Towards an Integrated Tool Support for the Analysis of IOPT Nets Using the Spin Model Checker

João Paulo Barros
Polytechnic Institute of Beja, Beja
Centre of Technology and Systems-UNINOVA, Caparica
Portugal
Email: joao.barros@ipbeja.pt

Luís Gomes
NOVA University Lisbon
Centre of Technology and Systems-UNINOVA, Caparica
Portugal
Email: lugo@fct.unl.pt

Abstract — This paper presents a model translation to allow automatic simulation and verification of controller models for cyber-physical systems. The models are constructed using IOPT nets, a non-autonomous Petri nets class. Those models are then translated into Promela models to be executed by the Spin model checker, a widely used open-source software verification tool. Three illustrative examples are presented: one autonomous model and two non-autonomous models. As future work, it is foreseen the integration with the freely available IOPT-Tools framework.

Index Terms — cyber-physical systems, controller design, verification, simulation, code generation, Petri nets, Spin, Promela, IOPT, IOPT-Tools

I. Introduction

Discrete event systems models are frequently used for several types of physical systems with a computational part — e.g., controllers, embedded systems, cyber-physical systems. These models allow system validation and verification, including the analysis of several properties.

Several languages offer support for the creation of this type of system model. In particular, graphical languages allow a visual and interactive construction and exploration of the model. Petri nets have a long tradition of offering solid support for constructing visual and highly readable models supported by precise and formal semantics (e.g., [6], [10], [19]. The graphical representations of Petri nets are directed graphs composed of two types of nodes: places and transitions. Places are commonly associated with the passive part of the modeled system, typically associated with conditions, states, or resources. They are represented by circles or ellipses, while transitions are associated with the active part associated with changes in state, actions or resources, and are represented by bars or rectangles. The graph's arcs connect nodes of different types (bipartite graph). Any number of input and output arcs can be connected to a node (place or transition). Each place can contain a set a tokens (the *place marking*) that together define the *net marking*. The model execution corresponds to transition *firing* where tokens in the transition input places are destroyed and new ones are created in the output places.

The IOPT-Tools framework [17] takes advantage of a nonautonomous class of Petri nets especially tailored for the specification, analysis, and synthesis of controllers — the IOPT nets - Input-Output Place-Transition nets [11], [13]) — allowing the automatic generation of C and VHDL code for the execution of discrete event-driven controllers.

The structure of the paper is as follows. Section II provides background information on IOPT nets, Spin and Promela language, complemented by Section III addressing related works. Section IV presents the IOPT net models proposed translation strategy into Promela models, while Section V briefly presents an application example, and Section VI concludes and point some future works.

II. Background

A. IOPT nets and Controller specification

Traditional Petri nets are sometimes classified as *autonomous* to explicitly state that their semantics is independent of the external environment, e.g., the transition enabling and firing is not influenced by any external elements. However, for practical applications, particularly when the Petri net models a controller, it becomes highly convenient to explicitly model the relation between the controller (and its Petri net model) and the environment (that is controlled) and change the model semantics accordingly. Petri nets have contributed to several industrial standards, such as MFG - mark flow graphs (adopted by JIS - Japanese Industrial Standards), GRAFCET (adopted by French AFCET and ADEPA), and the Sequential Function Chart (SFC) defined in international standard IEC 61131-3. Therefore, the Petri net classes with semantics dependent on the external environment are usually named as *non-autonomous*, which are augmented with dependencies on time, inputs sensors and output actuators. In the case of controllers for discrete-event systems, dependencies to the environment can be established by adding dependencies to two entities in the net: input and output signals. IOPT nets [11], [13] provide these and add two additional ones: input and output events. This interface adopts proposals from interpreted and synchronized nets [5],

[6], [23]. Several other classes of Petri nets have been proposed having factory automation applications in mind [8], [14].

The syntax and semantics of IOPT nets can be found elsewhere [13]. In summary, IOPT nets can be seen as an extension to Place-Transition nets, where several non-autonomous dependencies were introduced, namely input and output signals and events, where transition firing are conditioned by input event occurrence and guard functions evaluation (constrained by input signals conditions), and are able to generate output events and output signals updating. Output signals can also be generated associated with places.

In order to ensure a deterministic execution, which is of paramount importance for embedded controllers in many areas of application, a step-based execution semantics is selected, including several aspects: maximal step execution (which means that all transitions enabled and ready to fire will fire in the same execution step), cycle-accurate execution (which means that all events occurred during the execution of the last step will be considered at the next step execution), single server semantics (which means that even enabled multiple times, transitions will only fire once per execution step), and priorities associated with transitions (which can be used to embed into the model strategies to automatically solve conflicts, ensuring deterministic execution, even at the expense of an unfair strategy). Finally, test arcs are also available. Fig. 1 explicitly presents the way how input signals are acquired and used to generate input events, based on the comparison between current and previous values of input signals.

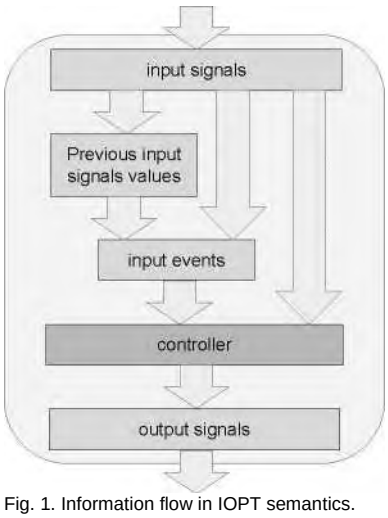

Fig. 1. Information flow in IOPT semantics.

IOPT-Tools [12], [17] are a cloud-based platform, publicly available at http://gres.uninova.pt/IOPT-Tools/, using IOPT nets and including an interactive graphical Petri net editor, simulation, debug and model-checking tools based on reachability graph generation, and automatic code generation tools. In particular, it can generate C code or VHDL hardware descriptions ready to be directly deployed into implementation platforms (without writing additional lines of code).

B. Spin and Promela

Explicit Model checking is a simple and obvious idea that, in practice, is still the best approach for many areas of application, namely verification of communication protocols and software [16]. To that end, the Spin model checker is one possible tool [3], [15]. Gerald J. Holzmann initially developed it to verify communication protocols. Interestingly, these have also been a long-standing motive to the use of Petri nets.

The Spin model checker allows two main execution formats: simulation and verification. Simulation can always be used but can only show the existence of failures, not their absence. Verification can indeed prove system correctness as it checks all possible states. The limit is the well-known state-explosion problem, but Spin is a particularly good tool to mitigate this as it implements partial order reduction theory to optimize the search. More details, including two other modes for using Spin, can be found at the Spin webpage [24] and the tool primary reference [15]. Spin has become widely used in industries that build critical systems and is arguably the world's most popular tool for checking concurrent systems and one of the most powerful [3]. It has been freely available since 1991, has a large and active community, and a periodically held international conference – the International SPIN Symposium on Model Checking of Software- currently in its 27th edition.

The Spin tool reads models written in the Promela language and automatically generates custom verifier code written in the C Programming language. The Promela language has a syntax similar to the ones of C-like languages. However, it differs from programming languages primarily due to its inherent non-determinism. For example, if two or more branches of its if statement evaluates as true, then the simulation will randomly choose one to be executed. Additionally, the verifier will check all possibilities.

III. RELATED WORK

To the best of our knowledge, the work by Gupta et al. was the first to present a translation from Petri nets to Promela [9]. The objective was the same as ours: to use the Spin tool to verify liveness and safety properties, including boundedness, of the Petri net model. They also propose integrating a modeling environment — where Petri net models are created — with Spin. However, the used Petri nets are the usual Place/Transitions nets, which are autonomous — with no external dependencies — and have non-deterministic

transition firing (e.g., [21]).

Ribeiro and Fernandes [22] present a proposal for the translation of Synchronous Petri net models into Promela. Similar to IOPT nets, Synchronous Petri nets are also based on P/T nets and non-autonomous as they allow values of external signals to be used as guards in transitions. Also, transitions firings are synchronized with the active edge of a global clock. However, they differ from IOPT nets because they have no direct support for input events. Also, IOPT nets add several forms of external output specification by allowing signal output changes based on place actions and output events in transitions. To the best of our knowledge, no tool currently supports creating models based on Synchronous Petri nets.

Barbosa et al. [2] emphasize the use of Model-Driven Architecture for systems development through Petri nets. They present a simple Promela model based on an IOPT net model in this context. However, the model ignores non-autonomous elements — namely signals and events — and the translation rules are not presented.

Venero and Silva [7] present a translation from a class of autonomous Coloured Petri nets (CPN) — named Nested Petri nets — to Promela. The semantics is the classic for CPN. Unfortunately, we could not identify a tool for creating Nested Petri nets.

Alam and He [1] present a translation from Predicate- Transition nets — another class of autonomous high-level nets — to Promela. An open-source tool is also mentioned but not accessible using the provided link. [18]

IV. MODEL TRANSLATION

A tool, named iopt2pml, was developed to translate IOPT net models, in the PNML format, to Promela models. The PNML format is an XML-based standard for the interchange of Petri net models [4], [20]. Hence, the tool starts from a structural model and creates an executable model. To that end, the tool has to express in the Promela language the IOPT net semantics. Interestingly, despite the concurrency intrinsic to Petri net models, the IOPT nets execution, for a single time domain — i.e. a single controller with a single processor and clock — can be executed as a single sequential process. As already presented in Section II-A, this is a direct consequence of the maximal step semantics and the eventual use of priorities in transitions which allow for a deterministic execution steps, where all enabled and ready transitions fire.

Next, we use two easily readable IOPT models to explain how the translation is conducted. We also show some important checks readily available from the generated Promela model.

1) Autonomous model: The Promela model, for the net in Fig. 2, contains a single process, named *Controller* (see Listing 1). In the case of an autonomous net like the one in Fig. 2 the process simply tries to fire all enabled transitions. To that end, the new net marking is stored in a new ”next” buffer. After each step, this ”next” buffer updates the current marking buffer. This loop repeats indefinitely.

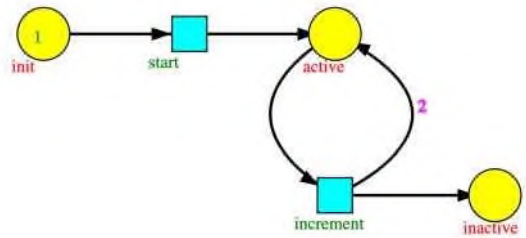

Fig. 2. Autonomous net model.

```
active proctype Controller() {
    start:
        print_current_state();
        printf("Start firing tests");
        testFire(t_start);
        testFire(t_increment);
        printf("End firing tests\n");
        copyNextToCurrentMarkings();
        checkPlaceMarkingBounds();
    goto start;
}
```

Listing 1. Promela process for autonomous IOPT net (no input signals)

The places *active* and *inactive* in the IOPT net model in Fig. 2 are both unbounded: after the firing of transition *start*, *active* gets one token; then, each fire of transition *increment* increases the marking by one each both place. As the IOPT-Tools, by default, defines a bound of three for all places, place *active* will be the first to overflow its number of tokens (this default value can be changed after analysis of the model, considering a specific initial marking). This is illustrated by the simulation output in Listing 2. The generated code includes a Promela assertion for each place marking bound and outputs an informative message with the illegal value and respective place name: e.g., ”BOUND OVERFLOW: place active has marking value 4 with bound 3”. Additionally, Spin itself produces an ”assertion

violated" error message containing the respective assertion line number.

The same model can be verified. The process is straightforward and is illustrated in Listing 3: (1) the Spin tool is used to generate the verifier C code; (2) the C code is compiled, and an executable program is generated; (3) the executable is run to exhaustively verify the Promela model. It then reports the assertion violation. It is important to note that this verification is exhaustive, contrary to the simulation.

Next, we present a non-autonomous model.

2) Non-autonomous model: The model in Fig. 3 is a nonautonomous net, as it contains transitions whose firing depends on external signals or events: transition *start* can only fire if event *ev*1 is true, and this event can only become true if there is a change, between two consecutive steps, in the value of signal *is*1; due to the associated guards, transitions *increment* and *stop* firings are only possible if signal *is*2 has the value 0 or 1, respectively.

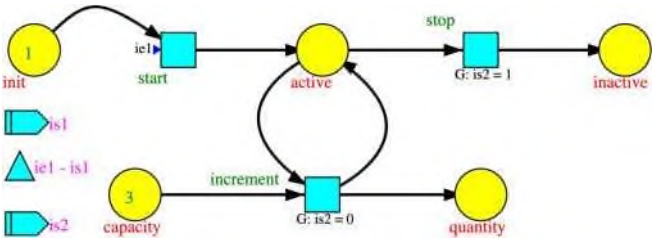

Fig. 3. Non-autonomous net model.

The code for a non-autonomous net has to include the generation of signal input values, as well as the computation

```
% spin simple.pml
      Current state before STEP 1:
      init       1       active       0
      inactive       0
Start firing tests
      Test transition 0
      ### Fired transition 0
      Test transition 1
      End firing tests
      Current state before STEP 2:
      init       0       active       1
      inactive       0
Start firing tests
      Test transition 0
      Test transition 1
      ### Fired transition 1
      End firing tests
      Current state before STEP 3:
      init       0       active       2
      inactive       1
Start firing tests
      Test transition 0
      Test transition 1
      ### Fired transition 1
      End firing tests
      Current state before STEP 4:
      init       0       active       3
      inactive       2
Start firing tests
      Test transition 0
      Test transition 1
      ### Fired transition 1
      End firing tests
      BOUND OVERFLOW: place       active
      has marking value 4 with bound 3
spin: simple.pml:150, Error: assertion violated
spin: text of failed assertion:
assert((current_marking[1]<=3))
(...)
1 process created
```

Listing 2. Spin simulation output for net in Fig. 2

of input events. These are the first two tasks in the each step — after label start — as can be seen in Listing 4.

Listing 5 shows the generated Promela code related to the input signals for the example in Fig. 3. These are two binary input signals: *is*1 and *is*2. A random assignment simulates the input signals reading to each signal, where the verifier code checks all values. The generated code is not minimal to improve readability and facilitate possible future changes.

Listing 6 shows the generated Promela code related to the input event for the example in Fig. 3. There is one input event: *ie*1. The event value is computed as a function of the value change of the associated signal value. Hence, all events have to be reset after each step. Transitions with no input events have an implicit ”always true” event.

Due to the limitation imposed by the initial marking in place *capacity*, this net is bounded: no place marking can ever surpass three, which can be readily checked by verification. However, this net has a not so obvious and possibly undesirable property: it becomes dead — no transition can fire — after a few steps.

For convenience, a Boolean variable, named readyAndEnabled, is defined in the generated code (see Listing 4). It has the value true when some transition

```
% spin -a autonomous.pml
% gcc -o pan pan.c
% ./pan
pan:1: assertion violated (current_marking[1]<=3)
(at depth 155)
pan: wrote simple.pml.trail

(Spin Version 6.5.2 -- 6 December 2019)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim                - (none specified)
        assertion violations       +
        acceptance   cycles        - (not selected)
        invalid end states        +

State-vector 24 byte, depth reached 155, errors: 1
      120 states, stored
        0 states, matched
      120 transitions (= stored+matched)
       36 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.006       equivalent memory usage for states
    (stored*(State-vector + overhead))
    0.274       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  128.730       total actual memory usage

pan: elapsed time 0 seconds
```

Listing 3. Spin verification output for net in Fig. 2.

can fire — is ready and enabled — and, consequently, will fire. This and other variables could also be easily added to the generated Promela model, allowing a simple verification of more specific properties.

```
bool readyAndEnabled = false;
active proctype Controller() {
  start:
    updateInputSignals();
    eval_input_events();
    print_current_state();

    printf("\nStart firing tests\n");
    readyAndEnabled =
        is_readyAndEnabled(t_start) ||
        is_readyAndEnabled(t_increment) ||
        is_readyAndEnabled(t_stop);
    testFire(t_start);
    testFire(t_increment);
    testFire(t_stop);
(...)
```

Listing 4. *readyAndEnabled* in generated code.

This can be checked using a linear temporal language (LTL) formula that checks if the variable *readyAndEnabled* is eventually true in all computations, e.g., eventually, some transition will be ready and enabled and, consequently, will fire. This is a liveness property as it states that something ”good” eventually happens in the computation. It is also essential to notice
that the formula applies to all possible computations. This is expressed by the LTL formula ◊*readyAndEnabled* or, in Promela syntax, <>readyAndEnabled. Listing 7 illustrates

```
#define N_INPUT_SIGNALS 2
bit is_bool_id_is1 = 0;
bit is_bool_id_is2 = 1;
bit previous_input_signals[N_INPUT_SIGNALS] = {0, 0};
bit current_input_signals[N_INPUT_SIGNALS] = {0, 0};

#define is1 current_input_signals[0]
#define is2 current_input_signals[1]

inline readPhysicalInputSignals() {
    // non-deterministic assignment
        int r
        select(r: false..true);
        current_input_signals[is_bool_id_is1] = r;

        select(r: false..true);
        current_input_signals[is_bool_id_is2] = r;
}

inline updateInputSignals() {
    byte ris;
    for(ris : 0..(N_INPUT_SIGNALS - 1)) {
        previous_input_signals[ris] =
           current_input_signals[ris];
    }
    readPhysicalInputSignals()
}
```

Listing 5. Promela generated code for input signal handling.

```
#define N_INPUT_EVENTS 1
#define  ie_ie1_is_is1 0
bool input_events[N_INPUT_EVENTS] = {0}
inline eval_input_events() {
        input_events[ie_ie1_is_is1] =
           previous_input_signals[is_bool_id_is1] <
           current_input_signals[is_bool_id_is1];
}
inline set_events_false() {
    byte ie_i;
    for(ie_i : 0..(N_INPUT_EVENTS - 1)) {
        input_events[ie_i] = false
    }
}
#define are_events_true(t) \
        ( t == 0 -> input_events[ie_ie1_is_is1]  : \
        ( t == 1 -> true : \
        ( t == 2 -> true : \
        false)))
```

Listing 6. Promela generated code for input event handling.

the verification of this property with the corresponding output staring the assertion violation.

As a further verification, Listing 8 shows how to check that eventually readyAndEnabled will become always false: $\Diamond(\Box\neg readyAndEnabled)$, in Promela syntax <>([]! readyAndEnabled). For a liveness property, a counterexample is an infinite computation in which something good never happens: the ”acceptance cycle” mentioned in the output.

V. APPLICATION EXAMPLE

The proposed analysis strategy is here further illustrated using a simple well-known application example adapted from [23]. This example has been used in many other works in automation controller modeling with Petri nets. The goal is to model the behavior of a controller for a three cars system,

```
% spin -a -f "<>readyAndEnabled"  non-autonomous.pml
% gcc -o pan pan.c
% ./pan -a
warning: for p.o. reduction to be valid the never
claim must be stutter-invariant
(never claims generated from LTL formulae are
stutter-invariant)
pan:1: assertion violated  !(readyAndEnabled)
(at depth 325)
pan: wrote non-autonomous.pml.trail

(Spin Version 6.5.2 -- 6 December 2019)
(...)
```

Listing 7. Check liveness — some transition will eventually be ready and enable (and will consequently fire). This is false as the net will become dead.

```
% spin -a -f "<>([]!readyAndEnabled)"
simple-na-bounded-test.pml
% gcc -o pan pan.c
% ./pan -a
warning: for p.o. reduction to be valid the never
claim must be stutter-invariant
(never claims generated from LTL formulae are
stutter-invariant)
pan:1: acceptance cycle (at depth 56)
pan: wrote simple-na-bounded-test.pml.trail

(Spin Version 6.5.2 -- 6 December 2019)
(...)
```

Listing 8. Check non liveness — eventually no transition will be ready and enabled (consequently firing will stop). Liveness does not hold, an acceptance cycle is found.

having the goal of transporting goods from one end to another (common in the chemical processes industry). The three cars should start moving from one end to the other only when all cars are stopped at the same end, under the supervision of an operator, who activates two input buttons (GO and BACK), forcing starting movements in the direction right or left, accordingly. End-of-course detectors are referred to as A*i* and B*i*; activation of motors is referred as Motor*i*, and M*i*Dir inform about the direction ofthe movement (with i=1,2,3). Fig. 4 presents the controller's IOPT model.

For this system, it is helpful to check if no cars are moving in opposite directions. This safety property must hold for all states and can be readily checked using an assertion. Listing 9 shows the code that can be added to the generated Promela code for this purpose. It was inserted after the checkPlaceMarkingBounds() call. The Boolean variables forward and backward use the already generated names for each place marking. After, the variable (oneWay) is defined with the assertion itself. Next, it was only necessary to use the assert statement. Yet, additional code can also be added to provide better feedback in case of a failed assertion.

VI. CONCLUSIONS AND FUT URE WORK

The implemented IOPT net model to Promela translation adds to the IOPT-Tools framework the possibility to use the industrial-strength Spin tool for efficient model verification. In addition, the Promela language is simple and powerful enough to allow for practical, efficient, and sophisticated queries with a small development overhead. In particular, it is already

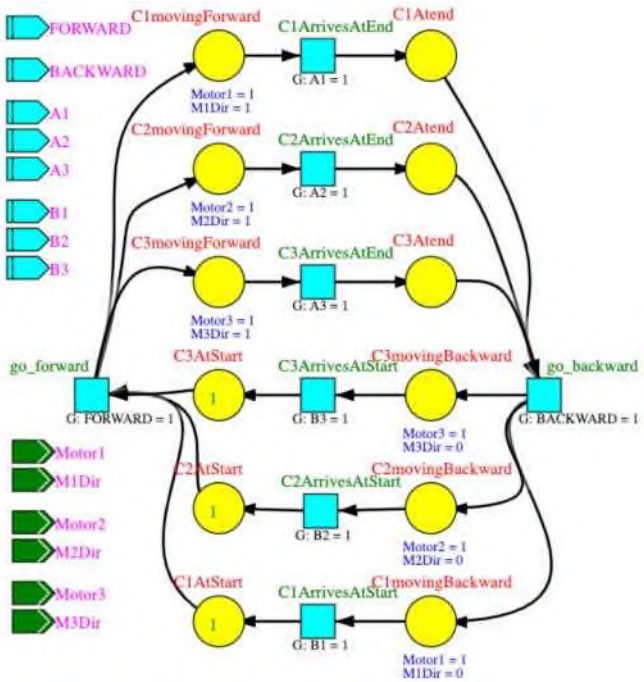

Fig. 4. Cars net model.

```
(...)

checkPlaceMarkingBounds();

bool forward = (p_C1movingForward_m +
                p_C2movingForward_m +
                p_C3movingForward_m > 0);
bool backward = (p_C1movingBackward_m +
                 p_C2movingBackward_m +
                 p_C3movingBackward_m > 0);
bool oneWay = !(forward && backward);
assert(oneWay);

(...)
```

Listing 9. Assertion added for safety test: no cars can ever be moving in opposite directions

possible to verify safety and liveness properties, including common ones in the Petri nets domain, such as boundedness and reachability.

Future work is foreseen to integrate the iopt2pml tool in the IOPT-Tools framework. This should include the definition of a user interface allowing the specification of important model checking parameters and the visualization of simulation and verification results in an easily readable format, backannotated to the original IOPT net model (namely the bound attribute for all places, which is a major convenience to adapt resources when generating execution code).

Another interesting addition will be the generation of Promela specifications from IOPT models with multiple time domains: a distinct Promela process will be created for each time domain. These processes can then communicate using Promela channels, thus taking additional advantage of the spin model checker capabilities.

REFERENCES

[1] D. M. M. Alam and X. He, "A method to analyze high level petri nets using spin model checker," in *SEKE*, 2017.

[2] P. E. S. Barbosa, A. Costa, J. C. A. de Figueiredo, F. Ramalho, L. Gomes, and A. D. dos S., "Modeling complex petri nets operations in the model- driven architecture," in *2009 35th Annual Conference of IEEE Industrial Electronics,* 2009, pp. 4359-4364.

[3] M. Ben-Ari, *Principles of the Spin Model Checker*, 1st ed. Springer, 2008.

[4] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber, "The Petri Net Markup Language: Concepts, Technology, and Tools," in *Proceeding of the 24th International Conference on Application and Theory of Petri Nets*, ser. LNCS, W. van der Aalst and E. Best, Eds., vol. 2679. Eindhoven, Holland: Springer-Verlag, jun 2003, pp. 483-505.

[5] R. David and H. Alla, *Petri Nets & Grafcet; Tools for Modelling Discrete Event Systems*. Prentice Hall International (UK) Ltd, 1992.

[6] ——, *Discrete, Continuous, and Hybrid Petri Nets*, 2nd ed. Springer Publishing Company, Incorporated, 2010.

[7] M. L. Fernández Venero and F. S. Corrêa Da Silva, "Model checking multi-level and recursive nets," *Softw. Syst. Model.*, vol. 16, no. 4, p. 1117-1144, oct 2017, https://doi.org/10.1007/s10270-015-0509-6.

[8] G. Frey and M. Minas, "Editing, Visualizing, and Implementing Signal Interpreted Petri Nets," in *Proceedings of the AWPN 2000, Koblenz*, Oct. 2000, pp. 57-62.

[9] G. Gannod and S. Gupta, "An automated tool for analyzing petri nets using spin," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 2001, pp. 404-407.

[10] C. Girault and R. Valk, *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Berlin, Heidelberg: SpringerVerlag, 2001.

[11] L. Gomes, J. P. Barros, A. Costa, and R. Nunes, "The Input-Output Place-Transition petri net class and associated tools," in *2007 5th IEEE International Conference on Industrial Informatics*, vol. 1, June 2007, pp. 509-514.

[12] L. Gomes, F. Moutinho, and F. Pereira, "IOPT-tools - a web based tool framework for embedded systems controller development using Petri nets," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013, pp. 1-1.

[13] L. Gomes and J. P. Barros, "Refining IOPT Petri nets class for embedded system controller modeling," in *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, 2018, pp. 4720-4725.

[14] H.-M. Hanisch and A. Luder, "A Signal Extension for Petri Nets and its Use in Controller Design," *Fundamenta Informaticae*, vol. 41, no. 4, pp. 415-431, 2000.

[15] G. Holzmann, *Spin Model Checker, the: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, 2003.

[16] R. Pelanek, "Fighting state space explosion: Review and evaluation," in *Formal Methods for Industrial Critical Systems*, D. Cofer and A. Fantechi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 37-52.

[17] F. Pereira, F. Moutinho, and L. Gomes, "IOPT-Tools - towards cloud design automation of digital controllers with Petri nets," in *ICMC'2014 - International Conference on Mechatronics and Control*, 2014.

[18] F. Pereira, F. Moutinho, L. Gomes, and R. Campos-Rebelo, "IOPT Petri net state space generation algorithm with maximal-step execution semantics," in *2011 9th IEEE International Conference on Industrial Informatics*, July 2011, pp. 789-795.

[19] Petri nets tool database. Accessed on (2022, January 14). [Online]. Available: http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/ db.html

[20] PNML.org. Accessed on (2022, January 14). [Online]. Available: http://www.pnml.org

[21] W. Reisig, *Petri nets: an Introduction*. Springer-Verlag New York, Inc., 1985.

[22] O. R. Ribeiro and J. M. Fernandes, "Translating Synchronous Petri nets into PROMELA for verifying behavioural properties," in *2007 International Symposium on Industrial Embedded Systems*, 2007, pp. 266-273.

[23] M. Silva, *Las Redes de Petri: en la Automática y la Informática.* Madrid: Editorial AC Madrid, 1985.

[24] Verifying multi-threaded software with Spin. Accessed on (2022, January 14). [Online]. Available: http://spinroot.com/