MCC: a Tool for Unfolding Colored Petri Nets in PNML Format

Silvano Dal Zilio

MCC: a Tool for Unfolding Colored Petri Nets in PNML Format

Silvano Dal Zilio[1]

[1]LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France

Abstract

MCC is a tool designed for a very specific task: to transform the models of High-Level Petri nets, given in the PNML syntax, into equivalent Place/Transition nets. The name of the tool derives from the annual Model-Checking Contest, a competition of model-checking tools that provides a large and diverse collection of PNML models. This choice in naming serves to underline the main focus of the tool, which is to provide an open and efficient solution that lowers the access cost for developers wanting to engage in this competition.

We describe the architecture and functionalities of our tool and show how it compares with other existing solutions. Despite the fact that the problem we target is abundantly covered in the literature, we show that it is still possible to innovate. To substantiate this assertion, we put a particular emphasis on two distinctive features of MCC that have proved useful when dealing with some of the most challenging colored models in the contest, namely the use of a restricted notion of "higher-order invariant", and the support of a Petri net scripting language.

Keywords— Tools, PNML, High-Level Petri nets, Colored Petri nets

1 Introduction

The Petri Net Markup Language (PNML) [4] is an XML-based interchange format for representing Petri nets and their extensions. One of its main goal is to provide developers of Petri net tools with a convenient, open and standardized format to exchange and store models. While its focus is on openness and extensibility, the PNML spotlights two main categories of models: standard Place/Transition nets (P/T nets), and a class of Colored Petri nets, called High-Level Petri Nets (HLPN), where all types have finite domains and expressions are limited to a restricted set of operators [6, 12].

In this paper we present `mcc`, a tool designed for the single task of *unfolding* the models of High-Level Petri nets, given in the PNML syntax, into equivalent Place/Transition nets. The name of the tool derives from the annual Model-Checking Contest (MCC) [1], a competition of "Petri tools" that makes an extensive use of PNML and that provides a large and diverse collection of PNML models, some of which are colored. Our choice when naming `mcc` was

to underline the main focus of the tool, which is to provide an open and efficient solution that lowers the access cost for developers wanting to engage in the MCC.

We seek to follow the open philosophy of PNML by providing a software that can be easily extended to add new output formats. Until recently, the tool supported the generation of Petri nets in both the TINA [3] (.net) and LOLA [16] formats; but it has been designed with the goal to easily support new tools. To support this claim, we have very recently added a new command to print the resulting P/T net in PNML format. This extension to the code serves as a guideline for developers that would like to extend `mcc` for their need.

The rest of the paper is organized as follows. In Sect. 2, we describe the basic functionalities of `mcc` and give an overview of the PNML elements supported by our tool, we also propose three new classes of colored models that are representative of use cases found in the MCC repository [11]. Next, we describe the architecture of `mcc` and discuss possible applications of its libraries. Before concluding, we compare `mcc` with other existing solutions. Despite the fact that the problem we target is abundantly covered in the literature, we show that it is still possible to innovate. We describe two particular examples of optimizations that have proved useful when dealing with some of the most challenging colored models in the contest, namely the use of a restricted notion of "higher-order invariant", and the support of a Petri net scripting language.

2 Installation, Usage and Supported PNML Elements

The source code of `mcc` is made freely available on GitHub[1] and is released as open-software under the CECILL-B license; see https://github.com/dalzilio/mcc. The code repository also provides a set of PNML files taken from the open collection of models from the MCC [11].These files are provided in the source code repository to be used for benchmarking and continuous testing. The tool can also be easily compiled, from source, on any computer that provides a recent distribution of the Go programming language.

Basic usage.

Tool `mcc` is a command-line application that accepts three primary subcommands: `hlnet`, `lola` and `pnml`. In this paper, we focus on the `mcc hlnet` command, that generates a Petri net file in the TINA *net* format [3]. Similarly, commands `lola` and `pnml` generate an equivalent output but targeting, respectively, the LoLa [16] and PNML formats for P/T nets.

We follow the UNIX philosophy and provide a small program, tailored for a precise task, that can be composed using files, pipes and shell script commands to build more complex solutions. As it is customary, option `-h` prints a usage message listing the parameters and options accepted by the command.

The typical usage scenario is to provide a path to a PNML file, say `model.pnml`, and invoke the tool with a command such as "`mcc hlnet -i model.pnml`". By default, the result is written in file

[1] See https://github.com/dalzilio/mcc for the source code. Binaries for Windows, Linux and MacOS are available at https://github.com/dalzilio/mcc/releases

`model.net`, unless option `-o` or `--name` is used. We discuss some of the other options of `mcc` in the sections that follow.

PNML elements supported by MCC.

The input format supported by `mcc` covers most of the PNML syntax defined in the ISO/IEC 15909-2 standard, which corresponds to the definition of HLPN.

High-Level Petri nets form a subset of colored nets defined by a restriction on the types and expressions that are allowed in a net [4, 10]. The core action language of HLPN is a simple, first-order declarative language organized into categories for types, values and expressions. Essentially, HLPN is built around a nominal type system where possible ground types include a constant for "plain tokens" (`dot`), and three different methods for declaring finite, ordered enumeration types (finite, cyclic, and integer range). The type system also includes "product types", used for tuples of values, and a notion of *partition elements*, which are (named) subsets of constants belonging to the same type.

Expressions are built from values and operations and describe multisets of colors, which act as the marking of places. For instance, the language include operators `add` and `subtract`, that correspond to multiset union and difference. The language also includes a notion of *patterns*, which are expressions that includes variables (in a linear way), and of *conditions*, which are boolean expressions derived from a few comparison operators. A simple way to describe the subset of the PNML standard supported in `mcc` is to list the XML elements supported in each of these categories (most of the element names are self-explanatory):

types	::=	`dot` | `cyclicenumeration` | `finiteenumeration`
	|	`finiteintrange` | `productsort`
	|	`partition` | `partitionelement`
values	::=	`dotconstant` | `feconstant` | `finiteintrangeconstant`
expressions	::=	`variable` | `successor` | `predecessor` | `tuple`
	|	`all` | `add` | `subtract`
conditions	::=	`or` | `and` | `equality` | `inequality`
	|	`lessthan` | `greaterthan`
	|	`greaterthanorequal` | `lessthanorequal`

The `mcc` tool, in its latest version, supports all the operators used in models of the Model-Checking Contest. To better understand this fragment, we give three examples of HLPN that can be expressed using these constructs, see Fig. 1 to 3. Each of these examples illustrate an interesting class of parametric models found in the MCC and will be useful later to discuss the strengths and weaknesses of our approach. None of these models are part of the MCC repository (yet), but their PNML specification can be found in the `mcc` source code repository.

Three representative examples.

Our first example, Fig. 1, illustrates the use of colors to model a complex network topology. While Diffusion is not part of the MCC repository, it is the colored equivalent of model Grid2d; it is also the main benchmark in [14]. In this model, values in the place Grid are of the form (x, y), with $x, y \in 0..4$. Hence we can interpret colors as cells on a 5×5 grid and values as "tokens" in these cells.

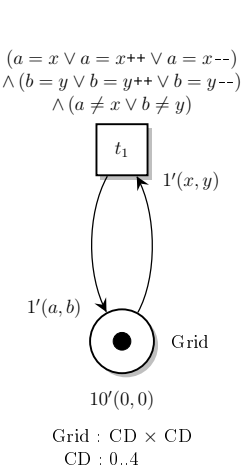

Figure 1: Diffusion

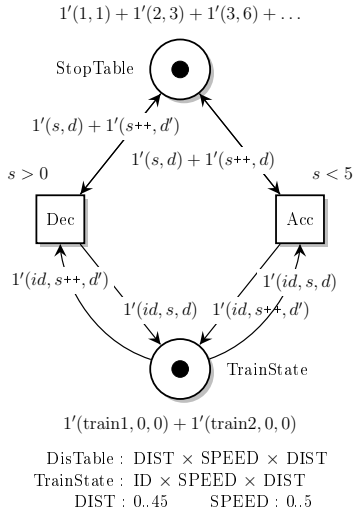

Figure 2: TrainTable

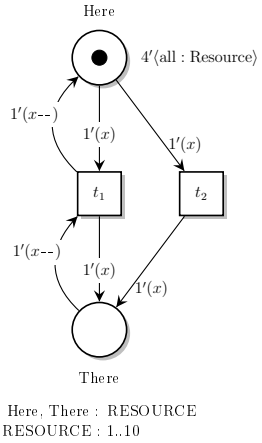

Figure 3: Resource Swap

Tokens can move to an adjacent cell by firing transition t_1 but cannot cross borders. (In our diagrams we use ++ for `successor`, $>$ for `greaterthan`, and $+$ for `add`.) All the behavior is concentrated on the condition associated with t_1. Since the expression contains four variables; we potentially have $|\text{CD}|^4$ different ways to enable t_1.

TrainTable, is an example where colors are used to simulate complex relations between data values. Place StopTable is initialized with a list of pairs associating, to each (integer) speed in 0..5, the safety distance needed for a train to stop. Hence TrainTable tabulates a non-linear constraint between speed and distance. Place TrainState stores the current state of two different trains. Each time a train accelerate (Acc), or decelerate (Dec), the safety distance is updated. TrainTable is a simplified version of the BART model. We can make this model more complex by storing the distance traveled instead of the safety distance (Traintable-Dist); or even more complex by storing both values (TrainTable-Stop+Dist).

Our last example, Swap, is typical of systems built from the composition of multiple copies of the same component and where interactions are limited to "neighbors". The model obeys some interesting syntactical restrictions: it does not use conditions on the transitions and inscriptions on arcs are limited to two patterns, x or x--. This is representative of many models, such as the celebrated *Dining Philosophers* example (known as Philosopher in the MCC).

3 Architecture of MCC

The `mcc` tool is a standalone Go program built from three main software components[2] (called *packages* in Go): `pnml`, `hlnet`, and `corenet`. Basically, the architecture of `mcc` is designed to resemble that of a compiler that translates

[2]See the documentation at https://godoc.org/github.com/dalzilio/mcc.

high-level code (HLPN) into low-level instructions (P/T net). We follow a traditional structure with three stages where: `pnml` corresponds to the front-end (responsible for syntax and semantics analysis); `hlnet` provides the intermediate representation; and `corenet` is the back-end, which includes functions for unfolding an `hlnet` and for "code generation". A last package, `cmd`, contains boilerplate code for parsing command-line parameters and manage inputs/outputs.

Each of these packages is interesting taken separately and can be reused in other applications. Package `pnml`, for instance, includes all the types and functions necessary for parsing a PNML file: it defines a `pnml.Decoder`, which encapsulates an efficient, UTF-8 compatible XML parser and can provide meaningful error messages in case of problems. The `hlnet` package, for its part, defines the equivalent of an Abstract Syntax Tree data structure for PNML files. Both of these packages can be easily reused in programs that need to consume PNML data. In particular, they can help build a standalone PNML parser with good error handling.

Finally, package `corenet` contains the code for unfolding an `hlnet.Net` value into a `corenet.Net`, which is a simple, graph-like data structure representing a P/T net. The package also contains the functions for marshalling a core net structure into other formats; see function `corenet.LolaWrite` for an example. More than compliance with the standard, `mcc` takes care of many of the "idiosyncrasies" in the way PNML model are written in the MCC. For instance we consider the case where `numberof` does not declare a multiplicity.

A tool developer that would like to adopt `mcc` to generate a "core net", using his own format, only needs to provide a similar `Write` function. In the case of the `pnml` subcommand, that was added on the last release of the tool, one hundred line of codes were enough to add the ability to generate PNML files. A figure that is similar to what we observed with the `lola` subcommand.

Using package hlnet for drawing Colored nets.

Package `hlnet` also includes a function to output a textual representation of an AST that is compatible with TINA's net syntax. It generates a net that includes all the places and transitions in a colored model as if it was a P/T net and uses labels to display the expressions associated with transitions and the initial marking of places. The net also includes "nodes" (comments similar to sticky notes) for information about types, variables and arc inscriptions. The result can be displayed and modified with `nd`, the *NetDraw* graphical editor distributed with TINA. We show such an example in the screen capture of Fig. 4, which is obtained by using `mcc` with option `--debug` on the HLPN model *TrainTable* of Fig. 2.

While modifications cannot be saved back into PNML, this capability is still useful to inspect colored model (and is often more accurate than the graphical information included in the "cover flow" provided with every model). We can also use the export function included in `nd` to generate a LaTeX(*tikz*) representation of the net. This is what we used to generate an initial version of the diagrams that appear in Fig. 1 to 3 of this paper.

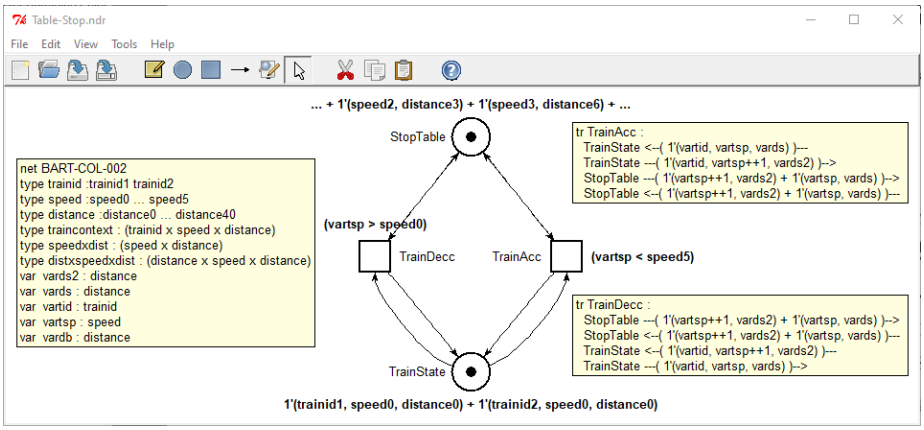

Figure 4: Result of option debug on model TrainTable, displayed in `nd`

4 Comparison with other Tools

The problem of (efficiently) unfolding colored models has been abundantly covered in the literature and many of the proposed algorithms have been implemented. We can cite the works of Mäkelä, with his tool MARIA [15]; of Heiner et al. with Marcie [14, 9]; or the work of Kordon et al. [13], that makes a clever use of decision diagrams in order to compute results for very large instances. This approach is implemented in CPN-AMI [8] and provides the reference for P/T instances derived from Colored model in the MCC. All these works provide good motivations for why it may be useful to unfold a HLPN instead of trying to analyze it directly.

We decided to compare `mcc` with three tools that participated in the Model-Checking Contest: Tapaal [7] (with its `verifypn` tool); Marcie [9] (with the `andl_converter`); and GreatSPN [2] (that includes a Java based unfolding tool in its editor). Since each tool is tailored for a different toolchain—and therefore generate very different results—it is difficult to make a precise comparison of the performances. Hence these results should only be interpreted as a rough estimate. For instance, Tapaal is the only tool in this list that do not output the unfolded net on disk. This means that its computation time do not include the time spent marshalling the result and printing it on file.

Unfolding algorithm.

We follow a very basic strategy. For each place p, of type say T, we create one instance of p for any value that inhabits T. (This part is common to most of the existing unfolding algorithms.) For each transition, t, we consider the set of variables occurring in the inscription of arcs attached to it (its *environment*). Then we enumerate all possible valuations of the environment and keep only those that satisfy the conditions associated with t.

Our main optimization is to follow a "constraint solving" approach where we can avoid enumerating a large part of the possible assignments when we know that the condition cannot be satisfied. For instance a subexpression in a conjunction is falsified. This is a less sophisticated approach than those described

Model	Places	Trans.	MCC	Tapaal	Marcie	GSPN
GlobalResAllocation-07	133	291 067	**1.7**	3	14.4	22.3
GlobalResAllocation-11	297	2.10^6	**15.1**	29.3	144.6	—
DrinkVendingMachine-16	192	10^6	15.5	10.7	52.8	108.1
DrinkVendingMachine-24	288	8.10^6	97.1	95.9	—	—
PhilosophersDyn-50	2 850	255 150	**1**	2.1	11.1	15.7
PhilosophersDyn-80	6 960	10^6	**4.1**	9.9	55.9	61.0
Diffusion-D050	2 500	8 109	14.5	**0.6**	4.1	—
Diffusion-D100	10 000	31 209	243.3	**8.6**	31.3	—
TokenRing-100	10 201	10^6	**4**	8.2	33.5	49.3
TokenRing-200	40 401	8.10^6	**67.4**	166.1	—	—
SafeBus-50	5 606	140 251	14.2	**1.4**	6.2	25.1
SafeBus-80	13 766	550 801	89.5	**7**	20.6	133.1
TrainTable-Dist	722	602	**1.4**	12.6	59.5	69.4
TrainTable-Stop+Dist	728	602	**2.1**	—	—	—
BART-002	764	646	**3.1**	—	—	—
BART-060	15 032	19 380	**3.2**	—	—	—
SharedMemory-000200	40 801	80 400	**0.3**	1.7	2.6	5.1
SharedMemory-001000	10^6	2.10^6	**8.9**	—	60.3	160.2
SharedMemory-002000	4.10^6	8.10^6	**55.3**	—	—	—
FamilyReunion-L800	2.10^6	2.10^6	**5.5**	—	84.8	143.0
FamilyReunion-L3000	28.10^6	27.10^6	**89.5**	—	—	—
Swap-P010000	20 000	20 000	0.1/0.6	0.4	0.9	5.0
Swap-P100000	200 000	200 000	0.4/4.8	26.1	15.7	—

Table 1: Execution time (in s) when unfolding complex PNML instances

in existing works [14, 15]. For instance, we do not try to detect particular kind of expressions where an unification-based approach could have better performances.

Typically, we should perform badly with instances similar to Diffusion, where we may fail to cut down the size of our search space. On the other hand, our approach is not hindered when we need to deal with complex expressions, such as with TrainTable, that involve at the same time tuples, `successor`, and `add`. Actually, our approach may also work with nonlinear patterns, where the same variable is reused in the same expression. Finally, all the algorithms should work equally well on examples like Swap, because of its simplicity.

Even if our approach is quite rustic, our experiments shows that this does not hinder our performances. This may be because few of the colored instances in the MCC fall in the category where clever algorithms shine the most.

Benchmarks.

We selected instances, with a processing time of over a second, from different models listed in the MCC repository [11] and from the three examples in Sect. 3. We give the results of our experiment in Table 1. Computations were performed with a time limit of 5 min and a limit of 16 GB of RAM. In each case we give

the number of places and transitions in the unfolded net and highlight the best time (when there is a significant difference). An absence of values (—) means a timeout.

Tapaal shows very good performances on many instances and significantly outperforms `mcc` for models SafeBus and Diffusion. On the opposite, we see that many instances can only be processed with `mcc`. This is the case with model BART (even with a time limit of 1 h). Other interesting examples are models SharedMemory and FamilyReunion. This suggest that we could further improve our tool by including some of the optimizations used in `verifytpn` that seems to be orthogonal to what we have implemented so far. We describe two of the optimizations performed by `mcc` below.

Actually, sheer performance is not our main goal. We rather seek to return a result for all the colored instances used in the MCC in a sensible time. (Who needs to unfold a model too big to be analyzed anyways?) At present, there are 193 *instances* of Colored nets in the MCC repository, organized into 23 different classes, simply referred to as *models*. We can return a result for 184 of these instances, with the condition of the competition. Moreover, to the best of our knowledge, `mcc` is the only tool able to return results (for at least one instance) in all the models. But some instances, like DrinkVendingMachine-48, should stay out of reach for a long time, mostly due to memory space limitations.

Use of colored invariants.

The first "additional" optimization added to `mcc` explains our good result on models such as BART. The idea is to identify invariant places; meaning places whose marking cannot be changed by firing a transition. A sufficient condition for place p to be invariant is if, for every transition t, there is an arc with inscription e from p to t iff there is an arc from t to p with inscription e' equivalent to e. (Syntactical equality between e and e' is enough for our purpose.) We say that such places are *stable*; a concept equivalent to "test arcs" for an HLPN. This is the case, for example, for place StopTable in model TrainTable. When a place is stable, we know that its marking is fixed. This can significantly reduce the set of assignments that need to be enumerated.

Use of a Petri scripting language.

The effect of our second improvement can be observed in model Swap (Fig. 3). In this case, like with model Philosopher of the MCC, it is possible to detect that the unfolded net is the composition of n copies of the same component; where $n = |\text{Resource}|$. Each component x (with $x \in \text{Resource}$) is a net with a local copy of the places. As for the transitions, we need to keep one copy for each "local interactions" (such as t_2) and two copies for distant interactions (t_1): one for the pair of components $(x\text{--}, x)$; the other for the pair $(x, x\text{++})$. Since type Resource is a cyclic enumeration—this is basically a "scalar set"—the composition of all these components form a ring architecture.

Our tool is able to recognize this situation automatically. In such a case we output a result that uses the *TPN format*, a scripting language for Petri net supported by the TINA toolchain. This scripting language includes operators for make copies of net; add and rename places and transitions; compute the product

or chaining of nets; ... It also provides higher-order composition patterns, such as pools or rings of components. We use the latter for model Swap.

Our benchmarks of Table 1 include the results on two instances of model Swap. The computation time for `mcc` (the first value) is mostly independent from the size of the instance (the only difference is in parsing the PNML file.) This result conceals a much more complex realty. Indeed, a tool that consumes a TPN script still needs to "expand it". This is why we added a second value in Table 1, which is the time taken to generate the result using the `mcc pnml` command. For information, the size of the PNML result for model Swap-P100000 is 99 MB, while it is only 200 bytes for the TPN version.

5 Conclusion

Tool `mcc` is a new solution to an old problem. It is also an unassuming tool, that focuses on a single, very narrow task. Nonetheless, we believe that it can still be of interest for the Petri net community, and beyond, by enriching the PNML ecosystem. As a matter of fact, there has been a total of 26 verification tools to participate to the MCC since its beginning [1], not all "Petri tools". Many of these tools could benefit from using `mcc`.

Development on `mcc` started in 2017, as a pet project for studying the suitability of the Go programming language to develop formal verification tools. Our assessment in this regard is very positive: performances are competitive with regards to C++, with good code productivity and mature software libraries; building executables for multiple platforms and distributing code is easy; ...

Since then, work has progressed steadily in-between each edition of the MCC, with a focus on stability of the tool and on compliance with the PNML standard. Three iterations later, `mcc` is now sufficiently mature to gain more exposure and provides a good showcase for an efficient PNML parser written in Go. But `mcc` is more than that. First, `mcc` was designed to lower the work needed by developers wanting to engage in the Model-Checking Contest. It also provides new features, such as the ability to display an interactive (read-only), graphical view of a PNML model; see Fig. 4. Finally, it provides a testbed for evaluating new unfolding algorithms (we show two of these ideas in Sect. 4).

In the future, we plan to enrich `mcc` by computing interesting properties of the models during unfolding. For example by computing invariants or by finding sets of places that can be clustered together. In that respect, the possibility to identify HLPN that can be expressed using a "Petri net scripting language" could potentially leads to new advances. For example to simplify the detection of symmetries, something that we have been working on recently in the context of Time Petri nets [5].

References

[1] Amparore, E., Berthomieu, B., Ciardo, G., Dal Zilio, S., Gallà, F., Hillah, L.M., Hulin-Hubard, F., Jensen, P.G., Jezequel, L., Kordon, F., Le Botlan, D., Liebke, T., Meijer, J., Miner, A., Paviot-Adet, E., Srba, J., Thierry-Mieg, Y., van Dijk, T., Wolf, K.: Presentation of the 9th edition of the model checking contest. In: Tools and Algo-

rithms for the Construction and Analysis of Systems. Springer (2019). https://doi.org/10.1007/978-3-662-58381-4_9

[2] Amparore, E.G., Balbo, G., Beccuti, M., Donatelli, S., Franceschinis, G.: 30 years of GreatSPN. In: Principles of Performance and Reliability Modeling and Evaluation. Springer (2016). https://doi.org/10.1007/978-3-319-30599-8_9

[3] Berthomieu, B., Ribet, P.O., Vernadat, F.: The tool TINA–construction of abstract state spaces for Petri nets and Time Petri nets. International journal of production research **42**(14), 2741–2756 (2004). https://doi.org/10.1080/00207540412331312688

[4] Billington, J., Christensen, S., Van Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri Net Markup Language: concepts, technology, and tools. In: International Conference on Application and Theory of Petri Nets. Springer (2003). https://doi.org/10.1007/3-540-44919-1_31

[5] Bourdil, P.A., Berthomieu, B., Dal Zilio, S., Vernadat, F.: Symmetry reduction for time Petri net state classes. Science of Computer Programming **132**(2) (2016). https://doi.org/10.1016/j.scico.2016.08.008

[6] Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: On well-formed coloured nets and their symbolic reachability graph. In: High-level Petri nets. Springer (1991). https://doi.org/10.1007/978-3-642-84524-6_13

[7] David, A., Jacobsen, L., Jacobsen, M., Jørgensen, K.Y., Møller, M.H., Srba, J.: Tapaal 2.0: Integrated development environment for timed-arc petri nets. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_36

[8] Hamez, A., Hillah, L., Kordon, F., Linard, A., Paviot-Adet, E., Renault, X., Thierry-Mieg, Y.: New features in CPN-AMI 3: focusing on the analysis of complex distributed systems. In: Sixth International Conference on Application of Concurrency to System Design (ACSD'06). IEEE (2006). https://doi.org/10.1109/ACSD.2006.15

[9] Heiner, M., Rohr, C., Schwarick, M.: Marcie–model checking and reachability analysis done efficiently. In: International Conference on Applications and Theory of Petri Nets and Concurrency. Springer (2013). https://doi.org/10.1007/978-3-642-38697-8_21

[10] Hillah, L., Kordon, F., Petrucci, L., Treves, N.: PN standardisation: a survey. In: International Conference on Formal Techniques for Networked and Distributed Systems. Springer (2006). https://doi.org/10.1007/11888116_23

[11] Hillah, L., Kordon, F.: Petri Nets Repository: A tool to benchmark and debug Petri net tools. In: Application and Theory of Petri Nets and Concurrency. LNCS, vol. 10258. Springer (2017). https://doi.org/10.1007/978-3-319-57861-3_9

[12] Jensen, K.: Coloured petri nets. In: Petri nets: central models and their properties, pp. 248–299. Springer (1987). https://doi.org/10.1007/BFb0046842

[13] Kordon, F., Linard, A., Paviot-Adet, E.: Optimized colored nets unfolding. In: International Conference on Formal Techniques for Networked and Distributed Systems. Springer (2006). https://doi.org/10.1007/11888116_25

[14] Liu, F., Heiner, M., Yang, M.: An efficient method for unfolding colored Petri nets. In: Winter Simulation Conference (WSC). IEEE (2012). https://doi.org/10.1109/WSC.2012.6465203

[15] Mäkelä, M.: Optimising enabling tests and unfoldings of algebraic system nets. In: International Conference on Application and Theory of Petri Nets. Springer (2001). https://doi.org/10.1007/3-540-45740-2_17

[16] Schmidt, K.: Lola a low level analyser. In: Int. Conference on Application and Theory of Petri Nets. Springer (2000). https://doi.org/10.1007/3-540-44988-4_27