INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2013, VOL. 59, NO. 4, PP. 317–324
Manuscript received October 28, 2013; revised November, 2013. DOI: 10.2478/eletel-2013-0038

Implementation of Algorithm of Petri Nets Distributed Synthesis into FPGA

Arkadiusz Bukowiec, Jacek Tkacz, Tomasz Gratkowski, and Tomasz Gidlewicz

***Abstract*—In the paper an implementation of algorithm of Petri net array-based synthesis is presented. The method is based on decomposition of colored interpreted macro Petri net into subnets. The structured encoding of places in subnets is done of using minimal numbers of bits. Microoperations, which are assigned to places, are written into distributed and flexible memories. It leads to realization of a logic circuit in a two-level concurrent structure, where the combinational circuit of the first level is responsible for firing transitions, and the second level memories are used for generation of microoperations. This algorithm is implemented in C# and delivered as a stand alone library.**

***Keywords*—C#, decomposition, FGPA, logic synthesis, Petri Net**

I. Introduction

APPLICATION specific logic controllers (ASLCs) [1]–[3] are one of the biggest and most popular group of electronic devices. They can be designed as dedicated software for microprocessor or as dedicated hardware. The second approach gives more possibilities of system integration as system on programmable chip (SoPC) with use of field programmable gate arrays (FPGAs). The most classical way of designing such controllers is application of hardware description languages (HDLs) but it is uncomfortable for designer and potentially it gives high risk of human mistake. The usage of graphical representation of algorithm is much more conformable [4]–[7]. In this case Petri nets (PNs) [8], [9] are one of the most adequate methods for formal design of ASLCs [10]. It gives easy way for representation of concurrent processes and additionally there could be applied mathematical algorithms for formal analysis and verification of the designed model [11]–[17]. There are also several algorithms of direct synthesis of Petri net model into FPGA devices [18]–[21]. The most typical implementation of Petri nets into FPGA devices use one-hot local state encoding where each single place is represented by a flip-flop [22]. Such an approach requires hardware implementation of a large number of several logic functions and flip-flops included in macrocells.

One of the main features of FPGA is an existence of separated logic elements (look-up tables) with restricted fixed number of inputs. Very frequently logic functions have more arguments than number of inputs of such logic element. It forces a functional decomposition during a synthesis process and consumes a large number of logic elements. One of the methods of decreasing a number of such functions is architectural decomposition of a sequential circuit [23], [24]. Such methods introduce several additional internal variables and very often consume more hardware than typical direct implementation. This issue can be resolved by using logic elements together with embedded memory blocks [25] that are available in modern FPGA devices.

There is proposed the fully automated implementation of the method of synthesis [26], [27] that allows to decrease the number of implemented logic functions depending on inputs and internal variables of Petri net-based ASLC in the paper. The algorithm was implemented with use of C# language and compiled as a stand alone library. The entry to the algorithm is object oriented model of colored Petri net and the output is a set of HDL files that consist the logic description of ASLC. The synthesis process is fully automated, it means that it does not need any interaction with the designer. To permit the minimal local state encoding the Petri net has to be initially colored [8], [28]. The Petri net is also compacted into macro Petri net [29] to shorten time of synthesis [27]. During the synthesis, the logic functions are classified into two sets. The first set contains functions responsible for describing preconditions and guards of transitions. This set is going to be synthesized as logic functions and then it can be implemented with use of logic elements. The second set contains functions responsible for generation of microoperations and it is going to be synthesized as memory block and then it can be implemented with use of the embedded memory blocks. Macroplaces that are colored by the same color create one state machine module. Consequently, places, represented by these macroplaces, could be encoded by a minimal-length binary vector. This encoding also allows a reasonable decomposition of a microoperation decoder into several concurrently working distributed memories. Each memory block controls only microoperations that belong to the subnet with the same color. In such a way it leads to balanced usage of all kinds of logic resources of the FPGA device. Very frequently such a method gives also an effective utilization of all FPGA resources by a whole digital system.

II. Control Interpreted Macro-Petri Net

Definition 1. *A simple Petri net [8], [9] is a triple*

$$PN = (P, T, F), \tag{1}$$

where:

P	*is a finite, non-empty set of places,* $P = \{p_1, \ldots, p_M\}$
T	*is a finite, non-empty set of transitions,* $T = \{t_1, \ldots, t_S\}$

The research was financed from budget resources intended for science in 2010–2013 as an own research project No. N N516 513939.

A. Bukowiec, J.Tkacz, T. Gratkowski, and T. Gidlewicz are with the Institute of Computer Engineering and Electronics, University of Zielona Góra, Licealna 9, 65-417 Zielona Góra, Poland (e-mails: {a.bukowiec;j.tkacz;t.gratkowski}@iie.uz.zgora.pl; tgidlewicz@gmail.com).

F *is a set of flow relations called arcs from places to transitions and from transitions to places:*

$$F \subseteq (P \times T) \cup (T \times P),$$
$$P \cap T = \varnothing.$$

Definition 2. *Sets of input and output transitions of a place $p_m \in P$ are respectively as follows:*

$$\bullet p_m = \{t_s \in T : (t_s, p_m) \in F\},$$
$$p_m \bullet = \{t_s \in T : (p_m, t_s) \in F\}.$$

Definition 3. *Sets of input and output places of a transition $t_s \in T$ are respectively as follows:*

$$\bullet t_s = \{p_m \in P : (p_m, t_s) \in F\},$$
$$t_s \bullet = \{p_m \in P : (t_s, p_m) \in F\}.$$

Definition 4. *A marking of a Petri net is a function:*

$$M : P \to \mathrm{N}.$$

It describes a number of tokens $M(p_m)$ situated in a place p_m. When a place or a set of places contains a token it is marked. A transition t_s can be fired if all its input places are marked. Firing of a a transition removes tokens from its input places and puts one token in each output place. There can be specified the initial marking M_0, then

Definition 5. *the initially marked Petri net is a 4-tupe:*

$$PN = (P, T, F, M_0). \tag{2}$$

A. Colored Petri Net

A Petri net can be enhanced by assigning colors to places and transitions [8], [28].

Definition 6. *A state machine subnet (SM-subnet) [8] of a Petri net PN is as a strongly connected subnet PN_i generated by places in PN, such that all input and output transitions of places in PN_i and their connecting arcs belong to PN_i and each transition of a subnet PN_i has exactly one input and one output arc.*

In colored Petri net colors help to validate intuitively and formally the consistency of all sequential processes covering the considered Petri net. Each color recognizes one SM-subnet.

The rules for Petri net coloring are as follows [18]:

- each place and transition must have at least one color,
- if the place has a color each of its input and output transition must have the same color,
- input places of each transition must hold different colors,
- output places of each transition must hold different colors,
- input and output places of transition must share the same set of colors,
- initially marked places can not share exactly the same set of colors,
- the number of different colors which are shared by the initially marked places is equal to the total number of colors.

B. Interpreted Petri Net

An interpreted Petri net is a Petri net enhanced with an additional feature for information exchange [9]. Such a Petri net is called interpreted Petri net or a colored interpreted Petri net if both enhancements are applied. This exchange is made by use of binary signals. Interpreted Petri nets are used as models of concurrent logic controllers.

The Boolean variables occurring in the interpreted Petri net can be divided into three sets:

X	is a set of input variables, $X = \{x_1, \ldots, x_L\}$,
Y	is a set of output variables, $Y = \{y_1, \ldots, y_N\}$,
Z	is a set of internal communication variables, typically it is not used and $Z = \varnothing$.

The interpreted Petri net has a guard condition φ_s associated with every transition t_s. The condition φ_s is defined as the Boolean function of some variables form sets X and Z. In the particular case the condition φ_s can be defined as 1 (always true). Now, transition t_s can be fired if all its input places are marked and current value of corresponding Boolean function φ_s is equal to 1. Conjunction ψ_m is associated with place p_m. ψ_m is an elementary conjunction of affirmation of some output variables form the set Y. If the place p_m is marked the output variables from corresponding conjunction ψ_m are set and other variables are reset.

C. Macro Petri Net

Macro Petri net is a Petri net where part of the net (subnet) is replaced by one macroplace [9]. It allows to enhance Petri nets with hierarchy [12] and it simplifies algorithms of coloring, verification and synthesis of Petri net. There are many classes of subnets that could be replaced by macroplace, for e.g.:

- SM-subnets [8],
- Two-pole blocks [30],
- Parallel places [8],
- P-blocks [9].

These classes create to many possibilities of merging Petri net into macro Petri net. For the synthesis purpose, the best solution is application of mono-active macroplaces [30]. These are macroplaces that have one input and one output and consist of only sequential places. Only macro Petri nets with such macroplaces will be used in this article.

III. Synthesis Method Overview

The synthesis method [27] is based on the minimal local states encoding of places together with functional parallel decomposition of the Petri net-based logic circuit. Places are encoded separately in every colored subset. Output variables (names of particular microoperations) assigned to places are placed in configured memories of FPGA. It leads to realization of a logic circuit in two-level structure (Fig. 1), where the combinational circuits (CC^i) of first level are responsible for generation of the excitation functions:

$$D^i = D^i(X, Q), \tag{3}$$

where $Q = Q^1 \cup Q^2 \cup \cdots \cup Q^I$ is the set of variables used to store the codes of currently marked places. The memory of the circuit is built from I concurrent colored D-type registers RG^i

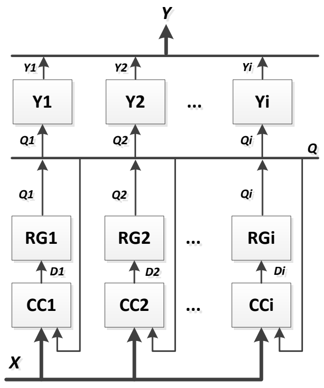


Fig. 1. Logic circuit of Petri net.

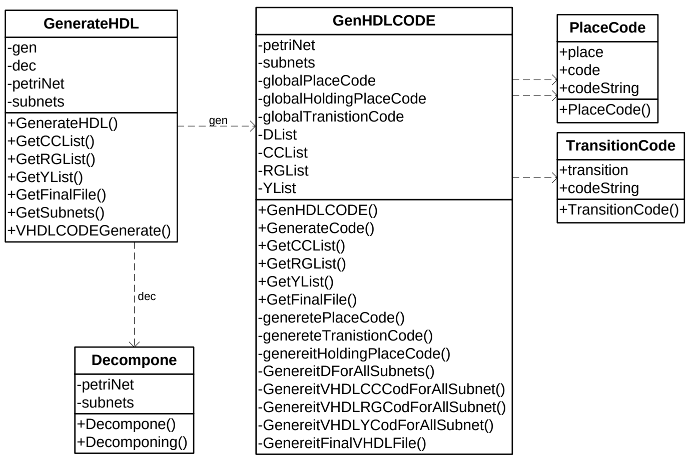


Fig. 2. Class diagram of DecomponeAndVHDLCodeGen library.

which hold a current state of each subnet. Here, $i = 1, 2, \ldots, I$ is a number of color and SM-subnet in Petri net colored by I colors. The second level decoders Y^i are responsible for generation of microoperations and they are implemented using memory blocks. Their functionality can be described by function:

$$Y^i = Y^i(Q^i). \tag{4}$$

Such approach allows to use logic elements and embedded memory blocks available in modern FPGA devices in a balanced way.

The entry point to the synthesis method is the colored interpreted macro Petri net. There are many algorithms of SM-coloring Petri nets, for example, the one described in: [31], [32]. The outline of synthesis process [27] includes following steps:

1) Formation of subnets
2) Encoding of places
3) Formation of conjunctions
4) Formation of logic equations
5) Formation of memory contents
6) Formation of logic circuit

IV. Implementation of Synthesis Method

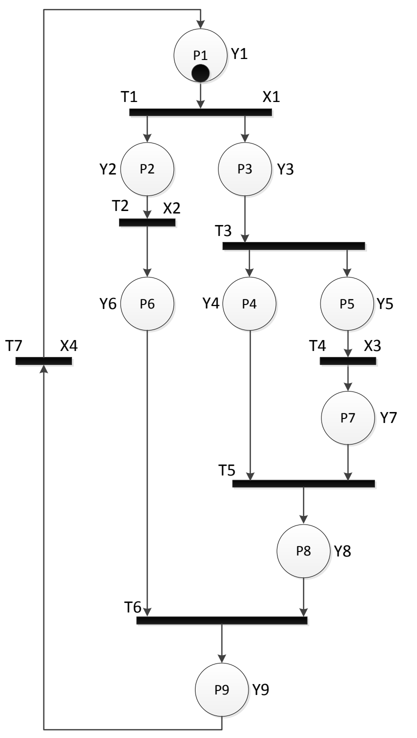
The algorithm [27] was implemented in C# in Microsoft .NET environment. It was compiled into *DecomponeAnd-VHDLCodeGen.dll* library. The whole process is fully automated and does not require any interaction with user. There is one public method for generation VHDL code. Other methods, which perform particular steps of synthesis algorithm, are internal and not available for end-user. They are invoked automatically by main method.

The whole algorithm was implemented with use of three classes (Fig. 2). The main class is *GenerateHDL*. The entry point is object of colored Petri net passed to the constructor of *GenerateHDL* class. The whole synthesis process is run by public method *VHDLCODEGenerate()*. This process is divided into two parts: decomposition of petri net (step 1.) – implemented in the *Decompone* class, and generation of VHDL description (step 6.) – implemented in the *GenHDL-CODE* class. The second part includes encoding (step 2.) and formation of conjunctions (step 3.), logic equations (step 4.), and memory contents steps (step 5.).

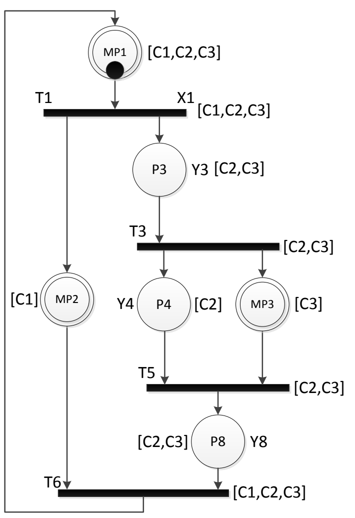
To make the description more clear it is illustrated by application for Petri net PN_1 (Fig. 3a). Of course, initially the Petri net PN_1 was colored and collapsed into colored macro-Petri net (Fig. 3b).

A. Object-Oriented Model of Petri Net

To store the model of Petri net in the computer memory there was designed the object-oriented model (Fig. 4). This model allows to store hierarchical and colored nets. The main class *PetriNet* is responsible for storing the whole model and it correspond to Petri net definition (2). The classes *Place* and *Transition* represent single place and transition of Petri net and their collections *places* and *transitions* in the main class represent all places and transitions of Petri net and these collections correspond to the sets P and T from (2). The class *Arc* describes flow relation between place and transition or transition and place. Its collection *arcs* in the main class describes all relations in Petri net and it corresponds to the set F from (2). There is no special collection to define initial marking M_0 of the Petri net but it can be defined by setting an adequate value of the property *token* of initially marked places. The class *Color* is responsible to store colors of Petri net. The classes *Place*, *Transition*, *Arc* have a collection *colors* to store all colors assigned to this object. The classical interpretation of Petri net ia defined in the classes *Place* and *Transition.* The property *outputs* in the class *Place* defines coma separated list of active outputs and it corresponds to the conjunction ψ_m. The property *condition* in the class *Transition* defines the logic formula of guard condition and it corresponds to the Boolean function φ_s. Proposed model has an extension of classical interpretation of Petri net by additional adding of the condition to the flow relation. It is defined by the property *condition* in the class *Arc.* If it is used in Petri net model then the full guard condition (the property *fullCondition*) of transition is formed as a conjunction of condition of transition and all conditions of its input arcs. The property *isMacroplace* is defined when considered place is a macroplace. Then the collection *placesInMacroPlace* consists the list of all places belonging to this macroplace. Additionally, each place belonged to the macroplace has set the property *parentMacroplace* to this macroplace. Such definition of the



(a) Interpreted Petri net PN_1



(b) Colored Macro-Petri net PN_1

Fig. 3. Example of Petri net PN_1.

hierarchy in the proposed object-oriented model of Petri net allows to construct multi-level hierarchy in Petri net model.

There are also defined some additional redundant properties (like *incidece* or *inputPlaces*) in this model that are used to speed-up some analytical algorithms that operate on this model (like coloring, decomposition, etc.). There are also defined other properties (like *xCord* or *labelPosition*) that are used to store information required for graphical representation of the Petri net.

B. Decomposition of Petri Net

The main synthesis process starts from decomposition of Petri net into subnets. It is done by the *Decomponing()* method

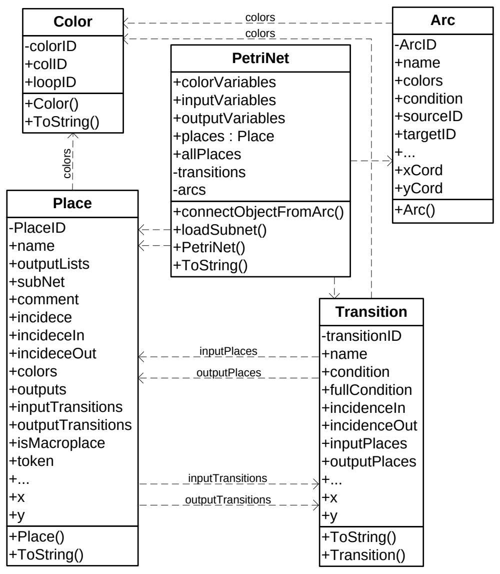


Fig. 4. Class diagram of Petri net model.

from the the *Decompone* class invoked on *dec* property. The Petri net has to be passed into constructor *Decompone(PetriNet net)* of this property. Each subnet corresponds to one color of Petri net. It has to include all transitions that are colored by this color (Lst. 1).

```
foreach (Transition t in petriNet.
    transitions)
{
   copy = false;
   foreach (Color c in t.colors)
       if (c.colorID == i)
          copy = true;
   if (copy)
      subnet.transitions.Add(t);
}
```

Lst. 1. Part of C# code to copy subnet transitions

It also has to include all places colored only by this color. If any place is colored by this and others colors it is assigned only to one subnet. In other subnets it is replaced by empty doubler macroplace (with prefix `DMP`) (Lst. 2). If there is a chain of such empty doubler macroplaces it could be replaced by one empty doubler macroplace in the further optimization of this algorithm. At this stage the macroplaces are treated the same as normal place.

```
foreach (Place p in petriNet.places)
{
   pColor = colorsNo;
   addPlace = false;
   if (!p.replacedByMacroplace)
   {
      foreach (Color d in p.colors)
      {
         if (pColor > d.colorID) pColor = d
             .colorID;
         if (d.colorID == i) addPlace =
             true;
      }
      if (pColor == i && addPlace)
      {
         Place n = p;
```

```
            tempNet.places.Add(n);
        }
        else if (pColor < i && addPlace)
        {
            macro_place++;
            Place dmp = new Place("DMP" + macro_place.ToString());
            dmp.colors = p.colors;
            dmp.isMacroplace = true;
            dmp.inputTransitions = p.inputTransitions;
            dmp.outputTransitions = p.outputTransitions;
            dmp.placesInMacroPlace.Add(p);
            tempNet.places.Add(dmp);
        }
    }
}
```

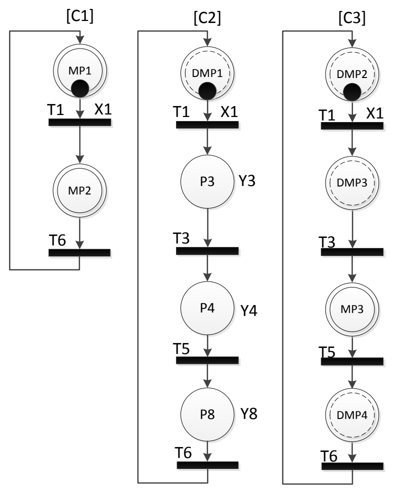
Lst. 2. Part of C# code to copy subnet places

The example of extracted subnets for the Petri net PN_1 are shown in the Fig. 5a. The next step of decomposition is to expand macroplaces to receive flat subnets. Now, each macroplace can occur only in one subnet, because other occurrences were replaced by empty doubler macroplaces. It means, that each macroplace has to be expanded (Lst. 3).

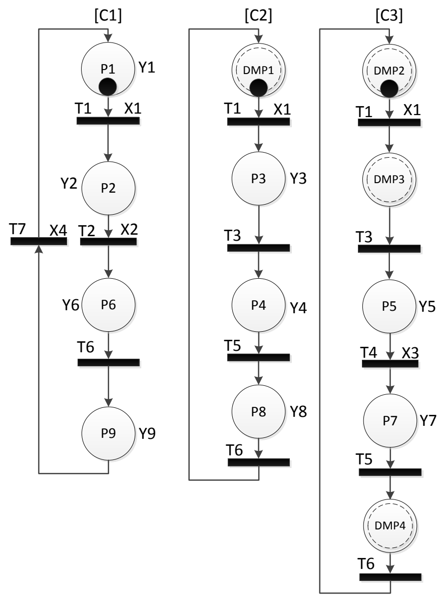
```
foreach (Place mp in tempNet.places)
{
    if (mp.isMacroplace)
    {
        foreach (Place p in mp.placesInMacroPlace)
            if (p.replacedByMacroplace)
                tempSubnet.places.Add(p);
    }
}
bool result = false;
foreach (Place p in tempSubnet.places)
{
    result = false;
    if (p.inputTransitions[0].colors.Count() < 2)
    {
        foreach (Transition t in tempNet.transitions)
        {
            if (!t.Equals(p.inputTransitions[0])) result = false;
            else
            {
                result = true;
                break;
            }
        }
        if (!result) tempSubnet.transitions.Add(p.inputTransitions[0]);
    }
}
foreach (Place p in tempSubnet.places)
    tempNet.places.Add(p);
foreach (Transition t in tempSubnet.transitions)
    tempNet.transitions.Add(t);
```

Lst. 3. Part of C# code to expand macroplaces

The example of subnets with expanded macroplaces for the Petri net PN_1 are shown in the Fig. 5b.



(a) Subnets with macroplaces



(b) Subnets with expanded macroplaces

Fig. 5. Extracted subnets of Petri net PN_1.

C. Formation of Description of Petri Net

The other steps of synthesis algorithm are performed by methods from the *GenHDLCODE* class. All equations are generated directly into VHDL syntax. It allows easy and fast further generation of description of hole circuit in VHDL. To start these steps the *GenerateCode()* method on the *gen* property have to be invoked. The petri net and subnets are passed into constructor *GenHDLCODE(PetriNet net, List⟨PetriNet⟩ subnets)* of this property. The *GenerateCode()* method runs in sequence all steps: *generetePlaceCode(), genereteTranistionCode(), genereitHoldingPlaceCode(), GenereitDForAllSubnets(), GenereitVHDLCCCodForAllSubnet(), GenereitVHDL-RGCodForAllSubnet(), GenereitVHDLYCodForAllSubnet(), GenereitFinalVHDLFile().*

First all places are encoded by the *generetePlaceCode()* method (Lst. 4).

```
1  int startR = 0;
2  int subnetID = 0;
3  foreach (PetriNet p in subnets)
4  {
5     Place startPlace = FindStartPlace(p);
6  
7     int itemIndex = 0;
8     List<byte> code = Int2Vector(itemIndex);
9     string codeString = genStringCodePlace(
          startR, code, startR+Ri[subnetID]);
10    globalPlaceCode.Add(new PlaceCode(
          start_place, code, codeString));
11    List<Place> nextPlaces = NextPlace(
          start_place, p);
12    itemIndex++;
13    Place lastPlace = startPlace;
14    while ((startPlace.placeID != nextPlaces
          [0].placeID))
15    {
16       if (nextPlaces.Count == 1)
17       {
18          code = Int2Vector(itemIndex);
19          codeString = genStringCodePlace(
                startR, code, startR + Ri[
                subnetID]);
20          globalPlaceCode.Add(new PlaceCode(
                nextPlaces[0], code, codeString
                ));
21          lastPlace = nextPlaces[0];
22          nextPlaces = NextPlace(nextPlaces
                [0], p);
23          if (nextPlaces.Count() < 1) break;
24          itemIndex++;
25       }
26       else
27       {
28          List<Place> t;
29          foreach (Place pr in nextPlaces)
30          {
31             t = LoopCounter(p, pr,
                   lastPlace, startPlace);
32             if (t.Count > 0)
33             {
34                foreach (Place pp in t)
35                {
36                   code = Int2Vector(
                         itemIndex);
37                   codeString =
                         genStringCodePlace(
                         startR, code, startR
                         + Ri[subnetID]);
38                   globalPlaceCode.Add(new
                         PlaceCode(nextPlaces
                         [0], code, codeString
                         ));
39                   itemIndex++;
40                }
41             }
42             else
43             {
44                   nextPlaces = new List<
                         Place>();
45                   nextPlaces.Add(pr);
46             }
47          }
48       }
49    }
50    startR += Ri[subnetID];
51    subnetID++;
52 }
```

Lst. 4. Part of C# code to encode places

Each subnet is encoded separately. The encoding is done on minimal number of required bits. The initial place of each subnet receives the code with value equal to 0 (lines 5 to 11 of Lst. 4) and reaming places receive following codes. It is required to use

$$R_i = \lceil \log_2 |P_i| \rceil \tag{5}$$

bits to encode places of each subnet, where $P_i \subseteq P \cup MP_i$ is a set of places in a subnet that was created based on the i-th color. MP_i is the set of doubles added to this subnet. The code of place is stored as an object of *PlaceCode* class. It is stored both as list of bits and string representation in VHDL syntax of conjunction of affirmation or negation of logic variables. This string also represent a place conjunction that is required to form logic equations. For example, for the p_1 place it is denoted as:

```
not Q(0) and not Q(1);
```

Then, the guard condition of transitions are generated by the *genereteTranistionCode()* method (Lst. 5).

```
string guard;
foreach (Transition t in petriNet.
      transitions)
{
   foreach (Place p in t.inputPlaces)
   {
      foreach (PetriNet subnet in subnets)
      {
         List<Place> DmpList = GetDMPList(
               subnet);
         if (PlaceIsInDMP(p, DmpList) &&
               tranisitionInSubnet(t, subnet))
            guard += ToVhdlCode(
                  FindDmpWithPlace(place,
                  DmpList).placeID);
      }
      foreach (PetriNet subnet in subnets)
      {
         List<Place> DmpList = GetDMPList(
               subnet);
         if (PlaceInSubnet(p, subnet) && !
               PlaceIsInDMP(p, DmpList))
            guard +=  ToVhdlCode(place.
                  placeID) ;
      }
   }
   guard += AddVhdlConditionCode(t);
   guard += ";";
   globalTranistionCode.Add(new
         TransitionCode(t, guard));
}
```

Lst. 5. Part of C# code to generate guard condition

Each guard of the transition is created as a conjunction. This conjunction itself consists of conjunctions of codes of transition input places and of a transition condition:

$$t_s = \bigwedge \bullet t_s \wedge \varphi_s. \tag{6}$$

It is stored as string representation formatted with VHDL syntax. For example, for the t_1 transition it is denoted as:

```
p(1) and DMP(1) and DMP(2) and X1;
```

Next, place hold-conjunctions are created by the *genereitHoldingPlaceCode()* method (Lst. 6).

```
foreach (PlaceCode pc in globalPlaceCode)
{
   foreach(byte b in pc.code)
   {
      if (b > 0)
      {
         foreach (Transition t in pc.place.
              outputTransitions)
            hpc += "(not␣" + ToVhdlCode(t.
                transitionID) + ")␣and";
         hpc += "␣" + ToVhdlCode(pl.place.
              placeID) + ";";
         globalHoldingPlaceCode.Add(new
              PlaceCode(new Place("H"+pl.place
              .placeID), new List<byte>(),hpc)
              );
      }
   }
}
```

Lst. 6. Part of C# code to generate place hold-conjunctions

This conjunction itself consists of conjunction of negation of all conjunctions of output transitions and of a place conjunction:

$$hp_m = \bigwedge \overline{p_m \bullet} \wedge p_m. \tag{7}$$

It is also stored as string representation formatted with VHDL syntax. For example, for the p_1 place it is denoted as:

```
(not t(1)) and p(1);
```

Now, the equations can be generated. The *GenereitDForAllSubnets()* method (Lst. 7) generates input equations for D flip-flops.

```
foreach (PetriNet p in subnets)
{
   bool first = true;
   string result = String.Empty;
   for (int i = 0; i < Ri[pi]; i++)
   {
      first = true;
      result += "D(" + i + ")␣<=";
      foreach (PlaceCode pc in
           globalPlaceCode)
      {
         if (PlaceInSubnet(pc.place, subnet
             ) || DMPPlaceInSubnet(pc.place,
             subnet))
         {
            if (pc.code.Count() > i)
            {
               if (pc.code[i] > 0)
               {
                  if (!first) result += "␣
                      or␣";
                  foreach (Transition t in
                       pc.place.
                       inputTransitions)
                     result += ToVhdlCode(t
                         .transitionID) + "␣
                         or␣";
                  result += ToVhdlCode("H"
                      + pc.place.placeID);
                  first = false;
               }
            }

         }
      }
      result += ";\r\n";
   }
   DList.Add(result);
   pi++;
}
```

Lst. 7. Part of C# code to generate D equations

They are built from conjunctions describing guard condition of transitions and place hold-conjunctions. If the variable q_r in the place code is set to 1 then the sum of corresponding variable D_r consists of transition conjunctions of all its input transitions and the place p hold-conjunctions:

$$D_r = \bigvee_{m=1}^{M} (\bigvee \bullet p_m \vee hp_m \wedge Code(p_m)[r]). \tag{8}$$

All equations are formatted with VHDL syntax. For example, for the D_0 equation it is denoted as:

```
D(0) <=t(1) or Hp(2) or t(6) or Hp(9);
```

Now all conjunctions and equations are generated and VHDL files can be build up from these expressions. The *GenereitVHDLCCCodForAllSubnet()* method generate files describing combinational circuits and it creates one VHDL file for each subnet. This file consists of equations for place encoding, transition conditions, place hold conditions and D flip-flops. The *GenereitVHDLRGCodForAllSubnet()* method create a one register for each subnet. The registers only differs in the length of vector and they are generated based on synthesis template [33]. The *GenereitVHDLYCodForAllSubnet()* generate all decoders in VHDL. There is created one decoder for each subnet. This method creates list of all outputs for each subnet and then generate truth table for them and store it directly in VHDL syntax. Finally, the *GenereitFinalVHDLFile()* method creates the top-level module. Such created model of logic circuit can be passed into third-party synthesis and implementation tools.

V. Summary

The implementation of the method of synthesis of application specific logic controllers into FPGAs with embedded memory blocks was presented in this article. It is compiled into the stand alone library that can be used in other systems. The colored Petri net as graphical representation of algorithm [4], [34]–[36] is used as entry point to the synthesis method. The special method of logic synthesis [27] is applied. The usage of designed library is fully automated and it makes that it could be easily integrated with design tools in CAD system. As the output there is a set of VHDL files which describes logic circuit.

The method was illustrated by a simple example, which shows the results of execution of designed library.

References

[1] N. Chang, W. H. Kwon, and J. Park, “Hardware implementation of real-time Petri-net-based controllers,” *Control Engineering Practice*, vol. 6, no. 7, pp. 889–895, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0967066198000768

[2] N. Marranghello, J. Mirkowski, and K. Bilinski, “Synthesis of synchronous digital systems specified by Petri nets,” in *Hardware Design and Petri Nets*, A. Yakovlev, L. Gomes, and L. Lavagno, Eds. Boston: Kluwer Academic Publishers, 2000, pp. 129–150. [Online]. Available: http://link.springer.com/chapter/10.1007%2F978-1-4757-3143-9_7

[3] B. W. Bomar, "Implementation of microprogrammed control in FPGAs," *IEEE Transactions on Industrial Electronics*, vol. 49, no. 2, pp. 415–422, 2002. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=993275

[4] D. Drusinsky and D. Harel, "Using statecharts for hardware description and synthesis," *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 7, pp. 798–807, 1989. [Online]. Available: ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=31537

[5] L. Gomes and A. Costa, "From use cases to system implementation: Statechart based co-design," in *Proceedings of 1st ACM & IEEE Conference on Formal Methods and Programming Models for Codesign MEMOCODE'03*. Mont Saint-Michel, France: IEEE Computer Society Press, 2003, pp. 24–33. [Online]. Available: ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1210083

[6] G. Łabiak, "From UML statecharts to FPGA - the HiCoS approach," in *Proceedings of the Forum on Specification & Design Languages FDL'03*. Frankfurt, Germany: ECSI, 2003, pp. 354–363.

[7] M. Doligalski, "Behavioral specification diversification for logic controllers implemented in FPGA devices," in *Proceedings of the Annual FPGA Conference*, ser. FPGAworld'12. New York, USA: ACM, 2012, pp. 6:1–6:5. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2451636.2451642

[8] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=24143

[9] A. Karatkevich, *Dynamic Analysis of Petri Net-Based Discrete Systems*, ser. Lecture Notes in Control and Information Sciences. Berlin: Springer-Verlag, 2007, vol. 356. [Online]. Available: http://link.springer.com/book/10.1007%2F978-3-540-71560-3

[10] T. Kozłowski, E. Dagless, J. Saul, M. Adamski, and J. Szajna, "Parallel controller synthesis using Petri nets," *IEE Proceedings – Computers and Digital Techniques*, vol. 142, no. 4, pp. 263–271, 1995. [Online]. Available: http://digital-library.theiet.org/content/journals/10.1049/ip-cdt_19951886

[11] M. Minoux and K. Barkaoui, "Deadlocks and traps in Petri nets as Horn-satisfiability solutions and some related polynomially solvable problems," *Discrete Applied Mathematics*, vol. 29, no. 2-3, pp. 195–210, December 1990. [Online]. Available: http://dl.acm.org/citation.cfm?id=108745.108751

[12] J. Esparza and M. Silva, "On the analysis and synthesis of free choice systems," in *Advances in Petri Nets 1990*, ser. Lecture Notes in Computer Science, G. Rozenberg, Ed. Berlin/Heidelberg: Springer-Verlag, 1991, vol. 483, pp. 243–286. [Online]. Available: http://link.springer.com/chapter/10.1007%2F3-540-53863-1_28

[13] K. Barkaoui and J.-F. Pradat-Peyre, "On liveness and controlled siphons in Petri nets," in *Application and Theory of Petri Nets*, ser. Lecture Notes in Computer Science, J. Billington and W. Reisig, Eds. Berlin/Heidelberg: Springer, 1996, vol. 1091, pp. 57–72. [Online]. Available: http://link.springer.com/chapter/10.1007%2F3-540-61363-3_4

[14] L. A. Cortés, P. Eles, and Z. Peng, "Modeling and formal verification of embedded systems based on a Petri net representation," *Journal of Systems Architecture*, vol. 49, no. 12–15, pp. 571–598, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=967630

[15] C. Girault and R. Valk, *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Berlin/Heidelberg: Springer-Verlag, 2003. [Online]. Available: http://www.springer.com/computer/swe/book/978-3-540-41217-5

[16] A. Karatkevich and T. Gratkowski, "Analysis of the operational Petri nets by a distributed system," in *Proceedings of the International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science TCSET'04*, Lviv Polytechnic National University. Lviv, Ukraine: Lviv, Publishing House of Lviv Polytechnic, 2004, pp. 319–322. [Online]. Available: ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1365973

[17] A. Węgrzyn, "Parallel algorithm for computation of deadlocks and traps in Petri nets," in *10th IEEE International Conference Emering Technologies and Factory Automation ETFA'05*, vol. 1, Universita di Catania. Catania, Italy: Piscataway, IEEE Operation Center, 2005, pp. 143–148. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1612513

[18] K. Biliński, M. Adamski, J. Saul, and E. Dagless, "Petri-net-based algorithms for parallel-controller synthesis," *IEE Proceedings – Computers and Digital Techniques*, vol. 141, no. 6, pp. 405–412, 1994. [Online]. Available: ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=331627

[19] E. Soto and M. Pereira, "Implementing a Petri net specification in a FPGA using VHDL," in *Design of Embedded Control Systems*, M. Adamski, A. Karatkevich, and M. Węgrzyn, Eds. New York: Springer, 2005, pp. 167–174. [Online]. Available: http://link.springer.com/chapter/10.1007%2F0-387-28327-7_14

[20] L. Gomes, A. Costa, J. Barros, and P. Lima, "From Petri net models to VHDL implementation of digital controllers," in *33rd Annual Conference of the IEEE Industrial Electronics Society IECON'07*. Taipei, Taiwan: IEEE, 2007, pp. 94–99. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4460403

[21] M. Węgrzyn and A. Węgrzyn, "Penlogic – system for concurrent logic controllers design," in *Design of Digital Systems and Devices*, ser. Lecture Notes in Electrical Engineering, M. Adamski, A. Barkalov, and M. Węgrzyn, Eds. Berlin: Springer-Verlag, 2011, vol. 79, pp. 215–228. [Online]. Available: http://link.springer.com/chapter/10.1007%2F978-3-642-17545-9_9

[22] M. Adamski and M. Węgrzyn, "Petri nets mapping into reconfigurable logic controllers," *Electronics and Telecommunications Quarterly*, vol. 55, no. 2, pp. 157–182, 2009.

[23] T. Łuba, M. Rawski, and Z. Jachna, "Functional decomposition as a universal method of logic synthesis for digital circuits," in *Proceedings of the 9th International Conference Mixed Design of Integrated Circuits and Systems MixDes'02*, Wrocław, Poland, 2002, pp. 285–290.

[24] A. Bukowiec and A. Barkalov, "Structural decomposition of finite state machines," *Electronics and Telecommunications Quarterly*, vol. 55, no. 2, pp. 243–267, 2009.

[25] A. Bukowiec, *Synthesis of Finite State Machines for FPGA devices based on Architectural Decomposition*, ser. Lecture Notes in Control and Computer Science. Zielona Góra: University of Zielona Góra Press, 2009, vol. 13.

[26] A. Bukowiec and M. Adamski, "Synthesis of Petri nets into FPGA with operation flexible memories," in *Proceedings of the IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits and Systems DDECS'12*, Tallinn, Estonia, 2012, pp. 16–21. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6219016

[27] ——, "Synthesis of macro Petri nets into FPGA with distributed memories," *International Journal of Electronics and Telecommunications*, vol. 58, no. 4, pp. 403–410, 2012. [Online]. Available: http://www.degruyter.com/view/j/eletel.2012.58.issue-4/v10177-012-0055-x/v10177-012-0055-x.xml

[28] K. Jensen, K. Kristensen, and L. Wells, "Coloured Petri nets and CPN tools for modelling and validation of concurrent systems," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 9, no. 3, pp. 213–254, 2007. [Online]. Available: http://link.springer.com/article/10.1007%2Fs10009-007-0038-x

[29] M. Adamski and J. Tkacz, "Formal reasoning in logic design of reconfigurable controllers," in *11th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems PDeS 2012*, Z. Bradáč, F. Bradáč, and F. Zezulka, Eds., Brno, Czech Republic, 2012, pp. 1–6. [Online]. Available: http://www.ifac-papersonline.net/Detailed/57207.html

[30] A. Karatkevich, "On macroplaces in Petri nets," in *Proceedings of IEEE East-West Design & Test Symposium EWDTS'08*. Lviv, Ukraine: IEEE, 2008, pp. 418–422. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5580151

[31] A. Węgrzyn, "On decomposition of Petri net by means of coloring," in *Proceedings of IEEE East-West Design & Test Workshop EWDTW'06*, Sochi, Russia, 2006, pp. 407–413.

[32] J. Tkacz, "State machine type colouring of Petri net by means of using a symbolic deduction method," *Measurement Automation and Monitoring*, vol. 53, no. 5, pp. 120–122, 2007. [Online]. Available: http://www.pak.info.pl/index.php?menu=artykulSzczegol&idArtykul=401

[33] S. Brown and Z. Vernesic, *Fundamentals of Digital Logic with VHDL Design*, 2nd ed. New York: McGraw-Hill, 2005. [Online]. Available: http://highered.mcgraw-hill.com/sites/0073380334/

[34] G. Borowik, M. Rawski, G. Łabiak, A. Bukowiec, and H. Selvaraj, "Efficient logic controller design," in *Fifth International Conference on Broadband and Biomedical Communications IB2Com'10*, Malaga, Spain, 2010, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5723633

[35] G. Łabiak, M. Adamski, M. Doligalski, J. Tkacz, and A. Bukowiec, "UML modelling in rigorous design methodology for discrete controllers," *International Journal of Electronics and Telecommunications*, vol. 58, no. 1, pp. 27–34, 2012. [Online]. Available: http://www.degruyter.com/view/j/eletel.2012.58.issue-1/v10177-012-0004-8/v10177-012-0004-8.xml

[36] M. Doligalski, "Behavioral specification of the logic controllers by means of the hierarchical configurable Petri nets," in *11th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems PDeS 2012*, Z. Bradáč, F. Bradáč, and F. Zezulka, Eds., Brno, Czech Republic, 2012, pp. 87–90. [Online]. Available: http://www.ifac-papersonline.net/Detailed/57239.html