Visual Microcontroller Programming Using Extended S-System Petri Nets

KOK MUN NG
Faculty of Electrical Engineering
Universiti Teknologi Mara (UiTM)
40450 Shah Alam
MALAYSIA
ngkokmun@salam.uitm.edu.my

ZAINAL ALAM HARON
Faculty of Electrical and Electronic Engineering
Universiti Tun Hussein Onn Malaysia
86400 Batu Pahat
MALAYSIA
zainalal@uthm.edu.my

Abstract: - In this paper, we present the development work on a visual microcontroller programming tool based on an extended form of S-System Petri Nets (SSPN). By using the extended form of SSPNs we were able to describe in visual form subroutines, interrupts, I/O operations, arithmetic operations, and other programming constructs in a microcontroller application program. Construction of the visual programming tool included the development of a drawing editor which utilized directed graphs as internal model for created SSPN diagrams, and a parser to check for correct diagram and sentence syntax. The parser developed uses context-free diagram and string grammars for the diagram and sentence syntax checking, and upon successful parsing the tool automatically translates the SSPN-represented application program into assembly code for a target microcontroller.

Key-Words: - S-System Petri Net (SSPN), S-System Petri Net Generator (S-PNGEN), Directed Graph Structure, Context-free Graph Grammar, Graph Transformation

1 Introduction

The ordinary Petri net (PN) of Carl Petri has been subjected to numerous extensions to give it the descriptive capability to visually describe control algorithms of logic controllers and embedded systems. Examples of such extension are Grafcet [2] and Signal Interpreted Petri Nets (SIPN) [5][7][15] which were used to express in graphical form application programs for programmable logic controllers. These graphical application programs were then compiled to generate source code for the logic controllers.

PN extensions such as Shobi-PN [4], Coloured Petri Net (CPN) [6][8], HPDS [9], Safe Petri Nets [16] and Grafcet [11] were specially developed for particular digital controller platforms. While such extensions to the PN have increased its capability to visually express application programs for many kinds of systems, nevertheless, there still exist some limitations in the descriptive capabilities of these extended PNs, particularly in regards to the availability of some useful features and construct suitable for microcontroller programming. Hence, this work addresses this issue by introducing extensions to the original S-System Petri net (SSPN) [3] to create those missing features which facilitate graphical microcontroller application program writing.

The extensions added to the original SSPN allow behavior specifications at the places and transitions in the SSPN diagram. With these extensions, it is now possible to associate the places and transitions with arithmetic expressions, assignments, or Boolean functions.

Such practical implementation and applications called for the development of good support tools to edit, visualize and translate the PN [4][5][6][7]. A number of PN-based programming tools have been developed for this purpose [5][7][10][13][17]. Nevertheless, these tools were customized for programmable logic controllers [5][7] and digital systems [4]. They have limitations when used for writing microcontroller application programs.

In this work, a visual microcontroller application programming tool called S-PNGEN (S-System Petri Net Generator) has been developed which utilized the extensions added to the basic SSPNs to fully capture and describe the logic and control algorithms of a microcontroller application program. The S-PNGEN environment allows on-screen drawing and editing of SSPN diagrams, diagram and textual syntax checking, diagram translation, and also assembly code generation.

The next section introduces SSPN and the extensions added to allow its use as a graphical programming language for microcontrollers. Section 3 describes S-PNGEN while section 4 explains how application programs are expressed as SSPN diagrams in S-PNGEN and how the SSPN diagram is parsed and checked for correct syntax before being translated into assembly code. Section 5 shows the parsing results and an example of code generated while section 6 concludes the paper.

2 Extended SSPN

SSPN is a bipartite directed graph represented by 5-tuple, PN = (P, T, A, W, M_0) where:

- P = {p1, p2, p3, ……., p_n} is a finite set of places
- T = {t1, t2, t3,……., t_n} is a finite set of transitions
- A $\subseteq$ (P x T) U (T x P) is a finite set of arcs
- W: A $\rightarrow$ 1 is the weight function attached to each arc
- M_0: P $\rightarrow$ {0,1,2,…..} is the initial marking

A defining characteristic of SSPN that differentiates it from other PNs [5][6][7][8][9] is its restriction of allowing only one input arc and one output arc for each transition. Places in an SSPN, on the other hand, can have more than one input and output arcs. While this property seems very trivial at first sight, it is nevertheless of fundamental consequence since its inclusion allows the SSPN to express in a formal manner the causality and the sequential attributes of an asynchronous control system. Hence, the sequential nature of SSPN can be utilized to mirror the sequential execution of a microcontroller program.

Together with its characteristic structure, the following extensions are added to the SSPN to control the information flow:

- Every transition can be associated with a Boolean function, an arithmetic assignment or an expression that denotes the firing condition.
- Timed transitions are used to model timing delays in the application program.
- Every place is associated with an output function that assigns a subset of output signals when it is marked.
- A macro place is introduced to indicate the execution of a subroutine or an interrupt handling function. For each macro place, a subnet is needed to represent the subroutine function or an interrupt.

Together with the asynchronous characteristic of the SSPN and the introduction of the above extensions, programming elements such as program control structures, I/O operations, arithmetic operations, timing delays, subroutines and interrupts can now be precisely and fully represented. Similar to a typical PN, the firing process of the extended SSPN abides by the following rules:

- A transition is enabled if all its pre-places are marked and all its post-places unmarked.
- A transition fires immediately if it is enabled and its firing condition is fulfilled or executed.
- A transition without any firing condition is allowed. The transition fires immediately if it is enabled.

2.1 An example

The extended SSPN has all the features required to represent microcontroller application programs in visual form. To illustrate this point, an SSPN-represented application program for the system in Fig. 1 will be used.

Fig. 1 shows a microcontroller-controlled mixing process that involves mixing and rinse operation. The mixing operation starts when push button *pb1* is pressed. This opens valves *v1* and *v2*, causing two different liquids to be released into the tank until the liquid level reaches level sensor *s2*, whereupon valves *v1* and *v2* then close. Motor *M1* then immediately starts spinning to stir the mixture for a full 60 seconds. Upon completion of the stirring step, the motor stops and valve *v4* is opened to drain the mixture out of the tank. Valve *v4* closes when liquid level falls below the level sensor *s1*. The rinse operation, on the other hand is activated when *pb2* is pressed; whereupon water is then supplied into the tank via valve *v3* until the level reaches *s2*. Subsequently, motor *M1* starts spinning to rinse the tank for 60 seconds. Valve *v4* then opens to drain water out of the tank and closes when the sensor *s1* level is reached.

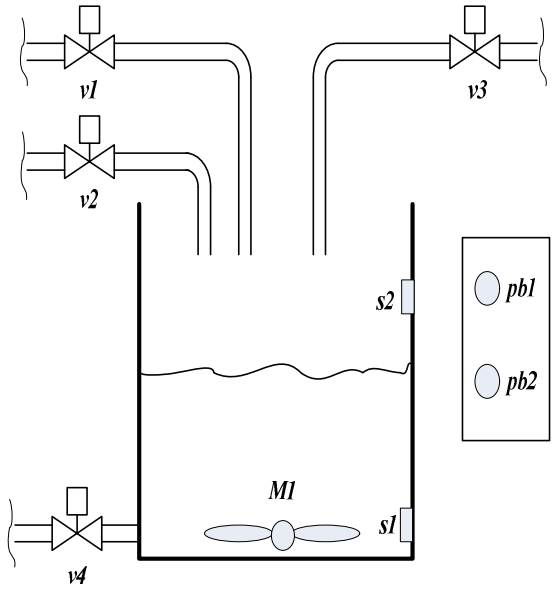

Fig. 1 Mixing Process

The extended SSPN-represented microcontroller application program for the system in Fig. 1 is shown in Figs 2 and 3, respectively. The figures show how the extended SSPN diagram is used to describe graphically the desired control algorithm to represent I/O operations, timing delays and subroutine in a typical microcontroller program.

In representing I/O operations, each transition is assigned an expression to examine the state of an input device while places are assigned output functions to

assign the states of output devices. This can be illustrated for instance in place *p1*, transition *t2* and place *p3*. Place *p1* is marked with a token which is also the starting place that indicates the first line of the application program. *p1* has output functions that deactivate valve *v1*, *v2*, *v3* and *v4* to bring the system to its initial state. The marking of *p1* enables transition *t2* and *t3* respectively. Therefore, if *pb1* is pressed at this stage, *t2* fires as pressing *pb1* fulfills expression "*pb1==1*". The firing of *t2* activates *p3* which has output functions that activate valve *v1* and *v2*. These extensions, which allow expression and output functions to be used to describe I/O operations, have proved useful in the tool development work and their application can be observed in other parts of the SSPN diagram in Fig. 2 and Fig. 3.

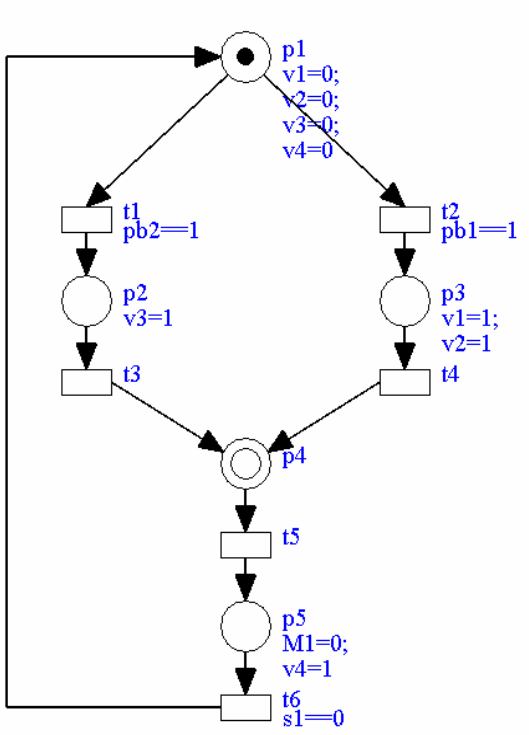

Fig. 2 SSPN diagram for mixing process

In a typical microcontroller program, a subroutine (function, method, procedure, or subprogram) is a portion of code within a larger program, which performs a specific task and can be relatively independent of the remaining code. The diagram in Fig. 2 represents a subroutine using place *p4* which is a macro place. This macro place is further defined by a subnet in Fig. 3 which represents the tasks within the subroutine that models the stirring operation of the system. It has an entry place *p4* which is named after the macro place that owns it. I/O expressions and functions in the subnet examine the states of sensors *s1* and *s2* (level detection) and assign output states to valves *v1*, *v2*, *v3* and the motor *M1* to start the stirring operation for 60 seconds. Transition *t9* illustrates the extension added to the SSPN to describe timing delays; specifically the 60 seconds delay intended for this stirring operation. The tasks in the subnet are completed when exit place *p8* is marked. Hence, the execution flow of the subnet returns to the main net in Fig. 2. Similar rule of using macro places is also applied to describe interrupts.

Another application of the extension added to the original SSPN is to use it in this work to model transitions without any firing conditions. Such transitions are *t3*, *t4* and *t5*. These transitions fire immediately when their pre-places are marked. Non-conditional transitions are deemed useful as they can be used as a "no operation" command in the microcontroller program.

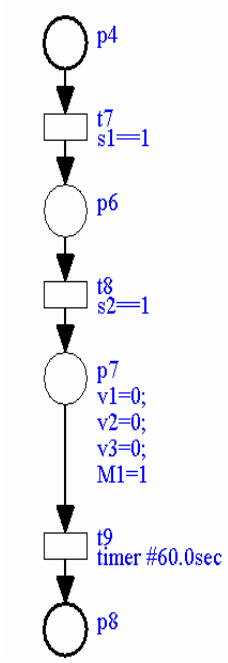

Fig. 3 Subnet for macro place *p4*

3 S-PNGEN

S-PNGEN (see Fig. 4) is wholly written in Java. The editor environment allows free-hand editing with PN components. Places, transitions and arcs can be created and deleted with the mouse. A simple layout routine in the editor ensures that places and transitions are "snapped" to a grid. However, the construction and drawing of the PN representation is solely the responsibility of the user as the layout routine neither auto-route arcs nor does it changes the position of places and transitions.

The editor also provides facilities for modifying the position and size of the components. *Attributes* of the places or transitions such as name and functions can be updated easily with the editor. When a transition

component is selected, arithmetic assignments, expressions or Boolean functions can be added while output functions can be added to places.

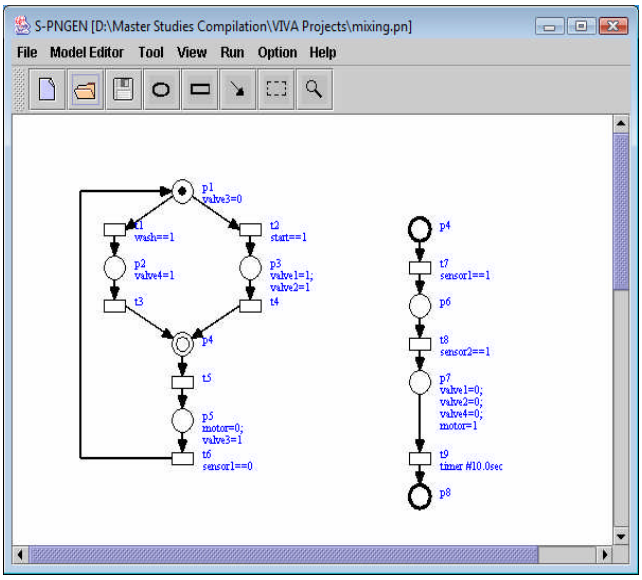

Fig. 4 Snapshot of the S-PNGEN

Fig. 5 shows the structure of S-PNGEN. It consists of the diagram editor where the SSPN is drawn; the data structures that store information of the spatial relationship and *attributes* of places and transitions; a parser that ensures a syntactically correct diagram and textual input, and finally, a code generator that generates assembly code for the target microcontroller from intermediate representations constructed by the parser.

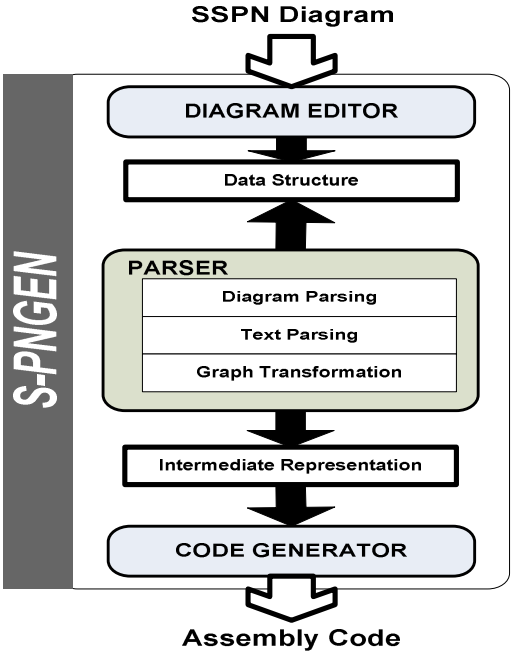

Fig. 5 Structure of S-PNGEN

4 Implementation

4.1 Graphs and Graph Grammar

Graphs have been widely used as internal models in visualization and diagramming tools [18]. Graphical tools such as DIAGEN [5][7] are based on hypergraphs and hypergraph grammars and they used them for syntax specification. Other graphical tools such as PROGRES [10] and VISPRO [13] similarly use graphs as internal models and customized graph grammars to evaluate diagram syntax.

S-PNGEN, in contrast, adapted a regular directed graph structure [12] as internal diagram models and context-free graph grammar for syntax specification. PN components (places or transitions) can be stored in the nodes of the graph as java objects while arcs are represented by edges that connect these nodes. The edges link a source node to its destination node. Each time a component such as a place or transition is created, a corresponding java class object (element) is instantiated and stored in the node of the directed graph structure. An arc connecting two components, on the other hand, is represented by an edge connecting the nodes.

Fig. 6 shows the corresponding directed graph structure for the SSPN diagram in Fig. 2. The nodes are represented by ovals and the edges by thin lines. Elements embedded into the nodes such as a place or a transition can be updated with *attributes* such as a name, Boolean functions, arithmetic assignments, expressions and output functions.

The graph structure provides the parser (see Fig. 5) a robust data representation to perform diagram and textual parsing. The parsing algorithm adopts context-free graph grammar for diagram syntax specifications and context-free string grammar for text syntax specification.

Context-free graph grammar syntax is described by a set of productions of the form $L::=R$ with L(left-hand side) and R(right-hand side). Each production has a non-terminal node on the left-hand side of the production and a sequence of terminal or non-terminal nodes on the right-hand side (see Fig. 7).

The non-terminal nodes in Fig. 7 are represented by ovals, whereas the terminals are represented by rectangular boxes. The productions are applied to a host graph by applying L as a sub-graph of the host graph and matching it with R, which is a set of terminally labeled nodes that define syntax of the diagram.

The parser algorithm iterates through the graph's nodes and applies these context-free graph grammar production rules. The productions depict that each place node may have as many input and output edges as the program algorithm requires, but with a minimum of

one input and one output edge each. A transition node, on the other hand, may only have one input edge and one output edge each to ensure SSPN diagram characteristics are met. With these rules in placed, the SSPN model can be ensured of correct syntax.

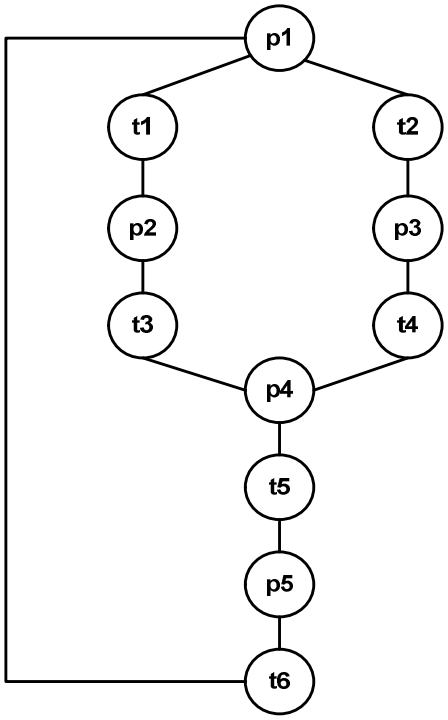

Fig. 6 Directed graph data structure

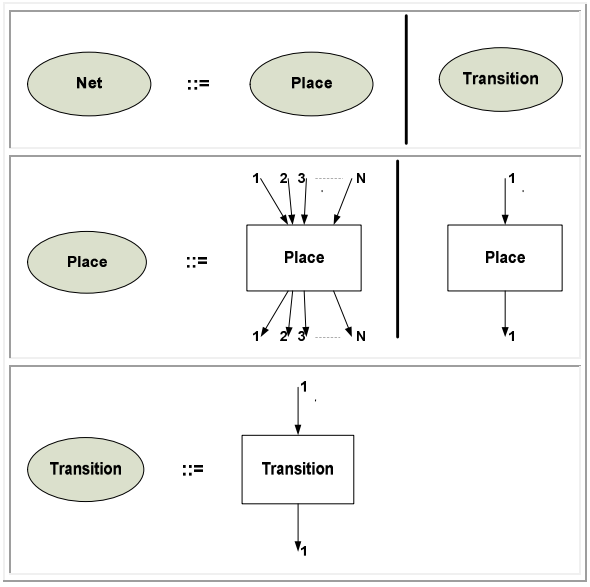

Fig. 7 Context-Free Graph Grammar

4.2 Textual Parsing

Context-free string grammar is adopted in this work to parse textual inputs at the respective places and transitions. The context-free string grammar is a set of production rules based on Backus-Naur Form (BNF) where each production has non-terminal symbols on the left-hand side of the production and a sequence of terminals or non-terminals on the right-hand side. Non-terminal symbols are further defined by other terminal or non-terminal symbols, while the terminal symbols cannot be further defined. All together, a total of 18 BNF rules are applied to ensure correct textual syntax for the SSPN model. Two of the rules are shown here where in these rules, non-terminal symbols are enclosed by angle brackets `<' and `>' while terminal symbols do not have angle brackets. The symbol ***::=*** and | in the production rules are meta-symbols of the BNF. The meta-symbol ***::=*** separates the left-hand side of the production rule from its right-hand side (the definition) whereas the meta-symbol **|** separates alternate forms of definitions. The right-hand side of a production consists of a sequence of terminals and non-terminals. Non-terminal symbols must be further defined. For instance, the non-terminal symbol <*rel op*> is further defined by terminal symbols =<, <=, =, <>, >= and >.

Rule 5
<*expression*> *::=* <*simple expression*> <*rel op*> <*simple expression*> |
port input identifier == <*unsigned number*>

Rule 6
<*rel op*> *::=* < | <= | = | <> | >= | >

These production rules are implemented using the *recursive-descent* method [1]. A parse tree gradually builds up in a top-down manner starting from the root, which is the top most non-terminal of the grammar as the recursive parser evaluates the sentence from left to right. Failure to construct the parse tree points to the presence of syntax errors in the sentence. For instance, rule 5 applies to expression "*pb1 == 1*" at *t1* which resemble definition *port input identifier == <unsigned number>*. The parse tree for this expression is shown in Fig. 8 where the sentence has correct syntax as all terminals are reached.

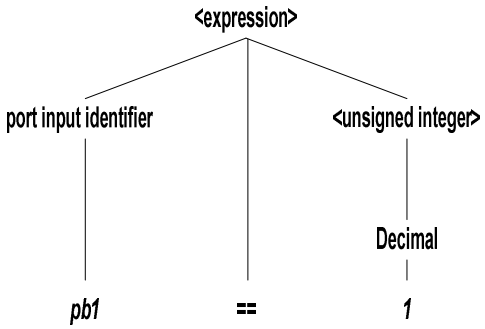

Fig. 8 Parse tree for "*pb1 == 1*"

4.3 Graph Transformations

While the parser ensures correct diagram and textual syntax, it finally generates as output a graph representation that models the semantics of the diagram. The representation is simply an abstraction from the original graph structure to hold the intermediate codes for constructing a reduced graph structure. The abstraction is usually carried out using graph transformation [13][14].

Graph transformation production rules are applied by the parser to the host graph to create the abstract graph. Fig. 9 shows the transformation rules. The rules in Fig. 9 find the sub-graph of the host graph that matches the structure in the left-hand side of Fig. 9 and replaces it with a single *switch* or *block* node (right-hand side). Hence, the *switch* or *block* node reduces the structure on the left-hand side into a single node. The reduced node is further associated with attributes necessary for proper code generation. These attributes include the starting node object (*s*), the ending node object (*e*), the sequential nodes' objects (*r*) in between the starting and ending nodes (if any) and the control flow information such as the firing sequence (*f*) between the objects.

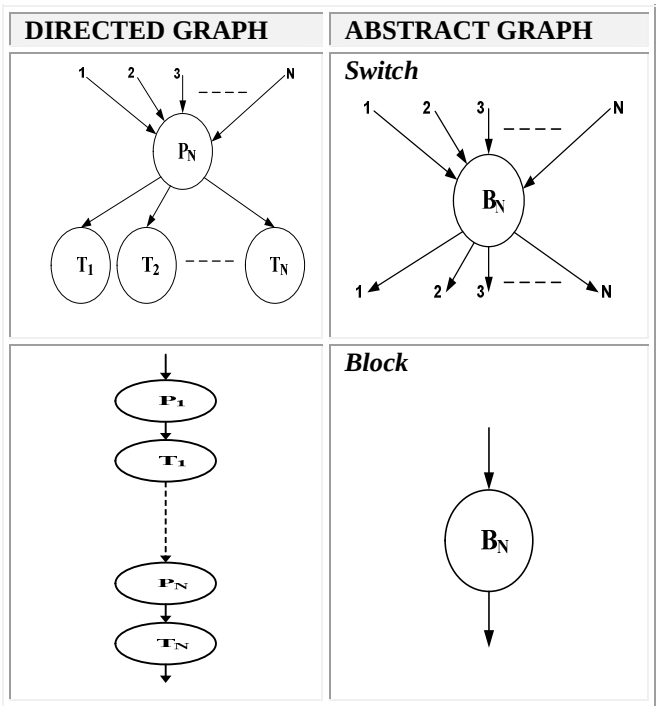

Fig. 9 Graph transformation rules

The transformation is best illustrated in Fig. 10. The figure shows the abstract graph derived from Fig. 6 after graph transformation has taken place. Here, switch B1 holds objects p1, t1 and t2. Block B2 holds objects p2 and t3 while block B4 holds objects p4, t5, p5 and t6.

B1, B2, B3 and B4 are associated with attributes defined as *s, r, e* and *f*, as mentioned earlier. Objects from the host graph were categorized and became the subset of *s, r, e* and *f*. Table 1 shows a summary of the objects reduced for each node under this transformation. These attributes provide a flexible mechanism for transporting information for evaluation by the code generator. For instance, *f* in B1 contains an ordered subset (t1,p2) which denotes the firing of *t1* passes control flow to *p2* which is the starting node object in B2.

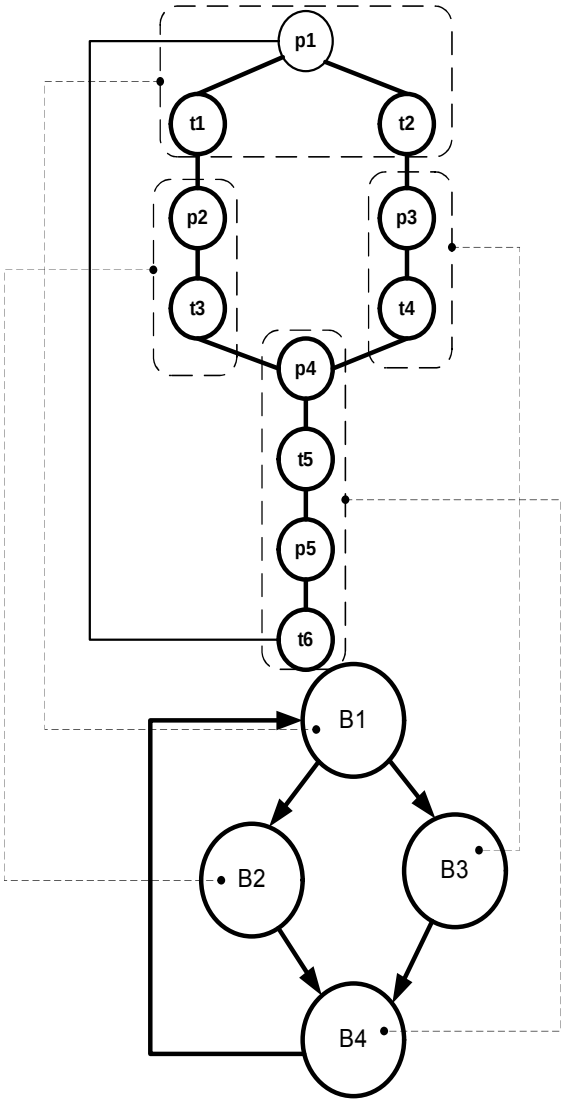

Fig. 10 Abstract graph

The abstract graph serves as an intermediate representation which lays out the semantic execution order which should be taken by the code generator. The semantic execution order for each node is evaluated in the order of *s - r - e*. The graph also provides the dependency information between nodes via *f*. Hence, this provides necessary control flow information to aid code generation by accessing the objects in *s*, *r* and *e* in a sequential manner.

Node	Attributes
B1	*s* = {p1} *r* = {} *e* = {t1, t2} *f* = {(t1, p2), (t2, p3)}
B2	*s* = {p2} *r* = {} *e* = {t3} *f* = {(t3, p4)}
B3	*s* = {p3} *r* = {} *e* = {t4} *f* = {(t4, p4)}
B4	*s* = {p4} *r* = (t5, p5) *e* = {t6} *f* = {(t6, p1)}

Table 1 Objects regionalized in abstract graph

5 Result

5.1 Parsing Result

Similar to any compiler tool, S-PNGEN reports the parsing results to indicate to the user on the occurrence of textual and diagram syntax errors when syntax rules are violated during development of the SSPN model. Both results of textual and diagram parsing in the tool are reported in a window as shown in Fig. 11 when the user starts the compilation via a button click.

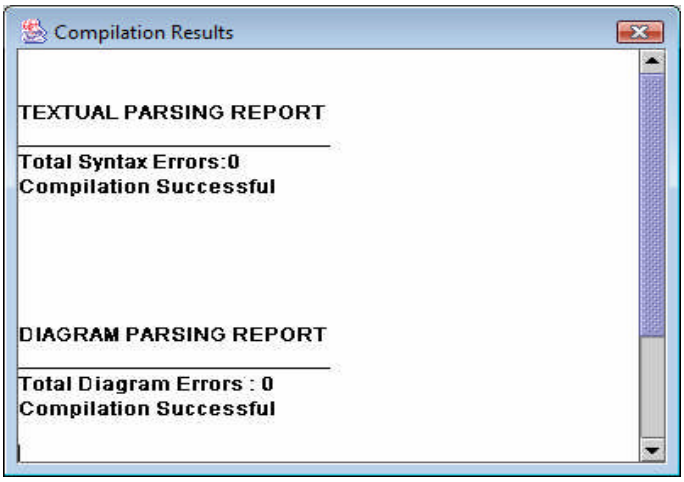

Fig. 11 Snapshot of parsing results

Fig. 11 shows the parsing result for the SSPN diagram for the mixing process in Fig. 2 and Fig. 3 where both textual and diagram parsing yielded no errors and compilation was indicated as successful.

5.2 Code Generation

In the final stage, the code generator traverses the abstract graph to generate assembly code. Like any other type of compiler, the code generator works in a prescribed order of execution. The following pseudo code outlines the function call *generateCode* that was utilized to execute code generation. The function *generateCode* in turn call another function *emitProgramTemplate* where code generation takes place in a layout that starts with I/O and variable declaration, followed by subroutines code, delay subroutine and finally the main program.

```
Public generateCode (Portdata pd){
    Generate output window
    IF buffer NOT null
        CALL emitProgramTemplate to generate code
    ENDIF
}

Public emitProgramTemplate (){
    Generate general I/O equates
    Generate variable equates
    Generate interrupt handling code
    CALL emitSubroutine to generate subroutine

    IF timer variable exists
        CALL emitDelaySubroutine to generate delays
    ENDIF

    CALL InitializedPorts to declare I/O port directions
    CALL EmitInitialValues to define initial values of variables
    CALL IterateASGgraph to iterate the abstract graph

    IF table is not empty
        CALL emitTables to generate table data
        CALL emitEEPROMcontent to generate EEPROM content
    ENDIF
}
```

I/O ports direction is initially declared in the main program, followed by definition of initial values which were assigned to certain variables. The algorithm then calls the *IterateASGgraph* function to access the abstract graph in Fig. 10. *IterateASGgraph* contains a recursive code generation algorithm developed to execute the code generation process for the main part of the program, wherein the algorithm traverses the abstract graph by visiting each node and accesses the objects contained within these nodes in a prescribed order. The algorithm evaluates each phrase of the object's sentence sequentially and returns a sequence of assembly code instructions automatically.

The code generator algorithm in S-PNGEN consists of code templates and the translation is straight-forward as it traverses the abstract graph. Fig. 12 shows the assembly code generated for the SSPN in Fig. 2 and 3. The code was generated according to the code layout outlined in *emitProgramTemplate*.

```
;WRITTEN BY: Ng Kok Mun
;DATE: 24 February 2010
:FOR PIC: 16F84
;RESONATOR:XT(4MHz)
;PURPOSE:Mixing Process

;**************GENERAL AND I/O EQUATES***********
INDF equ 0   ;counter
TMR0 equ 1   ;counter
PCL equ 2    ;program counter
STATUS equ 3 ;status register
FSR equ 4    ;file select register
PORTA equ 5
PORTB equ 6
PCLATH equ 0Ah
OPTION_REG equ 81h    ; option register
TRISA equ 85h
TRISB equ 86h

   LIST P=16F84

;**************VARIABLES EQUATES*****************
#Define pb2 PORTA,0
#Define M1 PORTB,3
#Define pb1 PORTA,1
#Define v5 PORTB,5
#Define v4 PORTB,4
#Define v3 PORTB,2
#Define v2 PORTB,1
#Define s2 PORTA,3
#Define v1 PORTB,0
#Define s1 PORTA,2

   CBLOCK 0x00C
count, Dlay:2, Outside:2, Temp0 , Left, Right
_fsr, count2, Stack
   ENDC

  org 00h
   goto INIT
;**************SUBROUTINES**************************

Sub1
lp1
Loop0
   btfsc  s1
   goto Skip0
   goto Loop0
Skip0
Loop1
   btfsc  s2
   goto Skip1
   goto Loop1
Skip1
   bcf  v1
   bcf  v2
   bcf  v3
   bsf  M1
   call  timer
   goto  lp2
lp2
   return

;**************DELAYS OR TIMERS********************
timer
   movlw   LOW 0x00258
   movwf   Outside
   movlw   HIGH 0x00258
   movwf   Outside+1
OuterLoop1
   movlw   LOW 0x04F20
   movwf   Dlay
   movlw   HIGH 0x04F20
   movwf   Dlay+1
Lp1
   decf   Dlay, f
   btfsc   STATUS,2
   decfsz   Dlay+1, f
   goto   Lp1
   movlw  -d'1'
   addwf  Outside, f
   btfss   STATUS, 0
   decf   Outside+1, f
   clrf  count2
   movf  Outside, w
   btfsc  STATUS, 2
   incf  count2, f
   movf  Outside + 1, w
   btfsc  STATUS, 2
   incf  count2, f
   movf  count2, w
   sublw d'2'
   btfss  STATUS, 2
   goto   OuterLoop1
   return


;**************PROGRAM START HERE ***************
INIT
   bsf STATUS, 5
   movlw b'00011111'
   movwf TRISA ^ 0x080
   movlw b'11000000'
   movwf TRISB ^ 0x080
   bcf STATUS, 5
   clrf PORTA
   clrf PORTB
MAINLINE

B1
   bcf  v1
   bcf  v2
   bcf  v3
   bcf  v4
   bcf  v5
Loop2
   btfsc  pb1
   goto  B2
   btfsc  pb2
   goto  B3
   goto Loop2
B3
   bsf  v3
   goto  B4
B2
   bsf  v1
   bsf  v2
   goto  B4
B4
   call Sub1
   bcf  M1
   bsf  v4
Loop3
   btfss  s1
   goto  B1
   goto Loop3
   end
```

Fig. 12 Assembly Code

In this work, the generator generates assembly code for a PIC microcontroller, specifically the PIC16F84 as the target microcontroller. However, an MPLAB assembler is employed to further assemble the assembly code into machine readable codes. The MPLAB assembler (see Fig 13) is an integrated development environment (IDE) that provides facilities to evaluate the assembly code

generated by S-PNGEN. The code generated by S-PNGEN can be easily copied and paste into the MPLAB text editor. The assembler's build result for the assembly code in Fig. 12 is shown in Fig. 14. Executable absolute machine code is then preceded by the loader into the target microcontroller.

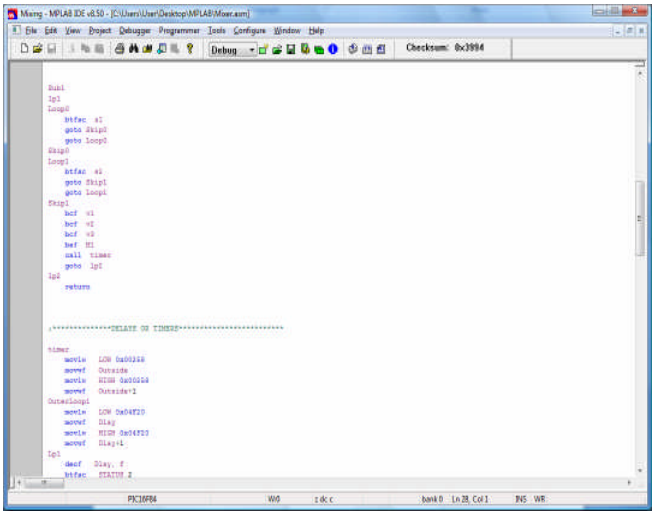

Fig. 13 MPLAB IDE

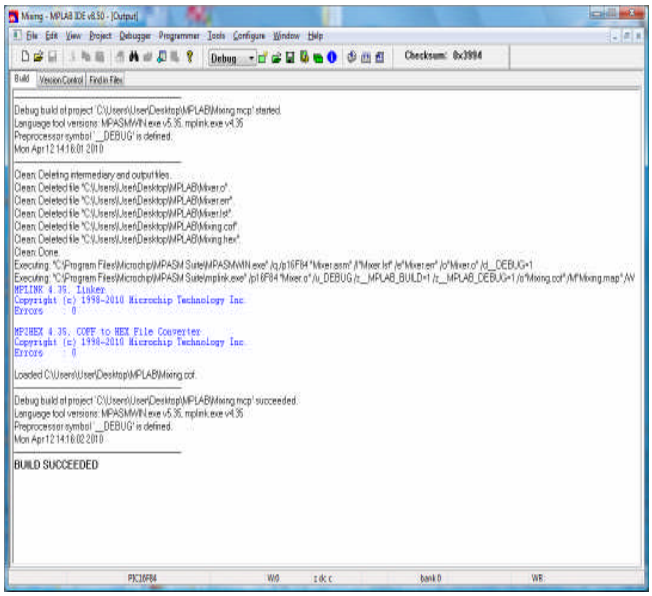

Fig. 14 Assembler's build result

Conclusion

This paper has introduced and described a method for visually expressing microcontroller application programs as SSPNs. Also described was the extensions added to the SSPN to enable it to precisely and fully represent the data flow and control algorithms of a microcontroller application program. The SSPN diagram drawing, editing, and translation were done using a customized tool (S-PNGEN) that employs directed graphs as internal diagram models and context-free graph grammar to evaluate diagram and text syntax. An innovative application of graph transformation has also been described which provided semantics execution sequence and control flow information to the code generator for efficient translation of the SSPN to assembly code. Upon successful parsing and translation of the SSPN programs, S-PNGEN produces machine readable code ready for downloading into the target microcontroller. Code optimization issues, however, were not addressed in this work.

References:

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1982.

[2] R. David, Grafcet: A Powerful Tool For Specification of Logic Controllers, *IEEE Transactions on Control System Technology*, Vol. 3, No.3, 1995.

[3] D. Jorg and J. Esparza, Free Choice Petri nets, Cambridge University Press, 1995.

[4] J.M. Fernandes, M. Adamski and A.J. Proenca, VHDL Generation From Hierarchical Petri Net Specifications of Parallel Controllers, *IEEE Proceedings-E Computers and Digital Techniques*, pp. 127-137, 1997.

[5] G. Frey and M. Minas, Editing, Visualizing, and Implementing Signal Interpreted Petri Nets, *Proceedings of the AWPN 2000*, pp. 57-62, 2000.

[6] W. Kurdthongmee, ertCPN: The adaptations of the coloured Petri-Net theory for real-time embedded system modeling and automatic code generation. *Songklanakarin Journal of Science & Technology*, Vol.25, No.3, 2003, pp. 381-394.

[7] M. Minas and G. Frey, Visual PLC-Programming using Signal Interpreted Petri Nets, *Proceedings of the American Conference 2002 (ACC2002)*, Anchorage, Alaska, pp. 5019-5024, 2002.

[8] K.H. Mortenson, Automatic Code Generation from Coloured Petri Nets for An Access Control System, *ICATPN 2000: International Conference on Application And Theory of Petri Nets, Aarhus, Denmark*, 1825, pp. 367-386, 2000.

[9] W.L.A. Oliveira and N.Marranghello, A High-level Petri Net for Digital Systems, *Proceedings of the XV SBMicro–International Conference on Microelectronics and Packaging*, pp.220–225, 2000.

[10] A. Schurr, A. Winter and A. Zundorf, Graph Grammar Engineering with PROGRES in: A. Schafer and O. Botella (eds.): *Proc. ESEC '95, LNCS 989, Springer Verlag*, 1995, pp. 219-234.

[11] A. Wallen, Using Grafcet to Structure Control Algorithm, *Proceedings of the Third European Control Conference (ECC95), Roma, Italy*, pp. 3160-3165, 1995.

[12] D.A. Watt, and D.F. Brown, *Java Collections: An Introduction to Abstract Data Types, Data Structures and Algorithms*, John Wiley & Sons Ltd. England, 2001.

[13] K.A.Zhang, M.A. Orgun, and K. Zhang, Visual Language Semantics Specification in the Vispro System, *VIP2002: Pan-Sydney Area Workshop on Visual Information Processing, Sydney, Australia*, 2002.

[14] B. Hoffman and M. Minas, A Generic Model for Diagram Syntax and Semantics in J.D.P Polim *et. al* (Eds.), ACALP Workshops – *Proceedings of the Satellite Workshops of the 27th International Colloqium on Automata, Languages and Programming*, No. 8, pp. 443-450, 2000.

[15] K. Loeis and E. Joelianto, Application of Control of Boiler using Signal Interpreted Petri Net (SIPN) Method, *WSEAS Int. Conf. on Electronics, Control & Signal Processing, Singapore*, 2002.

[16] V. Ababii, E.Gutuleac, V.Sudacevschi and D. Odobesco, FPGA-based Implementation of Safe Petri Nets Model, *Proceedings of the 4th WSEAS International Conference, Rio de Janeiro, Brazil*, 2005.

[17] P. Pivonka and L.Chomat, Real-Time Implementation of Petri Nets into PLC, *11th WSEAS International Conference on COMPUTERS, Crete Island, Greece*, 2007.

[18] V. Kayanov, Methods and Tools for Support of Graphs and Visual Processing, *(CSCC 2002): 11th WSEAS International Conference on COMPUTERS, Crete Island, Greece*, 2002.