Using SMT engine to generate Symbolic Automata*

Xudong Qin[3] Eric Madelaine[1,2]

[1] Univ. of Nice Sophia Antipolis, CNRS, UMR 7271, 06900 Sophia Antipolis, France
[2] INRIA Sophia Antipolis Méditérannée, BP 93, 06902 Sophia Antipolis, France
[3] Shanghai Key Laboratory of Trustworthy Computing, ECNU, China

Abstract. We implement the symbolic semantics of open pNets using their so-called "Open Automaton" behavioural semantics. This involve building predicates expressing the synchronisation conditions allowing some combination of events in the pNet system. These predicates are typically built using first order logic, plus some predicates specific of particular action algebras. To reduce the complexity of the generated open automata, we use the Z3 SMT engine to check satisfiability of the predicates, and prune the state space..

1 Introduction

imported from FORTE'16,

In the nineties, several works extended the basic behavioural models based on labelled transition systems to address value-passing or parameterised systems, using various symbolic encodings of the transitions [1–4]. In [4], H.M. Lin addressed value-passing calculi, for which he developed a symbolic behavioural semantics, and proved algebraic properties. Separately J. Rathke [5] defined another symbolic semantics for a parameterised broadcast calculus, together with strong and weak bisimulation equivalences, and developed a symbolic model-checker based on a tableau method for these processes. 30 years later, no practical verification approach and no verification platform are using this kind of approaches to provide proof methods for value-passing processes or open process expressions.

Context **[TODO: Summary of previous works on pNets and open pNets => 1/2 page.**
Longterm goals:
- open pNets to represent operators and program skeletons (ref to [?])
- compute semantics in term of open transitions
- compute equivalence (bisimulation of open automata)
- model-check properties of open systems]

* This work was partially funded by the Associated Team FM4CPS between INRIA and ECNU, Shanghai

Contribution

Related works

Structure.

2 Running Examples

Several publications [**?**,**?**] already have introduced many examples of pNets, encoding operators of various classical process algebras, or more complex synchronisation structures in distributed or parallel languages. In this section, we use two process expressions, respectively from CCS and from Lotos, to illustrate different features of pNets. They will serve as running examples in the whole paper.

2.1 Operators from Classical Process Algebra

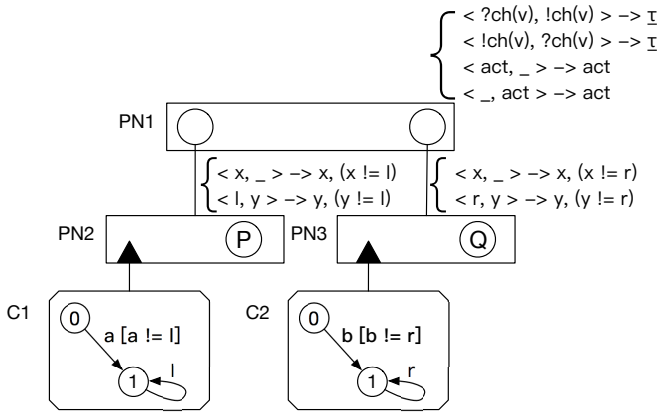

Fig. 1. The pNet encodings for CCS formula

First example we choose is a formula $a.P||b.Q$, composing prefix and parallel operators of value-passing CCS, and containing two process variables (holes) P and Q. In Fig. 1, we show the encoding of this CCS formula. The pNet has a tree-like structure.

The root node of the tree PN1 is the top level of the pNet structure. It acts as the parallel operator. There are two subnets, noted PN2 and PN3, each representing one of the prefix sub-expression. Their behaviors are synchronized in their parent PN1 according to a set of synchronization vectors, synchronizing input and output action from these two subnets to give out a silent action τ or

just making the action become visible on the top level when one of the subnets is working alone. The vector syntax is typically "<act, _ > -> act", in which "_" means that the corresponding subnet is not involved in this synchronisation. **[TODO:The vectors for communication are wrong, we cannot use ?ch(v) and !ch(v), see the PDP'15 and Forte'16 papers.]** Remark that these 4 synchronisation vectors correspond faithfully to the usual SOS rules of CCS operational semantics.

Each of the subnets act as one of the prefix operator. In each of them there is a parameterised labelled transition system (pLTS) and a hole. The pLTS is a controller managing the changes of state of the pNet; consider C1, from its initial state the only possible transition performs the action variable a; once in state 1, the controler can perform (infinitely often) action l that is a constant action local to C1 (that must be different from any other actino of the whole system). In the node PN2, there are 2 synchronisation vectors, the first one transmits to the upper level the action a (and thanks to C1, will only be activated one time); the second one transmit any action of the hole P, but only when C1 can perform l.

It should be easy to see that this pNet system encodes properly the behavioural semantics of the corresponding CCS expression. Remarkk that it is built in a systematic structural way, one pNet node encoding each CCS operator. And it is parameterised both at the level of actions variables (occuring within the controllers and the synchronisation vectors), and at the level of process parameters (holes).

2.2 Open pNet with Assignments in Leaves

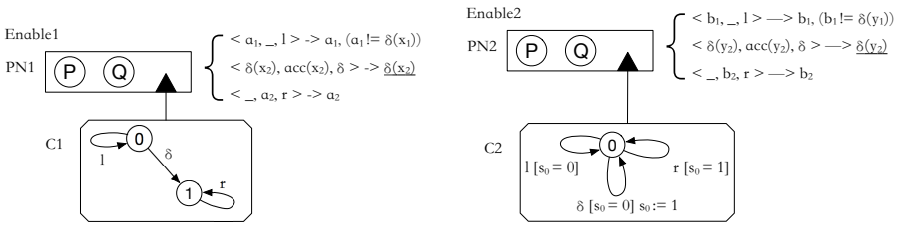

Fig. 2. Two pNet encodings for the Lotos Enable operator P»Q

In the second running example, we want to illustrate the use of local variables in the controller pLTSs. This example is based on the Enable operator of the LOTOS language. For the readers not familiar with Lotos, the pNet encodingss here should be sufficient to understand the Enable operator semantics.

In fact we give two encodings of this operator: the first one is "state oriented", in the sense that the controler has two states, indicating where the control point is in the P»Q process expression, either in P or in Q. The transition between

these occurs when process P performs a $\delta(x)$ action, transfering the control to Q, and sending a value x at the same time. Process Q must be ready to "accept" this value through an $acc(x)$ action. The controler C1, and the 3 synchronisation vectors of the Enable1 pNet should be easy to understand. Note that the first vector, transmitting an action $a1$ of P, can only occur if $a1$ is not a δ.

The second encoding is "data oriented", meaning that the state of the node controller is encoded in the value of the pLTS state variable(s). Here we have a single state variable s_0, initialised as 0. Its value is "0" when the control is on P, and "1" when the control is on Q. The transitions in the pLTS have the more general syntax: "<action-expr> [guard] {sequence of assignments}". Guards and assignments can only use the values of variables of their source state and values from input variables in the action expression; and a transition can only assign variables of its target state. The second vector of the node "Enable2" also shows a $\underline{\delta(y_2)}$ result action, where the underlined action stands for an internal (already synchronised) action, that cannot be further synchronised at upper levels. This is a straighforward generalisation of the notion of internal actions, that will be convenient for observing internal events during model-checking.

3 Parameterised Networks (pNets): definition

Keep minimum defs here too be independent from FORTE

This section introduces pNets and the notations we will use in this paper. Then it gives the formal definition of pNet structures, together with an operational semantics for open pNets.

pNets are tree-like structures, where the leaves are either *parameterised labelled transition systems (pLTSs)*, expressing the behaviour of basic processes, or *holes*, used as placeholders for unknown processes, of which we only specify the set of possible actions, this set is named the *sort*. 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.

Notations. We extensively use indexed structures over some countable indexed sets, which are equivalent to mappings over the countable set. $a_i^{i\in I}$ denotes 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. E.g., $a^{i\in\{3\}}$ is the mapping with a single entry a at index 3 ; abbreviated $(3{\mapsto}a)$ in the following. When this is not ambiguous, we shall use notations for sets, and typically write "indexed set over I" when formally we should speak of multisets, and write $x \in a_i^{i\in I}$ to mean $\exists i \in I.\, x = a_i$. An empty family is denoted $\emptyset$. We denote classically $\overline{a}$ a family when the indexing set is not meaningful. $\uplus$ is the disjoint union on indexed sets.

Term algebra. Our models rely on a notion of parameterised actions, that are symbolic expressions using data types and variables. As our model aims at encoding the low-level behaviour of possibly very different programming languages, we

do not want to impose one specific algebra for denoting actions, nor any specific communication mechanism. So we leave unspecified the constructors of the algebra that will allow building expressions and actions. 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.

Formally, we assume the existence of a term algebra $\mathcal{T}_{\Sigma,\mathcal{P}}$, where Σ is the signature of the data and action constructors, and $\mathcal{P}$ a set of variables. Within $\mathcal{T}_{\Sigma,\mathcal{P}}$, we distinguish a set of data expressions $\mathcal{E}_{\mathcal{P}}$, including a set of boolean expressions $\mathcal{B}_{\mathcal{P}}$ ($\mathcal{B}_{\mathcal{P}} \subseteq \mathcal{E}_{\mathcal{P}}$). On top of $\mathcal{E}_{\mathcal{P}}$ we build the action algebra $\mathcal{A}_{\mathcal{P}}$, with $\mathcal{A}_P \subseteq \mathcal{T}_{\mathcal{P}}, \mathcal{E}_P \cap \mathcal{A}_P = \emptyset$; 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.

pNets can encode naturally the notion of input actions in value-passing CCS [6] or of usual point-to-point message passing calculi, but it also allows for more general mechanisms, like gate negociation 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). We can also use more complex action structure such as Meije-SCCS action monoids, like in $a.b$, $a^{f(n)}$ (see [1]). The expressiveness of the synchronisation constructs will depend on the action algebra.

3.1 The (open) pNets Core Model

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. Transmitting variable values from one state to the other can be 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:

$\alpha \in \mathcal{A} ::=$	$a(p_1, \ldots, p_n)$	action terms
$p_i ::=$	$?x \mid Expr$	parameters (input variable or expression)
$Expr ::=$	$Value \mid x \mid op(Expr_1, .., Expr_n)$	Expressions

The input variables in an action term are those marked with a *?*. We additionally suppose that each input variable does not appear somewhere else in the same action term: $p_i = ?x \Rightarrow \forall j \neq i.\, x \notin vars(p_j)$

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* $iv(\alpha) \subseteq vars(s')$*,* $vars(\alpha) \setminus iv(\alpha) \subseteq vars(s)$*,* $vars(e_b) \subseteq vars(s')$*, and* $\forall j \in J.\ vars(e_j) \subseteq vars(s) \wedge x_j \in vars(s')$*.*

Now we define pNet nodes, as constructors for hierarchical behavioural structures. A pNet has a set of sub-pNets that can be either pNets or pLTSs, and a set of Holes, playing the role of process parameters.

A composite pNet consists of a set of sub-pNets exposing a set of actions, each of them triggering internal actions in each of the sub-pNets. The synchronisation between global actions and internal actions is given by *synchronisation vectors*: a synchronisation vector synchronises one or several internal actions, and exposes a single resulting global action. Actions involved at the pNet level (in the synchronisation vectors) do not need to distinguish between input and output variables. Action terms for pNets are 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* $\alpha'_k \in \mathcal{A}_{\mathcal{P}}$*,* $I_k \subseteq I$*,* $J_k \subseteq J$*,* $\forall i \in I_k.\ \alpha_i \in \mathrm{Sort}(pNet_i)$*,* $\forall j \in J_k.\ \alpha_j \in S_j$*, and* $vars(\alpha'_k) \subseteq \bigcup_{l \in I_k \uplus J_k} vars(\alpha_l)$*. The global action of a vector* SV_k *is* $\mathrm{Label}(SV_k) = \alpha'_k$*.*

The preceding definition relies on the auxiliary functions below:

Definition 3 (Sorts, Holes, Leaves of pNets).

– *The sort of a pNet is its signature, i.e. the set of actions it can perform. In the definition of sorts, we do not need to distinguish input variables (that specify the dataflow within LTSs), so for computing LTS sorts, we use a substitution operator*[4] *to remove the* input marker *of variables. Formally:*

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

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

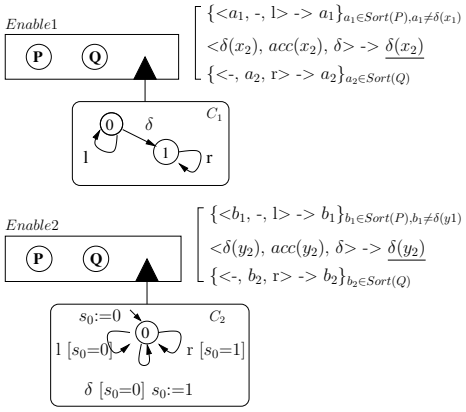

Fig. 3. Two pNet encodings for Enable

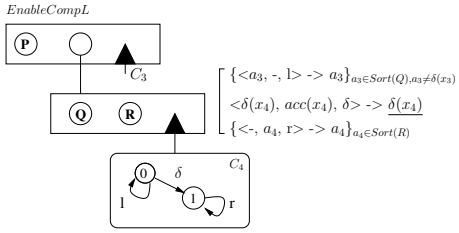

Fig. 4. Composed pNet for "P»(Q»R)"

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

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

– *The set of leaves of a pNet is the set of all pLTSs occurring in the structure, defined inductively as:*

$$\begin{aligned}
&\mathrm{Leaves}(\langle\!\langle S, s_0, \rightarrow \rangle\!\rangle) = \{\langle\!\langle S, s_0, \rightarrow \rangle\!\rangle\} \\
&\mathrm{Leaves}(\langle\!\langle pNet_i^{i\in I}, S_j^{j\in J}, \overline{SV} \rangle\!\rangle) = \bigcup_{i\in I} \mathrm{Leaves}(pNet_i)
\end{aligned}$$

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

4 Operational Semantics for Open pNets

The semantics of open pNets will be defined as an open automaton. An open automaton is an automaton where each transition composes transitions of several LTSs with action of some holes, the transition occurs if some predicates hold, and can involve a set of state modifications.

Definition 4 (Open transitions). *An* open transition *over a set* $(S_i, s_{0i}, \rightarrow_i)^{i\in I}$ *of LTSs, a set* J *of holes with sorts* $Sort_j^{j\in J}$*, and a set of states* $\mathcal{S}$ *is a structure of the form:*

$$\frac{\{s_i \xrightarrow{a_i}_i s_i'\}^{i\in I}, \{\xrightarrow{b_j}_j\}^{j\in J}, Pred, Post}{s \xrightarrow{v} s'}$$

Where $s, s' \in \mathcal{S}$ and for all $i \in I$, $s_i \xrightarrow{a_i}_i s'_i$ is a transition of the LTS $(S_i, s_{0i}, \rightarrow_i)$, and $\xrightarrow{b_j}_j$ is a transition of the hole j, for any action b_j in the sort $Sort_j$. Pred is a predicate over the different variables of the terms, labels, and states s_i, b_j, s, v. Post is a set of equations that hold after the open transition*, they are represented as a substitution of the form $\{x_k \leftarrow e_k\}^{k \in K}$ where x_k are variables of s', s'_i, and e_k are expressions over the other variables of the open transition.*

Example 1. An open-transition. The `EnableCompL` pNet of Fig. 4 has 2 controllers and 2 holes. One of its possible open-transition is:

$$OT_2 = \begin{array}{c} 0 \xrightarrow{\delta}_{C_3} 1 \quad 0 \xrightarrow{l}_{C_4} 0 \quad \xrightarrow{\delta(x4)}_P \quad \xrightarrow{accept(x4)}_Q \\ \hline A1_0 \xrightarrow{\delta(x4)} A1_1 \end{array}$$

Definition 5 (Open automaton). *An* open automaton *is a structure $A =< LTS_i^{i \in I}, J, \mathcal{S}, s_0, \mathcal{T} >$ where:*

- *I and J are sets of indices,*
- *$LTS_i^{i \in I}$ is a family of LTSs,*
- *$\mathcal{S}$ is a set of states and s_0 an initial state among $\mathcal{S}$,*
- *$\mathcal{T}$ is a set of open transitions and for each $t \in \mathcal{T}$ there exist I', J' with $I' \subseteq I$, $J' \subseteq J$, such that t is an open transition over $LTS_i^{i \in I'}$, J', and $\mathcal{S}$.*

Definition 6 (States of open pNets). *A state of an open pNet is a tuple (not necessarily finite) of the states of its leaves (in which we denote tuples in structured states as $\triangleleft \ldots \triangleright$ for better readability).*

For any pNet p, let $\overline{Leaves} = \langle\!\langle S_i, s_{i0}, \rightarrow_i \rangle\!\rangle^{i \in L}$ be the set of pLTS at its leaves, then $States(p) = \{\triangleleft s_i^{i \in L} \triangleright \,|\forall i \in L. s_i \in S_i\}$. A pLTS being its own single leave: $States(\langle\!\langle S, s_0, \rightarrow \rangle\!\rangle) = \{\triangleleft s \triangleright \,| s \in S\}$.

The initial state is defined as: $InitState(p) = \triangleleft {s_{i0}}^{i \in L} \triangleright$.

Predicates: Let $\langle\!\langle \overline{pNet}, \overline{S}, SV_k^{k \in K} \rangle\!\rangle$ be a pNet. Consider a synchronisation vector SV_k, for $k \in K$. We define a predicate *Pred* relating the actions of the involved sub-pNets and the resulting actions. This predicate verifies:

$$Pred(SV_k, a_i^{i \in I}, b_j^{j \in J}, v) \Leftrightarrow \begin{array}{l} \exists (a'_i)^{i \in I}, (b'_j)^{j \in J}, v'.\, SV_k = (a'_i)^{i \in I}, (b'_j)^{j \in J} \rightarrow v' \\ \quad \wedge \forall i \in I.\, a_i = a'_i \wedge \forall j \in J.\, b_j = b'_j \wedge v = v' \end{array}$$

In any other case (if the action families do not match or if there is no valuation of variables such that the above formula can be ensured) the predicate is undefined.

This definition is not constructive but it is easy to build the predicate constructively by brute-force unification of the sub-pNets actions with the corresponding vector actions, possibly followed by a simplification step.

We build the semantics of open pNets as an open automaton where LTSs are the pLTSs at the leaves of the pNet structure, and the states are given by Definition 6. The open transitions first project the global state into states of the leaves, then apply pLTS transitions on these states, and compose them with the sort of the holes. The semantics regularly instantiates *fresh* variables, and uses a *clone* operator that clones a term replacing each variable with a fresh one.

Definition 7 (Operational semantics of open pNets). *The semantics of a pNet p is an open automaton $A =< Leaves(p), J, \mathcal{S}, s_0, \mathcal{T} >$ where:*

- *J is the indices of the holes: $Holes(p) = H_j^{j \in J}$.*
- *$\overline{\mathcal{S}} = States(p)$ and $s_0 = InitState(p)$*
- *$\mathcal{T}$ is the smallest set of open transitions satisfying the rules below:*

The rule for a pLTS p checks that the guard is verified and transforms assignments into post-conditions:

$$\textbf{Tr1:}\quad \frac{s \xrightarrow{\langle \alpha,\ e_b,\ (x_j := e_j)^{j \in J} \rangle} s' \in \rightarrow}{p = \langle\!\langle S, s_0, \rightarrow \rangle\!\rangle \models \dfrac{\{s \xrightarrow{\alpha}_p s'\}, \emptyset, e_b, \{x_j \leftarrow e_j\}^{j \in J}}{\lhd s \rhd \xrightarrow{\alpha} \lhd s' \rhd}}$$

The second rule deals with pNet nodes: for each possible synchronisation vector applicable to the rule subject, the premisses include one open transition *for each sub-pNet involved, one possible* action *for each Hole involved, and the predicate relating these with the resulting action of the vector. A key to understand this rule is that the open transitions are expressed in terms of the leaves and holes of the pNet structure, i.e. a flatten view of the pNet: e.g. L is the index set of the Leaves, L_k the index set of the leaves of one subnet, so all L_k are disjoint subsets of L. Thus the states in the open transitions, at each level, are tuples including states of all the leaves of the pNet, not only those involved in the chosen synchronisation vector.*

Tr2:

$$\frac{\begin{array}{c} k \in K \qquad SV = clone(SV_k) = \alpha_m^{m \in I_k \uplus J_k} \rightarrow \alpha'_k \qquad Leaves(p) = pLTS_l^{l \in L} \\ \forall m \in I_k . pNet_m \models \dfrac{\{s_i \xrightarrow{a_i}_i s'_i\}^{i \in I'_m}, \{\xrightarrow{b_j}_j\}^{j \in J'_m}, Pred_m, Post_m}{\lhd s_i^{i \in L_m} \rhd \xrightarrow{v_m} \lhd s'_i{}^{i \in L_m} \rhd} \qquad I' = \biguplus_{m \in I_k} I'_m \\ J' = \biguplus_{m \in I_k} J'_m \uplus J_k \qquad Pred = \bigwedge_{m \in I_k} Pred_m \wedge Pred(SV, a_i^{i \in I_k}, b_j^{j \in J_k}, v) \\ \forall j \in J_k . \texttt{fresh}(b_j) \qquad \texttt{fresh}(v) \qquad \forall i \in L \backslash I' . s'_i = s_i \end{array}}{p = \langle\!\langle pNet_i^{i \in I}, S_j^{j \in J}, SV_k^{k \in K} \rangle\!\rangle \models \dfrac{\{s_i \xrightarrow{a_i}_i s'_i\}^{i \in I'}, \{\xrightarrow{b_j}_j\}^{j \in J'}, Pred, \uplus_{m \in I_k} Post_m}{\lhd s_i^{i \in L} \rhd \xrightarrow{v} \lhd s'_i{}^{i \in L} \rhd}}$$

Example 2. Using the operational rules to compute open-transitions In Fig. 5 we show the deduction tree used to construct and prove the open transition OT_2 of `EnableCompL` (see example page 8). The rule uses TR1 for the δ transition of C_3, for the l transition of C_4, then combines the result using the a_4 vector of the bottom pNet node, and the $\underline{\delta(x)}$ vector of the top node.

Note that while the scenario above is expressed as a single instantiation of the possible behaviours, the constructions below are kept symbolic, and each open-transition deduced expresses a whole family of behaviours, for any possible values of the variables.

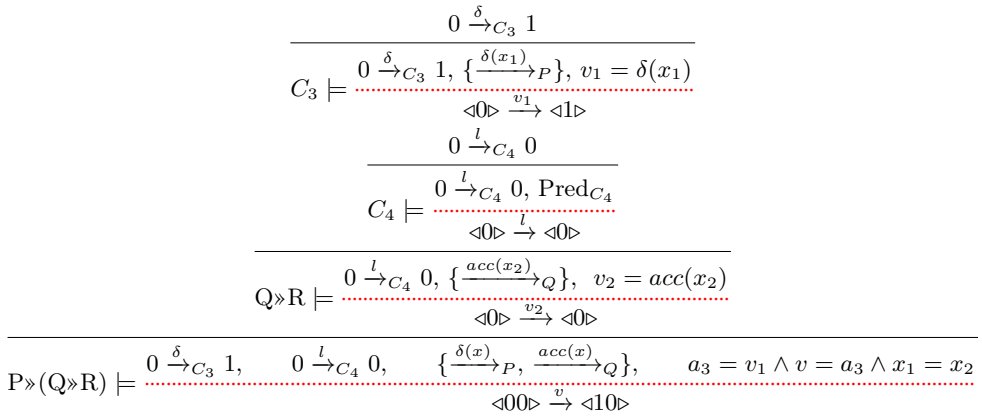

Fig. 5. Proof of transition OT_2 (with interaction of processes P and Q) for "P»(Q»R)"

Variable management. The variables in each synchronisation vector are considered local: for a given pNet expression, we must have fresh local variables for each occurrence of a vector (= each time we instantiate rule Tr2). Similarly the state variables of each copy of a given pLTS in the system, must be distinct, and those created for each application of Tr2 have to be fresh and all distinct. This will be implemented within the open-automaton generation algorithm, e.g. using name generation using a global counter as a suffix.

4.1 Presentation of algebra

[TODO:Eric: write an introductory paragraph]

Before the encoding, the presentation of the pNet should be given first. We can divide the presentation into two parts:

- Sorts: Constants sets of the algebra, types of the data and actions.
- Operators: Operators comes from other algebras, constructors for the parameterized actions.

Sorts Ownership of the variables and constants in a pNet acting as actions or data is probably various. Sorts are represented by their name. The sort of actions is built by the constant actions. For example, let $CCSAct$ be the only set of actions of the CCS running example. Obviously, $CCSAct$ contains constant actions l, r in the pLTSs. Sorts here act as sets of actions for pNet, pLTS or holes. Intuitively, Sorts of data and actions are completely independent. Sorts of the data like a type of them could be integer, boolean or a customized type. Sorts for integer and boolean are implicitly declared.

Operators Usually we need introduce the expressions from other algebra when encoding the algebra into pNets. So that is needed to know about the input and

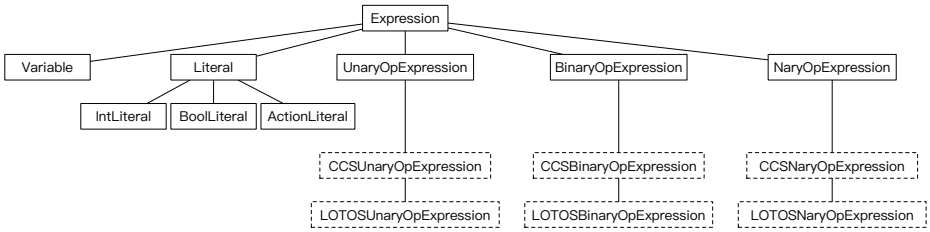

Fig. 6. The architecture of Expression class

output of the operators used in the expressions from those algebra. The input and output should belong to the sorts on the pNet. When there are k sorts in this pNet, the structure of a n-ary operator is vector as $< Sort_{i_1}, ..., Sort_{i_n} > \rightarrow Sort_j, i_1, ..., i_n, j \in [0..k]$.

Extension of other algebras Variable, constant are treated as a kind of expression together with the expression using operator. Operators can be categorized into unary operator, binary operator and N-ary operator, that make it possible to present operator from other algebras. In Fig .6, we show the Expression class we improved to enable user to extend other algebras in the pNet. There are classes implementing three kinds of expression extend the original Expression class. From these three classes, more classes are extended using the operators coming from other algebra such as CCS and LOTOS

4.2 Computing and using open automata

In this section we present an algorithm to construct the open automaton representing the behaviour of an open pNet, and we prove that under reasonable conditions this automaton is finite.

[TODO:Refine the algorithm description]

Algorithm 1 (Behavioural semantics of open pNets: Sketch) *This is a standard residual algorithm over a set of open-automaton states, but where transitions are open transitions constructively "proven" by deduction trees.*

1) Start with a set of unexplored states containing the initial state of the automaton, and an empty set of explored states.

2) While there are unexplored states:

2a) pick one state from the unexplored set and add it to the explored set. From this state build all possible deduction trees by application of the structural rules Tr1 and Tr2, using all applicable combinations of synchronisation vectors.

2b) For each of the obtained deduction trees, extract the resulting open-transition, with its predicate and Post assignments by exploring the structure of the pNet.

2c) The predicate is submitted to Z3 for checking satisfiability. If it is NOT stisfiable, the resulting OT is discarded. This will minimize the number of resulting transitions, and potentialy prune the search-space.

For each open-transition, add the transition in the outgoing transitions of the current state, and add the resulting state in the unexplored set if it is not already in the explored set.

To have some practical interest, it is important to know when this algorithm terminates. The following theorem shows that an open-pNet with finite synchronisation sets, finitely many leaves and holes, and each pLTS at leaves having a finite number of states and (symbolic) transitions, has a finite automaton:

Theorem 2 (Finiteness of open-automata.).
Given an open pNet $\langle\!\langle \overline{pNet}, \overline{S}, SV_k^{k\in K} \rangle\!\rangle$ *with leaves* $pLTS_i^{i\in L}$ *and holes* $Hole_j^{j\in J}$*, if the sets* L *and* J *are finite, if the synchronisation vectors of all pNets included in* $\langle\!\langle \overline{pNet}, \overline{S}, SV_k^{k\in K} \rangle\!\rangle$ *are finite, and if* $\forall i \in L.\ finite(states(pLTS_i))$ *and* $pLTS_i$ *has a finite number of state variables, then Algorithm 1 terminates and produces an open automaton* $\mathcal{T}$ *with finitely many states and transitions.*

5 Implementation

VerCors platform uses pNet as the intermediate language for some language or graphical formalism to be translated into both input for a checker and for generating code automatically to ease the specification from non-expert users[**?**], compiling it from those language or graphical formalism in the pNet API. We compute the open automaton of an open pNet also in that pNet API directly implementing the sketch showed before. However, we made some modification, merging the step 2a-2b and applying Tr2 with the premisses from subnets to generate open transitions, application of Tr1 is still conducted in a simple way. At same time, we apply the fresh function to rename all the variable to make them unique and readable. Besides that, we also have a management of assignments throughout the whole computation. Satisfiability check is done only at top level of the deduction tree construction. An alternative would be to submit the satisfiability check to the SMT solver at each level of the tree construction, potentially reducing the overall number of combinations. But the submission to the SMT engine is costly, and more complexity analysis is required before deciding if this would be worthwhile.

[TODO:Xudong: Missing a general summary before going into details. You can get inspiration from the following (reasonable level of abstraction):]

=> Context = use the existing software architecture from VCE (ref to [**?**] paper)
=> Current prototype: programatic creation of pNet objects, through the pNet API in VCE. On the middle term, this should be compiled from some language or graphical formalism.
=> Open transitions are computed as a direct implementation of the algorithm above. Steps 2a-2b are merged: at each pNet node, after application of rule Tr2, the resulting OT is built from the premisses, without explicitly constructing the deduction tree.

=> Variable names are generated for all "fresh" and "clone" operations, using structured variable names ensuring uniqueness of fresh names, but also reasonable readability of these names for debugging purposes.
=> Assignments
=> Satisfiability check is done only at top level of the deduction tree construction. An alternative would be to submit the satisfiability check to the SMT solver at each level of the tree construction, potentially reducing the overall number of combinations. But submission to the SMT engine is costly, and more complexity analysis is required before deciding if this would be worthwhile.

5.1 Fresh variable

The variables in each synchronization vector are considered local, so does the variables of an open transition such as its hole behaviors and its result action. So we want rename the variables to make them unique on every occurrence of a vector. Similarly, the name given to the hole behaviors for each hole or the result action for each open transition must be distinct. At the same time, we also hope them still readable. Here we define the fresh variables, the variables renamed with a regular format by the function fresh().

The fresh function is to generate a new name adding a suffix after the original name. The suffix contains three part combined by a colon.

Definition 8 (Fresh variable). *The format of* fresh variable *(renaming) is defined as:*

prefix : tree index : counter

- *prefix is a default internal name for the variable. So far the internal name can be sva (SV action), ra (result action), hb (hole behavior).*
- *tree index is the index of the node in the tree-like structure of the pNet.*
- *counter is the current value of the corresponding counter.*

Prefix avoids the confusion between variables from different structures with the same name by attaching the type of the structure. *Tree index* is given to every node of the pNets to mention which node the variable belongs to as pNets has a tree-like structure. To each node, the tree index is always a number sequence of the tree index of its higher level and the index of the node in this level at the last. *A set of counters* is used to count the current times the SV, Hole, subnet invoked, or the number of possible OT generated to provide an identity.

5.2 Predicate generation

The predicate of the open transition is a conjunction of two parts every time applying Tr2. One part composes the predicate from the subnets and the guard from transitions in sub-pLTS or synchronization vectors. The other part is the

conjunction of equations generating during synchronizing subnets' behaviors and synchronization vectors. Note that it is simple to collect required terms in the first part while in the second part it need to find out all the possible conjunction forms. We present here a algorithm divided in two step. Combining enumerates all the possible combinations of the working status. Matching synchronizes the subnets according to synchronization vectors to generate possible predicates.

Combining We first conduct combining. To each subnet, we enumerate all the possible cases of its working status. The result action of a pLTS or a pNet is the external action of the node shows the working status of it. The hole whose behavior is uncertain doesn't have an exact external presentation of action. We only give a variable to each hole and treat it as hole behavior to suppose working status of the hole. Here should have a constraint that this variable belongs to the sort of hole. In every case, the hole behaviors are unchanged, so we only make the combinations of subnets' open transitions.

Algorithm 1 shows the combining algorithm for enumerating all the possible combinations of open transitions. The algorithm initializes an empty list L_C. Dealing with the list TR containing several sets of open transitions from different subnets, the algorithm only chooses one of sets L_{ot} and does combining on the rest part of TR recursively then get a partial result of combining, L'_C. Since there could exist the case that the subnet is not working, a *null* is added into the L_{ot} to represent it. We get much more combinations after combining ot from L_{ot} with the partial result.

Algorithm 1 Combining

Input: List TR of result open transition sets from the subnets.
Output: The list of all the possible combinations of open transitions L_C.

```
1: Initial an empty combination list L_C;
2: Extract a list of open transitions L_ot from TR;
3: Insert null into L_ot;
4: Get the combinations L'_C of the TR;
5: for each ot ∈ L_ot do
6:     for each C' ∈ L'_C do
7:         Add ot into C' to get a new tuple C;
8:         Add C into L_C;
9:     end for
10: end for
11: return L_C;
```

Matching We now come to the matching using combinations from the previous step, together with hole behaviors and the result action, to match with the synchronization vectors.

Algorithm 2 shows the algorithm matches combination L_C, hole behaviors B with synchronization vectors SV. Every time we choose a synchronization

vector sv from SV for matching, we clone it and use the cloned one sv'. We use the definition declared before to construct all the possible predicates but the algorithm has not checked the correctness of these predicates. It needs a further refinement.

In order to filter a part of mismatches, at any time an action e_1 matching with an element e_2 from synchronization vector, the pair (e_1, e_2) must be checked: if both e_1 and e_2 are *null* or *not null*, then it continues to generate a new term of predicate. Otherwise, it is intuitive that the working status coming from the subnet e_1 is different from what asked by the synchronization vector e_2, then this case is filtered.

Algorithm 2 Matching

Input: The combination of subnets' open transitions L_C; The behaviors of the holes B; The set of synchronization vectors SV.
Output: The list of all the possible open transitions generated L_{ot}.

```
1: Initial an empty result list L_ot;
2: for each C ∈ L_C do
3:     for each sv ∈ SV do
4:         Clone the sv, the result is denoted as sv';
5:         Generate the fresh result action v;
6:         Combine C, B, v as a new tuple < C, B, v >;
7:         for each e_1 ∈< C, B, v > &&e_2 ∈ sv' do
8:             if (e_1 is null && e_2 is not null) || (e_1 is not null && e_2 is null) then
9:                 Skip;
10:            else
11:                Generate the term of predicate t;
12:                Add t into the Pred;
13:            end if
14:        end for
15:        Generate the result open transition ot with Pred;
16:        Add the ot into L_ot;
17:    end for
18: end for
19: return L_ot;
```

5.3 assignment

Management of state variables.
Translating assignments into predicate terms.

5.4 Pruning the unsatisfiable results

Interaction with Z3: pNets module —input–> Z3 module [JAVA API]

translation of presentation

checking satisfiability After we get the result OTs, for each result OT, declare the variables and assertions in the Java API. Conduct checking method and get the result.

[TODO: Xudong: full result of the tool execution on the running example

Open Automata of the CCS example.]

6 Conclusion and Discussion

[TODO:Conclusion: what you have accomplished, clearly state your contributions, and what is delivered at the end of your internship work]

[TODO:Discussion: next steps: 1) what is left as open questions on the algorithm itself, what you have learned (is Z3 useful and efficient, is it the right tool ?). 2) next important goal is bisimulation checking, what is neded for that (formalize the algo, define and implement simplification)]

=> Simplification is not yet implemented. It is not strictly required for the Open Automaton construction, but it will be critical later for predicate comparison in the bisimulation algorithm.

Optionally, simplify the predicate by eliminating the unnecessary intermediate variables, that were produced as fresh local variables of synchronisation vectors, and fresh result variables of intermediate OTs. In the resulting predicate the only significant variables are :
- the input variables of pLTS transitions
- the actions of holes
- the result action at toplevel.

References

1. De Simone, R.: Higher-level synchronising devices in MEIJE-SCCS. Theoretical Computer Science **37** (1985) 245–267
2. Larsen, K.G.: A context dependent equivalence between processes. Theoretical Computer Science **49** (1987) 184–215
3. Hennessy, M., Lin, H.: Symbolic bisimulations. Theoretical Computer Science **138**(2) (1995) 353–389

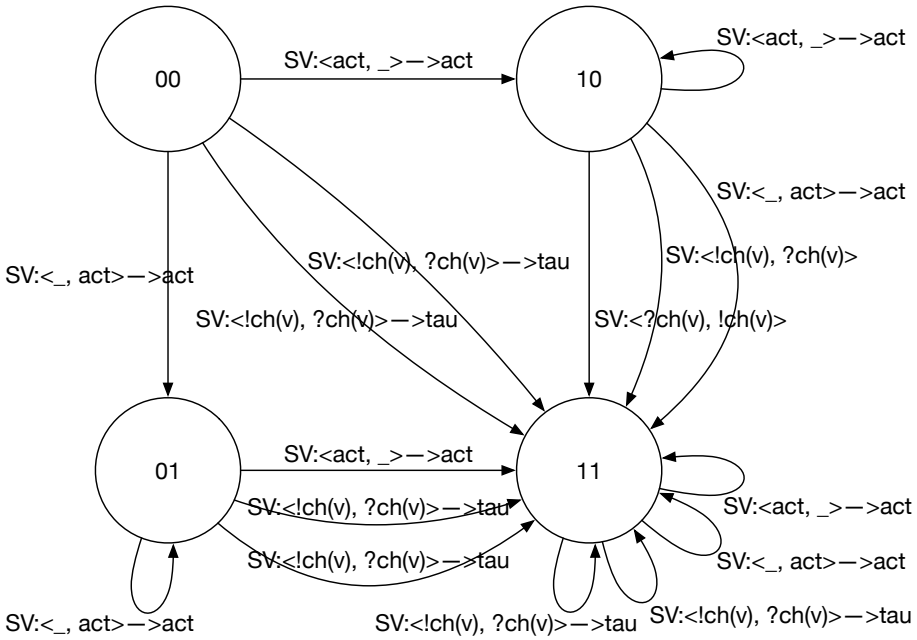

Fig. 7. The open automaton of the CCS formula a.P||b.Q

4. Lin, H.: Symbolic transition graph with assignment. In Montanari, U., Sassone, V., eds.: Concur'96. Volume 1119 of LNCS., Springer, Heidelberg (1996) 50–65
5. Hennessy, M., Rathke, J.: Bisimulations for a calculus of broadcasting systems. Theoretical Computer Science **200**(1-2) (1998) 225–260
6. Milner, R.: Communication and Concurrency. Int. Series in Computer Science. Prentice-Hall, Englewood Cliffs, New Jersey (1989) SU Fisher Research 511/24.