

Project Documentation

PN2CDL

A translator from Petri-Nets to the
VeriTech CDL language

Submitted by:
Efrat Dayagi

1. Abstract

The PN2CDL (Petri-Nets to Core Definition Language) system is part of the Veritech Project.

The current version includes translation of data from the Petri-Nets data structures into the Core data structures.

Future improvements will include translating common file types for representing petri nets into the petri nets data structures, as well as translating data from core data structures into a core program.

2. Contact Information

The heads of the Veritech project are Prof. Shmuel Katz (katz@cs.technion.ac.il) and Prof. Orna Grumberg (orna@cs.technion.ac.il) from the Technion.

The project was written by Efrat Dayagi (sdayagi@techst02.technion.ac.il).

We have received support from Dr. Katerina Pokozy (pokozy@cs.technion.ac.il).

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

3. Building and Running the Project

3.1 Project Location and Structure

The project files are located in the `/home/veritech/users/dayagi_sl/PN2CDL` directory on the CS file system in the Technion.

This directory includes only the relevant files for the final translation, and therefore we cannot check the translation from this location yet.

The PN2CDL directory contains 6 subdirectories:

1. **Hdr** – This is the directory containing the header file we use.
2. **Src** – This is the directory containing the source code we use.
3. **Obj** – Contains the object files that are created.
4. **Bin** – Contains the exe file.
5. **Samples** – Contains various examples.
6. **Docs** – Contains a version of this document.

3.2 Checking current version

Since the current version of the project supports only translating data from petri nets data structure into core data structure, we cannot run it directly from the location above. In order to be able to run and check the existing files we use the vice-versa project (CDL2PN). We first translate a core program into petri nets data structure (part of the CDL2PN project). Then we translate the petri nets data structure into the core data structure (our project), and check the results.

This workspace is located in `/home/veritech/users/dayagi_sl/CDL2PN` directory on the CS filesystem in the Technion.

You can build it elsewhere by copying this directory.

It contains the same subdirectories as the **PN2CDL** directory.

In order to build this system go into the **CDL2PN** directory and type the **make** command. Before making it is recommended to erase all object files by typing **rm obj/*.o**.

You can find more running instructions for this system in the CDL2PN documentation (written by Uri Dekel and Nadav Golbandi).

The CDL2PN translation is located in:

`/home/veritech/users/udekel/CDL2PN_1.1`

4. Code Documentation

4.1 Translation Process Overview

The final translation process will take a common file types for representing petri nets program, and translate it into a core program.

The final translation will be based on the following steps:

1. Reads a common file type for representing a petri nets program and creates the petri-net data structure out of it.
2. Translates the petri-net data structure into a core data structure.
3. Produces a core program from the core data structure.

In our project we completed the second part, and we will describe it here.

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.

In an object oriented petri net a connector can have a petri-net class as a source or target, and not only places or transitions.

Therefore a translation form an object oriented petri net will require special treats that are not needed for flat petri net.

The translation also does not support colored petri-net.

Our translation creates the core data structure from internal representation of the petri net.

The petri nets data structures we use are based on those used for the CDL2PN 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.

In our translation HOLD PREVIOUS is assumed for every translation.

4.2 Naming Conventions

In order to create names for the CDL elements based on the petri-net structure we needed to analyze the petri-net items names.

In case we handle a regular petri-net program the names remain the same.

It means that a petri place with the name X is translated into a CDL variable with the name X, and a petri transition with the name X is translated into a CDL transition with the name X.

Handling a petri-net program that was translated from a CDL program before is more complicated. In this case the petri-net items names have special meanings.

Therefore, when we handle a petri item we check its name format first in order to find out how to treat it.

In case we handle a petri-net program that was translated from a CDL program before, we take the following assumptions:

- Petri places which have names of the format $X=T$, $X=F$ and $X.initial$ (optional) are translated into one boolean CDL variable named X .
- Petri places which have names of the format $X=Y1$, $X=Y2$, ... (Y_i represents an integer value) are translated into one integer CDL variable named X . That includes PC variables.
- Petri place which has name of the format X^* is translated into a Boolean CDL variable named X^* (we keep the ‘*’ because it represents that this was an integer CDL variable before it was translated into an abstract petri place, and we want to distinguish it from a regular place).
- A petri transition, which has name of the format $X.Y$ is translated into a CDL transition named X (X is the CDL transition name, and Y is the index of the valuation for which this transition has been created).
- When we union a number of petri transitions ($X1.Y1$, $X2.Y2$, ...) into one CDL transition, we give the transition the name $X1X2X3...$
- We ignore transitions with the name format $X.initialize$. These transitions are created to represent a non-deterministic initialization of a variable, and therefore we don’t use them. If there is transition with this name format it means that there also exists a place with the name format $X.initial$, and we will use it to create a boolean variable which is not initialized.

4.3 Data Structures

Our data structures are based on the structures used for the CDL2PN translation, with few changes.

The description of the structures is written in the CDL2PN documentation.

We will describe here the changes we have made:

- To the **MPtransition** structure we added 3 fields: One is a flag that represents whether we already translated the transition into a CDL transition, and the second and third are dictionaries of MPplaces (that represent petri net places). One field contains places that there is a connector between them and the transition, and the other contains places that there is a connector between the transition and them.

4.4 Algorithms

4.4.1 Translation of OOP Petri-Net

As explained before, our system does not support object oriented petri-nets, and deals only with one class model.

Object oriented petri net requires special treats that are not needed for flat petri net. For example, a connector in an object oriented petri net can have a petri net class as source or target. Our program does not support a translation for this kind of connector.

Future extensions of the system will include translating an object oriented petri-net program.

In our project we gave efforts on building a program which will be adjusted easily to these extensions.

For this purpose our program goes over a list of MPclasses.

We assumed that each class represents one core module.

In this version we assume that there is only one petri-net class.

4.4.2 Translation of an individual class

Our system translates a single petri-net class into a single CDL module.

The translation begins by going over all the places declared in the class and translating them into CDL variables.

When all the variables are ready, the program starts iterating over each of the petri-net transitions, and translates them into CDL transitions using the petri-net connectors.

4.4.3 Translation of an individual place

There are many cases in which few petri places represent the same CDL variable.

This usually happens when the petri program we handle was translated first from CDL, and we want to translate it back to CDL.

Therefore when we run into a place, we need to find the related places, and created one CDL variable from them, in order to keep the program as closest as we can to the origin.

- If a place with name from the format $X=F$ exists, we can assume that a corresponding place with the name $X=T$ also exists. These two places represent one CDL variable. Therefore when we handle a place with name from the format $X=F$, we do not translate it, and wait until we will to the place with the name $X=T$, in order to perform this action then.

- When we handle a place with name from the format $X=T$ we check if a corresponding place with the name $X.initial$ also exists. If so, it means that there are three petri net places $X=F$, $X=T$ and $X.initial$ that represent the same CDL variable. Therefore in this case we do not translate the place $X=T$, and wait until we get to the place with the name $X.initial$, in order to perform this action then. If a place with the name $X.initial$ does not exist we create a boolean CDL variable. If the place $X=T$ contains an initial token we initialize the variable with the value 1, otherwise we initialize it with the value 0.
- When we handle a place with name from the format $X.initial$ we create a boolean not-initialized CDL variable.
- When we handle a place with name from the format $X=Y$ (Y represents an integer value) we first check if it contains any initial tokens. A place with this name format represents a specified value of an integer variable (can be also a PC variable). The initial value of this variable is depends on which one of the places representing this variable contains initial tokens (there can be only one place like this). Therefore if the place $X=Y$ contains initial tokens we create an integer CDL variable with initial value Y . Otherwise, we do nothing.
- When we handle a place with name from the format X^* (the $*$ represents that this place represents an abstract variable) we translate it into a boolean CDL variable. Its initial value depends on whether X^* contains initial tokens or not. If initial tokens exist, the initial value of the variable is 1, otherwise, 0.
- We handle all other variables exactly as we handle the abstract variable, which means that we create from each place a variable, initialized as described above.

4.4.4 Translation of transitions

There are cases in which we translate a set of petri transitions into a single CDL transition. A set of transitions contains transitions that share the same set of CDL variables produced for their input and output places. There is an example for a case like this in the **Appendixes** at the end of this document.

There is a special case in which an output place represents an integer variable. In this case we should make sure that all the transitions in the set contains the same output place, which means the same value of the corresponding CDL variable.

We use the connectors in order to be able to find out which transitions can be union in a set. We go threw each connector and check its origin and target.

If the origin is a transition, we add the target place to that transition as an output place. If the target is a transition, we add the origin place to that transition as an input place.

Once we checked all the connectors we start going threw all the transitions.

When we get to a transition that has not been handled yet we compare its set of input and output places with the other transitions that has not been handled.

We create a dictionary data structure from all the transitions that can be translated into one CDL transition.

In order to create the CDL transition we go over all the transitions in the set we created, and create the expressions for the enable and assign parts as described in the algorithm.

The enable part of the CDL transition consists of a number of disjuncts for each petri transition from the set. The process of creating the disjunct for each transition is described below:

- For input places with name from the format $X=T$ or $X=F$ we add X or $!X$ respectively.
- For input places with name from the format $X=Y$ (Y represents an integer value) we add to the enable part $X=Y$.
- When we handle an abstract or a regular place (the format of the name is X or X^*), which is an input place, we add X or X^* respectively to the disjunct.
- When we run into an abstract or a regular place while checking the output places while creating the assign part, we check if this place appears as an output place only. That means that it is not an input place as well. In this case we add $!X$ or $!X^*$ to the enable expression.

The assign part is built simultaneously with the enable part. For each transition we check the output places connected to it. When we start the process and get to the first transition in the set of transitions we initialize an assign for each output place that represents a CDL variable. We check the output places for each transition in the set, and update the assign part of the corresponding variable according to it. We create the assign part in a different way for every kind of variable.

- When we create an assign for an output place with name from the format $X=Y$ (Y represents an integer value) it means that we have to create an assign expression for the corresponding CDL integer variable (that includes PC variables). The expression we create for this assign is Y . We do not need to update this assign anymore (while iterating the other transitions in the set), since all the transitions in the set contain the same output place, $X=Y$, and not an output place that represents a different value. This is certain according to our rules for which petri transitions can be unionized to one CDL transition.
- For each boolean variable corresponding to an output place with name from the format $X=T$ or $X=F$, we add an assignment to the assign part of the CDL transition in the following way. The assign is built as a union of formulas. Each formula represents the value that the variable gets according to the petri transition we are handling. To build this formula we use the enable part that we created for this transition. If the enable condition takes place the value of the variable should be 0, if the output place is $X=F$, or 1 if it is $X=T$. A

formula that represents this condition is $a \rightarrow b$, while a represents the enable formula, and b represents the boolean value. The core parser does not allow us to create expressions using the \rightarrow operator, so we create an equal expression from the format $!aVb$.

- For a Boolean variable corresponding to an output place with name from the format X or X^* (abstract), we add the assign the expression 1 if X (or X^*) is only an output place. While checking the input places, if we run into an input place with name from this format, we add to the corresponding variable's assign the expression 0, if this is an input place only.

File listing

Below is the list of the files we are using. We added an explanation about the functionality of each file. We referred only to the header files, but most of them have associated implementation files.

In order to learn more about these files you can read the internal documentation.

Files Listing For the ModularPetri Directory

Filename	Purpose
ModularPetriToCore	This is the main file, which controls the translation process. It scans the classes, creates modules for them, creates variable for places and transitions for transitions.
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)

Other important files include:

Files Listing For the ModularPetri/DataSeture Directory

Filename	Purpose
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.

The file that was written for this translation only is ModularPetriToCore.

The other files were written by Uri Dekel and Nadav Golbandi for the CDL2PN translation, and we adjusted them for our purpose.

We added some functionalities for the MPtransition file that were described before.

5. Examples

In this section we will show some examples and results.

To demonstrate our project, we will take an original core program, translate it to PN, and translate the PN program that we get in result back to core.

In this way, we will be able to compare between the original core program, and the core program we get as result.

Since the translator is not completed yet, it does not produce the core program as an output.

In order to check the program we take the core data structures that we get in result, and translate it 'by hand' to a core program.

All the examples are of flat modules, which means one PN class.

All examples can be found under the **Examples** directory of the system.

Note: in the PN diagrams, the number inside each place represents the number of initial tokens it holds. The "1" on the connectors means that all the connectors are considered equal.

5.1 Examples with Booleans

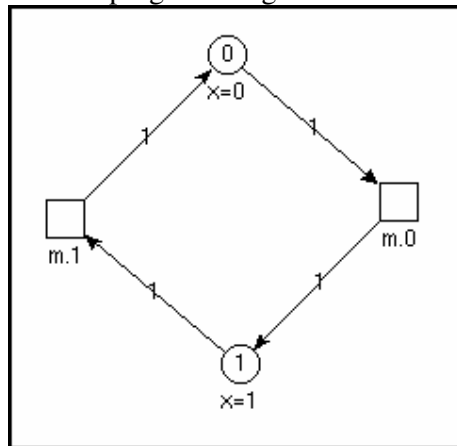
5.1.1 Single Variable and Transition Example

This is the simplest example. It contains single module, single variable and single transition. It can be found in the file **fige1_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:



After translating it back to core using our translation we get this target program. Note that the expressions for the enable and assign part 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);
}
```

5.1.2 Real World Example: Consumer-Producer

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

```

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

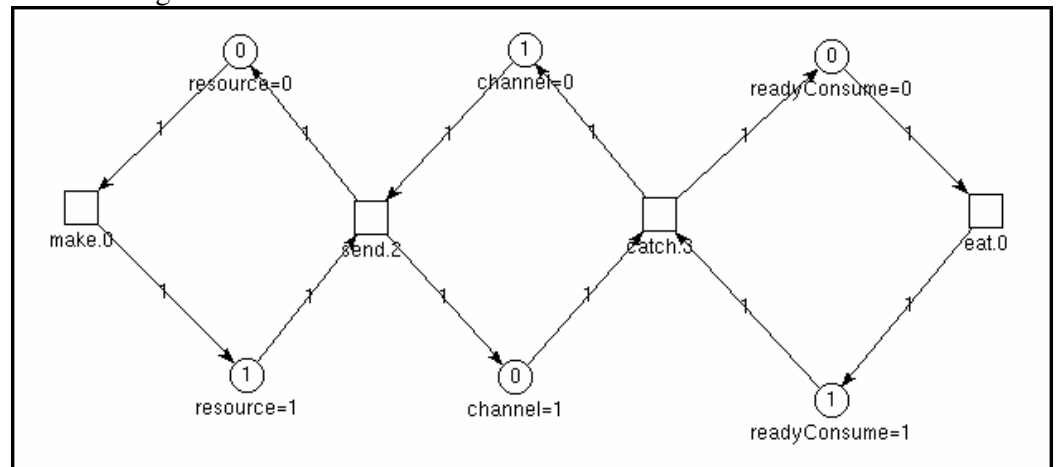
    TRANS send:
        enable: resource /\ !channel;
        assign: resource' := false;
                channel' := true;

    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:



After translating it back to core using our translation we get this target program.

```
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' := false /\ !(readyConsume
                /\ channel);

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

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

5.1.3 Nondeterministic Initialization of Boolean Example

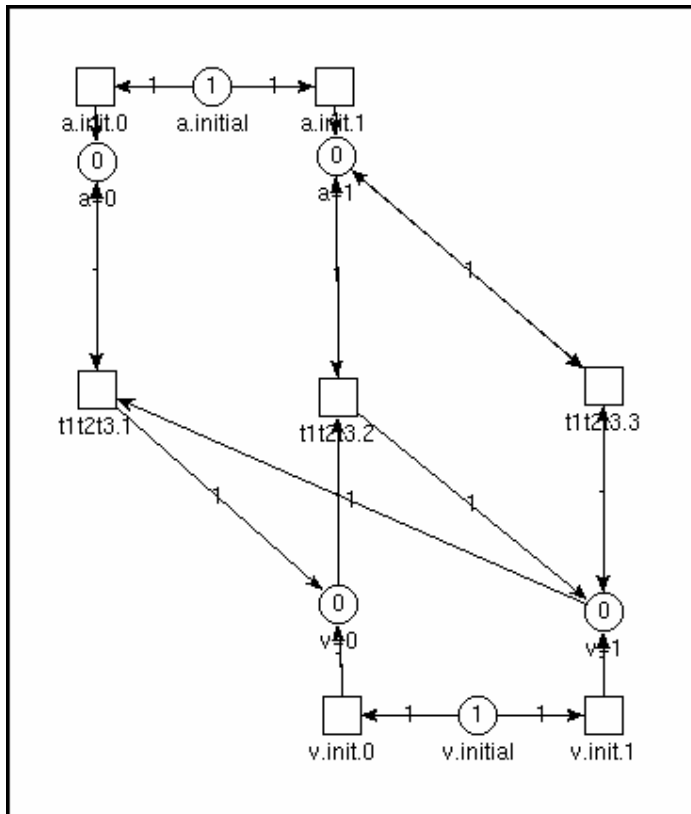
In this example all the Boolean variables have a non-deterministic initialization. Another thing we can see here is how a complex CDL transition is turned into many petri transition and back to one complex CDL transition. The code can be found in **fig2_1.cdl** file.

```

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:



After translating it back to core using our translation we get this target program. 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));
           v' := (false \/ !(v /\ !a)) /\
                (true \/ !(v /\ a));

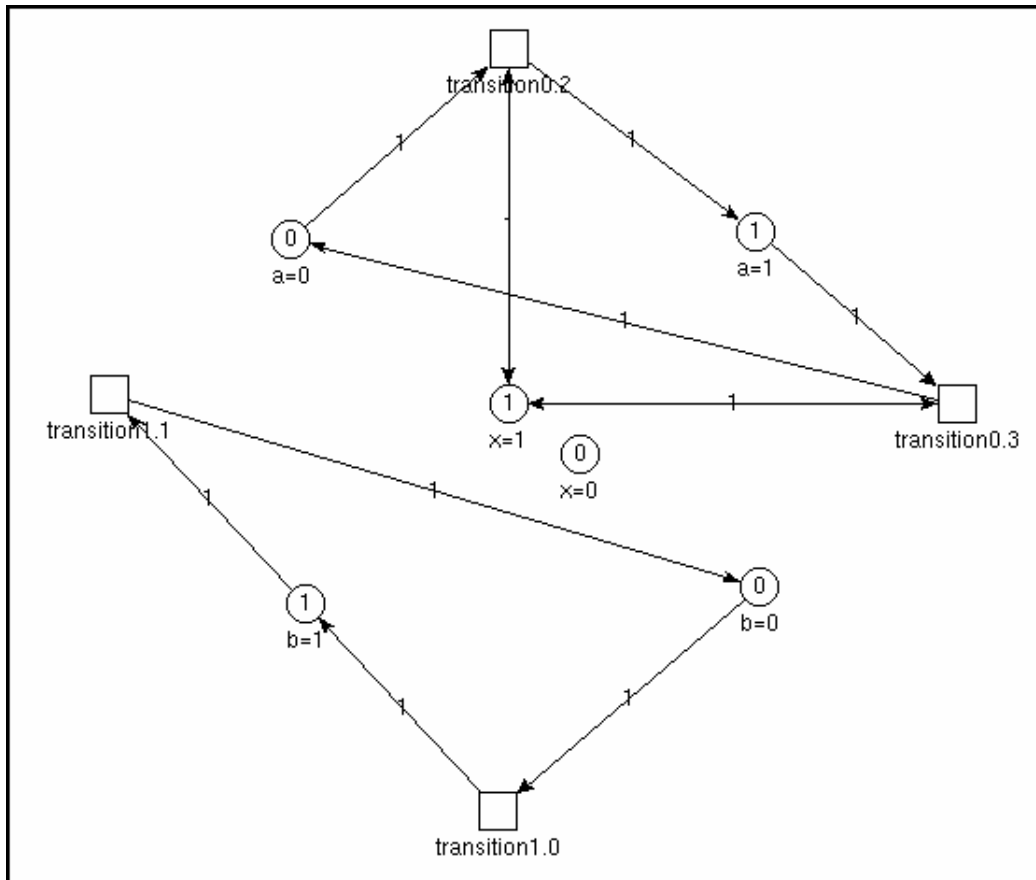
}
```

5.1.4 Another example with Boolean variables

This code for the first example can be found in **ex12.cdl** file.

```
HOLD_PREVIOUS
MODULE SYSTEM ()
{
  VAR
    x: boolean INITVAL true;
    a: boolean INITVAL true;
    b: boolean INITVAL true;
  TRANS transition0:
    enable: x;
    assign: a' := !a;
  TRANS transition1:
    enable: true;
    assign: b' := !b;
}
```

The petri result is:



After translating it back to core using our translation we get this target program.

```

HOLD_PREVIOUS
MODULE SYSTEM ()
{
  VAR
    b: boolean INITVAL true;
    a: boolean INITVAL true;
    x: boolean INITVAL true;
  TRANS transition1:
    enable: b \/ !b;
    assign: b' := (true \/ !!b) /\ (false \/ !b);
  TRANS transition0:
    enable: (a /\ x) \/ (!a /\ x);
    assign: x' := (true \/ !(a /\ x)) /\
                  (true \/ !(a /\ x));
              a' := (true \/ !(a /\ x)) /\
                  (false \/ !(a /\ x));
}

```

5.2 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.

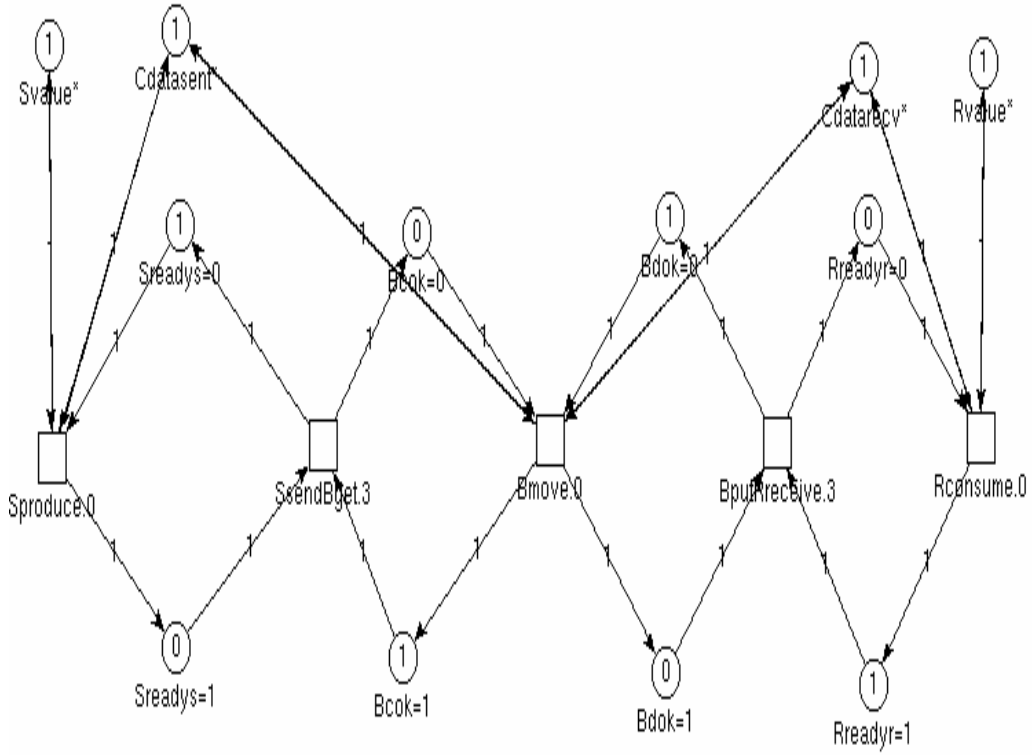
```

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 our translation we get this target program. 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);

}

```

5.3 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.

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

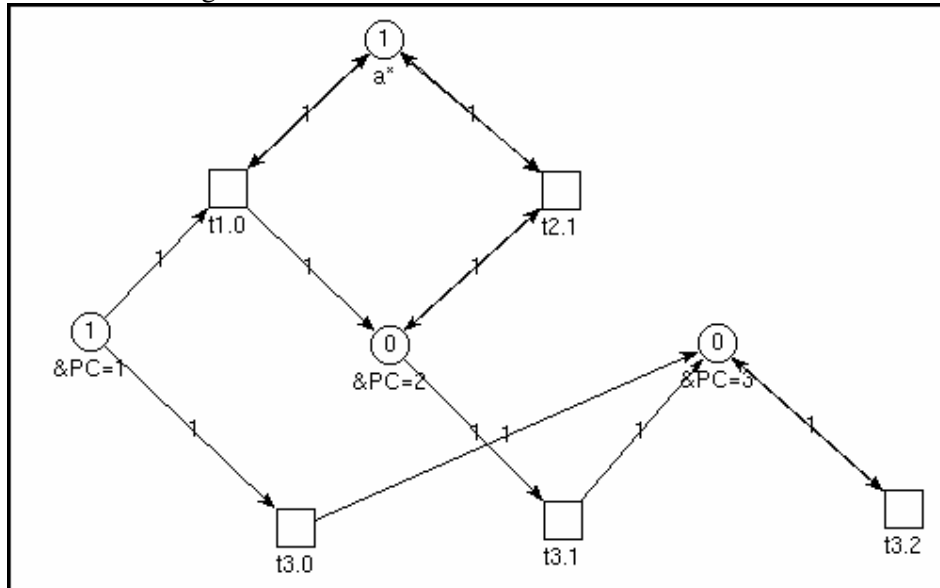
TRANS t1:
  enable: &PC=1;
  assign: a' :=2;
         &PC' :=2;

TRANS t2:
  enable: &PC=2;
  assign: a' :=1;

TRANS t3:
  enable: true;
  assign: &PC' :=3;

}
```

And the resulting net is:



PN2CDL Project Documentation

After translating it back to core using our translation we get this target program.

```
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;

}
```

Appendixes

Creating a Core Transition

In order to understand the code well, you can use the inner documentation.

The way of translating one or more petri transitions into one core transition (and especially creating the assign and enable parts) is the most complicated part of the code. The function 'AnalyzeTransition' is very long and hard to understand.

To make it easier, you can use the pseudo code below. It contains the pattern of this function and explains the reasons for its main loops.

AnalyzeTransition:

Call the function which unions one or more petri transitions that can be translated into a single core transition according to the algorithm.

Go over this set of transition and allocate an array with the proper size to the name of the core transition we create.

Go over all the transitions in the list

```
{
  Go over all input places of the transition to create the enable part
  {
    Create core expression according to the place name.

    Update the expression for the enable condition for this petri transition
    according to this place.

    If this is the first transition in the set:
    {
      If the input place represents a regular or an abstract variable
      and does not appear as an output place also:
      {
        Create assign for the variable representing this place
        with expression false.
      }

      Go over all output places of the transition to create assigns
      {
        Create assign with a proper name.

        If the place represents an Integer, create expression for
        this assign with the value this place represents.

        If the place has a name of the form X=T or X=F
        initialize the assign expression with NULL.
      }
    }
  }
}
```

If the place represents an abstract or a regular variable:

```
{
    If the place does not appear as an input place as
    well
    {
        Add the expression !X to the enable
        expression for this petri transition.
    }

    Initialize the assign with value true.
}
```

```
}
```

If this is the last input place for this transition (which means that this is the last iteration for this petri transition):

```
{
    Go over all the assigns we created
    {
        If the variable represents a petri output place with
        name of the form X=T or X=F:
        {
            Update the assign expression by adding (with  $\wedge$ )
            an expression  $(b \vee !a)$ .
            b - represents T or F.
            a - represents the enable expression for this
            petri transition.
        }
    }
}
```

Update the enable expression for the core transition by adding (with \vee) the enable expression we created for this petri transition.

Update the name of the core transition with the name of the petri transition.

```
}
```

Add the transition we created to the transitions of the core module.

Future Improvements

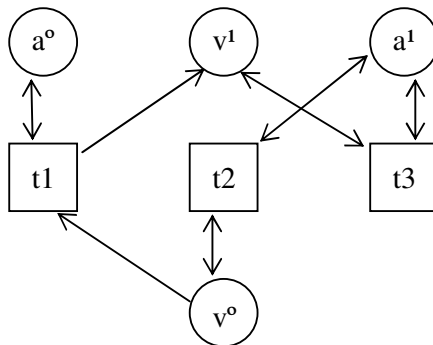
Most important tasks:

- A program that translates common file types for representing petri nets into the petri nets data structures has to be written.
- A program that translates data from core data structure into core program in XML has to be written.

Recommended tasks:

- The back translation (CDL2PN) uses **a table** that holds relevant data for translating core transition into petri net transitions. Maybe this table can be used in order to restore relevant data for translating the petri net transitions back to a core transition.
- The assign and enable expressions for core transitions may be very long and complicated. A program which simplifies them can be very helpful.

Example for Translating a Set of PN transitions into a Single CDL Transition



Sets of input places for the transitions above:

• $t1 = \{a^{\circ}, v^1\}$, • $t2 = \{a^1, v^{\circ}\}$, • $t3 = \{a^1, v^1\}$

These three sets of places correspond to the set of CDL variables $\{a, v\}$.

Sets of output places:

$t1 \bullet = \{a^{\circ}, v^{\circ}\}$, • $t2 = \{a^1, v^{\circ}\}$, • $t3 = \{a^1, v^1\}$

These three sets of places correspond to the set of CDL variables $\{a, v\}$ as well.

Since $t1, t2, t3$ share the same sets of input and output places they can be translated into one CDL transition named $t1t2t3$.

The enable condition for $t1$ is $(!a \wedge !v)$, for $t2$ is $(a \wedge !v)$ and for $t3$ is $(a \wedge v)$.

Therefore the enable condition for the generalized transition $t1t2t3$ is:

$(!a \wedge !v) \vee (a \wedge !v) \vee (a \wedge v)$. This expression can be simplified to the equal expression $(a \vee !v)$ if an appropriate program will be written (as mentioned in [Future Improvements](#)).

The assign expression for the variable v will be:

PN2CDL Project Documentation

$v' := (\text{true} \vee \neg(a \wedge \neg v)) \wedge (\text{false} \vee \neg(a \wedge \neg v)) \wedge (\text{true} \vee \neg(a \wedge v))$

This expression can be simplified to $(\neg a \vee v)$.

The assign expression for the variable a will be:

$a' := (\text{false} \vee \neg(\neg a \wedge \neg v)) \wedge (\text{true} \vee \neg(a \wedge \neg v)) \wedge (\text{true} \vee \neg(a \vee v))$

This expression can be simplified to (a) .