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 [6])
- 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 [6, 7] 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

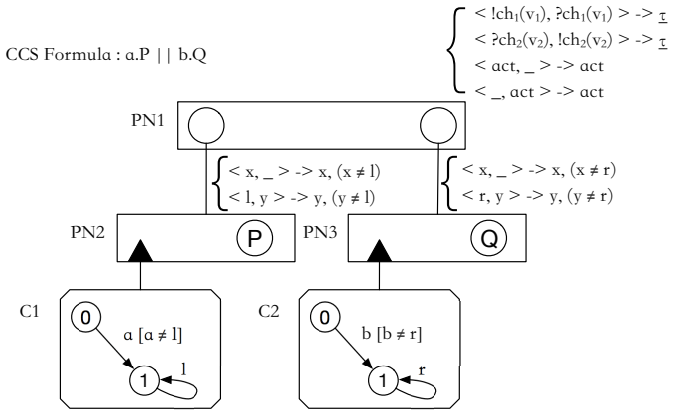


Fig. 1. The pNet encodings for CCS formula

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 concrete syntax is typically "<act, _ > -> act", in which "_" means that the corresponding subnet is not involved in this synchronisation. There is also a difference with the usual CCS notation of input/output actions: in the first vector, *ich* and *ch* are corresponding input and output channels (see next section for a more complete formalisation). 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

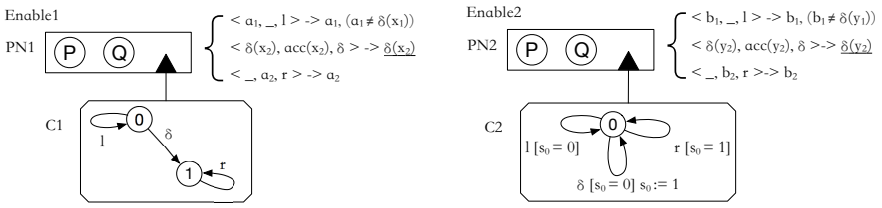


Fig. 2. Two pNet encodings for the Lotos Enable operator P»Q

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 [8] 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. Formally, value-passing actions *à la* CCS, denoted as "?ch(x)" and "!ch(exp)" in our CCS running example, stand respectively for $CCSact(i(ch), ?x)$ and $CCSact(ch, exp)$, where ch is a variable denoting a channel name, $?x$ is an input variable denoting the data argument received as input on ch, and exp some data expression (containing no input variables). In our example we will use the usual CCS syntax. 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* $\forall j \in J.\, vars(e_j) \subseteq vars(s) \wedge x_j \in vars(s')$*.*
- *The* x_j *here are state variables of state* s'*. State variables can only be modified by the assignment in the transitions targetting its owner state. A global state variable of a pLTS is a variable* x *satisfying* $\forall s \in S.\, x \in vars(s)$*.*

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:*

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

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

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

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

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

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* $(S_i, s_{0i}, \to_i)^{i\in I}$ *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* $(S_i, s_{0i}, \to_i)$*, 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.*

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

Example 1. An open-transition. The CCS pNet of Fig. 1 has 2 controllers and 2 holes. We show its full open Automaton in section 5.5, it has 16 open transitions. We detail 3 of them here:

$$OT_1 = \frac{0 \xrightarrow{a}_{C_1} 1 \quad 0 \xrightarrow{b}_{C_2} 1 \quad \{\} \quad [a \neq l \wedge b \neq r \wedge a = ?ch(v) \wedge b = !ch(v)]}{\triangleleft 00 \triangleright \xrightarrow{\tau} \triangleleft 10 \triangleright}$$

$$OT_3 = \frac{0 \xrightarrow{a}_{C_1} 1 \quad \{\} \quad [a \neq l]}{\triangleleft 10 \triangleright \xrightarrow{a} \triangleleft 11 \triangleright}$$

$$OT_{11} = \frac{1 \xrightarrow{l}_{C_1} 1 \quad 0 \xrightarrow{b}_{C_2} 1 \quad \{\xrightarrow{b_P} P\} \quad [a \neq l \wedge b \neq r \wedge b_P = ?ch(v) \wedge b = !ch(v)]}{\triangleleft 10 \triangleright \xrightarrow{\tau} \triangleleft 11 \triangleright}$$

[TODO:Eric: short comments]

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:

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

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.

The semantic of the open pNets is already defined as an open automaton in [7] using open transitions to present the transitions of its global states.

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:

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

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:

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

Example 2. Using the operational rules to compute open-transitions In Fig. 3 we show the deduction tree used to construct and prove the open transition OT_{11} of `CCS` (see example page 8). The proof tree uses TR1 twice, for the l transition of

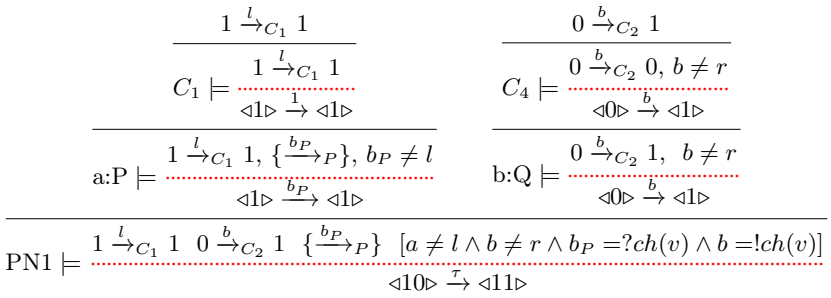


Fig. 3. Proof of OT_{11} (with interaction of processes P and action b) for "a.P||b.Q"

C_1 and for the b transition of C_2, then uses an action b_P of hole P, and combines the results using the second vector of the PN2 sub-pNet, and the third vector of the top node. This yields a final τ transition. The deduction tree in the figure shows how the predicates are generated in this process.

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

One of the important goal of the Open pNet framework is to provide us with a model able to express the semantics of many different algebras or languages. In particular, as we have seen in section 3, the *term algebra* used to express the action expressions and the data in their parameters is left open in the pNet formalisation.

So, for a given language, we now need to specify its data and action domains, giving them an abstract syntax (sorts, constructors, operators, and predicates). We call this a *Presentation*. We also define a concrete syntax, that will be used for pretty-printing, but also for translating the presentation, and the predicates, into Z3 syntax.

An algebra presentation contains:

- Sorts: Constants sets of the algebra, types of the data and actions.
- Operators: Operators used to construct action expressions (including their data parameters); they are constructors of data and action sorts, but also predicates over these objects.
- Constants: The satifiability solver need to know which actions in the pLTS behaviours are constants (like the l, r local actions of the controlers in the

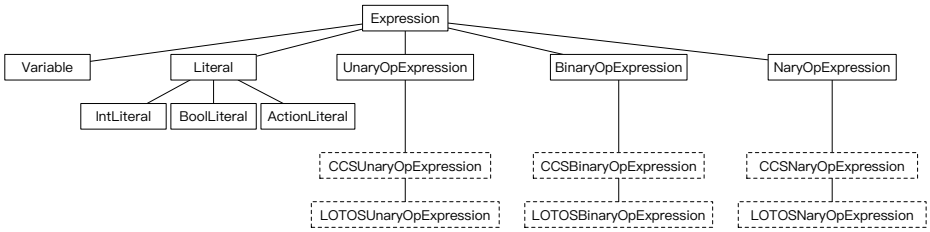


Fig. 4. The architecture of Expression class

CCS example). So for a given pNet system, the Action sort will include all these constants.

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. Sorts here act as sets of actions for pNet, pLTS or holes. There *must* be one sort for actions, and eventually several others for the types of data parameters. Sorts for integer and boolean are implicitly declared.

Operators We need introduce operators for building expressions, of both action and data sorts, when encoding the algebra into pNets. So that is needed to know about the input and 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 $< Sort_{i_1}, ..., Sort_{i_n} > \rightarrow Sort_j, i_1, ..., i_n, j \in [0..k]$. **[TODO: I do not understand this. As discussed today, I will wait till you include an example of algebra presentation before rewriting the comment.]**

Implementation of the expression meta-model The VerCors platform uses Ecore and EMF to define the structure of both pNets and actions algebras. There exists a predefined set of action and data expressions classes, including boolean, integer and action expressions, with predefined variables, literal (constants), and operators. Operators can be categorized into unary, binary and N-ary operators. The original version of VerCors did not allow to extend these classes, we have modified its architecture so that a user can extend them, defining more elements for his specific algebra.

In Fig 4, we show the Expression class we improved, including extensions for CCS and LOTOS.

[TODO:give an example of user-defined sorts/operators, and tell that you will give it's translation into Z3 in section 5.4]

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.

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 satisfiable, the resulting OT is discarded. This will minimize the number of resulting transitions. Then a reachability check could potentialy prune the state 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.*

The proof can be found in [7].

5 Implementation

The VerCors platform uses (closed) pNets as the intermediate language for some high-level language or graphical formalism to be translated into both input for a model-checker and for generating executable code automatically [9].

In our prototype we start directly with open pNets expressed programmaticaly, using the pNet API (graphical editors may be added later). From these, we compute the open automaton of an open pNet by directly implementing the sketch showed before. However, we made some modifications due to implementation choices:

- Steps 2a-2b are merged, applying Tr2 from the premisses OTs from subnets to generate open transitions without explicitely constructing the deduction tree.

– We defined a general naming schema for implementing the fresh and clone functions to rename all the variable to make them unique, but also readable enough for debugging purpose.
– Besides that, we also have designed a management of state variables and assignments throughout the whole computation.
– Satisfiability check is done only at top level of the open-transition construction. An alternative would be to submit the satisfiability check to the SMT solver at each level of the hierarchical OT 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.

In the following sections, we detail:

– The structure of fresh variable names.
– The details of the implementation for step 2a-2b, refined into:
  – combining all possibilities of subnets activity/inactivity,
  – matching each combination with synchronisation vectors.
– Management of local variables of pLTSs.
– Pruning of unsatisfiable open transitions using the Z3 SMT solver.

5.1 Fresh variables

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 have defined a structure for the name of the fresh variables, the variables renamed with a regular format by the function fresh().

More precisely, the fresh function generates a new name adding a suffix after the original name. The suffix contains three parts 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.*

[TODO:Option: Add the process and the vector ids ?]

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 Combining and Matching using TR2

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 generated during matching the subnets' behaviors with the synchronization vectors. Note that it is simple to collect required terms in the first part while in the second part we 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. For 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.

For holes, their behavior is unspecified, it doesn't have an exact external presentation of action. We only give an action variable to each hole and treat it as hole behavior to suppose working status of the 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 remaining 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.

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. Remember variables inside a synchronisation vector are consider local to the vector; so 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.

There is one mismatch case we can easily detect here, that is its working status: 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*,

Algorithm 1 Combining

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

```
1: Initialise an empty combination list L_C;
2: Extract a list of open transitions L_ot from TR;
3: Add null to 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;
```

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 out.

Then we build the resulting open transition. Satisfiability of the predicate is not checked at this level, but only later, at the toplevel of the pNet.

5.3 Management of assignments

The pLTS contains some variables in each of its states (state variables), or just as global states of the pLTS (global variables). Both can be assigned in the pLTS transitions. The algorithm composing assignments from subnets is different from the generation of predicates. When applying Tr1, we just keep the assignments in the Post of the open transition. And when it comes to Tr2, Post is the union of the assignments propagate from the subnets. The assignments provide value of the variables in predicates. There is usually not only one assignment for a state variable as several incoming transitions of a state can perform different assignments on the same variable. The values of the variables are necessary for checking satisfiability of the open transitions together with predicates, miss of them may drop the unsatisfiable results. In order to avoid such mistakes, the algorithm manages the variables together with a list of all its possible assignments.

We use a list of triples $< v,\ S,\ AssignRH >$ for each pLTS to record the information of both state and global variables of the pLTS where v is the variable in pLTS, S is its owner state and $AssignRH$ is a list of expressions over other variables in the right hand side of assignments of v. The S can be $null$ in the record triple, but it means the variable is global in the pLTS belonging to all the states instead of meaning it belongs to nothing. So it should be noted that the changing of the value of the global state variable every time a transition is conducted. $AssignRH$ is the collection of all the possible expressions coming from the the right hand side of assignments. It keeps the information of the assignments for each variable. We update the list of record triples every time Tr1

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;
```

is applied, adding the right side of the assignments into corresponding expression lists. So the list keeps the all the assignments in the pLTS. Then we can track all the possible value of the variables when computing open transitions. The initial value of the state variable is also seen as an assignment for that variable. We must give a initial value to the global state variable or there will be some problem in the start state. However, we sometimes don't give initial values to the state variables (except the global variables) as their initial values are decided by their first assignments. Checking whether each variable is initialized before being used is a separated problem, that should be done before starting our algorithm.

5.4 Pruning the unsatisfiable results

We have used a brute-force method to generate all the possible open transitions in the open automaton. It obviously contains some of unsatisfiability in the open transitions. In Fig 5, we display an unsatisfiable open transition from the result of the CCS running example. It shows the case that the top level transmits out the action from the PN2, formula $a.P$, as this time only PN2 is working and C1 is performing action a when hole P is also working. When matching this case against the second synchronization vector of PN2, transmitting the behavior from hole P to the higher level, is used while it is obviously violate the guard of the vector. So we can easily find the contradiction between the generated

$$ot = \frac{\{0 \xrightarrow{a} 1\},\ \{\xrightarrow{:hb:15:2}\},\ :ra:15:2 = act:sva:1:3 \wedge: ra:152:1 = l \wedge a =: ra:152:1 \wedge a \neq l \wedge: hb:15:2 = y:sva:15:2 \wedge: ra:15:2 = y:sva:15:2 \wedge y:sva:15:2 \neq l \wedge: ra:1:3 = act:sva:1:3}{\triangleleft 00 \triangleright \xrightarrow{:ra:1:3} \triangleleft 10 \triangleright}$$

Fig. 5. One of the unsatisfiable open transitions in CCS running example

Input

```
1  (declare-datatypes () ((Action
2   l  r  tau
3  (EMIT (chan1 Int)(arg1 Int))
4  (RECEIVE (chan2 Int)(arg2 Int)))))
5  (declare-const |:ra:15:2| Action)
6  (declare-const |act:sva:1:3| Action)
7  (declare-const |:ra:152:1| Action)
8  (declare-const a Action)
9  (declare-const |:hb:15:2| Action)
10 (declare-const |y:sva:15:2| Action)
11 (declare-const |:ra:1:3| Action)
12 (assert (= |:ra:15:2| |act:sva:1:3|))
13 (assert (= |:ra:152:1| l))
14 (assert (= a |:ra:152:1|))
15 (assert (not (= a l)))
16 (assert (= |:hb:15:2| |y:sva:15:2|))
17 (assert (= |:ra:15:2| |y:sva:15:2|))
18 (assert (not (= |y:sva:15:2| l)))
19 (assert (= |:ra:1:3| |act:sva:1:3|))
20 (apply ctx-simplify)
21 (check-sat)
```

Output (Result)

```
(goals
(goal
  (= |:ra:15:2| |act:sva:1:3|)
  (= |:ra:152:1| l)
  (= a l)
  (not and)
  (= |:hb:15:2| |y:sva:15:2|)
  (= |:ra:15:2| |y:sva:15:2|)
  (not (= |y:sva:15:2| l))
  (= |:ra:1:3| |act:sva:1:3|)
  :precision precise :depth 1)
)
unsat
```

Fig. 6. The input of the Z3 solver in SMT-LIB language and the output result

predicate term $ra:152:1 = l \wedge a =: ra:152:1$ and the guard composed into the predicate $a \neq l$.

To get a correct open automaton of the pNet, such unsatisfiable open transitions need to be eliminated. This is not easy to perform directly inside our algorithm, in particular because it requires some symbolic computation on the action expressions and the predicates, and that this reasonning may depend on the specific theory of the action algebra. We choose a SMT solver Z3 as the checker for the open transitions considering the predicates may contain the expression using uninterpreted functions declared by users. The "Modulo Theory" part of SMT solvers is important here, so that the solver can use specific properties of the action algebra.

Interaction with Z3 There are several ways to interact with the Z3 solver. It is possible to use Z3 interactively using the SMT-LIB (Python-based) language. From inside our algorithm code, we implement the dynamic interaction between the pNet API and Z3 solver through its JAVA API. But here we present the code in Z3 by SMT-LIB language instead of JAVA codes to make it easier to read. In Fig 6, we show the input of the unsatisfiable result before. The input starts with the declaration of the CCS action algebra types and constructors. Some *datatypes* are declared to input the sorts of the CCS example, here we only declare one *datatypes* named *Action* to involve all the constant actions. The *Int*

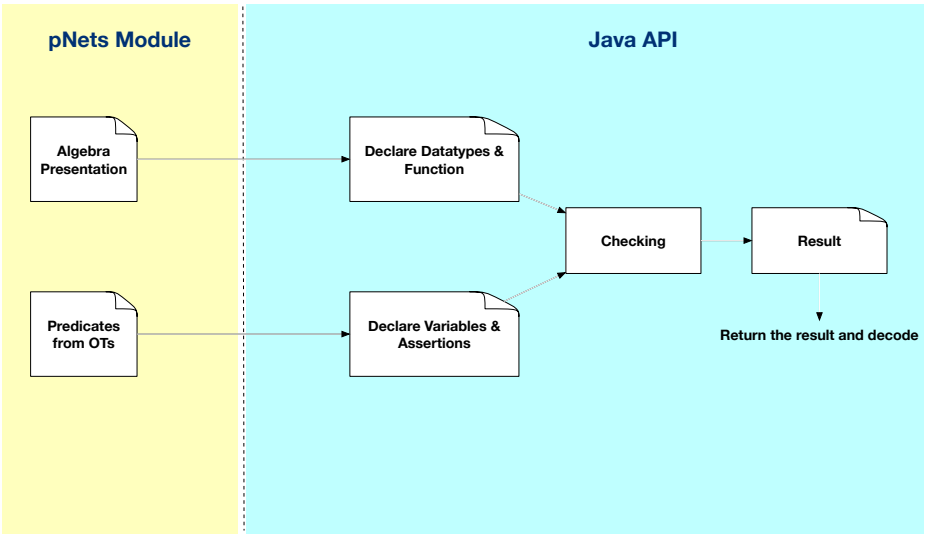


Fig. 7. The architecture of the interaction between pNet API and Z3

and *Bool* are already existed in Z3. Then we input the predicate of that open transition in two parts. The variables contained in the predicate are declared first, then the terms of predicate are inputed as *assertions* one by one. In the output, "(not and)" shows Z3 have found out the contradiction in the predicate and the result "unsat" is just what we expected.

We present the architecture of the interaction with Z3 in Fig 7. For each OT generated at the toplevel of a pNet, the predicate must be submitted for satisfiability check to Z3. For that we need to translate into Z3 (Java-API) syntax, both the elements of the action algebra, and the predicates itself, including the encoding of the variable assignments (the Post part).

The *presentation* of the action algebra is declared at the beginning of the algorithm as the *sorts* and the *operators* will be used throughout the whole algorithm. It is translated into *datatypes* and $functions$ then submitted to the Z3 at the same time. Each candidate predicate will then be translated into *assertions* before submitted to Z3 after the algorithm computing out all the possible open transitions, they act as intermediate results there. Together with the *datatypes* and $functions$, the *assertions* are used to check the satisfiability of the predicate in Z3.

Translation of Action algebra presentation The *datatypes* are declared according to the *sorts* the users declared in the *presentation*. If the return type of the $functions$ is one of the *datatypes*, we add this $function$ when declare the *datatypes*, it present the reality that every action expression also belongs to a sort of pNet. Although in Fig 6 we have not declared $functions$, they should be exactly declared if there are new *operators* declared in the *presentation*.

The input type of the *function* can be Int, $Bool$ or the *datatypes* declared all determined by the construct of the *operators* in the presentation. For example, we define a new operator to present the equation between two actions in CCS example EQUAL in the format likes:

$$< Action, Action > \rightarrow Bool.$$

It will be translated into the Z3 statement:

(declare-fun EQUAL (Action Action) Bool) .

Translating assignments into predicate terms The assignments are also taken into account when checking the satisfiability of the open transitions as they represent the value of the variables involved in an OT. In order to do that, we employs a translation from assignments into term of predicates. The assignments required to be translated are only those belong to the variables contain in the start states of the open transitions. The assignments are represented as equations between the variables and the expressions on the right side of assignment. For the assignments of the same variable, we do not figure out which state is the precursor of current state, we keep all these possible equations instead, and generate the disjunction of these equations. The state could always move to the target state if there exists one of the values of its variable satisfies the guard. **[TODO:It won't matter the variable of the next state except that the right hand side expression contain its variables. This is a problem left to be solved in our future work. WHICH PROBLEM ?]** After that, for the assignments of the different variables, we generate a conjunction of them. This way we obtain a new term of the predicate involving assignments of the variables.

checking satisfiability Here we declare the variables in the predicate first to make Z3 know all the parameters used for checking this time. The variable including the argument in the operators must has its sort as every pNet defines two default sorts Int, $Bool$ default and at least one sort for the actions. The variables can declared using the *datatypes* declared in the format as:

(declare-const a Action) .

The predicate submitted to the Z3 contain the assignments as them become a part of predicate after our translation. Each term of the predicate is an assertion in Z3, as an example, in the former unsatisfiable result in Fig 5, the contradiction is occured in terms:

$$(: ra : 152 : 1 = l) \wedge (a =: ra : 152 : 1) \wedge (l \neq a).$$

from which we generate following three assertion:

(assert (= |:ra:152:1| l)) (assert (= a |:ra:152:1|)) (assert (not (= a l)))

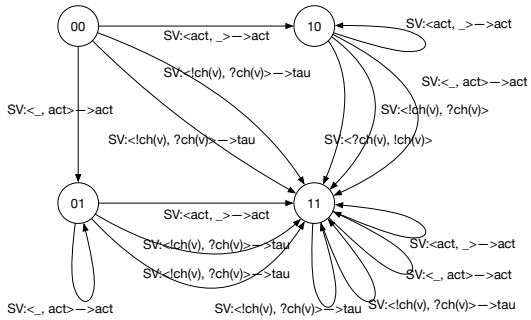
5.5 Result of the running example

[TODO:The open automaton is not the final one. The vectors are wrong, and you should add at least the OT name...] We get the final result after pruning all the unsatisfiable open transitions, 2/3 of the intermediate results; the resulting open automaton is shown in Fig. 8, we give out the open automaton manually through drawing the final result open transitions generated by algorithm. A part of results we computed by hand in advance are also list below. The ot_1 represents the value-passing between $a.P$ and $b.Q$ and they are synchronized then give out an silent action τ, process P and Q should not work in this case. The ot_3 and ot_5 represent another possible path to execute $a.P||b.Q$, action a and b are finished in sequence instead of synchronizing, they are just transmitted out as the external action. The behavior of the processes, presented as holes P and Q in this pNet, is uncertain, we only know the sort *Action* for them. So the action from P might be synchronized with the action b that is what ot_9 presents or be directly transmitted to the external as the ot_7 shows. And we can see more cases of the interaction between these two process in the open transitions start at the state 11, synchronizing two actions like ot_{13} or directly transmitting like ot_{15}.

To sum up, from the pNet encoding CCS formula inputed by the user we generate the open transitions and prune the unsatisfiability automatically. The final result open transitions represent all the possible movements of the CCS example. They can exactly construct the open automaton for the given 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] In this paper we presented the algorithm of generating the open automaton for the open pNet involving the implementation part of the algorithm. Our implementation includes two main parts. First, we compute all the possible open transitions. The actions in pLTS and hole behaviors are composed in the algorithm while the generation of predicate need to match the combinations of the subnets with the synchronization vectors. Then we ensure the correctness of the intermediate result. The method use a SMT solver Z3 for checking the satisfiability of the open transitions using the predicate of them. In order to do that, we have done some translation on the presentation, the predicate and the assignments of the variables which are also considered as a part of predicates before submitting them to the Z3. We implemented our algorithm in the VerCors platform and use some running examples, the formula from other process algebras, to test the algorithm. The result of the running examples are shown as the open automaton with all the parameters generated by the algorithm automatically. The open automaton successfully presents all the possible movement of the formula.



$$ot_1 = \frac{\{0 \xrightarrow{a}_{C_1} 1, 0 \xrightarrow{b}_{C_2} 1\},\quad a = !ch_{11}(v_{11}) \wedge b = ?ch_{11}(v_{11}) \wedge a \neq l \wedge b \neq r \wedge v_1 = \tau}{\lhd 00 \rhd \xrightarrow{v_1} \lhd 11 \rhd}$$

$$ot_3 = \frac{\{0 \xrightarrow{a}_{C_1} 1\},\quad a \neq l \wedge v_3 = a}{\lhd 00 \rhd \xrightarrow{v_3} \lhd 10 \rhd}$$

$$ot_5 = \frac{\{0 \xrightarrow{b}_{C_2} 1\},\quad b \neq r \wedge v_5 = b}{\lhd 10 \rhd \xrightarrow{v_5} \lhd 11 \rhd}$$

$$ot_7 = \frac{\{1 \xrightarrow{l}_{C_1} 1\},\quad \{\xrightarrow{hb_{17}}\},\quad v_7 = hb_{17}}{\lhd 10 \rhd \xrightarrow{v_7} \lhd 10 \rhd}$$

$$ot_9 = \frac{\{0 \xrightarrow{b}_{C_2} 1\},\quad \{\xrightarrow{hb_{19}}\},\quad b = !ch_{19}(v_{19}) \wedge b \neq r \wedge hb_{19} = ?ch_{19}(v_{19}) \wedge v_9 = \tau}{\lhd 10 \rhd \xrightarrow{v_9} \lhd 11 \rhd}$$

$$ot_{13} = \frac{\{1 \xrightarrow{l}_{C_1} 1, 1 \xrightarrow{r}_{C_2} 1\},\quad \{\xrightarrow{hb_{113}}, \xrightarrow{hb_{213}}\},\quad hb_{113} = !ch_{113}(v_{113}) \wedge hb_{213} = ?ch_{113}(v_{113}) \wedge v_{13} = \tau}{\lhd 11 \rhd \xrightarrow{v_{13}} \lhd 11 \rhd}$$

$$ot_{15} = \frac{\{1 \xrightarrow{l}_{C_1} 1\},\quad \{\xrightarrow{hb_{115}}\},\quad v_{15} = hb_{115}}{\lhd 11 \rhd \xrightarrow{v_{15}} \lhd 11 \rhd}$$

Fig. 8. The open automaton of the CCS formula a.P||b.Q (a part of open transitions is shown)

[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)] This paper raises the question of the efficiency and performance of the SMT solvers. It is not ensured that Z3 is the best one, in fact, there is some problem during the implementation the checking method using Z3. For example, if the *functions* and the *datatypes* independent to each other, checking the *assertions* is a simple work. However, *functions* on recursive *datatypes* make it more complex, some special rules might be defined by user for the induction.

The next work after the generation of the open automaton of the pNet is planned to check the bisimulation of the pNets. In previous paper, we have already found the FH-bisimulaiton[**?**] as a prototype. We will decide the definition of the bisimulation of the pNets and the algorithm to prove a relation between two pNets is a bisimulation. Before that work, we need to refine the result open transitions, eliminating the redundancy in the predicate of the result. The ideal

result is what we showed in the bottom part of Fig. 8 while there is many intermediate variables we used, such as fresh local variables of synchronization vectors, and fresh result variables of intermediate open transitions, in the algorithm for the propagation of the actions or values. So far, to eliminate the intermediate variables in the predicate, we define the range of the significant variables that we won't eliminated: the local variables of pLTS transitions, the hole behaviors and the result action at the top level.

=> 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
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. 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'15)
7. Henrio, L., Madelaine, E., Zhang, M.: A Theory for the Composition of Concurrent Processes. In Albert, E., Lanese, I., eds.: 36th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE). Volume LNCS-9688 of Formal Techniques for Distributed Objects, Components, and Systems., Heraklion, Greece (June 2016) 175–194
8. Milner, R.: Communication and Concurrency. Int. Series in Computer Science. Prentice-Hall, Englewood Cliffs, New Jersey (1989) SU Fisher Research 511/24.
9. Henrio, L., Kulankhina, O., Madelaine, E.: Integrated environment for verifying and running distributed components. In: in proc. of the 19th Int. Conf. on Fundamental Approaches to Software Engineering (FASE'16), Springer (2016)