Punf Documentation and User Guide
Version 8.51 (parallel)

Victor Khomenko

Newcastle University, UK

June 2012

1 Legal information

See the file LICENSE.PUNF for the copyright notice and legal information.

2 Installation

No special installation is required. Just copy the executable into a folder listed in the value of the PATH variable.

3 Description

PUNF is a Petri net unfolder/unraveller, i.e.,it takes a Petri net and produces a finite complete prefix of its unfolding, or a finite and complete merged process. Such a prefix/merged process can be used for efficient model checking (see, e.g., [14–16, 18, 22–24, 26, 28, 29, 31–38]). Currently the following classes of Petri nets are supported:

Low-level nets in the FORMAT_N or FORMAT_N2 (.ll_net) formats supported by the PEP tool.

Signal Transition Graphs in the STG (.stg or .g) format.

High-level nets (M-nets) in the FORMAT_N2 (.hl_net) format supported by the PEP tool.

The produced prefix is in the .mci format supported by the PEP tool. The methods used for implementing PUNF are described in [19, 20, 22, 25, 27].

3.1 Low-level nets

Currently only 1-safe nets without weighted/read/inhibitor arcs are supported.

Remark 1 *Currently* PUNF *does NOT check whether the given net is safe, and if you pass a non-safe Petri net to it then an incorrect prefix is likely to be produced. Sometimes, though,* PUNF *can detect non-safeness, e.g., in the following cases:*

- *the initial marking is unsafe;*

- *the final marking of some local configuration in the prefix is unsafe;*
- PUNF *detects that two configurations are equal w.r.t. the ERV adequate order (see [8–10, 27, 36]).*

3.2 Signal transition graphs

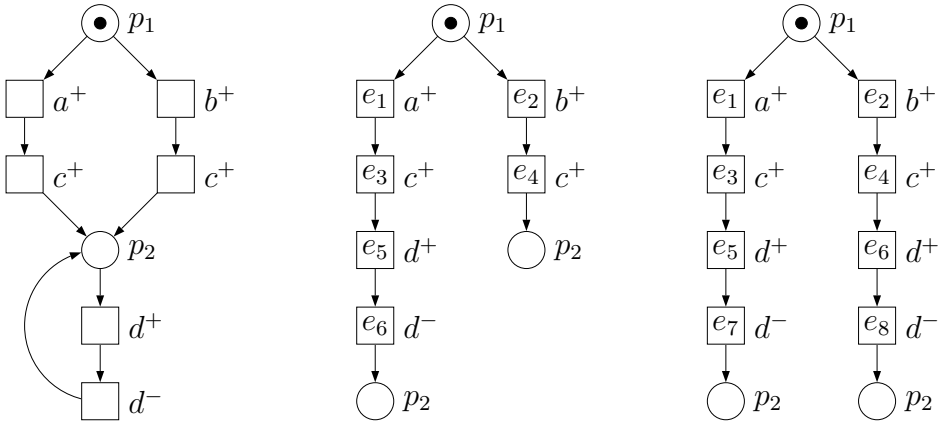
Currently STG must satisfy the following restrictions:

- The underlying Petri net must satisfy the restrictions formulated in Section 3.1.
- The STG must be *consistent* (see [5, 22, 28, 30, 37, 38, 40]). Note that PUNF does NOT check the consistency.
- The STG should contain neither 'don't care' (e.g., $a*$) nor level (e.g., $a\hat{}0$, $a\hat{}1$) transitions.
- Boolean guards on transitions are not supported.
- Signals with free return-to-zero are not supported.
- Channels are not supported.
- The '.capacity' clause is ignored.

The clauses '.initial state', '.concurrent', '.late', '.ordered', '.time', '.slow', '.slowenv', and '.implement' are not essential for building a prefix and ignored by PUNF.

The main distinctions between STG unfolding prefixes and the usual ones are:

- The signal codes are taken into account as a part of final state when checking the cut-off condition (see Figure 1).
- An event e corresponding to a dummy signal can only be declared cut-off if its corresponding configuration C is a subset of $[e]$, and $[e] \setminus C$ contains only dummies. This allows to determine the set of signals enabled (perhaps, via a chain of dummies) by a configuration of the prefix. This behaviour is automatically specified if the STG contains local (i.e., output or internal) signals, but can be suppressed with the `-d` command-line option.



(a) **(b)** **(c)**

Figure 1: A consistent STG **(a)**, the 'usual' finite and complete prefix of its underlying net **(b)**, and the 'proper' STG unfolding prefix **(c)**. Note that in the 'usual' prefix e_4 is a cut-off event and a reachable state with the code $\{a \mapsto 0, b \mapsto 1, c \mapsto 1, d \mapsto 1\}$ is not represented.

3.3 High-level nets

As an experimental feature, PUNF can unfold high-level nets, viz. M-nets (see [1,2,12,13,22]). The resulting prefix is very similar to that generated from the low-level expansion of an M-net, so all the verification tools employing prefixes can be re-used. This feature is useful for verification of data-intensive systems, since net expansions often blow up in such a case. The direct unfolding approach avoids building the intermediate low-level net, and thus solves this problem. Moreover, some extensions (e.g., infinite types) are now supported.

Currently PUNF supports the following token types:

- The black token • (`dot`);
- Booleans `ff` and `tt`;
- Integers;
- Tuples and sets (including nested ones) built upon the described types.

Channels and stacks are not supported yet.

If the type of a place is not specified, PUNF assumes that this place can hold any token, implementing thus the possibility of having infinite types. Note that the resulting prefix can be infinite in this case, so the termination cannot always be guaranteed. But if it is finite, it is possible to find the exact type of each place of the M-net.

The following restrictions must be satisfied:

- The M-net must be *ordinary,* i.e., the arcs cannot be labelled by more than one variable.
- The M-net must be *strictly safe,* i.e., no reachable marking puts more than one token in any place. Note the distinction between *safe* and strictly safe M-nets: safe M-nets don't put several tokens *of the same colour* in the same place, while strictly safe M-nets don't put several tokens in the same place even if all the tokens have different colours. The low-level expansion of a safe M-net is a safe low-level net. In the low-level expansion of a strictly safe M-net all places corresponding to the same low-level place are mutually exclusive.
- All the variables appearing in the guard of a transition or on its outgoing arcs must be *assigned a value* (see below).

3.4 The syntax of transition guards

Currently PUNF does not have a constraint solver implemented. Instead, it processes a transition guard as a sequence of assignments and predicates with the following syntax, subject to certain contextual restrictions formulated later in this section:

$$\begin{aligned} \textsc{Guard} &\rightarrow \textsc{Clause} \;\&\; \textsc{Guard} \mid \textsc{Clause} \\ \textsc{Clause} &\rightarrow \textsc{Predicate} \mid \textsc{Assignment} \\ \textsc{Predicate} &\rightarrow \textsc{Expression} \\ \textsc{Assignment} &\rightarrow \mathit{var} = \textsc{Expression} \mid \textsc{Expression} = \mathit{var} \end{aligned}$$

Each EXPRESSION is composed of constants and variables with the help of parentheses, the n-tuple constructor $(\cdot, \ldots, \cdot)$, the set constructor $\{\cdot, \ldots, \cdot\}$, the ternary conditional operator $\texttt{if}(\cdot,\cdot,\cdot)$, and the unary and binary operators listed in Table 1. For convenience, the following synonyms are defined:

Operators	Math notation	Explanation
`|`	$\vee$	logical disjunction
`&`	$\wedge$	logical conjunction
`!`	$\neg$	the unary logical negation
`=, #, <, <=, >, >=`	$=, \neq, <, \leq, >, \geq$	relations
`:, /:, <:, /<, <<:, /<<:, >:, />:, >>:, />>:`	$\in, \notin, \subset, \not\subset, \subseteq, \not\subseteq, \supset, \not\supset, \supseteq, \not\supseteq$	relations involving sets
`\/, /\, \`	$\cup, \cap, \setminus$	set union, intersection and difference
`+, -`	$+, -$	addition and substraction
`*, /, %`	$\cdot, /, \mathrm{mod}$	multiplication, division, and remainder of division
`-, card`	$-, \lvert\cdot\rvert$	unary minus and set cardinality

Table 1: Unary and binary operators (in order of decreasing precedence).

Operator	`|`	`&`	`!`	`#`	`%`
Synonym	`or`	`and`	`not`	`/=`	`mod`

Note that the conditional operator `if(·,·,·)` and the operations involving sets are an extension to the syntax provided by the PEP tool. Another extension is that it is allowed to label transition-place arcs by arbitrary valid expressions rather than only by variables. The place-transition arcs must be labelled by *patterns*, i.e., expressions built from constants and variables with the help of the brackets and the n-ary tuple constructors. Figure 2 illustrates the syntax.

Definition 1 *A variable* `v` *occurring in a* Clause *is* assigned a value *if either* `v` *appears in the labels of the transition's incoming arcs or it was the* `var` *in an* Assignment *textually preceding the* Clause *in the* Guard.

With this definition, the contextual restrictions can be formulated as follows:

- All variables occurring in an Expression must have been assigned a value.
- In the rule for Assignment, `var` must be a variable which is not assigned a value (otherwise this would be interpreted as a Predicate rather than an Assignment).

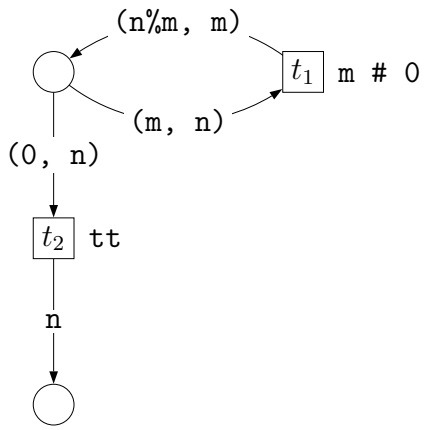


Figure 2: An M-net system modelling the Euclid's algorithm for computing the greatest common divisor of two non-negative integers.

For example, the CLAUSE `x=y` can be interpreted in three different ways, depending on the context:

- if both `x` and `y` have already been assigned values then `x=y` is a PREDICATE checking whether these values are equal;
- if only `x` has been assigned a value then `x=y` is an assignment to `y`, and `y` will be considered as having a value in the rest of the guard;
- if only `y` has been assigned a value then `x=y` is an assignment to `x`, and `x` will be considered as having a value in the rest of the guard;
- if neither `x` nor `y` has been assigned a value then `x=y` is a syntax mistake.

For example, the guard `x'='x+1 & z='x*'y & z=2*'y+'x`, where `'x` and `'y` are the variables appearing in the labels of the transition's incoming arcs, is parsed in the following way. Since `x'` is not assigned a value in the first CLAUSE, the it is interpreted as an assignment, and `x'` gets assigned a value (note that `'x` and `x'` are different variables). Similarly, `z` is assigned a value in the second CLAUSE. The third CLAUSE is interpreted as a PREDICATE since `z` has already been assigned a value. But the guard `x'='x+1 & z+1=2*'y+'x & z='x*'y` would cause a syntax error, since at the time the second CLAUSE is parsed `z` has not been assigned a value yet.

These rules might seem too restrictive, but in practice it is almost always possible to transform transition guards into a form acceptable for PUNF. The

conditional operator `if(·,·,·)` often comes in handy for this purpose. For example, consider the guard `'x=1 & z=5*'y | 'x#1 & z=7*'y`, where `'x` and `'y` are the variables appearing in the labels of the transition's incoming arcs and thus are assigned values. It would be interpreted as a `Predicate` (since `|` is its main operator) and cause a syntax error. In order to make the guard acceptable for PUNF one can rewrite it as `z=if('x=1, 5*'y, z=7*'y)`.

The EXPRESSION's labelling the transition's outgoing arcs are parsed after the guard, so all the variables which have been assigned values can be used there.

4 Command-line parameters

Help on command-line parameters can be obtained by running

```
punf -h
```

5 Bug reports

Please, submit bug reports to `Victor.Khomenko@ncl.ac.uk` with the original Petri net and a short description of the problem. We do not guarantee prompt bug fixes, though we will try our best.

References

[1] E. Best, H. Fleischhack, W. Fraczak, R. Hopkins, H. Klaudel, and E. Pelz: A Class of Composable High Level Petri Nets. Proc. of *Application and Theory of Petri Nets (ATPN'1995)*, G. DeMichelis and M. Diaz (Eds.). Springer-Verlag, Lecture Notes in Computer Science 935 (1995) 103–120.

[2] E. Best, H. Fleischhack, W. Fraczak, R. Hopkins, H. Klaudel, and E. Pelz: An M-net Semantics of $B(PN)^2$. Proc. of *International Workshop on Structures in Concurrency Theory (STRICT'1995)*, J. Desel (Ed.). Berlin (1995) 85–100.

[3] E. Best and B. Grahlmann: PEP. *Documentation and User Guide. Version 1.4*. Manual (1995).

[4] E. Best and B. Grahlmann: PEP— More Than a Petri Net Tool. Proc. of *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'1996)*, T. Margaria and B. Steffen (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1055 (1996) 397–401.

[5] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev: PETRIFY: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Transactions on Information and Systems* E80-D(3) (1997) 315–325.

[6] J. Engelfriet: Branching Processes of Petri Nets. *Acta Informatica* 28 (1991) 575–591.

[7] J. Esparza: Decidability and Complexity of Petri Net Problems — an Introduction. In: *Lectures on Petri Nets I: Basic Models*, W. Reisig and G. Rozenberg (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1491 (1998) 374–428.

[8] J. Esparza and S. Römer: An Unfolding Algorithm for Synchronous Products of Transition Systems. Proc. of *International Conference on Concurrency Theory (CONCUR'1999)*, J. C. M. Baeten and S. Mauw (Eds.). Invited paper, Springer-Verlag, Lecture Notes in Computer Science 1664 (1999) 2–20.

[9] J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. Proc. of *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'1996)*, T. Margaria and B. Steffen (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1055 (1996) 87–106.

[10] J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design* 20(3) (2002) 285–310.

[11] J. Esparza and C. Schröter: Reachability Analysis Using Net Unfoldings. Proc. of *Workshop on Concurrency, Specification and Programming (CS&P'2000)*, H. D. Burkhard, L. Czaja, A. Skowron, and P. Starke (Eds.). Informatik-Bericht 140(2). Humboldt-Universitat zu Berlin (2000) 255–270.

[12] H. Fleischhack, B. Grahlmann: A Petri Net Semantics for $B(PN)^2$ with Procedures which Allows Verification. Technical Report 21, Universität Hildesheim (1996).

[13] H. Fleischhack, B. Grahlmann: A Petri Net Semantics for $B(PN)^2$ with Procedures. Proc. of *2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'1997)*, IEEE Computer Society Press (1997) 15–27.

[14] K. Heljanko: Deadlock Checking for Complete Finite Prefixes Using Logic Programs with Stable Model Semantics (Extended Abstract). Proc. of *Workshop on Concurrency, Specification and Programming (CS&P'1998)*, Informatik-Bericht 110. Humboldt-Universitat zu Berlin (1998) 106–115.

[15] K. Heljanko: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets. Proc. of *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'1999)*, Springer-Verlag, Lecture Notes in Computer Science 1579 (1999) 240–254.

[16] K. Heljanko: Deadlock and Reachability Checking with Finite Complete Prefixes. Technical Report A56, Laboratory for Theoretical Computer Science, HUT, Espoo, Finland (1999).

[17] K. Heljanko: Minimizing Finite Complete Prefixes. Proc. of *Workshop on Concurrency, Specification and Programming (CS&P'1999)*, Informatik-Bericht, Humboldt-Universitat zu Berlin (1999) 83–95.

[18] K. Heljanko: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets. *Fundamentae Informaticae* 37(3) (1999) 247–268.

[19] K. Heljanko, V. Khomenko, and M. Koutny: Parallelisation of the Petri Net Unfolding Algorithm. Technical Report CS-TR-733, Department of Computing Science, University of Newcastle (2001).

[20] K. Heljanko, V. Khomenko, and M. Koutny: Parallelisation of the Petri Net Unfolding Algorithm. Proc. of *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*

(TACAS'2002), Springer-Verlag, Lecture Notes in Computer Science 2280 (2002) 371–385.

[21] K. Jensen: *Colored Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. EATCS Monographs on Theoretical Computer Science, Springer-Verlag (1992).

[22] V. Khomenko: *Model Checking Based on Petri Net Unfolding Prefixes*. PhD Thesis, Department of Computing Science, University of Newcastle (2003).

[23] V. Khomenko, A. Kondratyev, M. Koutny, and V. Vogler: Merged Processes — a New Condensed Representation of Petri Net Behaviour. *Acta Informatica* 43(5) (2006) 307–330.

[24] V. Khomenko and M. Koutny: Verification of Bounded Petri Nets Using Integer Programming. *Formal Methods in System Design* 30(2) (2007) 143–176.

[25] V. Khomenko and M. Koutny: Towards An Efficient Algorithm for Unfolding Petri Nets. Proc. of *International Conference on Concurrency Theory (CONCUR'2001)*, P. G. Larsen and M. Nielsen (Eds.). Springer-Verlag, Lecture Notes in Computer Science 2154 (2001) 366–380.

[26] V. Khomenko and M. Koutny: Branching Processes of High-Level Petri Nets. Proc. of *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2003)*, H. Garavel and J. Hatcliff (Eds.). Springer-Verlag, Lecture Notes in Computer Science 2619 (2003) 458–472.

[27] V. Khomenko, M. Koutny, and V. Vogler: Canonical Prefixes of Petri Net Unfoldings. Proc. of *International Conference on Computer Aided Verification (CAV'2002)*, E. Brinksma and K. G. Larsen (Eds.). Springer-Verlag, Lecture Notes in Computer Science 2404 (2002).

[28] V. Khomenko, M. Koutny, and A. Yakovlev: Detecting State Coding Conflicts in STGs Using Integer Programming. Proc. of *International Conference on Design, Automation and Test in Europe (DATE'2002)*, C. D. Kloos and J. Franca (Eds.). IEEE Computer Society Press (2002) 338–345.

[29] V. Khomenko and A. Mokhov: An Algorithm for Direct Construction of Complete Merged Processes. Proc. of *Application and Theory of Petri Nets (ATPN'2011)*, L. M. Kristensen and L. Petrucci (Eds.). Springer-Verlag, Lecture Notes in Computer Science 6709 (2011) 89–108.

[30] A. Kondratyev, J. Cortadella, M. Kishinevsky, E. Pastor, O. Roig, and A. Yakovlev: Checking Signal Transition Graph Implementability by Symbolic BDD Traversal. Proc. of *International Conference on Design, Automation and Test in Europe (DATE'1995)*, IEEE Computer Society Press (1995) 325–332.

[31] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, A. Taubin, and A. Yakovlev: Identifying State Coding Conflicts in Asynchronous System Specifications Using Petri Net Unfoldings. Proc. of *International Conference on Application of Concurrency to Sysytem Design (ICACSD'98)*, IEEE Computer Society Press (1998) 152–163.

[32] K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. of *International Conference on Computer Aided Verification (CAV'1992)*, G. von Bochmann and D. K. Probst (Eds.). Springer-Verlag, Lecture Notes in Computer Science 663 (1992) 164–174.

[33] K. L. McMillan: *Symbolic Model Checking: an approach to the state explosion problem*. PhD thesis, CMU-CS-92-131 (1992).

[34] S. Melzer: *Verifikation Verteilter Systeme mit Linearer — und Constraint-Programmierung*. PhD Thesis. Technische Universität München, Utz Verlag (1998).

[35] S. Melzer and S. Römer: Deadlock Checking Using Net Unfoldings. Proc. of *International Conference on Computer Aided Verification (CAV'1997)*, O. Grumberg (Ed.). Springer-Verlag, Lecture Notes in Computer Science 1254 (1997) 352–363.

[36] S. Römer: *Entwicklung und Implementierung von Verifikationstechniken auf der Basis von Netzentfaltungen*. PhD thesis, Technische Universitat Munchen (2000).

[37] A. Semenov: *Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfolding*. PhD Thesis, University of Newcastle upon Tyne (1997).

[38] A. Semenov, A. Yakovlev, E. Pastor, M. Peña, J. Cortadella, and L. Lavagno: Partial Order Approach to Synthesis of Speed-Independent Circuits. Proc. of *3rd IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'1997)*, IEEE Computer Society Press (1997) 254–265.

[39] N. Starodoubtsev, S. Bystrov, M. Goncharov, I. Klotchkov, and A. Smirnov: Towards Synthesis of Monotonic Asynchronous Circuits from Signal Transition Graphs. Proc. of *International Conference on Application of Concurrency to Sysytem Design (ICACSD'2001)*, IEEE Computer Society Press (2001) 179–188.

[40] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli: A Unified Signal Transition Graph Model for Asynchronous Control Circuit Synthesis. *Formal Methods in System Design* 9(3) (1996) 139–188.