PERGAMON

Control Engineering Practice 6 (1998) 889–895

Hardware implementation of real-time Petri-net-based controllers

Naehyuck Chang[a,*], Wook Hyun Kwon[b], Jaehyun Park[c]

[a]*Department of Computer Engineering, Seoul National Univ., Korea*
[b]*School of Electrical Engineering, Seoul National Univ., Korea*
[c]*Department of Industrial Automation Engineering., Inha Univ., Korea*

Received 13 March 1998

Abstract

This paper presents an implementation method for high-speed Petri-net-based controllers for time-critical control with parallelism, based on look-up tables. Unlike microprocessor-based software implementations, the method is based on hardware, and offers enough speed to control fast plants at low cost. The method is easily able to accommodate more complex control problems than hardwired implementations. The data structure and execution module are composed of common memory devices and simple logic gates. A modified matrix-based data structure, named the bi-column matrix-based data structure, is suggested. It has a regular form but retains the efficiency of list-based data structures. A sub-class of Petri nets is defined in order to fit Petri nets to the bi-column matrix-based data structure, and conversion to Petri nets of the sub-class is described. The matrix framework enables pipelined execution, which increases the performance in an efficient way. An event-driven evolution scheme is proposed as well, to enhance response time. This method is demonstrated by an implementation using a Xilinx 4000 series LCA. An example illustrates that the Petri-net-based controller is useful in a high-speed supervisor-control problem.

Keywords: Petri nets; parallel processing; hardware; implementation; real-time systems

1. Introduction

There are large numbers of *event controllers* in automation systems. Since events are asynchronous and instantaneous, event control is usually *time-critical*. Events are often independent, thus involving *parallelism*. *Petri nets* have better properties for designing parallel controllers than other models such as finite state machines (FSM) (Mutrata, 1989). In comparison with FSMs, however, Petri nets are difficult to implement due to their complex firing rules. A Petri-net-based controller should offer *flexibility*, *reusability*, less *complexity* and low *cost*, as well as sufficient *speed* and *functionality*. This paper focuses on an implementation method for Petri-net-based controllers, as event controllers, taking the above requirements into account.

In the current literature, numerous methods have been proposed for the implementation of Petri-net-based controllers. *Software* and *hardware* implementations have been suggested as the main categories.

Software implementations (Silva and Velilla, 1982, Valette et al., 1983, Colom et al., 1986) have flexibility and can manage versatile constraints, but are characterized by slow evolution speed. Decentralized implementation with net decomposition (Colom et al., 1986), utilization of a *linear enable function* and selection of a good set of *representative places* have been proposed, to cope with the slow speed and to enhance efficiency (Briz et al., 1994). Hendry (1994) optimizes the CPU's clock cycles, and Stefano and Mirabella (1991) enhance the speed by the use of a DSP (digital signal processor) *scanning processor*. However, due to inherent limitations, software implementations may not support some applications, which need enough speed to guarantee critical time bounds.

While software implementation of Petri-net-based controllers can be applied to comparatively slow targets such as manufacturing systems, hardware implementation of Petri-net based controllers is introduced for high-speed parallel controllers interacting with digital hardware systems.

Among the hardware implementations, hardwired implementations (Auguin et al., 1980; Adamski, 1991;

* Corresponding author. E-mail: naehyuck@snu.ac.kr

0967-0661/98/$ – See front matter
PII: S0967-0661(98)00076-8

Pardey and Bolton, 1991; Biliński et al., 1994, 1995; Kozlowski et al., 1995; Chang et al., 1996) offer higher speed because the net structures are formed from logic gates. CAD systems can be used to reduce complexity and design cost for various place-encoding algorithms (Biliński et al., 1994, 1995; Kozlowski et al., 1995). However, complexity and cost are still not satisfactory for some applications that need high functionality. Additional functionality directly increases the complexity and cost, more so than in software implementations. This implementation method sometimes has insufficient flexibility for field engineers, in spite of the support of CAD methods.

This paper presents a new event controller based on look-up tables in hardware. The design is a compromise between the software method and the hardwired method. Look-up-table-based implementations separate the storage of the data structure from the execution, and thus are flexible and reusable. The data structure is the same as for software implementations (Murakoshi et al., 1991). While list-based data structures have advantages in both cost and speed in software implementations (Silva and Velilla, 1982), it is difficult to adopt list-based data structures for fast hardware implementation, because operations on lists are irregular and complex. Due to their simple, regular and parallel structures, matrix-based data structures have potentially high performance in hardware implementations.

Since basic matrix-based data structures suffer from the memory explosion problem, an intensive memory-reduction technique is required. Therefore, a modified data structure, named the *bi-column matrix-based data structure* is suggested. This structure combines the advantages of the matrix-based data structure and the list-based one, in order to maintain enough speed to guarantee critical time bounds with realizable cost. In addition, it enables pipelined execution, which enhances speed efficiently. This paper suggests an event-driven evolution scheme to improve response time as well.

Section 2 introduces the *BPSPN* (*bi-arc priority synchronous Petri net*) and the bi-column matrix-based method. The conversion from Petri nets to BPSPNs is outlined, and a hardware structure for a BPSPN-based controller is presented as well. Section 3 shows a pipelined architecture for the BPSPN-based controller. Section 4 introduces an event-driven evolution scheme. Section 5 presents a hardware implementation and an automatic design support program. Section 6 demonstrates an application of the high-speed BPSPN-based controller. Conclusions are discussed in Section 7.

2. Bi-arc priority synchronous Petri net (BPSPN) and BPSPN controller

This paper describes Petri nets according to the following notation. A Petri net $\mathscr{C}$ is a five-tuple (P, T, IN, OUT, μ_0). A *Synchronous Petri net* $\mathscr{S}$ is a five-tuple $(\mathscr{C}, \Sigma, \Gamma, \Pi, \xi)$; $\mathscr{C}$ is a Petri net; $\Sigma = (\sigma_1, \ldots, \sigma_l)$ is a set of Boolean variables (predicates); $\Gamma: T \to \Sigma$ is a function representing a set of associated variables with transitions; $\Pi: P \to \Sigma$ is a function representing a set of associated variables with places; ξ is a system clock. External variables are sampled by the *positive edge* of the system clock (Chang et al. 1996).

In an asynchronous interpretation, the hypothesis that only one event occurs at a time resolves effective conflicts through the use of transition predicates. This hypothesis is usual in the implementation of Petri nets. However, it is sometimes too strong, especially in synchronous implementations; in most cases, clock-driven logic performs synchronous operations. This paper resolves effective conflicts when more than one active variable is sampled in the same clock cycle, by use of *priority Petri nets*. Priority Petri nets can resolve effective conflicts without the above hypothesis. This paper uses *bi-arc priority synchronous Petri nets* (*BPSPN*) for hardware implementation, without requiring the assumption that only one event occurs at a time. The BPSPN is defined as follows:

Definition 1. A *BPSPN* $\mathscr{B}$ is a sub-class priority synchronous Petri net, where $\forall t \in T$ and $\forall p \in P$ $|{}^\bullet t| \leqslant 2, |t^\bullet| \leqslant 2,$ and $|p^\bullet| \leqslant 2$.

Remark 1. For BPSPNs, the output transition connected with the left-hand arc has priority over the transition connected with the right-hand arc.

The *bi-arc transform* converts a priority synchronous Petri net to a BPSPN; it consists of the *bi-arc fork*, *bi-arc join* and *bi-arc decision transforms*. By *simple reduction rules* (Mutrata, 1989), the *bi-arc joins* and the *bi-arc forks*, as shown in Figs. 1(a) and (b), can be performed while preserving other properties such as safeness, liveness, boundedness, reachability and coverability. These transforms are two of the possible types of simple reductions. The *bi-arc decision* emulates a prioritized decision, as shown in Fig. 1(c), where λ is an *always-occurrent* (David and Alla, 1992) event. In a bi-arc decision, other transition predicates have priority over λ, because λ is associated with a transition connected to the right-hand arc.

This paper describes an implementation method for a safe BPSPN-based controller with binary marking. The proposed implementation method is named the *bi-column matrix-based method*. The net structure is stored in look-up tables: $\mathsf{O}(p)$, $\mathrm{I}(t)$, $\mathsf{O}(t)$ and $\mathsf{m}(p)$; $\mathsf{O}(p)$ is a $|P| \times 2$ memory table that contains $p^\bullet$; $\mathrm{P}(t)$ is a $|T| \times 4$ memory table that contains ${}^\bullet t$ and $t^\bullet$; $\mathsf{m}(p)$ is a $|P| \times 1$ memory table that contains $\mu(p)$. $\mathsf{R}(p)$ is a FIFO queue that contains *representative places* (Silva and Velilla 1982). $\Sigma(t)$ is a $|T| \times 2$ memory table that contains Σ. $\Pi(p)$ is a $|P| \times 1$ memory table that contains Π.

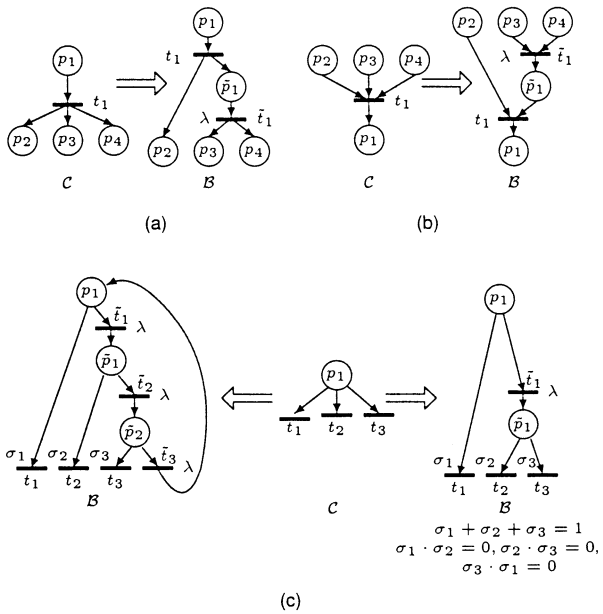

Fig. 1. Bi-arc transforms. (a) Bi-arc fork transform; (b) Bi-arc join transform; (c) Bi-arc decision transform.

The places and transitions may have level (status) or pulse (strobe) I/O variables (Chang et al., 1996). For bi-arc decisions, the controller has to resolve an effective conflict with two external variables. If both variables are active in the same clock cycle, a multiplexer selects the one that has priority over the other.

When a transition t_i is fired, its output places become new representative places. A multiplexer selects the output places in turn, to insert into $\mathsf{R}(p)$ sequentially. On the other hand, t_i is not fired when $\Sigma(t_i) \neq 1$, there exists ${}^\bullet t_i$ where $\mu({}^\bullet t_i) = 0$ (the receptivity is false) or both. Among these situations, t_i should be tested again by the same representative place, when $\Sigma(t_i) \neq 1$ but $\forall {}^\bullet t_i, \mu({}^\bullet t_i) \neq 0$ (the receptivity is true). In other words, when a transition is not fired even though the receptivity is true, its representative place is inserted again into $\mathsf{R}(p)$ (retry). Successive access to the tables leads to the evolution cycle: $\mathsf{R}(p) \rightarrow \mathsf{O}(p) \rightarrow \Sigma(t) \rightarrow \mathsf{P}(t) \rightarrow \mathsf{m}(p)$.

3. Pipeline architecture for a BPSPN-based controller

With common memory devices, the number of accesses for each device is as follows; $\mathsf{R}(p)$ is 1-read/2-writes, $\mathsf{O}(p)$ is 1-read, $\Sigma(t)$ is 2-reads, $\mathsf{P}(t)$ is 1-read, and $\mathsf{m}(p)$ is 4-reads/4-writes. The sequence of memory access can be broken down into four execution stages, as shown in Fig. 2(a). Stage 0, the *search stage*, picks out transitions that are the candidates to fire. Stage 0 also stores new representative places from Stage 3. Stage 1, the *synchronization stage*, checks the receptivities of the transitions and resolves effective conflicts. Stage 1 also obtains all the input and output places of any transition expected to fire. Stage 2, the *test stage*, determines whether or not to fire a transition, using markings of input places and transition predicates. Stage 3, the *update stage*, updates all the markings of input and output places of the transitions that have fired.

Each stage consists of four phases. The ϕ_2 phase in Stage 1 is the margin for the external signal access. When the number of arcs on a node is greater than two (the BPSPN limits the number to two), the balance of the number of accesses in each stage is destroyed, because the number of markings associated with a transition is twice the number of arcs; the BPSPN minimizes the imbalance of the stages.

4. Event-driven evolution scheme

Stages 0 and 1 use independent resources, but Stages 2 and 3 both use $\mathsf{m}(p)$ despite sequential access. However, $\mathsf{m}(p)$ requires less memory than other devices, as each entity in $\mathsf{m}(p)$ is only 1 bit. In contrast, the content of other devices is equal to or greater than $\log_2|T|$ or $\log_2|P|$. Therefore, it is not difficult to have $\mathsf{m}(p)$

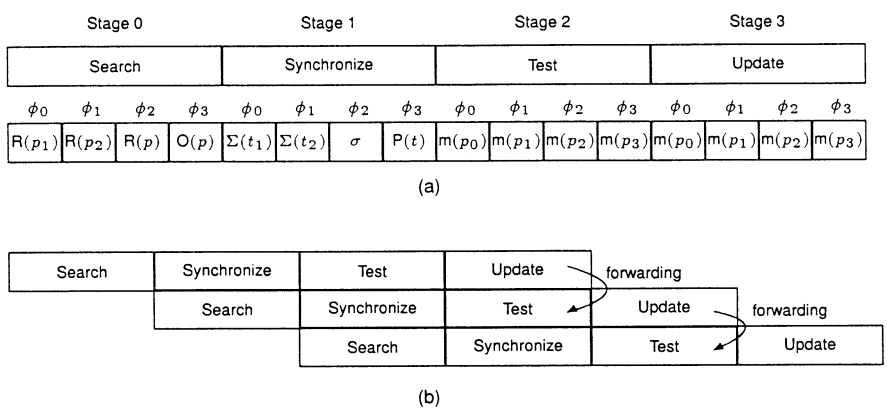

Fig. 2. Pipeline stages of the bi-arc matrix hardware. (a) Execution statges; (b) Pipelined execution.

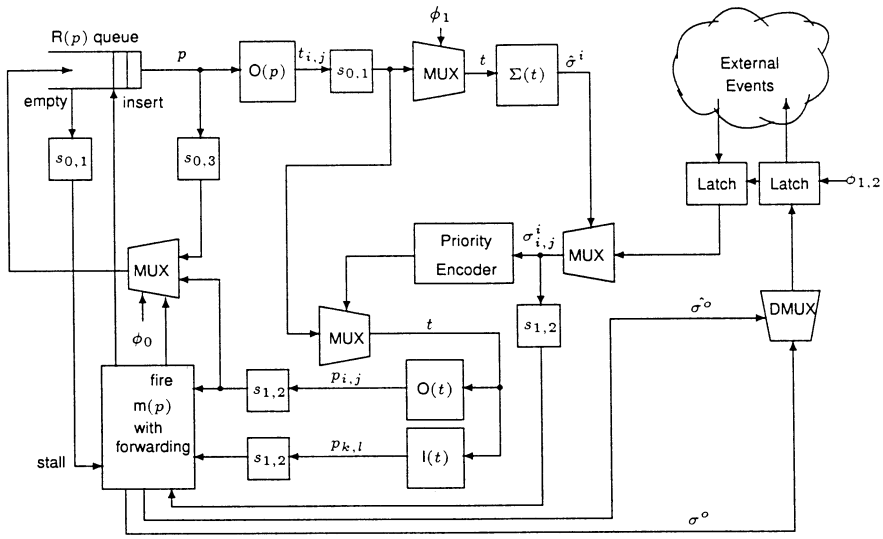

Fig. 3. Pipelined architecture of the bi-arc matrix hardware. $\hat{\bullet}$ is the ID of $\bullet$. σ^o and σ^i are transition predicate inputs and transition predicate outputs, respectively.

equipped with separate *data I/O ports*, and separate *cell decoders* in order to perform one read and one write per phase. This allows the construction of overlapped (pipelined) execution, as shown in Fig. 2(b). Pipelined execution increases performance by as many times as the pipeline depth: four. The architecture for pipelined execution is shown in Fig. 3. The boxes with the label $s_{i,j}$ are *pipeline registers* that isolate Stage i from Stage j. Since Stages 2 and 3 overlap, a *read-before-write hazard* occurs at $\mathsf{m}(p)$ when t_i in Stage 3 is fired and ${}^\bullet\hat{t}_i \cap \hat{t}_j{}^\bullet \neq \emptyset$ where t_j is in Stage 2. To avoid the hazard, *forwarding* of $\mathsf{m}(p)$ should be used. The forwarding logic consists of an *address cache* and a *write-back buffer*. When $\mathsf{R}(p)$ is empty at Stage 0, a *null* place pointer is returned, and induces a *pipeline stall*. The pointer inhibits firing, updating $\mathsf{m}(p)$ and disabling the address cache. The null place pointer is also used when a transition has only one input place or one output place, or a place has only one output transition. A very well-organized structure is required to build a pipelined architecture, and BPSPN allows this, thanks to its matrix framework.

Sustained speed is mainly determined by response time in time-critical systems. Response time is defined as the delay time from an input event to its corresponding output event. For a given clock frequency, the evolution order is an important factor in determining the response time. Evolution of the transition associated with a new event prior to other events enhances the response time. In the proposed architecture shown in Section 3, the evolution order is determined by representative places; the evolution order has nothing to do with events. *Event-driven* evolution makes it possible to evolve a transition associated with a new event, prior to others.

When all transitions have external event predicates, the BPSPN-based controller no longer requires $\mathsf{R}(p)$.

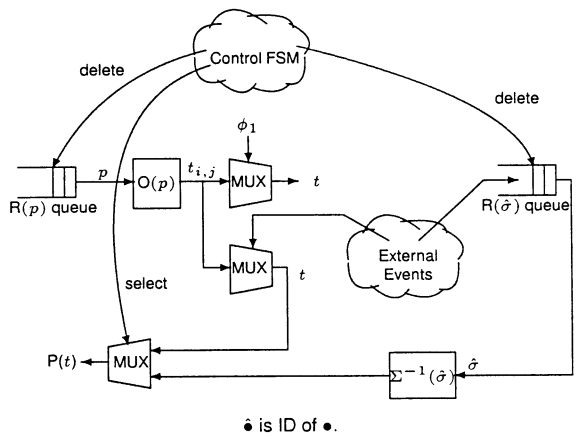

Fig. 4. Modified data-path for event-driven evolution.

Rather, the BPSPN-based controller needs an external event queue, $\mathsf{R}(\hat{\sigma})$ ($\hat{\sigma}$ is the ID of σ). When events occur, they are inserted into $\mathsf{R}(\hat{\sigma})$. However, to support λ as a transition predicate, both $\mathsf{R}(p)$ and $\mathsf{R}(\hat{\sigma})$ are required. Fig. 4 shows the modified structure that has both $\mathsf{R}(p)$ and $\mathsf{R}(\hat{\sigma})$.

When one of $\mathsf{R}(p)$ and $\mathsf{R}(\hat{\sigma})$ is empty, the non-empty queue is activated. After firing a transition, its output places are inserted into $\mathsf{R}(p)$. Initially both queues are empty, and then $\mathsf{R}(\hat{\sigma})$ is activated when an event occurs. $\mathsf{R}(\hat{\sigma})$ is deactivated and $\mathsf{R}(p)$ is activated after firing, and finally both become deactivated. These operations are the response actions of the event. If another event occurs before completion of the previous response action, the BPSPN-based controller evolves both events in sequence or simultaneously.

In Fig. 5, (a) shows sequential evolution, and (b) does simultaneous evolution. Because sequential evolution

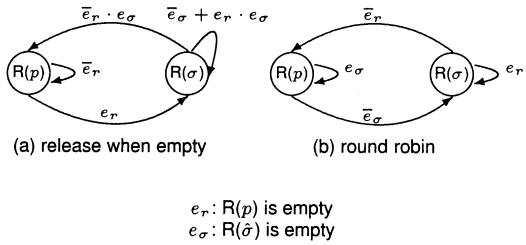

Fig. 5. Control FSM of $\mathsf{R}(p)$ and $\mathsf{R}(\hat{\sigma})$.

may cause livelock, this paper adopts simultaneous evolution with a round-robin method.

5. Implementation of a prototype and an automatic design support program

The BPSPN-based controller is demonstrated by prototyping. For easy verification, the prototype is designed to handle 32 places and 32 transitions; the word length of place and transition ID is five bits. The use of a 10-bit ID allows up to 1024 places and 1024 transitions. A Xilinx 4000 series LCA is used for the prototype, and its internal RAM function is used in place of common memory devices, in order to allow rapid prototyping.

An automatic design support program generates the contents of the look-up tables, thus giving user-flexibility. The BPSPN is drawn using a commercial schematic entry program: OrCAD. OrCAD is intended to draw logic symbols. This paper adds a BPSPN library to draw BPSPNs using OrCAD. OrCAD generates net-lists in a standard format. The conversion program produces the contents of the memory tables from the standard net-list.

6. Example

This section demonstrates a high-speed BPSPN-based controller for control problems. Consider the system in Fig. 6, composed of four continuous controllers and a supervisor controller. The plant is a MIMO system, and four PID controllers are used. As one can see from real-world plants, dependable PID controllers are often composed of analog components such as OP Amps. It can be assumed without loss of generality, that each controller has three gain modes, as shown in Fig. 7, for fault-tolerant control. A simple case of fault tolerant control is straightforward feedback gain scheduling according to the plant output (Warwick 1991).

Once a fault occurs, the detector generates an event f_i to notify the supervisor controller, which adjusts the controller gain. The adjustment of a controller gain may require reconfiguration of other controller gains as well, because the plant is originally a MIMO system. The adjustment dependency can be described by Boolean

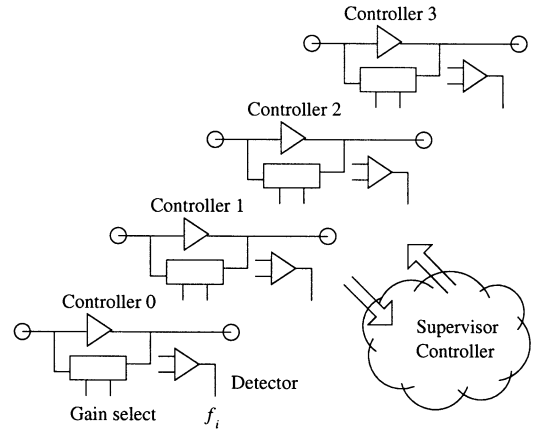

Fig. 6. Multi-mode SISO controllers for a MIMO plant.

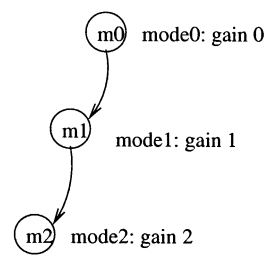

Fig. 7. State diagram of the multi-mode PID controller.

matrix equations as follows

$$M_0 : \begin{pmatrix} \bar{f}_1 & 0 & 0 & 0 \\ 0 & \bar{f}_2 & 0 & 0 \\ 0 & 0 & \bar{f}_3 & 0 \\ 0 & 0 & 0 & \bar{f}_4 \end{pmatrix} M_0$$

$$M_1 : \begin{pmatrix} f_1 & f_2 & 0 & 0 \\ 0 & f_2 & 0 & f_4 \\ f_1 & 0 & f_3 & 0 \\ 0 & f_2 & 0 & f_4 \end{pmatrix} M_0 + \begin{pmatrix} \bar{f}_5 & 0 & f_7 & 0 \\ f_5 & \bar{f}_6 & f_7 & 0 \\ f_5 & f_6 & \bar{f}_7 & 0 \\ 0 & 0 & f_7 & \bar{f}_8 \end{pmatrix} M_1$$

$$M_2 : \begin{pmatrix} f_5 & 0 & 0 & 0 \\ 0 & f_6 & 0 & 0 \\ 0 & 0 & f_7 & 0 \\ 0 & 0 & 0 & f_8 \end{pmatrix} M_1 + M_2$$

where M_0, M_1 and M_2 denote the controller modes, and $\bar{f}_i$ denotes the negated event of f_i. As the initial state, $M_0 = (1111)^T$, $M_1 = (0000)^T$ and $M_2 = (0000)^T$. When Controllers 1, 2, 3 and 4 are in $m0$, $m1$, $m2$ and $m0$, respectively, $M_0 = (1001)^T$, $M_1 = (0100)^T$ and $M_2 = (0010)^T$.

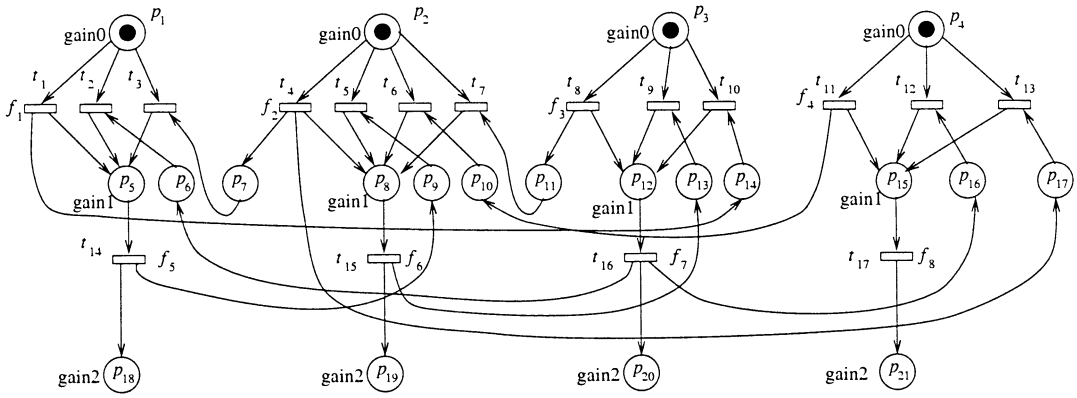

Fig. 8. BPSPN-based controller for the multi-mode SISO controllers.

The equations are state transition equations in which the left-hand side is the next state and the right-hand side is the present state. Multiplication and addition represent Boolean AND and OR, respectively. The coupling terms cause nonzero elements that are not on the diagonals of the matrices. For example, the equation shows that Controller 1 has to change its mode $m0$ to $m1$ when f_1 occurs, Controller 2 changes its mode $m0$ to $m1$, or Controller 3 changes its mode $m1$ to $m2$.

Various methods are able to describe the supervisor controller. One traditional method is to use an FSM with hardwired logic. The FSM design must start from shuffling the states of the controllers, resulting in 3^4 states. In contrast, a Petri-net description can describe parallelism, and yields a neat result, as shown in Fig. 8.

t_1, t_4, t_8, t_{11}, t_{14}, t_{15}, t_{16} and t_{17} are associated with transition predicates (pulse inputs) (Chang et al., 1996) f_1, $f_2, f_3, f_4, f_5, f_6, f_7$ and f_8, respectively. Other transitions are implicitly associated with λ. p_1, p_2, p_3 and p_4 are associated with the place predicate (level output) (Chang et al., 1996) gain0. p_5, p_8, p_{12} and p_{15} are associated with gain1, and p_{18}, p_{19}, p_{20} and p_{21} are associated with gain2.

In order to apply high-performance force controls, the PID controllers should have delays less than 100 μsec, and the supervisor controller should have delays much less than 100 μsec.

As mentioned earlier, there are three kinds of general implementation methods for Petri-net-based controllers. The hardwired method with FSM-like marking encoding utilizes flip-flops and logic gates, and the resulting delay is of the order of tens of nsec. However, it is unnecessarily fast, and the complexity grows proportionally to the size of the Petri net, while the delay increases logarithmically. The software method using commercial 16-bit microcontrollers may result in a more than 100 μsec delay. 32-bit RISC (reduced instruction set computer)-core microcontrollers may result in a tens of μsec delay; however, they are too costly for this kind of application.

Table 1
Memory requirements of the matrix-based look-up tables (bits)

	Existing matrix method	Bi-column matrix method
I(t)	$17 \times 21 \times \lceil \log_2 21 \rceil$	$31 \times 2 \times \lceil \log_2 31 \rceil$
O(t)	$17 \times 21 \times \lceil \log_2 21 \rceil$	$31 \times 2 \times \lceil \log_2 31 \rceil$
O(p)	$21 \times \lceil \log_2 17 \rceil$	$31 \times \lceil \log_2 31 \rceil$
m(p)	21	31
$\Sigma(t)$	$17 \times \lceil \log_2 8 \rceil$	$31 \times \lceil \log_2 8 \rceil$
Total	3747	899

The prototype of the BPSPN-based controller operates with a 100 nsec clock. Thus, the sustained firing time is a few μsec with scores of markings. Again, the controller consists of basic logic gates and common memory devices, and the structure is simple. In comparison with the hardwired method, the complexity grows logarithmically. Although the delay increase is proportional to the size of the Petri net, the look-up-table-based hardware is capable of managing several tens of markings in 100 μsec. This shows that the BPSPN-based controller is a suitable solution to build the supervisory controller.

When Fig. 8 is converted to a BPSPN, $|\hat{T}| = 31$ and $|\hat{P}| = 31$. Table 1 compares the memory requirements of the existing matrix-based look-up table method and the bi-column matrix-based method. It shows a reduction of more than 75%.

7. Conclusion

This paper describes an implementation method for high-speed parallel controllers based on Petri nets. The method can be classified in the category of hardware implementations, and consists of a separate data structure and execution module. It is called a *bi-column matrix-based method*, and is based on the *BPSPN* (*bi-arc priority synchronous Petri net*). The BPSPN is suggested

in order to reduce memory requirements, and to construct a simple and organized structure. The method is validated by prototyping with a Xilinx 4000 series LCA whose one evolution cycle is below 100 nsec. An *automatic design support* program has been developed, which generates the contents of the look-up tables from a graphical design entry, to give end-user flexibility. An example demonstrates that the high-speed BPSPN-based controller is useful in control problems.

Thanks to the structure of the bi-column matrix-based method, a four-stage pipelined architecture can be organized, enhancing performance. For time-critical systems, since response time is more important than average performance, this paper offers an event-driven evolution architecture. The resulting speed is fast enough to control time-critical systems.

Other advantages of the method are reusability, fast prototyping, and testability. Reusability and fast prototyping are possible because the controller is reconfigurable by changing the contents of the memory tables. Testability is ensured because the controller implements a Petri net without changing its properties. The BPSPN-based controller is suitable for improving the cost-performance for time-critical event controllers.

References

Adamski, M. (1991). Parallel controller implementation using standard PLD software. In: *W. MOORE and W. LUK*: '*FPGAs*' (*Abingdon EE & CS Books*), *edited paper from the international workshop on field programmable logic and applications*. pp. 296–304.

Auguin, M., Boeri, F., & Andre, C. (1980). Systematic method of realization of interpreted Petri nets. *Digit. Process*, *6*, 55–68.

Biliński, K, Saul, J.M., & Dagless, E.L. (1995). Efficient functional verification algorithm for petri-net-based parallel controller design. *IEE Proc.-Comput. Digit. Tech. 142*(4), 255–262.

Biliński, K, Adamski, M., Saul, J.M., & Dagless, E. L. (1994). Petri-net-based algorithm for parallel-cotroller synthesis. *IEE Proc.-Comput. Digit. Tech. 141*(6), 405–412.

Briz, J.L., Colom, J.M., & Silva, M. (1994). Simulation of Petri nets and linear enabling functions. In: *IEEE International Conference on System, Man and Cybernetics*. San Antonio, USA, pp. 210–226.

Chang, Naehyuck, Jaehyun Park & Wook Hyun Kwon (1996). FPGA-base implementation of synchronous Petri nets. In: *IEEE Proceedings of IECON*'96. Vol. 1. Taipei, Taiwan, pp. 451–456.

Colom, J.M., Silva, M., & Villarroel, J.L. (1986). On software implementation of Petri Nets and Colored Petri Nets using high level concurrent languages. In: *Proceedings of 7th European Workshop and Application and Theory of Petri nets*. pp. 207–241.

David, Rene & Hassane Alla (1992). *Petri Nets and Grafcet*. Prentice Hall Inc.. Englewood Cliffs, NJ 07632.

Hendry, D.C. (1994). Heterogeneous petri net methodlolgy for design of complex controllers. *IEE Proc. Comput. Digt. Tech. 141*(5), 293–297.

Kozlowski, T, Dagless, E.L. Saul, J.M. Adamski, M., & Szajna, J. (1995). Parallel controller synthesis using petri nets. *IEE Proc.-Comput. Digit. Tech. 142*(4), 263–271.

Murakoshi Hideki, Miki sugiyama, Guojun Ding, Tatsuya Oumi, Takashi Sekiguchi & Yasunori Dohi (1991). A high speed programmable controller based on Petri net. In: *Proceedings of IECON* '91, pp. 1966–1971.

Mutrata Tadao (1989). Petri nets: properties, analysis and applications. *Proceedings of IEEE*, *77*(4), 541–580.

Pardey, J., & Bolton, M. (1991). Logic synthesis of synchronous parallel controllers. In: *Proceedings of the IEEE International Conference on Computer Design*, pp. 454–547.

Silva, M., & Velilla, S. (1982). Programmable logic controllers and Petri Nets: A comparative study. In: *Proceedings of the Third IFAC/IFIP Symposium*, pp. 83–88.

Stefano, A. Di & Mirabella, O. (1991). A fast sequence control device based on enhanced Petri nets. *Microprocessors and Microsystems 15*(4), 179–186.

Valette, R., Courvoisier, J.M., Bigou, J.M., & Alburquerque (1983). A Petri net based programmable logic controller. In: *Proceedings of the IFIP conference on Computer Application s on Production and Engineering* (*CAPE*), pp. 103–116.

Warwick, K. (1991). A fault tolerant control scheme. In: *Failsafe Control Systems*. Chapman and Hall, pp. 47–55.