LACL technical report number 2001-04.

FIFO buffers in hot tie sauce

Franck Pommereau[1] and Christian Stehno[2]

[1] LACL, Université Paris 12,
61 avenue du Général de Gaulle,
94010 Créteil, France.
`pommereau@univ-paris12.fr`
[2] Fachbereich Informatik, Carl-von-Ossietzky-Universität Oldenburg
POBox 2503, D-26111 Oldenburg, Germany.
`Christian.Stehno@informatik.uni-oldenburg.de`

Abstract. This paper introduces a new semantics for FIFO buffers (usually called *channels*) in a parallel programming language, B(PN)^2. This semantics is given in terms of M-nets, which form an algebra of labelled high-level Petri nets. The proposed approach uses the asynchronous link operator, newly introduced in the algebra of M-nets, and repairs some drawbacks of the original M-net semantics. Channels are now fully expressible within the algebra (it was not the case), they are significantly smaller (in number of places), and they offer several other advantages.

Keywords. Parallel programming, Petri nets, process algebras, FIFO buffers, semantics.

1 Overview

B(PN)^2 [6] is a general purpose parallel programming language provided with features like parallel composition, iteration, guarded commands, communications through FIFO buffers or shared variables, procedures [9] and, more recently, real-time extensions with abortable blocks and exceptions [12, 13].

The semantics of B(PN)^2 is traditionally given in terms of Petri nets, using a low-level nets algebra called *Petri Box Calculus* [4, 3] or its high-level version called *M-nets* [5]. These two levels are related by an *unfolding* operation which transforms an M-net in a low-level net having an equivalent behavior. In this paper, we focus on the M-net semantics since it is much more compact and intuitive. Using PEP toolkit [8, 17], one may input a B(PN)^2 program and automatically generate its M-net or low-level net semantics in order to simulate its behavior or to model-check it against some properties.

The purpose of this paper is to propose a new M-net semantics for channels in B(PN)^2. This semantics uses asynchronous links capabilities newly introduced in M-nets and Petri Box Calculus [11]. The proposed semantics has three main advantages: it is completely expressible in the algebra of M-nets (using only trivial nets as base cases), its size (in terms of the number of places in the unfolding) is considerably smaller than the original semantics and finally, it avoids the "availability defect" of the original semantics (a message sent to a channel was not immediately available for receiving). The solution given here improves what was proposed in [16] by simplifying again the semantics.

$\mathrm{B(PN)}^2$ is presented in [6] and its M-net semantics is fully developed in [9]. In the following, we focus on the intuition in order to keep the paper compact but as complete as possible.

2 M-nets primer

M-nets form a class of labelled high-level Petri nets which were introduced in [5] and are now widely used as a semantic domain for concurrent system specifications, programming languages or protocols [1, 2, 7, 9, 12, 13, 14, 15]. The most interesting features of M-nets, with respect to other classes of high-level Petri nets, is their full compositionality, thanks to their algebraic structure. As a consequence, an M-net is built out of sub-nets with arbitrary "hand-made" nets as base cases.

A place in an M-net is labelled with its *type* (a set of values) which indicates the tokens that the place may hold. In order to define an algebra of M-nets, each place also has a *status* in $\{\mathsf{e}, \mathsf{i}, \mathsf{x}\}$ which reflects whether it is an *entry*, an *internal* or an *exit* place. An M-net is initially marked by its entry marking, in which entry places hold one token from their type and the other places are unmarked. Then, tokens are expected to flow from the entry places to the exit ones. A transition t is labelled with a triple $\alpha(t).\beta(t).\gamma(t)$ where $\alpha(t)$ contains synchronous communication actions, $\beta(t)$ carries asynchronous links annotations and $\gamma(t)$ is a guard which is a set of conditions for allowing or not the firing of t. Finally, arcs are labelled by multi-sets of annotations indicating what they transport on firing.

When a transition t fires, variables in its label and on its surrounding arcs are bound to values, with respect to its guard $\gamma(t)$, the types of its inputs and output places and the type of its asynchronous links annotations. Transition t is allowed to fire only if such a coherent binding can be found using tokens actually available in its input places. When firing occurs, tokens are consumed and produced coherently with respect to the binding.

M-net algebra provides various operations for control flow and communications setup as listed in figure 1. Let us give more details about communications.

Scoping an M-net is the way to perform *synchronous communications* between its transitions. Figure 2 gives an illustration of scoping in a trivial case: in M-net N, transition t_1 carries an action $A(x)$ and t_2 has an $\widehat{A}(y)$; the M-net resulting from scoping $[\![A : N]\!]$ has one transition t_1t_2 which is a mix of t_1 and t_2 such that x and y are unified (here to x) in order to allow an actual communication. (x and y may also be constants in which case unification is only possible when $x = y$.) In a more complex M-net, all maching pairs of transitions such as t_1 and t_2 in figure 2 are considered by the scoping. In the general case, annotations α are multi-sets of actions.

$N_1; N_2$	sequence	N_1 runs then N_2 does
$N_1 \| N_2$	parallel composition	N_1 runs concurrently with N_2
$N_1 \square N_2$	choice	N_1 or N_2 runs but not both
$[N_1 * N_2 * N_3]$	iteration	N_1 runs once (initialization), then N_2 runs zero or more times (loop) and finally N_3 runs once (termination)
$[A : N]$	scoping	sets-up synchronous communications between transitions
$N\,\mathbf{tie}\,b$	asynchronous links	makes asynchronous links between transitions

Fig. 1. Selected operations of the M-nets algebra.

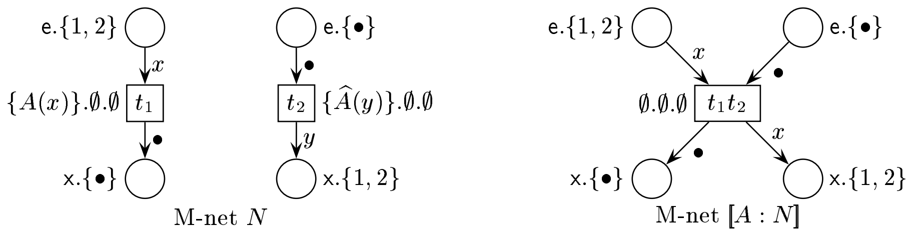

Fig. 2. An example of scoping. (Variables x and y have been unified to x.)

Asynchronous links are available through *links* annotations. A transition may *export* an item x on a *link symbol* b thanks to a link $b^+(x)$; such an exported item may be *imported* later with a link $b^-(y)$. (Here again, x and y may be constants or variables.) Figure 3 gives a basic example of an asynchronous communication between two transitions. In a more complex M-net N, there would be also a single place s_b for all the links on b and all the transitions in N with a link $b^+(x)$ (resp. $b^-(y)$) would be attached an arc to (resp. from) s_b. In general, annotations β are multi-sets of links. In order to give a type to the places added by operator **tie**, each link symbol b is associated a type which becomes the type of any place created by an application of **tie** b.

Notice that synchronous *and* asynchronous communications are allowed on the same transition, this does not mean that a communication can be both synchronous and asynchronous, but only that a transition can perform both kinds of communications, by different ways, at the same time. We will see an example of this in the proposed semantics for channels. Notice also that scoping and asynchronous links are commutative (each one with itself), so we use extended notations such as $[\{A, A'\} : N]$ or $N\,\mathbf{tie}\,\{b, b'\}$.

In the following, in order to avoid many figures, $\alpha(t).\beta(t).\gamma(t)$ will denote an M-net with a lonely transition t annotated by $\alpha(t).\beta(t).\gamma(t)$ and having only one input place and one output place, both of type $\{\bullet\}$ (see figure 4).

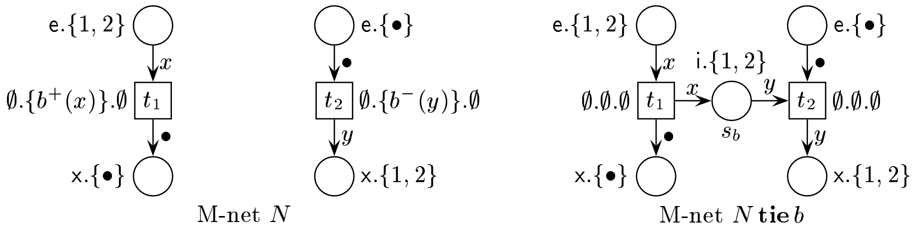

Fig. 3. An example of asynchronous link. (Link symbol b has type $\{1, 2\}$.)

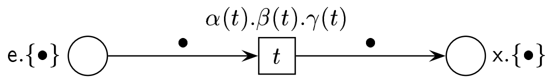

Fig. 4. A simple M-net, denoted by $\alpha(t).\beta(t).\gamma(t)$ in this paper.

3 B(PN)^2 and its M-net semantics

Figure 5 gives a fragment of the syntax of B(PN)^2, semantics is given compositionally: a function Mnet gives an M-net for each fragment of a B(PN)^2 program. The definition of Mnet is recursive on the syntax; base case is either an atomic action, giving an M-net like on figure 4 where t would be labelled in order to implement the action, or a declaration which semantics is given using some special "hand-made" *resource M-nets* (like for channels in the next section).

program	::=	**program** block	(main program)
block	::=	**begin** scope **end**	(block with private declarations)
scope	::=	com	(arbitrary command)
		| vardecl ; scope	(variable or channel declaration)
		| procdecl ; scope	(procedure declaration)
vardecl	::=	**var** *ident set*	(variable declaration)
		| **var** *ident* **chan** k **of** *set*	(channel declaration)

Fig. 5. A fragment of the syntax of B(PN)^2 (procdecl and com are not detailed here).

A B(PN)^2 program is basically a block which may start with some local declarations (variables, channels or procedures), followed by a command which may contain sub-commands and possibly nested blocks. A variable is named with an identifier *ident* and takes its value from the *set* given in its declaration; a channel is declared similarly but with an additional capacity k, it may be 0 for handshake communication, $k \in \mathbb{N}$ for a k-bounded channel or ∞ for an unbounded channel.

The semantics for such a block is obtained from the semantics of its components: we just put in parallel the M-net for the command and the M-net for all the declarations; then we scope on communication actions in order to allow private communications between the components. There is an additional *termination net* which is appended to the command M-net with the purpose to terminate the nets for the declarations. Terminating such a net consists in cleaning it for a possible re-usage. The semantics of any declared resource X contains a transition with an action $\widehat{X_t}$ which performs the cleaning, so the termination net just consists in a parallel composition of M-nets such as $\{X_t\}.\emptyset.\emptyset$.

In the next section, we show and discuss the original semantics for a channel declaration.

4 Original channels in B(PN)^2

Channels for B(PN)^2 were proposed in [6] and reworked in [5] with the M-net semantics depicted in figure 6. There are actually three different semantics, depending on the capacity k of the channel: N_0, N_1 and N_k. Three actions are available for a block which declares a channel C (regardless to its capacity): $\widehat{C}!$ for sending, $\widehat{C}?$ for receiving and $\widehat{C_t}$ for terminating it when the program leaves the block. In order to communicate with the channel, the M-net which implements the program carry actions $C!$ or $C?$. Action C_t can be found in the associated termination net.

In figure 6, transitions are named coherently on M-nets N_0, N_1 and N_k so, excepted when specified, the following description is generic.

The first action on the channel can be performed on transition t_1. For N_0 this means sending (with an action $C!$ in the program) and receiving (action $C?$) on the same transition (it is handshake communication), the guard ensures that the communication is actual; for N_1 and N_k a value is stored in the channel.

Transitions t_2 and t'_2 are for sending and receiving. In N_0, both actions are performed on the same transition t_2. For N_1 or N_k, these actions are separated. In N_1 a value $\varepsilon \notin \mathit{set}$ is used to denote an empty channel. Annotations on arcs ensure that one value can be sent to the channel only if place i holds value ε. The guards ensure that only values in *set* are stored in the channel and that ε is never used in a receiving. For N_k, the situation is more complex since the queue that the channel actually stores is encoded into structured tokens which are k-bounded lists. These lists are stored in i_1 whose type set^k contains all sequences of at most k values from *set*, plus an additional ε for the empty sequence. Transition t_2 adds a value at the end of the list. When place i_2 holds an ε, transition t_4 may extract the head of the list in i_1 and store it in i_2. On the other side, transition t'_2 in N_k is like t'_2 in N_1.

Transitions t_3 and t'_3 are for channels termination (whenever they have been used or not).

For N_k, it is easy to see that the mechanism is quite complex since it requires lists manipulations. Unfortunately, k-bounded channels are certainly the most commonly used... Moreover, an important drawback in N_k may be pointed out:

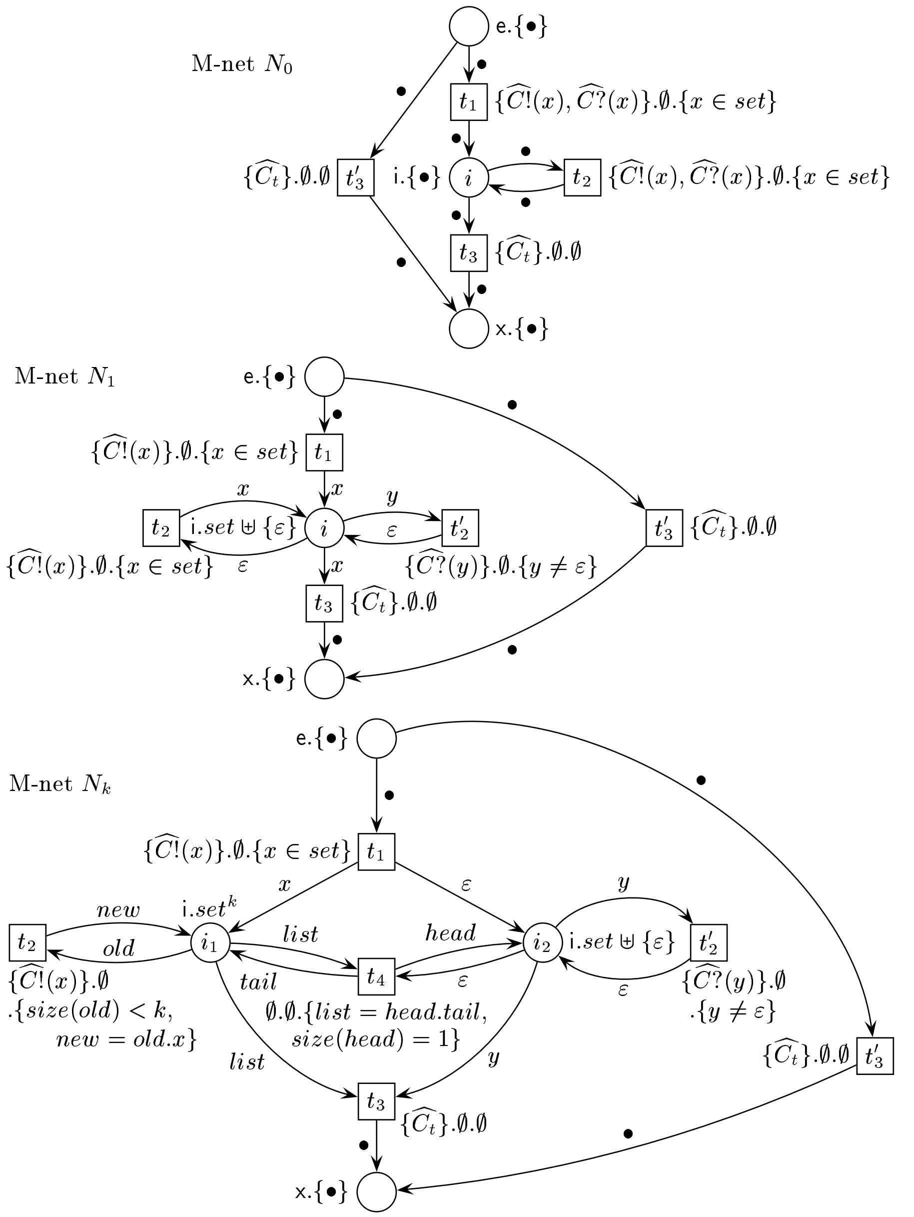

Fig. 6. Original semantics for a channel declaration "**var** C **chan** k **of** *set*". For capacity $k = 0$ (handshake communication) we use the M-net in the top, for $k = 1$ we use the M-net in the middle and for $k > 1$, including $k = \infty$, the M-net in the bottom.

the program has to wait for the firing of t_4 before to be able to receive a value which yet has been sent to the channel.

Let us conclude the current section with a remark on the size of the unfolded M-nets. As far as places are concerned, the unfolding operation produces one low-level place for each value of the type of each place in the M-net. As a direct consequence, unfolding N_k leads to $1 + |set|^k$ places just for i_1; the other places unfold in either 1 or $1 + |set|$ low-level places. So, we can state that the number of places in the unfolding of N_k is $O(|set|^k)$.

5 A new semantics for channels

First of all, let us eliminate the case of capacity $k = 0$ (handshake communications): the semantics for this particular case remains the same. Thus we give its expression (for a declared channel C) as follows:

$$\mathsf{Mnet}(\textbf{var}\ \mathrm{C}\ \textbf{chan}\ \infty\ \textbf{of}\ set) = \left(\{\widehat{C_t}\}.\emptyset.\emptyset\right) \,\Box\, \begin{bmatrix} \{\widehat{C!}(x), \widehat{C?}(x)\}.\emptyset.\{x \in set\} \\ *\ \{\widehat{C!}(x), \widehat{C?}(x)\}.\emptyset.\{x \in set\} \\ *\ \{\widehat{C_t}\}.\emptyset.\emptyset \end{bmatrix}$$

(As explained before, notations as $\{\widehat{C_t}\}.\emptyset.\emptyset$ stand for an M-net like on figure 4 with label $\{\widehat{C_t}\}.\emptyset.\emptyset$ on its transition.)

For an unbounded capacity, we implement the channel as two parallel transitions, one for sending and one for receiving, sharing a place where the data is stored in numbered pairs (x, n) where $x \in set$ is the data to store and $n \in \mathbb{N}$ is its number. Each transition maintains a counter in order to know the number for the next data to be sent and the next to be received. The numbered pairs and the counters are implemented trough asynchronous links. So we have:

$$\begin{aligned}\mathsf{Mnet}(\textbf{var}\ \mathrm{C}\ \textbf{chan}\ \infty\ \textbf{of}\ set) = &\left(\{\widehat{C_t}\}.\emptyset.\emptyset; \{\widehat{C'_t}\}.\emptyset.\emptyset\right) \\ &\Box \left(\, [\![\{I, T\} : core]\!]\ \textbf{tie}\ \{d, ns, nr\}\right)\end{aligned}$$

where the asynchronous links on d are used to store the data sent to the channel, and links on ns and nr implement the counter for the next value to send and the next to receive respectively. Actions I and T are used internally in order to synchronize initialization and termination in the send and receive parts.

The main part *core* is defined as follows:

$$core = [init * send * terminate] \parallel [wait_i * receive * wait_t]$$

It is made of two concurrent iteration, the first is used for sending data to the channel and the other for receiving data from the channel. Part *init* is composed of an unique transition which fires when the first sent takes place, is also sets up the counters and it is synchronized with $wait_i$ so both iteration start concurrently. Symmetrically, *terminate* is used when the resource net for the channel has to be terminated, it clears all the tokens and it is synchronized with $wait_t$ so all the net terminates. Parts *send* and *receive* do exactly what their names suggest.

$$
\begin{aligned}
init &= \{I, \widehat{C!}(x)\}.\{nr^+(0), ns^+(1), d^+((x,0))\}.\{x \in set\} \\
wait_i &= \{\widehat{I}\}.\emptyset.\emptyset \\
send &= \{\widehat{C!}(x)\}.\{ns^-(n), ns^+(n+1), d^+((x,n))\}.\{x \in set\} \\
receive &= \{\widehat{C?}(x)\}.\{nr^-(n), nr^+(n+1), d^-((x,n))\}.\emptyset \\
wait_t &= \{\widehat{T}\}.\emptyset.\emptyset
\end{aligned}
$$

Part *terminate* is the most complicated: it must clear all the tokens in the channel net. Since this number of tokens is not fixed, we must proceed in a loop which relies on the counters in ns and nr in order to know when the net is cleared. This way, termination is not atomic, this is not really a problem if we use a second termination symbol $\widehat{C'_t}$ which synchronizes with the program in order to notify the end of the termination. In other words, where termination was done with a single action C_t, it is now done with a sequence $\{C_t\}.\emptyset.\emptyset; \{C'_t\}.\emptyset.\emptyset$ instead. So we have:

$$
terminate = \Big[\begin{array}{l} \{\widehat{C_t}\}.\emptyset.\emptyset \\ * \ \emptyset.\{nr^-(n), nr^+(n+1), d^-((x,n))\}.\emptyset \\ * \ \{\widehat{C'_t}\}.\{nr^-(n), ns^-(m)\}.\{n = m\} \end{array} \Big]
$$

The semantics for channels of bounded capacity is obtained by simulating a ring buffer. The numbered pairs (x, n) become triples (x, n, b) where $b \in \{\bot, \top\}$, each triple is a slot in the ring buffer, it is marked empty when $b = \bot$ and occupied when $b = \top$. By marking the slots this way, we avoid comparisons between the counters which would disallow concurrent sending and receiving. So, we have:

$$
\mathsf{Mnet}(\textbf{var}\ \mathrm{C}\ \textbf{chan}\ k\ \textbf{of}\ set) = \left(\{\widehat{C_t}\}.\emptyset.\emptyset\right) \ \square \ \left(\llbracket \{I, T\} : core \rrbracket \ \textbf{tie}\ \{d, ns, nr\}\right)
$$

with the sub-parts being:

$$
\begin{aligned}
core &= [init * send * terminate] \parallel [wait_i * receive * wait_t] \\
init &= \{I, \widehat{C!}(x)\}.\{nr^+(0), ns^+(1), d^+((x, 0, \top)), \\
&\qquad\qquad d^+((x, 1, \bot)), \ldots, d^+((x, k-1, \bot))\}.\{x \in set\} \\
wait_i &= \{\widehat{I}\}.\emptyset.\emptyset \\
send &= \{\widehat{C!}(x)\}.\{ns^-(n), ns^+((n+1) \bmod k), \\
&\qquad\qquad d^-((x, n, \bot)), d^+((x, n, \top))\}.\{x \in set\} \\
receive &= \{\widehat{C?}(x)\}.\{nr^-(n), nr^+((n+1) \bmod k), \\
&\qquad\qquad d^-((x, n, \top)), d^+((x, n, \bot))\}.\emptyset \\
terminate &= \{T, \widehat{C_t}\}.\{nr^-(n), ns^-(m), d^-((x_0, 0, b_0)), \\
&\qquad\qquad \ldots, d^-((x_{k-1}, k-1, b_{k-1}))\}.\emptyset \\
wait_t &= \{\widehat{T}\}.\emptyset.\emptyset
\end{aligned}
$$

Notice that the termination is now atomic since we have a fixed number of tokens to remove. Notice also that the counters are incremented modulo k so we really have a ring buffer.

Let us consider the size of the unfolding for this net. We can safely ignore the places created for the control flow by the combination operators since there is a fixed number of such places and they all have type $\{\bullet\}$. We just have to consider the places added by asynchronous links: places for ns and nr have type $\{0, \ldots, k-1\}$ so they unfold in k low-level places; place for d has type $set \times \{0, \ldots, k-1\} \times \{\bot, \top\}$ so it unfolds in $2k|set|$ low-level places. So the unfolding of this net has $O(k|set|)$ places which is a significant improvement with respect to the original semantics. With respect to the semantics proposed in [16], the comparison is harder since the unfolding had $k(|set| + 4)$ places and here it is $2k(|set| + 1)$. Depending on the size of k and set, the new semantics may lead to bigger unfolding or the contrary. Nevertheless, this new semantics improves the one proposed in [16] because the termination is now atomic and not done with an iteration. Moreover, we believe that it is really simpler and much easier to understand.

6 Concluding remarks

We can see that the new proposed semantics has several advantages with respect to the original one. First, there is no need for complex list management and the program does not have to wait anymore before to receive an actually sent value: it is now immediately available. The new semantics is more homogeneous since exceptions are now for $k = 0$ and $k = \infty$ (instead of $k = 0$ and $k = 1$) which we feel to be exceptions *intrinsically*: a handshake is *not* a buffered communication and an unbounded buffer is certainly not realistic.

Moreover, unfolding the M-net for a channel gives now a low-level net with a number of places in $O(k|set|)$ while the original semantics unfolded into $O(|set|^k)$

places. This is a great improvement, especially when considering the model-checking of a $B(PN)^2$ program with channels. The implementation in PEP of the new channel semantics is already planned.

Finally, it is fully expressed in the algebra, with no more "hand-made" M-nets. This application of **tie** tends to show that it is an efficient way to introduce some places with arbitrary types, the control flow being left under the responsibility of the algebra. This allows avoiding the use of "hand-made" M-nets has it is strongly suggested in [10] which gives the semantics of the complete $B(PN)^2$ language without any "hand-made" M-net.

References

[1] V. Benzaken, N. Hugon, H. Klaudel, and E. Pelz. M-net calculus based semantics for triggers. LNCS 1420, 1998.
[2] E. Best. A memory module specification using composable high-level Petri nets. LNCS 1169, 1996.
[3] E. Best, R. Devillers, and J. Esparza. General refinement and recursion operators for the Petri box calculus. LNCS 665:130–140, 1993.
[4] E. Best, R. Devillers, and J. G. Hall. The box calculus: a new causal algebra with multi-label communication. LNCS 609:21–69, 1992.
[5] E. Best, W. Fraczak, R. Hopkins, H. Klaudel, and E. Pelz. M-nets: An algebra of high-level Petri nets, with an application to the semantics of concurrent programming languages. *Acta Informatica* 35, 1998.
[6] E. Best and R. P. Hopkins. $B(PN)^2$ — A basic Petri net programming notation. *PARLE'93*, LNCS 694:379–390, 1993.
[7] H. Fleischhack and B. Grahlmann. A Petri net semantics for $B(PN)^2$ with procedures. *PDSE'97*. IEEE Computer Society, 1997.
[8] B. Grahlmann. The PEP tool. *CAV'97*, LNCS 1254:440–443, 1997.
[9] H. Klaudel. Compositional high-level Petri net semantics of a parallel programming language with procedures. *Science of Computer Programming*, Elsevier, to appear.
[10] H. Klaudel. Parameterized M-expression semantics of parallel procedures. *DAPSYS'00*, Kluwer Academic Publishers, 2000.
[11] H. Klaudel and F. Pommereau. Asynchronous links in the PBC and M-nets. *ASIAN'99*, LNCS 1742:190–200, 1999.
[12] H. Klaudel and F. Pommereau. A concurrent and compositional semantics of preemption. *IFM'00*, LNCS 1945:318–337, 2000.
[13] H. Klaudel and F. Pommereau. A concurrent semantics of asynchronous exceptions in a parallel programming language. *ICATPN'01*, LNCS 2075, pages 204-223, 2001.
[14] H. Klaudel and R.-C. Riemann. Refinement-based semantics of parallel procedures. *PDPTA'99*, vol. 4. CSREA Press, 1999.
[15] J. Lilius and E. Pelz. An M-net semantics of $B(PN)^2$ with procedures. *ISCIS*, vol. 1, 1996.
[16] F. Pommereau. FIFO buffers in tie sauce. *DAPSYS'00*, Kluwer Academic Publishers, 2000.
[17] PEP Homepage. `http://theoretica.informatik.uni-oldenburg.de/~pep`