Using SMT engine to generate Symbolic Automata[⋆]

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

[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. **[TODO:not good for a FM, (to be rewriten by Eric)]**
Open pNets are used to model the behavior of open synchronous or asynchronous systems expressed in various calculi or languages. They are endowed with a symbolic operational semantics in terms of so-called "Open Automata". We implement an algorithm computing this semantics, and building predicates expressing the synchronization conditions allowing some combination of events in the pNet system. Checking such predicates requires symbolic reasoning over first order logics, and specific functions of each action algebra. To reduce the complexity of the generated open automata, we use the Z3 SMT engine to check satisfiability of the predicates, and minimize the number of significant symbolic transitions.

1 Introduction

In the nineties, several works extended the basic behavioral models based on labelled transition systems to address value-passing or parameterized systems, using various symbolic encodings of the transitions [?,?,?,?]. In [?], H.M. Lin addressed value-passing calculi, for which he developed a symbolic behavioral semantics, and proved algebraic properties. Separately J. Rathke [?] defined another symbolic semantics for a parameterized broadcast calculus, together with strong and weak bisimulation equivalences, and developed a symbolic model-checker based on a tableau method for these processes. Thirty 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.

Parameterized Networks of Synchronized Automata (pNets) were proposed to give a behavioral specification formalism for distributed systems, synchronous, asynchronous, or heterogeneous. It is used in VerCors, a platform for designing and verifying distributed systems, as the intermediate language for various high-level languages. The high-level languages in VerCors formalize each component of the distributed system and gives out the composition of these components. pNets provides the core low-level semantic formalism for VerCors, and is made of a hierarchical composition of (value-passing) automata, called parameterized labelled transition systems (pLTS), where each hierarchical level defines the possible synchronization of the lower levels. Traditionally, pNets have been used to formalize fully defined systems or softwares. But we want also to define and reason about incompletely defined systems, like program skeletons, operators, or open expressions of process calculi. The open pNet is proposed to solve the problem of the undefined components contained in the systems using "holes" as the process parameter dealing with these components. The hole acts as the placeholder for the uncertain component with a set of possible behaviors the component might conduct, presenting the "open" property. The pNet model was developed in a series of papers [?,?] in which many examples have been introduced showing its ability to encode the operators from some other algebras or program

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

skeletons. The operational semantic of an (open) pNet is defined as an Open Automaton in which Open Transitions contain logical predicates expressing the relations between the behavior of the holes, and the global behavior of the system. In the previous publication, only a sketch of a procedure allowing to compute this semantics was presented, together with a proof of finiteness of the open automaton, under reasonable hypotheses on the pNet structure.

Implementing this semantics raised several challenges, in order:

- to get a tool that could be applied to pNets representing various languages, in particular various actions algebras, with their specific decision theories,
- to separate clearly on one hand the algorithmic part, as an algorithm generating the transitions of the open automaton from combination of all possible (symbolic) behaviors; on the other hand the symbolic reasonning part, specificaly here using an SMT engine to check the satisfiability of the predicates generated by our algorithm,
- to build a prototype and validate the approach on our basic case-studies, and understand the efficiency of the interaction with the SMT solver.

In the long goal, we want to be able to check the equivalence between open systems encoded as pNets. The equivalence between pNets is "FH-bisimulation" taking the predicate of the open transitions into account each time matching such open transitions. We foresee that the interplay with the SMT solver that we use here for satisfiability of open transitions will be similar with what we need to prove (symbolic) equivalence between open transitions.

Contribution In the article we show how:

- We define the open automaton generation algorithm, and we implemented a full working prototype, within the VerCors platform. In the process, we improved the semantics rules from [?], and add features in the algorithm to deal the full model, including management of variables and assignments.
- We implement the interaction between our algorithm and the Z3 SMT solver, for checking satisfiability of the transitions generated by the algorithm.
- We show the interest of this approach on an industrial-inspired use-case, namely one architectural pattern extracted from the BIP specification of a nano satellite on-bord software.

Related works There is not much research work towards trying to develop symbolic bisimulation approaches for the value-passing process algebra and languages, as our long-term goals, especially no algorithm treatment of the symbolic systems developed by interacting with automatic theorem provers. The closest work is the one already mentioned from J. Rathke [?], who developed the symbolic bisimulation for a calculus of broadcasting system (CBS). CBS is similar with classic process calculi such as CCS and CSP, but communicating by broadcasting values, one-to-many communication instead of one-to-one communication and transmitting values without blocking. That makes the definition of the symbolic semantic and bisimulation equivalence different from the classic works.

For other applications, e.g. for the analyses of programming languages, there exist dedicated platforms making use of external automatic theorem provers (ATP), or even some automatic tactics from interactive theorem provers (ITP), to perform symbolic reasoning, and for example to discharge some subgoals in the proofs. Tools like Rodin [?,?,?] have already integrated several provers, like Z3, as modules for proving the proof obligations generated from the model inputed by user. The prover we used, which also happens to be Z3, is developed by Microsoft Research based on the satisfiability modulo theories (SMT), mainly applied in extended static checking, test case generation, and predicate abstraction. In a similar way, there are several ATPs/ITPs we could consider to use for the result pruning and bisimulation checking in our algorithm, as an alternative to Z3, such as CVC4 [?], Coq [?], Isabelle [?] or others.

Structure. In section 2 we give a description and a formal definition of the pNet model, as found in previous publications. Then in **??** we present a simple example that will illustrate the rest of the paper. Section 5 recalls the operational semantics of pNet, including its structural rules. Section 6 explains in details our implementation within the Vercors platform, and shows the full result of the semantic computation on the running example. Finally we conclude and discuss perspectives in section 7.

2 Background: pNets definition

[TODO:To be reduced, moving more material to the appendix]

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 *parameterized labelled transition systems (pLTSs)*, expressing the behavior of basic processes, or *holes*, used as placeholders for unknown processes, of which we only specify their set of possible actions, named *sort*. Nodes of the tree (pNet nodes) are synchronizing artifacts, using a set of *synchronization vectors* that express the possible synchronization between the parameterized 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 parameterized actions, that are symbolic expressions using data types and variables. As our model aims at encoding the low-level behavior 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 unification between two or more action expressions. This will be used in the semantics of synchronization vectors to express various kinds of communication or synchronization 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 sub-terms. The function $vars(t)$ identifies the set of variables in a term $t \in \mathcal{T}$.

pNets can encode naturally the notion of input actions as found e.g. in value-passing CCS [**?**] or of usual point-to-point message passing calculi, but it also allows for more general mechanisms, like gate negotiation in Lotos, or broadcast communications.

Algebra presentations

Definition 1. *An Algebra Presentation is a pair* $\mathcal{P} =< Sorts, Constrs, Ops >$ *where:*

- *Sorts is a set of Sort names* [4]

[4] Later we may want to extend this with Sort constructors, like Array or Pair, but this is not needed now

- *$Constrs$ is a set of constructor operators, with Con the constructor name, and arity: $arity(Con) = n \in \mathbb{N}$, and with their signature and their associated selectors, of the form: '$Con : (sel_1, sort_1), ..., (sel_n, sort_n)-$ $sort$'. For each argument, the pair $(sel_i, sort_i)$ defines an auxiliary operator of name sel_i with signature $sel_i : sort- > sort_i$.*
- *Ops is a set of other (auxiliary) operators, with their arity and signature, of the form: $Op : sort_1, ...sort_n- > sort$*
- *$Constrs(sortname), Sels(sortname)$, respectively define the set of constructors and selectors of sort $sortname$*

All sorts and operator names must be distinct.
Amongst Constructors, those of arity 0 are called constants, and we define $Consts(\mathcal{P}) = \{Con \in Constrs.arity(Con) = 0\}$.

2.1 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.

Each state has its set of variables called *State variables*, which can only be modified by the assignment in the transitions targeting its owner state. A global state variable of a pLTS is a variable x satisfying $\forall s \in S. x \in vars(s)$.

Note that we make no assumption on finiteness of the set of states nor on finite branching of the transition relation.

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

$$\begin{array}{rll} \alpha \in \mathcal{A} ::= & a(p_1, \ldots, p_n) & \text{action terms} \\ p_i ::= & Expr & \text{parameters} \\ Expr ::= & Value \mid x \mid op(Expr_1, .., Expr_n) & \text{Expressions} \end{array}$$

Definition 2 (pLTS). *A pLTS is a tuple $pLTS \triangleq \langle\!\langle S, s_0, \to \rangle\!\rangle$ where:*

- *S is a set of states.*
- *$s_0 \in S$ is the initial state.*
- *$\to \subseteq S \times L \times S$ 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 parameterized 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 $s \xrightarrow{\langle \alpha,\ e_b,\ (x_j := e_j)^{j \in J} \rangle} s' \in \to$ then $vars(\alpha) \subseteq vars(s)$, $vars(e_b) \subseteq vars(s')$, 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'.*

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

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 synchronization between global actions and internal actions is given by *synchronization vectors*: a synchronization vector synchronizes one or several internal actions, and exposes a single resulting global action.

Definition 3 (pNets). *A pNet is a hierarchical structure which 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.*
- *$J \in \mathcal{I_P}$ is the set over which holes are indexed. I and J are* disjoint*: $I \cap J = \emptyset$, $I \cup J \neq \emptyset$*
- *$S_j \subseteq \mathcal{A}$ is a set of action terms, denoting the* Sort[5] *of hole j.*
- *$SV_k^{k \in K}$ is a set of synchronization vectors ($K \in \mathcal{I_P}$). $\forall k \in K, SV_k = \alpha_l^{l \in I_k \uplus J_k} \to \alpha'_k$ where $\alpha'_k \in \mathcal{A_P}$, $I_k \subseteq I$, $J_k \subseteq J$, $\forall i \in I_k. \alpha_i \in \mathrm{Sort}(pNet_i)$, $\forall j \in J_k. \alpha_j \in S_j$, 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 $\mathrm{Label}(SV_k) = \alpha'_k$.*

3 BIP architectures, and their encodings into pNets

4 Running example

Several publications [?,?] already have introduced many examples of pNets, encoding operators of various classical process algebras, or more complex synchronization structures in distributed or parallel languages.

5 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 the actions 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 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*, represented as a substitution $\{x_k \leftarrow e_k\}^{k \in K}$ where x_k are variables of s', s'_i, and e_k are expressions over the other variables of the open transition.*

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 $\mathcal{S}$.*

When building an Open Automaton as the semantics of a pNet, its *states*, and the shape of the *predicates* in its transitions have a specific structure:

[5] The formal definition of *Sorts* (set of actions of a Hole or pNet), *Leaves* and *Holes* (all pLTSs (resp holes) in a pNet hierarchical system. These can be found in [?].

States of open pNets: A state of an open pNet is a tuple of the states of its leaves (in which we denote tuples in structured states as $\triangleleft \ldots \triangleright$). 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 synchronization vector SV_k, for $k \in K$. We build a predicate *MkPred* relating the actions of the involved sub-pNets and the resulting actions. This predicate verifies:

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

Structural Semantic Rules: Now we build the semantics of an open pNet as an open automaton where LTSs are the pLTSs at the pNet leaves, and the states are structured as in the previous section. To build an open transition one first projects the global state into states of the leaves, then applies pLTS transitions on these states, and compose them with actions of holes using synchronisation vectors.

The semantics regularly instantiates *fresh* variables, and uses a *clone* operator that clones a term replacing each variable with a fresh one. The variables in each synchronization 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.

Definition 6 (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 (**Tr1**) for a pLTS p checks that the guard is verified and transforms assignments into post-conditions:*

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

*The second rule (**Tr2**) deals with pNet nodes: for each possible synchronization 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.*

Tr2:

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

Example 1. 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_2.

The proof tree uses TR1 twice, for the δ transition of C_1 and for the $acc(x)$ transition of C_2, then uses an action hb_{12} of hole P, and combines the results using the first vector of the PN2 sub-pNet, and the second vector of the top node according to TR2. This yields a final $\underline{\delta(x)}$ transition. The deduction tree in the figure shows how the predicates are generated in this process.

6 Implementation

The VerCors platform uses 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 [?]. We have extended the pNet API in VerCors to deal with open pNets, and also to specify the structure of action algebras.

In this section we describe the algorithm implementing the pNet semantics, the interaction with the Z3 SMT solver, and we show the result on our example.

Algorithm 1 Open Automaton Generation

```
Input: The pNet node P.
Initialize sets $U$, $E$ for unexplored/explored global states, $L$ for result OTs;
while !isEmpty(U) do
    Chose $S$ in $U$;
    OT = MakeTransitions(P, S);
    Store $OT$ in $L$;
    for each target global state $T$ from $OT \in L$ do
        if (!$U.contains(T)$) && (!$E.contains(T)$) then Add $T$ into the $U$;
    end for
end while
Prune the open transitions in $L$ using the SMT solver;
return $L$;
```

6.1 The Generation of Open Automata

We have already have the sketch of computing the open automata [?]. While in our implementation we does not build explicitly a proof tree for every open transition. Instead, we repeat

Algorithm 2 MakeTransitions()

```
Input: The pNet node P; The start global state S.
Initialize a list $l$, $L$ for sub-transitions/transitions.
for each Subnet in P do
    \\ Recursively applying Tr1 or Tr2 on the sub-nodes.
    Store MakeTransitions(Subnet, S) in $l$;
end for
for each $sv \in SV$ do
    $comb = Combining(l)$;
    $ot = Matching(sv, comb, hole, v)$;
    Store $ot$ in $L$;
end for
return $L$;
```

applying the Tr1 or Tr2 to generate the sub-transitions. Tr1 is applied on the leaf (pLTS) simply take the pLTS transitions through the given start state and add the predicate in pNet to generate the open transition. When applying Tr2 we use two methods, combining and matching, to generate the new predicate of the open transitions in a hierarchical manner as in this case the composition the subnets brings more constraints for synchronizing behaviors. Beside the new generated predicate, the predicates from the subnets are also added as a conjunction with those new predicates.

Combining: The combining method enumerate all the possible status of the subnets. The result of the method hence call the combination as it present as all the possible combinations of their open transitions. Assume that there is a collection of n subnets $L = s_1, s_2, ..., s_n$, if we notate ot to be the set of open transitions of the subnet s and η means the subnet is not involved. Then the combination COMB, a set of n-tuples, can be computed as an outer product of sets :

$$\texttt{COMB} : (\eta \cup ot_1) \times (\eta \cup ot_2) \times ... \times (\eta \cup ot_n).$$

Matching: $Givenasynchronizationvector\ SV_k \in SV = (a'_i)^{i\in I}(b'_j)^{j\in J} \to v', G_k, k \in K$, and a tuple $C_n \in \texttt{COMB} = (v_i)^{i\in I}, n \in N$ where K and N are the indices of the set elements. As the behavior of subnets should match with the synchronization vector to compose the subnets. The matching method tries matching SV_k and C_n to generate the predicates. According to the definition of predicate, the hole behaviors $Hole$ and result action v are also involved.

$$Matching(SV_k, C_n, Hole, v) = \begin{array}{l} \forall (a'_i)^{i\in I}, (b'_j)^{j\in J}, v'. \\ v_i = a'_i \wedge b_j = b'_j \wedge v = v' \wedge G_k \end{array}$$

which is exactly the new part of predicate for one possible result.

Filtering:

6.2 Management of state variable assignments

In a pLTS, there may be several incoming transitions of some states that assign potentially different values to a state variable. To handle such cases, the algorithm manages the variables together with a list for each pLTS. The list we used contains several triples $<v, S, AssignRH>$

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. As a pragmatic extension to the formal definition, we also manage "global variables", defined in all states of the pLTS.

6.3 Pruning the unsatisfiable results

We use a brute-force method to generate all the possible open transitions in the open automaton, using all possible combinations of synchronization vectors. Naturally this builds some transitions where the predicates express incompatible constraints. Even if having an unsatisfiable (symbolic) transition in the open automaton would not be incorrect, we want to check them for satisfiability, and reduce the number of transitions and states in the automaton. In Fig. 4, we display an example of an unsatisfiable open transition from the result of our example. It shows the case where the controller C1 wants to move the control from P to $(acc(x); Q)$, conducting a δ transition. However, a synchronization vector is chosen that does not match with this action, and we can easily find the contradiction in the generated predicate term "$\texttt{:ra:11:1} = l \wedge \delta = \texttt{:ra:11:1}$".

Checking satisfiability requires some symbolic computation on the action expressions and the predicates, and this may depend on the specific theory of the action algebra. We use the SMT solver Z3 to check the predicate of the result open transitions, it will return whether the predicate is satisfiable or not. The "Modulo Theory" part of SMT solvers is important here, so that the solver can use specific properties of each action algebra.

6.4 Translation to SMTlib

Presentation of algebra **[TODO:Improve this, inserting the main definition from the "Checker" document]** As seen in section 2, the *term algebra* used to express the action expressions and data parameters is left open in the pNet definition.

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 (see the "declare-datatypes" statement in Fig 5).

An algebra presentation contains:

- Sorts: Constants sets of the algebra or types of the data and actions (integer and boolean sort are implicitly declared). There *must* be one sort for actions, and eventually others for the types of data parameters. In the LOTOS example, we have the *Action* sort: `Action =` $\{l, \delta, r, p, q, acc\}$.
- Operators: constructors and predicates for data and action sorts. For instance, in our LOTOS example, we have a generic action constructor "*ACT*", taking as argument an action litteral and a data parameter, declared as: `ACT:< Action, Data >` $\rightarrow$ `Action.`,
- Constants: The satisfiability solver need to know which actions in the pLTS behaviors are constants (like the l, δ, r of the controllers in running example). For a given pNet system, the sort *Action* will include all these constants.

Interaction with Z3 From inside our algorithm code, we submit satisfiability requests to Z3 using its JAVA API. Here for readability, we show the Z3 code using its SMT-LIB input language. As an example in Fig 5, we show the input for checking the transition of Fig. 4. It contains the declaration of the LOTOS action algebra types and constructors, then the declaration of variables, and finally the predicate to be checked, encoded as a set of assertions. The result "unsat" in the output is just what we expected.

To build the input submitted to Z3 for each OT, we translate the algebra *presentation*, the predicates and the variable assignments (the Post part) into Z3 (Java-API) syntax.

Translation of Action algebra presentation **[TODO:Exerpt of the formalisation]** 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* to the *datatypes* declaration. So when we declare the operator ACT we illustrated in section 4.1, it will be just declared together with the constants like :

```
(ACT (action3 Action)(data Int))
```

The *functions* can also be declared independently, it usually applies to functions returning integer or boolean, in the format like:

```
(declare-fun MAX (Int Int) Int)
```

Translating assignments into predicate terms **[TODO:Exerpt of the formalisation]**

State-variable assignments are also treated as a part of the predicates when checking the satisfiability. For each assignement in a *Post* predicate, such as $\{s_0 \leftarrow 1\}$ in section 5, we translate it into an equation $s_0 = 1$. For several assignments of the same variable in the same state, we generate the disjunction of these equations. Correspondingly, we generate a conjunction for the assignments from different variables.

Checking satisfiability Here we declare first all the variables in the predicates, with their sorts. An example of syntax is::

```
(declare-const |:ra:1:1| Action)
```

Then each term of the predicate is an assertion in Z3, as an example, in the former unsatisfiable result in Fig 4, the contradiction is occurred in terms:

```
(:ra:11:1=l) ∧ (delta=:ra:11:1)
```

from which we generate the following two assertions:

```
(assert (= |:ra:11:1| l))   (assert (= delta |:ra:11:1|))
```

The predicate submitted to the Z3 also contains the assignments encodings. In the case of ot_2, it gives the assertion:

```
assert (or (= var_s0 0) (= var_s0 01)))
```

6.5 Simplification of Predicates

6.6 Result of the running example

[TODO:Update: we have 11 satisfiable OTs here] Our tool builds 30 open transitions, out of which 27 are detected unsatisfiable by Z3, we get only 3 results left; the resulting open automaton is shown in Fig. 7, together with its open transitions.

The resulting open transitions represent all the possible movements of the LOTOS example. What can be observed is any actions of the process P from the beginning, until P performs a $\delta(x)$ and the value of x is transmitted to the $acc(x); Q$ giving out a silent action. Such behavior is performed by the ot_1 then ot_2. Then it keeps presenting the behavior of Q, as seen in ot_3.

7 Conclusion and Discussion

In this paper we presented the algorithm for generating an open automaton representing the semantics of an open pNet, and we described the implementation 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. Some of the open transitions obtained at this intermediate step, constructed in a structural manner from all possible combinations of possible logical predicates, may contain predicates which do not represent any possible concrete instantiations. To get rid of these useless transitions, we use the SMT solver Z3 for checking the satisfiability of the predicate in each open transition. In order to do that, we encode into Z3 the action algebra presentation, the a representation of the predicates, before submitting them to the Z3 solver. We implemented our algorithm in the VerCors platform and use some running examples, encoding expressions from various process algebras, to test the algorithm. Our use-cases show that our encoding identifies successfully all unsatisfiable open transitions and that the resulting automaton reflects correctly the expected movements of the encoded process expressions.

Among the questions arising during this work, an important one is the efficiency of Z3 for our needs. Z3 is famous for being very fast for solving very large sets of assertions, and this could be important for us. But we encountered some problems during the implementation the checking method using Z3. For example, if the *functions* and the *datatypes* part of the algebra presentation we submit to the solver are independent with each other, then 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. Also, we need more evidence that dealing with more complex action algebras, that would involve axiomatizing their data structures into Z3 theories, will indeed allow us to decide validity of predicates. We may also try other automatic theorem provers, depending on the structure of the proofs we need.

[TODO:Change perspective: we are working on the formal extensino of BIB architectures, and their semantics in terms of pNets. Then bisimulation and model-checking principles adapted to open pNet systems are under development]

Naturally, our next goals after the generation of the open automata will be to model-check, and to check equivalence of pNets. While model-checking open automata (with a suitable logic including predicates on data) seems easy to define, equivalence checking is more challenging. In previous paper, we have already found the FH-bisimulation, inspired by [?] as a suitable definition. But usual approaches of bisimulation by partition refinement does not work easily with symbolic transitions. In a first step we may want instead to define and implement an algorithm for checking that a given equivalence relation between open automata states is indeed a FH-Bisimulation. Finding automatically a bisimulation relation or the most general bisimulation will be more challenging. It appears also that when comparing different pNets with different structure, strong bisimulation will be too strong a notion, and that we will have to define proper notions of weak equivalence or refinements.

Before that work, we need to reconsider the result open transitions from our algorithm. The ideal result that we showed in the bottom part of Fig. 7 is not the one generated by the algorithm, because the original one contains many intermediate variables, such as fresh local variables of synchronization vectors, and fresh result variables of intermediate open transitions. Such intermediate variables come from the way predicates are generated, and depends on the particular structure of the pNet. But they are not significant in the resulting behavior, and should be eliminated before comparing such behavior with another pNet. To eliminate the intermediate variables in the predicate, we want to define the set of the significant variables that we won't eliminate: the local variables of pLTS transitions, the hole behaviors and the result action at the top level. The result of the elimination will be simpler predicates, that can be successfully compared (e.g. by Z3) within the bisimulation checking algorithm.

A More details on the generation algorithm

A.1 Algebra specification

A.2 Term algebra: type system and static semantics

Our models rely on a notion of parameterized actions, that are symbolic expressions using data types and variables. As our model aims at encoding the low-level behavior 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 unification between two or more action expressions. This will be used in the semantics of synchronization vectors to express various kinds of communication or synchronization 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 sub-terms. The function *vars*(t) identifies the set of variables in a term $t \in \mathcal{T}$.

pNets can encode naturally the notion of input actions as found e.g. in value-passing CCS [**?**] or of usual point-to-point message passing calculi, but it also allows for more general mechanisms, like gate negotiation in Lotos, or broadcast communications.

A.3 Fresh variables

Application of the semantic rules require generating a lot of fresh names, for different kind of variables. We could use a global name generator to guarantee unicity, but at the same time, we also hope them still readable. Here we rename the variables with a regular format using the fresh function. 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 7 (Fresh variable). *The format of* fresh variable *(renaming) is defined as:* “prefix : tree index : coun

- *prefix identifies the kind of the variable. current kinds are: sva (SV action), ra (result action), hb (hole behavior).*
- *tree index is the index of the node containing the variable in the tree-like structure of the pNet.*
- *counter is the current value of the corresponding counter.*

For example, in the running example, the first possible behavior of the hole, it doesn't have a name, only have the prefix “hb”. The hole is the second subnets of the root node so its “tree index” is “12”. The fresh variable name is “`:hb:12:1`”.

A.4 Variable managment

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.

A.5 Filtering the behaviour combinations

There is already able to filter some wrong open transitions during the matching. We can ensure that the synchronization vector is not matched with the subnets if there is only one side not involved. Then we can remove the transitions from the result of matching according to the condition γ where $\gamma = \bigcup_{i \in I}(v_i \in \eta \wedge a'_i \in \eta) \vee (v_i \in \eta \wedge a'_i \in \eta)$.

A.6 Translation to SMTlib input language

In order to submit satisfiability problems to Z3 (for the predicates in open transitions), we need to generate SMTlib programs, from the pNet Algebra presentation and predicates. More precisely, we need to translate to SMTlib:

- the presentation of the action algebra (sorts and operators) that is defined for a given language (process calculus, or high level programming language),
- for a given pNet, the set of local constants (actions or auxiliary data) that are used in the pLTSs,
- for each open transition: the declaration of variables, and the predicate (including action expressions).

During this translation, in order to guarantee that the generated code will cause no runtime errors during parsing and execution, we need to ensure that all objects used in the SMTlib code are properly declared, and that they are correctly typed.

Note that in principle, such an algebra correspond to a given high-level language (e.g. a process algebra), and that the algebra presentation will be defined once and for all in the framework of the pNet semantics of each specific language.

Algebra presentations

Definition 8. *An Algebra Presentation is a pair* $\mathcal{P} =< Sorts, Constrs, Ops >$ *where:*

- *Sorts is a set of Sort names* [6]
- *Constrs is a set of constructor operators, with* Con *the constructor name, and arity:* $arity(Con) = n \in \mathbb{N}$*, and with their signature and their associated selectors, of the form: '*$Con : (sel_1, sort_1), ..., (sel_n, sort_n) - > sort$*'. For each argument, the pair* $(sel_i, sort_i)$ *defines an auxiliary operator of name* sel_i *with signature* $sel_i : sort - > sort_i$*.*
- *Ops is a set of other (auxiliary) operators, with their arity and signature, of the form:* $Op : sort_1, ...sort_n - > sort$
- $Constrs(sortname), Sels(sortname)$*, respectively define the set of constructors and selectors of sort sortname*

All sorts and operator names must be distinct.
Amongst Constructors, those of arity 0 are called constants, and we define $Consts(\mathcal{P}) = \{Con \in Constrs.arity(Con) = 0\}$.

We have a minimal, predefined algebra presentation for all pNets, including three basic sorts *Bool*, *Action* and *Int* and their operators. Table 1 defines these elements.

In addition, and for convenience, we provide one generic construct for parameterized actions named FUN, which can accept any number of arguments of any sort. The result type, though, can only be *Action*, in order to keep the type-checking simple (see next section).

[6] Later we may want to extend this with Sort constructors, like Array or Pair, but this is not needed now

Table 1. Algebra Presentation: predefined Sorts and Operators

Sort	Constructors	Other Operators
Bool	true, false	$\wedge$, $\vee$
Action	FUN	
Int	$0, \{i, -i\}_{i\in\mathtt{Nat}}$	$-$(unary), $+$, $-$(binary), $\times$, $\div$, etc.
for any sort		$=$, $\neq$

For a given language, or for a given use-case, the designer can declare more sorts and operators, using our pNet API. As an example, a (value-passing) CCS action algebra, where we assume a single auxiliary value domain "Data", can be defined as:

Example 2.

- $Sorts_{CCS} = \{Action, Channel, Data, Int, Bool\}$
- $Constrs_{CCS} =$
  $\mathtt{Emit} : 2, \{(Chan\_E : Channel), (Value\_E : Data)\} - > Action$
  $\mathtt{Receive} : 2, \{(Chan\_R : Channel), (Value\_R : Data)\} - > Action$
  $\mathtt{Tau} : 0, \{\} - > Action$
  ... and all predefined operators

Then in pNet API these operators or constants (seemed as non-parameter operators) can be declared as:

```
AlgebraSort Action = new AlgebraSortImpl("Action");
AlgebraSort Channel = new AlgebraSortImpl("Channel");
AlgebraSort Data = new AlgebraSortImpl("Data");
Action.addConstructor("Emit",
        {"Chan_E", "Value_E"}, {Channel, Data});
Action.addConstructor("Receive",
        {"Chan_E", "Value_E"}, {Channel, Data});
Action.addConstructor("Tau");
```

[TODO:[DONE][Xudong]: Give an example of defining *Tau* and *Emit* using the (new version of) the API]

Expressions Once an algebra presentation is defined, we can construct expressions of the term algebra, using variables and operators:

Definition 9. *Let $\mathcal{V}$ be a set of variables. The Term Algebra $\Sigma_{\mathcal{P},\mathcal{V}}$ is the set of:*

- *variables $x \in \mathcal{V}$*
- *terms (or expressions) $op(arg_1, ..., arg_n)$ with op an operator in $\mathcal{P}$ of arity n, and each arg_i an expression.*

Constants and variables in pNets and in open transitions In addition to the objects defined in the Algebra Presentation, there are specific objects that are introduced by the pNet construction, and by the semantic rules used to build the open transitions and their predicates. This includes, for a pNet *pnet*

- $Const(pnet)$: Constants from the pLTSs (controllers) transitions: these are new constant constructors, usualy of sort Action, local to an instance of a controller. The pNet definition requires that all these constants are distinct from each other.

- $SVars(pnet)$: State variables of the controllers. Here also they are required to be distinct from those of other controllers.
- $IVars(pnet)$: Input variables of the controllers.
- $FVars(pnet, ot)$: Several kinds of "fresh" variables, created by application of rule Tr2, during the construction of each open transition *ot*: Action variables for the behavior of holes and the resulting actions of transitions, variables created during the cloning of synchronisation vectors.

All variable sets above include their sort. To define the typing rules for expressions and the translation functions, for any pNet *pnet* and open transition *ot*, we define an extended presentation $\mathcal{P}_{pnet}$ and environment $\Gamma_{pnet,ot}$ that includes all of the objects above:

Definition 10. *Given an algebra presentation* $\mathcal{P}_{pnet} =< Sorts, Constrs, Ops >$ *and a pNet pnet, we construct:*

- *An extended presentation* $\mathcal{P}_{pnet} < Sorts, Constrs \cup Const(pnet), Ops >$.
- *For a given open transition ot, an environment* $\Gamma_{pnet,ot} = SVars(pnet) \cup IVars(pnet) \cup FVars(pnet, ot)$.

Well-formed and well-typed expressions The purpose of this section is to define static semantic notions that will guarantee that the translation to the SMT language will be correct, i.e. will not yield errors at runtime. This includes well-formness (all sorts, operators, variables are defined, and expressions respect the arity of operators), and typing rules.

Definition 11. *Given a presentation* $\mathcal{P}$ *(possibly extended) and an environment* Γ*:*

- Γ *is well-formed if all sorts in* Γ *are defined in* $\mathcal{P}$
- *an expression is well-formed if all its operators are defined in* $\mathcal{P}$*, and used with the proper arity, and all its variables are defined in* Γ
- *an expression is well-typed if it can be typed by the typing rules in table 2*

The following judgment, and the typing rules in table 2 can be used to check both the wellformedness and well-typing of expressions in a pNet or in an open transition, given the corresponding $\mathcal{P}$ and Γ.

$\mathcal{P}, \Gamma \vdash M : A$	M is a well-formed term of type A in $\mathcal{P}, \Gamma$

Remark 1. These rules provide a simple type-checking algorithm: if all variables in an expression are known in Γ, then a bottom-up application of the rules will decide whether the expression is well-typed, and compute the type of each sub-expression.

Map to SMT-LIB language The pNet elements, as defined above, can be full translated into SMT-LIB language, but there are a number of differences in the structure of the models/langages, so the translation is not trivial.

In this section we shall define separately the translation of the (extended) algebra presentation for one pNet (so it will be fixed for the study of one use-case); and the translation of each predicate in the context of an open-transition of this pNet.

Table 2. Type Rules for Open pNets

(Var x)

$$\frac{\mathit{\Gamma} \vdash x : A}{\mathcal{P}, \mathit{\Gamma} \vdash x : A}$$

(Binary operators, e.g.: $\wedge, \vee$ for booleans, $+, -, \times, \div, \leq, \geq$ for integers, etc.)

$$\frac{\mathcal{P} \vdash \mathit{BinOp} :: \mathit{ty1}, \mathit{ty1} \rightarrow \mathit{ty2} \quad \mathit{\Gamma} \vdash x_1 : \mathit{ty1} \quad \mathit{\Gamma} \vdash x_2 : \mathit{ty1}}{\mathcal{P}, \mathit{\Gamma} \vdash x_1 \; \mathit{BinOp} \; x_2 : \mathit{ty2}}$$

(Unary operators, e.g. $\neg$ for booleans, - for integers)

$$\frac{\mathcal{P} \vdash \mathit{UnOp} :: \mathit{ty1} \rightarrow \mathit{ty2} \quad \mathit{\Gamma} \vdash x : \mathit{ty1}}{\mathcal{P}, \mathit{\Gamma} \vdash \mathit{Unop} \; x : \mathit{ty2}}$$

(Polymorphic EQ and NEQ)

$$\frac{\mathcal{P} \vdash A \quad \mathit{\Gamma} \vdash x_1 : A \quad \mathit{\Gamma} \vdash x_2 : A}{\mathcal{P}, \mathit{\Gamma} \vdash x_1 = x_2 : \mathit{Bool}} \qquad \frac{\mathcal{P} \vdash A \quad \mathit{\Gamma} \vdash x_1 : A \quad \mathit{\Gamma} \vdash x_2 : A}{\mathcal{P}, \mathit{\Gamma} \vdash x_1 \neq x_2 : \mathit{Bool}}$$

(Overloaded FUN)

$$\frac{\mathcal{P} \vdash \texttt{FUN} :: A_1, ..., A_n \rightarrow \mathit{Action} \quad \mathcal{P} \vdash A_1 \quad ... \quad \mathcal{P} \vdash A_n \quad \mathit{\Gamma} \vdash x_1 : A_1 \quad ... \quad \mathit{\Gamma} \vdash x_n : A_n}{\mathcal{P}, \mathit{\Gamma} \vdash \texttt{FUN}(x_1, ..., x_n) : \mathit{Action}}$$

(Presentation)

$\mathit{Sorts\&Constructors} \hookrightarrow$ declare-datatypes

$\mathit{Operators} \hookrightarrow$ declare-function

(Predicate)

$\mathit{\Gamma} \hookrightarrow$ declare-const

$\mathit{Pred} \hookrightarrow$ assert

$\texttt{FUN}(x_1, ..., x_n)$

`pNet expression` $\hookrightarrow$ SMTLib expression

...

My plan here is to provide the translation principles as some high-level pseudo code, together with precise definition of the translation functions of a presentation+environment, and of a predicate (including expressions)

In the following definitions, we use abstract functions corresponding to SMTlib/Z3 constructs. In practice they can be implemented either as SMTLib scripting programs, or as calls to the Z3 java API.

Presentation Translation We define here the translation of the algebra presentation, extended with the constant operators collected from the pNet. It produces the declaration of sorts (excepted Bool and Int) with their constructors and selectors as `declare-datatypes`, and the declaration of other operators as `declare-function` constructs. For sorts, we must distinguish the case of mutually defined sorts (e.g. edges and vertices in a graph), that must be declared within a single `declare-datatypes` construct. For this we define:

Definition 12. *Define the (strict) order "is-using" between sorts $S1 is-using S2$ iff $S2$ occurs as the sort of one argument in the constructors of $S1$.*

[TODO:[Xudong: this is certainly obsolete?] At the same time, we have several internal lists (actually hash maps) `exprs`, `funcDecls`, `sortDatatypes` to store the generated Z3 objects with their names for later proofs.]

```
Let Pres = <Sorts, Constrs, Ops> an extended presentation
(i.e. including the constants from the pnet),
- define MySorts = Sorts \ {Bool, Int}
- compute the strongly connected components in the graph
  of MySorts with respect to the relation is-using
- for each SCC in this graph, construct:
  dataype-declaration = TrPresentation (Pres, SCC)
- for each other operator op in Ops, construct:
  function-declaration = TrPresentation (Pres, op)
```

One datatype declaration for each SCC

$$\frac{name = name(Sort) \qquad constrs = Constrs(name) \;\hookrightarrow constrs}{SCC \;\hookrightarrow \texttt{(declare-datatypes () } name \texttt{ (map } \;\hookrightarrow constrs\texttt{))}}$$

Constants:

$$\frac{arity(constr) = 0}{constr \;\hookrightarrow \texttt{name(}constr\texttt{)}}$$

Other constructors:

$$\frac{n = arity(constr) \neq 0 \qquad |sels| = n \qquad sels = BuildSels(constr)}{constr \;\hookrightarrow \texttt{(name(}constr\texttt{) . sels)}}$$

Other operators:

$$\frac{op : sort_1, ..., sort_n : sort}{op \;\hookrightarrow \texttt{(declare-fun name(}op\texttt{) (}sortname_1 ... sortname_n\texttt{) } sortname\texttt{)}}$$

Special case of FUN:

$$\frac{}{FUN \;\hookrightarrow \texttt{map} \;\hookrightarrow CollectFunTypes(pNet)}$$

$$\frac{}{argstypes \;\hookrightarrow \texttt{(declare-fun } BuildFunInstance(argstypes) \texttt{ (map name } argstypes\texttt{) Action)}}$$

Where:

- the *BuildSels* function, for a constructor with n arguments, argument sorts $sort_i$, and selector names sel_i, builds the list $\{(sel_i, sort_i)\}_{i\in[1..n]}$.
- the *CollectFunTypes* function collects all possible instances of the overloaded FUN argument types found in the pNet, as computed by the typing rules; and *BuildFunInstance* use these argument types as suffixes to disembiguate

Example 3. For the CCS presentation above, we would get (in SMTLib syntax):

```
(declare-datatypes Data ())
(declare-datatypes Channel ())
(declare-datatypes ()
    ((Action (Emit (Chan\_E Channel) (Value\_E Data))
             (Receive (Chan\_R Channel) (Value\_R Data))
             Tau )))
```

[TODO:Need to add an exemple with 2 instances of FUN]

Predicate translation Each time we submit each open transition to Z3 module, we translate its predicate into Z3 language format and send it for satisfiability checking. Every term of the predicate is declared as an `assert` in Z3. A constant action or a parameterized expression is easy to get from the internal list storing the objects while all the variables are not declared at the beginning. So we declare them before the submission of a predicate term with the API method conducting `declare-const`.

The second part of the translation function is called for each open transition. More precisely we need here:

```
Let Pres = <Sorts, Constrs, Ops> be the extended presentation
and OT = <Leaves, Holes, Pred, Assign> an open transition.
- compute the environment $\EEnv=\EEnv_{pnet,OT}$ collecting
 all variables used in OT
- check that all pLTS labels (action, guard, assignement) in
 the transition, the OT predicate, and the OT assignments are
 well-formed and well-typed
- for each variable $v$ in $\EEnv$, construct:
  define-const = TrPredicate ($\EEnv,v$)
- turn the predicate into conjunctive normal form
- for each conjunct $P_i$ in the predicate, construct:
  assert = TrPredicate $(P_i)$
```

Variables

$$\frac{\Gamma \vdash x : A}{x \;\hookrightarrow\; \texttt{(declare-const x A)}}$$

Predicate conjunct:

$$\frac{}{P_i \;\hookrightarrow\; \texttt{(assert ...}}$$

To be completed with translation of expressions, including special cases for EQ, NOT_EQ, and FUN

A.7 Elimination of intermediate variables

We apply several structural rules to generate the predicates restricting the composition of the subnets by synchronization vectors. The predicates may contain some redundancy if there are variables in middle level are matching other variables both in lower and higher level at the same time.

We already have the operational semantics of open pNets restricting the open transitions using two rules. The generated equations in the *predicates* contain either intermediate variables or ground ones, but what we want is the equation between ground terms.

We want to reason about the equations to eliminate the redundancy. Considering pNet has a special tree-structure, the replacement between equations should be one-way, the direction is from leaves to the top level, until the root pNet node. And it is easy to figure out the intermediate variables by the constraint rules. While it is not needed figure out if an equation is "true" , it can be left to Z3.

Definition 13 (Intermediate Variable).

When applying Tr1, we do not collect intermediate variables. Though the result action might be the intermediate one between two result action from the SV if it is not the root of the pNet, it will be determined at the higher level instead of at the leaves.

Using the parameters from Tr1:

$$InterVar(OT) = \emptyset$$

When it comes to a pNet node, all the variables from the sources mentioned above should be contained in the $InterVar$ together with the intermediate variables from the subnets. Whatever its subnet is a pLTS or pNets. At the same time, the subnet's result action should be add into the $InterVar$.

If the result of the SV also occurs in its parameters, it means the action from the subnet will be propagated as the result action of the open transition on this level through this SV result action. So it is also the intermediate variable in such situation.

Using the parameters from Tr2:

$$InterVar(OT) = \bigcup_{m \in I_k} InterVar(OT_m) \bigcup_{m \in I_k} \{v_m\} \cup InterVar(SV_k)$$

$$InterVar(SV_k) = \begin{cases} \emptyset, & \alpha_k^{'} \notin \bigcup\limits_{m \in I_k \uplus J_k} \{\alpha_m\} \\ \alpha_k^{'}, & \alpha_k^{'} \in \bigcup\limits_{m \in I_k \uplus J_k} \{\alpha_m\} \end{cases}$$

- *OT is an open transition.*
- *I, J is the set of indices.*
- *k is the indices of the synchronization vectors.*
- *SV is the set of the synchronization vectors.*
- *$Vars()$ is the function that gets all the variables in the object.*
- *α and $\alpha^{'}$ is the element of the the synchronization vector.*

Definition of the intermediate variable shows that elimination should be done every time Tr2 is applied.

Term Rewriting Rules Predicates already have a format declaring what it verifies. According to that previous work, we define the predicates. Let $\langle\!\langle \overline{pNet}, \overline{S}, \overline{SV_k^{k \in K}} \rangle\!\rangle$ be a pNet. Choose a synchronisation vector $SV = clone(SV_k) = (a_i')^{i \in I}, (b_j')^{j \in J} \to v', G_k$, for $k \in K$. The predicate *Pred* relating the actions of the involved sub-pNets and the resulting actions.

Definition 14 (Predicate).

$$Pred(SV, \overline{pNet}, \overline{S}, v) = \begin{array}{l} \forall (a_i')^{i \in I}, (b_j')^{j \in J}, v'. \\ v_m = a_i' \wedge b_j = b_j' \wedge v = v' \wedge G_k \end{array}$$

The new predicates introduce more hypotheses for term rewriting. However, not every variable can be substituted. Here we introduce several rewriting rules.

$$v_m = a'_i \wedge v_m = v'_m \rightarrow a'_i = v'_m \qquad [RL-1]$$

$$v_m = a'_i \wedge v = v' \rightarrow v_m = v \;\; \texttt{if} \;\; a'_i = v', \;\; a'_i, v' \in InterVar \qquad [RL-2]$$

$$v_m = foo'_i(a_1, ..., a_p, ..., a_n) \wedge v = v' \rightarrow v_m = foo'_i(a_1, ..., v, ..., a_n)$$
$$\texttt{if} \;\; a_p = v', \;\; a_p, v \in InterVar \qquad [RL-3]$$

$$b_j = b'_j \wedge v = v' \rightarrow b_j = v \;\; \texttt{if} \;\; b'_j = v', \;\; b'_j, v' \in InterVar \qquad [RL-4]$$

$$b_j = foo'_i(a_1, ..., a_p, ..., a_n) \wedge v = v' \rightarrow b_j = foo'_i(a_1, ..., v, ..., a_n)$$
$$\texttt{if} \;\; a_p = v', \;\; a_p, v \in InterVar \qquad [RL-5]$$

Rule RL-1 applies on when the result action of subnets matching other arguments of the chosen SV. It merges the predicates from the subnets with the new generated predicates. Rule RL-2 shows that if the result of the SV is also in its arguments, we can straightly get an equation between the subnet action and the result action. RL-3 is the situation that there is an expression instead of a variable contains an intermediate argument. The substitution will only occur on the argument. Actually, what happened is similar if there is a hole behavior instead of a subnet result action attending in the equation. So we get RL-4 and RL-5. However, every hole behavior should be kept through out the computing. That's why there no rule similar to the RL-1 for hole behaviors.

B Full examples

[TODO:May be we use two architecture examples here, a simpler one, before the Failure Timer ?]

B.1 A very simple architecture

B.2 Full input for the FailureTimer architecture

B.3 Full output for the FailureTimer open automaton

B.4 Scaling up: several subsystems on the same bus

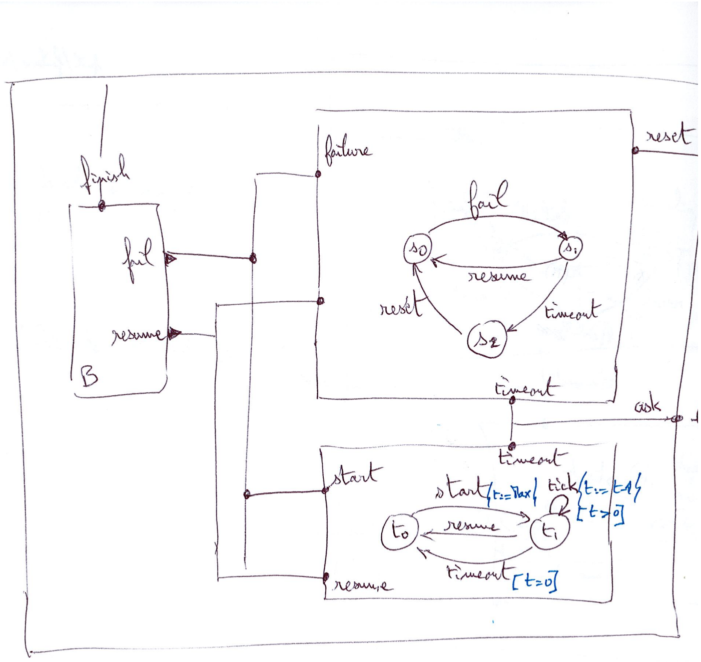
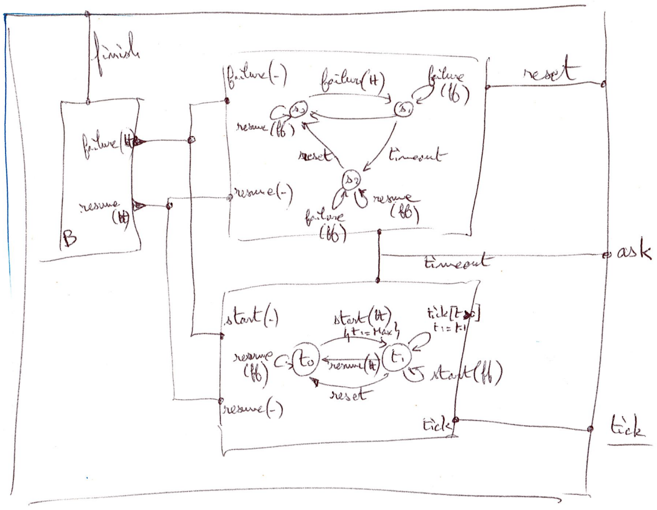


Fig. 1. FailureTimer Architecture in BIP graphical syntax



Pnet structure = $\langle$ Failure, Timer, B $\rangle$

SV_1 $\langle$ failure(b_1), ~~failure~~ start(b_2), failure(b_0) $\rangle$ fail $[b_1 = b_2 \wedge b_1 \vee b_2 \Rightarrow b_0]$

SV_2 $\langle$ resume(b_1), resume(b_2), resume(b_0) $\rangle$ resume $[b_1 = b_2 \wedge b_1 \vee b_2 \Rightarrow b_0]$

SV_3 $\langle$ -, tick, - $\rangle$ tick

SV_4 $\langle$ timout, timeout, - $\rangle$ timeout

SV_5 $\langle$ reset, -, - $\rangle$ reset

SV_6 $\langle$ - - finish $\rangle$ finish

Fig. 2. FailureTimer pNet encoding

$$
\dfrac{
\dfrac{0 \overset{\delta}{\rightarrow}_{C_1} 0 \quad [s_0 = 0]}{C_1 \models \dfrac{0 \overset{\delta}{\rightarrow}_{C_1} 0 \quad [s_0 = 0] \quad \{s_0 \leftarrow 1\}}{\lhd 0 \rhd \overset{\delta}{\rightarrow} \lhd 0 \rhd}}
\qquad
\dfrac{\dfrac{0 \xrightarrow{acc(x)}_{C_2} 1}{C_2 \models \dfrac{0 \xrightarrow{acc(x)}_{C_2} 1}{\lhd 0 \rhd \xrightarrow{acc(x)} \lhd 1 \rhd}}}{\text{acc(x);Q} \models \dfrac{0 \xrightarrow{acc(x)}_{C_2} 1}{\lhd 0 \rhd \xrightarrow{acc(x)} \lhd 1 \rhd}}
}{
\text{PN1} \models \dfrac{0 \overset{\delta}{\rightarrow}_{C_1} 0, 0 \xrightarrow{acc(x)}_{C_2} 1 \quad \{\xrightarrow{hb_{12}}_{P}\} \quad [s_0 = 0 \wedge hb_{12} = \delta(x)] \quad \{s_0 \leftarrow 1\}}{\lhd 00 \rhd \overset{\tau}{\rightarrow} \lhd 01 \rhd}
}
$$

Fig. 3. Proof of ot_2 (interaction of process P and action $acc(x)$) for “P»(acc(x);Q)”

$$
ot = \dfrac{\{0 \overset{\delta}{\rightarrow} 0\}, \quad \{\xrightarrow{:hb:12:1}\}, \quad \texttt{:ra:11:1} = l \ \wedge\ \delta = \texttt{:ra:11:1} \ \wedge\ \texttt{s0} = 0 \\ \wedge\ \texttt{:hb:12:1} = \texttt{a1:sva:1:1} \ \wedge\ \texttt{:ra:1:1} = \texttt{a1:sva:1:1} \ \wedge\ \texttt{a1:sva:1:1} \neq \delta(x), \quad \{\texttt{s0} \leftarrow 1\}}{\lhd 00 \rhd \xrightarrow{\texttt{:ra:1:1}} \lhd 00 \rhd}
$$

Fig. 4. One of the unsatisfiable open transitions in LOTOS running example

Input

```
(declare-datatypes () ((Action p  q  l  r  acc  delta
  (ACT (action3 Action)(data Int))
  (Syncho (action4 Action)))))
(declare-const var_s0 Int)
(declare-const x Int)
(declare-const |:ra:1:1| Action)
(declare-const |:ra:11:1| Action)
(declare-const |:hb:12:1| Action)
(declare-const |a1:sva:1:1| Action)
(assert (or (= var_s0 0) (= var_s0 1)))
(assert (= |:ra:11:1| l))
(assert (= delta |:ra:11:1|))
(assert (= var_s0 0))
(assert (= |:hb:12:1| |a1:sva:1:1|))
(assert (= |:ra:1:1| |a1:sva:1:1|))
(assert (not (= |a1:sva:1:1| (ACT delta x))))
(apply ctx-simplify)
(check-sat)
```

Output (Result)

```
(goals
(goal
  false
  :precision precise :depth 1)
)
unsat
```

Fig. 5. The input of the Z3 solver in SMT-LIB language and the output result

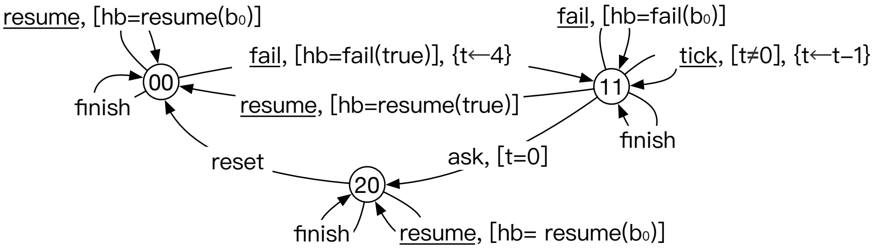


Fig. 6. Open Automaton**[TODO:syntax could be better...]**

[TODO:Detaisl of some OTs of the FailureTimer]

$$ot_1 = \frac{\{0 \xrightarrow{l}_{C_1} 0\},\quad \{\xrightarrow{hb_1}_{P}\},\quad [s_0 = 0 \wedge hb_1 \neq \delta(x) \wedge v_1 = hb_1]}{\lhd 00 \rhd \xrightarrow{v_1} \lhd 00 \rhd}$$

$$ot_2 = \frac{\{0 \xrightarrow{\delta}_{C_1} 0, 0 \xrightarrow{acc(x)}_{C_2} 1\},\quad \{\xrightarrow{hb_2}_{P}\},\quad [s_0 = 0 \wedge hb_2 = \delta(x) \wedge v_2 = \underline{\delta(x)}],\quad \{s_0 \leftarrow 1\}}{\lhd 00 \rhd \xrightarrow{v_2} \lhd 01 \rhd}$$

Fig. 7. The open automaton of the LOTOS formula