PUNF Documentation and User Guide
Version 6.03

Victor Khomenko

University of Newcastle upon Tyne

July 2003

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, i.e.,it takes a Petri net and produces a finite complete prefix of its unfolding. Such a prefix can be used for efficient model checking (see, e.g., [14–16, 18, 22–26, 33, 34, 36–43]). 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, 27–29, 31, 32].

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 unsafety, 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, 31, 32, 41]).*

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, 33–35, 42, 43, 45]). 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 distinction between STG unfolding prefixes and the usual ones is that the signal codes are taken into account as a part of final state when checking the cut-off condition (see Figure 1).

3.3 High-level nets

As an experimental feature, PUNF can unfold high-level nets, viz. M-nets (see [1, 2, 12, 13, 22, 29]). 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,

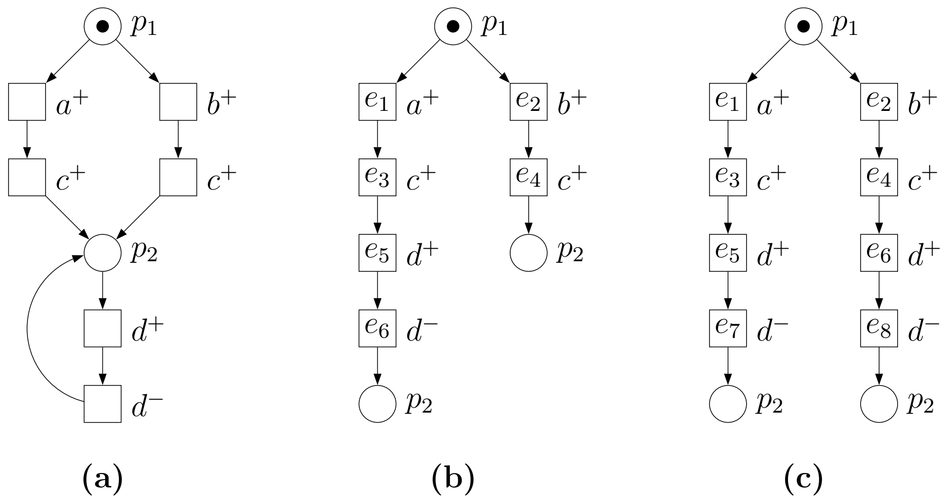

Figure 1: A consistent STG **(a)**, the 'usual' finite and complete prefix of its underlying net **(b)**, and the 'proper' STG unfolding prefix **(c)**. Note that in the 'usual' prefix e_4 is a cut-off event and a reachable state with the code $\{a \mapsto 0, b \mapsto 1, c \mapsto 1, d \mapsto 1\}$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

and thus solves this problem. Moreover, some extensions (e.g., infinite types) can now be supported.

Currently PUNF supports the following token types:

- the black token • (`dot`);
- booleans `ff` and `tt`;
- unsigned integers;
- tuples (including nested tuples) constructed from the types above.

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:

```
Guard -> Clause & Guard | Clause
Clause -> Predicate | Assignment
Predicate -> Expression
Assignment -> var=Expression | Expression=var
```

Each `Expression` is composed of constants and variables with the help of parentheses, the n-tuple constructors `(·,...,·)`, the unary logical negation `!`, the ternary conditional operator `if(·,·,·)`, and the following binary operators (in the order of increasing the precedence):

`|`	logical disjunction;
`&`	logical conjunction;
`=, #, <, <=, >, >=`	equality, non-equality, and ordering relations;
`+, -`	addition and substraction;
`*, /, %`	multiplication, division, and residue of division.

Note that the conditional operator `if(·,·,·)` is an extension to the syntax provided by the PEP tool. Another extension is that it is allowed to label

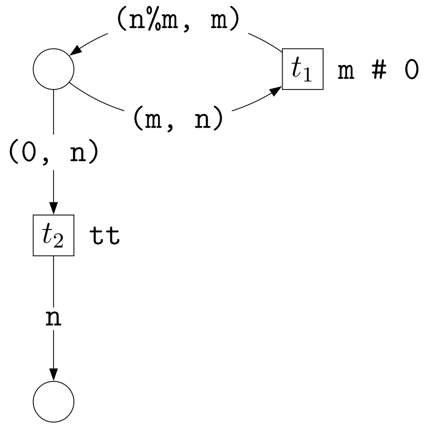

Figure 2: An M-net system modelling the Euclid's algorithm for computing the greatest common divisor of two non-negative integers.

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`).

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=2 & 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 transiton'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

Usage:

```
punf [-h]
```

or

```
punf [options] [-f=]filename
```

Options (may be given in any order):

-h Help.

-@[=n] Using ordering n (1-3). Specifies the ordering on configurations of the unfolding. If n is not specified or -@ is not used then the default value $n = 3$ is used.
If $n = 1$ then the McMillan's order is used.
If $n = 2$ then the size-lexicographical order is used.
If $n = 3$ then the Esparza/Römer/Vogler ordering is used.

-s Print statistics.

-t Print the running time.

-q Print running statistics (may be helpful to estimate the progress).

-p Print running statistics as multiple lines.

-w Wait for user input before exiting (prevents the window from closing).

-c Do not store cut-off events.

-! Do not optimize the preset trees. Cheap optimization, viz. gluing vertexes labelled by the same set, is performed anyway. This option is for maintenance purposes only.

-N=n Specifies the number of working threads (by default, it is equal to the number of processors).[1]

-n=n Abandon the concurrency relation and switch to a slower algorithm after the specified number of non-cut-off events is generated. The default value is 20000. If your computer has a lot of memory, increase it to improve the performance. If the value is 0, the concurrency relation is not generated.

-T=*filename* Save the preset trees in file *filename*. This option is for maintenance purposes only.

-m=*filename* Save the resulting unfolding in the .mci format in file *filename*. If this option is not specified then the name for the result file is obtained by removing from the name of input file the suffix '.ll_net', '.stg', '.g', or '.hl_net' if it is present there, and appending the suffix '.mci'.

[**-f**=]*filename* Specifies the input file.

[1]This option is not available in the sequential version of the unfolder.

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 and M. Koutny: Deadlock Checking Using Liner Programming and Partial Order Dependencies. Technical Report CS-TR-695, Department of Computing Science, University of Newcastle (2000).

[24] V. Khomenko and M. Koutny: Verification of Bounded Petri Nets Using Integer Programming. Technical Report CS-TR-711, Department of Computing Science, University of Newcastle (2000).

[25] V. Khomenko and M. Koutny: LP Deadlock Checking Using Partial Order Dependencies. Proc. of *International Conference on Concurrency Theory (CONCUR'2000)*, C. Palamidessi (Ed.). Springer-Verlag, Lecture Notes in Computer Science 1877 (2000) 410–425.

[26] V. Khomenko and M. Koutny: Verification of Bounded Petri Nets Using Integer Programming. *Formal Methods in System Design* (2002) submitted paper.

[27] V. Khomenko and M. Koutny: An Efficient Algorithm for Unfolding Petri Nets. Technical Report CS-TR-726, Department of Computing Science, University of Newcastle (2001).

[28] 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.

[29] V. Khomenko and M. Koutny: Branching Processes of High-Level Petri Nets. Technical Report CS-TR-763, Department of Computing Science, University of Newcastle (2002).

[30] 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.

[31] V. Khomenko, M. Koutny, and V. Vogler: Canonical Prefixes of Petri Net Unfoldings. Technical Report CS-TR-741, Department of Computing Science, University of Newcastle (2001).

[32] 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).

[33] V. Khomenko, M. Koutny, and A. Yakovlev: Detecting State Coding Conflicts in STGs Using Integer Programming. Technical Report CS-TR-736, Department of Computing Science, University of Newcastle (2001).

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

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

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

[43] 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.

[44] 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.

[45] 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.