Automatic Code Generation for Microcontrollers from Place-Transition Petri Net Models


* College of Technology of São José do Rio Preto, SP 15043-020 Brazil (e-mail: dezani@fatecriopreto.edu.br)
* State University of Campinas, Campinas, SP 13083-970 Brazil
** Sao Paulo State University, Rio Preto, SP 15054-000 Brazil (e-mail: {norian, aledir}@ibilce.unesp.br)
*** Sao Paulo State University, Ilha Solteira, SP 15385-000 Brazil, (e-mail: acrsilva@dee.feis.unesp.br)
**** University of Zielona Gora, Zielona Gora, PL 65-246 Poland (e-mail: m.wegrzyn@iie.uz.zgora.pl)

Abstract: This paper describes a program for the automatic generation of code for Intel’s 8051 microcontroller. The code is generated from a place-transition Petri net specification. Our goal is to minimize programming time. The code generated by our program has been observed to exactly match the net model. It has also been observed that no change is needed to be made to the generated code for its compilation to the target architecture.

Keywords: Petri Nets, Automatic Code Generation, CASE tools, Microcontroller-based Systems, Hardware-Software Co-design, Discrete Event Systems, Digital Systems Synthesis.

1. INTRODUCTION

Electronic industry has undergone a tremendous market expansion due to the embedding of electronic devices in various systems such as automobiles, home appliances and communication equipment (De Micheli, and Sami 1996). The complexity of these devices and the ever shorter time-to-market brought the need to efficient design environments, which can help designers to quickly put their prototypes up to work.

Thus, the need for an efficient design environment to which the designer could inform what the system functionality should be like and from which a prototype could be obtained at the end of the design process grow. Such design environments began to appear more than a decade ago and are named hardware-software co-design (Wolf 2003). They can be used for a large variety of digital systems including embedded systems, which is the goal of our work.

Among co-design phases is software synthesis, in which the required code is generated for the target architecture. The software for this phase receives a formal model of a digital system, and transforms this model in computer programs for specific target architecture.

Our environment uses Petri nets as its internal model. The high level description of the system is formalized with the help of a user friendly environment interface. The internal model is then refined by environment tools, and at the end of the co-design process these tools produce a low level Petri net model for each module of the digital system under development. These models are used as input for hardware and software synthesis tools, that is coupled as part of the co-design environment. The main advantage we claim that justifies the development of this environment is that it uses the same internal model from specification to implementation, eliminating the computational cost of moving from one language to another during the design process.

The co-design environment architecture has already been described elsewhere (Marranghello, 2004). This paper deals with a software tool developed to automatically generate the assembly code for Intel's 8051 microcontroller used within the co-design environment. We decided to utilize the 8051 because this microcontroller is widely used not only within our research group but also worldwide. Nevertheless, this tool can easily be extended to take the instruction sets of other microcontrollers and/or microprocessors, automatically generating the corresponding code. The tool takes a place/transition Petri net (Murata, 1989) as input – produced by the co-design environment – and creates, as output, the corresponding assembly code for the 8051 (Intel, 1994). In this work we describe the functionality of this tool, as well as test results achieved.

The structure of this paper is as follows: in section 2 we review some related works; in section 3 we show how code is generated by our tool; and in section 4 we discuss some test results.

2. RELATED WORK

There are several code generation tools taking Petri nets as inputs. Among the most well known are ACDC (Mortensen, 1999), LOOP++ (Lakos and Keen, 1994), CPN-AMI (Kordon, 2010), and Artifex (R-Soft, 2010). In this section
we discuss the most relevant features of these tools with respect to our tool.

Automatic code generation from Design/CPN (ACDC) is a University of Aarhus-based project that aims to develop techniques and tools for the automatic code generation from colored Petri net descriptions of systems. The tool generates an ML code, which has to be compiled to run on the target machine. The software can be specified through a colored Petri net (Jensen, 1997) and simulated with Design/CPN to which the code generation tool is directly connected. After generated the code needs to be compiled.

LOOPN++ is a project based at the University of Adelaide that takes an object oriented Petri net (a CPN extension including object oriented features) model as input to generate the corresponding C++ code through an internal translator. The code generated needs to be subsequently compiled into the target machine code. The code generated by LOOPN++ is a free translation of the Petri net, i.e., it does not necessarily presents a one-to-one correspondence between the Petri net description and the generated code lines.

CPN-AMI is a University of Paris VI Petri net-based CASE environment. It offers functions such as modeling facilities, simulation, model checking and computation of structural properties. The environment includes a simulator and an ADA code generator. The code generator can produce processes from the Petri net model such that the resulting software can be run on a multiprocessor environment.

Artifex is a commercial tool similar to CPN-AMI in the sense that it presents a simulator and a code generator. It allows the user to model, design, simulate and analyze discrete event systems. The system dynamic behavior is drawn through a Petri net-based formalism. The software validates the Petri net model and generates code in C/C++ language. The models can be compiled and run in Artifex environment or compiled and executed as standalone applications.

Our tool takes a Petri Net Markup Language (PNML, 2010) description of a Petri net as input and generates an assembly code for the target architecture as its output. The environment outlined in section 3 does all the Petri net analysis, and produces a PNML description of the system to be synthesized. Our tool takes this description directly transforming the net structure into lines of assembly code for the 8051 microcontroller. It is shown latter in this paper that the code generated does not need further tuning to be run and tested in a microcontroller.

3. CODE GENERATION

The automatic code generation tool has been developed in Java (J2SE – JAVA, 2010) so that it can be run on any hardware platform running JVM (Java Virtual Machine). The tool takes as input a PNML code, which is an extension of XML for Petri Net description. We have got this code from PIPE (2010), which allows the user to describe and to simulate a place/transition Petri net.

Figure-1 displays an example place/transition net. Transition T0 is enabled to fire whenever place P0 has at least two tokens and place P1 has at least one token. When it fires, it will remove two tokens from place P0 and one token from place P1, and will create one token in place P2. For the case presented in Fig. 1 all places have infinite capacity. Now assuming place P2 has capacity equal to one, the first time T0 fires it will create a token in P2, a situation which would disable T0 even if P0 and P1 would remain with enough tokens. In this situation, T0 would only be enabled again if the firing of another transition could remove the token from place P0.

![Fig. 1. Example place-transition net](image-url)

Fig. 2 displays the PNML code corresponding to the Petri net model on the Fig. 1, in which we can observe places declarations between the tags <place> and </place>, transitions declarations between the tags <transition> and </transition>, and arcs declarations between the tags <arc> and </arc>.

Relevant data in place declarations are the id, generated by within tag <place>, and the initial number of tokens, found between the tags <initialMarking> and </initialMarking>.

With respect to transitions declarations the relevant data are their ids, which can be found within tag <transition>.

Three relevant data are found in arcs declarations, they are: arc weight, declared between tags <inscription> and </inscription>; as well as the net elements at the origin and destination of the arc, both declared within tag <arc> by keywords source and target, respectively.

The automatic code generation tool is divided on two major modules: code generator and source code analyzer. Code generation module is further divided into three sub-modules: data memory manager, transition firing generator, and transition generator.

Source code analyzer interprets the PNML code and stores the relevant structures into hash tables. Upon finishing processing of this module three tables are created, one with places information, another with arcs information, and a third one (a hash set) with transition names.

Fig. 3 displays the reduced class diagram. These classes are used to store corresponding data of PNML into structure of data (collections). The class SourceCode is responsible to analyze the code PNML and create the structure of data. The class Code is responsible to generate the code for target-architecture defined in partitioning phase. In our work, this
class generate code for Intel’s 8051, but can be extends to other languages.

```xml
<?xml version="1.0" encoding="iso-8859-1"?>
<pnml>
<place id="P0">
  <name/>
  <value>P0</value>
</place>
... 
<arc id="P0 to T0" source="P0" target="T0">
  <inscription>
    <value>3</value>
  </inscription>
</arc>
</pnml>
```

Fig. 2. PNML code corresponding the Petri Net in Fig. 1.

Class SourceCode takes as input Place and Arc hashables as well as Transition hashset and generates an assembly language code, through its sub-modules.

![Reduced class diagram](image)

Fig. 3. Reduced class diagram

Inside class Code there is a memory data manager method that is used to allocate data memory space for the net places. The management is done through a counter that stores the present memory address, starting from 0020h. Place hashable is read and for each place found a method called setInitialmarking() is run, when it returns the counter is incremented. This procedure includes all place memory locations in class Place. Besides that this method generates assembly instructions to establish memory locations to the initial marking.

Sub-module transition firing generator organizes transition firings according to Petri nets firing rules. In spite of the fact that enabled transitions can fire at random, we are constrained to a target architecture that is sequential in nature, thus an order must be established for the transitions to fire. The program executes every generated instruction in such a sequential order, each corresponding to a transition sub-routine call. After returning from the last sub-routine it unconditionally jumps back to the beginning of the program.

In class Code there is a Transition generator method that reads Transition hashset all the way through and produces check marks for transition firings as well as the corresponding actions. The algorithm presented in Figure-4 illustrates this procedure that is commented in the following paragraphs.

```c
01 begin
02 while (transitions <> empty) do
03   while (arcs <> empty) do
04     if (transition = arc destination) then
05       while (places <> empty) do
06         if (place = arc source) then
07           write("MOV A," + place memory)
08           write("CLR C")
09           write("SUBB A," + arc weight)
10           write("MOV A," + place memory + ",A")
11          endif
12        endif
13      endwhile
14      if (transition = arc source) then
15        if (arc weight = 1) then
16          write("INC" + place memory)
17        else
18          write("MOV A," + place memory)
19          write("ADC A," + arc weight)
20          write("MOV A," + place memory + ",A")
21        endif
22      endif
23    endwhile
24   endwhile
25 endwhile
26 endif
27 endwhile
28 write("EI" + transition + ": RET")
29 endwhile
30 write("END")
```

Fig. 4. Transitions’ code generation algorithm

The algorithm runs until all transitions in the hash set have been properly analyzed, as can be seen from line 02 in Figure-4. Within this loop there is another one to read all arcs from the corresponding hash table (line 03). Within this inner loop there are two conditions to check if the transition under analysis is at the destination (line 04) or the origin (line 14) of the arc.

Being the transition a destination for the considered arc, all places are read from the Place hash table (line 05) to find out those at the origin of the arc (line 06). A set of assembly instructions is generated to verify transition enabling from this set of source places.

Otherwise, being the transition at the origin of the arc all places are searched (line 15) to find out those at the destination of the arc (line 16). A set of assembly instructions is created so that the appropriate number of tokens is added to the output place memory locations.

After the transition generation is complete the code is saved with .asm extension and can be run without further modifications. Figure-5 displays the assembly code generated by algorithm of Figure-4 for the net presented in Figure-1. Note that place P0 corresponds to memory address 0021h, place P1 to address 0020h, and place P2 to address 0022h.
These memory locations were determined by the data memory management module. This module also generated lines 01 and 02, which correspond to the creation of the initial marking.

Module transition firing generator created lines 03 and 04 that are used for the transition sub-routine call and for returning to the first instruction. The program will thus run on an infinite loop. Interesting enough is that as the net in Figure-1 has only one transition, the main loop has only one sub-routine call. A complex program would have as many sub-routine calls as there were transitions in the net.

```
01 MOV 0020h,#02h
02 MOV 0021h,#03h
03 main: ACALL ET0I
04       SJMP main
05 ET0I: MOV A,0020h
06       CLR C
07 SUBB A,#1h
08 JC ET0F
09 MOV A,0021h
10 CLR C
11 SUBB A,#2h
12 JC ET0F
13 INC 0022h
14 ET0F: RET
15 END
```

Fig. 5. Assembly code for the net in Fig. 1

Transition generator module has created the code corresponding to transition T0 (lines 05 to 14). Lines 05 to 07 check the availability of tokens in place P1, and lines 09 to 11 check the availability of tokens in place P0. If either of these checks is false the program jumps to the end of the sub-routine (lines 08 or 12, respectively) avoiding the execution of the transition action. If both checks are true line 13 executes creating a new token in P2.

It is important to note that place capacities are strictly bounded to the target architecture to which the code is being generated. In the case of the work described in this paper the Intel 8051 microcontroller that can store eight bits per register. Thus, the capacity of each place is represented by 8 bits meaning that something between 0 and 255 tokens can be stored in each and every place of the net. During assembly code generation the amount of used memory (either on chip, or external) is checked to ensure it is within the microcontroller storage capacity, as is discussed in the next section of this text.

4. TEST RESULTS

Besides the didactic example presented in the previous sections other tests were run using four nets created with PIPE tool. The details of each of these examples can be found elsewhere (Dezani and Marranghello, 2006). A summary of the test results is presented in Table-1. All Petri nets were implemented and simulated with PIPE. For the sake of comparison, all generated codes were run on PEQui (2010).

The first three models, the dining philosophers, the producers/consumers, and the readers/writers, are well-known computer science problems which can be found in many practical situations such as in factory automation. The last one, a temperature control system consisting of a system used to turn on or off an air conditioning equipment, according to the environment temperature as measured by a temperature sensor. Besides the air conditioning equipment and the temperature sensor the system includes an 8051 microcontroller, an A/D converter and an inverter logic gate.

<table>
<thead>
<tr>
<th>Net models</th>
<th>Amount of</th>
<th>Code size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Places</td>
<td>Arcs</td>
</tr>
<tr>
<td>Dining Philosophers</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td>Consumers Producers</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>Readers Writers</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Temperature control</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

The 8051 microcontroller used has 128 bytes of on chip RAM, 4 Kbytes of internal ROM and can access up to 64 Kbytes of external ROM. As can be seen from Table-1, all generated programs fit into such limitations. Moreover, every program behavior perfectly matched those of the corresponding Petri net simulations observed with PIPE.

Besides producing a code that is compatible to the size of the microcontroller available memory the program must also satisfy two Petri net properties, namely: concurrency and conflict. These properties are illustrated in Figure 6 where a temperature control system example is shown. This system is used for instance to keep stable the temperature in henhouses.

Fig. 6. Place/transition net model for the temperature control system

The system whose Petri net model is depicted in Fig. 6 has a temperature sensor connected to the 8051 microcontroller through an A/D converter between pins 1 and 5 of port P0. Thus, bit 0 is input to the A/D converter through pin 1, and...
its complemented value through pin 5. When this bit is at the logic value high it indicates that the temperature of the henhouse is equal or above 25° Celsius (0.25V) and the fans have to be turned on (as indicated by transition T0 in Fig. 6). Otherwise, when bit 0 is at the low logic value the temperature of the environment is below 25° Celsius and the fans should be turned off (as indicated by transition T1 in Fig. 6).

To represent data input and output in the Petri net model as illustrated in Fig. 6 it is necessary to define the port to be used, as well as the direction of data movement (input or output). Optionally, the pin may also be determined. In this example place input(P0.1) corresponds to the reading of the value available at pin 1, of port P0.

In Fig. 6 we find a conflict between transitions T0 and T1 to which place P1 is said to be a choice-place. Thus, when one of the transitions is fired the other is automatically disabled. To avoid the resulting program always firing the same transition we took an approach, which is to fire each conflicting transition once in alternate turns. To implement that we associate to each choice-place two memory locations, one indicating how many transitions are in conflict for that place and the other to dynamically reflect how many of those transitions have already fired. During the discussion of this example we will name the former as CT (for conflicting transitions), and the latter as FT (fired transitions). The resulting code for transition T0 from the net in Fig. 6 is presented in Fig. 7. A code segment analogous to the one in Fig. 7 is produced for transition T1, and it would be in lines 22-42 if it were to be placed in Fig. 7. This means that the transitions T0 and T1 are executed in alternate occasions.

In our example FT has been allocated to address 17h. CT is not shown in Fig. 7, but it is set to the value #2h, meaning that there are two transitions in conflict for place P1.

Lines 01-04 in Fig. 7 check if transition T2 has already been executed. In line 01 the value of memory location 17h is copied to variable A. In line 02 the carry bit is cleared, and in line 03 the transition’s sequence number (in this case #1h) is subtracted from A. If the carry bit is zero the value of variable A is larger than the transition’s sequence number meaning that the transition has already been fired. Then, in line 04 the jump if not carry instruction will divert the program control flow to line 21, and consequently for the next transition in the sequence, which in our example is transition T1. If the jump is not taken, i.e., transition T0 has not yet been executed, the execution goes on by removing a token from P0 and adding a token in output(P0.2), which will put high logic level in pin 2 of port 0. (lines 05 through 19).

The command in line 20 increments memory location 17h indicating that transition T0 has just been executed and that the next one to be executed is transition T1.

Execution of transition T1 follows the same procedure except that instead of subtracting #1h in line 03, it subtracts #2h. After going through all the sequence of conflicting transitions there is a code segment that tests if CT and FT are equal. If so, it clears FT so that the next time the program executes this part of the code it will execute transition T0 again.

Even the net representing the conflict between transitions T0 and T1 the firing of such transitions is determined from the environment state, that is, the input signals at pins 0 and 5 of port P0 define the conflicting transitions firing. Thus, even knowing this is not the most appropriate way of dealing with conflicts this approach is acceptable as more robust conflict resolution is somewhat redundant being needed only when there is a flaw during circuit assembly, i.e., when both pins are set to high logic level.

5. CONCLUSIONS

Besides perfectly matching the behavior of the corresponding Petri net model, the assembly code produced by our tool run as generated, without the need for any fine tuning. The possibility of directly storing and executing it on the microcontroller speeds up the software synthesis process for the target architecture.

We made some tests with high-level Petri nets too. From these tests we observed that although high-level net models are more compact the assembly code generated for these nets is considerably larger than the one generated for a corresponding place/transition net.

The use of the Java language allows us to execute this application on any computer running a JVM. Thus, we intend to make it available to anyone needing to use such a tool.

The CASE tool we created generates code for the 8051 microcontroller. In future work we intend to develop variations of this tool to produce code for other target architectures of interest and investigate how the code generated by our tool compares to the code generated by a conventional C/C++ compiler.
As some embedded systems applications present time constraints to execute certain functions, in future work we intend to investigate code generation for timed Petri nets.

ACKNOWLEDGEMENTS

The authors would like to express their gratitude for the financial support of the Brazilian Council for Research and Development - CNPq (grant number 132876/2005-1), São Paulo State Research Foundation - FAPESP (grant number 2011/08908-6), and College of Technology of São José do Rio Preto - FATEC

REFERENCES


