# A Programming Language for Spatial Distribution of Net Systems

Paweł Sobociński and Owen Stephens

ECS, University of Southampton, UK

**Abstract.** Petri nets famously expose concurrency directly in their statespace. Building on the work on the compositional algebra of nets with boundaries, we show how an algebraic decomposition allows one to expose both concurrency and spatial distribution in the statespace. Concretely, we introduce a high-level domain specific language (DSL), PNBml, for the construction of nets in terms of their components. We use PNBml to express several well-known parametric examples.

**Keywords:** Modelling approaches; system design using nets; net-based semantical, logical and algebraic calculi.

## Introduction

Composition of nets is of fundamental importance in constructing models of large systems [22]. A successful theory must combine simplicity, so as not to overburden users with unnecessary technicalities, with a rigorous formal semantics that can be harnessed for reasoning and automated verification, for example via model checking or theorem proving. The interplay between simplicity and rigorous, practical foundations allows the development of modelling languages, tools and techniques that support the user in model design and evaluation, the elimination of bugs, verification, refinement and finally code generation and deployment. The field of compositional concurrent/distributed system specification thus collects insights and techniques from the various communities: models of concurrency, process algebra and programming languages, amongst others.

While Petri nets [20] were introduced in part to study chemical and physical phenomena, their applications until recently have been chiefly in computing. Their popularity is now extending to other disciplines, both scientific and industrial, where distributed, concurrent systems abound [11,15,28]. Their vivid graphical formalism is intuitive, and clearly expresses the concurrency inherent in the systems they model. This information is moreover crucial in verification tasks: for instance, partial order reduction that can alleviate the state-explosion problem. The applicability of such methods relies on the fact that in nets, unlike in mere (interleaving) transition systems, *concurrency is explicit in their structure*: at any point in the computation, one can readily determine the enabled transitions that may be executed concurrently.

Process algebra [10,19] on the other hand, focusses on the study of syntax for component-wise composition of systems. The emphasis is usually on *compositionality*, whereby the behaviour of a composition system is defined exactly in terms of its components' behaviour: there is no emergent behaviour when composing sub-systems, simplifying intuitive and formal reasoning. The practical focus is often on using behavioural pre-orders and equivalences to simplify global complexity. For the latter, the pre-orders and equivalences must be *congruences* with respect to the composition operation(s): one can switch two behaviourally equivalent components without affecting the behaviour of the system as a whole.

As opposed to process algebras, net models are often *monolithic*, with the entire system being modelled in one net. As opposed to Petri nets, the semantics of a process algebra specification is often given in terms of a transition system (TS), that "hides" the concurrency: a TS represents interleavings of concurrent events, thereby obscuring which events can occur concurrently.

In order to combine the advantages of both approaches, one needs to define composition operations on nets that:

- are as simple and intuitive for users as nets themselves,
- are supported by a suite of high-level specification languages and tools for modelling and verification,
- have a compositional formal semantics where typical behavioural pre-orders and equivalences are congruences, thus enabling the use techniques from process algebra,
- have an intuitive graphical presentation that qualitatively eases the task of modelling and quantitatively leads to efficient verification algorithms, for example through the use of partial order reduction.

There have been many proposals in the literature for net composition operators, some of which we discuss below. As observed by Reisig [22], many are quite technical and/or specific for a particular class of nets, making them inconvenient for use by practitioners. Some are equipped with a compositional semantics, yet no formalism has become standard nor widely used. The challenges of modelling large complex systems, given the increasing popularity of Petri nets in several fields, make the quest for a successful theory of net composition timely.

The algebra of Petri nets with boundaries [25,6,7] (PNB) gives a compositional algebra of Petri nets, allowing large nets to be constructed from smaller "components" in the style of process algebra. It features two associative but non-commutative composition operations, and it handles both 1-bounded elementary nets [25] as well as potentially infinite-state P/T nets [6]. The focus of research thus far has been on theory: for instance the algebra was shown to be compositional and lead to congruent behavioural pre-orders and equivalences. Moreover it was proved that all 1-bounded elementary nets can be composed from a small set of primitive PNBs (two nets that express a marked and unmarked place and eight primitive nets for "wiring" together transitions). In this paper, our aim is to demonstrate that PNBs are not merely a theoretical curiosity.

The "raw" algebra of PNBs is not convenient for describing real world systems directly. Syntactic repetition is unavoidable, and expressions soon grow to

be unmanageably large; in a sense the algebra is too low-level, lacking abstraction techniques that give compact and expressive representations. In order to make progress in this direction, we use insights from the functional programming languages community: function abstraction, name binding, type-checking and iteration. The result is a Domain Specific Language, the *Petri Nets with Boundaries Meta Language* (PNBml), that is a central contribution of this paper. We show that programs written in PNBml allow us to construct PNB expressions, and thus nets in a compositional fashion. Programming language features such as type-checking can catch simple, yet important specification bugs.

The language is expressive enough to express all of the parametric examples we have come across in the literature. In order to support our claims, we include programs for a representative selection, including several of Corbett's [9]. These have proved to be popular as benchmark for model checkers and one immediate convenience is that our PNBml programs can generate arbitrary instances as input to a model checker. The type system and intuitive geometric nature of PNBs mean that using PNBml is more convenient and less likely to lead to specification errors than an ad-hoc solution for generating such nets.

Our chief claim is that PNBs retain the distinct benefits of process algebra and Petri nets - compositional reasoning and descriptive graphical representation, whilst allowing for interaction between (sub-)components of a larger net. In this paper, we take several steps to show that the algebra of PNB has what it takes to become a mainstream low-level foundation for the modular specification of complex concurrent and distributed systems. Concretely:

- we show that the algebra of PNBs can be used to write natural specification of realistic systems,
- we introduce of a high-level DSL called PNBml. Programs in PNBml evaluate to PNB expressions, yet PNBml provides a more expressive and convenient setting to specify realistic, parametric systems,
- we provide parametric PNBml programs that generate several well-known examples from the literature.

In our discussion above we have focussed on qualitative issues, and these indeed are an important consideration in this paper. The theoretically minded researcher or tool builder may ask what one gains by writing a PNB expression rather than a global net, apart from the convenience of a program that generates examples of arbitrary size. We believe that, just as one gains techniques such as partial order methods through the explicit treatment of concurrency in the statespace of nets, one can gain additional insight and techniques through expressing a system as an explicit network of synchronised Petri nets. As we show in our examples, the algebra allows the specification of systems constructed from spatially separated components, we have made initial investigations into how this information can be exploited by model checkers [27,26]: briefly, identifying identical components allows *memoisation* to be used when calculating and composing component reachability information. Further, component unreachability implies composite unreachability, allowing the reachability check to "fail fast".

*Related work.* While Petri nets are sometimes accused of being an inherently non-compositional theory, already Mazurkiewicz [17] defined a compositional algebra of nets, based on fusion of named transitions. While in PNBs it is also transitions that are fused, the composition operations are quite different in nature: for one, they are not commutative. Mazurkiewicz's composition is a commutative parallel composition in the spirit of CSP and CCS.

Similar operations were used for the development of the Petri Box calculus (PBC), a process algebra of labelled Petri nets [5]. The PBC features two kinds of composition: the first is a control-flow-style sequential composition that utilises certain places labelled as "entry" or "exit" places in order to enforce a computation order, the second a synchronising composition, introducing new transitions based on the global fusion of transitions with conjugate labels, whilst preserving the original transitions. The composition operations of PNBs, instead, are closely related to the geometry of nets, with no control-flow style composition and only local synchronisation through shared boundary ports.

Reisig's [22] simple composition of nets (SCN) is an elegantly simple way of composing nets and is conceptually quite close to our work. His nets, similarly to PNBs, have left and right interfaces that are made up of ports and ought not to be confused with notions of input and output, rather reflecting the structural geometry of nets. Differently, in SCN the interfaces typically expose places, whereas PNB interfaces expose only transitions. Another difference is that in PNB, composition $N_1 \; ; \; N_2$ is only defined when the right interface of $N_1$ is equal to the left interface of $N_2$. Nevertheless, Reisig's composition is intuitively quite similar to our composition operation ';'. While [22] demonstrates that the operation is very natural for composing real systems, the compositional semantic aspects of the theory have not, so far, been developed.

Component-wise construction of nets was emphasised by Kindler [14] who worked with a partial order semantics. The interfaces are a set of input and output places, that are connected with a transition when composed. The semantics was shown to be compositional with respect to this operation. Because the composition introduces additional transitions, it is not always clear how to divide a net into components with input and output places. This issue is also problematic in formalisms such as open Petri nets [4].

*Structure of the paper.* The remainder of this paper is organised as follows: in §1 we describe the component algebra of nets with boundaries, used to construct composite Petri nets. In §2 we motivate the use of a DSL for more convenient specification of net compositions, informally introducing our DSL, PNBml. In §3 we encode several of Corbett's parametric examples using our DSL. We formally introduce the syntax and semantics of PNBml in §4, and prove that type-correct expressions are guaranteed to correctly evaluate. Finally, we discuss future work and conclude in §5. We have tried to keep our presentation as intuitive and non-technical as possible, illustrating the theory with a large number of examples.

## 1 Nets With Boundaries

In this section we give an intuitive introduction to the algebra of Petri nets with boundaries [25,6,7] (PNB). The formal details can be found in [7]. There are two versions of the algebra, one for $k$-bounded elementary nets (in which the number of tokens at each place is restricted to a positive integer $k$, typically 1) and one for ordinary, potentially infinite-state P/T nets. While explanations and examples in this section can be understood in either version, the bounded version is typically used in applications, since it suffices to characterise the behaviour of safe nets, and its semantics is finite-state. The language of PNBs is inspired by the Span(Graph) algebra of transition systems [12] and the recent wave of formal graphical languages in various fields: for instance graphical languages for quantum information [2], boolean circuits [16], signal-flow graphs [23] etc.

We start by explaining the graphical notation used in this paper. The mathematical structure represented by a classical, unmarked Petri net is a *directed hypergraph*: a directed graph in which edges have arbitrarily many sources and targets. The usual graphical notation in net theory has a transition drawn as a rectangle with directed incoming and outgoing arrows, thus identifying the sources and targets. We use a different notation: instead of orienting transitions, each place is drawn as "directed," having an *in* and *out* port. Transitions are represented by undirected *links*—that is, sets of connected ports—that similarly identify the sources and targets. Indeed, the sources are those places to which it is connected via an out port, and the targets are those places to which it is connected via an in port. The out port of a place is represented by a small triangle pointing out of it, the in port by a triangle pointing out. This alternative notation is compact and particularly convenient when reasoning about transitions in composite PNBs. In order to distinguish individual links and increase legibility, transitions are drawn with a small perpendicular mark. For example, consider the graph with set of nodes $\{A, B, C, D\}$ and a single edge from $\{A, D\}$ to $\{B, C\}$. In the left part of Fig. 1 we illustrate the classical graphical notation of this structure, and the notation used in this paper is given on the right.

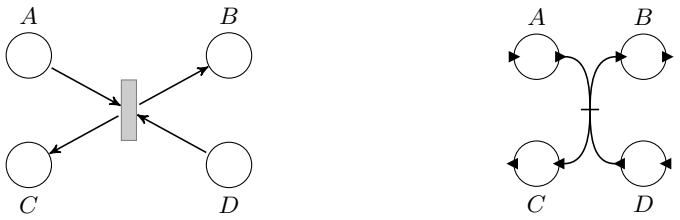



Fig. 1: Two ways of drawing hypergraphs

A PNB is a Petri net with extra structure: two finite ordinals of *boundary ports*, to which net transitions can connect. The two sets of ports are drawn, from

top to bottom, on the left and right hand sides of an enclosing box. An example is given in Fig. 2: here the left set of boundary ports is empty and the right contains two ports. We use the notation $P : (0, 2)$ to mean that $P$ is a PNB with left boundary of size 0 and right boundary of size 2. As in the classical graphical representation, the presence of a token at a place is represented by the small black disc. Now consider the two transitions of $P$, the first, $t$, which connects the first right boundary port with the out-port of place $p$, and the second, $u$, which connects the second right boundary port with the in-port of place $q$.

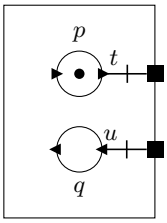



Fig. 2: An example PNB, $P : (0, 2)$

Intuitively, transitions $t$ and $u$ are not yet completely specified because they connect to a boundary port. Thus when composed into a larger net, $t$ may result in several different transitions, all of which will include $p$ in their pre-sets. There are two operations for composing PNBs: synchronisation along a common boundary and a non-commutative parallel composition.

The most interesting operation on PNBs is synchronisation along a common boundary; we illustrate this operation in Fig. 3. In each of the examples, the size of the right boundary of the first net agrees with the size of the left boundary of the second net—this is a general requirement for composition to be defined: nets that do not agree on the size of their common boundary cannot be synchronised. Given nets $X : (k, l)$ and $Y : (l, m)$, their composition is denoted $X ; Y : (k, m)$. In general, transitions of the composed net—called the *minimal synchronisations*—will be subsets of transitions of the individual component nets. We describe this operation informally with examples because the graphical presentation is quite intuitive.

Consider the top left quadrant of Fig. 3. The composed net $P ; Q$ has a transition $\{t, a\}$ that results from synchronising transitions $t$ and $a$. The transition $\{t, a\}$ is now fully specified and will not be further altered because it is not connected to any boundary port in the composed net. The situation for transition $\{u, b\}$ is similar. In the top right quadrant, there are two separate transitions, $c$ and $d$, that can synchronise with $t$. Both the choices are taken into account in the composed net and result in two different transitions $\{t, c\}$ and $\{t, d\}$, which intuitively mean that the transition $t$ in the left net can synchronise in two different ways with transitions in the right net. The transition $e$ does not connect to any places, only to the second boundary port. Thus the corresponding synchronised

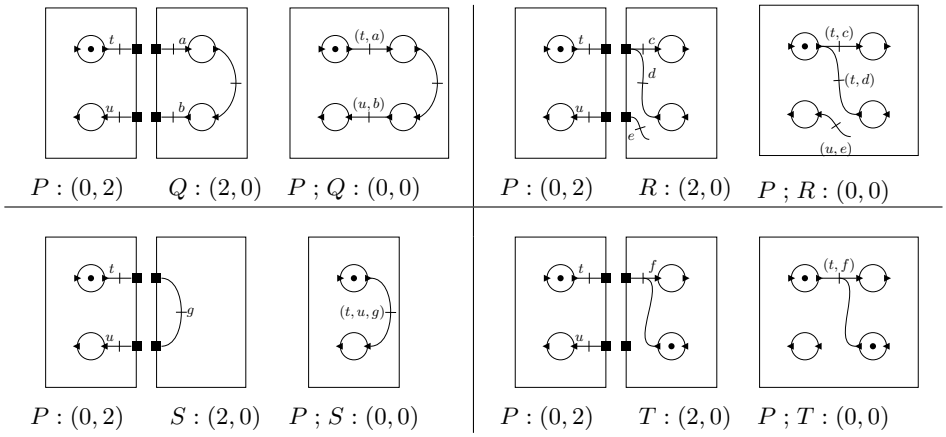
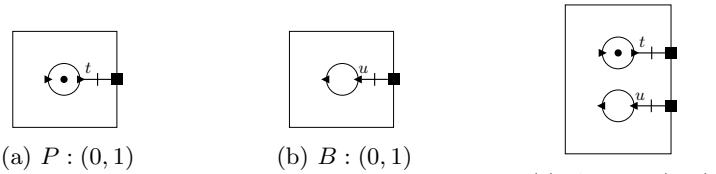



Fig. 3: Examples of compositions of PNBs

transition $\{u, e\}$ has precisely the same pre and post set as transition $u$. In the bottom left quadrant, the transitions $t$ and $u$ are fused into a single transition after composition. In the final example, $u$ has no complementary transition to synchronise with and thus no composite transition results.

The second operation for composing PNBs is called tensor. Graphically, it can be described as "stacking" one net over the other, and intuitively, it acts as a non-communicating parallel composition. Differently from synchronisation along common boundary, any two nets can be tensored: given nets $X : (k, l)$ and $Y : (m, n)$, we have $X \otimes Y : (k + m, l + n)$. A simple example is given in Fig. 4. Both ';' and '$\otimes$' are associative, but neither is commutative.




(a) $P : (0, 1)$ (b) $B : (0, 1)$ (c) $A \otimes B : (0, 2)$

Fig. 4: Example of tensor

PNBs have a labelled transition semantics that is compositional in two ways. First, for any composable nets $N$ and $M$, we have that $[\![N\ ;\ M]\!] \cong [\![N]\!]\ ;\ [\![M]\!]$. The relation $\cong$ is isomorphism of transition systems; on the left of the equation, ';' denotes composition of PNBs, illustrated in Fig. 3, while on the right, ';' denotes composition of their LTSs, see [7].

Similarly, we have that $[\![N \otimes M]\!] \cong [\![N]\!] \otimes [\![M]\!]$. This means that the *behaviour of a composed net depends only on the behaviours of its components*, an important

principle in formal semantics of programming languages: there is no unexpected emergent behaviour in a composite net.

Secondly, the semantics is compositional w.r.t. several standard notions of behavioural equivalence $\sim$, (bisimilarity, weak bisimilarity, language equivalence, etc.): if $[\![N]\!] \sim [\![N']\!]$ then also $[\![N\,;M]\!] \sim [\![N'\,;M]\!]$, $[\![M\,;N]\!] \sim [\![M\,;N']\!]$, $[\![N \otimes M]\!] \sim [\![N' \otimes M]\!]$ and $[\![M \otimes N]\!] \sim [\![M \otimes N']\!]$, whenever the compositions are defined. In particular, this means that *behaviourally equivalent nets can be substituted for each other in any context*. This powerful principle of process algebra is useful when reasoning about the behaviour of complex systems.

### 1.1 Specifying systems algebraically

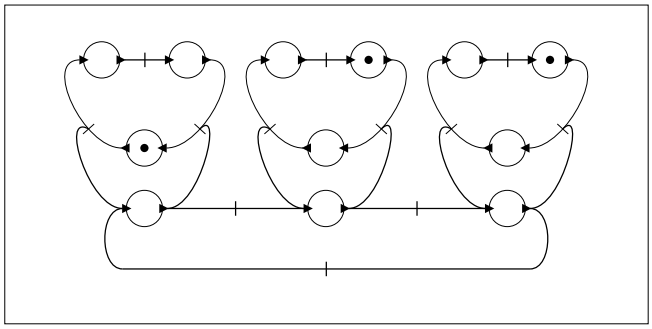
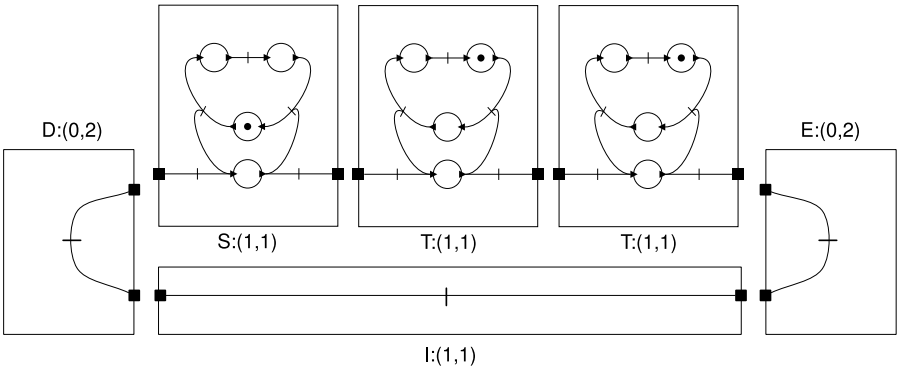


Fig. 5: Token ring network

The examples we have considered thus far have not been of practical interest, having been chosen for their simplicity in order to illustrate the basic operations of PNBs. We now show how a more interesting system can be expressed with the algebra. We will consider other realistic examples in §3.

Consider a model of simple token ring network, taken from [1], and illustrated in Fig. 5. Note that the (1-safe) net contains three identical components, which differ only in their "internal state" (the local marking). Initially, only the leftmost component can proceed: after it finishes its internal computation it relinquishes its token, meaning that the next component can proceed. The modular structure of the system is made explicit with the algebra of PNBs, illustrated in Fig. 6, where we show how the system can be expressed formally as a collection of component PNBs, wired together appropriately with simple connector PNBs. Indeed, when the expression (†) is evaluated by composing nets with boundaries, the resulting Petri net is isomorphic to the net in Fig. 5.

The example is an illustration of the fact that the operations for composing PNBs are closely linked to the underlying geometry of nets – the logical structure of the system is reflected in the structure of the algebraic expression.




$$D\ ;\ ((S\ ;\ T\ ;\ T) \otimes I)\ ;\ E \qquad (\dagger)$$

Fig. 6: A token ring network as a PNB expression

### 1.2 Explicit spatial distribution

Using transition systems as a model of concurrency has a long history (see e.g. [3]). Indeed, the semantics of a Petri net is usually a transition system. Two reasons are often cited by researchers and practitioners in support of working with Petri nets, rather than, for example, products of automata. One is qualitative: the graphical syntax results in vivid, intuitive and informative models of real concurrent and distributed systems. A more empirical, quantitative reason is that transition systems have a monolithic statespace that does not contain inherent information about concurrency. Instead, a state of a Petri net, i.e. a marking, has structure from which one can extract useful information. This leads to practical techniques for mitigating state explosion when model checking, e.g. partial order reduction [18] and symmetry-reduction [24], that would not be possible if working with mere transitions systems.

$$\text{Transition system} \xleftarrow{\text{State graph}} \text{Petri net} \xleftarrow{\text{Composition}} \text{PNB expression} \qquad (\ddagger)$$

A PNB expression can be understood as representing a collection of Petri nets that synchronise with each other on shared transitions. Thus, while inheriting the explicit concurrency of Petri nets, PNB expressions add explicit spatial separation of state to individual component nets. Such a distribution can be obtained directly from the description of the problem and thus reflect the actual physical separation of a distributed system in terms of its components. Several of the examples in this paper (e.g. §3.3,§3.2 and §3.1) demonstrate this principle.

On the other hand, in general, a given Petri net can be described by several different PNB expressions, allowing a logical separation that may not necessarily reflect a physical separation (e.g. the $n$-bit counter example §3.4. One would usually not consider a counter to be distributed in this manner). Considering

different *decompositions* [21] of a Petri net in this way, however, may be beneficial for algorithmic purposes, such as model checking.

Just as Petri nets can be evaluated into a transition system, forgetting the concurrency, a PNB expression can be composed into a Petri net, forgetting the spatial distribution. As we have shown, the close connection between the algebra and net geometry is a qualitative reason for working with PNB expressions. The information can also be exploited quantitatively [27,26] in order to improve the performance of model checking in suitable examples – the statespace of a PNB expression contains information both about concurrency (because the components are Petri nets) as well as spatial distribution.

## 2 A Language for Net Composition

In the previous section we demonstrated the algebraic description of Petri Net systems in terms of their component nets with boundaries. We now motivate using a Domain Specific Language (DSL), PNBml, that evaluates to the algebra of PNB, but adds expressive high-level functional programming language features.

Consider again the algebraic description of a token ring network, as illustrated in Fig. 6. An algebraic expression representing the network with 10 worker components can be written as follows:

$$D \,;\, ((S \,;\, T \,;\, T \ldots ;\, T) \otimes id) \,;\, E$$

where $T$ appears precisely 10 times. However, this is clearly not scalable: for large numbers of components, or more complex components (that may themselves be formed of component compositions) it becomes inconvenient to construct such low-level expressions, and furthermore, ensure that they are correctly composed.

Indeed, consider the expression $t \,;\, (t \otimes t)$ that we might (accidentally) write when composing task nets $t : (1, 1)$. The result of evaluating this expression is undefined, since the two nets being sequentially composed have different size boundaries: it is easy to confirm that $t \otimes t : (2, 2)$. Yet, for the sequential composition to be well-defined, we must have that $t \otimes t$ has boundaries $(1, i)$ for some $i$, a contradiction, indicating an invalid expression. To ensure we disallow such invalid expressions, we must use an appropriate notion of *type*, to ensure that incompatible nets are never composed during evaluation.

When describing complex components, we would like any repeated subcomponents to be described only once, rather than each time they are used.For example, we might consider an extended token ring network model where each task, $T$, is comprised of two sub-components: $T_1$ and $T_2$. Using name binding, we might describe a sequence of such tasks by writing:

$$\textbf{bind } t = (t_1 \;;\; t_2) \textbf{ in } t \;;\; t \;;\; \ldots \;;\; t$$

Another improvement we can make is to abstract over procedures; in the previous example, we perform the procedure "sequentially compose with $t$" several times. Using a lambda notation common to functional programming languages,

we might write this procedure as $\boldsymbol{\lambda} x \,.\, x \,;\, t$ that is, take a suitable net, represent it by the variable $x$ and compose it with $t$. Using this abstraction, we can write:

$$\begin{aligned}&\textbf{bind } t = (t_1 \,;\, t_2) \textbf{ in}\\&\textbf{bind } addt = \boldsymbol{\lambda} x \,.\, x \,;\, t \textbf{ in}\\&addt\,(addt\,(\ldots(addt\,t)))\end{aligned}$$

Finally, we introduce a way of compactly writing an expression to represent "apply a procedure $n$ times to an initial argument", that is, allowing us to represent sequences of tasks as per Fig. 6, but with *parameterised* length.

The notion of repeating an same operation $n$ times is described by "folding" over the number $n$ and repeatedly performing the operation, until $n$ becomes 0.

$$\begin{aligned}&\textbf{bind } t = (t_1 \,;\, t_2) \textbf{ in}\\&\textbf{bind } addt = \boldsymbol{\lambda} x \,.\, x \,;\, t \textbf{ in}\\&\textbf{fold } n\ addt\ t\end{aligned}$$

where **fold** $n\ f\ x$ is an expression that applies $f$ to $x$, $n$ times: $f(f(\ldots(fx)))$. Other examples, such as all of those in §3, are naturally *parametric* and can thus be compactly represented for any particular parameter choice. As an example, we may represent the token ring network of Fig. 6, with the expression:

$$\begin{aligned}&\textbf{bind } procs = \textbf{fold } 2\ \ (\boldsymbol{\lambda} x \,.\, x \,;\, T)\ \ S \textbf{ in}\\&D \,;\, (procs \otimes I) \,;\, E\end{aligned}$$

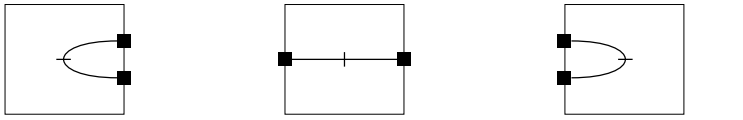
Since PNBml programs evaluate to PNB expressions, we can extend (‡):

$$\text{PNB expression} \xleftarrow{\text{Evaluation}} \text{PNBml program}$$

We defer formally introducing PNBml to §4. Instead, in the next section, we show how it is used to encode several well-known examples.

## 3 Examples

In the following examples we frequently use the "wiring" component nets of Fig. 7, which do not contain places, but are useful when connecting components.

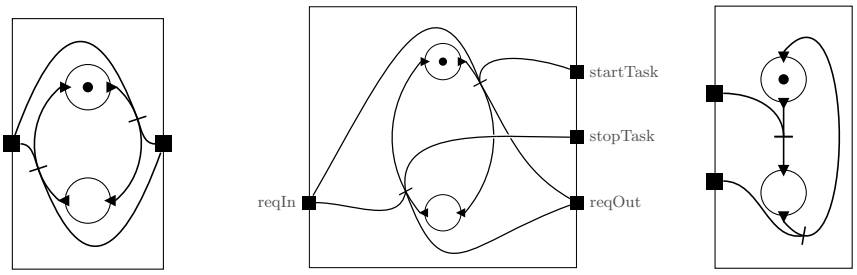


(a) ETA : $\text{Net}\langle 0, 2\rangle$ (b) ID : $\text{Net}\langle 1, 1\rangle$ (c) EPSILON : $\text{Net}\langle 2, 0\rangle$

Fig. 7: Commonly-used wiring component nets

### 3.1 Hartstone

The Hartstone net models a program that starts $i$ tasks in some order, lets them compute, before instructing them to stop them in the same order. In the original description of the problem, a central controller is directly connected to the $i$ tasks. Using nets with boundaries, we can simplify this description: we construct $i$ controllers, each responsible for a single task, with each controller passing signals to the next controller.




(a) MASTER : Net$\langle 1, 1\rangle$ (b) CONTROLLER : Net$\langle 1, 3\rangle$ (c) TASK : Net$\langle 2, 0\rangle$

Fig. 8: Hartstone component nets

The PNBml expression to represent a Hartstone net, with $i$ tasks is as follows:

$$
\begin{aligned}
HART(i) \stackrel{\text{def}}{=}\ & \textbf{bind } \mathit{contTask} = \text{CONTROLLER} \textbf{ ; } (\text{TASK} \otimes \text{ID}) \textbf{ in} \\
& \textbf{bind } \mathit{contTasks} = \textbf{fold } i\ (\boldsymbol{\lambda} x : \text{Net}\langle 1, 1\rangle \textbf{ . } \mathit{contTask} \textbf{ ; } x)\ \text{ID} \textbf{ in} \\
& \text{ETA} \textbf{ ; } (\text{MASTER} \textbf{ ; } \mathit{contTasks}) \otimes \text{ID} \textbf{ ; } \text{EPSILON}
\end{aligned}
$$

This expression represents a sequence of controller/tasks, which are wired to a

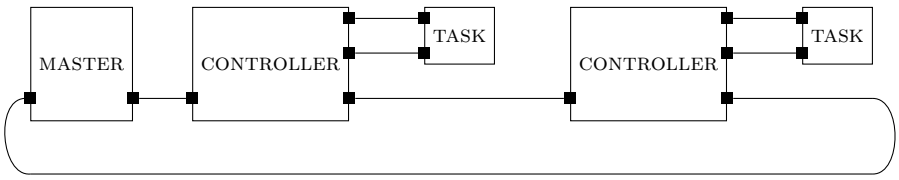


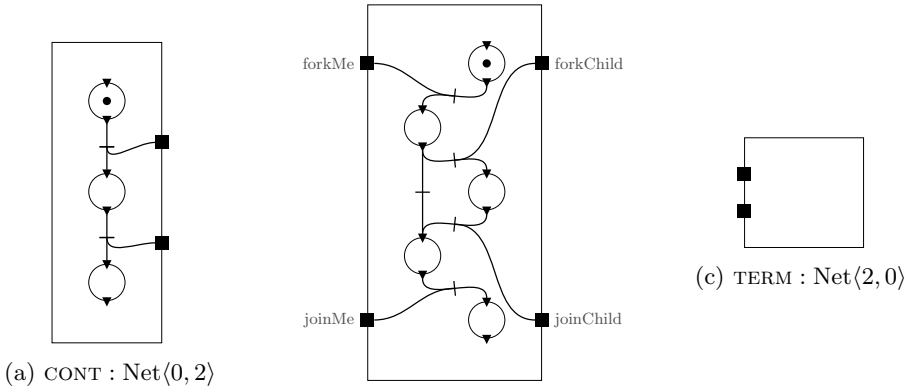

Fig. 9: Schematic of Hartstone(2)

master controller (that models the protocol of repeatedly starting all processes and then stopping them). Signals are looped back around (via ETA/EPSILON), such that MASTER receives the signal when all controllers have already received it. We show a schematic diagram of Hartstone for $i = 2$ in Fig. 9.

### 3.2 DAC: Divide and Conquer

The DAC nets [9] model the recursion in divide and conquer approaches to problem solving. The components of DAC are illustrated in Fig. 10. Each worker can chose to invoke a computation in a child process, or perform all computation itself. If a worker invokes a child process, it must then wait for it and all of its descendants to finish. Each layer in the recursion is modelled by the addition of a worker net. Varying the number of worker nets allows one to treat recursion up to any depth. The worker chain is terminated by a net without synchronising transitions, forcing the last worker to do any remaining work itself. The parametric PNBml expression for DAC with $i$ workers (see Fig. 11), is given below:

$$DAC(i) \stackrel{\text{def}}{=} \textsc{cont} \mathbin{;} \big(\textbf{fold}\ i\ (\boldsymbol{\lambda} x : \mathrm{Net}\langle 2,0\rangle \mathbin{.} \textsc{worker} \mathbin{;} x)\ \textsc{term}\big)$$




(a) CONT : $\mathrm{Net}\langle 0,2\rangle$

(b) WORKER : $\mathrm{Net}\langle 2,2\rangle$

(c) TERM : $\mathrm{Net}\langle 2,0\rangle$

Fig. 10: DAC component nets

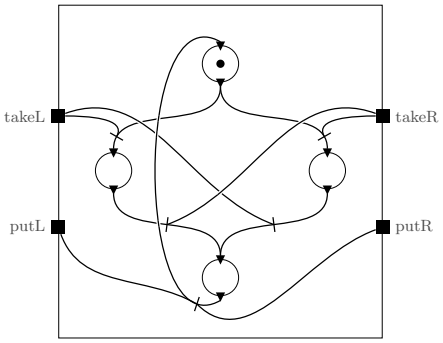

Fig. 11: Schematic of DAC(3)

### 3.3 Dining Philosophers

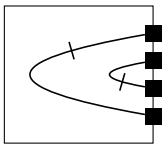
The dining philosophers is the classic example of a concurrent system modelled by Petri nets. The PNBml expression to represent a table of dining philosophers,

with $i$ philosophers and forks requires a simple $id2$ net, which is just $id \otimes id$:

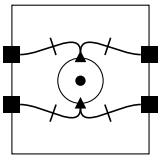
$$
\begin{aligned}
DPH(i) \stackrel{\text{def}}{=}\ & \textbf{bind } phfk = \textsc{ph} \textbf{ ; } \textsc{fk} \textbf{ in} \\
& \textbf{bind } phfks = \textbf{fold } i\ (\boldsymbol{\lambda} x : \mathrm{Net}\langle 2,2\rangle \textbf{ . } phfk \textbf{ ; } x)\ id2 \textbf{ in} \\
& \textsc{lend} \textbf{ ; } (id2 \otimes phfks) \textbf{ ; } \textsc{rend}
\end{aligned}
$$



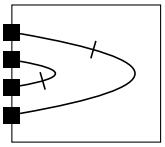

(a) PH : $\mathrm{Net}\langle 2,2\rangle$



(b) LEND : $\mathrm{Net}\langle 0,4\rangle$



(c) FK : $\mathrm{Net}\langle 2,2\rangle$



(d) REND : $\mathrm{Net}\langle 4,0\rangle$

Fig. 12: Dining Philosophers component nets

The expression represents the composition of a philosopher with a fork, before forming a sequence of $i$ such compositions. Then, to form the "table", the last fork is wired together with the first philosopher, as illustrated in Fig. 13.

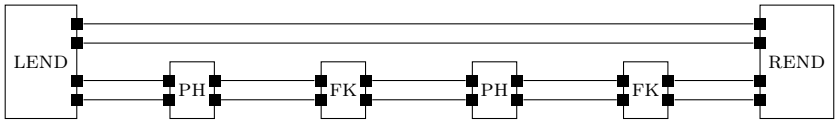
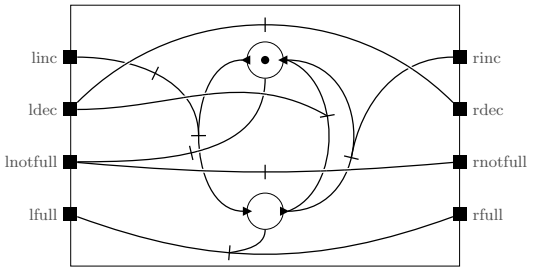


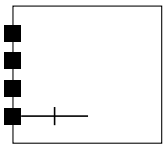

Fig. 13: Schematic of DPH(2)

### 3.4 *n*-bit Counter

An $n$-bit counter net models a "counter" from $0 - n$ that may be incremented/decremented. Counters make use of transitions connected to place "query" ports. Such transitions are represented graphically as an edge that connects to the side of a place — the semantics are that the corresponding transition can only fire if a token is present, and that no transitions connecting to the place's in/out ports are also being fired. Our query ports are equivalent to the read arcs [8] found in contextual nets.




(a) ONEBIT : $\mathrm{Net}\langle 4, 4\rangle$



(b) ZEROBIT : $\mathrm{Net}\langle 4, 0\rangle$

Fig. 14: $n$-bit Counter component nets

An $n$-bit counter net is formed by sequentially composing $n$ 1-bit counter nets, terminating with a net that always reports as being full, see Fig. 15. The intuitive description of a 1-bit component is that it is either "empty" or "full". A full component may be directly decremented, or may pass its token to the next component, in either case it becomes empty. Passing tokens along a chain of components allows the chain to become full — a $n$-bit counter is not full if any component has a token in the empty place; it is full if all places have tokens in the full place. The PNBml expression is:

$$COUNT(i) \stackrel{\text{def}}{=} \textbf{fold}\ i\ (\boldsymbol{\lambda} x : \mathrm{Net}\langle 4, 4\rangle\ .\ \textsc{onebit}\ ;\ x)\ \textsc{zerobit}$$

Clients of the counter interact with the 4 boundary ports on its left boundary: they may increment, decrement and test for notfull/full (for example if the counter is at capacity, transitions connected to *lfull* may fire and otherwise not).

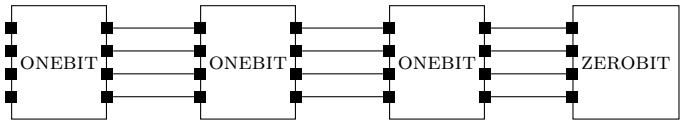



Fig. 15: Schematic of Counter(3)

## 4 PNBml: a DSL for constructing Nets

In this section we give a formal description of the language PNBml for specifying net compositions. In particular, we show how PNBml expressions are evaluated to nets, and how the type system ensures that only valid expressions are processed.

PNBml is a monomorphic call-by-value functional language, following the tradition in classic functional programming languages such as ML of extending the syntax of the lambda calculus with convenient programming constructs. In particular, we add net literals and composition, syntax for variable binding and iterated function application.

The abstract syntax of PNBml is given in Fig. 16: $x$ is drawn from a countable set of variables, $n$ is a net literal (defined using a low-level syntax that we do not describe here) and $i \in \mathbb{N}$. Function application is indicated by simple juxtaposition: $(e_1\, e_2)$. Following mathematical convention $;/\otimes$ associate to the left, and $\otimes$ binds tighter than $;$. We require type annotations on function abstractions to enable type checking, which is discussed in §4.2.

| | |
|---|---|
| $e = x$ | (variable) |
| $\mid n$ | (net literal) |
| $\mid \textbf{bind}\ x = e_1\ \textbf{in}\ e_2$ | (variable binding) |
| $\mid \boldsymbol{\lambda} x : \tau\ .\ e$ | (function abstraction) |
| $\mid e_1\, e_2$ | (function application) |
| $\mid e_1\ ;\ e_2$ | (sequential net composition) |
| $\mid e_1 \otimes e_2$ | (tensor net composition) |
| $\mid \textbf{fold}\ i\ e_1\ e_2$ | (iterated function application) |

Fig. 16: Syntax of PNBml

### 4.1 Operational Semantics

We define the big-step operational semantics of PNBml in Fig. 17, using an explicit variable binding environment $E$. The language is call-by-value, and the evaluation rules reduce PNBml expressions to values: either (composite) nets or function closures (an environment and lambda abstraction).

The evaluation rules are almost standard, with $E \vdash e \Downarrow v$ meaning that in environment $E$, expression $e$ reduces to value $v$. Variables are simply looked up in the environment, nets evaluate to themselves and lambdas evaluate to a closure over the current environment. Bindings evaluate their body in an extended environment, similarly to applications. Sequential and tensor composition evaluate both expressions to nets, before applying the appropriate net operation.

Finally, folds are implemented in terms of (iterated) function application until $i$ reaches the base case, 0.

$$(\textsc{EVar})\ \frac{}{E \vdash x \Downarrow E(x)} \qquad (\textsc{ELam})\ \frac{}{E \vdash \boldsymbol{\lambda} x : \tau \,.\, e \Downarrow \langle E, \boldsymbol{\lambda} x : \tau \,.\, e \rangle}$$

$$(\textsc{ENet})\ \frac{}{E \vdash n \Downarrow n} \qquad (\textsc{EBind})\ \frac{E \vdash e_1 \Downarrow v_1 \qquad E, x \mapsto v_1 \vdash e_2 \Downarrow v_2}{E \vdash \textsf{bind}\ x = e_1\ \textsf{in}\ e_2 \Downarrow v_2}$$

$$(\textsc{EApp})\ \frac{E \vdash e_1 \Downarrow \langle E', \boldsymbol{\lambda} x : \tau \,.\, e_3 \rangle \qquad E \vdash e_2 \Downarrow v_2 \qquad E', x \mapsto v_2 \vdash e_3 \Downarrow v_3}{E \vdash e_1\, e_2 \Downarrow v_3}$$

$$(\textsc{ESeq})\ \frac{E \vdash e_1 \Downarrow n_1 \qquad E \vdash e_2 \Downarrow n_2 \qquad n_3 = n_1 \,;\, n_2}{E \vdash e_1 \,;\, e_2 \Downarrow n_3}$$

$$(\textsc{ETen})\ \frac{E \vdash e_1 \Downarrow n_1 \qquad E \vdash e_2 \Downarrow n_2 \qquad n_3 = n_1 \otimes n_2}{E \vdash e_1 \otimes e_2 \Downarrow n_3}$$

$$(\textsc{EFold0})\ \frac{E \vdash e_2 \Downarrow v}{E \vdash \textsf{fold}\ 0\ e_1\ e_2 \Downarrow v} \qquad (\textsc{EFoldI})\ \frac{E \vdash e_1\ (\textsf{fold}\ (i-1)\ e_1\ e_2) \Downarrow v}{E \vdash \textsf{fold}\ i\ e_1\ e_2 \Downarrow v}$$

Fig. 17: Operational Semantics of PNBml

### 4.2 Static Type Checking

PNBml expressions are statically type-checked, allowing us to rule out errors before evaluation is performed. Such errors include treating a net as a function, sequentially composing incompatible nets, or trying to tensor two lambda abstractions. The (monomorphic) types assigned to PNBml expressions are shown in Fig. 18, with the corresponding typing rules in Fig. 19. As is standard for a monomorphic language, $\Gamma$ is simply a sequence of bindings of variables to types, extended in the TBind and TLam rules.

$$\begin{array}{lll} \tau = \mathrm{Net}\langle i, j \rangle \ \ (i, j \in \mathbb{N}) & & \text{(Net Component)} \\ \quad \mid \tau_1 \to \tau_2 & & \text{(Function Type)} \end{array}$$

Fig. 18: Types of PNBml

$$(\text{TVar})\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad (\text{TBind})\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x \mapsto \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{bind}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}$$

$$(\text{TLam})\ \frac{\Gamma, x \mapsto \tau_1 \vdash e : \tau_2}{\Gamma \vdash \boldsymbol{\lambda} x : \tau_1\ .\ e : \tau_1 \to \tau_2} \qquad (\text{TApp})\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$(\text{TTen})\ \frac{\Gamma \vdash e_1 : \text{Net}\langle i, j\rangle \qquad \Gamma \vdash e_2 : \text{Net}\langle k, l\rangle}{\Gamma \vdash e_1 \otimes e_2 : \text{Net}\langle i + k, j + l\rangle}$$

$$(\text{TSeq})\ \frac{\Gamma \vdash e_1 : \text{Net}\langle i, j\rangle \qquad \Gamma \vdash e_2 : \text{Net}\langle k, l\rangle \qquad j = k}{\Gamma \vdash e_1\ ;\ e_2 : \text{Net}\langle i, l\rangle}$$

$$(\text{TNet})\ \frac{n : (i, j)}{\Gamma \vdash n : \text{Net}\langle i, j\rangle} \qquad (\text{TFold})\ \frac{\Gamma \vdash e_1 : \tau \to \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{fold}\ i\ e_1\ e_2 : \tau}$$

Fig. 19: Typing of PNBml Expressions

As an example use of the typing rules, consider composing the three wiring nets illustrated in Fig. 7 to form a loop, with the expression:

$$\text{ETA}\ ;\ (\text{ID} \otimes \text{ID})\ ;\ \text{EPSILON}$$

We can confirm that the expression is well-composed with the proof illustrated in Fig. 20 (using $\eta, i, \epsilon$ in place of ETA, ID, EPSILON).

$$\dfrac{\dfrac{\vdash \eta : \text{Net}\langle 0, 2\rangle}{} \qquad \dfrac{\dfrac{\dfrac{}{\vdash i : \text{Net}\langle 1, 1\rangle} \qquad \dfrac{}{\vdash i : \text{Net}\langle 1, 1\rangle}}{\vdash i \otimes i : \text{Net}\langle 2, 2\rangle} \qquad \dfrac{}{\vdash \epsilon : \text{Net}\langle 2, 0\rangle}}{\vdash i \otimes i\ ;\ \epsilon : \text{Net}\langle 2, 0\rangle}}{\vdash \eta\ ;\ i \otimes i\ ;\ \epsilon : \text{Net}\langle 0, 0\rangle}$$

Fig. 20: Example typing proof

Our type checker rules out *all* possible run-time failures for PNBml expressions. Indeed, the novel feature of our type system is the tracking of component boundary sizes, achieved by parametrising the Net base type: $\text{Net}\langle l, r\rangle$ by $l, r \in \mathbb{N}$, which describe the left and right boundary sizes respectively.

We say an expression $e$ is *well-typed*, if there exists a type environment binding all free variables appearing in $e$, $\Gamma$, and a type, $\tau$, such that $\Gamma \vdash e : \tau$. Furthermore, an operational environment (mapping lambda-bound-variables to values), $E$ respects a type environment $\Gamma$ iff the domains of $\Gamma$ and $E$ are equal and $E$ point-wise respects $\Gamma$, as per Fig. 21.

$$\frac{dom(E) = dom(\Gamma) \qquad \forall x \in dom(E) \vdash E(x) : \Gamma(x)}{\vdash E : \Gamma}$$

$$\frac{n : (i, j)}{\vdash n : \mathrm{Net}\langle i, j\rangle} \qquad \frac{\vdash E : \Gamma \qquad \Gamma, x \mapsto \tau_1 \vdash e : \tau_2}{\vdash \langle E, \boldsymbol{\lambda} x : \tau_1 \,.\, e\rangle : \tau_1 \to \tau_2}$$

Fig. 21: Typing of values and operational environments

Now we can state our formal notion of "well-typed expressions can always be evaluated to values":

**Theorem 1 (Well-typed expressions are well-composed).**
*For a well-typed expression, $\Gamma \vdash e : \tau$, and operational environment $E$:*

$$E : \Gamma \implies E \vdash e \Downarrow v \text{ with } \vdash v : \tau$$

*Proof.* Induction over the structure of the proof of $\Gamma \vdash e : \tau$.

## 5 Conclusions and Future Work

We have shown that the algebra of Petri nets with boundaries can be used to write natural specification of realistic concurrent, distributed systems. We introduced a high-level DSL, PNBml, which provides an expressive and convenient setting to specify parametric systems, using several techniques: name binding, function abstraction, static types and iterated function application. Although PNBml is a simple language, we have supported our claims of its applicability by providing succinct PNBml programs that generate arbitrary instances of well-known, parametric examples from the literature.

In future work we plan to investigate allowing *polymorphic* functions (w.r.t. parameter boundaries), allowing more expressions to be typed. With suitable *equational unification* and *type inference* type annotations could be omitted from lambda arguments. For example, the "design pattern" of closing a chain of nets using a loop (e.g. the Hartstone example, §3), could be typed:

$$\mathrm{close} : \mathrm{Net}\langle n + 1, m + 1\rangle \to \mathrm{Net}\langle n, m\rangle$$

that is, removing the last boundary on either side by connecting them to one another (here $n$ and $m$ are variables). Another example that could be typed with polymorphic boundaries (and a suitable algebra on boundary sizes) would be the $n$-way tensor:

$$\textbf{fold}\ n\ (\boldsymbol{\lambda} x : \mathrm{Net}\langle i, j\rangle \,.\, x \otimes id)\ x$$

which, given an expression $x$, with type $\mathrm{Net}\langle i, j\rangle$ would perform the $n$-way tensor of $x$ with $id$ giving type $\mathrm{Net}\langle i+n, j+n\rangle$. A well-known example of a type system with similar features is Kennedy's dimension types [13].

## References


1. P. A. Abdulla, S. P. Iyer, and A. Nylén. SAT-Solving the Coverability Problem for Petri Nets. *Formal Methods in System Design*, 24(1):25–43, Jan. 2004.
2. S. Abramsky and B. Coecke. A categorical semantics of quantum protocols. In *LiCS '04*. IEEE Press, 2004.
3. A. Arnold. Nivat's processes and their synchronization. *TCS*, 281(1-2):31–26, 2002.
4. P. Baldan, A. Corradini, H. Ehrig, and R. Heckel. Compositional modelling of reactive systems using open nets. In *CONCUR '01*, pages 502–518, 2001.
5. E. Best, R. Devillers, and M. Koutny. *Petri Net Algebra*. Springer, 2001.
6. R. Bruni, H. C. Melgratti, and U. Montanari. A connector algebra for P/T nets interactions. In *CONCUR '11*, pages 312–326. Springer, 2011.
7. R. Bruni, H. C. Melgratti, U. Montanari, and P. Sobociński. Connector algebras for C/E and P/T nets' interactions. *Log. Meth. Comput. Sci.*, 2013. To appear.
8. S. Christensen and N. D. Hansen. Coloured Petri Nets Extended With Place Capacities, Test Arcs and Inhibitor Arcs. In *ATPN '93*, pages 186–205, 1993.
9. J. C. Corbett. Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Transactions on Software Engineering*, 22(3):161–180, Mar. 1996.
10. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
11. B. H. Junker and F. Schreiber. *Analysis of Biological Networks*. Wiley, 2008.
12. P. Katis, N. Sabadini, and R. F. C. Walters. Span (Graph): A Categorical Algebra of Transition Systems. In *AMAST '97*, pages 307–321. Springer, 1997.
13. A. Kennedy. Relational Parametricity and Units of Measure. In *POPL '97*, pages 442–455. ACM, 1997.
14. E. Kindler. A compositional partial order semantics for petri net components. In *ATPN '97*, volume 1248 of *LNCS*, pages 235–252. Springer, 1997.
15. I. Koch. Petri nets - a mathematical formalism to analyze chemical reaction networks. *Molecular Informatics*, 29(12):838–843, 2010.
16. Y. Lafont. Towards an algebraic theory of boolean circuits. *J Pure Appl Alg*, 184:257–310, 2003.
17. A. Mazurkiewicz. Compositional semantics of pure place/transition systems. In *Advances in Petri nets*, volume 340 of *LNCS*, pages 307–330. Springer, 1988.
18. K. McMillan. A technique of a state space search based on unfolding. *Form Method Syst Des*, 6(1):45–65, 1995.
19. R. Milner. *A Calculus of Communicating Systems*. Prentice Hall, 1989.
20. C. A. Petri. Communication with automata. Technical report, Air Force Systems Command, Griffiss Air Force Base, New York, 1966.
21. J. Rathke, P. Sobociński, and O. Stephens. Decomposing Petri nets. arXiv:1304.3121v1, 2013.
22. W. Reisig. Simple composition of nets. In *ATPN '09*, volume 5606 of *LNCS*, pages 23–42. Springer, 2009.
23. J. Rutten. A tutorial on coinductive stream calculus and signal flow graphs. *Theor Comput Sci*, 343(3):443–481, 2005.
24. K. Schmidt. How to calculate symmetries of Petri nets. *Acta Inf*, 36:545–590, 2000.
25. P. Sobociński. Representations of Petri net interactions. In *CONCUR '10*, number 6269 in LNCS, pages 554–568. Springer, 2010.
26. P. Sobociński and O. Stephens. Penrose: Putting Compositionality to Work for Petri Net Reachability. In *CALCO '13*, pages 346–352. Springer, 2013.
27. P. Sobociński and O. Stephens. Reachability via compositionality in Petri nets. arXiv:1303.1399v1, 2013.
28. W. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.