

# CDL2PN

A translator from the  
VeriTech CDL language to  
Petri-Nets

## Project Documentation

Uri Dekel  
and  
Nadav Golbandi

CDL2PN Version 1.2

Revised: May 2002

# Table Of Contents

<b>TABLE OF CONTENTS</b> .....	<b>2</b>
<b>ABSTRACT</b> .....	<b>3</b>
<b>CONTACT INFORMATION</b> .....	<b>3</b>
<b>LIMITATIONS ON THE INPUT CDL FILE</b> .....	<b>4</b>
<b>BUILDING AND RUNNING THE PROJECT</b> .....	<b>5</b>
PROJECT LOCATION AND STRUCTURE .....	5
BUILDING INSTRUCTIONS.....	5
<i>Integrate a Module-Flattening System</i> .....	5
<i>Integrate the Core Parser</i> .....	5
<i>Build the CDL2PN system</i> .....	6
RUNNING IN THE PROJECT DIRECTORY .....	6
<b>ADDING NEW OUTPUT FORMATS</b> .....	<b>8</b>
<b>RUNNING PNK</b> .....	<b>9</b>
<b>CODE DOCUMENTATION</b> .....	<b>10</b>
TRANSLATION PROCESS OVERVIEW .....	10
NAMING CONVENTIONS.....	10
DATA STRUCTURES .....	11
<i>Dictionary / Hash</i> .....	11
<i>Petri-Net Structures</i> .....	11
<i>Valuation Structures</i> .....	12
ALGORITHMS .....	12
<i>Translation of Multiple-Modules</i> .....	12
<i>Translation of an Individual Module</i> .....	12
<i>Translation of An Individual CDL Transition</i> .....	13
<i>Output to a PNK/PNMK Format</i> .....	14
FILES LISTING .....	14
<i>Files Listing For the ModularPetri Directory</i> .....	14
<i>Files Listing For the ModularPetri/DataStructure Directory</i> .....	16
<b>EXAMPLES</b> .....	<b>17</b>
EXAMPLES WITH BOOLEANS .....	17
<i>Single Variable and Transition Example</i> .....	17
<i>Real World Example: Consumer-Producer</i> .....	19
<i>Nondeterministic Initialization of Booleans Example</i> .....	20
<i>The code can be found in fig2_1.cdl file:</i> .....	20
<i>Another Nondeterministic Initialization of Booleans Example</i> .....	21
MODULE SYNCHRONIZATION EXAMPLE .....	22
ABSTRACTION AND PROGRAM COUNTERS .....	25
<i>Abstraction of Integers – The Buffer Example</i> .....	25
<i>Program Counters Example</i> .....	27

## **Abstract**

The CDL2PN (Core Definition Language To Petri Net) system is part of the Veritech Project.

Our system provides translation into two common file types for representing Petri nets. The default one is the PNK format used by the Petri Net Kernel project. We also have a module for creating output in PNML (Petri Net Markup Language), an XML-based language for the notation of Petri-Nets

## **Contact Information**

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

This project was written by Uri Dekel ([udekel@cs.technion.ac.il](mailto:udekel@cs.technion.ac.il)) and Nadav Golbandi ([golbandi@cs.technion.ac.il](mailto:golbandi@cs.technion.ac.il)). We have received support from Dr. Katerina Pokozy. ([pokozy@cs.technion.ac.il](mailto:pokozy@cs.technion.ac.il)).

The Veritech project is hosted at the SSDL lab, headed by Mr. Shahar Dag.

# Limitations on the input CDL file

Before running the translation on actual CDL input, it is useful to become acquainted with the various limitations our translator imposes.

Our translation process currently accepts only a subset of CDL. Following is the list of things which our translator cannot handle or handles in a special way:

- `HOLD_PREVIOUS` must be specified. Variables keep their old values if not changed.
- Enumerations, sets and sub ranges are not supported.
- All integer variables which do not serve as program counters are abstracted to a single Petri place, which emulates all accesses to that variable.
- All program counters must be defined as integer, and must receive integer values. They must also be initialized.
- No complex assignments or mathematical evaluations (such as  $1+2$ ) are allowed.
- Non-PC abstract variables should be initialized, otherwise a warning will follow if their values are read in a transition.
- Nondeterministic initialization of boolean must be done as placing  $\{0,1\}$  as the initializing value. Not specifying an initialization value will result in an undefined value (There will be no token in any of the Petri-places representing that variable).
- We do not support the **relation** clause in assignments. This does not reduce the descriptive power of the system because all the relations can be expressed using **assign** clauses.
- Comparisons of integral variables to integral literals will not work because all integral variables are considered abstract or program counters.

If one of the above condition does not hold, the program will halt and an explanatory message will be printed in the log file.

# **Building and Running The Project**

## **Project Location and Structure**

The project files are located in the `/home/veritech/users/udekel/CDL2PN2` directory on the CS files system in the Technion. This is the root directory of the project. If you'd like to build elsewhere, please copy this directory recursively.

The CDL2PN makefile assumes that the current core parser can be found at `/home/veritech/VeriTech_project/CoreParser`. The makefile has to be changed accordingly if the parser is moved, by changing the `PARSER_DIR` variable.

The CDL2PN directory contains a makefile and the following subdirectories:

- **Bin** - Contains the the resulting executable and the CDL2PN library.
- **Doc** – Contains this document.
- **Hdr** – Contains the header files for the project.
- **Obj** – Contains intermediate object files created during a build.
- **Samples** - Contains various sample files.
- **Src** – Contains the C files for the project

The CDL2PN makefile needs to use the core parser and the flattener. The `PARSER_DIR` and the `FLAT_DIR` variables in the makefile are used to locate them.

## **Building Instructions**

Follow the steps as indicated in the following subsections.

### **Integrate a Module-Flattening System**

We are using the version 1.2 of the VeriTech module flattening utility which was available at run time the time of release, and should be located at `../utils/flat`. The **`FLAT_DIR`** variable in the makefile refers by default to the newest version. If you wish to use a different version, please change this variable.

relatively to the CDL2PN directory. If you wish to integrate a new version of the flattener, either update that “flat” directory, or change the **`FLAT_DIR`** variable in the CDL2PNmakefile.

If you want to use the current version, no further steps have to be taken.

### **Integrate the Core Parser**

We are using version 1.4 of the VeriTech core parser library which was available at the time of release. The **PARSER\_DIR** variable in the makefile refers by default to the newest version. If you wish to use a different version, please change this variable.

The makefile automatically looks for the parser in `/home/veritech/VeriTech_project/CoreParser`, a link into the most recent version. You can change the **PARSER\_DIR** variable to use a different parser. Otherwise, no further steps have to be taken.

### **Build the CDL2PN system**

Go into the **CDL2PN** directory. The system is built by using the **make** command. Before performing **makemaking**, it is recommended to erase all the contents of the **bin** and **obj** directories. object files by typing **rm obj/\*.o**. Now type **make**, this will take a fewseconds.

### **Producing a different output format(PNK/PNML)**

In order to change the type of output produce (PNML instead of the default PNK), one has to go to the ModularPetri directory, open the **ModularPetri.c** file, replace the comment on one of the related includes (PNML or PNK), and then open the **CoreToModularPetri.c** file, replace the required include, and then replace the comment at the end of the file with the actual function call. A full rebuild isnecessary.

### **Running in the Project Directory**

The executable for the parser is in the **bin** directory, a file named **cdl2pn**. A static library is also created in case a user wishes to integrate the functionality into a different system. The executable file contains everything needed (including the parser) and can be freely moved elsewhere in the same operating system.

To execute the parser, type:

#### **cdl2pn Inputfile Outputfile Format**

All three parameters are mandatory.

**Inputfile** is the CDL file which will be translated. It must adhere to the requirements stated earlier.

**Outputfile** is the name (without extension) of the output file and the log file.

**Format** is either PNK or PNML

As soon as execution begins, a log file named **Outputfile.log** (where **Outputfile** was specified as a parameter) is created. This log will contain reports, warnings and error messages about the translation process.

If parsing is successful, the user will get a few lines of information on the standard error stream (screen). A file named **Outputfile.net** or **Outputfile.pnml** (where **Outputfile** was specified as a parameter) will be created.

If parsing is unsuccessful, the program will halt and the user will receive a message referring to the log file. If the log file is nearly empty, it means that the CDL code had some syntax error that crashed the core parser, and the Petri-net translation has not begun.

The input file is a CDL file which adheres to the requirements stated earlier in the document. The log file lists some information, including tables listing the valuations that took place for each CDL transition. The actual petri net is written into a file named "**test.out**" in the same directory, which should be renamed according to its type (for example **.net** if you usePNK).

The log file should always generated (+ on line printing to the screen). The name should be according to the inputfile.

If the net **test.out** file has not been created, it indicates that a severe error has occurred, you will typically find an explanation at the end of the log file which is always created. If the log file is nearly empty, it means the CDL code had some syntax error which didn't go past the interpretation stage, and therefore no attempt at translation to a petri net were done.

The log file contains a transcript of the execution of the system. All error reports and warnings are recorded into the log file (the log file is the only way to know of warnings). The log file contains two useful types of information. For each Petri-place which is created in the system, a line exists in the log which explains the origin of this place (for example, which CDL variable and which value it represents).

The log file also contains the truth tables representing the transition. Watch for a title saying "**Analyzing CDL transition named XXX**" where XXX is the CDL transition name. Following is a line specifying the order of variables in the table. The next lines specify the valuations. Each valuation has an index, and then a set of values corresponding to the variables in the ordered specified at the top of the table. Following is the **EBL** (enable) value. If it is a false value (0), it means that this transition cannot occur for the given valuation. However, if the EBL value is 1, then the line resumes with a listing of all the variables which are assigned (perhaps to their existing values) as a result of this transition. In the actual Petri-nets, each transition will have an index into the table, specifying the line for which it stands.

SD: you should put here an example of the truth table

Note that our system translates nets without actually laying them out. This means that they are topologically correct (all the connections are in place), but you will have to lay them out visually using **PNK**.

By default, the log file contains all warnings, errors and messages. It is possible to instruct only important messages and warnings to appear in the file, by changing the `REPORTLEVEL` and `WARNINGLEVEL` variables in the `MPgeneral.h` file.

## Adding new output formats

In this release, CDL2PN permits output in PNK or PNML format. To add new formats, you need to create a new output module (use `ModularPetriToPNK.c` as an example), and then update `ModularPetriToFile.c` and `wrapper.c` to accept the new format.

# Running PNK

PNK is a graphical system for editing and simulating Petri-nets. Because it requires **Python** to run, it cannot be installed on csd at this time. Instead, one has to perform the following:

Login to sd37nt19.cs.technion.ac.il from an X-terminal. This computer is run by Shahar Dag, and one needs an account to use it. (Contact [ssdlhelp@cs.technion.ac.il](mailto:ssdlhelp@cs.technion.ac.il) for info). You may want to transfer your **.net** files by FTP to this system.

Once you are logged in, you need to set the X-terminal display to your system. If you use **eXceed**, you may need to add the window serial number. For example: **setenv DISPLAY computer\_name:0.0**. (you can use the ip number instead of the computer name)

Before executing the PNK system, you need to execute the following lines. It is available as a script under the PNK scripts directory.

```
setenv PYTHONPATH  
/usr/local/VeriTech/PetriNetKernel/PNK_2.0/Kernel
```

```
setenv PYTHONPATH  
{PYTHONPATH}:/usr/local/VeriTech/PetriNetKernel/PNK_2.0/Kernel/NetTypes
```

```
setenv PYTHONPATH  
{PYTHONPATH}:/usr/local/VeriTech/PetriNetKernel/PNK_2.0/Editor
```

```
setenv PYTHONPATH  
{PYTHONPATH}:/usr/local/VeriTech/PetriNetKernel/PNK_2.0/Applications
```

Now you can execute the editor, by typing the following line (replacing \$1 with a net file name). You may also use a script.

```
python  
/usr/local/VeriTech/PetriNetKernel/PNK_2.0/Editor/PNKedit.py ST_Specification $1
```

In some UNIX workstations, PNK may not work because the workstation does not allow a window to be on your system for security reasons.

**For more information on PNK and its file formats, please refer to the documentation under the PNK\_2.0 directory.**

# Code Documentation

## Translation Process Overview

The translation process begins as a regular execution of the core parser. The input CDL file is parsed and the core data structure is built. Our part of the process is activated at the end of the CDL processing, from the wrapper part. Naturally, it is activated only if the parsing of the CDL file itself was successful.

Our translation process works for flat modules only. If the code contains more than one module, an external flattening library is used, and we obtain a single module. Consult the documentation on that system to see how it works.

It is recommended to consult the Limitations section in the [Running Instructions](#) to running instructions chapter to become acquainted with the limitations of our translation process.

Our translator works by building a new internal representation of the resulting Petri net from the core data structure. For this, we are using three major data structures: **MPplace**, **MPtransition** and **MPconnector**, which naturally correspond to Petri-net places, transitions and connectors. For more information about these structures, please consult the [data structures](#) data structures section. All these objects for the net corresponding to a CDL module are located in an **MPclass** object.

Once we have built our representation of the Petri net, all that remains is to produce output in a particular output format. We currently support PNML and PNK (PNK is the default), but more formats can be easily added in the future. This involves going over our **MPclass** objects and writing the net in the required output format.

The PNK website can be found at: <http://www.informatik.hu-berlin.de/top/pnk/index.html>

The PNML website can be found at <http://www.informatik.hu-berlin.de/top/pnml/>

## Naming Conventions

In order to allow maximal flexibility, we have moved all the code that gives names to Petri-net elements to a separate module (the **names.c** file). The functions there create names for the Petri items based on the original CDL structure. Described below are the conventions in the current version of the code:

- Petri places which correspond to nonabstract variables in the CDL module, have a name of the format  $X=Y$  where  $X$  is the original variable name, and  $Y$  is the current value. For booleans this means F or T, for program counters it is the literal value.

- Petri places which correspond to an abstract variable in the CDL module have a name of the format **X\***, where **X** is the original variable name.
- Transitions corresponding to CDL transitions (see explanation later) have a name format of **X.Y**, where **X** is the CDL transition name, and **Y** is the index of the valuation for which this transition has been created. This number corresponds to the index in the table which appears in the translation log file.
- In the case of a nondeterministic initialization of a variable, a place named **X.initial** is created, where **X** is the variable name. A transition named **X.initialize** follows.
- The names for connectors are not important as they do not appear in the final output, and are typically the concatenation of the names of their two sides.

## Data Structures

### Dictionary / Hash

In order to store and retrieve the various objects in the system, we have built a simple data structure which is indexed by a string, which we named **Key**. One can think of this structure, which we called **Dictionary** as a hash-table, although we have implemented it as a simple linked list of nodes, each with a key and an item. It can be easily replaced with an actual hash implementation if a performance boost is needed in the future. Because performance was not important for this implementation, Keys and Names are often used interchangeably, although we are using function calls to get the key from a name. Currently the hashing function is simply the identity functions, but again, it is easy to modify this.

### Petri-Net Structures

Since future extensions may support multiple modules, we instantiate an **MPclass** structure for every CDL module, including the system module.

The **MPclass** structure contains three dictionaries, for holding the places, transitions and connectors in that class. The name of the **MPclass** structure is the same as the module name. The manipulation of the fields of this structure should be done using the provided functions.

The **MPplace** structure represents a Petri-net place. Each place has a name, a pointer to the **MPclass** object containing it, a serial number (used for some output formats) and an indicator of the number of tokens it contains initially. Functions are provided for manipulation of these fields. A function for retrieving the key of the place from its name (for hashing purposes) is available as well.

The **MPtransition** structure represents a Petri-net transition, and is similar to the **MPplace** structure, except that no field for the initial tokens is provided.

The **MPconnector** structure represents a connector inside the Petri-net. Each connector has a unique name and serial number, which is used for indexing and hashing purposes. In addition, it has two pairs of fields, one pair corresponds to the source node of the connector, and one to the destination node of the connector. Each of these two pairs consists of a **void\*** pointer and of an **enum** field specifying what kind of object the pointer refers to. This is because each side of the connector can refer to either a transition or a place.

## **Valuation Structures**

As explained in the algorithm overview, we are producing for each CDL transition all the possible valuations of the set of variables that take part in the enable part, and then for each valuation where the enable part becomes true, we create the larger valuation which contains the variables from both the enable part and the assign part, which we use to produce the actual Petri transition.

For each variable in a particular valuation, we have a **VariableInValuation** structure. This structure holds the variable name, its type, and its current value. All the variables for a particular valuation are collection in a **Valuation** structure where they are arranged as an array. There are functions provided for resetting the valuation, advancing to the next combination, and checking the value of a particular variable.

## **Algorithms**

### **Translation of Multiple-Modules**

As explained before, our system deals only with single-module systems. However, because we are planning for future extensions, our system lays some infrastructure for multiple-module support. Each module should be represented using an **MPclass** object. Many of our functions work by going over the list of **MPclass** objects (at present there should be only one) and execution another function on each of them.

Also, Some preliminary actions are performed on the original core structure, and would remain the same for a multiple-module system. For instance, we perform a search of the core structure is done to detect all the integer literal values and store them in a special array, for later treatment of program counters (see later in this section).

### **Translation of an Individual Module**

Our system essentially translates a single CDL with no embedded module into a single Petri-net.

The translation begins by going over all the variables declared in the module. (Because no embedded modules are allowed and no parameters are passed, these are necessarily all the variables which can appear in that module). For each variable, we create the corresponding Petri places.

For each variable, we determine the number of corresponding Petri-places according to its value. For a boolean variable, we create two places, for the false(0) and true(1) values accordingly.

For abstract and program-counter variables, the creation of these places is problematic. The range of integers is practically infinite, and determining the possible valuations of an integer variable in the program is an undecidable problem. We have therefore taken the following strategy: For all integer variables which are not program counters, we are abstracting them to a single Petri place. Therefore values will not be changed, but tokens will go to- or from them, preserving the data-flow itself, this is not a problem because Petri-nets are mostly used to model the data flow and not the actual values. However, program counters are impossible to abstract. We therefore assume that a program counter can receive the value of any integer literal that appeared in the program (remember that we collected these earlier). We create corresponding places. We also do not allow any calculation expressions over the integers.

Once the places are created, initial tokens can be placed. If the initialization of a variable is deterministic, we place a token in the corresponding place for each of the variables. However, if the initialization is nondeterministic, we create an initial place for that variable, in which we place the token, and then connect it using two separate transitions to both of the actual variable places, utilizing the nondeterministic nature of such Petri-net transitions to do the nondeterministic initialization.

Now that all the places are ready for the variables, our program starts iterating over each of the CDL transitions, and translates each of them individually.

### **Translation of An Individual CDL Transition**

Our system treats each of the CDL transitions individually. We are doing the translation by the method of using a truth table.

We begin by collecting all the variables that appear in both the enable part of the transition, and in all the assignments and their expressions. We are collecting them into a **valuation** structure.

Because we are building a truth table, we shall iterate over all the possible combinations of valuations of the enable valuation. We do this using a simple "counter algorithm". All the variables begin with their minimal values. When incrementing is requested, the rightmost variable is incremented once. If it has reached its maximal value, then it is reset to the minimum, and its left neighbor is incremented, and so forth. If all the variables are at the maximum, then we would be incrementing the "carry", which means that we have finished going over all the combination. The possible values for each variable depend on its type. Booleans have two values, Abstracts have a single value (a constant) and program counters can have all the literal values in the program.

In each iteration, we increment the valuation once, triggering the process described above. We now take the values of all these variables, and evaluate the **enable** expression of the CDL transition for the current values. If we have received a false

result, it means that the enable part does not hold for this valuation, and we can move to the next. If it is true, it means that the enable condition holds and we can start handling the assignments.

SD: but even if the enable is true the assignment part can still prevent the transition from executing

If the enable condition does indeed hold, then the CDL transition will occur when the variables in the **enable** condition have the current value. We therefore create a new Petri transition (which we name after the CDL transition and index according to the index of the valuation). We pass connectors from the places corresponding to the current values .

Now that we arranged the transition to "fire", we have to take care of the assignments. We create a new valuation, which we shall call the **assignment valuation** (we call the previous one the **general valuation**). This new valuation will hold all the variables who are affected at the end of this transition. The first ones we add are of course the variables being reassigned. Each of them is added to the valuation with its new value: the result of evaluating its assignment expression using the general valuation (which has values for all variables in the transition). However, this is not enough. All the variables which participate in any of the expressions but are not reassigned will lose their token. We therefore add all of them into the assignment valuation, with their new value being the old value. Once the assignment valuation is ready, we create connectors from the Petri-transition we have created into each of the places which represent the new value of a variable.

The result of this entire process is that the transition now occurs as needed, and after the transition will take place, all the tokens would go to the correct places. While it is true that there may be ways of optimizing this, the truth table system ensures a correct translation, which was the priority of this work.

The entire truth table appears in the log file which is created as a result of the translation. Petri-Transitions for CDL transitions have an index after their name which refer to the corresponding line in the table.

### **Output to a PNK/PNMK Format**

The output creation is very straightforward, and is based on the grammar rule for the required target format.

### **Files Listing**

This section will list the various files we are using, and what each of them is used for. We are listing only the header files, most of them have associated implementation files.

Please refer to the source code for the internal documentation.

### **Files Listing For the ModularPetri Directory**

Filename	Purpose
CoreToModularPetri	This is the main file, which controls the translation process. It scans the modules, collects variables, creates places for them, etc. The CDL transitions themselves are handled in <b>ExpressionToPetriNet</b> , and the output is in <b>ModularPetriToPNK</b> .
ExpressionToPetriNet	This module translates a transition into a Petri-net structure, as explained in the algorithms section.
ExpressionUtilities	This module contains utilities for handling expressions, such as predicates (checking simple things about expressions), conversion to DNF, displaying as string, etc.
Mpclass	Definition and functions for the MPclass structure (Petri-Net class)
Mpconnector	Definition and functions for the MPconnector structure (Petri-Net connector)
MPgeneral	General definitions for the translators. In addition, this module controls the logging, warning and error reporting mechanisms.
Mpplace	Definition and functions for the MPconnector structure (Petri-Net place)
Mptransition	Definition and functions for the MPtransition structure (Petri-Net transition)
ModularPetri	This is the entry point to our system. In order to minimize changes to the CoreParser system, the C file actually includes all the other C files in the system, and the H file includes a single function which should be called from the wrapper file, which is activated by the parser. SD: ???
ModularPetriToFile	“Switchboard” for initiating output.
ModularPetriToPNK	Output module to PNK format (Petri-Net Kernel)
ModularPetriToPNML	Output module to PNML format (Petri-Net Markup Language)
Names	This module contains functions which apply the naming conventions we have used.
Valuation	This module defines the valuation structure and the various functions (Such as incrementation) to manipulate them.
Wrapper	The wrapper file (.c) only is the entry point. It is the implementation for the <b>wrapper.h</b> file defined in the CoreParser.

Other important files include:

**makefile** - The makefile for the ModularPetri system

## **Files Listing For the ModularPetri/DataStructure Directory**

<b>Filename</b>	<b>Purpose</b>
data_structure	Definition and utilities of the generic element which we are using for our structure (essentially a linked list node).
Dictionary	A dictionary (lookup by key of void ptr objects) system
List	An ordered list system.

## Examples

In this section we shall give various examples and results, which illustrate the capability of our program. All these examples are of flat modules, some of them have been manually flattened from their original structure. In the future, the flattening module will be able to do that automatically. All examples can be found under the **Examples** directory of the system, with the results placed in its **Results** subdirectory. The screenshots are under the **Screenshots** subdirectory.  
SD: it is time to recheck all examples using the flattening utility

The net diagrams provided here are based on screenshots of PNK screens. All these nets were created automatically, but were layed out manually (somethink that PNK and our translator do not perform). The number inside each place represents the number of initial tokens it holds. The “1” on the connectors mean that all the connectors are considered equal.

In this version, Boolean variables had 0 or 1 values, instead of the T and F of newer versions.

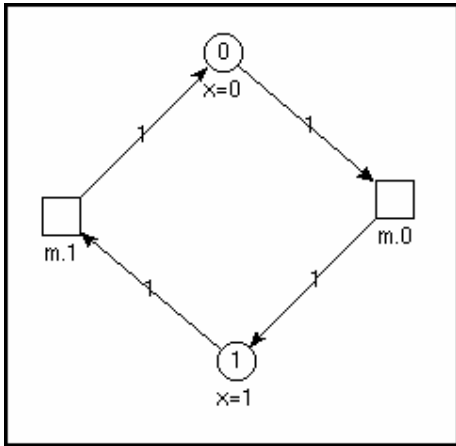
## Examples with Booleans

### Single Variable and Transition Example

Following is the most simple example possible: A single variable and a single transition in a single module. This example can be found in the **fig1\_2.cdl** file. The source CDL is:

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

The result is a simple diamond form:

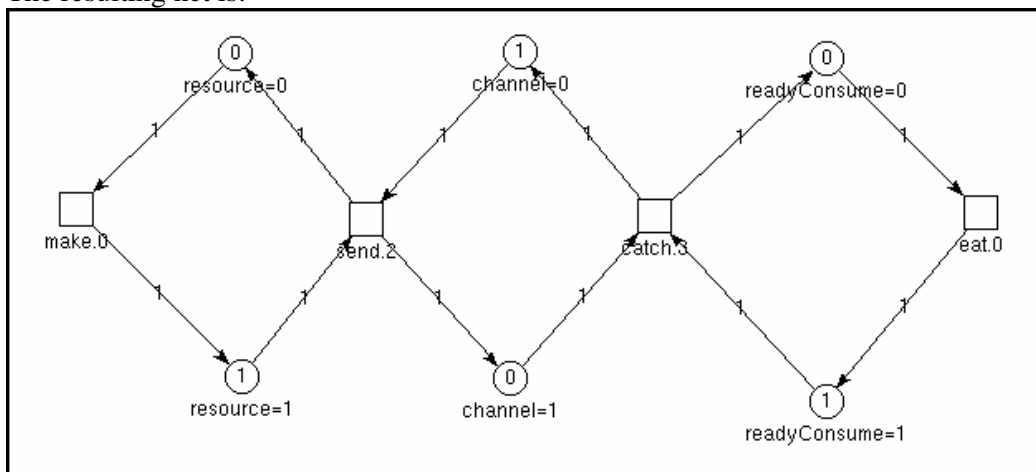


## Real World Example: Consumer-Producer

Follows is a real world example, of a consumer-producer system, listed in **cycle.cdl**. The existence of a resource in this case is indicated by the **resource** flag.

```
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:



## Nondeterministic Initialization of Booleans Example

The following example is more elaborate. All the boolean variables have a nondeterministic initialization. Therefore (as explained in the algorithms section), we have two extra transitions and a place for each such variable. We also see here how a complex CDL transition is turned into many Petri transitions.

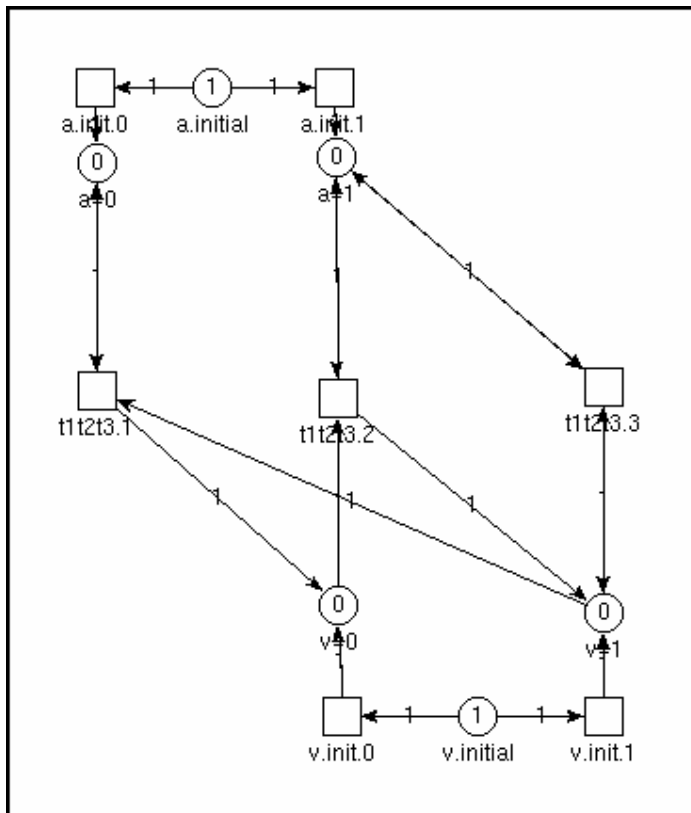
The code can be found in **fig2\_1.cdl** file:

```
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:

SD: here you should add the log file or at list the truth table



## Another Nondeterministic Initialization of Booleans Example

This example is similar to the previous one, this time with three variables. It appears in the **drawing1.cdl** file.

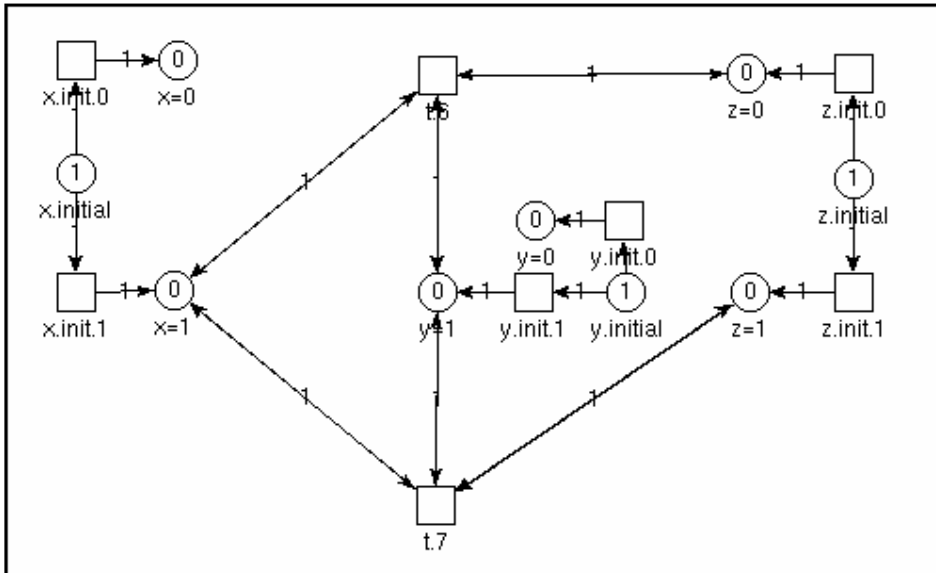
```

HOLD_PREVIOUS
MODULE SYSTEM ()
{
  VAR
    x: boolean INITVAL {0,1};
    y: boolean INITVAL {0,1};
    z: boolean INITVAL {0,1};

  TRANS t:
    enable: (x /\ y) \/ !y /\ (y /\ !z);
    assign: y' := true;
}

```

The result is:



## Module Synchronization Example

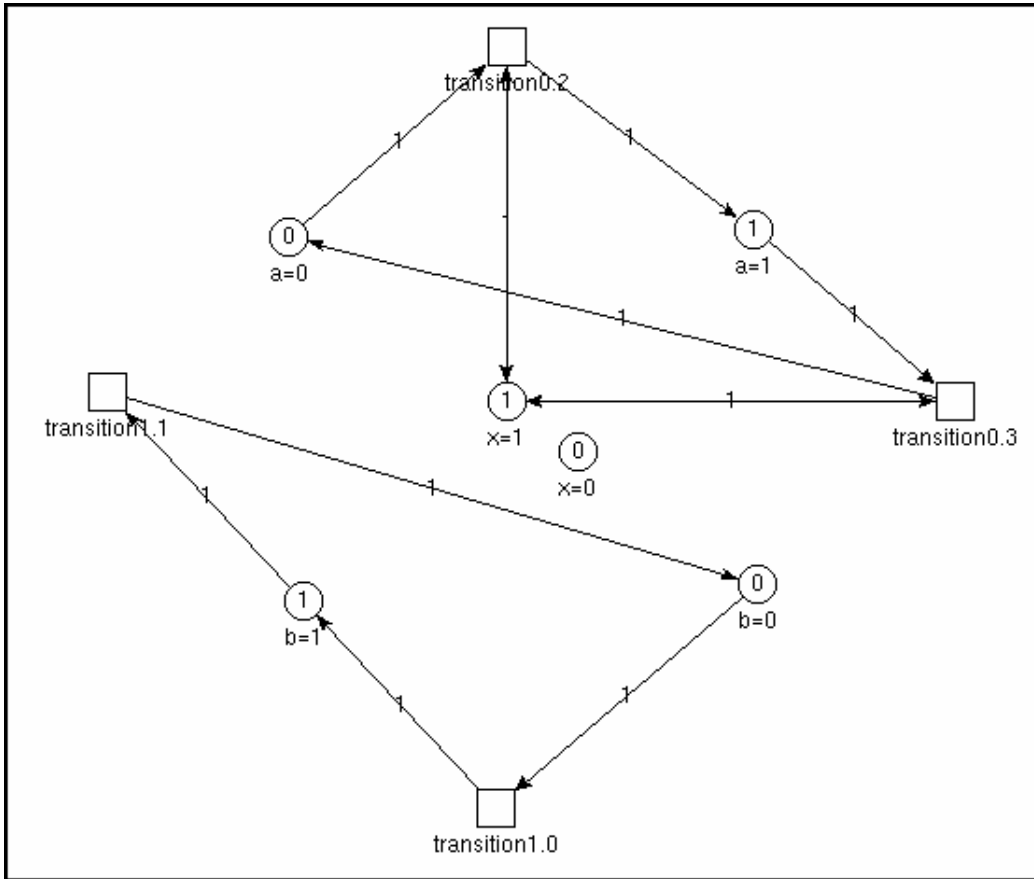
Let us examine the following CDL code:

```
HOLD_PREVIOUS;
    VAR x: boolean INITVAL true;
MODULE m1 () {
    VAR
        a: boolean INITVAL true;
    TRANS transition_0:
        enable: x;
        assign: a' := !a;
}
MODULE m2 () {
    VAR
        b: boolean INITVAL false;
    TRANS transition_1:
        enable: true;
        assign: b' := !b;
        relation: x;
}
MODULE SYSTEM() {
Some line which either fully synchronizes both modules or
makes them fully asynchronous.
}
```

In the case that we would like these two modules to work asynchronously, we would have a flatten version which would look like this (**ex12.cdl**):

```
HOLD_PREVIOUS
MODULE SYSTEM ()
{
    VAR
        x: boolean INITVAL true;
        a: boolean INITVAL true;
        b: boolean INITVAL true;
    SD: why is b initialized to true
    TRANS transition0:
        enable: x;
        assign: a' := !a;
    TRANS transition1:
        enable: true;
        assign: b' := !b;
    SD: what happened to the relation part
}
```

In the result (follows), we can see that the two transitions occur independently:



On the other hand, If we would like both of the modules to work in full synchronization, so that both transitions can only occur together, the code would look like this (**ex13.cdl**):

```
HOLD_PREVIOUS
```

```
MODULE SYSTEM ()
```

```
{
```

```
  VAR
```

```
    x: boolean INITVAL true;
```

```
    a: boolean INITVAL true;
```

```
    b: boolean INITVAL true;
```

```
SD: why is b initialized to true
```

```
  TRANS synchronized:
```

```
    enable: x /\ true;
```

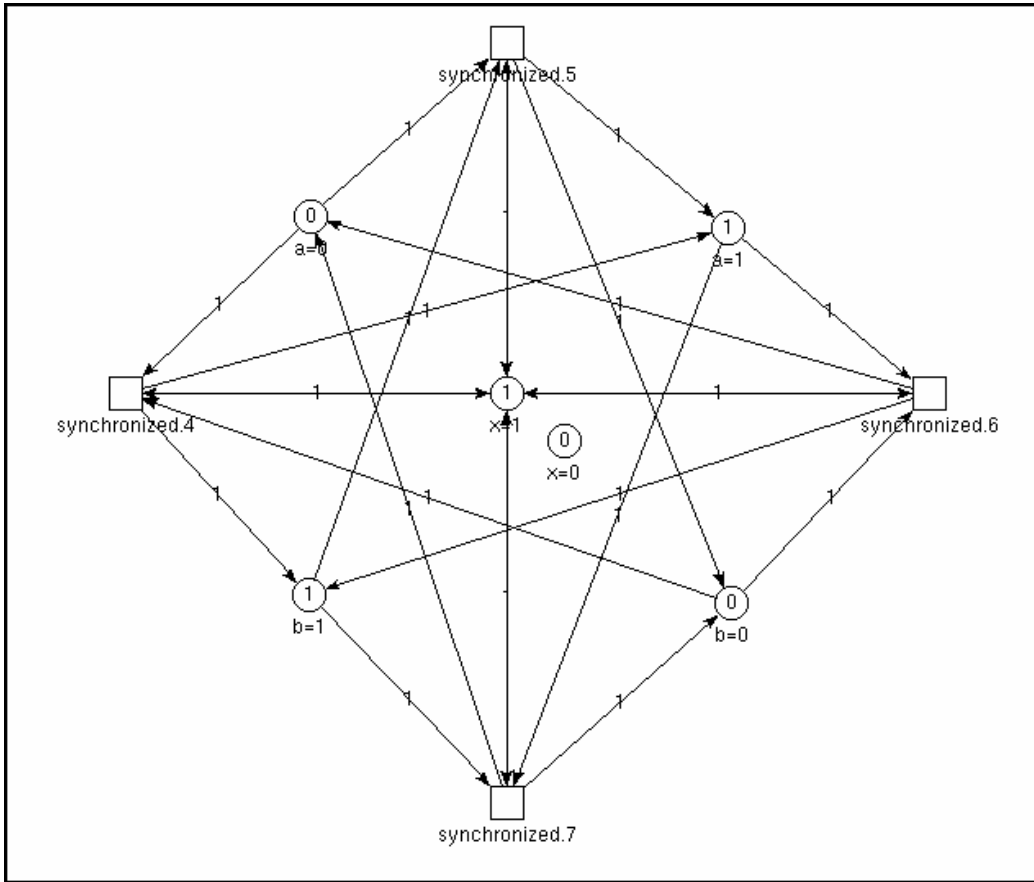
```
    assign: a' := !a;
```

```
           b' := !b;
```

```
SD: what happened to the relation part
```

```
}
```

And in the result we can now see that the transitions are now coupled:



We can verify this by consulting the log file which was produced by the translation, following is the truth table:

```

Order Of Variables: [ x a b ]
Valuation 0: 0 0 0   EBL:0
Valuation 1: 0 0 1   EBL:0
Valuation 2: 0 1 0   EBL:0
Valuation 3: 0 1 1   EBL:0
Valuation 4: 1 0 0   EBL:1   NEW VALUES: a=1   b=1   x=1
Valuation 5: 1 0 1   EBL:1   NEW VALUES: a=1   b=0   x=1
Valuation 6: 1 1 0   EBL:1   NEW VALUES: a=0   b=1   x=1
Valuation 7: 1 1 1   EBL:1   NEW VALUES: a=0   b=0   x=1

```

# Abstraction and Program Counters

## Abstraction of Integers – The Buffer Example

To demonstrate how our system abstracts integer, we have manually

SD: now use the flattener

flattened the buffer example from the Veritech introduction paper, where two modules (Sender and Receiver) pass data using a Buffer module. The data is represented as integer, and is therefore abstracted. The data flow is indicated by the passing of the token between the corresponding places.

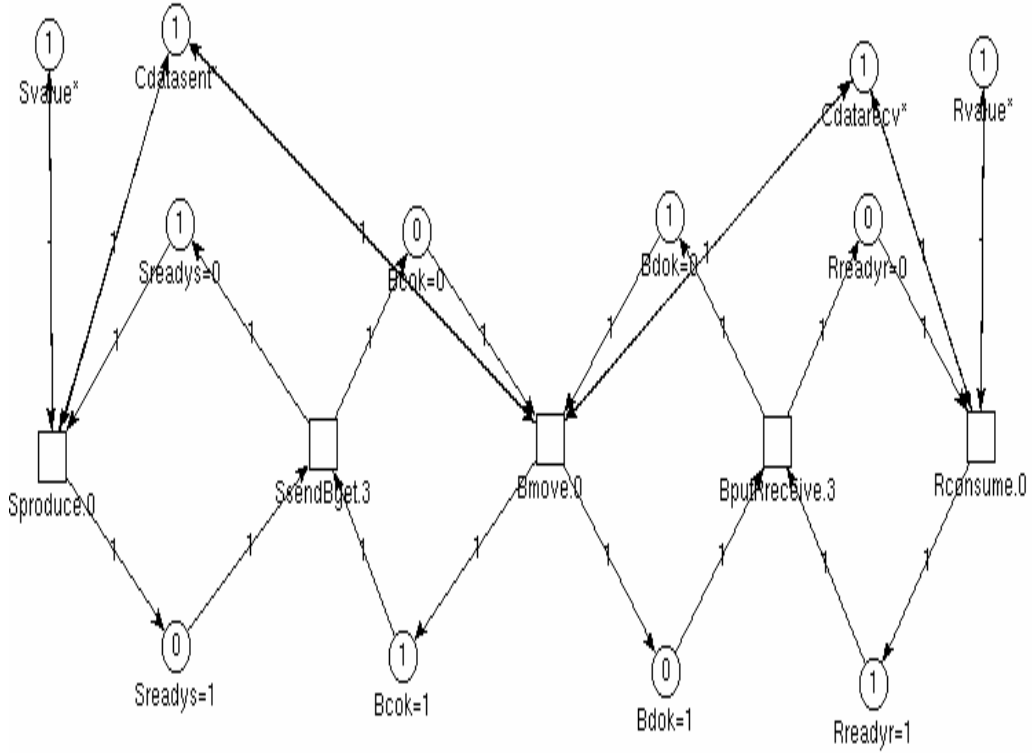
In the following code (**buffer.cdl**), all identifiers are prefixed by a letter indicating which of the three modules they have come from:

SD: put here the original CDL program

```
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:

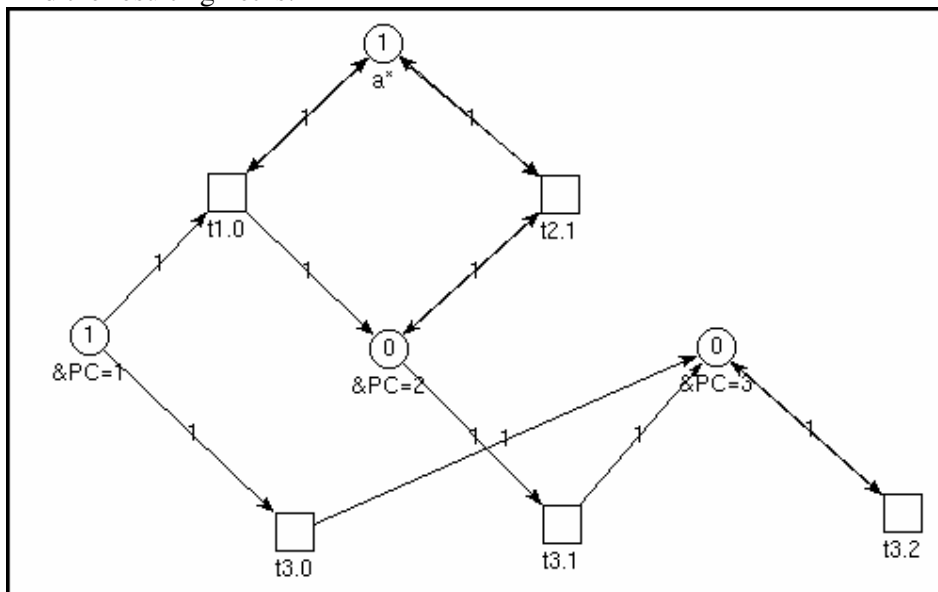


## Program Counters Example

Program counters in our system are integers which are not abstracted. There is a Petri place for every possible valuation of the program counter. The following code utilizes both program counters and abstract values (**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:



SD: fix all spelling mistakes