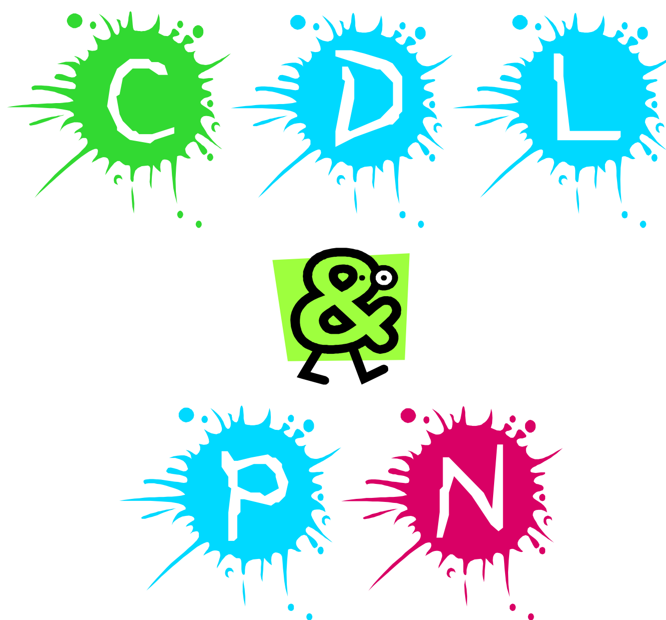
Project Documentation



CDL ↔ PN

A translator from Petri-Nets to the VeriTech CDL language and back

Submitted by:

Anastasia Braginsky

July 2004

1. Table of Contents

1. Table of Contents **2**

2. Abstract **3**

3. Who and What **4**

4. Programmer Guide **5**

5. User Guide **7**

5.1 Translating from .cdl to .net 7
5.2 Translating from .net to CDL 7
5.2.1 Manual usage 7
5.2.2 Automatic usage 8

6. Code Documentation **10**

6.1 CDL2PN 10
6.2 PN2CDL 10
6.2.1 Parsing part overview 11
6.2.2 Translating part overview 14
6.2.3 Creating a .xml part overview 14
6.2.4 Presenting .xml file as .html 15

7. Examples **16**

7.1 Examples with Booleans 17
7.1.1 Single variable and transition example 17
7.1.2 Real world example: consumer-producer 20
7.1.3 Nondeterministic initialization of Boolean example 26
7.2 Example with abstraction of integers 29
7.3 Example with program counters 34

8. TODO list **36**

2. Abstract

The CDL<->PN (Petri-Nets to Core Definition Language and back) system is part of the Veritech Project.

This document is documentation of CDL<->PN project, here you can find exactly explanations about PN2CDL parsing part and general information about both sides translation. For missed explanations please look in to the CDL2PN and PN2CDL documentations. Files cdl2pn.doc and pn2cdl.doc you can find in current directory.

The current version includes translating common Petri net files (PNK format) into the files representing a core program (HTML view). The translation is visa versa, so we could also translate CDL (.cdl files) into Petri net files (PNK format).

3. <u>Who & What</u>

The Veritech project is headed by Prof. Orna Grumberg (orna@cs.technion.ac.il) and Prof. Shmuel Katz (katz@cs.technion.ac.il) at the Technion.

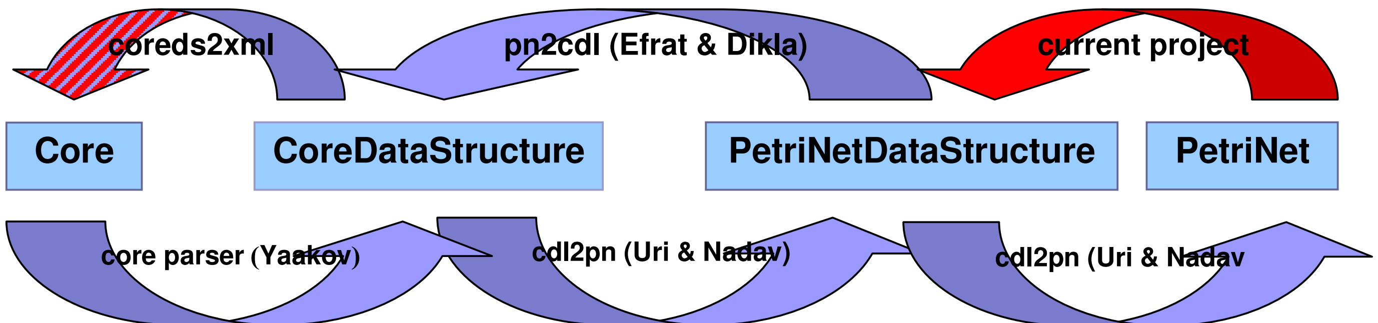
The last part of the project was written by Anastasia Braginsky (sanastas@techst02.technion.ac.il). (parsing .net file and creation of a Petri net data structure, translation of CDL data structure to the readable file, and integration of all written parts)

Other parts of the project were written by:

Uri Dekel & Nadav Golbandi - translation of .cdl files to .net files

Efrat Dayagi & Dikla Yanay - translation of Petri net data structure to CDL data structure

The Veritech project is hosted at the SSDL lab, headed by Mr. Shahar Dag (shahar_d@cs.technion.ac.il).



4. Programmer Guide

The project files are located in the **/home/veritech/VeriTech_project/CDL<->PN** directory on the CS file system in the Technion.

The CDL<->PN directory contains those subdirectories:

1. **hdr** – This is the directory containing the header files.
2. **src** – This is the directory containing the source code.

   ptc.c – is the main file for pn2cdl part, includes the Petri net files parsing entity

   wrapper.c – is the main file for cdl2pn part

   Information about other files could be found in "File listing" part in pn2cdl.doc document.

3. **obj** – Contains the object files that are created.
4. **bin** – Contains the exe files and libraries.

   cdl2pn – executable that translate .cdl file into .net file

   pn2cdl – executable that translates .net files into .xml files

   CDL2PN_LIB, PN2CDL_LIB - libraries

5. **exampl** – Contains various examples.
   a. *pn2cdl2pn* – contains a directory for each example (those examples was used by Uri, Nadav, Dikla and Efrat to check their programs)
   b. *Bad_core_programs* – contains .cdl files (given by Shahar) on which pn2cdl failed to run due to problems explained in example_results.txt

c. *real_pn2core* – contains .net files taken from Linux

6. **doc** – Contains a version of this document,

   cdl2pn.doc – documentation of cdl2pn part of this project.

   pn2cdl.doc – documentation of translation Petri net data structure to the CDL data structure, that is also part of this project

   PetriNet2CoreParsing.ppt – presentation that explains parsing of Petri net files.

7. **xml2cdl** – contains all the files that help to translate .xml files to the .html files that have CDL syntax.

5. User Guide

5.1 Translating from .cdl to .net

In directory bin/ you can find cdl2pn executable, invoke this executable in such a way:

cdl2pn <file_name>.cdl <file_name> PNK

After this you will find files <file_name>.net and <file_name>.log in your work directory. Use <file_name>.net to translate it back to CDL format (if visa versa usage is what you need). For more information you can use cdl2pn.doc documentation (Running in the Project Directory part)

5.2 Translating from .net to CDL

There are two options to use pn2cdl command.

1. to use it manually
2. to use pn2cdl_all automatic script

5.2.1 Manual usage:

In directory bin/ you can find pn2cdl executable, invoke this executable in such a way:

pn2cdl <file_name>.net > > <file_name>.logfile

If you will not use redirection to the logfile all the output will be printed out on a screen. It will be not easy to use, so better use redirection.
If program is used in other way, the program will be terminated with respective printout. After redirection in <file_name>.logfile one can find a report how parsing is done and if there were some errors while parsing. After all you will find file <file_name>_cdl.xml in your work directory.

To see <file_name>_cdl.xml file in the easy way you can use the following:

Copy <file_name>_cdl.xml file to the xml2cdl directory and invoke the following command inside xml2cdl directory

java -classpath jdom.jar:. XML2HTML <file_name>_cdl.xml xml2cdl.xsl <file_name>.html

the command is also written in **run** file in xml2cdl directory.

After invoking this command .html file is created and it is easy to open the file using Internet Explorer in Windows OS.

5.2.2 Automatic usage:

Invoke pn2cdl_all script in such a way:

pn2cdl_all <file_name>.net <file_name>

For example:

pn2cdl_all example/net_examples/cycle.net cycle

pn2cdl_all is location depended script (it assumes current directory structure) so if you want to invoke it anywhere else you need recursively copy all the directory there.

Results: If you run pn2cdl_all on file with name xxx after completion you will find files: xxx.log xxx.html and xxx_cdl.xml in current directory. Results will be created only if the run was successful, if there are not all files look in to .log file, if there is not also .log file then you invoked the script not correctly so check yourself.

xxx.log - log file includes run information and error messages if the run wasn't successful .

xxx_cdl.xml - xml file, the xml view on created CDL file, not so easy to understand it.

xxx.html - html file, the html view on created CDL file, it is easy to understand it. One could open it from WINDOWS.

6. Code Documentation

6.1 CDL2PN

Full CDL2PN code documentation could be found in cdl2pn.doc

6.2 PN2CDL

The translation works for flat Petri nets only, and not for object oriented Petri nets. That means that we support a system of only one Petri net class. The translation also does not support colored Petri-net. PN2CDL part consists of four subparts:

- Parsing of .net file and creating Petri net data structure
- Translating PN data structure to CDL data structure
- Creating a .xml file from CDL data structure
- Presenting .xml file as .html in order to make it easy to use.

6.2.1 Parsing part overview

Used data structures:

Three dictionary data structures are used in the parser:

- The main one that will hold the PN data structure, and will be passed to the pn2cdl. This is the Petri Net data structure.
- Places dictionary will hold places which were found in places part.
- Transitions dictionary will hold transitions which were found in transitions part.

Places and Transitions dictionaries are only for inner use and will be destroyed after using.

We could not use the main DS in this purpose since in the PNDS the keys to the elements are their names, but we want to find them by the identical number.

The Petri nets data structures, which are created, are based on those used for the PN2CDL translation.

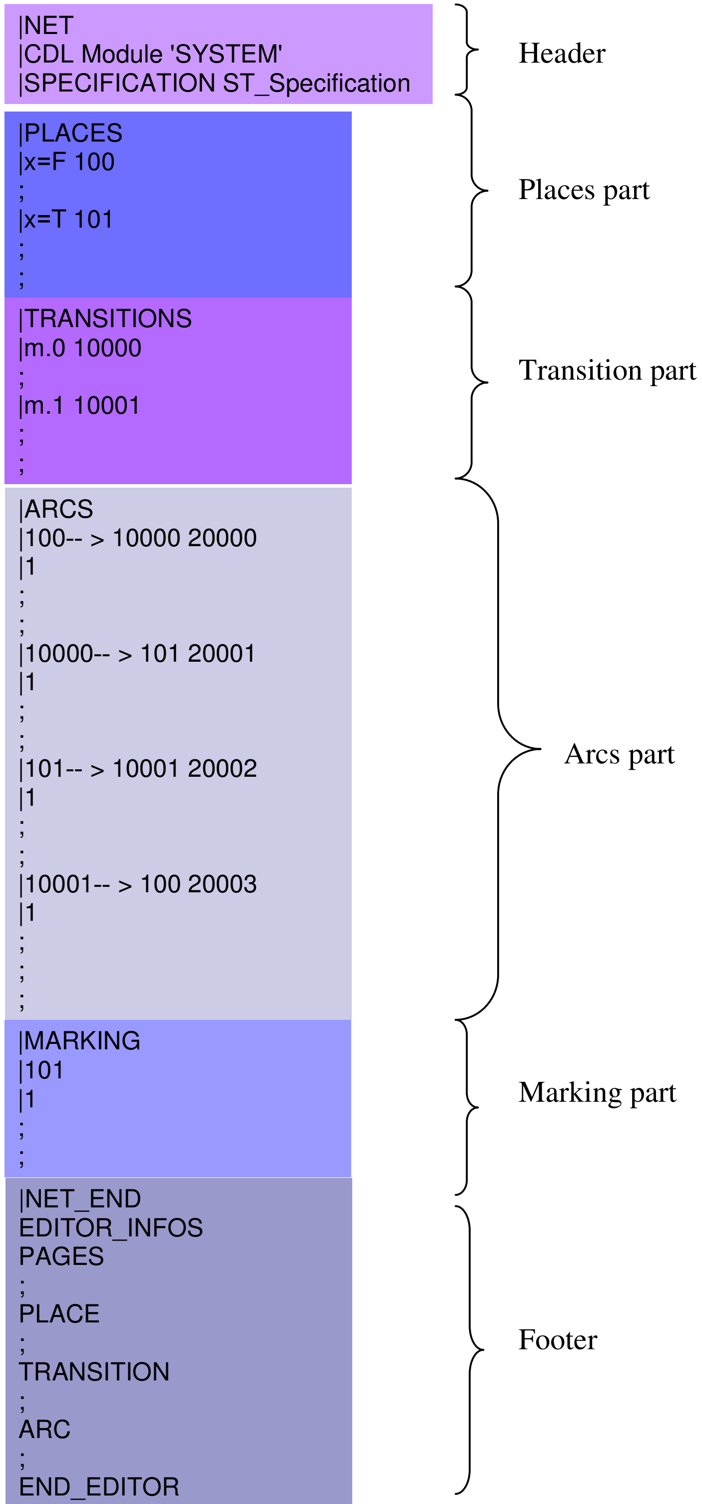
The three major data structures are **MPplace**, **MPtransition** and **MPconnector**, which correspond to Petri-net places, transitions and connectors.

All these objects for the net corresponding to a CDL module are located in an **MPclass** object.

More information about Petri net data structures could be found in pn2cdl.doc document (4.3 Data Structures).

Parsing:

Parsing works according to the .net file syntax and takes only one pass. .net file has the following syntax:


|NET
|CDL Module 'SYSTEM'
|SPECIFICATION ST_Specification
|PLACES
|x=F 100
;
|x=T 101
;
;
|TRANSITIONS
|m.0 10000
;
|m.1 10001
;
;
|ARCS
|100-- > 10000 20000
|1
;
;
|10000-- > 101 20001
|1
;
;
|101-- > 10001 20002
|1
;
;
|10001-- > 100 20003
|1
;
;
;
|MARKING
|101
|1
;
;
|NET_END
EDITOR_INFOS
PAGES
;
PLACE
;
TRANSITION
;
ARC
;
END_EDITOR
Header
Places part
Transition part
Arcs part
Marking part
Footer

Header:

Header is used for finding Petri Net class name. Name is derived from previous CDL Module Name, if it is mentioned; else it is derived from Petri Net file name.

The additional info comes from transition and place names. So each place/transition name is translated as it is.

Place and transition part:

Place and transition parts are used to detect their names, and to insert places/transitions to the places/transitions dictionary. Element's identical number is used as insertion key.

Also places and transitions are inserted as MPplaces and MPtransitions to the main dictionary (Petri net data structure).

Arcs part:

Using origin and destination identical number we would find origin place/transition and destination place/transition, using places and transitions dictionaries.

If origin or destination is not found connector is not created. The program will run till the end of the parsing part (will not continue to the core data structure creation) and will terminate with the respective message in the log file. Else MPconnector is created and inserted to the main data structure.

Pay attention that there is no check if origin and destination elements are from different element type.

Marking part:

Marking part: Each place is found, using id number. Number of tokens, which follows the id number is added to MPplace as initial tokens.

6.2.2 Translating part overview

After parsing is done, the following command is invoked in order to translate pn data structure (modulesDictionary) to the CDL data structure (CoreDataStructure)

CoreDataStructure = TranslatePetriNetClasses(&modulesDictionary);

More about translation part you can read in pn2cdl.doc it is generally all this document talks about that.

6.2.3 Creating a .xml part overview

This information could be relevant to any person, who needs to transform CDL data structure into .xml file.
In order to create file.xml one should use:

void DS2XML(CoreProgram* c,FILE* out_file)

function.
Also this file should be included:

VeriTech_project/CoreParser/utils/CdlDs_2_XML/hdr/coreprogram_print.h

DS2XML is compiled separately and as executable is compiled with our executable in the last stage of compiling.
All above creates 0ut_file.xml file in work directory.

6.2.4 Presenting .xml file as .html

In order to see the results in easy way xml view could be translated to the html view and than easy viewed via Internet Explorer from Windows. In order to do this you need all the files which are located inside xml2cdl directory, so if you want to do it anywhere else, please copy the entire xml2cdl directory.

Copy file.xml file to the xml2cdl directory and invoke the following command inside xml2cdl directory

java -classpath jdom.jar: . XML2HTML file.xml xml2cdl.xsl file.html

The command is also written in **run** file in xml2cdl directory.

After invoking this command .html file is created and it is easy to open the file using Internet Explorer in Windows OS.

Pay your attention: xml file syntax can't contain any '&' since '&' is reserved character in xml. So if you want to use this utility and you know that your program include &PC variable, you should replace any instance of &PC by '&PC'.

7. Examples

Here will be presented only examples of pn2cdl part, for any other examples see other documents in current directory. All examples (and all example views) could be found in exampl/ directory of the project. Also more examples could be created using PNK. More about running PNK you can read in "Running PNK" paragraph of CDL2PN documentation (page 9). Warning: while testing or running PNK you could find some difficulties, if so please contact ssdl administrator.

7.1 Examples with Booleans

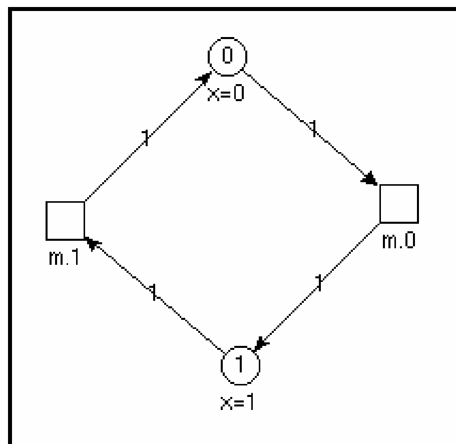
7.1.1 Single Variable and Transition Example

This is the simplest example. It contains single module, single variable and single transition. It's original CDL view can be found in directory **exampl/fig1_2** in the file **fig1_2.cdl**.

The source is:

```
HOLD_PREVIOUS
MODULE SYSTEM ()
{
    VAR
        x: boolean INITVAL true;
    TRANS m:
        enable: true;
        assign: x' := !x;
  }
```

The PN program we get in result has the form of a diamond:



The Petri net view can be found in directory **exampl/fige1_2** in the file **fige1_2.net.** It's looks so:

```
|NET
|CDL Module 'SYSTEM'
|SPECIFICATION ST_Specification
|PLACES
|x=F 100
;
|x=T 101
;
;
|TRANSITIONS
|m.0 10000
;
|m.1 10001
;
;
|ARCS
20000 10000 <-- 100|
1|
;
;
20001 101 <-- 10000|
1|
;
;
20002 10001 <-- 101|
1|
;
;
20003 100 <-- 10001|
1|
;
;
;
|MARKING
101|
1|
;
;
|NET_END
EDITOR_INFOS
PAGES
;
PLACE
;
TRANSITION
;
ARC
;
END_EDITOR
```

After translating it back to core using pn2cdl executable we get this target program. There are actually two results files: fig1_2.xml and fig1_2.html.

This file was "predicted" by pn2cdl example, note that the expressions for the enable and assign parts are equal to those in the source file:

```
HOLD_PREVIOUS
MODULE SYSTEM ()
{
VAR
     x: boolean INITVAL true;
     TRANS m:
     enable: x \/ !x ;
     assign: x':=(true \/ !!x) /\ (false \/ !x);
  }
```

And this is what we really get in html_examples directory:

MODULE SYSTEM() {

VAR

x: **boolean INITVAL true** ;

Trans m:

enable: x ∨ ! x ;

assign: x ' := **true** ∨ ! ! x ∧ **false** ∨ ! x ;

}

Results are same, but in the Boolean functions of enable and assign have no parenthesis.

7.1.2 Real World Example: Consumer-Producer

Follow is a real world example, of a consumer-producer system, listed in **cycle.cdl**.

It's original CDL view can be found in directory **exampl/cycle** in the file **cycle.cdl**.

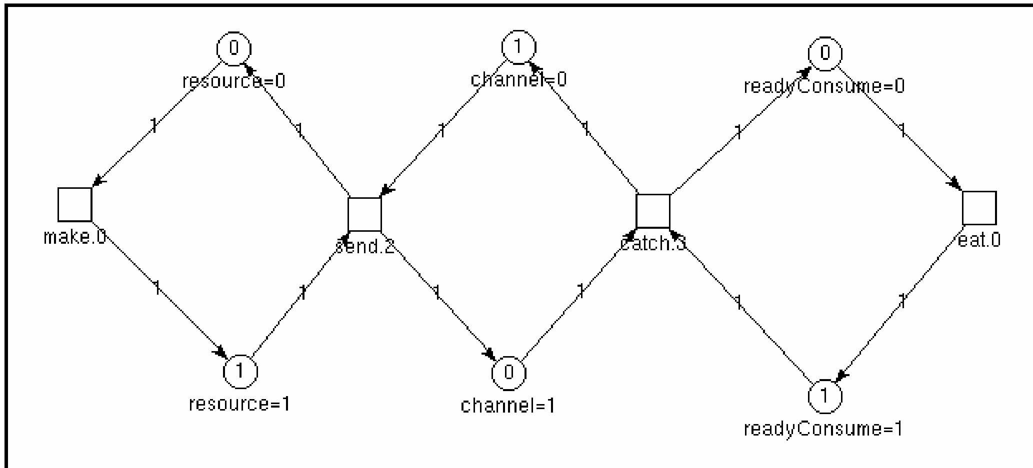
The source is:

```
HOLD_PREVIOUS
MODULE SYSTEM()
{
VAR
resource: boolean INITVAL true;
     channel: boolean INITVAL false;
     readyConsume: boolean INITVAL true;

TRANS send:
     enable: resource /\ !channel;
     assign: resource':=false;
             channel':=true;
TRANS make:
     enable: !resource;
     assign: resource':=true;

TRANS catch:
     enable: channel /\ readyConsume;
     assign: readyConsume':=false;
             channel':=false;
TRANS eat:
     enable: !readyConsume;
     assign: readyConsume':=true;
}
```

The resulting net is:



The Petri net view can be found in directory **exampl/cycle** in the file **cycle.net.** It's looks so:

```
|NET
|CDL Module 'SYSTEM'
|SPECIFICATION ST_Specification
|PLACES
|resource=F 100
;
|resource=T 101
;
|channel=F 102
;
|channel=T 103
;
|readyConsume=F 104
;
|readyConsume=T 105
;
;
|TRANSITIONS
|send.2 10000
;
|make.0 10001
;
|catch.3 10002
;
|eat.0 10003
;
;
|ARCS
20000 10000 <-- 101|
1|
;
;
20001 10000 <-- 102|
1|
;
;
20002 100 <-- 10000|
1|
;
;
20003 103 <-- 10000|
1|
;
;
20004 10001 <-- 100|
1|
;
;
20005 101 <-- 10001|
1|
;
;
20006 10002 <-- 103|
1|
;
;
```

```
20007 10002 <-- 105|
1|
;
;
20008 104 <-- 10002|
1|
;
;
20009 102 <-- 10002|
1|
;
;
20010 10003 <-- 104|
1|
;
;
20011 105 <-- 10003|
1|
;
;
;
|MARKING
101|
1|
;
102|
1|
;
105|
1|
;
;
|NET_END
EDITOR_INFOS
PAGES
;
PLACE
;
TRANSITION
;
ARC
;
END_EDITOR
```

After translating it back to core using pn2cdl executable we get this target program. There are actually two results files: cycle.xml and cycle.html.

This file was "predicted" by pn2cdl example:

```
HOLD_PREVIOUS
MODULE SYSTEM()
{
VAR
readyConsume: boolean INITVAL true;
channel: boolean INITVAL false;
resource: boolean INITVAL true;

TRANS eat:
enable: !readyConsume;
assign: readyConsume':=true \/ !!readyConsume;

TRANS catch:
enable: readyConsume /\ channel;
assign: channel':=false \/ !(readyConsume /\
                  channel);
        readyConsume':=fale \/ !(readyConsume
                       /\ channel);

TRANS make:
enable: !resource;
assign: resource':=true \/ !!resource;

TRANS send:
enable: !channel /\ resource;
assign: channel':=true \/ !(!channel /\
        resouce);
}
```

And this is what we really get in exampl/cycle directory:

MODULE SYSTEM() {

VAR

readyConsume: **boolean INITVAL true** ;

channel: **boolean INITVAL false** ;

resource: **boolean INITVAL true** ;

Trans eat:

enable: ! readyConsume ;

assign: readyConsume ' := **true** ∨ ! ! readyConsume ;

Trans catch:

enable: readyConsume ∧ channel ;

assign: channel ' := **false** ∨ ! readyConsume ∧ channel ; readyConsume ' :=
false ∨ ! readyConsume ∧ channel ;

Trans make:

enable: ! resource ;

assign: resource ' := **true** ∨ ! ! resource ;

Trans send:

enable: ! channel ∧ resource ;

assign: channel ' := **true** ∨ ! ! channel ∧ resource ; resource ' := **false** ∨ ! !
channel ∧ resource ;

}

Results are same, but in the Boolean functions of enable and assign have no parenthesis.

7.1.3 Nondeterministice Initialization of Boolean Example

In this example all the Boolean variables have a non-deterministic initialization.

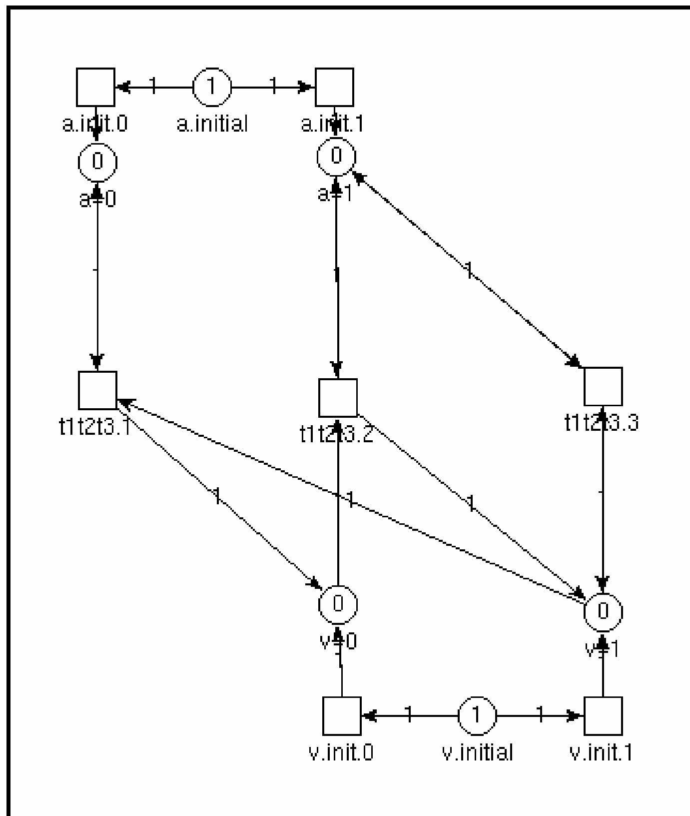
Another thing we can see here is how a complex CDL transition is turned into many petri transition and back to one complex CDL transition.

It's original CDL view can be found in directory **exampl/fig2_1** in the file **fig2_1.cdl**.

```
HOLD_PREVIOUS
MODULE SYSTEM ()
{
    VAR
        a: boolean INITVAL {0,1};
        v: boolean INITVAL {0,1};

    TRANS t1t2t3:
        enable: a \/ v;
        assign: v' := a \/ !v;
}
```

The result is:



After translating it back to core using pn2cdl executable we get this target program. There are actually two results files: fig2_1.xml that can be found in **exampl/fig2_1/** directory and fig2_1.html that can be found in **exampl/fig2_1/** directory. This file was "predicted" by pn2cdl example, note that the expressions for the enable and assign parts are equal to those in the source file and note that t1t2t3.1 t1t2t3.2 and t1t2t3.3 are translated to one CDL transition t1t1t2.

Also, we can ignore the transitions a.initial and v.initial since they indicate only on a non-deterministic initialization:

```
HOLD_PREVIOUS
MODULE SYSTEM ()
{
    VAR
        v: boolean INITVAL {0,1};
        a: boolean INITVAL {0,1};

    TRANS t1t2t3:
        enable: (v /\ a) \/ ((!v /\ a) \/ (v /\ !a));
        assign: a':= (false \/ !(v /\ !a)) /\
                     (true \/ !(!v /\ a)) /\
                     (true \/ !(v /\ a));
                v':= (false \/ !(v /\ !a)) /\
                     (true \/ !(!v /\ a)) /\
                     (true \/ !(v /\ a));

}
```

And this is what we really get in **exampl/fig2_1/** directory:

MODULE SYSTEM() {
VAR
v: **boolean** ;
a: **boolean** ;

Trans t1t2t3:
enable: v ∧ a ∨ ! v ∧ a ∨ v ∧ ! a ;
assign: a ' := **false** ∨ ! v ∧ ! a ∧ **true** ∨ ! ! v ∧ a ∧ **true** ∨ ! v ∧ a ; v ' := **false** ∨ ! v ∧ ! a ∧ **true** ∨ ! ! v ∧ a ∧ **true** ∨ ! v ∧ a ;
}

Results are same, but in the Boolean functions of enable and assign have no parenthesis. Also V and a are not initialized, but if Boolean variable can be initialized to 0 or 1 it is the same as it can not be initialized at all.

7.2 Example with Abstraction of integers

This example demonstrates abstraction of an integer. Two modules (Sender and Receiver) pass data using Buffer module. The data is represented as an integer, and is therefore abstracted. The data's flow is represented by passing the token between the corresponding places.

As explained before, in our translation we treat the abstract places as regular places, and therefore translate them into Boolean core variables.

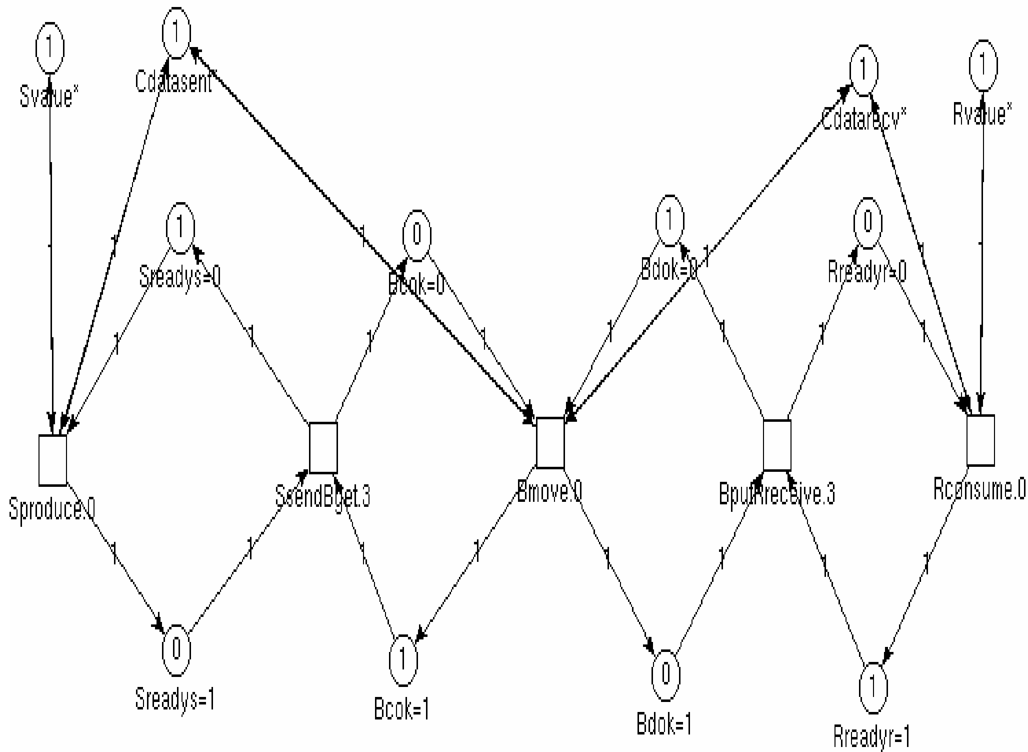
This example (**buffer.cdl**) is a flattened version of the buffer example from the Veritech introduction paper. The prefix of all identifiers in this code is a capital letter. This letter indicates from which of the three modules the identifier come.

This file original CDL view can be found in directory **exampl/buffer** in the file **buffer.cdl**.

The source:

```
HOLD_PREVIOUS
MODULE BUFFER()
{
    VAR
        Cdatasent: integer INITVAL 0;
        Cdatarecv: integer INITVAL 0;
        Sreadys: boolean INITVAL false;
        Bcok: boolean INITVAL true;
        Bdok: boolean INITVAL false;
        Rreadyr: boolean INITVAL true;
        Svalue: integer INITVAL 0;
        Rvalue: integer INITVAL 0;
    TRANS Sproduce:
        enable: !Sreadys;
        assign: Sreadys':=true;
            Cdatasent':=Svalue;
            Svalue':=1;
    TRANS SsendBget:
        enable: Sreadys /\ Bcok;
        assign: Sreadys':=false;
            Bcok':=false;
    TRANS Bmove:
        enable: !Bcok /\ !Bdok;
        assign: Cdatarecv':=Cdatasent;
            Bcok':=true;
            Bdok':=true;
    TRANS BputRreceive:
        enable: Bdok /\ Rreadyr;
        assign: Bdok':=false;
            Rreadyr':=false;
    TRANS Rconsume:
        enable: !Rreadyr;
        assign: Rreadyr':=true;
            Rvalue':=Cdatarecv;
}
```

In the result (follows), we can see the places representing abstract variables with asterisks (*) in their names:



After translating it back to core using pn2cdl executable we get this target program. There are actually two results files: buffer.xml that can be found in **exampl/buffer/** directory and buffer.html that can be found in **exampl/buffer/** directory. This file was "predicted" by pn2cdl example, note that the abstract places keeps on the ‘*’ at the end of their name. We recommend using the algorithm in order to understand the way of treating these places.

```
HOLD_PREVIOUS
MODULE BUFFER()
{
  VAR
     Rvalue*: boolean INITVAL true;
     Svalue*: boolean INITVAL true;
     Rreadyr: boolean INITVAL true;
     Bdok: boolean INITVAL false;
     Bcok: boolean INITVAL true;
     Sreadys: boolean INITVAL false;
     Cdatarecv*: boolean INITVAL true;
     Cdatasent*: boolean INITVAL true;

  TRANS Rconsume:
     enable: Cdatarecv* /\ Rvalue* /\ !Rreadyr;
     assign: Cdatarecv*':=true;
               Rvalue*':=true;
               Rreadyr':=true \/ !(Cdatarecv* /\ Rvalue*
                         /\ !Rreadyr);
  TRANS BputReceive:
     enable: Rreadyr /\ Bdok;
     assign: Rreadyr':=false \/ !(Rreadyr /\ Bdok);
               Bdok':=false \/ !(Rreadyr /\ Bdok);
  TRANS Bmove:
     enable: Cdatasent* /\ Cdatarecv* /\ !Bcok /\ !Bdok;
     assign: Cdatasent*':=true;
               Bdok':=true \/ !(Cdatasent* /\ Cdatarecv*
                         /\ !Bcok /\ !Bdok);
               Bcok':=true \/ !(Cdatasent* /\ Cdatarecv*
                         /\ !Bcok /\ !Bdok);
               Cdatarecv*':=true;
  TRANS SsendBget:
     enable: Bcok /\ Sreadys;
     assign: Bcok':=false \/ !(Bcok /\ Sreadys);
               Sreadys':=false \/ !(Bcok /\ Sreadys);
  TRANS Sproduce:
     enable: Svalue* /\ Cdatasent* /\ !Sreadys;
     assign: Svalue*':=true;
               Cdatasent*':=true;
               Sreadys':=true \/ !(Svalue* /\ Cdatasent*
                         /\ !Sreadys);

}
```

And this is what we really get in **exampl/buffer/** directory:

MODULE BUFFER() {
VAR
Rvalue*: **boolean INITVAL true** ;
Svalue*: **boolean INITVAL true** ;
Rreadyr: **boolean INITVAL true** ;
Bdok: **boolean INITVAL false** ;
Bcok: **boolean INITVAL true** ;
Sreadys: **boolean INITVAL false** ;
Cdatarecv*: **boolean INITVAL true** ;
Cdatasent*: **boolean INITVAL true** ;

Trans Rconsume:
enable: Cdatarecv* ∧ Rvalue* ∧ ! Rreadyr ;
assign: Cdatarecv* ' := **true** ∨ ! Cdatarecv* ∧ Rvalue* ∧ ! Rreadyr ; Rvalue* ' := **true** ∨ ! Cdatarecv* ∧ Rvalue* ∧ ! Rreadyr ; Rreadyr ' := **true** ∨ ! Cdatarecv* ∧ Rvalue* ∧ ! Rreadyr ;
Trans BputRreceive:
enable: Rreadyr ∧ Bdok ;
assign: Rreadyr ' := **false** ∨ ! Rreadyr ∧ Bdok ; Bdok ' := **false** ∨ ! Rreadyr ∧ Bdok ;
Trans Bmove:
enable: Cdatasent* ∧ Cdatarecv* ∧ ! Bdok ∧ ! Bcok ;
assign: Cdatasent* ' := **true** ∨ ! Cdatasent* ∧ Cdatarecv* ∧ ! Bdok ∧ ! Bcok ; Bdok ' := **true** ∨ ! Cdatasent* ∧ Cdatarecv* ∧ ! Bdok ∧ ! Bcok ; Bcok ' := **true** ∨ ! Cdatasent* ∧ Cdatarecv* ∧ ! Bdok ∧ ! Bcok ; Cdatarecv* ' := **true** ∨ ! Cdatasent* ∧ Cdatarecv* ∧ ! Bdok ∧ ! Bcok ;
Trans SsendBget:
enable: Bcok ∧ Sreadys ;
assign: Bcok ' := **false** ∨ ! Bcok ∧ Sreadys ; Sreadys ' := **false** ∨ ! Bcok ∧ Sreadys ;
Trans Sproduce:
enable: Svalue* ∧ Cdatasent* ∧ ! Sreadys ;
assign: Svalue* ' := **true** ∨ ! Svalue* ∧ Cdatasent* ∧ ! Sreadys ; Cdatasent* ' := **true** ∨ ! Svalue* ∧ Cdatasent* ∧ ! Sreadys ; Sreadys ' := **true** ∨ ! Svalue* ∧ Cdatasent* ∧ ! Sreadys ;
}

Results are same, but in the Boolean functions of enable and assign have no parenthesis.

7.3 Example with Program Counters

Program counters are not abstracted in the CDL2PN translation. Therefore we treat them differently and translate them back to integer variables.

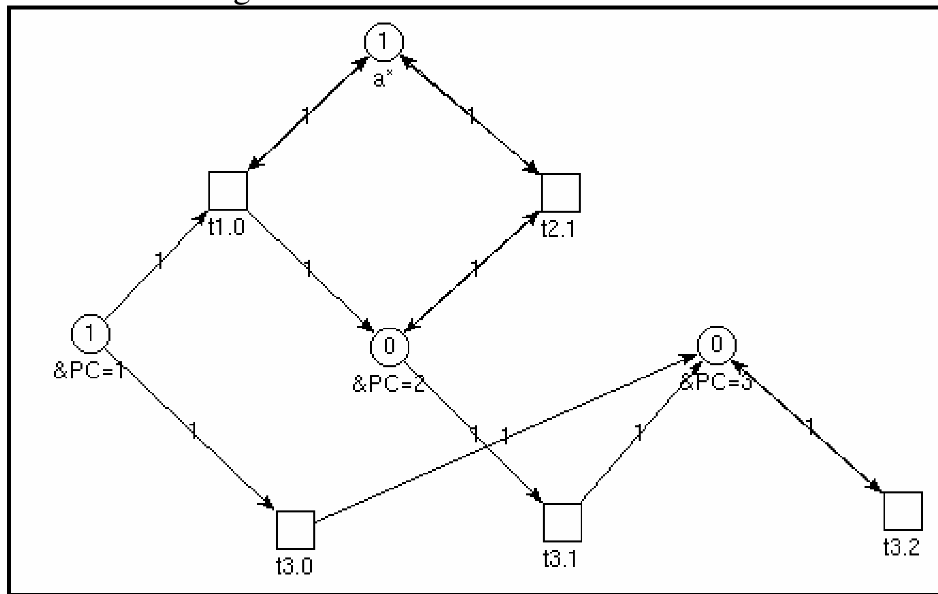
This code (**pc.cdl**) includes both PC and abstract variables. This file original CDL view can be found in directory **exampl/pc** in the file **pc.cdl**.

```
HOLD_PREVIOUS
MODULE SYSTEM()
{
VAR
  &PC: integer INITVAL 1;
  a: integer INITVAL 1;

TRANS t1:
  enable: &PC=1;
  assign: a':=2;
          &PC':=2;
TRANS t2:
  enable: &PC=2;
  assign: a':=1;
TRANS t3:
  enable: true;
  assign: &PC':=3;

}
```

And the resulting net is:



After translating it back to core using pn2cdl executable we get this target program. There are actually two results files: pc.xml that can be found in **exampl/pc/** directory and pc.html that can be found in **exampl/pc/** directory. This file was "predicted" by pn2cdl example

```
HOLD_PREVIOUS
MODULE SYSTEM()
{
VAR
  a*: boolean INITVAL true;
  &PC: integer INITVAL 1;
TRANS t3:
  enable: &PC=3 \/ &PC=2 \/ &PC=1;
  assign: &PC':=3;
TRANS t1t2:
  enable: (a* /\ &PC=2) \/ (a* /\ &PC=1);
  assign: &PC':=2;
         a*':=true;

}
```

And this is what we really get in **exampl/pc/** directory:

```
MODULE SYSTEM() {
        VAR
        a*: boolean INITVAL true ;
        &PC: INTEGER INITVAL 1 ;

        Trans t3:
        enable: &PC = 3 \/ &PC = 2 \/ &PC = 1 ;
        assign: &PC ' := 3 ;
        Trans t1t2:
        enable: a* /\ &PC = 2 \/ a* /\ &PC = 1 ;
        assign: &PC ' := 2 ; a* ' := true \/ ! a* /\ &PC = 1 /\ true \/ ! a* /\ &PC = 2 ;
        }
```

Results are same, but in the Boolean functions of enable and assign have no parenthesis.

8. TODO list:

1. Create a textual CDL program out of the HTML/XML version. Currently the pretty print in the coreparser/util doesn't work. When fixed add an appropriate call to the end of main() in ptc.c
2. To compile with new flat utility from CoreParser.
3. While parsing the arcs part of the .net file pay attention that there is no check if origin and destination elements are from different element type.