ALPiNe: A Hardware Computing Platform for High-Level Petri Nets

S. Bulach, H. Baur, H.-J. Pfleiderer, Z. Kucerovsky*

Department of Microelectronics, University of Ulm,
Ulm, D-89069, Germany

*Department of Electrical & Computer Engineering,
University of Western Ontario
London, Ontario, N6A 5B9, Canada

May 5, 1998

***Abstract**: A motivation for the design of a novel hardware platform for processing algorithms based on High-Level Petri Nets is presented. ALPiNe (Asynchronous High-Level Petri Net) processor is aimed at embedded discrete-event control applications and is characterized by its natural incorporation of external stimuli into the computation flow. The processor consists of two layers of hardware: one for determining when and which computations will take place, and another for effectively performing the actual computations. A hybrid architecture and hardware organization are described in detail. The process of software development is presented, augmented with an illustrative example. In conclusion, comments on advantages and possible future implementations are made.*

Introduction

The last thirty five years have demonstrated that Petri Nets (PN) have the inherent ability to survive, and not only to survive. The rate at which this concept was applied, modified and extended is perhaps analogous to the exponential PN state explosion phenomenon. One such extended version of a Petri Net is known as a Coloured Petri Net (CPN) [KJ96]. The CPNs could be classified as High-Level PNs, being more abstract than their predecessors. In addition to allowing abstract token types they may also incorporate hierarchical capabilities and time extensions [WA94]. Several commercially available software packages support efficient creation, simulation and analysis of algorithms based on CPNs [AW91]. This paper deals mainly with the Design/CPN [MS93] package and its application to modelling complex discrete-event control systems.

Due to their flexibility Petri Nets have been successfully applied to a wide range of problems [TM89], and were found to be best pertinent in performance evaluation and in the design of communication protocols. In general, PNs are well suited for modelling and analysis of systems that may possess concurrent, distributed, parallel, event-driven, asynchronous, reactive and non-deterministic qualities. In yet another dimension, Petri Nets can be used from gate-level hardware design, through register-transfer level (RTL) modelling, all the way to the design of complex software systems [JP81]. However, it should be noted that because Petri Nets have well defined static and dynamic properties they do well in

modelling processes, systems and algorithms that in some sense exhibit analogous characteristics. In other words, it is not very efficient to model spherical objects using triangles!

A very broad class of systems that Petri Nets handle well is known as *reactive systems* [BM91]. A reactive system, informally defined, is said to have an on-going interaction with its environment. It receives the input stimuli from sensors, processes them, communicates back to the environment through actuators, and keeps track of the state [HP85]. Furthermore, embedded systems could be viewed as reactive systems that must satisfy hardware constraints, whereas real-time systems are reactive systems which must also satisfy timing constraints [MP96]. Design, verification and efficient implementation of reactive (embedded and/or real-time) systems still remains an active research field.

At the gate-level, Petri Nets could be used for both combinational and sequential circuit design, especially when gate delays are taken into account. Asynchronous (operating without a clock) sequential circuits are particularly well suited for modelling with PNs, although attempts have been made to extend PNs to handle synchronous sequential circuits, or even mixed asynchronous-synchronous systems [TT97]. This is mainly due to the asynchronous event-driven nature of processes that PNs represent. Note that asynchronous sequential circuits represent simplest reactive processes. Recently, PN based methods were developed which allow automatic synthesis of asynchronous circuits based on Signal Transition Graphs (derivative of marked PN), or signal switching specifications [JC97]. However, this methodology is restricted to the relatively small controller circuits. There is also ongoing research at utilizing PNs at a register-transfer level for system verification and synthesis [SK97]. However, both gate-level and RTL represent so-called *direct solution* or implementation, where a given algorithm is hardwired and is intended for a specific application, as is the case, for example, in the ASIC (Application Specific Integrated Circuit) design. Another type of solution is known as *indirect* and involves use of programmable devices, such as microprocessors and processor-based systems.

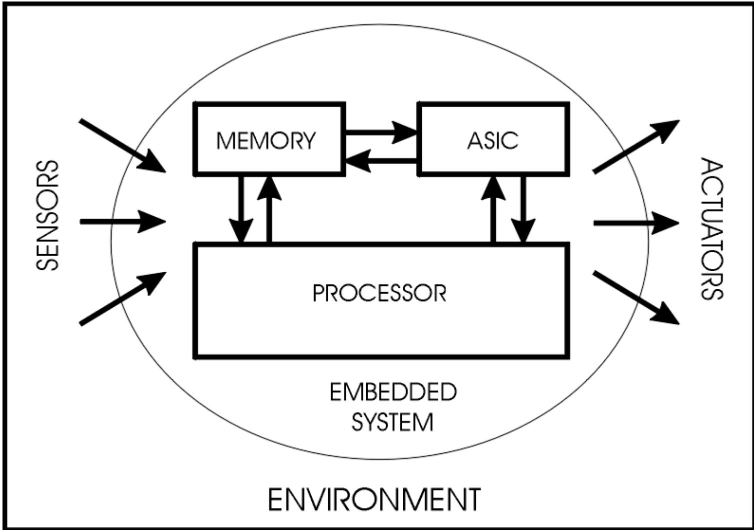

Figure 1. A Simplified Structural View of an Embedded System.

An indirect solution is essentially a software design because the hardware computing resources are provided and are fixed, while the algorithm resides in memory. A complex reactive system would normally contain a processor which has I/O capabilities in order to

communicate with the environment through sensors and actuators. A simplified structural view of such a system is given in Figure 1 [GD94]. Much attention has been given to the design of complex software systems using High-Level Petri Net concepts (for examples, refer to [TH92], [RS96]). However, most of the efforts are directed at employing commercially available processors, thus shifting the focus to the efficient computing strategy given the computing resources and I/O capabilities. The weakness of this approach is that while reactive systems, or in other words, most of the embedded systems, are intended to be reactive or event-driven, commercially available processors are predominantly of the control flow architecture. They are excellent number crunchers but inefficient communicators. Thus, processing cores are normally surrounded by a plethora of peripherals to enable interaction with the environment. However, the overall architecture still remains control driven.

In system design it is often the case that the system description at the early stage of the design is done in dataflow, functional, or some other unconventional language. However, the final implementation is forced to be translated into the procedural language equivalent and then compiled into the target code. The point is that even if the desired programmed functionality of a software system is reactive, event or data driven, the final implementation is done on a hardware platform that is strongly control driven. Could these inefficiencies be addressed at an architectural level?

ALPiNe Architecture

The universality of Petri Nets could be attributed to their flexibility in ascribing a meaning to the concept of tokens, places and transitions. For different applications both places and transitions may mean different things. Some typical interpretations are given in Table 1 [TM89].

Table 1. Possible Interpretations of Transitions and Places

Input Places	Transitions	Output Places
Preconditions	Event	Postconditions
Input Data	Computational Step	Output Data
Input Signals	Signal Processing	Output Signals
Resources Needed	Task or Job	Resources Released
Condition(s)	Clause in logic	Conclusion(s)
Buffers	Processor	Buffers

Note that the second interpretation with input and output data, and a computational step could be well applied to reactive systems. That is to say that a computational step is invoked only when the required input data is present. This principle is reminiscent of the dataflow architecture with a data driven computing organization [TB82], where the computational step is executed once the necessary data becomes available. The computational step itself may represent the execution of one or more instructions, or in other words, a subroutine of arbitrary size, which is concerned with pure computations.

The above observation leads to the conception of a hybrid architecture comprising both data flow and control flow characteristics. Thus, data flow principle is employed to

determine whether or not (and which) computational step is invoked, while control flow principle is used to efficiently perform number crunching once it has been determined which step to execute. These architectural principles are realized in the proposed ALPiNe (Asynchronous High-Level Petri Net) processor. ALPiNe processor has two primary modules: a Petri Net Decision Unit (PNDU) and a Computing Engine (CE). The PNDU module is responsible for making decisions regarding the PN structure, such as determining which transition fires based on the Precondition, or a special condition that must be satisfied for a transition execution to initiate. The PNDU processes its own code, and coding is described in the following section. The CE module is a conventional processor core, optimized for logic, arithmetic and bit manipulation instructions. The CE module also has its own code. Detailed software and hardware organization of the ALPiNe hybrid architecture is presented below.

ALPiNe Software

First of all, the design flow must be thoroughly examined in order to see how High-Level Coloured Petri Nets (in particular, developed with Design/CPN) could be processed by the ALPiNe processor. Clearly, the CPNs manipulated by the Design/CPN package could not be implemented directly on the ALPiNe processor. They are simply too complex, have abstract data types, and the code size is most likely too large. The design flow, shown in Figure 2, should help to see the role of the ALPiNe with respect to CPNs, as well as what type of CPNs could be processed by this processor.

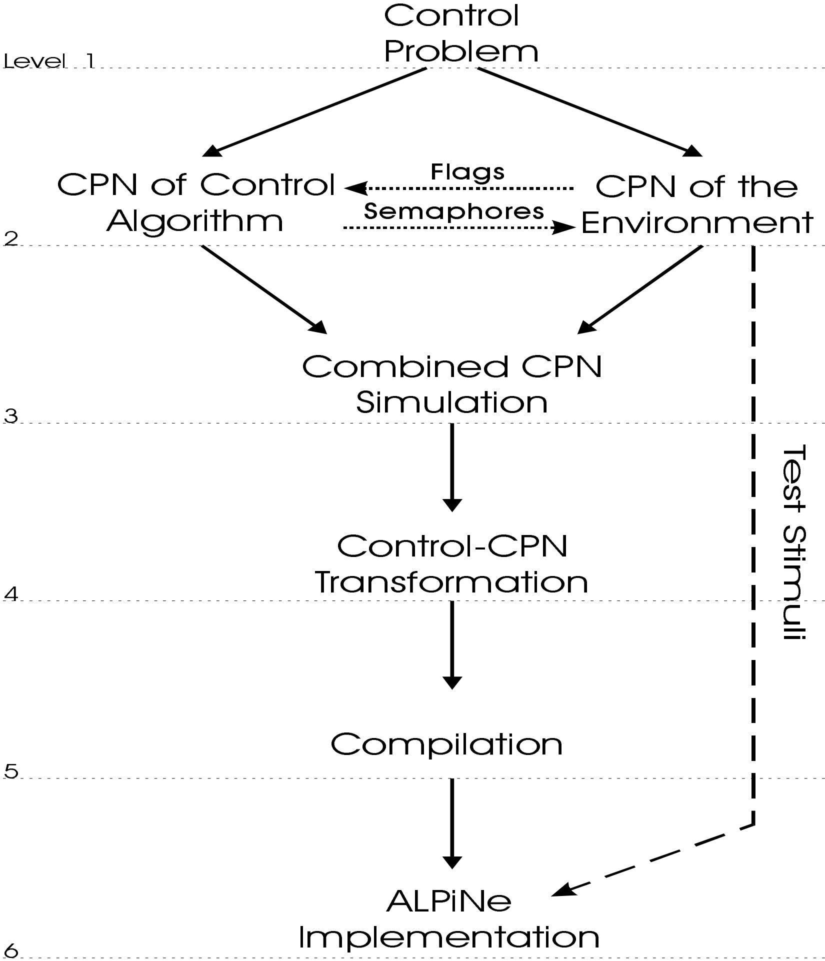

Figure 2. Design Flow for a Control Problem Solution Using the Coloured Petri Nets.

Given a control problem, a solution, which involves the CPN of the control algorithm and the CPN of the environment, is developed with the Design/CPN. In this approach, the environment CPN communicates to the control CPN through *Flags*, while the control CPN uses *Semaphores* to signal its state to the environment. This method allows to model the on-going interactions between the two. The combined model can be simulated and formally analyzed to verify the correctness of the algorithm. If the control model is satisfactory, it must be transformed from its *high-level* CPN (level 2,3) to the *intermediate-level* CPN (level 4). The intermediate-level CPN must be of the form which would allow compilation into the ALPiNe machine code. Thus, the high-level CPN (level 2, 3) is in the format understood by the Design/CPN. The intermediate-level CPN must be in the format understood by the compiler. The rules of transformation and compilation must be clearly defined and are based, of course, on the intermediate-level CPN. The important issue is to optimally specify the intermediate-level CPN which will serve as a bridge between the two different worlds of high-level CPN software (Design/CPN) and the hardware (ALPiNe). Furthermore, the intermediate CPN must deal with real-world binary variables which are digital signals on the wires and processor input pins.

It was chosen that the current version of ALPiNe would handle problems modelled with the CPNs of the Finite State Machine (FSM) subclass. That is, at any given time the processor could only be found in one state (or place). This means that one or more transitions could be enabled at the same time. However, the execution algorithms and the CPN structure ensures that only one will fire, remove the enabling token from the input place and produce a token for a corresponding output place. The FSM restriction on CPNs is achieved by allowing at most one input and output place from any given transition. This restriction does not apply to special Flag and Semaphore places.

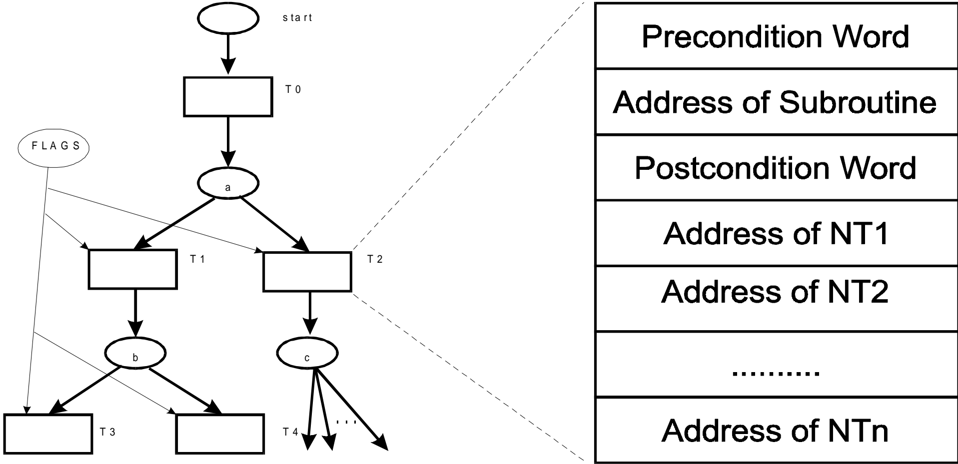

Figure 3. A General Transition Coding Format

The program to be executed on the ALPiNe processor (machine code) mimics the intermediate-level CPN and consists of a list of transitions. Each transition is coded according to the general pattern shown in Figure 3. It contains a Precondition Word, an Address of a Subroutine, a Postcondition Word and a list of addresses of Next Transitions (pointers). Note that places do not appear explicitly in the code. The concept of places is dissolved by the presence of preconditions, postconditions and pointers; these three combined precisely determine the unique state of the processor at any given time. General rules of converting a Design/CPN executable intermediate-level CPN into the ALPiNe code are given below.

The principal components of a CP net are: Data, Places, Transitions, Arcs, Input Arc Inscriptions, Guards, and Output Arc Inscriptions [MS93]. A full translation of the CP Net into the ALPiNe machine code requires the specification of conversion for each of these components.

CPN Data: Data objects are known as tokens, while datatypes are called colorsets. The high-level CPN has fairly complex tokens and colorsets. The intermediate CPN, on the other hand, must reflect hardware dependence in that all (or most of the) tokens are binary variables. These binary objects could be combined into different binary multisets such as Flags (reflect the current state of the environment) and Semaphores (reflect the current state of the system). The size of Flags and Semaphore multisets is determined by the width of their respective registers (Figure 4). This sets the upper boundary on the number of encoded states. Refer to the Railroad Crossing example presented below to see how these multisets could be arranged.

Places: According to the Design/CPN definition, places are locations for holding data. For high-level CPNs places are important components of the algorithm. They represent the state of the modelled system. At the machine code level, places have no real interpretation, as only transitions are encoded as a program. Note, this does not mean that the modelled system becomes stateless! The state may be encoded with Semaphore tokens as required.

Transitions: In high-level CPNs these are defined as activities that transform data. At the machine code level these are primary elements of the program. They specify what computational step (i.e. subroutine) is to be executed once the transition is enabled. In addition, at the intermediate-level transitions also encode Preconditions and Postconditions. This procedure could be observed again from the Railroad Example.

Input Arc Inscriptions (IAI): These specify the data that must exist for an activity to occur.

Guards: Define conditions that must be true for an activity to occur. In terms of the present ALPiNe encoding for the intermediate-level CPN, both IAI and Guards form a Precondition Word. Ideally, the IAI should indicate which tokens (or binary elements) are important to test in the Precondition. Guard specifies the binary AND operation performed on these variables. If the outcome is TRUE, the transition will fire.

Output Arc Inscriptions: Specify data that will be produced if an activity occurs. This CPN component is encoded into the Postcondition Word, in which binary variables of the Semaphore multiset (current state indicators) are set or cleared upon requirements.

Again, refer to the Railroad Crossing Example to observe the relationship between the Design/CPN components and the corresponding elements of the ALPiNe encoding, i.e. Preconditions, Postconditions and Transitions, as shown in Figures 9, 10 and 11. The overall PNDU code structure has the following components: Precondition Word, Address of Subroutine, Postcondition Word, Number of Next Transitions (NNT) indicator, and the actual addresses of Next Transitions, NTA1, NTA2,..., NTAn. Thus, the PNDU has a variable length instruction format. This means that the information about the transition length must be explicitly encoded. The NNT field contains the explicit number of next transitions

The execution of a subroutine is the task of the CE module. The encoding of its logic and arithmetic instructions is done according to the principles described in [HP90].

ALPiNe Hardware

As mentioned above, ALPiNe consists of two modules, each optimized for its own unique function. The block diagram of the ALPiNe processor is shown in Figure 4. Both PNDU and CE have their dedicated memory, with their own data and address buses. This ensures optimum concurrency of the operation.

The PNDU is responsible for processing PN structure related information. It receives new information from the environment through the FLAGS pins. Every new event or change on FLAGS is registered and queued in the FIFO block. At the same time New Event Detector produces a signal to indicate that the new event has been detected. The PNDU communicates to the outside world through Semaphores. The Semaphores are combined with "oldest" Flags to form a State Word. The Comparator checks for equivalence between the State Word and the Precondition Word. A Transition Register File is where Transition information is temporarily stored while a comparison is performed. The Precondition Word, Subroutine Address and the Postcondition Word of the currently executed transition become available to the Computing Engine. The CE also receives a prompting signal to start processing a subroutine and upon execution indicates that it is finished. It has an option of altering the Postcondition Word based on the results of computation.

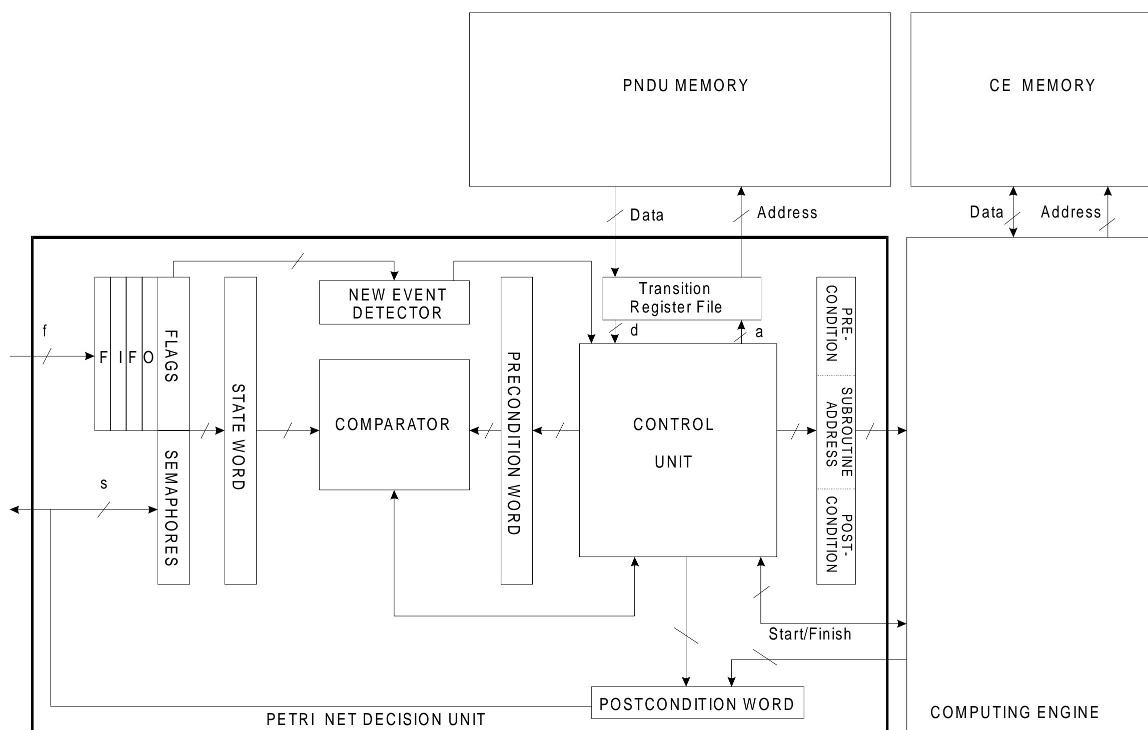

Figure 4. Simplified Block Diagram of the ALPiNe Processor.

The functionality of the PNDU module is flowcharted in Figure 5. For the following discussion refer to Figure 10, transitions T0, T1 and T2. Upon Power On, the Transition Pointer (equivalent of the Program Counter) is initialized to point at the memory location of the first transition in the program (Step 0). The PNDU Memory is accessed and the transition is loaded into the Transition Register File (Step 1). The Precondition Word is then compared to the Status Word (Step 2). If they are equal, the transition is said to fire. The Subroutine Address is passed on to the Computing Engine along with the Start signal (Step 3). The Precondition is also available to the CE for examination, and the Postcondition could even be

modified by the CE. After the CE completes the subroutine, it asserts the signal Finished and may overwrite the Postcondition (Step 4). If the Postcondition was not modified by the CE, the SEMAPHORES register receives the original Postcondition. At the end of this cycle the Transition Pointer is given the first address contained in the Next Transition List of the fired transition (Step 5). The execution then continues in the same fashion from Step 1 to Step 2.

If upon testing the Precondition against the State Word it is found that they are not equal, the Transition Pointer is given the second address from the Next Transition List of the previously fired transition. If the second Next Transition does not fire the subsequent transitions are all tested until the Next Transition List is exhausted. This is represented by the cycle of Steps 1,2,6 and 7 until all the transitions on the list are tested. In this case the execution cycle suspends (at place p6) and waits for new Flags or new events to arrive. Once a new event arrives, the Transition Pointer is updated to the first address on the Next Transition List and the testing of Next Transitions starts all over again until either one of them fires, or after testing them all the execution suspends and awaits a new Flag.

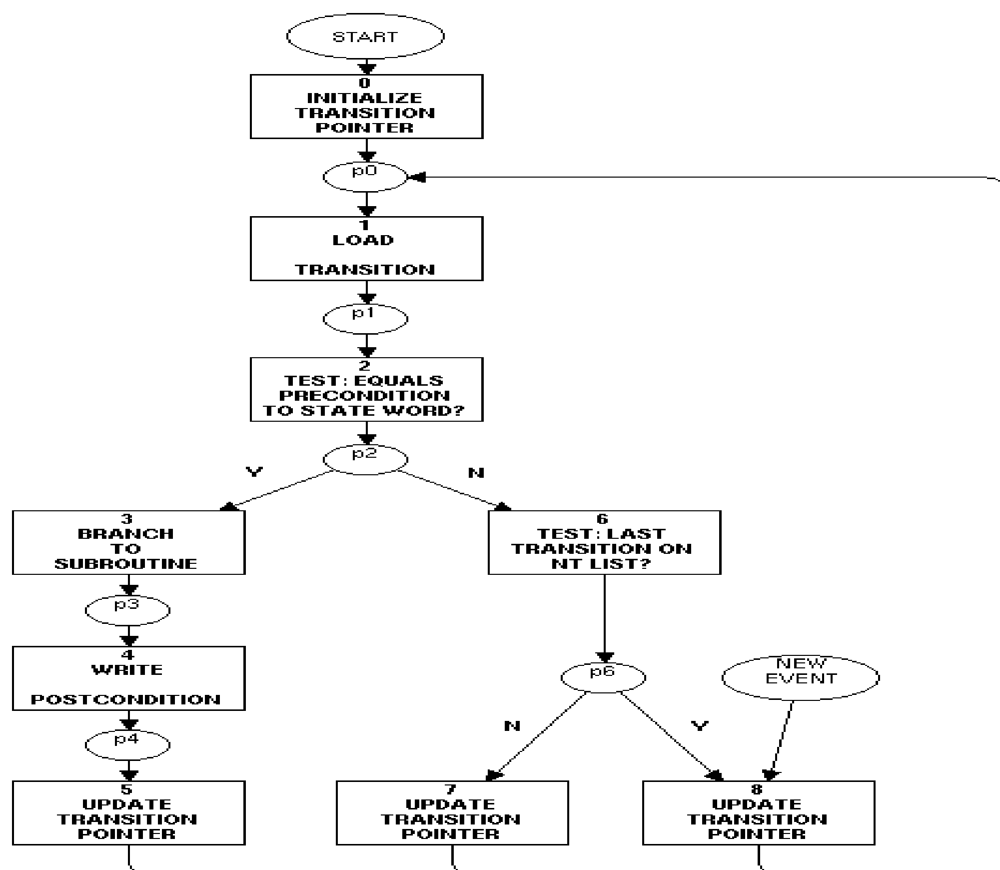

Figure 5. Execution Cycle of the ALPiNe Processor.

The architecture and organization of the CE is that of a simple processor core, as outlined in [HP90]. It is a processor with typical Fetch, Decode and Execute stages and a small instruction set (RISC), extended with bit manipulation instructions. The overall organization is based on the implementation described in [MG95]. Additions include the interface to the PNDU with ability to read Precondition, Subroutine Address and Postcondition, and to write Postcondition, and to assert control signals Start and Finished. In the execution cycle of Figure 5, the CE task is embedded into the Step 3 - Branch to

Subroutine. This is where the CE receives the Address of Subroutine from the PNDU along with the Start prompt, executes the code contained in the subroutine, signals Finished to the PNDU, and may return a modified Postcondition.

Railroad Crossing Example

A simple example is used to demonstrate the full translation of the CPN into the proper ALPiNe encoding. Figure 6 shows four sets of waveforms that represent operation of the railway crossing controller [CP96]. There are two sensors "r" and "l" located on the right and on the left of the crossing respectively. Long and short trains may come from either direction. For example, a long train coming from the right would produce a waveform shown in Figure 6 a), whereas a short locomotive coming from the left is shown in Figure 6 d). Based upon the sensor detection of the train the output variable "z" is set high in order to activate the closure of the railway crossing.

The control algorithms developed for this problem is given by the CPN of Figure 7. There are three main places (a, b and c) and the corresponding transitions T1, T2 and T3. The execution starts from the extra place START which introduces a machine token "m". This token is used to symbolically follow the flow of the execution of the net. The extra transition T0, which is always enabled at the start of the execution, is used to set or clear appropriate binary variables (Semaphores) before the execution of the main algorithm. The global fusion place FLAGS introduces tokens received from the environment CPN into the control CPN.

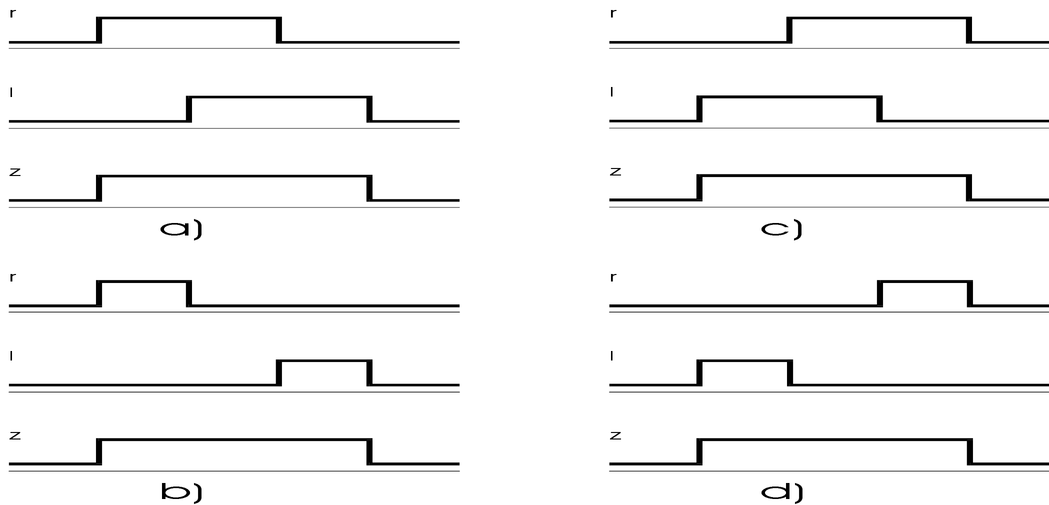

Figure 6. Signal Waveforms for the Control of the Railway Crossing.

A Coloured Petri Net of the environment is shown in Figure 8. The given net describes the detection of a long train coming from the right. The first transition initializes the net by producing (r,l) vector equal to logical (0,0). Then there are four events corresponding to the four transitions at times 20, 40, 70 and 90 (arbitrary time units). The token values are derived from the sensor waveforms and are deposited into the global fusion place FLAGS. The very instance they are deposited, they are picked up by the control CPN and processed. Note that this example demonstrates one way communication from the environment to the control CPN. However, two way communication can be easily achieved by introducing another global fusion place SEMAPHORES which would transfer status indicator tokens of the control CPN to the environment CPN.

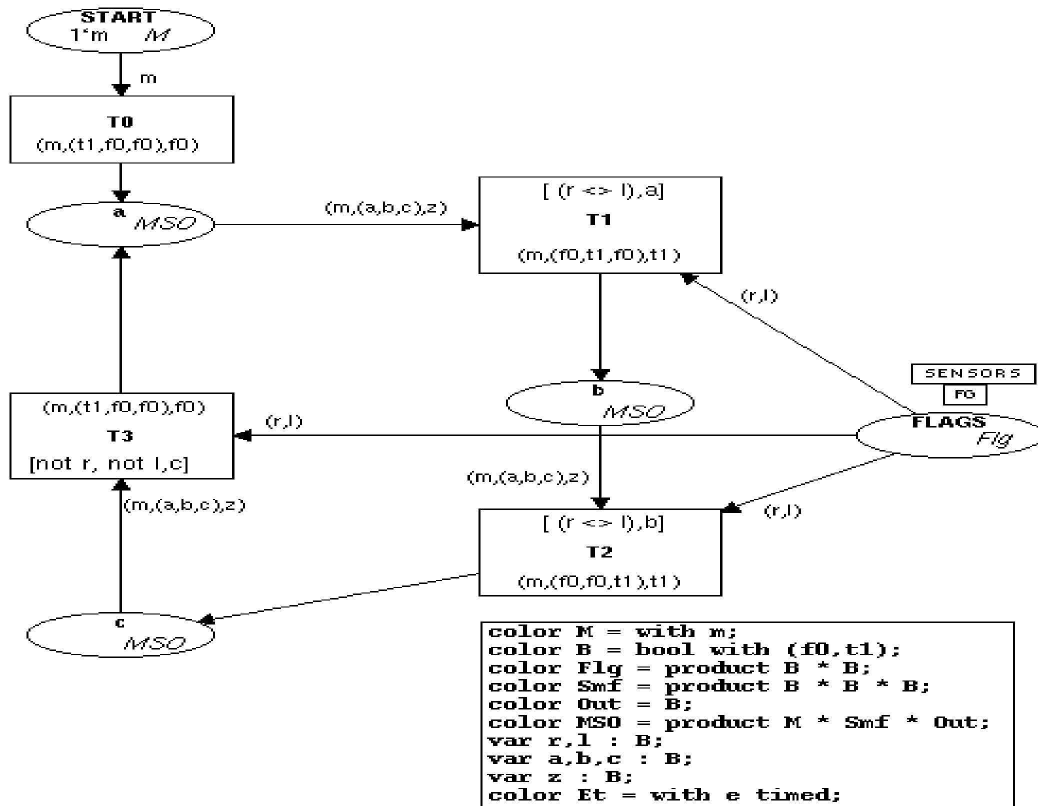

Figure 7. The CPN of the Control Algorithm for the Railway Crossing.

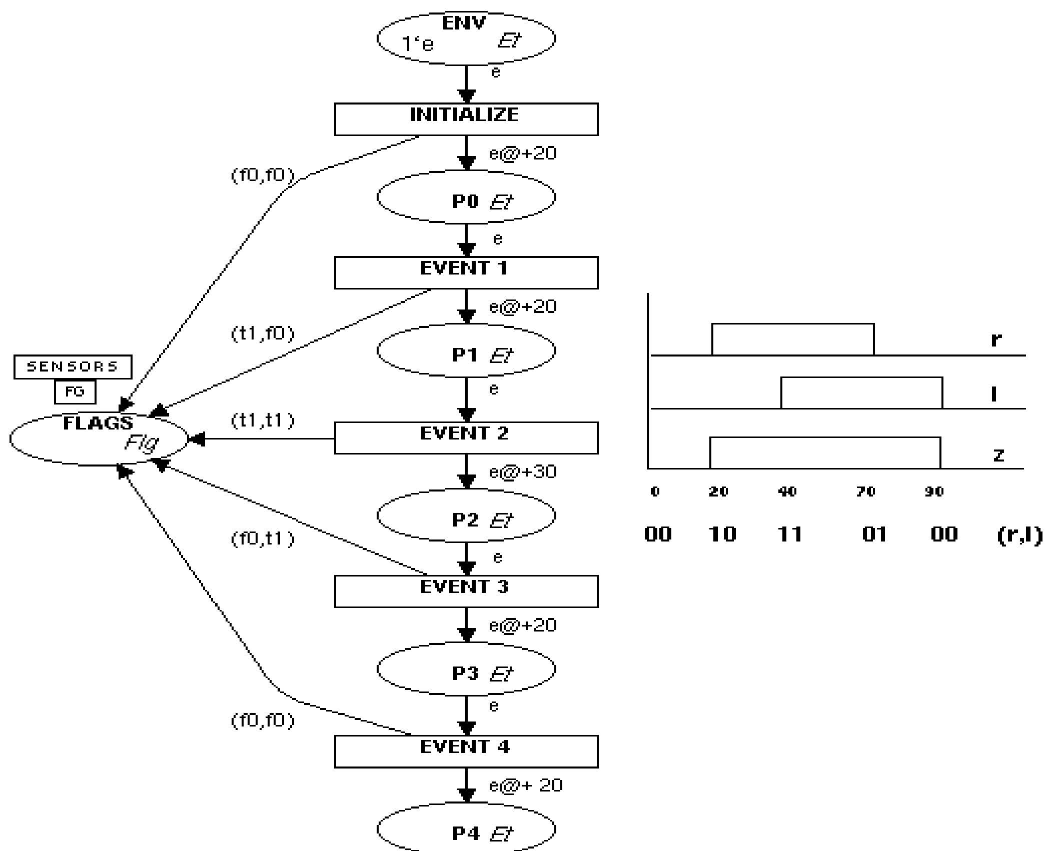

Figure 8. The CPN of the Environment for the Railway Crossing Problem.

The global declaration node contains a description of the variables used in this problem. Color M is assigned to the simple machine token *m* which flows from place to place and indicates in what place (or state) the algorithm is at a given time. Color B defines Boolean type with *false* assigned to an alphanumeric *f0* and *true* to *t1*. Each place on the main program is of the MSO type, which is a tupple of Machine, Semaphore and Output colors. Semaphore variables *a*, *b* and *c* are used to indicate the state, while the Output variable *z* sets the proper output value. In the hardware sense, there is no difference between the Semaphore and Output variables because both are the elements of the Postcondition Word register. However, for programming clarity it helps to separate them. The Flg color is a tupple of the Flags variables *r* and *l*.

The control CPN is an executable net. It allows to verify the correctness of the control algorithm and to perform formal analysis, for example, by constructing an occurrence graph. Note that the Guards are placed inside the transition on the side of the input arc to hint that they form a Precondition. Also, the output arc inscriptions are placed inside the box to signal that they are, in fact, Postconditions. The verification is done by processing the tokens which get introduced to the FLAGS place during the execution of the environment CPN. For example, the sequence for the long train coming from the right is a product (r,l) with binary values {(t1,f0),(t1,t1),(f0,t1),(f0,f0)} which represent four distinct events corresponding to the waveform of the environment CPN of Figure 8.

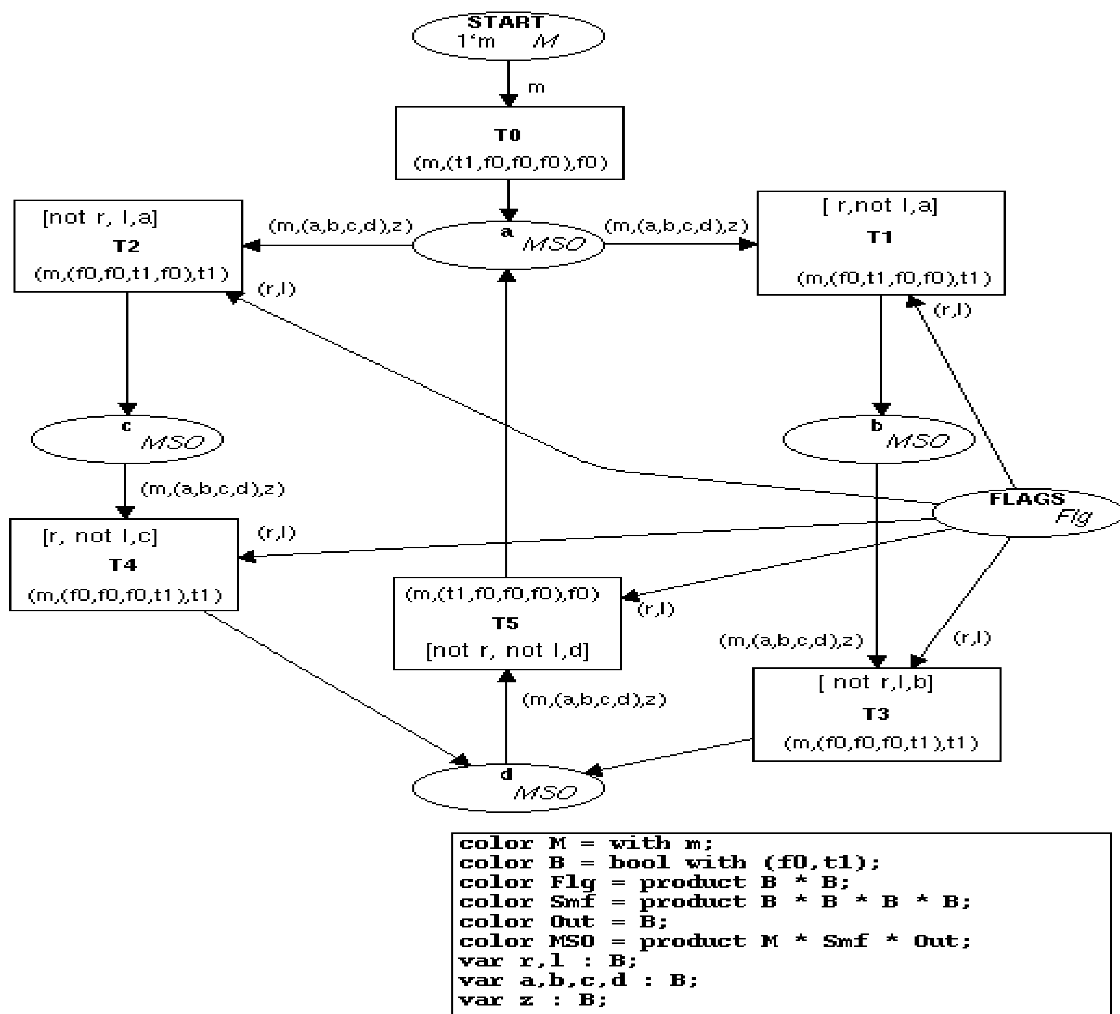

Figure 9. Intermediate-Level CPN for the Railroad Crossing Problem.

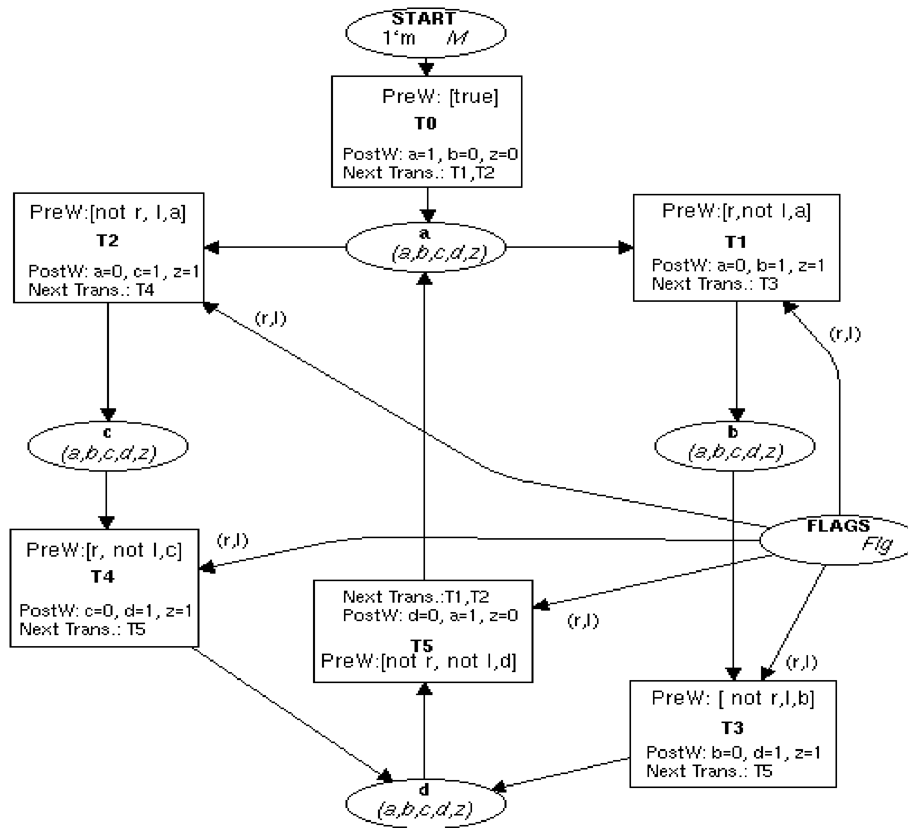

Figure 10. Encoding of the CPN into the ALPiNe Format.

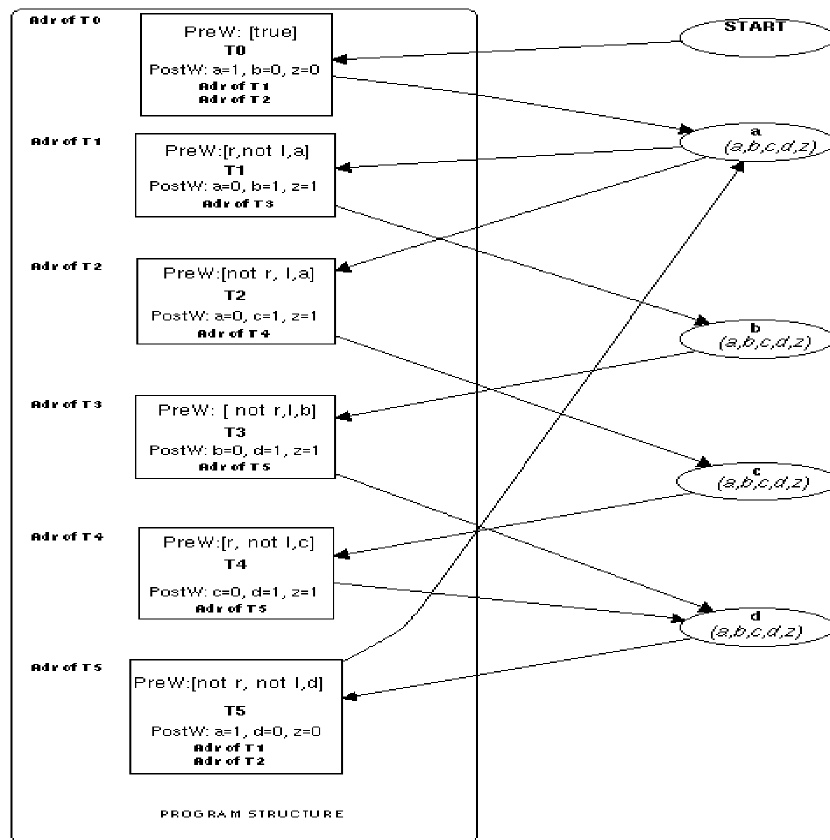

Figure 11. Final Step of Encoding the CPN into the ALPiNe Format.

Taking into account that presently ALPiNe is capable of testing only Boolean-AND operation of the Guard, the CP Net of Figure 7 could not be directly encoded into the ALPiNe machine code. This is observed from the Guard of T1, G = [(r not equal l) and a]. The Guard of T2 is also complex. Only the Guard of T3 is of the required Boolean-AND format. Thus, this CP net is "somewhat" high-level CPN. Enabling more complex Guards to be tested would solve this problem. However, at this point ALPiNe does not support complex Guards. Therefore, the CP net must be transformed into the intermediate-level CP net with more transitions to explicitly test complex Guards. This lower level CP net is shown in Figure 9, where Guards of each transition are of the required format. This CP net could be encoded into the ALPiNe machine code. The process of encoding is given by Figures 10 and 11.

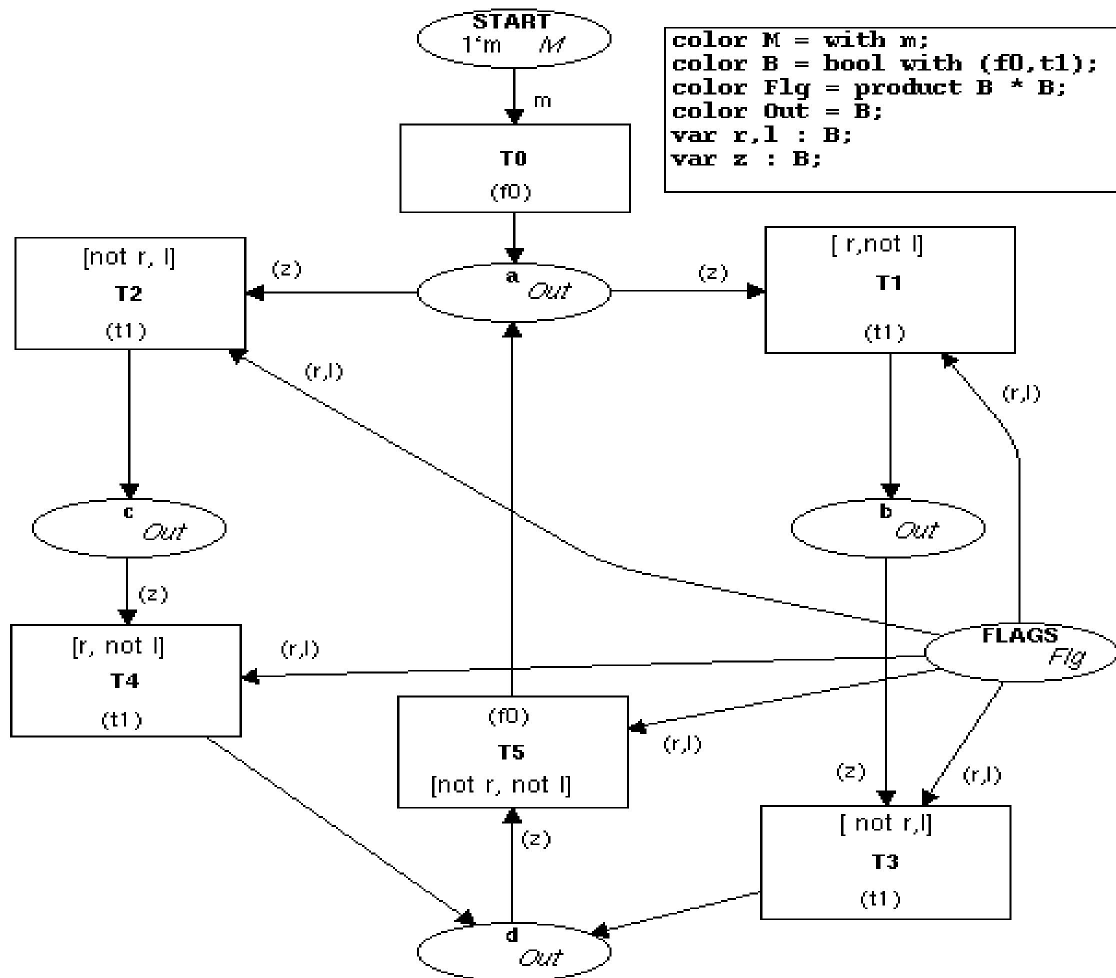

Figure 12. Simplified Intermediate-Level CPN.

The encoding proceeds as follows. Information provided by the Input Arc Inscription and the Guard is combined to form a Precondition Word, PreW. The Output Arc Inscription is converted into a Postcondition Word, PostW. In this example there is no subroutine associated with transitions, only the output variable is set or cleared. Subsequently, the corresponding addresses of the Next Transitions are appended to each transition. For example, the Next Transition List section of T0 will contain the addresses of T1 and T2. This process is shown in Figure 11. At this point, places and arcs lose their meaning as only transitions constitute the body of the program. This program can be executed on ALPiNe according to the algorithm described in Figure 5.

Careful analysis of the above coding scheme will show that encoding the Semaphore (current state) information is superfluous and could be omitted because for the correct solution of this control problem it is sufficient to test Flags tokens only. In the example presented the CPN of the environment does not need the information about the present state of the control algorithm. Therefore, the control CPN of Figure 9 could be simplified to produce a net with simpler arc inscriptions as shown in Figure 12. However, for some more complicated processes it may be crucial to explicitly know the state of the control. In such cases, the state could be encoded as already explained.

Advantages of ALPiNe

ALPiNe is an interesting processor from both academic and commercial perspectives. First of all, it is a dedicated platform optimized for fast implementation of control algorithms developed using High-Level Petri Nets. It is a programmable device, and thus it is not limited to one specific application, rather it is suitable for the broad class of reactive systems. ALPiNe has two layers of hardware: one for deciding when and which computations should take place, and another one for performing those computations. This results in an efficient hardware utilization, and promises power efficiency depending on the final implementation.

One aspect not addressed at this time is the absence of interrupts. This would make it impossible to use ALPiNe in hard real-time applications, excluding some special cases. However, this issue will be addressed at the subsequent versions of the processor, with inclusion of a timer. This may be particularly beneficial for implementing Time-Extended CPNs.

Implementation Strategy

ALPiNe processor described above is currently being implemented using the VHSIC Hardware Description Language (VHDL). The first version of ALPiNe has a global clock for both PNDU and CE modules. There are plans to experiment with an asynchronous (self-timed) realization of the PNDU, which naturally achieves the event-driven nature of operation, and a gated-clock CE. The ultimate goal is to have a complete self-timed version of ALPiNe and to analyze its performance against its synchronous counterpart. In addition, work is in progress on overcoming the FSM restriction and allowing more complex Guards.

Conclusions

ALPiNe is a dedicated processor whose execution algorithm is based on the event-driven property of Petri Nets. Its unique architecture is geared to ease the development of complex reactive systems, such as embedded control systems, by providing special hardware to handle High-Level Petri Net algorithm and by incorporating external events into the program execution flow. This paper describes a detailed procedure for software development and presents an overview of the ALPiNe architecture. The successful completion of the ALPiNe project is expected to increase control system designer's awareness of the modelling power of Petri Nets and open new implementation horizons. Another anticipated positive side effect is the encouragement of new developments in CPN based software tools that would automate the transformation and compilation of the High-Level Petri Nets into the ALPiNe machine code.

Acknowledgements

The authors wish to thank Soren Christensen and Kurt Jensen for their helpful comments regarding this work. This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada.

References

[AW91] W. M. P. van der Aalst, A. W. Waltmans, "Modelling Logistic Systems with EXPECT", In *Dynamic Modelling of Information Systems*, Eds. H. G. Sol and K. M. van Hee, Elsevier Science Publishers, Amsterdam, pp. 269-288, 1991.

[BM91] A. D. Ben, I. Miron, "Concurrent Modelling and Simulation of Reactive Manufacturing Systems Using Petri Nets", *Computers and Industrial Engineering*, Vol.20, No. 1, pp.45-54, 1991.

[CP96] C. Piguet, "Low-Power Design of Finite State Machines", in *Proceedings of PATMOS'96*, Eds. W. Nebel and B. Ricco, Pitagora Editrice Bologna, Italy, pp. 25-34, 1996.

[GD94] Giovanni De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill Inc., Chapter 11, 1994.

[HP85] D. Harel, A. Pnueli, "On the Development of Reactive Systems", in NATO ASI Series, Vol. 13, *Logics and Models of Concurrent Systems*, K.R. Apt, Editor, Springer-Verlag Berlin Heidelberg, pp. 447-498, 1985.

[HP90] J. H. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers, 1990.

[JC97] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, "Petrify: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers", *IEICE Transactions on Information and Systems*, E80-D(3):315-325, 1997.

[JP81] James L. Peterson, *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, Chapter 3, 1981.

[KJ96] Kurt Jensen, *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, Volume 1, EATCS Series Monographs in Theoretical Computer Science, Springer-Verlag, Chapter 1&2 , 1996.

[MG95] Martin Gumm, *VHDL - Modelling and Synthesis of the DLXS RISC Processor*, University of Stuttgart, Institute of Parallel and Distributed High-Performance Systems, Dept. of Integrated Systems Engineering, December 1995.

[MP96] Olivier Maffeis, Axel Poigne, "Synchronous Automata for Reactive, Real-Time or Embedded Systems", Technical Report No. 967, German National Research Centre for Information Technology (GMD), January 1996.

[MS93] Kurt Jensen *et al*, *Design/CPN Reference Manual*, Meta Software and Computer Science Department, University of Aarhus, Denmark, 1993. On-line version: http://www.daimi.aau.dk/designCPN/.

[RS96] J. L. Rasmussen, M. Singh, "Designing a Security System by Means of Coloured Petri Nets", *Lecture Notes in Computer Science*, Vol. 1091, Springer-Verlag, pp. 400-419, 1996.

[SK97] A. Semenov, A. M. Koelmans, L. Lloyd, A. Yakovlev, "Designing an Asynchronous Processor Using Petri Nets", *IEEE Micro*, Vol. 17, No. 2, pp. 54-63, March/April 1997.

[TB82] P. C. Treleaven, D. R. Brownbridge, R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture", in *Dataflow and Reduction Architectures*, Editor S. S. Thakkar, Computer Society Press of the IEEE, pp.4-54, 1987.

[TH92] Kenneth Hinz, Daniel Tabak, *Microcontrollers - Architecture, Implementation and Programming*, McGraw-Hill Inc., Chapter 3, 1992.

[TM89] Tadao Murata, "Petri Nets: Properties, Analysis and Applications", *Proceedings of the IEEE*, 77(4), pp.541-580, 1989.

[TT97] J. Teich, L. Thiele, S. Sriram, M. Martin, "Performance Analysis and Optimization of Mixed Asynchronous Synchronous Systems", *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 16, No. 5, pp. 473-484, May 1997.

[WA94] W. M. P. van der Aalst, "Putting High-Level Petri Nets to Work in Industry", *Computers in Industry*, 25(1):45-54, 1994.