaceNameset1 .
  op P2 : -> PlaceNameset2 .
  op P3 : -> PlaceNameset3 .
  op <_,A(_) B(_)> : PlaceNameset1 Int Int -> Place .
  op <_,A(_) C(_)> : PlaceNameset2 Int Int -> Place .
  op <_,B(_) D(_)> : PlaceNameset3 Int Int -> Place .
  ops null initial : -> Marking .
  op _ _ : Marking Marking -> Marking [ctor assoc comm id: null] .
  eq initial = < P1,A(2) B(1) > < P2,A(1) C(1) > < P3,B(0) D(0) > < P4,
              A(0) B(0) > .
endfm
```

According to that, the behavioral aspects can be given as follows:

```
mod CPN is
inc CPN-SIGNATURE .
vars x1 x2 x3 y1 y2 y3 z t : Int .
crl [T] : < P1,A(x1) B(y1) > < P2,A(x2) C(z) > < P3,B(y2) D(t) > < P4,A(x3)
          B(y3) > => < P1,A(x1 - 1) B(y1 - 1) > < P2,A(x2 - 1) C(z - 1) >
          < P3,B(y2 + 2) D(t + 1) > < P4,A(x3 + 1) B(y3 + 1) > if ((x1 >= 1)
          and (y1 >= 1) and (x2 >= 1) and (z >= 1)) .
endm
```

Let us now use the command `rewrite` to explore the behavior of this CPN. The result of execution of this specification is given as follows:

```
rewrite [1] in COLOURED-PN : initial .
rewrites: 17 in 541555185225ms cpu (0ms real) (0 rewrites/second)
result Marking: < P1,A(1) B(0) > < P4,A(1) B(1) > < P2,A(0) C(0) > < P3,B(2)
                 D(1) >
```

6 Conclusion

In this paper, a new enhanced semantics, based on a highly structured notation for Petri nets in rewriting logic, is introduced and compared to the existing one. The main advantage of both semantics is that the basic paradigm of Petri net computations (true concurrency involving several non-conflicting transitions) is preserved. However, the proposed semantics is structurally and behaviourally straightforward and clear, i.e., while tokens are repeated (and not counted) in

the existing flat notation, the number of tokens in a place is counted in the enhanced semantics. In addition, the new semantics facilitates the simulation and analysis of both basic and high-level Petri nets and deals unambiguously with variable arc weights, inhibitor arcs, and boundedness testing.
For the future, we intend to exploit the benefits of the proposed semantics for the verification of models described in terms of extended high-level Petri nets as parametric and recursive Petri nets [16, 8, 13]. We also aim to incorporate the new semantics into existing Petri net tools as plug-ins to offer an automated way for the conversion of high-level Petri nets to rewriting logic and therefore facilitating their analysis.

Acknowledgement

Foremost, we would like to express our sincere gratitude to Prof. Peter Csaba Ölveczky who provided insights and expertise that greatly helped this research. His criticisms were quite constructive and helped to get this work in the current form. Then, this work would not have been possible without the financial support (PhD scholarship) N°:034/PNE/ENS/Spain/13-14 received through the Algerian ministry of higher education and scientific research.
Finally, we greatly appreciate the support received through the collaborative work undertaken with Laura M Castro at the MADS (Models and Applications of Distributed Systems) research group, A Coruña University, Spain.

References

1. Tilak Agerwala. Complete model for representing the coordination of asynchronous processes. Technical report, Johns Hopkins Univ., Baltimore, Md.(USA), 1974.
2. Andrea Asperti. A logic for concurrency. Technical report, Technical report, Dipartimento di Informatica, Universit a di Pisa, 1987.
3. Kamel Barkaoui, Hanifa Boucheneb, and Awatef Hicheur. Modelling and analysis of time-constrained flexible workflows with time recursive ecatnets. In *International Workshop on Web Services and Formal Methods*, pages 19–36. Springer, 2008.
4. T Benarbia, K Labadi, A Omari, and JP Barbot. Balancing dynamic bike-sharing systems: A petri nets with variable arc weights based approach. In *Control, Decision and Information Technologies (CoDIT), 2013 International Conference on*, pages 112–117. IEEE, 2013.
5. Manuel Clavel, Francisco Durán, Steven Eker, P Lincoln, N Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.3), 2007. *URL: http://maude. cs. uiuc. edu/maude2-manual*, 2007.
6. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Metalevel computation in Maude. *Electronic Notes in Theoretical Computer Science*, 15:331–352, 1998.

7. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All about Maude — a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.
8. Nicolas David. *Discrete Parameters in Petri Nets.(Réseaux de Petri à Paramètres Discrets).* PhD thesis, University of Nantes, France, 2017.
9. Nicholas J Dingle, William J Knottenbelt, and Tamas Suto. Pipe2: a tool for the performance evaluation of generalised stochastic petri nets. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):34–39, 2009.
10. Francisco Durán and José Meseguer. Maude's module algebra. *Science of Computer Programming*, 66(2):125–153, 2007.
11. Gerald C Gannod and Sunil Gupta. An automated tool for analyzing petri nets using spin. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 404–407. IEEE, 2001.
12. Alessandro Giua and Manuel Silva. Modeling, analysis and control of discrete event systems: a petri net perspective. *IFAC-PapersOnLine*, 50(1):1772–1783, 2017.
13. Serge Haddad and Denis Poitrenaud. Recursive petri nets. *Acta Informatica*, 44(7):463–508, 2007.
14. Xudong He, Reng Zeng, Su Liu, Zhuo Sun, and Kyungmin Bae. A term rewriting approach to analyze high level petri nets. In *Theoretical Aspects of Software Engineering (TASE), 2016 10th International Symposium on*, pages 109–112. IEEE, 2016.
15. Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*, volume 1. Springer Science & Business Media, 2013.
16. Ahmed Kheldoun, Kamel Barkaoui, and Malika Ioualalen. Formal verification of complex business processes based on high-level petri nets. *Information Sciences*, 385:39–54, 2017.
17. Narciso Martí-Oliet and José Meseguer. From petri nets to linear logic. In *Category Theory and Computer Science*, pages 313–340. Springer, 1989.
18. Diego C Martinez, Maria Laura Cobo, and Guillermo Ricardo Simari. A petri net model of argumentation dynamics. In *International Conference on Scalable Uncertainty Management*, pages 237–250. Springer, 2014.
19. José Meseguer. *A logical theory of concurrent objects*, volume 25. ACM, 1990.
20. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science*, 96(1):73–155, 1992.
21. José Meseguer. Membership algebra as a logical framework for equational specification. In *Recent Trends in Algebraic Development Techniques*, pages 18–61. Springer, 1997.
22. José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and computation*, 88(2):105–155, 1990.

23. Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Darmstadt University of Technology, Germany, 1962.
24. Wolfgang Reisig. *Elements of distributed algorithms: modeling and analysis with Petri nets*. Springer Science & Business Media, 2013.
25. Alexander Schulz. Model checking of reconfigurable Petri nets. *arXiv preprint arXiv:1409.8404*, 2014.
26. D Singh, AM Ibrahim, T Yohanna, and JN Singh. An overview of the applications of multisets. *Novi Sad Journal of Mathematics*, 37(2):73–92, 2007.
27. Ścibor Sobieski and Bartosz Zieliński. Modularisation in Maude of parametrized rbac for row level access control. In *East European Conference on Advances in Databases and Information Systems*, pages 401–414. Springer, 2011.
28. Mark-Oliver Stehr, José Meseguer, and Peter Csaba Ölveczky. Rewriting logic as a unifying framework for Petri nets. In *Unifying Petri Nets*, pages 250–303. Springer, 2001.
29. Glynn Winskel. Categories of models for concurrency. In *International Conference on Concurrency*, pages 246–267. Springer, 1984.
30. Dianxiang Xu. A tool for automated test code generation from high-level petri nets. *Applications and Theory of Petri Nets*, pages 308–317, 2011.