

## **From non-autonomous Petri net models to executable state machines**

[Formato pós-print]

### **Abstract:**

Petri nets have long been known as a readable and powerful graphical modelling language. In particular, Petri nets also allow the creation of high-level models of embedded controllers. These models can be translated to executable code. This possibility is already available in some tools including the IOPT Tools. Another possibility is to translate the Petri net model into a state machine, which can then be easily executed by an even larger number of platforms for cyber-physical systems. In that sense, this paper presents a tool that is able to generate a state machine from a non-autonomous class of Petri supported by the IOPT Tools framework (which is publicly available). These state machines would be too large to be manually generated, but can now be automatically created, simulated, and verified using an higher-level modelling language. The state machines can then be used for execution or even as input for additional verification tools. This paper presents the translation algorithm and an illustrative example.

**Published in:** [2019 IEEE 28th International Symposium on Industrial Electronics \(ISIE\)](#)

**DOI:** <http://dx.doi.org/10.1109/ISIE.2019.8781246>

**Publisher:** IEEE

Palavras-chave:

Model-driven development

Cyber-physical systems

Petri nets

Design tools

Embedded systems

SECTION I.

## **Introduction and motivation**

Petri nets are well-known for allowing a readable graphical specification of concurrency and synchronization. Many classes of Petri nets have been created, allowing the use of Petri nets for many different purposes. One possible dual classification is based on the explicit semantic dependency on external entities, namely signals and events. These two types are named, respectively, autonomous nets, the "classical" non-deterministic Petri nets, and non-autonomous Petri nets. The latter can have a non-deterministic or a deterministic semantics, depending on the application area; for controller modeling, a deterministic semantics is usually used.

This paper presents an algorithm that takes a non-autonomous and deterministic Petri net model and generates a state machine, which is behaviorally equivalent to the Petri net model. This is amenable to verification by state exploration, but also and most notably to its execution on low resource platforms. One may argue that this procedure will revert Petri nets intrinsic capabilities to model concurrency and will generate a representation which may suffer from the state explosion problem. Even though, the procedure can be very useful, as it supports generation of intermediate code amenable to be directly executed by (potentially) all implementation platforms and allows users to benefit from an initial specification using Petri nets, complemented by an extended platform support for implementation (including low-cost platforms with few resources). Users can still model a concurrent system, yet its implementation becomes sequential. To the best of our knowledge this is the first algorithm allowing the creation of state machines from non-autonomous Petri net models already supported by a set of freely available tools. The Petri net models are executable and can be simulated and verified before the application of the new here defined algorithm. As the Petri net models also provide an higher level of abstraction, the algorithm also allows a new way to create state machines too big to be manually defined.

The following section presents a more detailed motivation and background, including a brief and informal presentation of the used class of Petri nets. After, Section III presents the tool and Section IV the developed algorithm. Section V exemplifies the application of that same algorithm to a small example, amenable to graphical representation and inspection. Finally, we conclude with some conclusions and foreseen future work.

## SECTION II.

# Background

The Input-Output Place-Transition nets class [3] was created as a readable and precise graphical language to model discrete event systems controllers. It was inspired by other non-autonomous classes of Petri nets [1], [2], [6], [7], [13] and motivated the development of a web-based tool set named IOPT Tools [4] [9].

In the following section, we present the IOPT Petri nets class and the IOPT Tools framework.

## A. IOPT nets

Here, we present an informal introduction to the IOPT Petri nets class and its semantics. A formal specification of its syntax and semantics can be found elsewhere [3].

IOPT nets is a class of non-autonomous Place/Transition Petri nets (e.g. [12]). In the case of IOPT nets, the non-autonomous characteristic means that the model semantics depends on external elements, namely signals and events: for a transition to fire it must be *enabled* and *ready*. It is *enabled* when there are enough tokens in its input places — the usual fire condition for Petri nets. It is *ready* when the conditions that depend on the external environment are true. These conditions assume two forms, both associated to transitions: (1) guards that depend on input signal values and (2) input events. The guards are logical expressions that must evaluate to true for the transition to be *ready*. These logical expressions use input signals as variables. The events are Boolean values that are true when the associated input signal changes between two execution steps. If the event should be generated when the associated signal increases or decreases its value, we specify an "up edge" or a "down edge" associated with the event, respectively.

As usual with Petri nets, the net semantics is based on a step by step execution. For IOPT nets, each step has the following parts:

1. Acquire input signal values;
2. Store those values for the next step;
3. If first step, set events to false and jump to point 4; else compare the newly acquired values with the ones from the previous step to compute the events values;
4. Identify enabled and ready transitions;
5. Fire all enabled and ready transitions (maximal step semantics), except in cases of effective conflicts, which are resolved by priorities associated to transitions; each fired transition can change signal output values, either by assigning new values or by incrementing/decrementing the current values;
6. Execute actions associated to places; these can also change the output signal values.

## B. IOPT Tools

The IOPT Tools framework is a web-based set of tools that only requires the use of a browser. Hence, no local software installation is needed. It includes the following main tools: a graphical editor, a graphical simulator (normally known as a token-player, having a graphical user interface similar to the graphical editor animated by transition occurrence, complemented by a timing diagram), a remote debugger, C and VHDL code generation, and a state space generator that can be queried using an *ad hoc* language [10].

Besides, each model is stored and can be retrieved in the PNML interchange format, an XML based language for Petri net models [11]. The newly developed and here presented tool for state machine generation uses the PNML file and the generated C code.

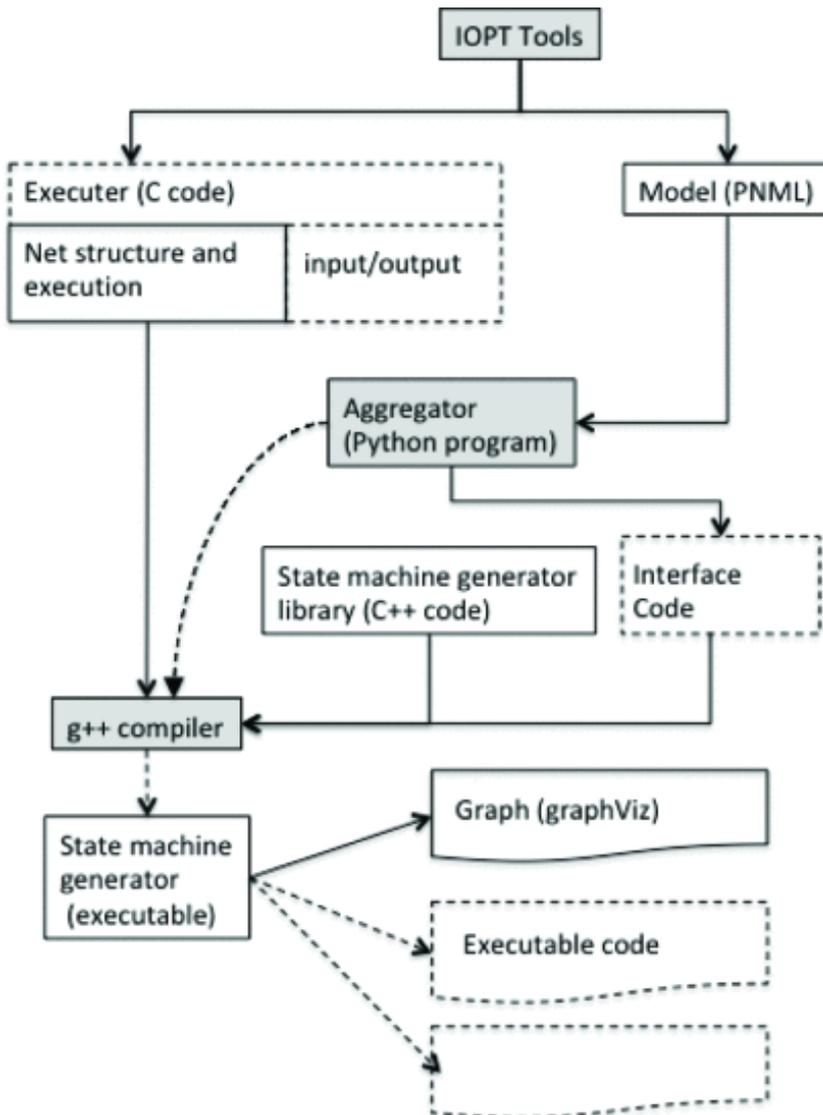
The IOPT Tools is freely available at <http://gres.uninova.pt/IOPT-Tools> and a list of related publications can be found at [http://gres.uninova.pt/iopt\\_publications.html](http://gres.uninova.pt/iopt_publications.html).

SECTION III.

## State machine generator architecture

The state space generator, available on the IOPT Tools, considers enabled transitions but ignores associated conditions on signal values, as well as possible variations on the input signal values. Hence, it is useful to check some safety and liveness properties, yet it cannot be used neither for verification that take into account the signal values, nor for implementation purposes.

Fig. 1 illustrates how the newly developed state machine generator is structured to allow the generation of state machines from the outputs provided by the IOPT Tools.



**Fig. 1.**

Generation process.

[View All](#)

The IOPT Tools is able to generate C code that can be compiled to be execute in low cost controllers (see top of Fig. 1). This code has two main parts: (1) the *Net structure and execution* code and (2) the code to read and write external signals (*input/output* in Fig. 1). Additionally, the IOPT Tools is able to generate a PNML file storing all the model data (see top right corner in Fig. 1). This file is used to store the model independently from the tool set, allowing its use by other tools. In fact, this possibility is also used between several tools in the IOPT Tools.

An *Aggregator* program, written in the Python programming language, reads the PNML model file and generates C++ *interface code*. This code allows the communication between the *Net structure and execution* code and the *state machine generator library*. Hence, the *Aggregator* program invokes the C++ compiler to create an executable from the three parts:

(1) *Net structure and execution code*, (2) the *state machine generator library*, and the (3) *interface code* (see Fig. 1).

The executable is then run and creates an internal representation of the state machine. From this internal representation, it is possible to generate several outputs. In Section V we present several graphs. These were generated from dot specifications using the Graphviz software [5] and the dot specifications were generated from the internal state machine representation. Other straightforward outputs are executable code for the state machine, as well as for verification tools allowing graph exploration and query. It must be stressed that the state machine generation uses the C code currently generated by IOPT Tools and executable on embedded controllers, namely the code that computes each marking and each output.

The following section presents the algorithm implemented in the *state machine generator library* to create an implementable state machine from the IOPT net model.

SECTION IV.

## State machine generation algorithm

When the translation to a state machine comes from an autonomous Place-Transition net the translation is very straightforward: each state corresponds to a net marking and each arc from one state to the other corresponds to the firing of a combination of enabled transitions. Yet, in an IOPT net, with its maximal step semantics and explicit dependencies on input signals, the arcs also are associated with combinations of signal values, and each state of the net model has additional information. Hence, compared to a typical state machine, and from a visual perspective, the generated state machine has two particularities:

1. States include three parts in their representation (1) the place markings; (2) the output signal values, as these act as global variables that can be used in transition guards; and (3) input signal values, when those values are necessary to compute events associated with enabled transitions;
2. The starting state is not a regular state; instead, the starting state, represented as smaller black circle, has one or more output arcs, each one targeting one initial state depending on the initial values of input signals; so, each arc carries a combination of signal's input values.

The algorithm for the generation of a state machine from an IOPT net model is here presented split in five parts: Algorithms 1, 2, 3, 4 and 5.

Algorithm 1 checks which transitions are enabled and collects the signals used (in *st*). It creates the starting special state (*startingPointState*), as well as one arc and one initial node for each combination of signal values. All those nodes are pushed to the stack pending for latter processing in procedure *handlePendingNodes* (see Algorithm 2).

**Algorithm 1:** State machine generation — starting point and initial nodes.

```

1 Stack<Node> pending;
2 List<SignalName> st ← signalsEnabledTran();
3 State startingPointState ← currentState();
4 Node startingNode ← new Node(startingPointState);
5 List<Binding> combinations ← generateFrom(st);
6 foreach c in combinations do
7   | newNode ← generateNewInitialNode(c);
8   | pending.push(newNode);
9 end
10 handlePendingNodes(pending);

```

Algorithm 2 processes the nodes in stack while this is not empty. For each node it loads the executor with the state data (marking, previousInputs, and outputSignals), updates which signals are stored (see Algorithm 3), checks if there are new signals (relative to the ones in the state), and proceeds to Algorithm 4 or 5.

**Algorithm 2:** State machine generation

```

1 Procedure handlePendingNodes (pending) :
2   | while pending.size() > 0 do
3     | Node node ← pending.pop()
4     | assignValuesFromNode(node)
5     | List<SignalName> newSignals ←
6       | updateSignals(node)
7     | if newSignals.size() > 0 then
8       | | replaceNodeWithReplicas(pending, node,
9       | | newSignals)
10    | else
11    | | // no new signals
12    | | handleSameSignals(pending, node)
13    | end
14  | end

```

Algorithm 3 checks if there are more signals in the state than the ones really needed. If that is the case, they are removed (removeUselessSignalsInState). The function returns the newly identified signals.

**Algorithm 3:** State machine generation – updateSignals function

```

1 Function updateSignals (node) :
2   | List<SignalName> sie ← signalsInEventsOfEnabledTran()
3   | node.removeUselessSignalsInState(sie)
4   | return node.signalsNotInState(sie)

```

Algorithm 4 handles the case where more signals than the ones in the node being processes are needed (this is the case for node 3 in Fig. 2(left). In this case, we need to create replicas of this node, one for each combination of signal values — nodes 4 and 5 in Fig. 2(middle) are replicas of node 3.

**Algorithm 4:** State machine generation – replaceNodeWith-Replicas function

```

1 Procedure replaceNodeWithReplicas (node) :
2   | deleteNode(node)
3   | // to be replaced by replicas
4   | List<Binding> combinations ← generateFrom(newSignals)
5   | foreach c in combinations do
6   |   | Node pNewNode ← new Node(node, c)
7   |   | oldNode = findNode(pNewNode.state)
8   |   | if oldNode found then
9   |   |   | oldNode.addInputEdgesFrom(node)
10  |   |   | delete pNewNode
11  |   | else
12  |   |   | pNewNode ← addInputEdgesFromNode(node)
13  |   |   | addNode(pNewNode)
14  |   |   | pending.push(pNewNode)
15  |   | end
16  | end

```

Algorithm 5 handles the case where no more signals than the ones in the node being processes are needed (this is the case, e.g., for node 5 in Fig. 2(middle)). In this case, we need to create arcs for each signal combination of signal values — node 5 gets two output arcs.

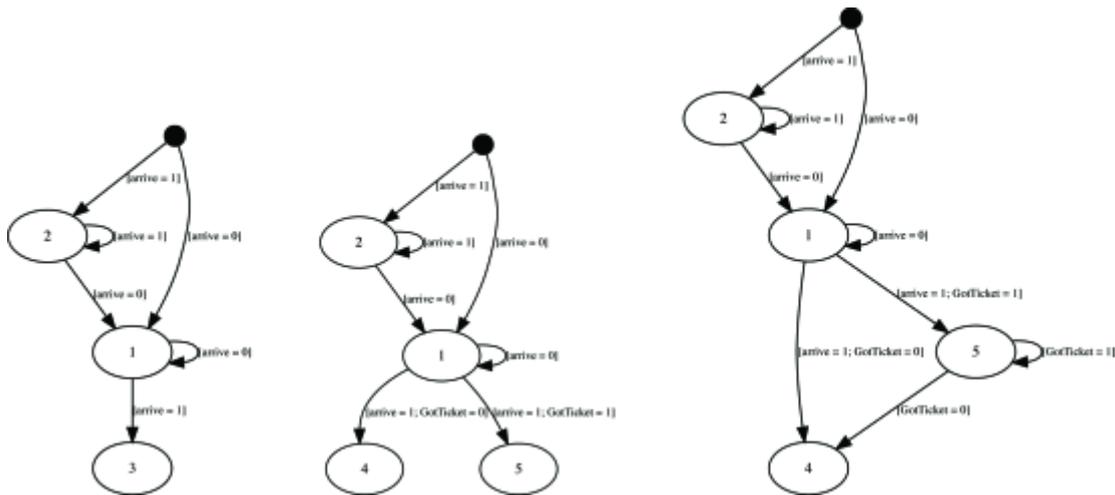


Fig. 2.

Initial processing of state 1 (left); after splitting state 3 in states 4 and 5 (middle); after processing state 5 (right).

[View All](#)

**Algorithm 5:** State machine generation – handleSameSignals function

```

1 Procedure handleSameSignals (node):
2   deleteNode(node)
   // it will be replaced by old or itself
3   oldNode ← findNode(node.state)
4   if oldNode found then
5     | oldNode.addInputEdgesFrom(node)
6   else
7     | addNode(node)
8   end
9   Set<string> signalsInNode ← node.state.signalNames;
10  Set<SignalName> signalNames ← signalsEnabledTran()
11  signalNames ← signalNames + signalsInNode
12  if signalNames.size() == 0 then
13    | executeStep(pending, node, emptyCombination);
14  else
15    | Combinations combinations ←
      generateCombinations(signalNames)
16    foreach c in combinations do
17      | executeStep(pending, node, c)
18    end
19  end

```

## Illustrative example

To exemplify the application of the algorithm, we use three variations of a model already presented elsewhere [8] (see Fig. 3). We proceed in a step by step fashion as new nodes are popped out of the *pending* stack. To make this feasible and readable, we use the shorter model, as the respective state machine is still very small and hence amenable to visual inspection.

The three models are presented in Fig. 3 and the three respective generated state machines are available at <http://gres.uninova.pt/~jpb/isie2019>. The shorter one at the top represents the entrance to a parking lot. The token in place "Entrance Free" specifies the initial state before entering the park. When the input signal "arrive" goes from 0 to 1, the event "ArriveIn" gets the value 1, which makes transition "arrive\_occupied" ready to fire. As it is also enabled, it fires, as IOPT nets have a deterministic maximal step semantics. Then, for transition "got\_ticket" to fire signal "GotTicket" must go from 0 to 1, similarly to transition "arrive\_occupied" for signal "arrive". While a token is in place "GateInOpen" the output signal with the same name is set to 1. Finally, when event "arrive" goes from 1 to 0, event "ArriveOut" is set. Notice that the activation of this event on the down edge is signaled by an inverted triangle.

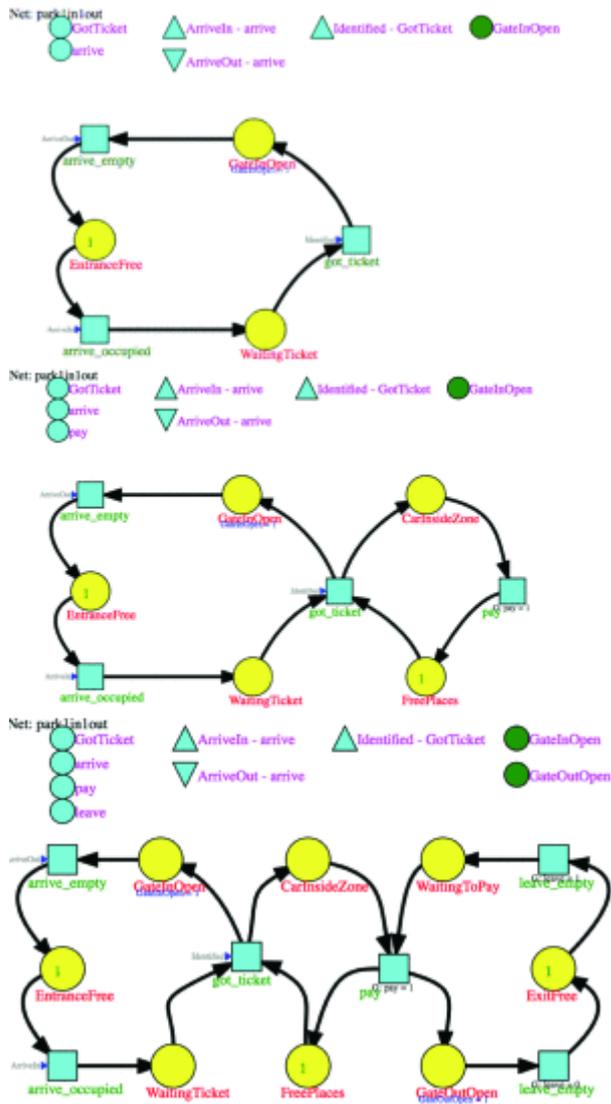
The second model in Fig. 3 adds the modelling of the places in the parking lot, and the third model adds an exit to the parking lot controller model.

The following sequence of graphical representations for state machines were generated using the Graphviz software [5], more specifically the dot tool. To that end, the state machine generator created a Dot language specification from its internal state machine representation after each pop out of the stack.

In the initial state, only transition "arrive\_occupied" is enabled and it only depends on the event "ArriveIn". Hence, the only relevant signal is "arrive", which can assume values 0 or 1. For that reason, our state machine starts in one of two possible states: with "arrive=0" and "arrive=1", as illustrated in Fig. 4(left).

When handling state 2 (popping it out of the stack), we have two possibilities, either "arrive=1" or "arrive=0". None of these fires the transition, hence the state machine does not get any more states, only arcs (see Fig. 4(right)).

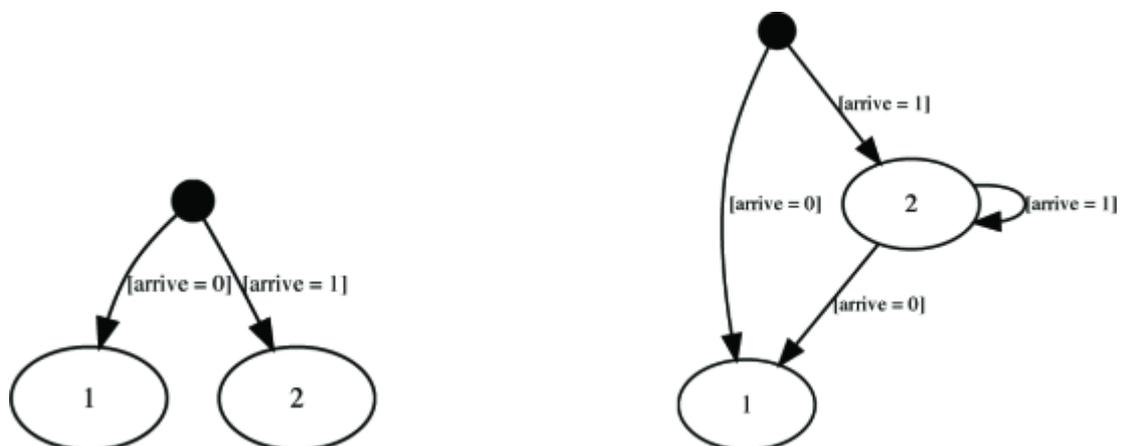
Regarding state 1, as the "arrive" signal is with value 0, it remains in the same state if the signal maintains its value. Yet, if signal "arrive" changes to 1, the "ArriveIn" is activated. This allows transition "arrive\_occupied" to fire thus changing the net marking. The new marking, together with the "arrive" signal value defines a new state in the state machine: state 3 in Fig. 2(left).



**Fig. 3.**

Entry to the parking lot (top); entry and parking area (middle); entry, parking area, and exit (bottom)

[View All](#)



**Fig. 4.**

Initial states (left); after processing state 2 (right).

[View All](#)

**TABLE I** State Data

<b>Node 1</b>	<b>Node 2</b>	<b>Node 4</b>
?arrive = 0	?arrive = 1	?GotTicket = 0
!GateInOpen = 0	!GateInOpen = 0	!GateInOpen = 0
"EntranceFree" = 1	"EntranceFree" = 1	"EntranceFree" = 0
"GateInOpen" = 0	"GateInOpen" = 0	"GateInOpen" = 0
"WaitingTicket" = 0	"WaitingTicket" = 0	"WaitingTicket" = 1
<b>Node 5</b>	<b>Node 7</b>	<b>Node 8</b>
?GotTicket = 1	?arrive = 0	?arrive = 1
!GateInOpen = 0	!GateInOpen = 1	GateInOpen = 1
"EntranceFree" = 0	"EntranceFree" = 0	"EntranceFree" = 0
"GateInOpen" = 0	"GateInOpen" = 1	"GateInOpen" = 1
"WaitingTicket" = 1	"WaitingTicket" = 0	"WaitingTicket" = 0

When state 3 is handled (popped out of the stack *pending*), the algorithm checks if the enabled transition depends on more signals than the ones in state 3 and also if all the signals in state 3 are needed to check readiness of transitions. In this case, the value of the "arrive" signal is important to define state 3, as the change from 0 to 1 was the motivation for this state. Also, the value of signal "GotTicket" is also important, as the event "Identified" depends on it and is used by transition "got\_ticket". Hence, an enabled transition needs to "know" about a possible last change in the signal "GotTicket" due to its associated event being used by the transition. For this reason, state 3 is removed and split into states 4 and 5, including dependencies on the signal "GotTicket" (see Fig. 2(middle)).

From state 5, the machine can simply move to state 4 if "GotTicket" signal value changes to 0, or remain in state 5 if no change in the signal value occurs (see Fig. 2(right)).

When in state 4, the machine can move to state 6 if the "GotTicket" signal value changes to 1 (see Fig. 5(left)).

Similarly, to state 3 that was split in 4 and 5, state 6 is removed and split into two (states 7 and 8) as illustrated in Fig 5(middle).

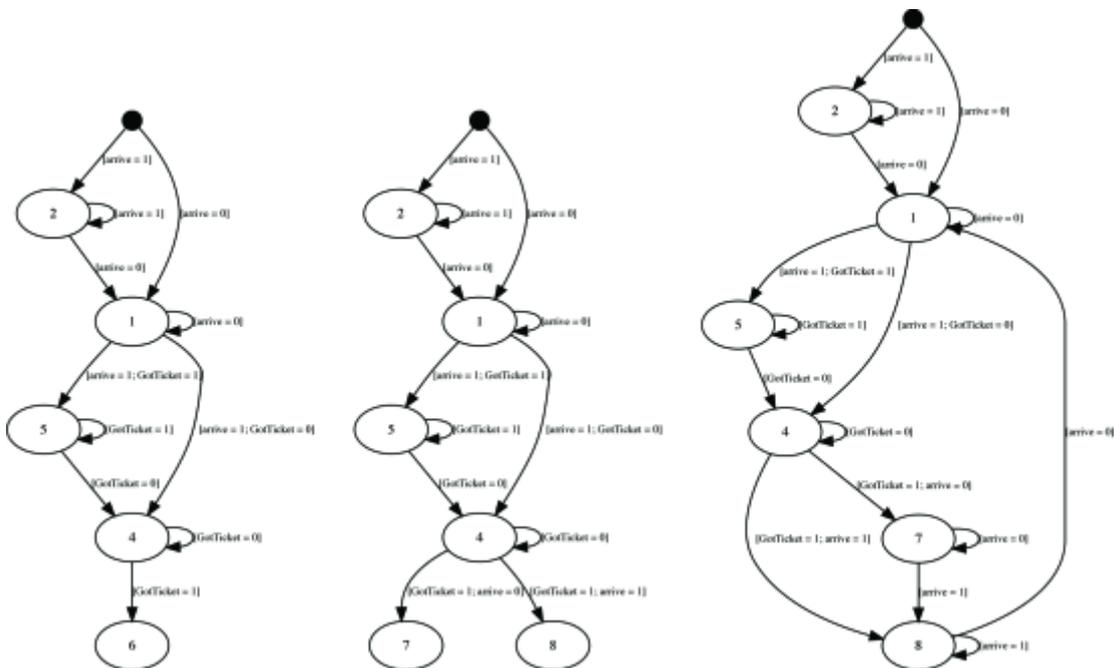
Finally, nodes 8 and 7, are processed and the result is illustrated, respectively in Fig. 5(middle) and 5(right).

Table I shows the markings and the values of input and output signals associated to each state — we denote the input and output signals with prefixes "?" and "!" respectively.

The state machine generator was also run on the three models in Fig. 3 with different values for the initial marking in place "FreePlaces", which models the parking lot capacity. Table II shows the results. The state machine generates roughly 3 to 10 times more states. Yet, with wider ranges for possible signal values the difference would be much larger.

## Conclusions and future work

The presented algorithm and tool provide support for the execution of Petri nets models in controllers with scarce resources relying on their behaviorally equivalent state machine, as long as they are able to execute a state machine. The tool already uses the output provided by the IOPT-Tools [4] [9]. As future work, we intend to fully integrate the tool in the IOPT-Tools cloud-based environment and to add code generation directly from the state machine representation, as well as benefiting from IOPT-Tools automatic code generation capabilities. Another line of work will explore integration of the state machine representation with state space verification tools, allowing back annotation of the initial Petri net model.



**Fig. 5.**

Initial processing of state 4 (left); after splitting state 6 in states 7 and 8 (middle); after processing state 7 — no more states to process (right).

[View All](#)

**TABLE II** Total states in state space from IOPT Tools and in state machine

<b>Model</b>	<b>IOPT Tools</b>	<b>State machine</b>
Park entry (Fig. 3(top))	3	6
Park entry and storage of 1 (Fig. 3(middle))	6	12
Park entry and storage of 2 (Fig. 3(middle))	9	26
Park entry and storage of 3 (Fig. 3(middle))	12	44
Park entry and storage of 10 (Fig. 3(middle))	33	170
Park entry and storage of 100 (Fig. 3(middle))	303	1255
Park entry, storage of 1 and exit (Fig. 3(bottom))	18	48
Park entry, storage of 2 and exit (Fig. 3(bottom))	27	85
Park entry, storage of 3 and exit (Fig. 3(bottom))	36	122
Park entry, storage of 10 and exit (Fig. 3(bottom))	99	381
Park entry, storage of 100 and exit (Fig. 3(bottom))	909	3016

## References

1. R. David, H. Alla, *Petri Nets & Grafcet; Tools for Modelling Discrete Event Systems*, Prentice Hall International (UK) Ltd, 1992.
2. R. David, H. Alla, *Discrete Continuous and Hybrid Petri Nets*, Springer Publishing Company, 2010.
3. L. Gomes, J. P. Barros, "Refining IOPT Petri nets class for embedded system controller modeling", *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, 2018.
4. L. Gomes, F. Moutinho, F. Pereira, "IOPT-tools – a web based tool framework for embedded systems controller development using Petri nets", *2013 23rd International Conference on Field programmable Logic and Applications*, pp. 1-1, Sept 2013.
5. "Graphviz", 2018, [online] Available: <http://graphviz.org/credits/>.
6. H.-M. Hanisch, A. Lüder, "A Signal Extension for Petri Nets and its Use in Controller Design", *Fundamenta Informaticae*, vol. 41, no. 4, pp. 415-431, 2000.
7. L. E. Holloway, B. H. Krogh, A. Giua, "A Survey of Petri Net Methods for Controlled Discrete Event Systems", *Discrete Event Dynamic Systems*, vol. 7, pp. 151-190, 1997.
8. F. Moutinho, L. Gomes, F. Ramalho, J. Figueiredo, J. P. Barros, P. Barbosa, R. Pais, A. Costa, "Ecore representation for extending PNML for Input-Output Place-Transition nets", *IECON 2010 - 36th Annual Conference on IEEE Industrial Electronics Society*, pp. 2156-2161, Nov 2010.
9. F. Pereira, F. Moutinho, L. Gomes, "IOPT-Tools – towards cloud design automation of digital controllers with Petri nets", *ICMC'2014 - International Conference on Mechatronics and Control*, 2014.
10. F. Pereira, F. Moutinho, L. Gomes, R. Campos-Rebelo, "IOPT Petri net state space generation algorithm with maximal-step execution semantics", *2011 9th IEEE International Conference on Industrial Informatics*, pp. 789-795, July 2011.
11. "PNML.org", 2015, [online] Available: <http://www.pnml.org>.
12. W. Reisig, *Petri nets: an Introduction*, Springer-Verlag New York, Inc, 1985.
13. M. Silva, *Las Redes de Petri: en la Automática y la Informática*, Madrid:Editorial AC, 1985.

## ACKNOWLEDGMENT

This work was partially financed by Portuguese Agency FCT Fundação para a Ciência e Tecnologia, in the framework of project UID/EEA/00066/2019.