A Denotational Semantics for Parameterised Networks of Synchronised Automata

Siqi Li, Eric Madelaine

A Denotational Semantics for Parameterised Networks of Synchronised Automata

Siqi Li[†] and Eric Madelaine[§]

[†]Shanghai Key Laboratory of Trustworthy Computing, ECNU, China
[§]INRIA Sophia-Antipolis Méditérannée,
Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, France

Abstract. Parameterised Networks of Synchronised Automata (pNets) is a machine-oriented semantic formalism used for specifying and verifying the behaviour of distributed components or systems. In addition, it can be used to define the semantics of languages in the parallel and distributed computation area. Unlike other traditional process calculi, pNets only own one pNet node as an operator which composes all subnets running in parallel. Using this single synchronisation artifact, it is capable of expressing many operators or synchronisation mechanisms. In this paper, we explore a denotational semantics for parameterised networks. The denotational semantics of parameterised networks we investigate is based on the behaviours of their subnets. The behaviour of a subnet is determined by both its state and the actions it executes. Based on the traces of a set of subnets, the behaviour of a pNet consisting of those subnets can be deduced. A set of algebraic laws is also explored based on the denotational semantics.

1 Introduction

With the rapid development of network technology, a number of software environments or middlewares emerge for facilitating the development of applications distributed over networks. These tools can be used in a variety of contexts, ranging from multiprocessors or clusters of machines, to local or wide area networks, to pervasive and mobile computing. In order to describe the behaviour of distributed systems and to verify properties of such systems, several formal languages and process calculi have been proposed in [3, 9, 11].

Parameterised Networks of Synchronised Automata, abbreviated as pNets, is an element of a pragmatic approach based on graphical specifications for communicating and synchronised distributed objects, in which both events (messages) and agents (distributed objects) can be parameterised. In this framework, pNets is a low level semantic model used for expressing the operational semantics of dedicated programming languages or high-level formalisms for distributed systems. The pNet model is based on the general notion of labelled transition systems, and on hierarchical networks of communicating systems (synchronisation networks), with explicit handling of data parameters in communication

events and in the topology of processes. The agents in pNets can also be parameterised to encode sets of equivalent agents running in parallel. In order to realize communications and synchronisation among the agents in the networks, we use a notion of synchronisation vectors inherited from Arnold [1], but augmented with explicit data values. It provides a general and flexible way to compose any number of components, which matches the expressiveness of many different usual process algebras [2]. Recently we have extended the model towards *open pNets*, that contain *Holes* playing the role of process variables. Open pNets are able to express operators of process algebras or distributed systems, and provides us with a methodology to prove properties of program skeletons, or generic algorithms where we don't care about the details of some parts of the system. They are endowed with an operational semantics and a bisimulation based symbolic equivalence [5].

The concept of pNets was targeted towards the behavioural specification of distributed systems. In the last decade, (closed) pNets have been used to model the behaviours of a number of distributed systems featuring queues, futures, component systems, one-to-many communications, or fault-tolerance protocols Also, the pNets model offers good properties as a formalism for defining the semantics of distributed and heterogeneous systems: it provides a compact hierarchical format, easy to produce from source code. It can also be transformed using abstract interpretation of data domains, and the authors use this approach to construct finite pNets that can be analysed by model-checking [2].

Some research has been done on the formal semantics for distributed computing in order to provide a strong theoretical foundation for those languages or frameworks used in this area. R. Koymans proposed a denotational semantics for a real-time distributed language called Mini CSP-R in [7]. A formal semantics was developed for a distributed programming language named LIPS using Dijkstra's weakest preconditions [10] . Both of the works focus on the parallel execution of the processes but put little emphasis on the hierarchy. As for the pNets model, the study on formalism has just started. An operational semantics and a bisimulation theory for closed pNets are proposed in [4]. Their work also employs some examples to illustrate the expressiveness of pNets. Based on these discussions on formal semantics for pNets, the model checking technology that has been applied to verify the correctness of distributed applications or systems can be improved. Also, it becomes more persuasive and reasonable to be used on safety-critical systems.

This paper proposes a denotational semantics for pNets using UTP theory [6], which can provide another understanding of the formalism complementing the operational approach, and help deduce interesting algebraic properties of parameterised networks. A process (a subnet) in pNets is formalized by a predicate with structured traces and process states. Similar to traditional programming languages, the execution state of a pNet has *completed* state, *waiting* state and *divergent* state to represent the current status and control of the behaviour. A trace is introduced to record the interactions among subnets in the pNets system. The behaviour of a pNets system can be deduced by merging the behaviours of

all subnets together. Besides, we investigate the behaviours of pNets composition with sub-pNets filling some holes by merging the traces of the sub-pNet into the upper-level pNet. Based on the formalized denotational semantics, a set of algebraic laws is obtained.

The rest of this paper is organized as follows. Section 2 recalls the formal definition of pNets with the explanation on the notations and term algebra. Section 3 presents the semantic model of parameterised networks. Section 4 explores a denotational semantics defined structurally on the different elements of the pNet model. Section 5 investigates a set of algebraic laws, including a set of laws concerning parallel composition and pNets composition. Also, we show how we prove properties of various constructs from other languages that we encode using pNets.

2 Parameterised Networks (pNets)

In this section, we recall the formal definition of pNets and the notations that are used in the definition. pNets are tree-like structures. Nodes of the tree (pNet nodes) are synchronising artifacts, using a set of *synchronisation vectors* that express the possible synchronisation between the parameterised actions of a subset of the sub-trees. The leaves of the tree are either pLTSs or Holes. pLTSs (*parameterised labelled transition systems*), are transition graphs with explicite data values and assignments. Holes are placeholders for unknown processes, only specified by their set of possible actions, named the *sort*. A pNet tree with at least one hole is called an *open pNet*.

Notations In the following definitions, indexed structures are extensively used over some countable sets, which are equivalent to mapping over the countable set. We use $a_i^{i \in I}$ to denote a family of elements a_i indexed over the set I. $a_i^{i \in I}$ defines both I the set over which the family is indexed (called range), and a_i the elements of the family. An empty family is denoted $\emptyset$. $\uplus$ is the disjoint union on indexed sets (meaning both indices and elements should be distinct).

Term algebra The pNets model relies on the notion of parameterised actions, that are symbolic expressions using data types and variables. We leave unspecified the constructors of the algebra that will allow building actions and expressions. Moreover, we use a generic *action interaction* mechanism, based on (some sort of) unification between two or more action expressions, to express various kinds of communication or synchronisation mechanisms. We denote $\mathcal{P}$ the set of variables and $\mathcal{T}_\mathcal{P}$ the term algebra over the set of variables $\mathcal{P}$. Within $\mathcal{T}_\mathcal{P}$, we distinguish a set of *action terms* (*parameterised actions*) $\mathcal{A}_\mathcal{P}$ and a set of *expression terms* $\mathcal{E}_\mathcal{P}$ including a set of *Boolean expressions* (guards) denoted as $\mathcal{B}_\mathcal{P}$ (with: $\mathcal{E}_\mathcal{P} \cap \mathcal{A}_\mathcal{P} = \emptyset \wedge \mathcal{B}_\mathcal{P} \subseteq \mathcal{E}_\mathcal{P} \wedge \mathcal{A}_\mathcal{P} \cup \mathcal{E}_\mathcal{P} = \mathcal{T}_\mathcal{P}$). Naturally action terms will use data expressions as subterms. To be able to reason about the data flow between pLTSs, we distinguish *input variables* of the form $?x$ within terms. The function $vars(t)$ identifies the set of variables in a term $t \in \mathcal{T}$, and $iv(t)$ returns its input variables.

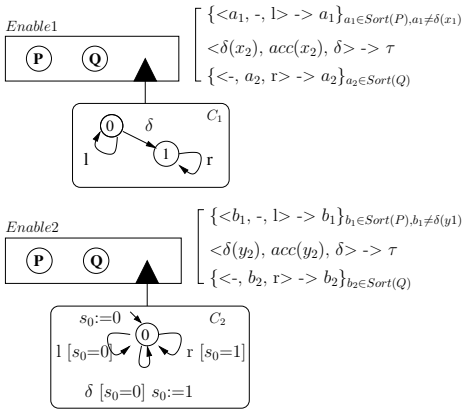


Fig. 1. Two pNets encodings for Enable

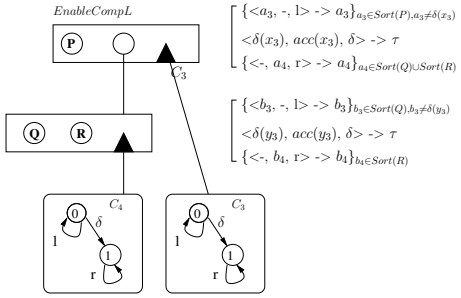


Fig. 2. Composed pNet for `"P»(Q»R)"`

pNets can encode naturally the notion of input actions in value-passing CCS [8] or of usual point-to-point message passing calculi, but it also allows for more general mechanisms, like gate negotiation in Lotos, or broadcast communications. Using our notations, value-passing actions *à la* CCS would be encoded as $a(?x_1, ..., ?x_n)$ for inputs, $a(v_1, .., v_n)$ for outputs (in which v_i are action terms containing no input variables). Our action algebras also include a notion of *local actions*, that cannot be further synchronised; to simplify the notations in this paper we shall simply denote them as τ as in CCS.

Example 1. As a running example, we use pNets representing the *Enable* operator of the Lotos specification language. In the Lotos expression "P»Q", an `exit(x)` statement within process P terminates P, carrying a value x that is captured by the `accept(x)` statement of Q. In Fig. 1 we show two possible pNet encodings for the Lotos operator in a graphical format. Fig. 2 show a hierarchical pNet representing the expression "P»(Q»R)". A pNet is graphically represented by a box, containing circles with a process name, empty circles connected to a subnet and triangles with a line pointing to a box containing a pLTS.

We use a simple action algebra, containing two constructors $\delta(x)$ and $acc(x)$, for any possible value of the variable x, corresponding to the statements `exit(x)` and `accept(x)`. Both $\delta(x)$ and $acc(x)$ actions are implicitely included in the sorts of all processes. The rest of the graphical elements will be explained below.

To begin with, we present the definition of pLTS: a pLTS is a labelled transition system with variables; variables can be manipulated, defined, or accessed inside states, actions, guards, and assignments. Without loss of generality and to simplify the formalisation, we suppose here that variables are local to each state: each state has its set of variables disjoint from the others, denoted $vars(s)$. Transmitting variable values from one state to the other is done by explicit assignment. Note that we make no assumption on finiteness of the set of states nor on finite branching of the transition relation.

We first define the set of actions a pLTS can use, let a range over action labels, op are operators, and x_i range over variable names. Action terms are:

$$\begin{array}{rll} \alpha \in \mathcal{A} ::= & a(p_1, \ldots, p_n) & \text{action terms} \\ p_i ::= & ?x \mid \mathit{Exp} & \text{parameters (input variable or expression)} \\ \mathit{Exp} ::= & \mathit{Value} \mid x \mid \mathit{op}(\mathit{Exp}^*) & \text{Expressions} \end{array}$$

We suppose that each input variable does not appear somewhere else in the same action term: $p_i = ?x \Rightarrow \forall j \neq i.\, x \notin \mathit{vars}(p_j)$. Input variables are only used as binders local to a pLTS, capturing data values coming from synchronisation with other pNets. They will not appear in the action alphabets of pLTSs and pNets, nor in the synchronisation mechanism.

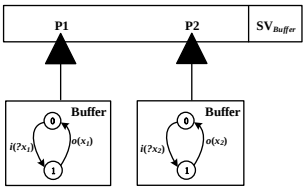
Definition 1 (pLTS). *A pLTS is a tuple* $pLTS \triangleq \langle\!\langle S, s_0, \rightarrow \rangle\!\rangle$ *where:*

- S *is a set of states.*
- $s_0 \in S$ *is the initial state.*
- $\rightarrow \subseteq S \times L \times S$ *is the transition relation and* L *is the set of labels of the form* $\langle \alpha,\ e_b,\ (x_j := e_j)^{j \in J} \rangle$*, where* $\alpha \in \mathcal{A}$ *is a parameterised action,* $e_b \in \mathcal{B}$ *is a guard, and the variables* $x_j \in P$ *are assigned the expressions* $e_j \in \mathcal{E}$*. If* $s \xrightarrow{\langle \alpha,\ e_b,\ (x_j := e_j)^{j \in J} \rangle} s' \in \rightarrow$ *then* $\mathit{iv}(\alpha) \subseteq \mathit{vars}(s')$*,* $\mathit{vars}(\alpha) \setminus \mathit{iv}(\alpha) \subseteq \mathit{vars}(s)$*,* $\mathit{vars}(e_b) \subseteq \mathit{vars}(s)$*, and* $\forall j \in J.\ \mathit{vars}(e_j) \subseteq \mathit{vars}(s) \wedge x_j \in \mathit{vars}(s')$*.*

Example 2. Both pNets in Fig. 1 have a pLTS acting as a controller, in a state-oriented style at the top, and a data-oriented style at the bottom. In a pLTS, states have names, and transitions have labels, written as "`action [guard] assignment`*". The initial state can also have an initial assignment, marked with an arrow. Variables assigned are those of the target state, while variables used in guards or expressions are those of the source state, and input variables of the action. For example the pLTS C_2 has a single state, with a state variable s_0, its transitions include guards (e.g. $[s_0 = 0]$) and assignments (e.g. $s_0 := 1$).

Remark that the conditions on variable sets imply that the local variables of a state s include all input variables received in incoming transitions of s, as well as all local variables explicitely assigned in incoming transitions of s. We denote **Trans**(s) the set of outgoing transitions of s and $tgt(t)$ the target state of t.

Hierarchy and Synchronisation : Now we define the hierarchical operator, called *pNet node* that is the only constructor required for building complex pNets. A pNet node has a set of sub-pNets that can be either pNets or pLTSs, and a set of Holes, playing the role of process parameters. The synchronisation between action of sub-nets is given by a set of *synchronisation vectors*: a synchronisation vector synchronises one or several internal actions, and exposes a single resulting global action. Communication of data between the partners of a synchronisation is done by unification. This synchronisation method is very flexible and generic. It allows to model classical synchronous communication The selection of specific vectors in the set (depending on the actions offered by subnets) models nondeterminism and interleaving. Channels or queues are not handled directly, they



$$SV_{Buffer} = \{< i(x), - >\rightarrow gi(x)\} \cup \{< o(y), i(y) >\rightarrow \tau(y)\} \cup \{< -, o(z) >\rightarrow go(z)\}$$

Fig. 3. A pNet showing data flow

have to be modelled using a pLTS, that will be synchronised with the subnets involved. This is a very versatile and expressive schema, as shown in [4].

Action terms for pNets are simpler than for pLTSs, and defined as follows:

$$\alpha \in \mathcal{A}_S ::= a(Expr_1, \ldots, Expr_n)$$

Definition 2 (pNets). *A pNet is a hierarchical structure where leaves are pLTSs and holes:*
$pNet \triangleq pLTS \mid \langle\!\langle pNet_i^{i\in I}, S_j^{j\in J}, SV_k^{k\in K}\rangle\!\rangle$ *where*

- $I \in \mathcal{I}$ *is the set over which sub-pNets are indexed.*
- $pNet_i^{i\in I}$ *is the family of sub-pNets.*
- $J \in \mathcal{I}_{\mathcal{P}}$ *is the set over which holes are indexed.* I *and* J *are* disjoint*:* $I \cap J = \emptyset$, $I \cup J \neq \emptyset$
- $S_j \subseteq \mathcal{A}_S$ *is a set of action terms, denoting the* Sort *of hole* j*.*
- $SV_k^{k\in K}$ *is a set of synchronisation vectors* ($K \in \mathcal{I}_{\mathcal{P}}$)*.* $\forall k \in K, SV_k = \alpha_l^{l\in I_k \uplus J_k} \rightarrow \alpha'_k$ *where* $\alpha'_k \in \mathcal{A}_{\mathcal{P}}$, $I_k \subseteq I$, $J_k \subseteq J$, $\forall i \in I_k.\, \alpha_i \in \mathrm{Sort}(pNet_i)$, *and* $\forall j \in J_k.\, \alpha_j \in S_j$*. The global action of a vector* SV_k *is* $\mathrm{Label}(SV_k) = \alpha'_k$*.*

Definition 3 (Sorts and Holes of pNets).

– *The sort of a pNet is its signature: the set of actions it can perform. For a pLTS we do not need to distinguish input variables. More formally*[1]*:*

$$\mathrm{Sort}(\langle\!\langle S, s_0, \rightarrow\rangle\!\rangle) = \{\alpha\{\!\{x \leftarrow ?x | x \in iv(\alpha)\}\!\} | s \xrightarrow{\langle \alpha,\ e_b,\ (x_j := e_j)^{j\in J}\rangle} s' \in \rightarrow\}$$
$$\mathrm{Sort}(\langle\!\langle \overline{pNet}, \overline{S}, \overline{SV}\rangle\!\rangle) = \{\alpha'_k | \ \alpha_j^{j\in J_k} \rightarrow \alpha'_k \in \overline{SV}\}$$

– *The set of holes of a pNet is defined inductively; the sets of holes in a pNet node and its subnets are all disjoint:*

$$\mathrm{Holes}(\langle\!\langle S, s_0, \rightarrow\rangle\!\rangle) = \emptyset$$
$$\mathrm{Holes}(\langle\!\langle pNet_i^{i\in I}, S_j^{j\in J}, \overline{SV}\rangle\!\rangle) = J \cup \bigcup_{i\in I} \mathrm{Holes}(pNet_i)$$
$$\forall i \in I.\ \mathrm{Holes}(pNet_i) \cap J = \emptyset$$
$$\forall i_1, i_2 \in I.\, i_1 \neq i_2 \Rightarrow \mathrm{Holes}(pNet_{i_1}) \cap \mathrm{Holes}(pNet_{i_2}) = \emptyset$$

A pNet Q is *closed* if it has no hole: $\mathrm{Holes}(Q) = \emptyset$; else it is said to be *open*.

[1] $\{\!\{x_k \leftarrow e_k\}\!\}^{k\in K}$ is the parallel substitution operation.

Graphical syntax: When describing examples, we usually deal with pNets with finitely many sub-pNets and holes, and it is convenient to have a more concrete syntax for synchronisation vectors. When $I \cup J = [0..n]$ we denote synchronisation vectors as $< \alpha_1, .., \alpha_n > \rightarrow \alpha$, and elements not taking part in the synchronisation are denoted $-$ as in: $< -, -, \alpha, -, - > \rightarrow \alpha$.

Example 3. Back to Fig. 1, the first synchronisation vector of pNet `Enable1` means: for every action a_1 in the sort of P with $a_1 \neq \delta(x_1)$ for some x_1, this a_1 action can synchronise with the l action of the controller, and this synchronisation is seen as a global action a_1 of `Enable1`. Vectors are defined in a parameterised manner, using variables universally quantified, and local to each vector.

More examples can be found in [4].

Composition operator: Open pNets can be composed by replacing one hole (at some arbitrary position in the tree) by a pNet with a compatible sort:

Definition 4 (pNet Composition). *Let* $N1 = \ll pNet_i^{i \in I}, S_j^{j \in J}, \overline{SV} \gg$ *and* $N2$ *be two pNets,* ho *a hole of* $N1$ *such that* $Sort(N2) \subseteq S_{h0}$*, their composition denoted* $N1[N2]_{ho}$ *is:*

if $ho \in J$ *then* $N1[N2]_{ho} = \ll (pNet_i)^{i \in I} \uplus N2, S_j^{j \in J \setminus \{ho\}}, \overline{SV} \gg$
else $\exists i0 \in I.ho \in Holes(pNet_{i0})$
and $N1[N2]_{ho} = \ll (pNet_i)^{i \in I} [pNet_{i0} \leftarrow pNet_{i0}[N2]_{ho}], S_j^{j \in J}, \overline{SV} \gg$

Remark that the composition operation does not change synchronisation vectors at any level in the pNet structure: only the hole involved is replaced by a subnet, and the sort inclusion condition ensures the actions of the subnets are properly taken into account by the synchronisation vectors. This is essential for keeping the compositional features of the model.

Example 4. Figure 2 shows that the hole Q in the pNet `Enable1` in Figure 1 is instantiated by another instance of `Enable1` where $Sort(Enable1) \subseteq Sort(Q)$. The composed pNet represents the Lotos process expression "P $\gg$ (Q $\gg$ R)", denoted as Enable(P,P')[Enable(Q,R)]$_{P'}$. Both C_3 and C_4 contain instances of the controller pLTS. Here a_3, a_4, b_3 and b_4 in the synchronisation vectors are variables that can take any value in the sort of their corresponding holes.

3 The Semantic Model

Now we define the denotational semantic model for pNets based on the UTP theory [6] in this section. UTP uses relational calculus as a unifying basis to define denotational semantics for programs across different programming paradigms. In the semantic models, different programming paradigms are equipped with different *alphabets* and a selection of laws called *healthiness conditions*. An alphabet is a set of observational variables recording external observations of the program behaviour. The healthiness conditions are kind of invariants, imposing

constraints on the values and evolution of variables. The observational variables are defined in a structural manner, using relational predicates relating the possible values of the variables of a given program construct with those of its parts.

In our semantic model, we use the notion of *process*, which is widely used in a number of process algebra and calculus, to denote the various forms of processes in the pNet formalism, namely pLTS, sub-pNets, holes and even a whole pNet system. We say that a process *fires* a transition, that means a transtion of a pLTS, an action of a hole, or a "global action" generated by the execution of a synchronisation vector in the case of a pNet node.

The predecessor of a pNet process is a process executed just before the current execution step. This process may either have terminated successfully so that the current process can start, or it may have not terminated and its final values are unobservable.

With the understanding of the specific meaning of these notions, we introduce the following variables defined for the alphabet to observe the behaviours of pNets processes.

- **Status** st, st': express the execution state of a process before and after a transition is fired, with values in $\{comp, wait, div\}$.
  - *completed* state: A process may complete all its execution and terminate successfully. "$st = comp$" means that the predecessor of the process has terminated successfully and the control passes into the process for activation. "$st' = comp$" means that the process itself terminates successfully.
  - *waiting* state: A process may wait for receiving messages from its environment. "$st = wait$" indicates that the predecessor of the process is at waiting state. Hence the considered process itself cannot be scheduled. "$st' = wait$" indicates that the current process is at waiting state.
  - *divergent* state: A process may perform an infinite computation and enter into a divergent state. "$st = div$" indicates that the predecessor of the process has entered into a divergent state, whereas "$st' = div$" indicates that the process itself has entered into a divergent state.
- **Current state** cs, cs': denote the state (corresponding to the set of states in pLTS) where the current execution begins and terminates. This is used to help relate all the transitions and figure out which transition should be fired.
- **Data store** $ds(s)^{s \in S}$, $ds(s)'^{s \in S}$: record the values of the local variables of the state s in the set S before and after an observation. We will use the notations $\overline{ds}$ and $\overline{ds}'$ to denote the full set of Stores for simplicity.
- **Trace** tr, tr': record a sequence of observations on the interaction among the subnets. The elements in the trace variable are in the form of $\alpha(v_1, \ldots, v_n)$ where $n \geq 1$ or just a value v. Here, $v_1, \ldots, v_n$ can be values either recorded directly from the message transmission or computed from the expressions.

Notations for traces. In the following, $t[i]$ is the i^{th} element in the trace t; $t_1 \preceq t_2$ denotes that sequence t_1 is a prefix of sequence t_2; $\langle l^k \rangle$ is the trace where the element l is repeated k times; $\langle l^* \rangle$ the trace where l is repeated in any finite

number of times; $t^\frown t'$ the concatenation of traces t and t'; and $s \upharpoonright A$ means trace s is restricted to the elements in set A.

Before we present the denotational semantics of each process in pNets, we will define some healthiness conditions that a pNet process should satisfy. The first point is that the trace variable introduced to our semantics cannot be shortened: an execution step can only add an event to the trace. This is encoded as the $\mathcal{H}1$ law below: a predicate P defining the semantics of any pNet process must satisfy :

$(\mathcal{H}1) \quad P = P \wedge Inv(tr)$ where $Inv(tr) =_{df} tr \preceq tr'$.

The next point deals with divergent processes: "$st = div$" means that the predecessor process has entered the divergent state and the current process will never start. Therefore, a pNets process P has to meet the healthiness condition below:

$(\mathcal{H}2) \quad P = P \vee (st = div \wedge Inv(tr))$

A process may wait for receiving message from other subnets or the environment. If the subsequent process is asked to start in a waiting state of its predecessor, it leaves all the states unchanged, including the trace and all its other observational variables. It should satisfy the following healthiness condition:

$(\mathcal{H}3) \quad P = \mathrm{II} \triangleleft (st = wait) \triangleright P$

where we denote the logical choice: $P \triangleleft b \triangleright Q =_{df} b \wedge P \vee \neg b \wedge Q$,

the II relation: $\mathrm{II} =_{df} Inv(tr) \triangleleft st = div \triangleright Id$.

and the identity relation: $Id =_{df} (st' = st) \wedge (tr' = tr) \wedge (cs' = cs) \wedge (\overline{ds}' = \overline{ds})$.

Now we give the definition for $\mathcal{H}$-function:

$$\mathcal{H}(X) =_{df} (X \wedge Inv(tr)) \triangleleft st = comp \triangleright (Inv(tr) \triangleleft st = div \triangleright \mathrm{II})$$

From the definition of $\mathcal{H}$-function, we know that $\mathcal{H}(X)$ satisfies all the healthiness conditions. This function can be used in defining the denotational semantics for pNets model.

The definitions here are similar to the one in [6], with the following correspondance with the variables ok and $wait$ from the original UTP theory: $st = comp$ corresponding to the situation that $ok \wedge \neg wait$, $st = wait$ corresponding to $ok \wedge wait$ and $st = div$ corresponding $\neg ok$.

4 Denotational Semantics

In this section, we present the denotational semantics for the four constructs of pNets: pLTSs, Holes, pNet nodes and pNet composition. We use $\mathbf{beh}(P)$ to describe the behaviour of a pNet process after it is activated. Here P can be any type of pNet or a transition of a pLTS.

4.1 Parameterised Labelled Transition System

The order of execution in a pLTS relies on the relations between states and transitions, encoded in its transition relation. The variable cs is used to keep tracking the execution of the pNets processes so that we know at which step the execution will continue. The denotational semantics of a pLTS is given below.

$$\mathbf{beh}(\langle\!\langle S, s_0, \rightarrow \rangle\!\rangle) =_{df} \mathbf{beh}(\mathbf{Init}((x_j := e_j)^{j \in J}), s_0) \fatsemi \mathbf{beh}(\langle\!\langle S, s_0, \rightarrow \rangle\!\rangle_{s_0})$$

$$\mathbf{beh}(\langle\!\langle S, s_0, \rightarrow \rangle\!\rangle_{cs}) =_{df} \bigvee_{t \in \mathbf{Trans}(cs)} \left(\mathbf{beh}(t) \fatsemi \mathbf{beh}(\langle\!\langle S, s_0, \rightarrow \rangle\!\rangle_{\mathbf{tgt}(t)})\right)$$

where $P \fatsemi Q$ denotes the sequential composition in the form of relational calculus, meaning that $P \fatsemi Q = \exists obs_0 . P[obs_0/obs'] \wedge Q[obs_0/obs]$. The term obs (resp. obs_0, obs') represents the set of variables st, cs, ds and tr.

The behaviour of a pLTS is the set of traces computed from its initial state. The set of traces computed from an arbitrary state cs is the union of all traces obtained using its set of outgoing transitions $\mathbf{Trans}(cs)$, followed by the traces of their target states.

$$\mathbf{beh}(\mathbf{Init}((x_j := e_j)^{j \in J}), s_0) =_{df}$$

$$\mathcal{H}\left(\, st' = comp \ \wedge ds(s_0)' = \{x_j := e_j\}^{j \in J}) \wedge tr' = \langle\,\rangle \wedge cs' = s_0 \,\right)$$

The above semantics deals with the initialisation on the local variables of the initial state as well as other observational variables.

Now we look into the details of the execution of a transition. For the actions, as we defined in the action terms, we will mainly use action algebras in this form: $\alpha(?x_1, \ldots, ?x_{n_1}, e_1, \ldots, e_{n_2})$. For simplicity, we will use the notation $\alpha(?x, e)$ instead when giving our semantics. Note that the forms of the actions are not limited to this, but are out of the scope of this paper. The execution of one single transition is atomic without interruption by the other processes. If the guard evaluates to `false`, then the trace remains unchanged and the variables stay in the initial state. Otherwise there will be two stages. One stage is the waiting state, at which the process is waiting for input values and all other observational variables stay unchanged. The other stage is the terminating state. If there are input variables in the action, they will be assigned input values. Then, the values of the local variables in the assignments will be updated accordingly.

$$\mathbf{beh}(t) = \mathbf{beh}(s \xrightarrow{\langle \alpha(?x,e), e_b, (x_j := e_j)^{j \in J} \rangle} s') =_{df}$$

$$\mathcal{H}\left(\begin{array}{l} \left(\begin{array}{l} st' = wait \wedge tr' = tr \wedge cs' = cs \wedge ds(s')' = ds(s') \\ \vee \\ st' = comp \wedge \exists m \in \mathbf{Value} \,. \\ \qquad \left(\begin{array}{l} tr' = tr^\frown \langle \alpha(m, e) \rangle \wedge cs' = s' \wedge \\ ds(s')' = ds(s')[m/x, (e_j/x_j)^{j \in J}] \end{array}\right) \end{array}\right) \\ \lhd e_b \rhd \\ st' = comp \wedge tr' = tr \wedge cs' = cs \wedge ds(s')' = ds(s') \end{array}\right)$$

Here, **Value** stands for all possible values which can be transmitted by subnets in the whole pNet. In the action $\alpha(?x, e)$, there is an input variable x, an

expression e whose value will be sent. It is obvious that a value (denoted m here) is assigned to x, thus we have $\alpha(m, e)$ recorded in the trace and the value of x changed in the local variables of the target state s'. Each transition produces a single action, and a single step in the relational semantics. After the execution of the transition, the pLTS moves to the target state at which the next execution will start.

Example 5. Recall the pLTS C_1 from Figure 1. There are three transitions in the pLTS and we would like to show how its semantics is obtained.

We start unfolding the definitions for Initialization,

$\mathbf{beh}(C_1) = \mathcal{H}\,(st' = comp \wedge \emptyset \wedge tr' = \langle\,\rangle \wedge cs' = 0) \fatsemi \mathbf{beh}((C_1)_0)$

Then the definition for states:

$\mathbf{beh}((C_1)_0) = \mathbf{beh}(0 \xrightarrow{l} 0) \fatsemi \mathbf{beh}((C_1)_0) \vee \mathbf{beh}(0 \xrightarrow{\delta} 1) \fatsemi \mathbf{beh}((C_1)_1)$

$\mathbf{beh}((C_1)_1) = \mathbf{beh}(1 \xrightarrow{r} 1) \fatsemi \mathbf{beh}((C_1)_1)$

and for each transition, e.g.:

$\mathbf{beh}(0 \xrightarrow{l} 0) = \mathcal{H}(st' = comp \wedge tr' = tr^\frown\langle l\rangle \wedge cs' = 0)$

Now we apply the semantics of the sequence ($\fatsemi$) operator, producing recursive equations on the value of the observational variables:

$\mathbf{beh}((C_1)_0) = \mathcal{H}(B)$ such that

$B = \exists st_1, tr_1, cs_1.(st_1 = comp \wedge tr_1 = tr^\frown\langle l\rangle \wedge cs_1 = 0) \wedge B[st_1/st, tr_1/tr, cs_1/cs]$

Unfolding this equation k times, eliminating intermediate variables, and adding the initialisation step finally gives us:

$B = \exists st_0, st_1, ..., st_k, tr_k, cs_k.\ \vee\,(st_k = comp \wedge tr_k = tr^\frown\langle l^k\rangle \wedge cs_k = 0) \wedge B[st_k/st, tr_k/tr, cs_k/cs]$.

Building now the semantics of the full C_1 pLTS yields to a set of mutually recursive equations on predicate variables, with one such variable for each state of the pLTS. Here the solution is $\{tr, cs\}$, where:

$tr = \langle l^*\rangle, cs = s_0 \vee \quad tr = \langle l^*\rangle^\frown\langle\delta\rangle,\ cs = s_1 \vee \quad tr = \langle l^*\rangle^\frown\langle\delta\rangle^\frown\langle r^*\rangle,\ cs = s_1.$

Example 6. Consider now the pLTS C_2 in Fig. 1, who has a state variable s_0. The initialization gives:

$\mathbf{beh}(C_2) = \mathcal{H}\,(st' = comp \wedge ds' = \{s_0 := 0\} \wedge tr' = \langle\,\rangle \wedge cs' = 0) \fatsemi \mathbf{beh}((C_2)_0)$

Its semantics is $\{tr, ds, cs\}$, where:

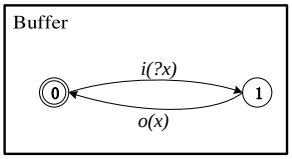
$tr = \langle l^*\rangle,\ ds = \{s_0 := 0\},\ cs = 0.$

$\vee \quad tr = \langle l^*\rangle^\frown\langle\delta\rangle,\ ds = \{s_0 := 1\},\ cs = 0$

$\vee \quad tr = \langle l^*\rangle^\frown\langle\delta\rangle^\frown\langle r^*\rangle,\ ds = \{s_0 := 1\},\ cs = 0.$

Example 7. Finally we give an example of a pLTS with value-passing, that is using an input variable.

The set of solutions is $\{tr, ds, cs\}$, where traces a formed by a sequence of i/o actions a finite number of times, eventually followed by a single i action (when moving to state 1). Naturally in each cycle the value carried can be different:



A pLTS with a data store

$$tr = Cycles(l),\ ds = \emptyset,\ cs = 0$$
$$\vee \quad tr = Cycles(l)^\frown\langle i(v_{l+1})\rangle,\ ds = \{x := v_{l+1}\},\ cs = 1,\ \forall v_{l+1} \in \mathbf{Value}.$$

in which $Cycles(l) = \langle (i(v_k), o(v_k))^* \rangle$ with $\forall k \in [1..l].v_k \in \mathbf{Value}$

4.2 Holes

Now we investigate the semantics for the holes, where we benefit from the semantics of transitions in pLTSs. For a hole H with sort $Sort(H)$, we define the maximum behaviour of H by building a single state pLTS, being able to perform any sequence of actions of the sort.

$$MaxLTS(H) = \langle\!\langle \{s_0\}, s_0, \rightarrow \rangle\!\rangle, with \qquad \forall a \in Sort(H).s_0 \xrightarrow{\langle a \rangle} s_0$$
$$\mathbf{beh}(H) =_{df} \mathbf{beh}(MaxLTS(H))$$

4.3 Parallel Composition

This section investigates the behaviour of a pNets system composed of a set of subnets running in parallel. Let $pNet = \ll pNet_i^{i\in I}, S_j^{j\in J}, \overline{SV} \gg$. Its behaviour is the composition of the behaviours of all the subnets by merging the traces together.

We do not put any constraint on the finiteness of the pNets model - a pNets system is able to compose an unbounded number of subnets. But for readability we assume here $I \cup J = [1, n]$, thus n pairs of (st, tr) are used to observe each subnet, working concurrently to contribute to the composition result. Also remark that the internal states and stores (cs, ds) of subnets are not observed. The composition is described by the following definition:

$$\mathbf{beh}(\ll pNet_i^{i\in I}, S_j^{j\in J}, \overline{SV} \gg) =_{df}$$

$$\left(\begin{array}{l} \exists\ st_1, st_1', \ldots, st_n, st_n', tr_1, tr_1', \ldots, tr_n, tr_n', \overline{ds_1}, \overline{ds_1}', \ldots, \overline{ds_n}, \overline{ds_n}'. \\ \quad tr_1 = \ldots = tr_n = tr \wedge st_1 = \ldots = st_n = st \wedge \\ \quad \forall i \in I.\mathbf{beh}(pNet_i)[st_i, st_i', \overline{ds_i}, \overline{ds_i}', tr_i, tr_i'/st, st', \overline{ds}, \overline{ds}', tr, tr'] \wedge \\ \quad \forall j \in J.\mathbf{beh}(S_j)[st_j, st_j', \overline{ds_j}, \overline{ds_j}', tr_j, tr_j'/st, st', \overline{ds}, \overline{ds}', tr, tr'] \wedge \\ \quad Merge \end{array}\right)$$

in which the $Merge$ predicate captures the behaviours of a parallel composition:

$Merge =_{df}$

$$\left(\begin{array}{l} (\forall i \in [1, n].\ st_i' = comp) \Rightarrow st' = comp \wedge \\ (\exists i \in [1, n].\ st_i' = div) \Rightarrow st' = div \wedge \\ \left(\exists i \in [1, n] \cdot \left(\begin{array}{l} st_i' = wait \wedge \\ \forall j \in [1, n] \cdot st_j' \neq div \end{array}\right)\right) \Rightarrow st' = wait \wedge \\ \overline{ds}' = \bigcup_{i\in[1,n]} \overline{ds_i}' \wedge \\ \exists u \in (tr_1' - tr_1 \| \ldots \| tr_n' - tr_n).\ tr' = tr^\frown u \end{array}\right)$$

The status of the composed behaviour is determined by the n parallel components together. The composition terminates if all the processes terminate and diverges as long as one of the processes diverges. Then the composition stays at waiting state if one of the processes waits and none of the others diverges. Finally, the composition of these n traces is produced by the *trace synchronisation* operator $\|$:

Trace synchronisation. This operator takes n arguments $tr'_i - tr_i$, each being a subsequence of arbitrary length of actions of the corresponding subnet. It computes a set of subtraces that $Merge$ will append to the traces of the composed pNet.

case-1 If all the input traces are empty, the result is a set of an empty sequence:

$$\langle\rangle || \ldots || \langle\rangle = \{\langle\rangle\}$$

case-2 If there is a synchronised action (τ in this paper) in the head of one of the input traces, it is automatically visible at the upper level of the pNet, so we directly record this action in the merged traces.

$$= \exists k \in [1..n].\ e_k = \tau \implies$$

$$\langle e_1\rangle^\frown t_1 || \ldots \| \langle e_n\rangle^\frown t_n = \left\{\langle e_k\rangle^\frown l \mid l \in \langle e_1\rangle^\frown t_1 || \ldots \| t_k \| \langle e_{k+1}\rangle^\frown t_{k+1} \| \ldots \| \langle e_n\rangle^\frown t_n\right\}$$

case-3 In all other situations, we need to select one synchronisation vector matching an event group within the set of the first observations of all the n input traces (Definition 5) and then figure out a synchronised event. Remember that a synchronisation vector concerns any (non-empty) subset of the subnets of the current pNet node. Let us denote as **Value** the set of all possible values which can be transmitted by subnets in the whole pNet and $vars(SV)$ the variables of a synchronisation vector SV.

Definition 5 (Events Match).
Given a set of events $\{e_o, e_p, ..., e_q\} \subseteq \{e_1, e_2, ..., e_n\}$, we say that they match if there exists a synchronisation vector $SV = \alpha_l^{l\in L} \to \alpha' \in \overline{SV}$ and a valuation function $\phi = \{x \to \textbf{\textit{Value}} | x \in var(SV)\}$ that lets both $(\alpha_l)^{l\in L}\phi = \{e_o, e_p, \ldots e_q\}$ and $L = \{o, p, \ldots, q\}$ satisfied. We write $\mathsf{EMatch}(SV, \alpha', \phi, e_o, e_p, \ldots, e_q)$.

With this definition, we can complete the definition of trace synchronisation:

$$\mathsf{EMatch}(SV, \alpha', \phi, e_o, e_p, \ldots, e_q) \implies$$

$$\langle e_1\rangle^\frown t_1 || \ldots \| \langle e_n\rangle^\frown t_n = \left\{\langle \alpha'\phi\rangle^\frown l \mid l \in \langle e_1\rangle^\frown t_1 \| \ldots \| t_o \| \langle e_{o+1}\rangle^\frown t_{o+1} \| \right.$$
$$\left. \ldots \| t_p \| \langle e_{p+1}\rangle^\frown t_{p+1} \| \ldots \| t_q \| \langle e_{q+1}\rangle^\frown t_{q+1} \| \ldots \| \langle e_n\rangle^\frown t_n\right\}$$

Example 8. Now we use the pNets example in Fig. 3 with explicit data transmission to present how its denotational semantics is computed by using our definition. In this 2-places buffer, you can see two pLTSs with identical transitions. It is easy to obtain one possible trace for P1, with an arbitrary number of i/o cycles:

$$t_{P1} = \langle i(e_1), o(e_1), i(e_2), o(e_2), i(e_3)\rangle.$$

And below is a corresponding trace of P2.

$t_{P2} = \langle i(e'_1), o(e'_1), i(e'_2)\rangle$.

From the set of synchronisation vectors defined, we can figure out that on the first execution step, P1 receives some value e_1 assigned to its input variable x_1. This uses the first synchronisation vector of the Buffer pNet, and generates the global action $gi(e_1)$. In the next step, P1 emits e_1 of x_1, synchronised with action $i(x_2)$ of P2, thus the x_2 in P2 is assigned the value e_1, using the second vector, and generating action $\tau(e_1)$.

Then we can obtain one trace for the whole pNets by using the trace synchronisation operator. We have omitted the values of variables st and ds, that the reader will easily guess.

$$t_{P1} \,||\, t_{P2} = \langle gi(e_1)\rangle^\frown l_1.$$

where $l_1 \in \langle o(e_1), i(e_2), o(e_2), i(e_3)\rangle || \langle i(e'_1), o(e'_1), i(e'_2)\rangle$.

There is only one choice in l_1, matching the 2^{nd} vector of the Buffer pNet:

$t_{P1}||t_{P2} = \langle gi(e_1), \tau(e_1)\rangle^\frown l_2$.

where $l_2 \in \langle i(e_2), o(e_2), i(e_3)\rangle || \langle o(e'_1), i(e'_2)\rangle$. Now there are two possible choices, either use the first vector with event $i(e_2)$ (yielding $gi(e_2)$), or the third one with $o(e'_1)$ (yielding $go(e_1)$).

Finally the full composition is given by the following regular expression:

$$\begin{aligned} t_{P1} \,||\, t_{P2} = {} & \langle gi(e_1), \tau(e_1), gi(e_2), go(e_1), \tau(e_2), gi(e_3)\rangle \\ & \vee \langle gi(e_1), \tau(e_1), go(e_1), gi(e_2), \tau(e_2), gi(e_3)\rangle, \end{aligned}$$

after which both traces t_{P1} and t_{P2} are exhausted.

4.4 Composition operator

This section explores the behaviour of the composition of two pNets. In order to simplify the notation, we only consider here a composition operator that replaces a hole at the first (top) level of a pNet tree, that is less general than the composition operator in Definition 4.

Let $N_1 = \ll pNet_i^{i\in I_1}, S_j^{j\in J_1}, \overline{SV_1} \gg$, $N_2 = \ll pNet_i^{i\in I_2}, S_j^{j\in J_2}, \overline{SV_2} \gg$. The pNet composition $N_1[N_2]_{ho}$ indicates that a pNet N_2 fills a hole indexed ho in N_1.

Now we describe the behaviour of $N_1[N_2]_{ho}$:

$\mathbf{beh}(N_1[N_2]_{ho}) =_{df}$

$$\left(\begin{array}{l} \exists st_1, st'_1, st_2, st'_2, tr_1, tr'_1, tr_2, tr'_2, \overline{ds}_1, \overline{ds}'_1, \overline{ds}_2, \overline{ds}'_2. \\ \quad st_1 = st_2 = st \wedge tr_1 = tr_2 = tr \wedge \\ \quad \mathbf{beh}(N_1)[st_1, st'_1, tr_1, tr'_1, \overline{ds}_1, \overline{ds}'_1/st, st', tr, tr', \overline{ds}, \overline{ds}'] \wedge \\ \quad \mathbf{beh}(N_2)[st_2, st'_2, tr_2, tr'_2, \overline{ds}_2, \overline{ds}'_2/st, st', tr, tr', \overline{ds}, \overline{ds}'] \wedge \\ \quad NM(ho) \end{array}\right)$$

The first four predicates describe the two independent behaviours of pNets N_1 and N_2 being composed (running in parallel in essence). The last predicate

$NM(ho)$ mainly does the merging of the contributed traces of the two behaviour branches for recording the communication, which is defined below.

$NM(ho) =_{df}$

$$\begin{pmatrix} st'_1 = comp \wedge st'_2 = comp \Rightarrow st' = comp \wedge \\ st'_1 = div \vee st'_2 = div \Rightarrow st' = div \wedge \\ st'_1 = wait \wedge st'_2 \neq div \Rightarrow st' = wait \wedge \\ st'_2 = wait \wedge st'_1 \neq div \Rightarrow st' = wait \wedge \\ \overline{ds}' = \overline{ds}'_1 \cup \overline{ds}'_2 \wedge \\ \exists u \in (tr'_1 - tr_1)[tr'_2 - tr_2]_{ho}.\, tr' = tr^\frown u \end{pmatrix}$$

The control state of the composed behaviour of the pNet is determined by the combination of the status of the two pNets, which is similar to parallel composition. The trace of the composition is a member of the set of traces produced by *trace composition* operator $[\,]_{ho}$.

Trace composition. Operator $[\,]_{ho}$ models how to merge two individual traces (under some constraints) of pNets $N1$ and $N2$ into a set of traces of $N_1[N_2]_{ho}$.

case-1 If both input traces are empty, the result is a set of an empty sequence:

$\langle\,\rangle[\langle\,\rangle]_{ho} = \{\langle\,\rangle\}$

case-2 If the trace of the subnet is empty, the result is determined by the first observation of the non-empty trace:

$\langle e\rangle^\frown t[\langle\,\rangle]_{ho} = \{\langle e\rangle^\frown l \mid l \in \{\langle\,\rangle\}\} = \{\langle e\rangle\}$

case-3 In the situation where the inner input trace is not empty, we need to check first whether these two traces match (see Definition 7). Only two matching traces can be merged. Then we find out the first pair of matching events (see Definition 6) from the matched traces respectively and compute the corresponding action for the merged trace.

Let t_1 and t_2 be two traces of N_1 and N_2 respectively.

Definition 6 (Events Match for pNets Composition). *Given a pair of events e_1 and e_2, we say that they are matched for pNets composition if there exists a synchronisation vector $SV = \alpha_l^{l\in L} \rightarrow \alpha' \in \overline{SV_1}$, with $ho \in L$, and a valuation function that lets $\alpha_{ho}\phi_{ho} = e_2$. We have $\alpha'\phi = e_1$ and we define an updated valuation function $\phi' = \phi + \phi_{ho}$ which replaces some of the values defined in ϕ by the ones in ϕ_{ho}. We write $< \alpha', \phi' >= \mathsf{CEMatch}(e_1, e_2, ho)$.*

Definition 7 (Traces Match). *We say that the two traces t_1 and t_2 are matched (denoted as $\mathsf{TMatch}(t_1, t_2, ho)()$ if they satisfy such conditions:*

1) For each element e_2 except synchronised action τ in t_2, there exists an element e_1 in t_1 (where e_1 can be τ) that satisfies $\mathsf{CEMatch}(e_1, e_2, ho)$;

2) Matching pairs of events are ordered consistently: given two such pairs $(t_1[i] = e_1, t_2[i'] = e_2)$ and $(t_1[j] = e_3, t_2[j'] = e_4)$ such that $\mathsf{CEMatch}(e_1, e_2, ho)$ and $\mathsf{CEMatch}(e_3, e_4, ho)$ are satisfied, then $i < j \implies i' < j'$.

Now we present how the $[\,]_{ho}$ operator works under the third case.

Let $t_1 = s_1^\frown\langle e_1\rangle^\frown r_1$ and $t_2 = s_2^\frown\langle e_2\rangle^\frown r_2$, where we have $\mathsf{TMatch}(t_1, t_2)$, $\neg\mathsf{TMatch}(s_1, s_2)$ and $< \alpha', \phi' >= \mathsf{CEMatch}(e_1, e_2)$ all satisfied. Then:

$$s_1^\frown\langle e_1\rangle^\frown r_1[s_2^\frown\langle e_2\rangle^\frown r_2]_{ho} = \{l_1^\frown\langle\alpha'\phi'\rangle^\frown l_2 \,|\, l_1 \in s_1 \,|||\, s_2 \wedge l_2 \in r_1[r_2]_{ho}\}$$

where the shuffle operator $|||$ is defined as:

$$\langle\,\rangle \,|||\, \langle\,\rangle = \{\langle\,\rangle\}; \qquad \langle\,\rangle \,|||\, \langle e_2\rangle^\frown t_2 = \{\langle e_2\rangle^\frown l \,|\, l \in \langle\,\rangle \,|||\, t_2\}$$

$$\langle e_1\rangle^\frown t_1 \,|||\, \langle e_2\rangle^\frown t_2 = \{\langle e_1\rangle^\frown l \,|\, l \in t_1 \,|||\, \langle e_2\rangle^\frown t_2\} \cup \{\langle e_2\rangle^\frown l \,|\, l \in \langle e_1\rangle^\frown t_1 \,|||\, t_2\}$$

Example 9. Now we consider the semantics for Enable(P,P')[Enable(Q,R)]$_{P'}$ expressing $P \gg (Q \gg R)$ that we mentioned in Example 4. The behaviours of the composed pNet is computed from the behaviours of two pNets, which may contain a large number of observations. In order to make the illustration more readable, we select single traces from the sets to show the merging of the traces.

Let t_1 and t_2 be two traces of Enable(P,P') and Enable(Q,R) respectively where $t_1 = \langle\alpha_1, \tau, \alpha_2, \alpha_3\rangle$, $t_2 = \langle\alpha_2, \tau, \alpha_3\rangle$. We have here $\alpha_1 \in Sort(P)$, $\alpha_2 \in Sort(Q)$ and $\alpha_3 \in Sort(R)$.

According to Definition 6, $< \alpha_2, \{a_4{\mapsto}\alpha_2\} >= \mathsf{CEMatch}(\alpha_2, \alpha_2, P')$ and $< \alpha_3, \{a_4{\mapsto}\alpha_3\} >= \mathsf{CEMatch}(\alpha_3, \alpha_3, P')$ are satisfied, both firing the synchronisation vector $\{< -, a_4, r >\!\!\rightarrow a_4\}$ in Figure 2. Also, a_2 and a_3 are well ordered in t_1 and t_2 conforming with Definition 7. Then we can obtain one trace of the newly constructed pNet by merging the two traces above:

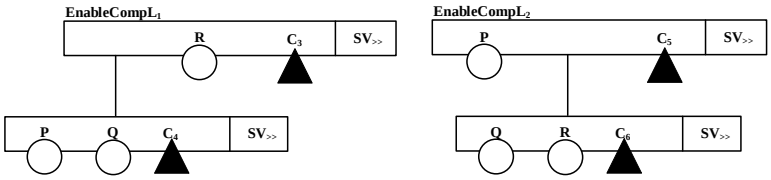
$$\begin{aligned} t_1[t_2]_{P'} &= \langle\alpha_1, \tau, \alpha_2, \alpha_3\rangle[\langle\alpha_2, \tau, \alpha_3\rangle]_{P'} \\ &= \langle\alpha_1, \tau\rangle^\frown\langle\alpha_2\rangle^\frown\langle\alpha_3\rangle[\langle\,\rangle^\frown\langle\alpha_2\rangle^\frown\langle\tau, \alpha_3\rangle]_{P'} = \langle l_1, \alpha_2, l_2\rangle \\ &\quad \text{where } l_1 \in \langle\alpha_1, \tau\rangle|||\langle\,\rangle \text{ and } l_2 \in \langle\alpha_3\rangle[\langle\tau, \alpha_3\rangle]_{P'} = \langle\tau, \alpha_3\rangle. \end{aligned}$$

So we get one trace for Enable(P,P')[Enable(Q,R)]$_{P'}$: $\langle\alpha_1, \tau, \alpha_2, \tau, \alpha_3\rangle$.

5 Algebraic Properties

The main purpose of the formalisation of a programming language is to prove its interesting properties. Most of them are elegantly expressed in the form of algebraic laws and equations. In this section, we explore a set of basic algebraic laws for pNets, based on standard trace semantics: 2 pNets are equivalent if they have same (potentially infinite) set of (finite) traces. pNets being a low level model used to express the semantics of high level languages, we have two categories of properties: general properties about pNets themselves, and specific properties about operators encoded using pNets. We start with a property of the Enable operator of Lotos encoded in Fig. 1.

Associativity of Enable. Consider the two pNets expressing $(P \gg Q) \gg R$ (in Fig 4(a)), built as Enable(P',R)[Enable(P,Q)]$_{P'}$, and $P \gg (Q \gg R)$ (in Fig 4(b)), built as Enable(P,P')[Enable(Q,R)]$_{P'}$ respectively. We would like to prove the associativity law: $(P \gg Q) \gg R = P \gg (Q \gg R)$.



(a) $(P \gg Q) \gg R$ (b) $P \gg (Q \gg R)$

Fig. 4. Two pNets expressing Enable operators

Sketch of the proof: On the left we have traces of Enable(P,Q) and Enable(P',R) whose traces are given below, denoted as t_1 and t_2 respectively.

$t_1 = \langle a_1^{n_1} \rangle \vee t_1 = \langle a_1^{n_1} \rangle^\frown \langle \tau \rangle^\frown \langle a_2^{n_2} \rangle, \forall n_1, n_2 \in \mathbb{N}$

$t_2 = \langle a_2^{n_2} \rangle \vee t_2 = \langle a_2^{n_2} \rangle^\frown \langle \tau \rangle^\frown \langle a_3^{n_3} \rangle, \forall n_2, n_3 \in \mathbb{N}.$

Here, we can see that the traces from each set to be merged complying with the form $s_1^\frown \langle a_2 \rangle^\frown t_1[\langle a_2 \rangle^\frown t_2]$, where $< \alpha_2, \{a_4 \mapsto \alpha_2\} >= \mathsf{CEMatch}(\alpha_2, \alpha_2, P')$ is satisfied. Also, $\mathsf{TMatch}(t_1, t_2, P')$ is satisfied if $n_2 = n_3$. Then we deduce the traces for Enable(P',R)[Enable(P,Q)]$_{P'}$:

$$t_{PQ-R} = \langle a_1^{n_1} \rangle \vee t_{PQ-R} = \langle a_1^{n_1} \rangle^\frown \langle \tau \rangle^\frown \langle a_2^{n_2} \rangle \vee t_{PQ-R} = \langle a_1^{n_1} \rangle^\frown \langle \tau \rangle^\frown \langle a_2^{n_2} \rangle^\frown \langle \tau \rangle^\frown \langle a_3^{n_3} \rangle$$

and symmetrically for Enable(P,P')[Enable(Q,R)]$_{P'}$:

$$t_{P-QR} = \langle a_1^{n_1} \rangle \vee t_{P-QR} = \langle a_1^{n_1} \rangle^\frown \langle \tau \rangle^\frown \langle a_2^{n_2} \rangle \vee t_{P-QR} = \langle a_1^{n_1} \rangle^\frown \langle \tau \rangle^\frown \langle a_2^{n_2} \rangle^\frown \langle \tau \rangle^\frown \langle a_3^{n_3} \rangle$$

Thus, we can say that the behaviours of $(P \gg Q) \gg R)$ and $(P \gg (Q \gg R))$ are equivalent based on traces, and conclude that $(P \gg Q) \gg R = P \gg (Q \gg R)$ is satisfied. □

Now we list a small number of typical laws that can be proved in a similar manner. For most of them, the proofs are straightforward, but long and tedious. Writing them formally requires a lot of notations that we have not introduced in the paper, and we will not do it here.

Symmetry/permutation of pNet nodes. The pNet node as a general parallel operator is symmetric.

For a pNet $\ll pNet_i^{i \in I}, S_j^{j \in J}, SV_k^{k \in K} \gg$, we assume that $I \cup J = [1, n]$ and we abstract each subnet as a process P. Then we alter the pNet node as:$\ll P_1, \ldots, P_n, SV_k^{k \in K} \gg$. It is easy to get

$$\ll P_1, \ldots, P_n, SV_k^{k \in K} \gg = \ll P_{\pi(1)}, \ldots, P_{\pi(n)}, SV_k'^{k \in K} \gg$$

where the structure for each SV_k is $< P_1, \ldots, P_n >$, the structure for each SV_k' is $< P_{\pi(1)}, \ldots, P_{\pi(n)} >$, and π is a permutation on $[1, n]$.

Guarded Choice. We introduce a concept of guarded choice, which enriches the language to support the algebraic laws. The guarded choice is expressed in the form:

$$\{h_1 \to P_1\} [\!] \ldots [\!] \{h_n \to P_n\}.$$

Each element $h \to P$ of the guarded choice is a guarded component, where h can be a guard in the form of either $\alpha(?x, e)$ or τ. After one of the h_i is performed or fired, the subsequent process is P_i.

(**par**-2) Let $P_g = \{a_g \to P'_g\}[\![]\!]_{i \in I_g}\{\tau \to P'_{ig}\}$,

where $g \in [1, n]$ and a_g is the action that will be synchronised with others according to the synchronisation vectors while τ_{ig} cannot be further synchronised. We here put an index to τ to make it easy to be extended to be in a more flexible form. Then we have:

$$
\begin{aligned}
&\ll P_1, \ldots, P_g, P_{g+1}, \ldots, P_n, \overline{SV} \gg \\
&= \{\Updownarrow (a_{x_1}, a_{y_1}, ..., a_{z_1})^{SV_1} \to \ll P_1, ..., P'_{x_1}, ..., P'_{y_1}, \ldots, P'_{z_1}, ..., P_n, \overline{SV} \gg\} \\
&\quad \ldots \\
&[\![\{\Updownarrow (a_{x_m}, a_{y_m}, ..., a_{z_m})^{SV_m} \to \ll P_1, ..., P'_{x_m}, ..., P'_{y_m}, ..., P'_{z_m}, ..., P_n, \overline{SV} \gg\} \\
&[\![]\!]_{i \in I_1}\{\tau_{i1} \to \ll P'_{i1}, P_2 ..., P_n, \overline{SV} \gg\} \\
&\quad \ldots \\
&[\![]\!]_{i \in I_n}\{\tau_{in} \to \ll P_1, ..., P_{i(n-1)}, P'_{in}, \overline{SV} \gg\}
\end{aligned}
$$

where $Card(\{x_1, y_1, ..., z_1\})^2 + ... + Card(\{x_m, y_m, ..., z_m\}) = n$ and $\{x_1, y_1, \ldots, z_1\} \cap \ldots \cap \{x_m, y_m, \ldots, z_m\} = \emptyset$. The notation $\Updownarrow (a_x, a_y, \ldots, a_z)^{SV}$ is used to indicate that the parallel execution of all the actions $a_x, a_y, \ldots, a_z$ can trigger a synchronisation presented by SV, of which the result can be either an action to be synchronised or one that cannot be further synchronised.

Identity operator. A pNet with only one hole indexed 0 of sort S is the identity of pNet composition:

(**ncomp**-1) $I_s[N]_0 = N$ where $I_s := \ll \emptyset, (0 \mapsto S), \{(0 \mapsto a) \to a | a \in S\} \gg$

Composition operator. If N_2 and N_3 instantiate two holes in N_1 respectively meaning that $Sort(N_2) \subseteq Sort(S_{ho_1})$ and $Sort(N_3) \subseteq Sort(S_{ho_2})$ are satisfied, then we have:

(**ncomp**-2) $(N_1[N_2]_{ho_1})[N_3]_{ho_2} = (N_1[N_3]_{ho_2})[N_2]_{ho_1}$

Proof. Sketch: expanding the definition of $\mathbf{beh}((N_1[N_2]_{h0})$, and removing parts that are trivialy equal, the equation boils down to (here, *obs* stands for the set of observational variables we defined for our semantics: $\{st, tr, \overline{ds}\}$):

$$
\begin{aligned}
LHS = \; & \exists obs_1, obs'_1, obs_2, obs'_2.\, obs_1 = obs_2 = obs \wedge \\
& \left(\begin{array}{l} \exists obs_1, obs'_1, obs_2, obs'_2. \\ \quad \mathbf{beh}(N_1)[obs_1, obs'_1/obs, obs'] \wedge \\ \quad \mathbf{beh}(N_2)[obs_2, obs'_2/obs, obs'] \wedge \\ \quad \exists u \in (tr'_1 - tr_1)[tr'_2 - tr_2]_{ho_1}.\, tr' = tr^\frown u \end{array} \right) [obs_1, obs'_1/obs, obs'] \wedge \\
& \mathbf{beh}(N_3)[obs_2, obs'_2/obs, obs'] \wedge \\
& \exists u \in (tr'_1 - tr_1)[tr'_2 - tr_2]_{ho_2}.\, tr' = tr^\frown u
\end{aligned}
$$

[2] $Card(A)$ returns the number of elements in the set A

We suppose that any two traces we select to merge are matched according to Definition 7, thus we can always find pairs of events that are matched. If the two pairs of matched events (e, e_2) and (e', e_3) (e_2 and e_3 are events in the trace of N_2 and N_3 respectively, and e and e' are events in the trace they are to be merged.) does not fire the same synchronisation vector, it does not matter which trace merge with the trace of N_1 first. Otherwise, the event $\alpha'\phi'$ recorded in the composed the trace is determined by both e_2 and e_3. Then we have the variables commuted:

$$
\begin{aligned}
& \exists obs_1, obs_1', obs_2, obs_2'.\, obs_1 = obs_2 = obs \wedge \\
= & \left(\begin{array}{l} \exists obs_1, obs_1', obs_2, obs_2'. \\ \quad \mathbf{beh}(N_1)[obs_1, obs_1'/obs, obs'] \wedge \\ \quad \mathbf{beh}(N_3)[obs_2, obs_2'/obs, obs'] \wedge \\ \quad \exists u \in (tr_1' - tr_1)[tr_2' - tr_2]_{ho_2}.\, tr' = tr^\frown u \end{array} \right) [obs_1, obs_1'/obs, obs'] \wedge \\
& \mathbf{beh}(N_2)[obs_2, obs_2'/obs, obs'] \wedge \\
& \exists u \in (tr_1' - tr_1)[tr_2' - tr_2]_{ho_1}.\, tr' = tr^\frown u \\
= & \; RHS
\end{aligned}
$$

In a similar way, if N_2 instantiates one hole in N_1 and one of the holes in N_2 is instantiated by N_3 indicating that $Sort(N_3) \subseteq Sort(S_{ho_2}) \subseteq Sort(S_{ho_1})$ is satisfied, then:

(**ncomp**-3) $\quad N_1[N_2[N_3]_{ho_2}]_{ho_1} = (N_1[N_2]_{ho_1})[N_3]_{ho_2}$

6 Conclusions

In this paper we have formalized a denotational semantics for Parameterised Networks of Processes. A pNet node is considered as an operator composing a number of subnets running in parallel, which ensures the model's flexibility in expressing various operators, and we do not introduce any specific parallel operator to weaken this feature in pNets model. In our semantics, the subnets in the pNets are viewed as processes and the behaviours of each process are investigated by the execution of a subnet. A trace has been introduced to record the interactions among all the subnets in a pNets system. We have investigated the behaviours of subnets including holes. Then the behaviour of a pNet system can be achieved by merging the behaviours of a set of subnets. A set of algebraic laws on both parallel composition and pNet composition has been achieved based on the denotational semantics.

For the future, we plan to continue our formalisation of pNet systems. One aspect of our future work is to explore the algebraic semantics of the pNets model and study the relations among the three semantics: denotational semantics, operational semantics and algebraic semantics. Moreover, the pNets model can be extended by adding other features such as time issues or probabilities. And it is challenging to explore the semantics with these features.

Acknowledgment

This work was partially funded by the Associated Team FM4CPS between INRIA and ECNU, Shanghai. It was also supported by the Danish National Research Foundation and the National Natural Science Foundation of China (No. 61361136002) for the Danish-Chinese Center for Cyber Physical Systems, National Natural Science Foundation of China (Grant No. 61321064) and Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (No. ZF1213).

References

1. Arnold, A.: Finite transition systems - semantics of communicating systems. Prentice Hall international series in computer science, Prentice Hall (1994)
2. Barros, T., Ameur-Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural models for distributed fractal components. Annales des Télécommunications 64(1-2), 25–43 (2009)
3. Baude, F., Caromel, D., Dalmasso, C., Danelutto, M., Getov, V., Henrio, L., Pérez, C.: GCM: a grid extension to fractal for autonomous distributed components. Annales des Télécommunications 64(1-2), 5–24 (2009)
4. Henrio, L., Madelaine, E., Zhang, M.: pnets: An expressive model for parameterised networks of processes. In: 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015. pp. 492–496 (2015)
5. Henrio, L., Madelaine, E., Zhang, M.: A theory for the composition of concurrent processes. In: 36th IFIP Int. Conference on Formal Techniques for Distributed Objects, Components and Systems. Springer-Verlag (June 2016)
6. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall International Series in Computer Science (1998)
7. Koymans, R., Shyamasundar, R., de Roever, W., Gerth, R., Arun-Kumar, S.: Compositional semantics for real-time distributed computing. Information and Computation 79(3), 210 – 256 (1988)
8. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
9. Nierstrasz, O.: Piccola - A small composition language. In: Object-Oriented Technology, ECOOP'99 Workshop Reader, ECOOP'99 Workshops, Panels, and Posters, Lisbon, Portugal, June 14-18, 1999, Proceedings. p. 317 (1999)
10. Rajan, A., Bavan, S., Abeysinghe, G.: Semantics for a distributed programming language using sacs and weakest pre-conditions. In: Advanced Computing and Communications, 2006. ADCOM 2006. International Conference on. pp. 434–439 (Dec 2006)
11. Schmitt, A., Stefani, J.: The kell calculus: A family of higher-order distributed process calculi. In: Global Computing, IST/FET International Workshop, GC 2004, Rovereto, Italy, March 9-12, 2004, Revised Selected Papers. pp. 146–178 (2004)