Extending GPenSIM for Model Checking on Petri Nets

Reggie Davidrajuh [1], Damian Krenczyk [2]

[1] *Electrical & Computer Engineering,* University of Stavanger, Stavanger, Norway, Reggie.Davidrajuh@uis.no
[2] Faculty of Mechanical Engineering, Silesian University of Technology, Gliwice, Poland, Damian.Krenczyk@polsl.pl

***Abstract* - Cyber-physical systems involve hardware and software that are highly interconnected and complex. Unexpected failures in these systems can cause material damages, cost a lot of money and reputation, and can risk human lives too. The definite way of avoiding unexpected failures is to make a model of the system and to perform model checking on it. Petri nets are a highly effective way of modelling discrete-event systems. General-purpose Petri Net Simulator (GPenSIM) is a tool for modelling, simulation, performance evaluation, and control of discrete-event systems. GPenSIM lacks the automatic model checking facility until now. Firstly, this paper explores the potentials of incorporating the model checking functions to GPenSIM. Secondly, three functionalities are identified and proposed for extending GPenSIM for automatic model checking.**

Keywords - Model checking; Petri Nets; GPenSIM

I. INTRODUCTION

This paper focuses on extending the General-purpose Petri Net Simulator (GPenSIM) for model checking. GPenSIM is developed by the first author of this paper [1]. GPenSIM is freely available for academic and commercial use [2]. GPenSIM runs on MATLAB platform and is being used by some universities around the world, for modeling and simulation large discrete-event systems (e.g., [3-7]). The reasons for the acceptance being the simplicity of learning and using, and its flexibility to incorporate newer functionality, and its ability to control (model-based control) of external hardware and software [3-4].

II. GENERAL-PURPOSE PETRI NET SIMULATOR (GPENSIM)

Implementing a Petri Net model with GPenSIM usually happens via four MATLAB files (M-files) [1]:

1. Petri Net Definition File (PDF) declares the static Petri Net graph. The set of places, the set of transitions, and the set of arcs that make up the static Petri Net are declared in this file.
2. Main Simulation File (MSF) declares the initial dynamics (e.g., initial tokens in the places, firing times of the transitions, firing costs of the transitions) and runs the simulations. When the simulation is completed, the code for plotting the results are also coded in this file.
3. The pre-processor file (COMMON_PRE) is for coding the additional conditions for the enabled transitions to satisfy before they start firing.
4. The post-processor file (COMMON_POST) is for coding any post-firing actions to be performed after firing of the transitions.

With the four simple M-files, many industrial large-scale discrete problems were modeled and solved (e.g., repetitive production processes [8], airport capacity expansion [9], Atlantic fish supply chain in Norway [10], North-sea Oil drilling [11], and process mining [12]).

III. MODEL CHECKING AND PETRI NETS: FORMAL DEFINITIONS

This section is for formal definitions of model checking and Petri Nets. This section is concise. For a detailed study on model checking, the standard textbooks such as ref. [13] is suggested. The textbook ref. [25] is recommended for a detailed study on Petri Nets. A summary of related works is also presented in this section.

A. Model Checking

Model checking exhaustively checks all the states of the system, to see whether any state meets a given property specification [13]. Formally, the problem can be stated as follows: $\boldsymbol{M, s \models \phi}$

Where, $\boldsymbol{\phi}$ is a desired property expressed as a temporal logic formula, and $\boldsymbol{M}$ is a mathematical model (e.g., reachability graph of a Petri Net, in the form of a Labeled Transition System), then model checking is to decide whether there exists a state (or a set of states) $\boldsymbol{s}$ that satisfies the property specification $\boldsymbol{\phi}$.

B. Petri Nets

Petri Nets is a formalism for modeling and simulation of discrete-event systems. Since its inception, Petri Nets have gone through many versions (extensions and subclasses) mainly to incorporate time and to increase its model power [14].

The Place-Transition Petri Net is defined as a four-tuple $PTN = (P, T, A, m_0)$ [15], where,

- P is the set of places, $P = \{p_1, p_2, \ldots, p_{n1}\}$,
- T is the set of transitions, $T = \{t_1, t_2, \ldots, t_{n2}\}$,

- *A* is the set of arcs (from places to transitions and from transitions to places), *A* is a subset of $(P \times T) \cup (T \times P)$ and,
- *m* is the row vector of markings (tokens) on the set of places, $m = [m(p^1), m(p^2), \ldots, m(p^{n1})] \in N^{n1}$, *m0* is the initial marking.

Figure 1 shows a simple Petri Net that consists of three places and one transition, connected by three arcs. The initial marking m_0 on the Petri Net is four, three, and one token, on the places p_1, p_2, and p_3, respectively.

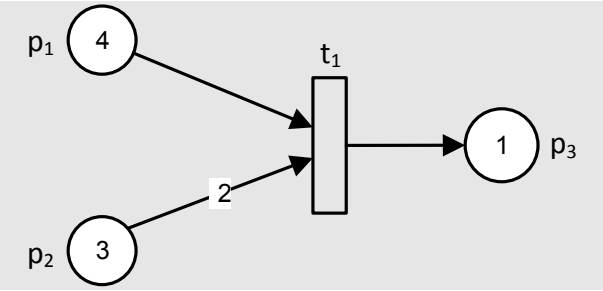

Figure 1. A simple P-T Petri Net.

C. Model Checking on Petri Nets (Related Works)

Literature study presents some works on model checking on Petri Nets. Ref. [26] presents a tool known as ROMEO for model checking on Petri Nets. A Binary Decision Diagrams (BDD) based technique is proposed in [27, 28]. Ref. [29] presents an approach for Colored Petri Nets. Finally, the issue of distributed time in model checking is discussed in [30]. Though there are works describing tools and techniques for model checking on Petri Nets, this work is unique as it is about extension of a specific tool (GPenSIM) for model checking, which is an entirely new attempt.

D. Computational Tree Logic (CTL)

Computational Tree Logic (CTL) is a class of temporal logic that models time in a tree-like structure [16-Ryan & Ruth]. Thus, CTL belongs to the branching-time logic group. Simpler and less expressive (in most cases) is the Linear-Time Logic (LTL) which model time only on a single path. For model checking of Petri Nets, CTL is more suitable, as the model checking is carried out on the reachability graph generated by the Petri Net (explained in the next section). An extension of CTL known as CTL* combines CTL and LTL to take advantages of the strengths of these two logic classes. However, this paper focuses only on CTL for simplicity.

CTL Grammar is defined as follows [16]:

$\phi ::= \square \mid T \mid p \mid - \phi \mid \phi \wedge \phi \mid AN\ \phi \mid EN\ \phi \mid AF\ \phi \mid EF\ \phi \mid AG\ \phi \mid EG\ \phi \mid A(\phi\ U\ \phi) \mid E(\phi\ U\ \phi)$,

where p is an atomic property, and ϕ is a path formula when it is used together with the CTL operators A (for all paths), E (there exists a path), G (globally or always), F (in future or eventually), N (next), and U (until).

TABLE I. THE ELEMENTS OF THE PETRI NET

Set of Transitions	Set of Places
-	**Ready**: the system is in the ready state.
tInsertCoin: the act of inserting the coin.	**Paid**: a user has inserted a valid coin.
tPushRequest: pressing the button for requesting delivery of a drink. **tPushMB**: pressing the button for requesting money back.	**AboutToYield**: (after inserting the coin) the user has already pressed the button "Deliver Product." **AboutToPB**: (after inserting the coin) the user has already pressed the button "Money Back."
-	**Products**: Number of drinks available for sale (1-3).
tGiveProd: the act of delivering a drink. ***tMB***: the act of returning the coin (Money Back) to the user.	**ProductGiven**: A drink is already delivered to the user. **MoneyBack**: The inserted coin is already returned to the user.
tAutoReset1 and *tAutoReset2*: the act of resetting back to the ready state after delivery of a product or money back; these activities will be automatically run (*internal transitions*).	-

IV. MODEL CHECKING ON A PETRI NET

This section explains the process of model checking a Petri Net. For this purpose, this section starts with a simple problem of "a soda vending machine." The model checking process explained in this section is applicable to any large discrete-event system. However, the simple problem is purposely chosen so that the steps involved in the process can be clearly explained. The problem given below is a version of the problem that is presented in some books on model checking (e.g., [13]).

The vending machine has the following properties:

- The vending machine can hold a maximum of three drinks after refilling.
- A user can insert only one coin (e.g., a 10 Norwegian Kroner) to get a drink (product).
- After the inserting the coin, there will be two options: both the push button for "Deliver Product" and the push button for "Money Back" will be highlighted.
- If the user presses the "Deliver Product" button, then the drink (there is only one type of drink) will be delivered.
- On the other hand, if the user presses the "Money Back" button, the inserted coin will be returned to the user.
- When the products are emptied, the push button for requesting a drink will not be highlighted; only the push button for requesting money back will be highlighted.
- When the products are emptied or running low, the process of refilling is not covered in this problem.

A. The Petri Net model

Figure 2 shows the Petri Net model of the soda vending machine. Table-I describes the elements (places and transitions) involved in the model.

B. GenSIM Implementation

As described in the introduction, implementing a Petri Net model with GPenSIM usually requires coding of four files, namely PDF, MSF, the pre-processor file, and the post-processor file. However, the soda vending machine is so simple that it doesn't impose any pre-conditions for firing (to be coded in the pre-processor) and post-firing actions (to be coded in the post-processor). Thus, implementing the soda vending machine requires only two files, the PDF and MSF.

PDF:

```
% PDF (Petri Net Definition File):
vendingMC2_pdf.m
function [png] = vendingMC2_pdf()
% Assigning a name (label) for the module
png.PN_name = 'Pet Net for soda vending machine-
2';

% Declaring the set of places
png.set_of_Ps = {'Ready','Paid', 'AboutToPB',...
'MoneyBack','AboutToYield','ProdGiven','Products'}
;

% Declaring the set of transitions
png.set_of_Ts = {'tInsertCoin','tPushMB','tMB',...
'tAutoReset1','tPushRequest','tGiveProd','tAutoRes
et2'};

% Declaring the set of arcs (connections)
png.set_of_As = {'Ready','tInsertCoin',1,...
    'tInsertCoin','Paid',1, ...
    'Paid','tPushMB',1, 'tPushMB','AboutToPB',1,
...
    'AboutToPB','tMB',1, 'tMB','MoneyBack',1, ...
    'MoneyBack','tAutoReset1',1,
'tAutoReset1','Ready',1, ...
    'Paid','tPushRequest',1,
'Products','tPushRequest',1, ...
    'tPushRequest','AboutToYield',1,
'tPushRequest','Products',1,...
    'AboutToYield','tGiveProd',1,
'Products','tGiveProd',1, ...
    'tGiveProd','ProdGiven',1, ...
    'pProdGiven','tAutoReset2',1,
'tAutoReset2','Ready',1};
```

MSF:

```
% MSF (Main Simulation File): vendingMC.m
% Declare the PDF file
pns = pnstruct('vendingMC2_pdf');
% Assign initial tokens
dyn.m0 = {'Ready',1,'Products',3};
% Combine the static and initial dynamics to
create
%  the initial Petri Net dynamic structure
pni = initialdynamics(pns, dyn);
% create the reachability graph
cotree(pni, 1, 1);
```

The final statement in the MSF is the creation of the reachability graph. Figure 3 shows the reachability graph of the soda vending machine. The reachability graph consists of three blocks: the starter block, the middle block, and the end block.

The reachability graph starts with the starter block on the top; the first state of the reachability graph is the state of three available drinks (in the MSF only three initial tokens as assigned to the place Products) and system readiness. In the middle block, the reachability graph is divided into two columns. The right column shows the process of delivering a drink after the insertion of a coin. The left column shows the process of delivering the money back if the user opts to press the money back button after inserting a coin.

The lower end of the reachability graph shows a group of states when all three products are sold off. In this situation (when the vending machine is empty for products), if a user inserts a coin, he can only press the button for money back (the button for product delivery will not be highlighted), and the money will be paid back eventually.

C. Model Checking the Petri Net Model for the Vending Machine

In this subsection, the Petri Net model will be checked for satisfying some basic properties. The basic properties fall into the group of "fairness" ("something right or good will eventually happen") and "liveness" ("something wrong or bad will never happen").

C1. Model Checking for Fairness

Property-A: If the user has ***inserted a coin*** and has not pressed the "***Request Product***," then the user has the possibility to ask for his ***money back***.

In CTL, the property-A can be formulated as follows:

$$\mathbf{Paid} \wedge \neg(\mathbf{first(tPushRequest)}) \rightarrow \mathbf{AF\ MoneyBack}$$

Property-B: If the user has ***inserted a coin*** and pushed the button for "***Request Product***," then the user will be given a ***product*** in ***the future***.

In CTL, the property-B can be formulated as follows:

$$\mathbf{Paid} \wedge \mathbf{first(tPushRequest)} \rightarrow \mathbf{AF\ ProductGiven}$$

Figure 4 shows that the property-A is satisfied. In figure 4, the states that have "Paid" property (marked by red circles), if they are directed by the "PushMB" (press the button for requesting money back) will eventually end up in states that have the "MoneyBack" property (marked by green solid rectangles).

Figure 4 also shows that the property-B is satisfied too. In figure 4, the states that have "Paid" property (marked by red circles), if they are directed by the "PushRequest" (press

the button for requesting a product) will eventually end up in states that have the "ProductGiven" property (marked by blue dotted rectangles).

C2. Model checking for Liveness

Property-C: If the machine has ***given a product***, then the ***next state*** has to be the ***Ready*** state.

In CTL, the property-C can be formulated as follows:

ProductGiven → AN Ready

Property-D: If the machine has paid the ***money back***, then the ***next state*** has to be the ***Ready*** state.

In CTL, the property-D can be formulated as follows:

MoneyBack → AN Ready

Figure 5 shows that the property-C is also satisfied. In figure 5, the states that have "ProductGiven" property (marked by red ovals) are always immediately followed by the states that have the "Ready" property (marked by green solid rectangles).

Finally, figure 5 also shows that the property-D is satisfied too. In figure 5, the states that have "MoneyBack" property (marked by blue dotted ovals) are always immediately followed by the states that have the "Ready" property (marked by blue dotted rectangles).

V. EXTENDING GPENSIM FOR MODEL CHECKING

In the previous section, the model checking was done by hand; all the states of the reachability graph was checked manually to see whether any state (or set of states) satisfy the property specification. The MATLAB-based software GPenSIM (version 10, as it is now in December 2018) cannot be used for automatic model checking, as it lacks the following three functionalities:

1. Generating complete reachability graph,
2. Facilitating formulation of property specifications using CTL, and
3. Tackling the problem of state explosion.

The following subsections discuss these issues.

A. Making the reachability graph complete

The reachability graph that is shown in figure 3 is complete in the sense that incorporates all the model logic described in the two files MSF and PDF. The old-fashioned algorithm [17] that is powering the reachability graph only needs the model logic (the static Petri Net graph details and the initial tokens) that is described via the files MSF and PDF.

However, when modeling real-life discrete-event system, there will be firing conditions (enabling conditions or guard conditions) and post actions that have to be coded in the processor files. In this case, the reachability graph generated by GPenSIM is incomplete as it ignores the logic coded in the processor files. Hence, the function in GPenSIM for generating the reachability graph has to be extended so that it captures the model logic programmed in the processor files too.

B. Implementing the CTL functions

There are a large number of toolboxes and libraries developed on the MATLAB platform for various purposes. However, when it comes to mathematical logic functions, there are only a few toolboxes (e.g., Structured Array-Based Logic [18-19]) that are available on the MATLAB platform. Moreover, there isn't a single toolbox that implements CTL model operators in MATLAB.

The simple Boolean logic operators such as "conjunction" and "inversion" are already available in MATLAB as inbuilt functions. However, the CTL model operators (such as "for all the paths," "for some of the paths," "globally," "future," "next," and "until") are the ones that are to be implemented. Table-II shows a proposal for the GPenSIM (MATLAB) implementation of the CTL operators. Table-III shows formulating the four properties in GPenSIM.

TABLE II. IMPLEMENTATION OF CTL OPERATORS IN GPENSIM.

CTL operator	CTL Notation	GPenSIM function
For all paths	A	forAll()
For some paths	E	forSome()
Globally	G	Globally()
Eventually (future)	F	Future()
Next	N	Next()
Until	U	Until()

TABLE III. GPENSIM FORMULATION OF THE SPECIFICATIONS OF THE PROPERTIES A-D.

CTL formulation	GPenSIM formulation
Paid ∧ ⁓(first(tPushRequest)) → AF MoneyBack	impl(… and(state({'Paid'}), … not(first('tPushRequest'))), … forAll(Future({'MoneyBack'})) …);
Paid ∧ first(tPushRequest) → AF ProductGiven	impl(… and(state({'Paid'}), … first('tPushRequest')), … forAll(Future({'ProductGiven'})) …);
ProductGiven → AN Ready	impl(… state({'ProductGiven'}), … forAll(Next({'Ready'})) …);
MoneyBack → AN Ready	impl(… state({'MoneyBack'}), … forAll(Next({'Ready'})) …);

C. The Problem of State Explosion

If the reachability graph that is used for model checking is of finite size, then the model checking reduces to a simple tree search, using simple graph algorithms (e.g., depth-first search and breadth-first search). However, for real-life discrete-event systems, the resulting reachability graphs are either of infinite size or simply huge (the so-called problem of "State Explosion" [13]). During model checking, searching for the states that satisfy compound property specifications will take a huge amount of time. Literature reveals various ways of confronting the state explosion problem:

- Symbolic model checking: in this approach, all the states are not checked one at a time. Instead, the state space is traversed by taking large numbers of states (e.g., a block of states) at a single step [20].
- Abstraction: in this approach, a large model is reduced into a smaller one (simplified model) while preserving the properties under scrutiny [21].
- Slicing of Petri Nets: A large Petri Net is sliced into two or more smaller Petri Nets, and the model checking is done on the smaller models individually [22].
- Modular model checking: A large Petri Net is decomposed to smaller and modular components (modules), and each module is checked individually [23].

GPenSIM provides some basic support for modular model building [24]. In GPenSIM, only one level of modules can be used; hierarchical modules (e.g., modules within modules) is not possible. However, since GPenSIM already has the modular model building capability, the modular model checking will be the more suitable direction for extending GPenSIM for resolving the problem of state explosion.

VI. Conclusion

This paper discusses and proposes the extension of GPenSIM for model checking. This paper identifies the extensions in three fronts, 1) generating complete reachability graphs by GPenSIM, 2) implementing computational tree logic (CTL) functions on GPenSIM/MATLAB, and 3) Resolving the problem of state explosion.

GPenSIM is developed with a long-term goal of modeling, simulation, performance analysis, and control of "safety-critical applications" in Cyber-Physical Systems (CPS). CPS are complex systems in which distributed software modules are tightly connected with control software. GPenSIM is to be developed as a tool that will be useful for modeling CPS. To achieve this goal, there are two steps to be completed:

Step-1: A software that can model, simulate, and perform control algorithms, both continuous (e.g., using the classical feedback control) and discrete (e.g., Petri Net-based Supervisory control) control; GPenSIM (current version 10) is capable working with discrete-event systems. Through its interface, GPenSIM (being a MATLAB toolbox) can also use the functions of the classical control toolboxes on the MATLAB, to work with continuous systems. Thus, this step is complete.

Step-2: Perform model checking: this paper is the starting point of this step-2, identifying the issues and how to proceed to realize this step.

References

[1] R. Davidrajuh. Modeling Discrete-Event Systems with GPenSIM: An Introduction. Springer International Publishing. R. 2018.

[2] GPenSIM-Website: http://www.davidrajuh.net/gpensim/

[3] A. Cameron, M. Stumptner, N. Nandagopal, W. Mayer, & T. Mansell, "T. Rule-based peer-to-peer framework for decentralised real-time service oriented architectures." Sci. Comput. Program. 97, 2015, pp. 202–234.

[4] U. Mutarraf, K. Barkaoui, Z. Li, N. Wu, & T. Qu, "Transformation of Business Process Model and Notation models onto Petri nets and their analysis." Advances in Mechanical Engineering, 10(12), 2018, 1687814018808170.

[5] H. Chang, "A Method of Gameplay Analysis by Petri Net Model Simulation," J. Korea Game Soc. 15, 2015, pp. 49–56, doi:10.7583/JKGS.2015.15.5.49.

[6] S. D. Jyothi, "Scheduling Flexible Manufacturing System Using Petri-Nets and Genetic Algorithm," Department of Aerospace Engineering, Indian Institute of Space Science and Technology: Thiruvananthapuram, India, 2012.

[7] L. Z. S. Hussein, "Simulation of Food Restaurant Using Colored Petri Nets," J. Eng. Dev. 2014, 18, pp. 77–88.

[8] B. Skolud, D. Krenczyk, & R. Davidrajuh, "Solving repetitive production planning problems. An approach based on Activity-oriented Petri nets." In International Joint Conference SOCO'16-CISIS'16-ICEUTE'16 (pp. 397-407). Springer, Cham. 2016.

[9] R. Davidrajuh & B. Lin, "Exploring airport traffic capability using Petri net based model." Expert Systems with Applications, 38(9), 2011, pp. 10923-10931.

[10] R. Melberg & R. Davidrajuh, "Modeling Atlantic salmon fish farming industry" In Industrial Technology, ICIT 2009. IEEE International Conference on. IEEE. 2009, pp. 1-6.

[11] R. Davidrajuh & N. Saadallah, "Implementing A Cohesive PT Nets with Inhibitor Arcs in GPenSIM," International Journal of Simulation: Systems, Science & Technology, 18(2), 2017, pp. 8.1 – 8.5.

[12] A. Roci & R. Davidrajuh, "Scheduling Jobs in Multi-Grid Environment," In Computer Modelling and Simulation (UKSim), 2018 UKSim 20th International Conference on. IEEE. 2018, pp. 67-72.

[13] C. Baier & J. P. Katoen. Principles of model checking. MIT press. 2008.

[14] T. Murata, "Petri nets: Properties, analysis and applications," Proceedings of the IEEE, 77(4), 1989, pp. 541-580.

[15] J. L. Peterson, Petri net theory and the modeling of systems. Rentice-Hall, NJ. 1981.

[16] M. Huth, & M. Ryan, M. Logic in Computer Science: Modelling and reasoning about systems. Cambridge university press. 2004.

[17] E. W. Mayr, "An Algorithm for the General Petri Net Reachability Problem," SIAM J. Comput., 13(3), 441–460.

[18] R. Davidrajuh & B. Hussein, "Modeling logic systems with structured array-based logic," Modeling, Identification and Control, 24 (1), 2003, pp. 27-35.

[19] R. Davidrajuh, "Array-Based Logic for Realizing Inference Engine in Mobile Applications". I J of Mobile Learning and Organisation (IJMLO), Vol. 1, No.1. 2007, pp. 41-57

[20] M. Kwiatkowska, G. Norman, & D. Parker, "Probabilistic symbolic model checking with PRISM: A hybrid approach". International Journal on Software Tools for Technology Transfer, 6(2), 2004, pp. 128-142.

[21] D. Dams, & K. S. Namjoshi, "The existence of finite abstractions for branching time model checking." Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science. IEEE, 2004, pp. 335-344.

[22] A. Rakow, "Slicing petri nets with an application to workflow verification". In International Conference on Current Trends in Theory and Practice of Computer Science. Springer, Berlin, Heidelberg. 2008, pp. 436-447.

[23] O. Kupferman, & M. Y. Vardi, "Modular model checking," In Compositionality: The Significant Difference. Springer, Berlin, Heidelberg. 1998, pp. 381-401.

[24] R. Davidrajuh, "Modular Petri Net models of Communicating Agents: A GPenSIM Approach," International Joint Conference SOCO'17, León, Spain, September 6–8, 2017, Proceedings, Advances in Intelligent Systems and Computing 649, Springer, DOI 10.1007/978-3-319-67180-2_32

[25] W. Reisig, Petri nets: an introduction (Vol. 4). Springer Science & Business Media. 2012.

[26] G. Gardey, D. Lime, & M. Magnin, "Romeo: A tool for analyzing time Petri nets." In International Conference on Computer Aided Verification Springer, Berlin, Heidelberg. 2005, pp. 418-423.

[27] T. Yoneda, H. Hatori, A. Takahara, & S. I. Minato, "BDDs vs. Zero-Suppressed BDDs: for CTL symbolic model checking of petri nets," In International Conference on Formal Methods in Computer-Aided Design. Springer, Berlin, Heidelberg. 1996, pp. 435-449.

[28] O. Roig, J. Cortadella, & E. Pastor, "Verification of asynchronous circuits by BDD-based model checking of Petri nets," In International Conference on Application and Theory of Petri Nets. Springer, Berlin, Heidelberg, 1995, pp. 374-391.

[29] A. Cheng, S. Christensen, & K. H. Mortensen, "Model checking Coloured Petri Nets-exploiting strongly connected components," DAIMI report series, 26(519), 1997.

[30] M. Nielsen, V. Sassone, & J. Srba, "Towards a notion of distributed time for Petri nets." In International Conference on Application and Theory of Petri Nets. Springer, Berlin, Heidelberg. 2001, pp. 23-31.

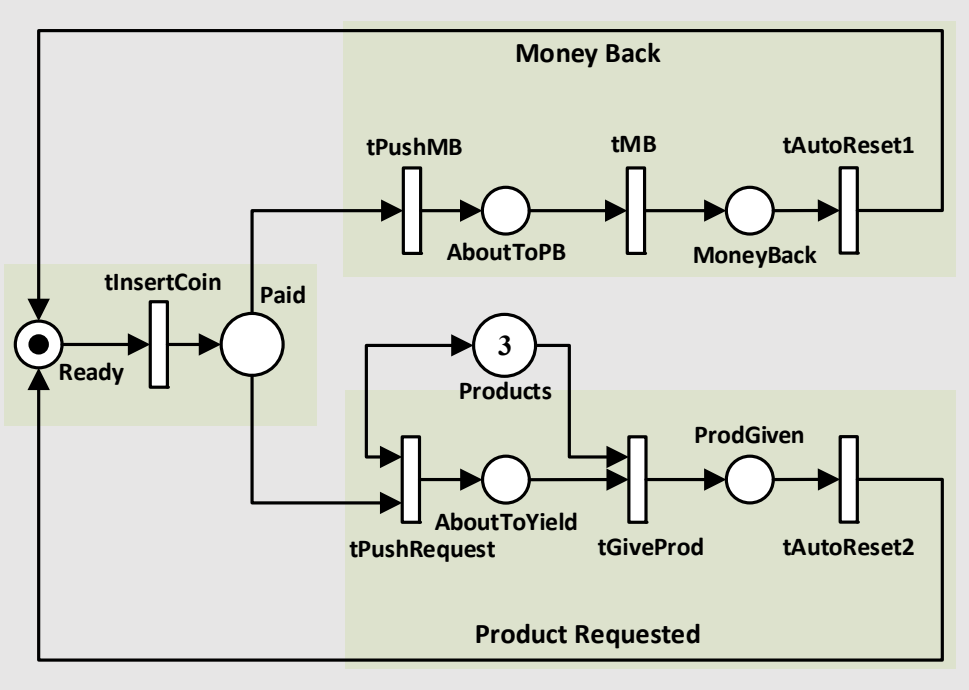

Figure 2. The Petri Net model of a simple soda vending machine.

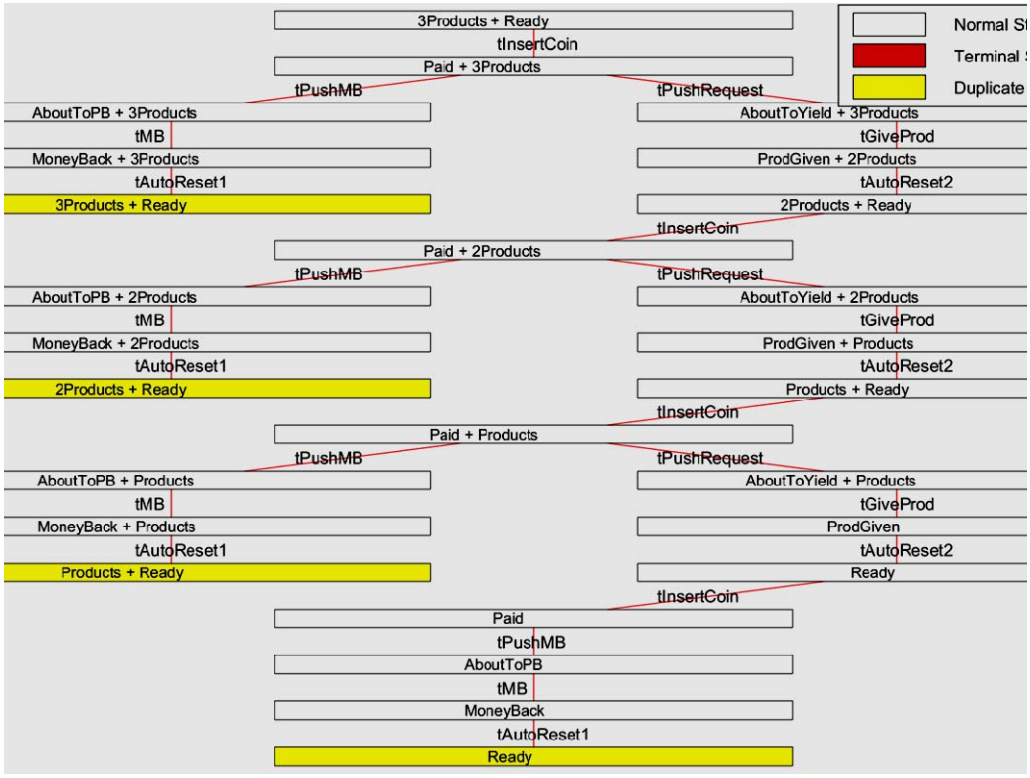

Figure 3. The Reachability Graph.

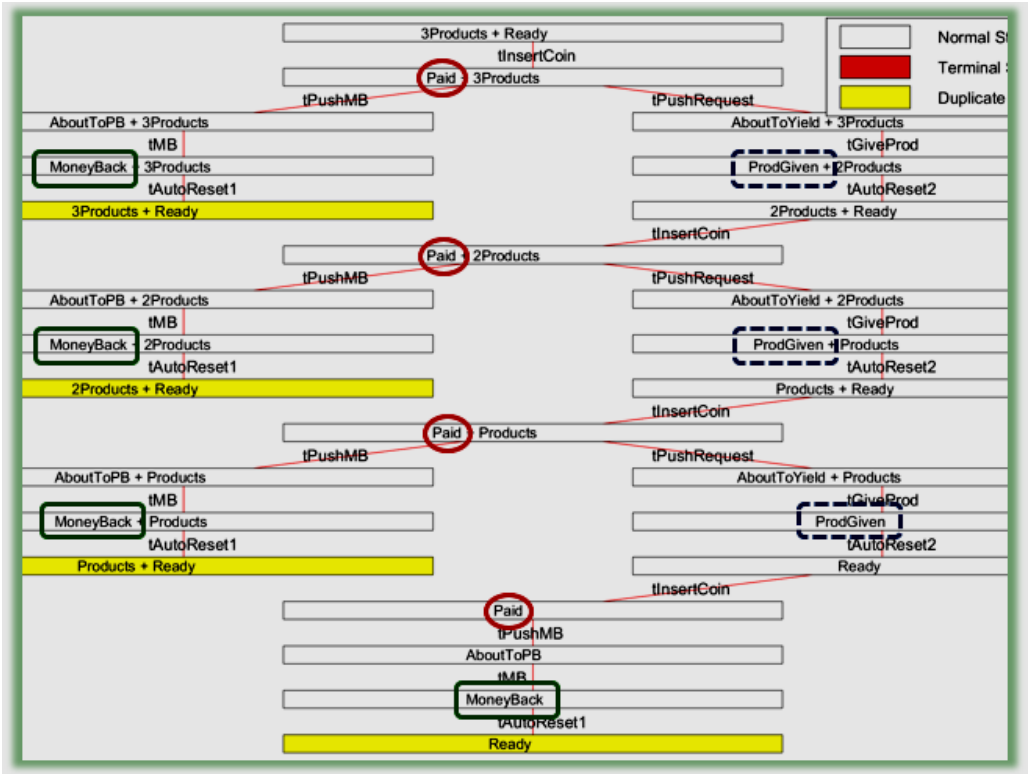

Figure 4. The Petri Net model satisfies the properties A and B.

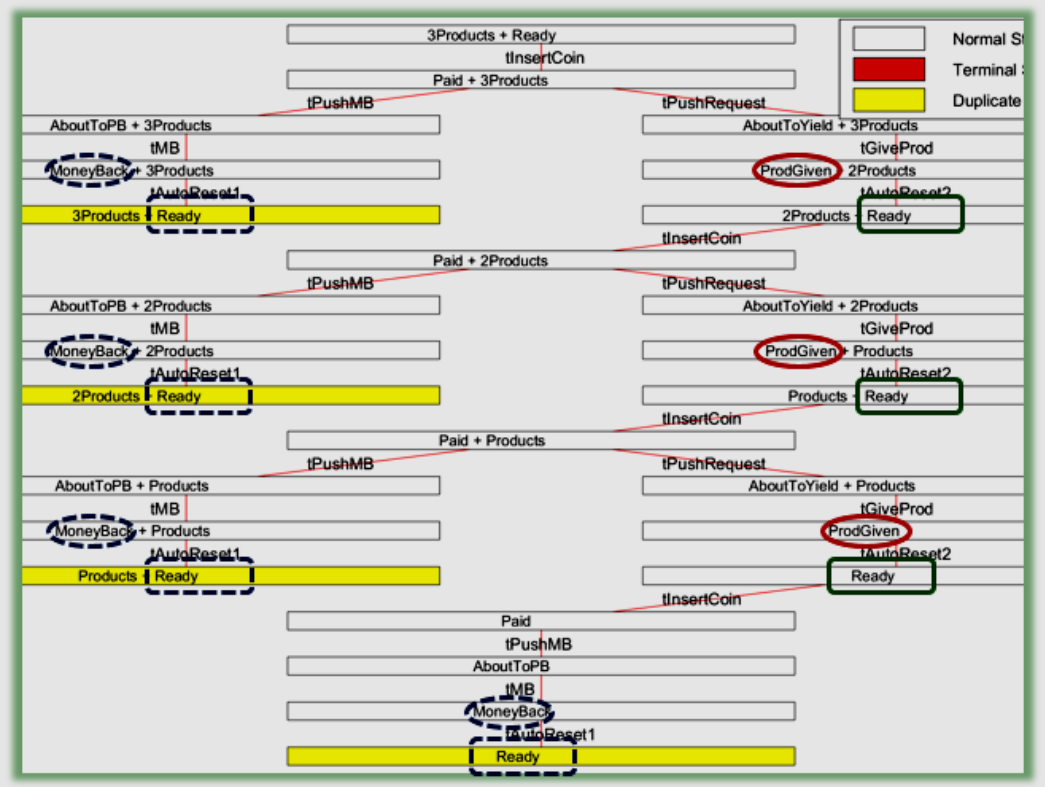

Figure 5. The Petri Net model satisfies the properties C and D too.