Technical report: A prototype for reconfigurable GSPNs

Samir Tigane
LINFI Laboratory,
Computer Sciences Department
Biskra University, Algeria
Email: s.tigane@univ-biskra.dz

Laid Kahloul
LINFI Laboratory,
Computer Sciences Department
Biskra University, Algeria
Email: kahloul2006@yahoo.fr

Samir Bourekkache
LINFI Laboratory,
Computer Sciences Department
Biskra University, Algeria
Email: s.bourekkache@gmail.com

I. Introduction

We aim to develop a tool that deals with the reconfiguration in generalized stochastic Petri nets (GSPNs) [4]. GSPNs represent an extension of Petri nets (PNs) which allows to verify qualitative properties (reachability, liveness, deadlock-freedom, etc.) as well as quantitative properties (system throughput, system reliability, machine utilization, etc). Thus, GSPNs provide a suitable formal tool for the performance evaluation of systems. However, GSPNs have a rigid structure and are not able to specify intuitively reconfigurable systems. We aim to propose a new extension of GSPNs suitable for the formal modeling and verification of reconfigurable systems, based on the improved net rewriting systems (INRSs) formalism [3]. To this end, we have developed a tool that has as inputs a GSPN that models an initial configuration and a set of rules each of which models a possible change in the structure of the reconfigurable net. Then, our tool applies these rules to reconfigurable net and computes an isomorphic Markov chain to reconfigurable net that describes its behavior. Once the latter is completely constructed, the tool can compute the quantitative properties such as: throughput of a transition, mean number of tokens in a place, the mean sojourn time at a marking, etc.

II. Description

In the following, we give a basic description of developed classes, their important fields and methods.

A. `Place`

This class allows to create an instance of a place.

Fields

```
public String IP;
//This is a place label.
public int MP;
//This is a place marking.
```

Methods

```
public Place(String p,int m);
```

This method is used to create an instance of a place, where `p` is a place label and `m` is a place marking.

```
public boolean equals(Place p);
```

This method is used to compare two places, where `p` is the place to be compared with. It returns True if two places are having same label and marking.

B. `Transition`

This class allows to create an instance of a transition.

Fields

```
public String IT;
//This is a transition label.
public Transition.Type type;
//This is a transition type: timed or
   immediate.
public double rate;
//This is a transition rate/weight.
```

Methods

```
public Transition(String l, Transition.Type
   t, double r);
```

This method is used to create an instance of a transition, where `l` is a transition label, `t` is a transition type : timed or immediate, and `r` is a transition rate/weight.

```
public boolean equals(Transition t);
```

This method is used to compare two transitions, where `t` is a transition to be compared with. This method returns True if two transitions are having same label, rate and type.

C. `Rule`

This class allows to create an instance of a rule.

Fields

```
public GSPN L;
//This is a left-hand side.
public GSPN R;
//This is a right-hand side.
public ArrayList<GSPN> NAC;
//This is a list of negative application
   conditions.
public String[] IL;
```

```
//This is an input interface of left-hand
    side.
public String[] OL;
//This is an output interface of left-hand
    side.
public String[] IR;
//This is an input interface of right-hand
    side.
public String[] OR;
//This is an output interface of right-hand
    side.
public double weight;
//This is a weight application of rule.
public String ID;
//This is an identifier of rule.
public Place[] activatingMarking;
//This is an activator marking that controls
    rule application.
```

Methods

```
public Rule(String ID, GSPN L, GSPN R,
    ArrayList<GSPN> NAC, String[] IL,
    String[] OL, String[] IR, String[] OR,
    double weight, Place[] AM);
```

This method is used to create an instance of a rule, where `ID` is an identifier of rule, `L` is a left-hand side, `R` is a right-hand side, `NAC` is a list of negative application conditions, `IL` is an input interface of left-hand side, `OL` is an output interface of left-hand side, `IR` is an input interface of right-hand side, `OR` is an output interface of right-hand side, `weight` is a weight application of rule, and `AM` is an activator marking that controls rule application.

```
public boolean isInIL(String n);
```

This method is used to check whether a node belongs to input nodes of left-hand side of a rule, where `n` is a node label. It returns True if node `n` belongs to input nodes of left-hand side of a rule.

```
public boolean isInOL(String n);
```

This method is used to check whether a node belongs to output nodes of left-hand side of a rule, where `n` is a node label. It returns True if node `n` belongs to output nodes of left-hand side of a rule. Analogously to methods `isInIR` and `isInOR` with respect to right-hand side are defined.

D. `GSPN`

This class allows to (i) create an instance of a GSPN from a PNML file describing its structure and (ii) compute its reachability graph. As well, it allows to compute quantitative properties, such as: mean number of tokens, token probability density, throughput, etc.

Methods

```
public GSPN(Place[] setOfP, Transition[]
    setOfT, int[][] pr, int[][] po);
```

This method is used to create an instance of a GSPN, where `setOfP` is a set of places, `setOfT` is a set of transitions, `pr` is presets of transitions and `po` is postsets of transitions.

```
public GSPN(String xFile);
```

This method is used to create an instance of a GSPN, where `xFile` is the path of PNML file containing the description of a GSPN created by a third-party.

```
public int getNumberOfTangibleStates();
```

This method is used to get the number of tangible states in reachability graph.

```
public int getNumberOfStates();
```

This method is used to get the number of states in reachability graph.

```
public boolean isFireable(String t, Place[]
    M);
```

This method is used to check whether a transition `t` is fireable at a marking `M`.

```
public void fire(String t);
```

This method is used to fire a transition `t` at current marking of a GSPN.

```
public Place[] getMarkingAfterFiring(String
    t, Place[] M);
```

This method is used to compute obtained marking after firing a transition `t` at marking `M`.

```
public JSONArray getReachabilityGraph();
```

This method is used to get a reachability graph as a JSON object.

```
public JSONArray getMarkingsDistProba();
```

This method is used to get a marking distribution probability.

```
public JSONArray getMeanNumberOfTokens();
```

This method is used to get mean number of tokens.

```
public JSONArray getTokenProbabilityDensity();
```

This method is used to get token probability density.

```
public JSONArray
    getProbabilitiesFiringTransition();
```

This method is used to get firing transition probability density.

```
public JSONArray getThroughputOfTransitions();
```

This method is used to get throughput of transitions.

```
public JSONArray getMeanSojournTime();
```

This method is used to get mean sojourn time.

E. `RecGSPN`

This class allows to create an instance of a reconfigurable generalized stochastic Petri net describing its dynamic structure. As well, it allows to apply rules to reconfigurable nets and compute their quantitative properties, such as: mean number of tokens, token probability density, throughput, etc. **Methods**

```
public RecGSPN(GSPN G0, ArrayList<Rule>
    setOfRules);
```

This method is used to create an instance of a RecGSPN, where `G0` is a initial configuration, and `setOfRules` is a list of rules.

```
public boolean isApplicable(Rule r, GSPN G,
    Place[] M);
```

This method is used to check whether a rule `r` is applicable to a GSPN `G` at a marking `M`.

```
public GSPN getGSPNAfterApplayingRule(Rule r,
    GSPN G, Place[] M);
```

This method is used to compute obtained GSPN after applying a rule `r` to a GSPN `G` at marking `M`.

```
public int getNumberOfTangibleStates();
```

This method is used to get the number of tangible states in reachability graph.

```
public JSONArray getReachabilityGraph();
```

This method is used to get reachability graph.

```
public JSONArray getMarkingsDistProba();
```

This method is used to get marking distribution probability.

```
public JSONArray getMeanNumberOfTokens();
```

This method is used to get mean number of tokens.

```
public String[][]
    getMeanNumberOfTokensAsMatrix();
```

This method is used to get mean number of tokens as matrix.

```
public JSONArray getTokenProbabilityDensity();
```

This method is used to get token probability density.

```
public String[][]
    getTokenProbabilityDensityAsMatrix();
```

This method is used to get token probability density as matrix.

```
public JSONArray
    getProbabilitiesFiringTransition();
```

This method is used to get firing transition probability density.

```
public String[][] getTransitionsStat();
```

This method is used to get firing transition probability density and throughputs as matrix.

```
public JSONArray getThroughputOfTransitions();
```

This method is used to get throughput of transitions.

```
public JSONArray getMeanSojournTime();
```

This method is used to get mean sojourn time.

III. Demonstration

In this section, we demonstrate how to model/verify (quantitatively) a reconfigurable net. First, the user can use a third-party tool to create a GSPN that models an initial configuration of a reconfigurable net. Note that the GSPN must be described by the standard format PNML as used by **PIPE** tool [1]. As well, left- and right-hand sides of each rule are GSPNs that can be created by a third-party tool.

Let us consider a reconfigurable system composed of machine M_1 permanently active and machine M_2 which is activated when the number of raw materials in the buffer (having ten spaces) exceeds five. The initial configuration containing M_1 is highlighted in Fig. 1. The interpretation of places and transitions is given as follows.

1) *as* (resp. *rm*): Its marking represents the number of free spaces (resp. raw materials) in the buffer.
2) m_1 (resp. m_1'): A token in m_1 (resp. m_1') means that machine M_1 has begun (resp. has finished) processing.
3) m_1f: A token in m_1f means that machine M_1 is idle.
4) *ra*: Raw material is loaded in the central buffer.
5) ld_1: M_1 loads an item from the buffer.
6) m_1p: Machine M_1 is processing.
7) uld_1: M_1 unloads a product.

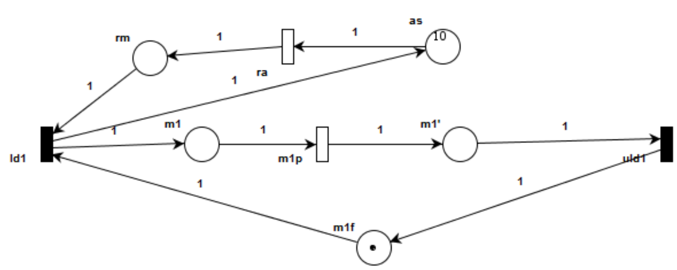


Fig. 1: Initial configuration.

Once the number of raw materials in the buffer exceeds five, machine M_2 is activated and the system switches to its second configuration. This reconfiguration is modeled by a rule r_1, where its left- and right-hand sides are shown in Figs. 2

and 3, its input nodes are ({ld1},{ld1,ld2}), and its output nodes are ({uld1},{uld1,uld2}).

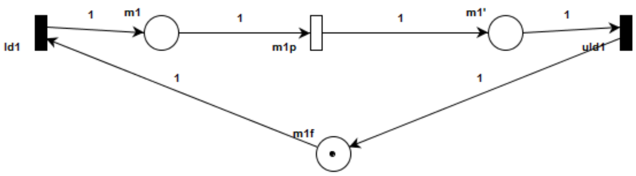


Fig. 2: Left-hand side of rule r_1 and right-hand side of rule r_2.

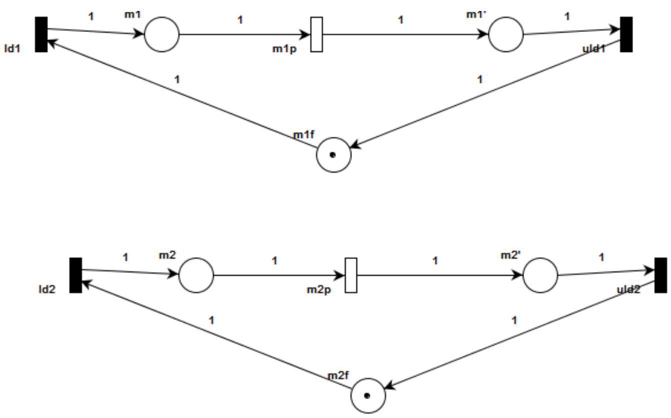


Fig. 3: Right-hand side of rule r_1 and left-hand side of rule r_2.

Consider the following code.

```
1  import java.util.ArrayList;
2    public class Main {
3      public static void main(String[]
          args) {
4        GSPN G= new GSPN("C_0.xml"),
5          L1=new GSPN("L_1.xml"),
6          R1=new GSPN("R_1.xml"),
7          L2=new GSPN("L_2.xml"),
8          R2=new GSPN("R_2.xml");
9
10       String[]IL1={"ld1"};
11       String[]OL1={"uld1"};
12       String[]IR1={"ld1","ld2"};
13       String[]OR1={"uld1","uld2"};
14       Place[] am1=new Place[5];
15       am1[0]=new Place("as",0);
16       am1[1]=new Place("rm",6);
17       am1[2]=new Place("m1",0);
18       am1[3]=new Place("m1'",0);
19       am1[4]=new Place("m1f",1);
20       ArrayList<GSPN> NAC1=new
             ArrayList();
21       NAC2.add(L2);
22
23       Rule r1 = new Rule("r1", L1, R1,
             NAC1, IL1, OL1, IR1, OR1, 2,
             am1);
24
25       String[]IL2={"ld1","ld2"};
26       String[]OL2={"uld1","uld2"};
27       String[]IR2={"ld1"};
28       String[]OR2={"uld1"};
29       Place[] am2=new Place[8];
30       am2[0]=new Place("as",10);
31       am2[1]=new Place("rm",0);
32       am2[2]=new Place("m1",0);
33       am2[3]=new Place("m1'",0);
34       am2[4]=new Place("m1f",1);
35       am2[5]=new Place("m2",0);
36       am2[6]=new Place("m2'",0);
37       am2[7]=new Place("m2f",1);
38       Rule r2 = new Rule("r2", L2, R2,
             null, IL2, OL2, IR2, OR2, 2,
             am2);
39
40       ArrayList<Rule> lr= new
             ArrayList<>();
41       lr.add(r1);
42       lr.add(r2);
43
44       RecGSPN rgspn = new RecGSPN(G,lr);
45       System.out.println
46         (rgspn.getNumberOfGSPNs());
47       System.out.println
48         (rgspn.getNumberOfStates());
49       System.out.println
50         (rgspn.getMeanNumberOfTokens());
51       System.out.println
52 (rgspn.getProbabilitiesFiringTransition());
53       System.out.println
54 (rgspn.getThroughputOfTransitions());
55     }
56   }
```

L_1 (left-hand side) and R_1 (right-hand side) of r_1 are instantiated at Lines (5) and (6), respectively. Input and output nodes of L_1 are defined as arrays of String at Lines (10) and (11). As well, input and output nodes of R_1 are defined at Lines (12) and (13). Aforementioned, rule r_1 is applicable to initial configuration when the number of raw materials in the buffer exceeds five. The activator marking of rule r_1 is defined as an array of Place at Lines (14)–(19). The instruction at Line (16) states that the marking of place *rm* (its marking models the number of raw materials in the buffer) is six. Finally, rule r_1 is instantiated at Line (23), where its set of negative application conditions [2] contains L_2 (Fig. 3). Indeed, r_1 is not applicable if machine M_2 is already activated.

Once the buffer is empty, the system switches to its initial configuration. This switching is modeled by rule r_2, where its left- and right hand sides are depicted in Figs. 3 and 2, respectively.

L_2 (left-hand side) and R_2 (right-hand side) of r_2 are instantiated at Lines (7) and (8), respectively. Input and output nodes of L_2 are defined as arrays of String at Lines (25) and (26). As well, input and output nodes of R_2 are defined at Lines (27) and (28). Rule r_2 is applicable to second configuration when the buffer is empty. The activator marking of rule r_2 is defined as an array of Place at Lines (29)–(37). The instruction at Line (30) states that the marking of place *as* (its marking models the number of available spaces in the buffer) is ten. Finally, rule r_2 is instantiated at Line (38), where its set of negative application conditions is empty.

Rule r_1 and r_2 are inserted into list `lr` at Lines (41) and (42) to create a set of rules.

The reconfigurable net is instantiated at Line (44), where its set of rules is `lr` and its initial configuration is `G` instantiated at Line (4).

Finally, we can compute different parameters, such as the number of obtained GSPNs after applying the set of rules to the initial configuration, the number of states in the isomorphic Markov chain, the mean number of tokens at each place, etc.

The result of execution of the code in above is the following.

```
2//The number of obtained GSPNs after
    applying the set of rules to the initial
    configuration.
96//The number of states in the isomorphic
    Markov chain

//The mean number of tokens at each place
[{"values":8.163158068311944,"id":"as"},
{"values":0.8025364509473432,"id":"m1"},
{"values":0.0,"id":"m1'"},
{"values":0.19746354905265673,"id":"m1f"},
{"values":1.8368419316880582,"id":"rm"},
{"values":0.1929087837708813,"id":"m2"},
{"values":0.0,"id":"m2'"},
{"values":0.012781191714198437,"id":"m2f"}]

//Firing transition probability
[{"values":0.37306141543396915,"id":"m1p"},
{"values":0.5602739943546192,"id":"ra"},
{"values":0.06666459021141184,"id":"m2p"}]

//Transition throughput
[{"values":0.8025364509473432,"id":"m1p"},
{"values":0.9954452347188375,"id":"ra"},
{"values":0.1929087837708813,"id":"m2p"}]
```

IV. Other examples

A. Example 1: A Reconfigurable Manufacturing System (RMS)

In this section, we illustrate how to apply our proposed method on an example of an RMS. First, we present the RMS and give a description of its behavior. Second, we apply a set of rules in order to reconfigure the initial RMS. In this case study, we consider an RMS that is composed of one robot with a capacity of one space, a buffer for each machine with capacity of 3 spaces, an exit zone used to hold products that have already finished their processing, 2 machines that operate in parallel and cooperate to product a variety of products, and a special buffer space used to recover the potential deadlocks.

The Robot has a random access to any part in the buffer or in the exit zone, and it can move the products from/to the exit zone, the machine M_1, the machine M_2, the buffer buf_1, and the buffer buf_2.

A product *Pct* can be processed by the two machines, or by only one machine. The system products four types of products *A*, *B*, *C*, and *D*. Product *A* (resp. *B*) is processed only by machine M_1 (resp. M_2). Product *C* (resp. *D*) is processed firstly

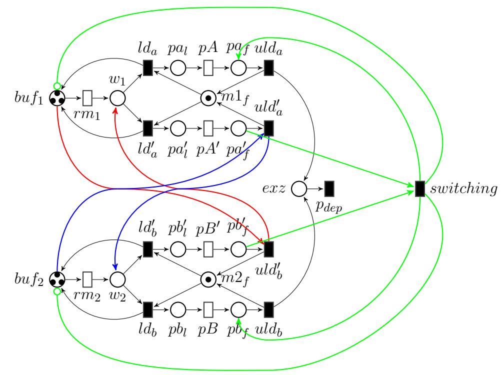


Fig. 4: GSPN model of the RMS.

by machine M_1 (resp. M_2) and finally by machine M_2 (resp. M_1).

When an item *I* arrives at the RMS, it enters buf_1 (resp. buf_2) to be processed by machine M_1 (resp. M_2) , and it waits until the downstream machine to be free (*downstream machine means the next machine that the product will visit*). If the downstream machine M_i is idle and there is no item *W* in buffer buf_i or in M_j waiting for M_i, the robot loads *I* into M_i. When a machine M_i becomes free, and if there is an item *W* in the buffer waiting for M_i, then the robot loads *W* into M_i. When a machine M_i finishes the first step of processing of an intermediate product *IP*, and if there is a free buffer space in buf_j then the robot puts *IP* in buf_j, otherwise machine M_i maintains holding *IP*. When a buffer space becomes free in buf_i (resp. buf_j), and if machine M_j (resp. M_i) is holding an item *W* which is waiting for M_i (resp. M_j) to be free, then the robot loads *W* into the buffer buf_i. When a machine M_i finishes the final processing step of a product *Pct*, the robot unloads *Pct* into the *exit zone*.

The described RMS is modeled by GSPN depicted in Fig. 4. The meanings of places and transitions of this GSPN, are given as follows :

- Places
  - buf_1 (resp. buf_2): The number of tokens, inside this place, models the number of free buffer spaces in bf_1 (resp. bf_2).
  - $m1_f$ (resp. $m2_f$): A token in $m1_f$ (resp. $m2_f$) means that the machine M_1 (resp. M_2) is idle.
  - w_1 (resp. w_2): The number of tokens in w_1 (resp. w_2) models the number of items waiting, at the buffer buf_1 (resp. buf_2), to be processed by machine M_1 (resp. M_2).
  - pa_l/pa'_l (resp. pb_l/pb'_l): A token in pa_l/pa'_l (resp. pb_l/pb'_l) models that an item is loaded into machine M_1 (resp.M_2).
  - pa_f (resp. pb_f): A token in pa_f (resp. pb_f) models that machine M_1 (resp. M_2) has finished producing product *Pct*.
  - pa'_f (resp. pb'_f): A token in pa'_f (resp. pb'_f) models that machine M_1 (resp. M_2) has finished producing

  an intermediate product *IP*.
  - *exz*: The number of tokens in this place is the number of finished products at the exit zone, in the current time.
- Transitions :
  - rm_1 (resp. $rm2$): Raw material is loaded in buffer buf_1 (resp. buf_2).
  - ld_a/ld'_a (resp. ld_b/ld'_b): The robot loads an item into machine M_1 (resp. M_2).
  - pA (resp. pB): Machine M_1 (resp. M_2) has finished processing a product Pct.
  - pA' (resp. pB'): Machine M_1 (resp. M_2) has finished processing an intermediate product IP.
  - uld_a (resp. uld_b): The robot unloads final product Pct from machine M_1 (resp. M_2) to the exit zone.
  - uld'_a (resp. uld'_b): The robot unloads intermediate product IP from machine M_1 (resp. M_2) to buffer buf_2 (resp. buf_2).
  - p_{dep}: Final product Pct exits the RMS.
  - *switching*: The robot switches an intermediate product IP held by machine M_1 and an intermediate product IP' held by machine M_2.

The robot uses a special buffer space to switch intermediate products which are held by M_1 and M_2, hence when an item has, finally, finished its processing by two machines it can be unloaded to the *exit zone*, yielding its place to another waiting item in the buffer. This switching is performed when a deadlock occurs. The deadlock situation is represented by this marking $(M(buf_1)=0, M(buf_2)=0, M(pa'_f)=1, M(pb'_f)=1)$. This means that the number of free spaces in buf_1 and buf_2 is zero, and machine M_i is holding an intermediate product waiting for M_j.

We aim now to reconfigure the RMS presented above by adding a new machine M_3. This machine cooperates with machine M_2 to perform extra treatments on intermediate products IP treated already by M_2 (i.e., machine M_2 has finished the first treatment on IP). This reconfiguration is illustrated and modeled by GSPN shown in Fig. 8. This GSPN model is obtained by applying three rules on GSPN presented in Fig. 4, these rules are described in the following.

Machine M_3 operates when machine M_2 has finished processing an intermediate product, this event is modeled by firing of transition pB' and deposing a token in place pb'_f. Once machines M_3 and M_2 have finished their treatment on the intermediate product, this later can be loaded in buffer buf_1. This modification is reflected in GSPN model shown in Fig. 8 by applying three rules, as follows :

1) r_1: Substitute place pb'_f in GSPN model G_0 in Fig. 4 by (OSM) net block [5] presented in Fig. 5(R_1). A subnet of the resulting graph G_1 is shown in Fig. 5.
2) r_2: Substitute immediate transition pB'' in G_1 by (CMG) net block [5] illustrated in Fig. 6(R_2). The resulting graph is called G_2.
3) r_3: Substitute subnet $sn = \{uld''_b, pb''_f, uld'_b\}$ in G_2 by a ST net block [5], where $T = T1 = \{uld'_b\}$. The resulting graph is shown in Fig. 8.

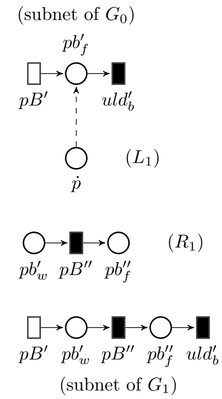


Fig. 5: Applying r_1 on G_0.

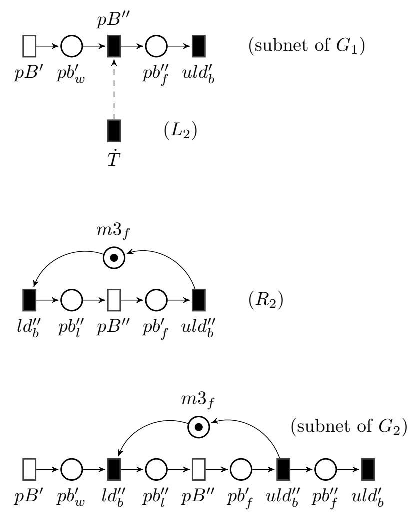


Fig. 6: Applying r_2 on G_1.

The obtained GSPN model G_3 after applying three rules r_1, r_2 and r_3 is shown in Fig. 8. The meaning of the new places and transitions is described as follows.

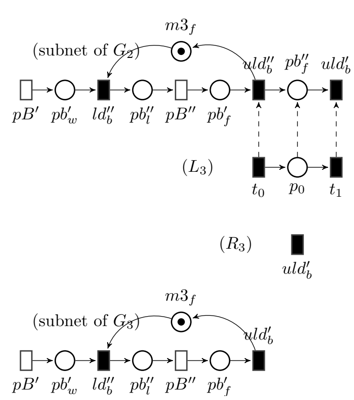


Fig. 7: Applying r_3 on G_2.

- Places
  - pb'_w: A token in pb'_w models that machine M_2 has finished the first processing on immediate product IP.
  - pb''_l: A token in pb''_l models that an immediate product IP is loaded into machine M_3.
  - $m3_f$: A token in $m3_f$ models that machine M_3 is idle.
- Transitions
  - ld''_b: The robot loads an intermediate product IP into machine M_3.
  - pB'': Machines M_2 and M_3 have finished processing an intermediate product IP.

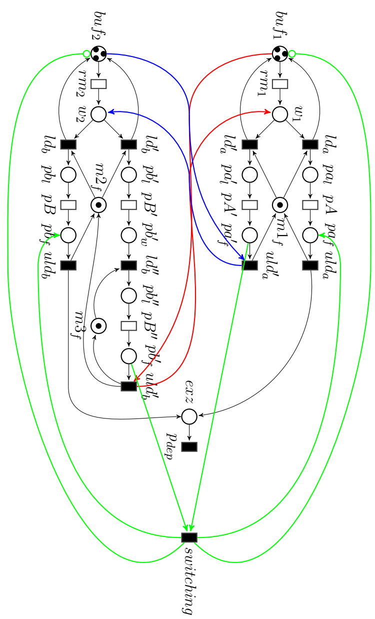


Fig. 8: Final RMS model after applying rules r_1, r_2 and r_3.

B. Example 2: Data Center

In this subsection, we illustrate the application of the proposed formalism on a data center case study. Firstly, we give a description of the structure and the behavior of the data center. Secondly, we show how to apply a set of rules in order to reconfigure the initial model of the data center and how to evaluate its performance.

In this case study, we consider a data center composed of three servers S_1, S_2, S_3, and two buffers (i) buf_h with capacity of p spaces that receives jobs with high priority and (ii) buf_n with capacity of q spaces that receives jobs with normal priority. Both of buffers are implemented with the policy “Fist In First Out”. Server S_1 treats jobs with high priority, whereas Server S_2 is dedicated to jobs with normal priority. In sake of reducing the power consumption, Server S_3, initially, is standby. It starts working when the number of waiting prioritized (resp. normal) jobs exceeds the threshold S_h (resp. S_n). The system has three configurations and the switching from configuration to another is conducted according to the number of waiting jobs.

The described date center, at its first configuration, is modeled by the GSPN C_0 depicted in Fig. 9.

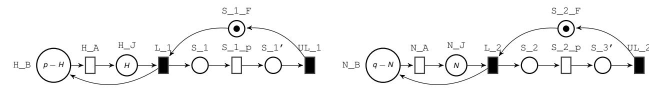


Fig. 9: Configuration C_0 where $H < S_h$ and $N < S_n$.

The description of places and transitions at configuration C_0 model are given in Table I.

Place	Description
H_B	The number of tokens, inside this place, models the number of available spaces in buffer buf_h
H_J	The number of tokens models the number of waiting jobs with high priority
N_B	The number of tokens, inside this place, models the number of available spaces in buffer buf_n
N_J	The number of tokens models the number of waiting jobs with normal priority
S_1	A token in S_1 means that S_1 has begun treating a job
S_2	A token in S_2 means that S_2 has begun treating a job
S_1'	A token in S_1' means that S_1 has finished treating a job
S_2'	A token in S_2' means that S_2 has finished treating a job
S_1_F	A token in S_1_F means that S_1 is idle
S_2_F	A token in S_2_F means that S_2 is idle
Transition	**Description**
H_A	Arrival of a job with high priority
N_A	Arrival of a job with normal priority
L_1	S_1 loads a job from buffer buf_h
L_2	S_2 loads a job from buffer buf_n
S_1_p	S_1 processes a prioritized job
S_2_p	S_2 processes a normal job
UL_1	S_1 unloads a finished job
UL_2	S_2 unloads a finished job

TABLE I: Meanings of places and transitions at configuration C_0.

Once the number of waiting jobs with high priority exceeds the threshold S_h Server S_3 is activated which yields the second configuration. After its activation, Server S_3 joins Server S_1 in treating prioritized jobs. This configurations is illustrated in Fig. 10.

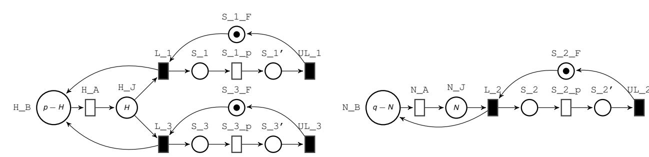


Fig. 10: Configuration C_1 where $S_h \leq H$

The system switch to the third configuration C_2 if the number of waiting jobs with normal priority is bigger than threshold S_n and the number of prioritized jobs is less than S_h. In this configuration C_3, Server S_2 and Server S_3 along together process jobs with normal priority. This configurations is shown in Fig. 11.

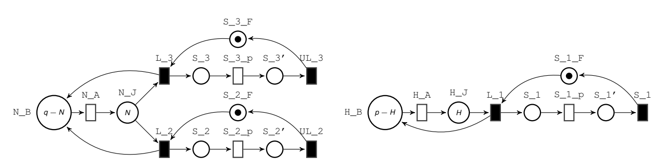


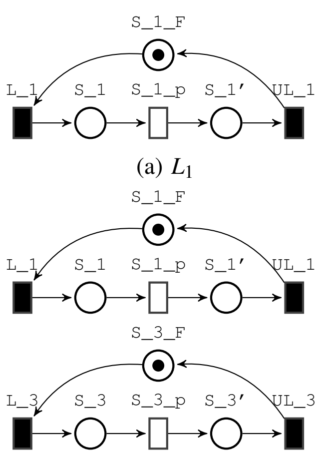
Fig. 11: Configuration C_2 where $S_n \leq N$.

The description of the new places and transitions in configurations C_1 and C_2 are given in Table II.

Place	Description
S_3	A token in S_3 means that S_3 has begun treating a job
S_3′	A token in S_3′ means that S_3 has finished treating a job
S_3_F	A token in S_3_F means that S_3 is idle
Transition	**Description**
L_3	S_3 loads a job
S_3_p	S_3 processes a job
UL_3	S_3 unloads a finished job

TABLE II: Meanings of the new places and transitions at configurations C_1 and C_2.

1) Rewriting rules of the system reconfigurations: In this subsection, we model RecGSPN based reconfiguration of the described system. First, we reconfigure the configuration C_0 to get the second configuration. This reconfiguration is illustrated and modeled by GSPN C_1 shown in Fig. 10. C_1 is obtained by applying rule r_1 on C_0 as follows.



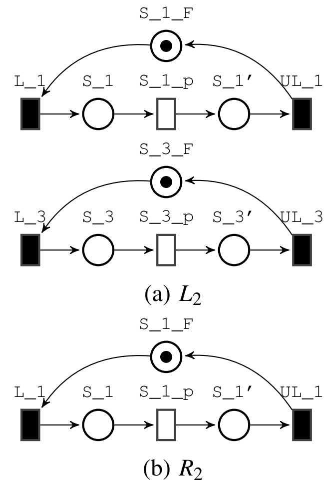
(a) L_1

(b) R_1

Fig. 12: Left-hand side and Right-hand side of r_1

Rule r_1 means to substitute the mapping at C_0 of its left-hand side depicted in Fig. 12a by its right-hand side shown in Fig. 12b. Applying of rule r_1 models the activation of Server S_3.

Once buffer $\mathrm{buf_h}$ is empty, Server S_3 will be deactivated. This reconfiguration is obtained by applying rule r_2 on C_1, the resulting GSPN is C_0.



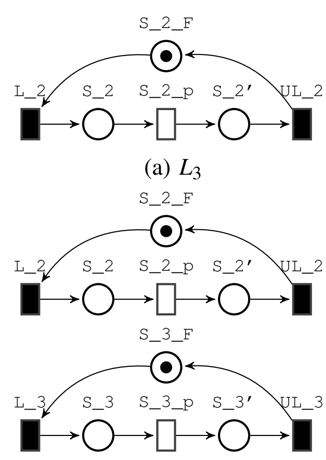
(a) L_2

(b) R_2

Fig. 13: Left-hand side and Right-hand side of r_2

Rule r_2 means to substitute the mapping at C_1 of its left-hand side depicted in Fig. 13a by its right-hand side depicted in Fig. 13b.

Aforementioned, when the number of waiting normal jobs exceeds threshold S_n and Server S_3 is not yet activated (i.e., the number of waiting jobs with high priority is less than threshold S_h), Server S_3 is activated to join Server S_2 in processing normal jobs. This reconfiguration is modeled by GSPN C_2 shown in Fig. 11. C_2 is obtained by applying rule r_3 on C_0 as follows.



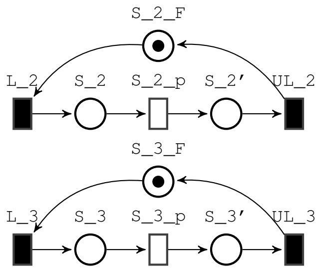
(a) L_3

(b) R_3

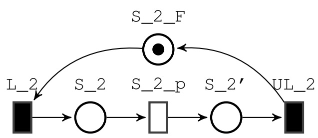
Fig. 14: Left-hand side and Right-hand side of r_3

Rule r_3 means to substitute the mapping at C_0 of its left-hand side depicted in Fig. 14a by its right-hand side shown in Fig. 14b.

Once buffer buf_n is empty, Server S_3 will be deactivated. This reconfiguration is obtained by applying rule r_4 on C_2, the resulting GSPN is C_0.



(a) L_4



(b) R_4

Fig. 15: Left-hand side and Right-hand side of r_4

Rule r_4 means to substitute the mapping at C_2 of its left-hand side depicted in Fig. 15a by its right-hand side depicted in Fig. 15b.

V. CONCLUSION

In this report, we have presented a tool that implements several classes used to model/verify reconfigurability in GSPNs. This tool allows to define a set of rules each of which has left- and right-hand sides. Theses rules are applied to an initial configuration of a reconfigurable net, and therefor an isomorphic Markov chain is computed. Once the latter is completely constructed, we can compute several quantitative properties.

In future version of this tool, we are interested to develop an integrated tool that allows to users to model GSPNs, left- and right-hand sides of rules and to plot charts of different quantitative properties.

REFERENCES

[1] Nicholas J. Dingle, William J. Knottenbelt, and Tamas Suto. PIPE2: A tool for the performance evaluation of generalised stochastic Petri nets. *SIGMETRICS Perform. Eval. Rev.*, 36(4):34–39, 2009.

[2] Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Conflict detection for graph transformation with negative application conditions. In *Graph Transformations*, pages 61–76, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[3] Jun Li, Xianzhong Dai, Zhengda Meng, Jianping Dou, and Xianping Guan. Rapid design and reconfiguration of Petri net models for reconfigurable manufacturing cells with improved net rewriting systems and activity diagrams. *Computers & Industrial Engineering*, 57(4):1431–1451, 2009.

[4] Marco Ajmone Marsan, G. Balbo, Gianni Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[5] S. Tigane, L. Kahloul, and S. Bourekkache. Net rewriting system for GSPN a RMS case study. In *Proc. of ICAASE*, pages 38–45. IEEE, Oct 2016.