PETRI NET APPROACH FOR ALGORITHMS DESIGN AND IMPLEMENTATION

Peter JAKUBČO, Slavomír ŠIMOŇÁK
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice,
Letná 9, 042 00 Košice, Slovak Republic, tel.: +421 55602 3178, e-mail: peter.jakubco@tuke.sk, slavomir.simonak@tuke.sk

ABSTRACT

The paper deals with the design of a new hybrid RISC computer architecture, with the computation driven by a Petri net. A class of Petri nets, suitable for the purpose is proposed within the paper, based on Colored Petri nets. The design of the architecture follows - structure and memory organization, control unit and processing elements. The architecture is implemented using emuStudio emulation platform and its operation is demonstrated by an example at the end of the paper.

Keywords: *Petri nets, Hybrid RISC architecture, Petri net-driven computation, APNA*

1. INTRODUCTION

Petri nets today are well-known and widely accepted formal method for the design and analysis of discrete systems, including those with parallel and distributed nature. Modeling capabilities are easy accessible thanks to their graphical representation, which supports imagination of the system designer. Very valuable and unique are analytical properties too, including automatic calculation of system invariants based on a net structure [1]. Although Petri nets have a lot of unique properties supporting modeling and analysis of systems, the process of implementation of the system designed usually is not as straightforward as in case of some other formal methods. Methods like B-Method [2] or Perfect developer [3] offer the support for generating the implementation form the specification directly and this feature can be considered as an advantage compared to Petri nets. System implementation thus can be time consuming and error-prone task, so our intention here is to address this problem and propose solutions to mitigate it.

We believe a suitable solution could be a new computer architecture, with the computation driven by an associated Petri net. Architecture named APNA is a hybrid one, with the idea of separating the program control-flow from the performing of calculations, which will lead to better support of concurrent programming. APNA is a parallel architecture and its ISA (Instruction Set Architecture) is of RISC type. Hybridity of the architecture results from event-based calculation of Petri net, which is combined with the control-flow execution of a segment of code associated with the Petri net's event.

2. RELATED WORKS

There exist some system implementations based on Petri nets. However, Petri nets are mostly used as formal tool for analysis and verification of systems under development. Systems based acting like some kind of Petri net simulators are available too – it is a different idea of utilizing Petri nets in system design and development.

Within [4], authors present an asynchronous implementation of a Petri net based discrete event control system (DECS), using a Xilinx field programmable gate array (FPGA). The paper reports on the implementation of a Petri net based DECS for an experimental manufacturing system. They aim to provide a guideline to show how to obtain very high speed, concurrent and asynchronous Petri net based controllers.

Another approach is used in [5], where Petri nets are used for controlling and operating flexible manufacturing system (FMS). At the beginning, the paper describes how to synthesize such Petri nets with properties like liveness, boundedness, and reversibility. Then the FMS is described and a hybrid methodology for the design of PN model of the manufacturing system is illustrated. Finally, a PN description language and a PN execution algorithm for supervisory control are shown.

Discussion of implementing a Petri net in VHDL is presented in [6]. The paper discusses how the FPGA architectures affect the implementation of Petri net specifications. Authors propose a method for obtaining VHDL descriptions amenable to synthesis, and tested against other standard methods of implementation.

The works mentioned above are concentrating on hardware implementation of systems based on Petri nets. However, there exists another solution, namely a tool for automatic generation of controllers' implementation code from Petri nets models, presented in [7]. The generated code is amenable to be deployed into common platforms using widely used high level programming languages, such as C, C++, and Java. The generated code is linked with platform specific functions, supporting different types of implementation platforms, ranging from low-cost microcontrollers to workstations, and including microcontroller IPs (Intellectual Property) to be embedded into FPGAs (Field Programmable Gate Arrays). The system controller behavior is modeled using IOPT (Input-Output Place-Transition) Petri Nets models, which are represented through PNML (Petri nets Mark-up Language) notation.

3. PROGRAM FLOW OF ALGORITHMS AND PETRI NETS

Algorithms can be expressed in many ways: algebraically [8], using flowchart diagrams [9] or programming language constructions. Every programming language has defined its own set of control structures for controlling the program flow. According to [10, 11] every computable function can be expressed by using the three basic control structures only:

- Sequence (Fig. 1);
- Alternative/branching (Fig. 2);
- Loop/iteration (Fig. 3).

Ordinary Petri nets however are not suitable for deterministic expressing of them, since branching and iteration cause non-determinism (see Fig. 2 and Fig. 3). Using some of high level Petri nets, with higher modeling power we can solve the problem.

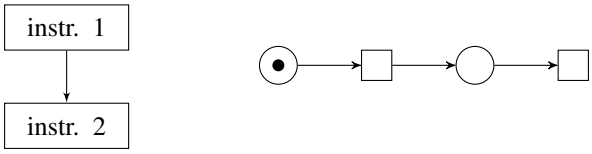

Fig. 1 Transcription of sequence into ordinary Petri net

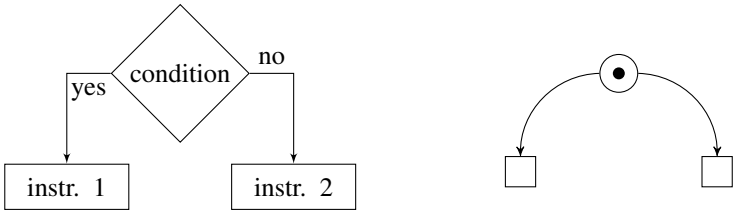

Fig. 2 Transcription of alternative into ordinary Petri net

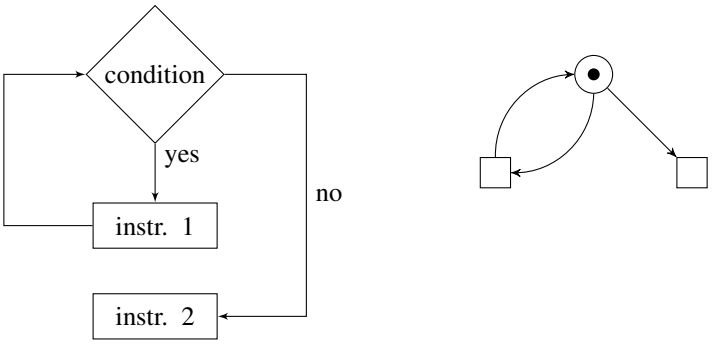

Fig. 3 Transcription of loop into ordinary Petri net

Three basic control structures may not be still enough, if we consider real problems connected with program-program communication, or (at lower level) with program-operating system communication. There exists 'technical' control structures, which are used to solve these kinds of problems:

- Procedure calling (including passing parameters and recursion);
- Exception and interrupt processing.

There exist some sorts of Petri nets that allow modelling of such control structures, however with very limited analytical properties, or they are too complex.

For example, procedure calling introduces a stack that would contain returning addresses. The stack could be modelled in Petri nets in a case when so-called time marks could be used, ie. tokens with the age attribute. When a token is 'pushed' into a place, the age of all tokens already in the place is increased. New token would have the age 0. With that, we can model pushing and popping values to/from the stack: pushed tokens would have the lowest age in the place, and in a case of the 'pop' operation, tokens with the lowest age will be popped (similarly it is possible to model a FIFO queue). Within concurrent procedure execution, there are even more complex questions how to solve racing. We can see that ordinary Petri nets cannot be used. Questions about procedure calls is discussed for example in [12].

Next example – in case of Petri nets, expressing the interrupts would cause spontaneous generation of tokens in some places that would correspond to incoming external

signals representing given event. These questions are discussed in works like [13, 14].

4. COLORED PETRI NET FOR PROGRAM FLOW DESIGN

As it was mentioned above, within the APNA architecture, the flow of computation is driven by a Petri net. But what kind of Petri net it should be? Since we want the Petri net with the modeling power suitable for full program flow control, ordinary Petri nets thus are not suitable due to inability to model deterministic versions of two of three basic control structures (branching, looping). Resulting type of net should be autonomous too, because we want the formal analysis remain possible.

As most interesting of Petri net dialects we consider the Colored Petri nets. Their modeling power goes over the requirements for the APNA architecture and in case of expressing power, they are equal to ordinary Petri nets, so valuable analytical properties are still available. Colored Petri net however is equipped with a modeling language, which can be Turing-powerful. So the original language should be developed for net-expressions and code segments (a basis for performing calculations running on the APNA architecture).

Some restrictions are introduced here to the original concept of Colored Petri nets in order to simplify and speedup the APNA's control unit. In the APNA architecture, one Petri net will correspond to one procedure. Petri net will be used to control the calculations only, not to perform the calculations itself or data transfers. So we are looking for a kind of CPN, with the modeling power as low as possible, but still able to express all the basic control structures in procedures, including the concurrency. The list of restrictions follows:

- Petri net will contain ordinary places only (UNIT type), with the capacity restricted by the architecture implementation.
- Within every procedure, flags (represented by places of a BOOL type) can be used in guards.
- Guarding expressions will consist of one or more simple comparisons using the '=' operator only.
- CPN variables can be used to transfer values of flags only.
- Reference variables will serve as processor registers or addresses in data memory.
- No arc expressions will be used. All the actions will be performed within code segments.
- Simple arcs (no weights) are allowed only.

Another properties correspond to those of standard (non-hierarchical, non-timed) CPN according to the definition found in [15]. The language used in code segments of transitions will be described later on.

[1] Capacity of places P', according to [16] is 8.

Definition 4.1. *CPN for APNA is a net* $CPN_{APNA} = (CPN, F)$*, where CPN is standard CPN, defined as 9-tuple* $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ *(according to [15]) and it holds further:*

- F *is a subset of a set of reachable markings,* $F \subseteq \mathbb{R}(CPN) \cup \emptyset$*, called the set of final markings. Final markings bring the net into deadlock state;*
- Σ *is a set of colors,* $\Sigma = \{UNIT, BOOL\}$*, where* $UNIT = \{()\}$ *and* $BOOL = \{0, 1\}$*;*
- P *is a set of places,* $P = P' \cup P_G$*, where:*
  - P' *are places of color* $C(P') = UNIT$ *with the capacity* K[1]*;*
  - $P_G = \{G_0, G_1, G_2\}$ *are places of color* $C(P_G) = BOOL$ *with the capacity* $K_G = 1$*. These places are initialized with a token* $false$*;*
- G *is a guarding function; every guard expression* $G(t)$ *can use only* 3 *variables (associated from arc expressions)* $a, b, c \in BOOL$*. Every guard expression thus can have one form of the following (supposing pre-places* $\bullet t \in P_G$ *and arc expressions bound the variables):*
  - $[a = \{0, 1\}]$*;*
  - $[b = \{0, 1\}]$*;*
  - $[c = \{0, 1\}]$*;*
  - $[a = \{0, 1\} \wedge b = \{0, 1\}]$*;*
  - $[a = \{0, 1\} \wedge c = \{0, 1\}]$*;*
  - $[b = \{0, 1\} \wedge c = \{0, 1\}]$*;*
  - $[a = \{0, 1\} \wedge b = \{0, 1\} \wedge c = \{0, 1\}]$*;*
- *All arc expressions except those connecting places from* P_G *express to carry only a single simple token* $()$ *(it's identity is from UNIT color) with no variables involved. Variables however can be used in arc expressions connecting places from* P_G*. In this case, variables a, b, and c are bound with a token from* G_0*,* G_1 *resp.* G_2 *(i.e.* $a = G_0, b = G_1$ *and* $c = G_2$*).*

Definition 4.2. $CPN_{APNA} = (CPN, F)$ *computation is defined in the same way as in the case of standard CPN.*

4.1. Example: Algorithm for Factorial Computation

Algorithm of a factorial computation in ordinary programming languages has two variants – recursive one and non-recursive one. Within this example, we will show how to build CPN_{APNA} for the implementation of non-recursive algorithm for factorial computation. The algorithm is shown in Algorithm 1.

Algorithm 1 Non-recursive algorithm for factorial computation

Require: Number $n \in \mathbb{N}$.
Ensure: Factorial $n!$.
procedure `factorial`(n)
begin
$result \leftarrow 1$
while $n > 0$ **do**
$result \leftarrow result \cdot n$
$n \leftarrow n - 1$
end while
return *result*
end

Transcription of the algorithm into CPN_{APNA} must hold to a sequence of steps. The first step is to find a program flow in a graph form, that will be then redrawn into Petri net. The APNA architecture also allows the design of recursive programs, but it is not visible in Petri net structure. Program flow graph of the algorithm introduced above is shown in Fig. 4.

Next step is a transcription into Petri net. The transcription in fact holds to standard transcription schema, shown in figures 1, 2 and 3. Within the program flow graph, it is possible to see two of three fundamental control structures – sequence and loop. According to the standard transcription schema, it is possible to use proposed Petri net structures. However, a non-determinism must be solved. For this purpose, the CPN_{APNA} allows to use guard expressions near the transitions. The guard expressions express required state of one or more places within a P_G set. If the guard conditions are met, the transition can be fired.

Resulting Petri net is shown in Fig. 5.

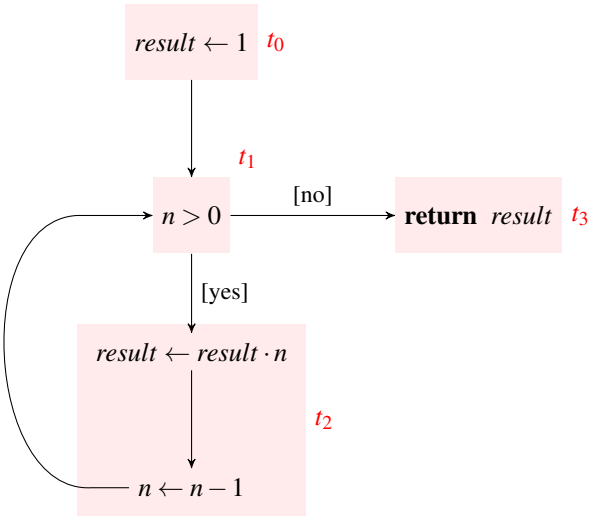

Fig. 4 Program flow of non-recursive algorithm for factorial computation

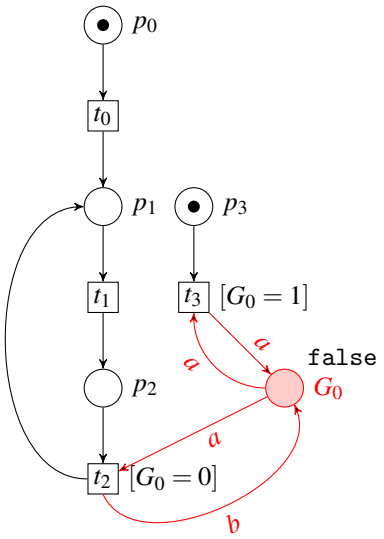

Fig. 5 CPN_{APNA} for program flow from Fig. 4

The designed Petri net is not complete, however. It must be supplied with code segments, one for each transition. By code segments, actions and side-effects of the Petri net are specified. The side effects in fact are computations performed by the algorithm itself. Therefore, computation of the algorithm does not affect the behavior of Petri net. A code segment, assigned to each transition t, is a block of sequential code (instructions) that will be executed when the transition t is fired.

Method shown above is good to be used within analyzing existing algorithms, for example determining if a deadlock can be achieved in the procedure, or in what circumstances the procedure would halt.

We can take the problem also from the other point of view. If a researcher is designing a new algorithm, he can use Petri nets for visualizing algorithm flow and naturally design parallel algorithms, what is enabled by the nature of Petri nets. The visualization can be the language of our imagination.

However, implementation possibilities, described in short in the next section, restrict the programmer to creating only limited structures of such Petri nets.

5. THE APNA ARCHITECTURE

Basic property of APNA architecture is that it separates control of program flow from the computation. The program control is represented by Petri net (CPN_{APNA}) and the computation is represented by sequential code within code segments. Each transition of the Petri net has assigned a single code segment. The code segment is executed when the assigned transition is fired.

APNA is a multiprocessor architecture, therefore all transitions that can be fired concurrently will be really concurrently fired. The architecture has a few other properties, too:

- Parallel modified Harvard RISC architecture enabling more coarse-grained parallelism;

- Programming is relative high-level – program flow is separated from computation;
- Supports procedure calling including recursion.

The APNA architecture consists of two parts – control unit (CU) and from eight process elements (PE) that are performing the computation. The control unit does:

- Loading Petri nets' representations dynamically from main memory;
- Mapping of code segments onto given PE's;
- Realization of Petri net computation (simulation), what includes detection of fireable transitions; detection of concurrent transition firing possibility; firing all concurrently fireable transitions – by mapping and execution of PE's; and computation of new marking of given Petri net.

Process elements perform the computation itself, using simple RISC instructions that do not include jumps[2]. Each process element has its own private program memory that is filled with mapped code segment. Abstract schema of structure organization of the APNA architecture is shown in Fig. 6.

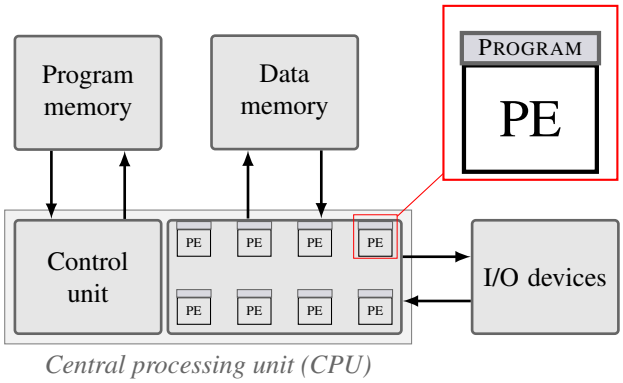

Fig. 6 Modified Harvard architecture for APNA

A virtual machine of the APNA architecture was implemented, within the emuStudio emulation platform [17]. The emulator performs all necessary computations and communicates with virtual program and data memories. CPU, and both memories are implemented as plug-ins for the emuStudio platform. A user besides executing the virtual machine, can also write and compile programs for the APNA, using designed and implemented compiler that is included. The language is similar to standard assembly languages, but it includes also Petri net textual representation. The compiler translates the source codes into binary representation, that can be loaded directly into program memory.

6. DISCUSSION/CONCLUSIONS

The CPN_{APNA}, i.e. Colored Petri net for APNA architecture, excluding the assembly language, is autonomous net that is intended for representation of program flows. This restrict the net usage and it can be stated that the net itself cannot express an algorithm, only its control flow. This is ensured by possibility of modelling of three fundamental control structures (sequence, branching, loop). The computation of the algorithm is expressed in separated code segments with independent language that is a kind of assembler.

The APNA architecture, implemented as virtual machine, combines control and event driven program flow approaches. It is a RISC and multiprocessor architecture that simulates CPN_{APNA} Petri nets, located in memory. It is also modified Harvard architecture, therefore binary forms of Petri nets and code segments are located in the read-only program memory, while data manipulation is performed upon the read-write data memory.

The architecture should help implementing parallel and sequential algorithms allowing to express the program flow using Petri nets. The advantage of this approach is the ability of program flow visualization, simple expression of concurrent program blocks and their synchronization, and formal analyzability of expressed flows, e.g. detection of deadlocks (or procedure halt conditions).

This architecture is designed for researchers and should be improved in the future. We wish to implement it in FPGA chip, and allow interconnect the APNA computation with host processes. Another improvement we want achieve is to create a visual tool for creating algorithms that would allow translate the visual diagram into the assembly language.

ACKNOWLEDGEMENT

This work is the result of the project implementation: Development of the Center of Information and Communication Technologies for Knowledge Systems (ITMS project code: 26220120030) supported by the Research & Development Operational Program funded by the ERDF.

REFERENCES

[1] Štefan Hudák, *Reachability Analysis of Systems Based on Petri Nets*. Elfa, Košice, 1999.

[2] K. Lano, *The B Language and Method - A Guide to Practical Formal Development*, ser. FACIT: Formal approaches to computing and information technology. Springer, 1996.

[3] G. Carter, R. Monahan, and J. M. Morris, "Software Refinement with Perfect Developer," in *SEFM*, 2005, pp. 363–373.

[4] M. Uzam, d. Koç, G. Gelen, and B. Aksebzeci, "Asynchronous Implementation of Discrete Event Controllers Based on Safe Automation Petri Nets," *The International Journal of Advanced Manufacturing Technology*, vol. 41, pp. 595–612, 2009. [Online]. Available: http://dx.doi.org/10.1007/s00170-008-1497-2

[2] All control structures described in previous sections are in standard control-flow architectures implemented using jumps.

[5] M. C. Zhou, F. Dicesare, and D. L. Rudolph, "Design and Implementation of a Petri Net Based Supervisor for a Flexible Manufacturing System," *Automatica*, vol. 28, no. 6, pp. 1199–1208, 1992. [Online]. Available: http://www.sciencedirect.com/science/article/pii/000510989290061J

[6] M. A. Adamski, A. Karatkevich, M. Wegrzyn, E. Soto, and M. Pereira, "Implementing a Petri Net Specification in a FPGA Using VHDL," in *Design of Embedded Control Systems*. Springer US, 2005, pp. 167–174. [Online]. Available: http://dx.doi.org/10.1007/0-387-28327-7_14

[7] L. Gomes, R. Rebelo, J. Barros, A. Costa, and R. Pais, "From Petri Net Models to C Implementation of Digital Controllers," in *Industrial Electronics (ISIE), 2010 IEEE International Symposium on Industrial Electronics*. Fac. de Cienc. e Tecnol., Univ. Nova de Lisboa, Lisbon, Portugal, 2010, pp. 3057–3062.

[8] S. Hudák, "Teoretická Informatika – Algebry Algoritmov," Košice, 2002. [Online]. Available: http://hornad.fei.tuke.sk/~korecko/teorInf/SH_algAloritmov_final.pdf

[9] "Information Processing – Documentation Symbols and Conventions for Data, Program and System Flowcharts, Program Network Charts and System Resources Charts," ISO, p. 25, 1985.

[10] C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Commun. ACM*, vol. 9, pp. 366–371, May 1966. [Online]. Available: http://doi.acm.org/10.1145/355592.365646

[11] E. W. Dijkstra, "Letters to the Editor: Go To Statement Considered Harmful," *Commun. ACM*, vol. 11, pp. 147–148, March 1968. [Online]. Available: http://doi.acm.org/10.1145/362929.362947

[12] S. Wendt, "Modified Petri Nets as Flowcharts for Recursive Program," *Softw., Pract. Exper.*, vol. 10, no. 11, pp. 935–942, 1980.

[13] J. Borkowski, "Region-Based Petri Nets for Modeling Interrupts and Cancellations," in *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, ser. PARELEC '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 67–. [Online]. Available: http://portal.acm.org/citation.cfm?id=824476.825969

[14] S. Christensen and N. D. Hansen, "Coloured Petri Nets Extended with Channels for Synchronous Communication," in *Application and Theory of Petri Nets, Proc. of 15th Intern. Conf.* Springer, 1994, pp. 159–178.

[15] K. Jensen, "An Introduction to the Theoretical Aspects of Coloured Petri Nets," in *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, vol. 803. London, UK: Springer-Verlag, 1994, pp. 230–272. [Online]. Available: http://portal.acm.org/citation.cfm?id=648145.750149

[16] P. Jakubčo, "A Petri Net driven Architecture," Ph.D. dissertation, Faculty of Electrical Engineering and Informatics, Technical University of Košice, 2011.

[17] P. Jakubčo, "emuStudio – An Universal Emulation Platform," Master's thesis, Faculty of Electrical Engineering and Informatics, Technical university of Košice, 2009.

Received October 15, 2011, accepted December 20, 2011

BIOGRAPHIES

Peter Jakubčo was born on 2. 6. 1985. In 2009 he graduated from the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics at Technical university in Košice. He defended his PhD thesis titled 'APNA – A Petri Net driven Architecture' in 2011. He is interested in computer emulation, operating systems and programming languages.

Slavomír Šimoňák was born on 23. 9. 1974. In 1998 he graduted from the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics at Technical University in Košice and defended his PhD thesis titled 'Formal method integration based on transformation of Petri nets and Process algebras' in 2003. Currently he is working as an assistant professor with the Department of Computers and Informatics. His research interests include formal methods integration and application, machine-oriented languages and computer emulation.