HAL
open science

FIFO Buffers in tie Sauce

Franck Pommereau

To cite this version:

Franck Pommereau. FIFO Buffers in tie Sauce. Distributed and Parallel Systems, Sep 2000, Balatonfüred, Hungary. pp.95-104, 10.1007/978-1-4615-4489-0_13 . hal-02310048

HAL Id: hal-02310048

https://hal.science/hal-02310048v1

Submitted on 10 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

FIFO buffers in tie sauce

Franck Pommereau

LACL, Université Paris 12,
61 avenue du Général de Gaulle,
94010 Créteil, France
pommereau@univ-paris12.fr

Abstract. This paper introduces a new semantics for FIFO buffers in a parallel programming language, B(PN)^2. This semantics is given in the algebra of M-nets, which is a process algebra with a semantics in terms of labelled high-level Petri nets. The proposed semantics makes usage of newly introduced asynchronous link operator and repairs some drawbacks of the previous semantics: channels are now fully expressible within the algebra, they are considerably smaller, and they offer several other advantages.

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

B(PN)^2 [2] is a general purpose parallel programming language provided with features like parallel composition, iteration, guarded commands, communications through FIFO buffers (more usually called *channels*) or shared variables, procedures [4] and, more recently, real-time extensions with abortable blocks and exceptions [6].

The semantics of B(PN)^2 is traditionally given in terms of Petri nets, using a low-level nets algebra called *Petri Box Calculus* [1] or its high-level version, *M-nets* []. These two levels are related by an *unfolding* operation which transforms an M-net in a low-level net having an equivalent behaviour. In this paper, we focus on M-net semantics since it is much more compact and intuitive.

Using PEP toolkit [3], one may input a B(PN)^2 program and automatically generate its M-net semantics for simulation purpose or the low-level unfolded net in order to model-check ones program 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 an Petri Box Calculus [5]. The proposed semantics has three main advantages: it is completely expressible in the algebra of M-net, its size (in number of places in the unfolding) is considerably smaller that

for the existing semantics and finally, it avoids the "availability defect" of the existing semantics (a message sent to a channel was not immediately available for receiving).

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

M-nets primer

M-nets form a class of labelled high-level Petri nets which were introduced in [] and are now widely used as a semantic domain for concurrent systems specifications, programming languages or protocols []. 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 it may hold. (In order to define an algebra over 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. This particular point is not crucial for our purpose.) On the other hand, a transition t is labelled with a triple $\alpha(t).\beta(t).\gamma(t)$ where $\alpha(t)$ contains (synchronous) communication actions, $\beta(t)$ holds (asynchronous) links annotations and $\gamma(t)$ is a guard which is a condition for allowing or not the firing of t. Finally, arcs are simply labelled by multi-sets of values or variables, indicating what they transport.

When a transition t fires, variables in its annotation and on its surrounding arcs are bound to values, according to the tokens actually available in its input places and with respect to its guard $\gamma(t)$. Transition t is allowed to fire only if such a coherent binding can be found using available tokens. When fire occurs, tokens are consumed and produced coherently with respect to the choosen binding.

M-nets algebra provides various operators for flow control and communications setup as described in figure 1. Let us give more details about communications.

Scoping is the normal mean to perform *synchronous communications* between transitions in an M-net. Figure 2 gives an illustration of scoping in a trivial case: in M-net N, transition t_1 performs an action $A(x)$ and t_2 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 make the communication actual. (x and y may also be constants in which case unification is only

$N_1; N_2$	sequence	N_1 runs then N_2 does
$N_1 \| N_2$	parallel composition	N_1 runs in concurrence 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 (iteration) and finally N_3 runs once (termination)
$[A : N]$	scoping	sets-up synchronous communications between transitions
N **tie** b	asynchronous links	links transitions asynchronously

Fig. 1. Operator on M-nets

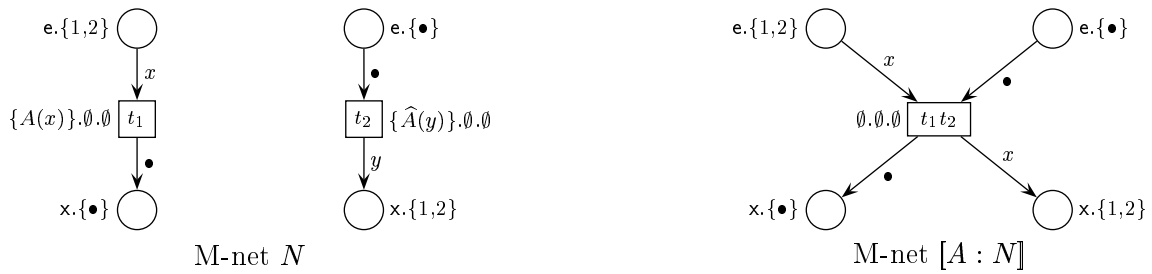

Fig. 2. An example of scoping. (x and y have been unified to x.)

possible when $x = y$.) In a more complex M-net, scoping is performed pairwise, between couples of transitions such as t_1 and t_2. In the general case, annotations α are multi-sets of actions.

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 an example of a basic asynchronous communication between two transitions. In a more complex M-net N, their would be also a single place s_b for all the links on b (there is only one place s_b for N **tie** b but later, there may be new ones if N **tie** b is reused in a context with new 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. Like for synchronous communication actions, annotations β are actually 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.

To conclude on communications, let us add that it is possible to perform synchronous *and* asynchronous communications on the same transition. We will see an example of this in the proposed semantics for channels. Notice also that scoping and asynchronous links being commutative (each one with itself), we use extended notations such as $[\{A, A'\} : N]$ or N **tie** $\{b, b'\}$.

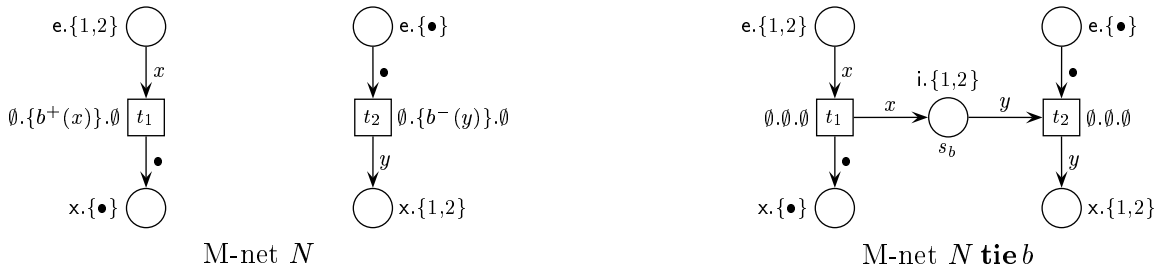

Fig. 3. An example of asynchronous link (b has type $\{1, 2\}$)

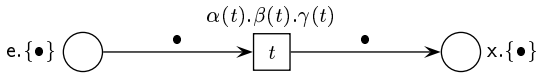

Fig. 4. A simple M-net, denoted by $\alpha(t).\beta(t).\gamma(t)$ in this paper.

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).

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

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

Figure 5 gives a fragment of the syntax of B(PN)^2, semantics is given compositionally and is guided by syntax: it exists a function Mnet which gives its semantics to each fragment of a B(PN)^2 program and whose definition is recursive on the syntax. Base case is either for 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" *ressource M-nets* (like for channels in next section).

A B(PN)^2 program is basically a block which may start with some declarations (they are

kept local to the block) and continue with a command (which may contain sub-commands and possibly nested blocks). Each block may declare its own variables or channels (or procedures): 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 capacity.

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 make the communications between components actual and private. There is an additional *termination net* which is added in sequence to the command and whose goal is to terminate the nets for the declarations: terminating such a net consists in removing all its tokens in order to make it clean for a possible re-usage. The semantics of any declared ressource X contains a transition with an action $\widehat{X_t}$ which performs the emptying, so the termination net just consists in a parallel composition of M-nets such as $\{X_t\}.\emptyset.\emptyset$.

In the following section, we show and discuss current semantics for a channel declaration.

Existing channels in B(PN)^2

Channels for B(PN)^2 were proposed in [] with the M-net semantics depicted in figure 6. There is actually three semantics, depending on capacity k for the channel. Three actions are available for a block which declares a channel C (regardless to is 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 wich 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.

On transition t_1, the first action on the channel can be performed. 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 we just have to put one value in the channel.

Transitions t_2 and t_2' are for sending and receiving. In N_0, like for t_1, both action are performed on the same transition t_2. For N_1 or N_k, these actions are separated. In N_1 we use a value $\varepsilon \notin set$

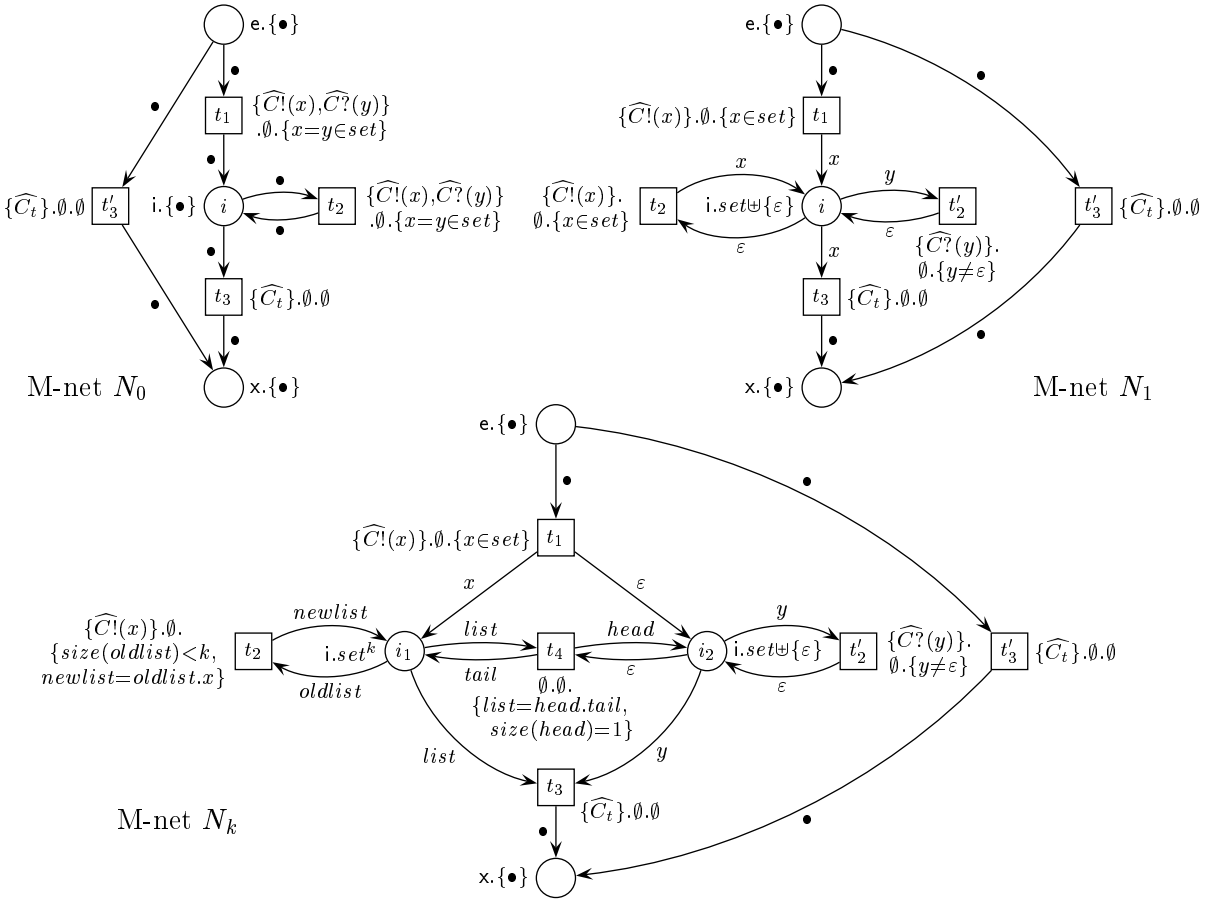

Fig. 6. Existing semantics for a channel declaration "**var** C **chan** k **of** set". Handshake communication (capacity $k = 0$) is shown on the top left, for $k = 1$ we use the M-net on the top right and the M-net of the bottom is for $k > 1$, including $k = \infty$.

to denote an empty channel, annotations on arcs ensure that one value can be wrote 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 read. For N_k, the situation is more complex since the queue that a channel actually stores is encoded into tokens structured has k-bounded lists. These lists are stored in i_1 (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 one value at the end of the list and there is an additional transition t_4 whose goal is to extract the head of the list and to store it in i_2 (only if it holds an ε); on the other side, transition t'_2 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 list manipula-

tions. And unfortunately, k-bounded channels are certainly the most commonly used... Additionally, we can point out an important drawback in N_k: the program has to wait for t_4 to fire before it can receive a value which yet has been sent to the channel.

We will see in the following section that these problems are solved in our semantics of channels.

Let us conclude current section with a remark on the unfolding. We already said that any M-net can be unfolded into a low-level equivalent; it is a labelled place/transition net with the only available token value being $\bullet$. As far as places are concerned, the unfolding operation produces one low-level place for each possible value of each place in the M-net. As a direct consequence, unfolding M-net 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 M-net N_k is $O(|set|^k)$.

A new semantics for channels

First of all, let us eliminate the case of handshake communications: the semantics for this particular case remains the same. For a channel C, it can be expressed as the following M-net:

$$\left(\{\widehat{C_t}\}.\emptyset.\emptyset\right) \ \square \ \left[\ \{\widehat{C!}(x), \widehat{C?}(y)\}.\emptyset.\{x = y \in set\} \ * \ \{\widehat{C!}(x), \widehat{C?}(y)\}.\emptyset.\{x = y \in set\} \ * \ \{\widehat{C_t}\}.\emptyset.\emptyset\ \right]$$

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

For bounded non-zero capacities ($1 \leq k < \infty$), we will use a single M-net, parametrized by k. In order to avoid lists handling, the stored values are numbered modulus k and the M-net takes care to remember the numbers for the next value to send and, separately, the next to receive.

In order to bound capacity, we cannot use a single counter, shared by the sending and receiving actions, which would count how many values are stored at a given moment: this would forbid concurrent sendings and receivings since accessing to this counter would be a conflict on a place. Instead of this, we use a collection of tokens, one for each possible value in the channel, which are considered as "tickets for sending" or "tickets for receiving": in order to store a value, a transition must take a ticket, if none is available, then sending is impossible. Each send transforms a "ticket for sending" into a "ticket for receiving" and symmetrically. So when one action (say sending) is not allowed to fire because of a lack of ticket, it must wait for the other action (here receiving)

to convert one of its tickets (here it means that the channel is not full anymore). This system avoids conflicts since the sending transition consumes "tickets for sending" while the receiving transition does not, and symmetrically.

In the following, we use name K for the set $\{0, \ldots, k-1\}$. The M-net for channels uses asynchronous links for data storage and values numbering; the following link symbols are use with the following meaning:

- c is where numbered data are stored, under the form of couples (v, n) where v is the value and n its number (we can deduce from this that c must have type $set \times K$);
- nw and nr are used to remember the number for the next value to send or receive (both have type K);
- tw and tr store tickets for sending or receiving (also of type K).

The complete semantics for a k-bounded channel C ($1 \leq k < \infty$) is as follows:

$$\mathsf{Mnet}(\textbf{var C chan } k \textbf{ of } set) \;=\; [\![\, \{I, T\} : core \,]\!] \;\; \textbf{tie} \;\; \{c, nw, nr, tw, tr\}$$

where I and T are synchronous communication actions used internally. The core of the semantics can be expressed as two concurrent iteration, one for sends and the other for receives:

$$core = \Big[\; init \;*\; send \;*\; terminate \;\Big] \;\;\Big\|\;\; \Big[\; wait_i \;*\; receive \;*\; wait_t \;\Big]$$

In this M-net, the sending part can be considered as active and the receiving part as passive: M-net $wait_i$ just waits for *init* to be executed and similarly, $wait_t$ waits for *terminate*. The waiting M-nets can be simply expressed has: $wait_i = \{\widehat{I}\}.\emptyset.\emptyset$ and $wait_t = \{\widehat{T}\}.\emptyset.\emptyset$.

M-net *init* is composed of a single transition which fires when the first send takes place.

$$init = \{\widehat{C!}(x), I\} \,.\, \{c^+((x,0)),\; tr^+(0),\; tw^+(1),\; \ldots,\; tw^+(k-1),\; nw^+(1 \bmod k),\; nr^+(0)\} \,.\, \emptyset$$

It stores the value, with number 0, produces all the tickets and initializes "next counters". It also triggers M-net $wait_i$ thanks to a synchronization on I. (Notation "$tw^+(1), \ldots, tw^+(k-1)$" is void when $k = 1$.)

Notice that there is no need to add a guard such as $x \in set$ because the type of c ensures it must be the case. It is the same with *send* and *receive* whose definitions are quite natural:

$$\begin{aligned} dend &= \{\widehat{C!}(x)\} \,.\, \{tw^-(t),\; tr^+(t),\; nw^-(n),\; nw^+(n+1 \bmod k),\; c^+((x,n))\} \,.\, \emptyset \\ receive &= \{\widehat{C?}(y)\} \,.\, \{tr^-(t),\; tw^+(t),\; nr^-(n),\; nr^+(n+1 \bmod k),\; c^-((y,n))\} \,.\, \emptyset \end{aligned}$$

Termination cannot be done in one single action because we do not know where tickets are and how many tokens c holds (whenever it has some). This problem is solved by an iteration wich triggers $wait_t$ on starts:

$$\begin{aligned} terminate \;=\; \Big[\; \{\widehat{C_t}, T\}.\emptyset.\emptyset \quad & * \quad \emptyset.\{tr^-(t),\ tw^+(t),\ c^-((y,n))\}.\emptyset \\ & * \quad \{\widehat{C'_t}\}.\{tw^-(0),\ \ldots,\ tw^-(k-1),\ nr^-(r),\ nw^-(w)\}.\emptyset \;\Big] \end{aligned}$$

(notation "$tw^-(0), \ldots, tw^-(k-1)$" reduces to "$tw^-(0)$" when $k = 0$)

The goal of this iteration is to consume "for free" each value in the channel, and for each such value, one ticket is converted. When (and only when) all the values are consumed (and so, all the tickets are converted), iteration may terminate, consuming all the "tickets for sending" and the "next counters" while synchronizing with the program on a second termination action $\widehat{C'_t}$.

In order to model unbounded capacity, we just have to remove modulus arithmetic on the "next counters" and to forget the system of tickets (just by removing all the links on tw and tr): sending is always possible since the capacity is unbounded and receive is controled by its "next counter" and the availability of data in c (actually, "tickets for receiving" were useless from the beginning but it was necessary to manage them in a coherent way and so, on receive also).

In this case, we cannot rely on tickets to know if the channel is empty but we can trust "next counters": while performing action $\widehat{C'_t}$, we just have to add a guard $r = w$ which means that the count of $c^-(\cdots)$ equals the count of $c^+(\cdots)$ and so that the channel is empty.

Now, let us consider the size of the unfolding in the k-bounded case. Here no place is directly visible since we just gave expressions, but since control flow operators just produce places with types $\{\bullet\}$, it is enough to focus on places added for asynchronous communications. Places for nw, nr, tw and tr have type K so they all unfold into k low-level places. Place s_c for c as type $set \times K$ so it unfolds into $|set| \times k$ low-level places. So we can state a total of $O\left((4 + |set|) \cdot k\right)$ which is a considerable improvement with respect to the exponential size in k for the old semantics.

As a price for this improvement, we now have two termination actions instead of only one. In the termination net, a channel C with the old semantics contributed an M-net $\{C_t\}.\emptyset.\emptyset$, for our semantics, we just have to use instead a sequence $(\{C_t\}.\emptyset.\emptyset); (\{C'_t\}.\emptyset.\emptyset)$.

Concluding remarks

We can see that the new proposed semantics has several advantages over the old one. First, it is expressed in the algebra, with no more "hand-made" M-nets. We think that this application of **tie** tends to show how it can be useful: it is an efficient way to introduce in an M-net some places with arbitrary types, without having to use "hand-made" M-nets.

Additionally, there is no complex list management to do and the program does not have to wait any more before to receive an actually sent value: it is now immediately available. Moreover, 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 *intrinsically* exceptions: a handshake is *not* a buffered communication and an unbounded buffer is certainly not realistic.

Finally, unfolding the M-net for a channel now gives a low-level net with $O\left((4 + |set|) \cdot k\right)$ places while the old semantics unfolded into $O(|set|^k)$ places. This is a great improvement, especially if we consider the problem of model-checking a $\mathrm{B(PN)}^2$ program with channels.

References

1. E. Best, R. Devillers, and J. G. Hall. The box calculus: a new causal algebra with multi-label communication. *Lecture Notes in Computer Science*, 609:21–69, 1992.
2. E. Best and R. P. Hopkins. $\mathrm{B(PN)}^2$ — A basic Petri net programming notation. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *Proceedings of PARLE '93 – Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science, pages 379–390, Munich, Germany, 1993. Springer-Verlag.
3. B. Grahlmann. The PEP tool. In *Proceedings of CAV'97*, volume 1254 of *LNCS*, pages 440–443, 1997.
4. H. Klaudel. Compositional high-level Petri net semantics of a parallel programming language with procedures. Submitted paper (available on http://).
5. H. Klaudel and F. Pommereau. Asynchronous links in the PBC and M-nets. In *Advances in Computing Science – ASIAN'99*, volume 1742 of *LNCS*, pages 190–200. Springer, 1999.
6. H. Klaudel and F. Pommereau. Petri net nemantics of abortion, timeout and exceptions. Technical Report 0021, LACL, Université Paris 12, Feb 2000.