The Petri Net Kernel Team

The Petri Net Kernel

File Format Documentation

Humboldt-Universität zu Berlin
Computer Science Department
Unter den Linden 6
10099 Berlin
Germany

March 29, 1999

Contents

1 Intention

This documents contains the documentation of the file format as used by the Petri Net Kernel. It is intented to deliver information needed to support the file format in applications *not* based upon the Petri Net Kernel.

First a few commented example files serve as an informal illustration of the file format. They are followed by two contextfree grammars.

An application may need to interpret files created by the Petri Net Kernel or store its data in files complying with the Petri Net Kernel file format. This document therefore includes two slightly different contextfree grammars, a so called analytical and a constructive grammar. The first corresponds with the format of files written by the Petri Net Kernel, the latter with the format the Petri Net Kernel is able to interpret properly. Naturally, the analytical grammar produces a language which is a true subset of the language corresponding to the constructive grammar. Consequently one can rely on the stricter analytical grammar, when working on the task of interpreting a Petri Net Kernel file. The constructive grammar offers a greater degree of freedom, which facilitates producing Petri Net Kernel compatible files.

2 Notation

The EBNF notation used in this documentation is oriented at the ISO/IEC EBNF standard[1]. The productions are divided into groups of one or more rules. Each group contains rules closely related to one another. Any comments are presented as an indented list beneath the rule definition(s).

3 Example Nets

The three example files presented within the next two sections serve as an informal illustration of the Petri Net Kernel file format. The third example is commented in detail and may in some cases be sufficient to implement support of the file format.

3.1 Empty Nets

The following two examples are files containing an empty P/T–Net and an empty High–Level–Net respectively. They show the minimal number of terminals and their order for either net specification.

[1]ISO/IOC 14977:1996(E)

```
|NET
|no_name
|SPECIFICATION ST_Specification
|PLACES
;
|TRANSITIONS
;
|ARCS
;
|MARKING
;
|NET_END
EDITOR_INFOS
PAGES
;
PLACE
;
TRANSITION
;
ARC
;
END_EDITOR
```

```
|NET
|no_name
|SPECIFICATION HL_Specification
|DECLARATION
|
;
|VARIABLES
|
;
|PLACES
;
|TRANSITIONS
;
|ARCS
;
|MARKING
;
|NET_END
EDITOR_INFOS
PAGES
;
PLACE
;
TRANSITION
;
ARC
;
END_EDITOR
```

3.2 A more complex High–Level–Net

This example file is split up into consecutive parts. Each part is explained individually. For further details please consult the analytical grammar.

```
|NET
|no_name
|SPECIFICATION HL_Specification
```

The header of the net information part of the file. The keyword `NET` is followed by the name of the net (in this case the default value `no_name`), the `SPECIFICATION` keyword and the actual name of the net specification (in this case `HL_Specification` for a High–Level–Net).

```
|DECLARATION
|f(x)
|g(y)
;
|VARIABLES
|x
|y
;
```

The High–Level–Net specification defines two global extensions called `DECLARATION` and `VARIABLES`. They and their values immediately follow the specification's name.

```
|PLACES
|p1 1
;
|p4 4
;
|p5 5
;
|p6 6
;
;
```

The keyword `PLACES` introduces a list terminated by a semicolon. Each entry, itself terminated by a semicolon, represents one place of the net. Its name is followed by a unique identification number. A net specification might define extensions local to a place. Then each extension's name and its values are stored within the place entry as well.

```
|TRANSITIONS
|t2 2
;
|t3 3
;
;
```

The `TRANSITIONS` section is similar to the `PLACES` section.

```
|ARCS
|4 --> 3 7
|[0]
;
;
|3 --> 5 8
|[0]
;
;
|1 --> 3 9
|x
;
;
|1 --> 2 10
|f(x)
;
;
|2 --> 6 11
|[9]
;
;
;
```

ARCS starts the list containing an entry for each arc of the net. The unique identification number of the source node is followed by the --> token, the target node's number and the arc's number. The following line equals the arc inscription.

```
|MARKING
|1
|[7, 8]
;
|6
|[11, 9]
;
;
|NET_END
```

The last section of the net information part consists of the initial marking of the net. For each place carrying an initial marking different from the empty marking specified by the net specification, an entry is present in the list. The place is identified by its number. Its marking is given on the following line. The net information part is terminated by NET_END.

```
EDITOR_INFOS
PAGES
   1 516x516+91+199 1
   2 516x516+669+217 1
;
```

The editor information part is introduced by EDITOR_INFOS. The first items to be defined are the pages administered by the editor. A net can be distributed on several pages. Each page has a unique identification number and a certain geometry: width, height, coordinates (x, y) of the left-top window corner and a visibility flag (1 = visible). The PAGES section as well as any of the following sections is terminated by a semicolon.

```
PLACE
   6 :
   2 346.0 210.0
   {'Initial Marking': (0, -20, 1)}
   .
   5 :
   1 128.0 141.0
   {'Initial Marking': (0, -20, 0)}
   .
   4 :
   1 180.0 300.0
   {'Initial Marking': (0, -20, 0)}
   .
   1 :
   1 356.0 216.0
   {'Initial Marking': (0, -20, 1)}
   2 78.0 206.0
   {'Initial Marking': (0, -20, 1)}
   .
;
```

Each entry (terminated by a dot) in the `PLACE` section represents a single place of the net and its graphical representatives. The place is identified by its number. For each representative the page number, its absolute coordinates within the page and the relative coordinates and visibility of any extensions, identified by their names, are given.

```
TRANSITION
   3 :
   1 231.0 214.0
   .
   2 :
   2 190.0 214.0
   .
;
```

The same is valid for the `TRANSITION` section.

```
ARC
   11 :
   2 0 0
   {}
   .
   10 :
   2 1 0
   {}
   .
   9 :
   1 0 0 SMOOTH POINT 256.5 116.0
   {}
   .
   8 :
   1 0 0
   {}
   .
   7 :
   1 0 0 POINT 375.5 323.5
   {}
   .
;
END_EDITOR
```

The `ARC` section differs slightly. Naturally there is always only one graphical representation for an arc. The arc is identified by its number. It follows the page number. The following two numbers identify the n$^{\text{th}}$ graphical representative of source and target node respectively. The keyword `SMOOTH` means, that a smooth line has to be drawn from source to target controled by the dragpoints, whose coordinates are delivered by the list following the `POINT` keyword. If `SMOOTH` is omitted, the arc is drawn as a set of straight lines along the dragpoints. The editor information part and the file as a whole is terminated by `END_EDITOR`.

4 Analytical Grammar

As mentioned above, this section deals with the grammar corresponding to the format of files written by the Petri Net Kernel. It is stricter than the following constructive grammar in the sense that its language is a true subset of the language produced by the constructive grammar.

- LF = '`\n`';
- SPC = ' ';
- wSPC = SPC | TAB;
- wSPCSeq = wSPC | wSPC, wSPCSeq;
- string = {char - LF};
- nonEmptyString = char - LF, string;
- word = char - (wSPC | LF), {char - (wSPC | LF)};
  - wSPC: whitespace characters; at least(!) space and tabulator

  - wSPCSeq: a non-empty sequence of whitespace characters
  - char: any ASCII character
  - string: sequence (possibly empty) of any characters, but linefeed
  - nonEmptyString: non–empty string
  - word: non-empty sequence of any characters, but linefeed or white space characters

---

- stringCollection = startOfLine, string, LF |
startOfLine, string, LF,
stringCollection;
  - one or more lines containing an arbitrary string

- endOfBlock = '`;`';
  - common lexical tokens appearing in both the net and editor data part

---

- startOfLine = '`|`';
- startOfNet = '`NET`';
- endOfNet = '`NET_END`';
- startSpecification = '`SPECIFICATION`';
- startOfPlaces = '`PLACES`';
- startOfTransitions = '`TRANSITIONS`';
- startOfArcs = '`ARCS`';
- arcArrow = '`-->`';
- startOfMarking = '`MARKING`';
  - lexical tokens appearing in the net data part only

- startOfEditor = '`EDITOR_INFOS`';
- endOfEditor = '`END_EDITOR`';
- startOfPages = '`PAGES`';

- edStartOfPlaces = '`PLACE`';
- edStartOfTransitions = '`TRANSITION`';
- startOfEdges = '`ARC`';
- endOfObject = '`.`';
- toggleSmooth = '`SMOOTH`';
- startOfDragPoints = '`POINT`';
  - lexical tokens appearing in the editor data part only

- net = startOfLine, startOfNet, LF,
  startOfLine, nameOfNet, LF,
  netSpecification,
  startOfLine, startOfPlaces, LF,
  {place, {extension}, endOfBlock, LF},
  endOfBlock, LF,
  startOfLine, startOfTransitions, LF,
  {transition, {extension}, endOfBlock, LF},
  endOfBlock, LF,
  startOfLine, startOfArcs, LF,
  {arc, {extension}, endOfBlock, LF},
  endOfBlock, LF,
  startOfLine, startOfMarking, LF,
  {initialMarking},
  endOfBlock, LF,
  startOfLine, endOfNet, LF, [editorNet];
- `nameOfNet` = string;

- netSpecification = startOfLine, startSpecification, SPC, nameOfSpecification, LF,
  {startOfLine, nameOfSpecificationExtension, LF,
  valueOfSpecificationExtension, endOfBlock, LF};
- nameOfSpecification = word;
- nameOfSpecificationExtension = nonEmptyString;
- valueOfSpecificationExtension = stringCollection;

- extension = startOfLine, nameOfExtension, LF,
  valueOfExtension, endOfBlock, LF;

- nameOfExtension = nonEmptyString;
- valueOfExtension = stringCollection;

- place = node;
- transition = node;

- node = startOfLine, nameOfNode, SPC, idOfNode, LF;
- idOfNode = integer;

- arc = startOfLine, idOfArcSource, SPC, arcArrow, SPC, idOfArcTarget, SPC,
  idOfArc, LF,
  arcInscription, endOfBlock, LF;

- idOfArcSource = idOfNode;
- idOfArcTarget = idOfNode;
- idOfArc = integer;
- arcInscription = stringCollection;

- initialMarking = startOfLine, idOfPlace, LF,
  valueOfInitialMarking;
- idOfPlace = idOfNode;
- valueOfInitialMarking = stringCollection;

- editorNet = startOfEditor, LF,
  {editorObjects}, endOfEditor, LF;
- editorObjects = startOfPages, LF,
  {page}, endOfBlock, LF,
  edStartOfPlaces, LF,
  {SPC, SPC, SPC, idOfPlace, SPC, ':', LF,
  editorNode, SPC, SPC, SPC, endOfObject, LF},
  endOfBlock, LF,

edStartOfTransitions, LF,
{SPC, SPC, SPC, idOfTransition, SPC, ':', LF,
editorNode, SPC, SPC, SPC, endOfObject, LF},
endOfBlock, LF,
startOfEdges, LF,
{SPC, SPC, SPC, idOfEdge, SPC, ':', LF,
editorEdge, SPC, SPC, SPC, endOfObject, LF},
endOfBlock, LF;

- page = SPC, SPC, SPC, idOfPage, SPC, pageGeometry, SPC, viewstate, LF;
- idOfPage = integer;
- pageGeometry = width, 'x', height, '+', aboveLeftX, '+', aboveLeftY;
- width = integer;
- height = integer;
- aboveLeftX = integer;
- aboveLeftY = integer;
- viewstate = '0' | '1';

- objectExtension = '\{}',[extensionEntries],'\url{\}';
- extensionEntries = extension | extension, ',', SPC, extensionEntries;
- extension = '’', nameOfExtension, '’', ':', SPC, '(', relativX, ',', SPC, relativY, ',', SPC, viewstate, ')';
- nameOfExtension = nonEmptyString;
- relativX = integer;
- relativY = integer;

- editorNode = SPC, SPC, SPC, idOfPage, SPC, absoluteX, SPC, absoluteY, LF, [objectExtension], LF;
- editorEdge = SPC, SPC, SPC, idOfPage, SPC, numberOfSourceReference, SPC, numberOfTargetReference, [dragpointList], LF,
objectExtension, LF;
- numberOfSourceReference = integer;

- numberOfTargetReference = integer;
- dragpointList = [SPC, toggleSmooth], SPC, startOfDragPoints, points;
- points = SPC, absoluteX, SPC, absoluteY | SPC, absoluteX, SPC, absoluteY, points;

5 Constructive Grammar

This grammar corresponds with the files the Petri Net Kernel is able to interpret correctly. In the case of creating a Petri Net Kernel compatible file, it might be helpful that the Petri Net Kernel is somewhat tolerant when loading a file.

Any non-terminal not completely defined by the following productions can be considered to be defined as in the previous Analytical Grammar section.

- tolerance = {endOfBlock, string, LF
  | startOfLine, {wSPC}, {char} - (startSpecification | startOfPlaces | startOfTransitions | startOfArcs | startOfMarking | wSPC | LF), [wSPC, nonEmptyString], LF};

- `editorNet` = [{wSPC}, {char} - (startOfEditor | wSPC | LF),
  [wSPC, nonEmptyString], LF],
  {wSPC}, startOfEditor, [wSPC, nonEmptyString], LF,
  {string, LF},
  startOfPages, [wSPC, nonEmptyString], LF, pages,
  {string, LF},
  edStartOfPlaces, [wSPC, nonEmptyString], LF, nodes,
  {string, LF},
  edStartOfTransitions, [wSPC, nonEmptyString], LF, nodes,
  {string, LF},
  startOfEdges, [wSPC, nonEmptyString], LF, edges,
  {string, LF},
  {wSPC}, endOfEditor, [wSPC, {char}];

- pages = {page}, endOfBlock, [wSPC, nonEmptyString], LF;
- page = {wSPC}, idOfPage, [wSPCSeq, pageGeometry, [wSPCSeq,
  viewstateOfPage, [wSPC, nonEmptyString]]], LF;

- nodes = {{wSPC}, idOfNode, [wSPC, nonEmptyString], LF,
  {{wSPC}, idOfPage, wSPCSeq, absoluteX, wSPCSeq, absoluteY, [wSPC,
  nonEmptyString], LF,

[nodeExtension, LF]},
endOfObject, [wSPC, nonEmptyString], LF},
endOfBlock, [wSPC, nonEmptyString], LF;

- edges = {{wSPC}, idOfEdge, [wSPC, nonEmptyString], LF,
{{wSPC}, idOfPage, wSPCSeq, numberOfSourceReference, wSPCSeq,
numberOfTargetReference, [dragpointList], LF,
[edgeExtension, LF]}, endOfObject, [wSPC, nonEmptyString], LF},
endOfBlock, [wSPC, nonEmptyString], LF;

- net = startOfLine, {wSPC}, startOfNet, [wSPC, nonEmptyString], LF,
startOfLine | endOfBlock, nameOfNet, LF, tolerance,
startOfLine, {wSPC}, startSpecification,
[wSPCSeq, nameOfSpecification], [wSPC, nonEmptyString], LF,
specificationExtension, tolerance,
{startOfLine, {wSPC}, startOfPlaces, [wSPC, nonEmptyString], LF,
places}, tolerance,
{startOfLine, {wSPC}, startOfTransitions, [wSPC, nonEmptyString], LF,
transitions}, tolerance,
{startOfLine, {wSPC}, startOfArcs, [wSPC, nonEmptyString], LF,
arcs}, tolerance,
{startOfLine, {wSPC}, startOfMarking, [wSPC, nonEmptyString], LF,
initialMarking}, tolerance,
endOfNet, (wSPC, {char} | [wSPC, nonEmptyString], LF, editorNet);

- places = nodes;

- transitions = nodes;

  - the order of loading of places and transitions is arbitrary; place and transition sections can be mixed without any restrictions
  - correct loading of an arc requires the corresponding place and transition to be already loaded
  - correct loading of an initial mark requires the corresponding place to be already loaded
  - the grammar is more restrictive in the sense that an order is introduced; this order need not to be followed, if the constraints above are met

---

- specificationExtension = {startOfLine, {wSPC}, nameOfSpecificationExtension,
[wSPC, nonEmptyString], LF,

{startOfLine, string, LF},
endOfBlock, string, LF;

- nodes = {char - (LF | endOfBlock), node,
  {char - (LF | endOfBlock), extension},
  endOfBlock, string, LF},
  endOfBlock, string, LF;

- node = nameOfNode, wSPC, idOfNode, LF;

- nameOfNode = word;

- extension = nameOfExtension, LF,
  valueOfExtension, endOfBlock, string, LF;

- nameOfExtension = nonEmptyString;

- valueOfExtension = {char - (LF | endOfBlock), string, LF};

- arcs = {startOfLine, {wSPC}, numberOfSourceReference,
  wSPCSeq, word, wSPCSeq,
  numberOfTargetReference, wSPCSeq, idOfArc, [wSPC, nonEmptyString], LF,
  arcInscription,
  {char - (LF | endOfBlock), extension},
  endOfBlock, string, LF},
  endOfBlock, string, LF;

- arcInscription = nonEmptyString, LF
  | {nonEmptyString, LF},
  endOfBlock, string, LF;

- initialMarking = {startOfLine, {wSPC}, idOfPlace, [wSPC, nonEmptyString], LF,
  {char - (LF | endOfBlock), string, LF}, endOfBlock, [wSPC, nonEmptyString],
  LF},
  endOfBlock, [wSPC, nonEmptyString], LF;