Noname manuscript No.
(will be inserted by the editor)

SMT-Based Generation of Symbolic Automata

Xudong Qin · Simon Bliudze · Eric Madelaine · Zechen Hou · Yuxin Deng · Min Zhang

Received: date / Accepted: date

Abstract Open pNets are formal models that can express the behaviour of open systems, either synchronous, asynchronous, or heterogeneous. They are endowed with a symbolic operational semantics in terms of open automata, which allows us to check properties of such systems in a compositional manner. We present an algorithm computing these semantics, building predicates expressing the synchronisation conditions between the events of pNet sub-systems. Checking such predicates requires symbolic reasoning about first order logics and application-specific data. We use the Z3 SMT engine to check satisfiability of the predicates. We also propose and implement an optimised algorithm that performs part of the pruning on the fly, and show its correctness with respect to the original one. We illustrate the approach using two use-cases: the first one is a classical process-algebra operator for which we provide several encodings, and prove some basic properties. The second one is industry-oriented and based on the so-called "BIP architectures", which have been used to specify the control software of a nanosatellite at the EPFL Space Engineering Center. We use pNets to encode a BIP architecture extended with explicit data, compute its semantics and discuss its properties, and then show how our algorithms scale up, using a composition of two such architectures.

Keywords Networks of synchronised automata, software components, symbolic semantics, SMT

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

Xudong Qin(E-mail: marsxd@gmail.com) · Zechen Hou (E-mail: zechen@sei.ecnu.edu.cn) · Yuxin Deng (E-mail: yxdeng@sei.ecnu.edu.cn) · Min Zhang (E-mail: mzhang@sei.ecnu.edu.cn)
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China

Simon Bliudze
INRIA Lille – Nord Europe, 40 avenue Halley, 59650 Villeneuve d'Ascq, France
E-mail: simon.bliudze@inria.fr ORCID: 0000-0002-7900-5271

Eric Madelaine
Université Côte d'Azur, Inria, CNRS, I3S, 06902 Sophia Antipolis, France,
E-mail: eric.madelaine@inria.fr ORCID: 0000-0002-5552-5993

1 Introduction

In the nineties, several works extended the basic behavioural models based on labelled transition systems to formalise value-passing or parameterised systems, using various symbolic encodings of the transitions [16,32,23,34]. In [34], Lin considered value-passing CCS [39], for which he developed a symbolic behavioural semantics, and developed algorithms for checking symbolic bisimulations. Separately, Rathke [24] 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. Later on, symbolic semantics was extended to the π-calculus [40] and decision algorithms for various symbolic bisimulations were proposed [33,19]. However, no practical verification platform has been developed to use this kind of approaches to provide proof methods for value-passing processes or open process expressions.

Parameterised networks of synchronised automata (pNets) were proposed to give a behavioural specification formalism for distributed systems, synchronous, asynchronous, or heterogeneous. They are used in VerCors [25], a platform for designing and verifying distributed systems, as the intermediate language for various high-level languages. The high-level languages in VerCors formalise each component of a distributed system and their composition. The model of pNets provides the core low-level semantic formalism for VerCors, and is made of a hierarchical composition of (value-passing) automata, called parameterised labelled transition systems (PLTS), where each hierarchical level defines the possible synchronisation of the lower levels. Traditionally, pNets have been used to formalise fully defined systems or softwares. But we also want to define and reason about incompletely defined systems, like program skeletons, operators, or open expressions of process calculi. The open pNet model addresses this problem, using *holes* as process parameters, representing *unspecified* subsystems. The open pNet model was developed in a series of papers [26,27] in which many examples have been introduced showing its ability to encode the operators from some other algebras or program skeletons. The operational semantics of an open pNet is defined as an open automaton in which open transitions contain logical predicates expressing the relations between the behaviour of the holes and the global behaviour of a system. In previous work, only a sketch of a procedure allowing for computing these semantics was presented, together with a proof of finiteness of the open automaton, under reasonable hypothesis on the pNet structure.

Implementing these semantics raised several challenges:

- to obtain a tool that could be applied to pNets representing various languages, in particular various action algebras, with their specific decision theories;
- to separate clearly the algorithm that generates the transitions of an open automaton from the symbolic reasoning part that uses an SMT engine to check the satisfiability of the predicates generated by the algorithm;
- to build a prototype of the algorithm, and validate the approach on our basic case-studies, meanwhile, understand the efficiency of the interaction with the SMT solver.

In the long term, we would like to check the equivalence between open systems encoded as pNets. The equivalence between pNets is FH-bisimulation [27], a dedicated version of symbolic bisimulation taking predicates into account when matching open transitions. We foresee that the interplay with the SMT solver that we use here for

satisfiability of the predicates of open transitions will be similar to what we need when proving (symbolic) equivalence between open pNets.

Contribution. In the current work we contribute in the following aspects.

- We present an open automaton generation algorithm. We implement a full working prototype, within the VerCors platform. In the process, we improve the semantic rules from [27], and add features in the algorithm to deal with the full model, including management of variables and assignments.
- We implement the interaction between our algorithm and the Z3 SMT solver, for checking the satisfiability of the transitions generated by the algorithm.
- We define a strategy checking satisfiability of open transitions in the course of the residual algorithm, and show the correctness of the optimised algorithm w.r.t. the original one.
- We show the usage of this approach on various use-cases from different settings, including an industry-inspired case-study, namely one architectural pattern extracted (and extended) from the BIP specification of a nanosatellite on-board software.

Related work. As mentioned above, early attempts were made by Lin and others to define symbolic bisimulations for process calculi such as value-passing CCS and the π-calculus, and design algorithms to compute the most general condition under which two processes are bisimilar [23,34,35]. The idea of symbolic semantics was also generalised to the setting of quantum CCS [20]. However, there is no algorithmic treatment of the symbolic systems developed by interacting with automatic theorem provers. The closest work is the one already mentioned by Rathke [24], who developed the symbolic bisimulation for a calculus of broadcasting system (CBS). CBS is similar to classic process calculi such as CCS and CSP [29], but communicating by broadcasting values, transmitting values without blocking. That makes the definition of the symbolic semantics and bisimulation equivalence different from the classic works.

From a broader perspective, after the seminal work of Hennessy, Lin, and their colleagues on value-passing systems with assignments, many different works addressing various classes of infinite-state systems and/or parameterised topologies have been published, using combinations of approaches, often including predicate abstraction and SMT satisfiability (e.g. [1,11,13,14,22]). Among these works, several have demonstrated the usefulness of SMT engines (either Z3 or Yikes) as servers for solving verification conditions of algorithms, especially for large case-studies (e.g. [12,15]). With respect to these, we use symbolic representations not only to get a finite representation of infinite spaces, but also to express the (data-sensitive) synchronisations with the environment, making our models suitable for compositional verification.

For other applications, such as the analysis of programming languages, there exist dedicated platforms using external automatic theorem provers (ATPs) or automatic tactics from interactive theorem provers (ITPs), to perform symbolic reasoning, and for example to discharge some subgoals in the proofs. Tools like Rodin [17,18] have already integrated several provers, like Z3, as modules for proving the proof obligations generated from a user model. The prover we use, which also happens to be Z3, is developed by Microsoft Research based on the satisfiability modulo theories framework (SMT), and is 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 result pruning and bisimulation checking in our algorithm, as an alternative to Z3, such as CVC4 [5], Coq [2] or others.

BIP (Behaviour-Interaction-Priority) [7] is a framework for component-based design of concurrent software and systems. In particular, the BIP tool-set comprises compilers for generating C/C++ code, executable by linking with one of the dedicated engines, which implement the BIP operational semantics [4]. This approach ensures that any property, shown to hold on a given BIP model, will also hold by construction on the generated code. BIP architectures [3] formalise design patterns, which enforce global properties characterising the coordination among the components of the system. They provide a compositional approach, ensuring correctness by construction during the design of BIP models. In [3], it was shown that application of architectures is compositional w.r.t. safety properties, i.e. when several architectures are applied, each enforcing a safety property, the resulting system satisfies their conjunction. However, [3] did not provide any mechanism for verifying the correctness of individual architectures. The encoding presented in this paper provides such a mechanism.

Structure. In Section 2 we give a description and a formal definition of the pNet model. In Section 3 we briefly recall the operational semantics of pNets. In Section 4 we present an algorithm to compute this semantics, including the interaction with Z3. In Section 5 we give a few use-cases, in particular, a BIP architecture from the nanosatellite case-study. Finally, we conclude and discuss perspectives in Section 6.

2 Preliminaries

We first introduce pNets and some notations, then give the formal definition of pNet structures, together with an operational semantics for open pNets.

A pNet has a tree-like structure, 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 their set of possible actions, named *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. The symbol $a_i^{i\in I}$ denotes a family of elements a_i indexed over the set I. When this is unambiguous, we shall use notations for sets, and typically write "indexed sets over I" which formally means multisets, and write $x \in a_i^{i\in I}$ to mean $x = a_i$ for some $i \in I$. An empty family is written $\emptyset$. We denote by $\overline{a}$ a family when the indexing set is irrelevant. The symbol $\uplus$ stands for the disjoint union of indexed sets.

Term algebra. Our models rely on a notion of parameterised actions that are symbolic expressions using data types and variables. As we want to encode the low-level behaviour of possibly very different programming languages, we do not impose one specific algebra for denoting actions, nor any specific communication mechanism. So we leave unspecified the constructors of the algebraused to build 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 synchronisation vectors to express various kinds of communication or synchronisation mechanisms.

Table 1 Algebra presentation: predefined sorts and operators

Sort	Constructors	Auxiliary Operators
Bool	`true`, `false`	$\wedge$, $\vee$, $\neg$, $\Longrightarrow$, $=$, $\neq$
Action	`Synchro`, `FUN`	
Int	$0, \{i, -i\}_{i \in \texttt{Nat}}$	$-$`(unary)`, $+$, $-$`(binary)`, $\times$, $\div$ etc.
Extension for the Enable use-case of Example 1		
Action	`l`, `r`, δ, `acc`	

Formally, we assume the existence of a term algebra $\mathcal{T}_{\Sigma,\mathcal{V}}$, where Σ is the signature of the data and action constructors, and $\mathcal{V}$ a set of variables. Within $\mathcal{T}_{\Sigma,\mathcal{V}}$, we distinguish a set of data expressions $\mathcal{E}_{\mathcal{V}}$, including a set of Boolean expressions $\mathcal{B}_{\mathcal{V}}$ ($\mathcal{B}_{\mathcal{V}} \subseteq \mathcal{E}_{\mathcal{V}}$). On top of $\mathcal{E}_{\mathcal{V}}$ we build the action algebra $\mathcal{A}_{\mathcal{V}}$, with $\mathcal{A}_{\mathcal{V}} \subseteq \mathcal{T}_{\mathcal{V}}$. Action terms can use both data and action 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 [39] or of usual point-to-point message passing calculi, but they also allow for more general mechanisms, like gate negotiation in LOTOS [30], or broadcast communications.

2.1 Algebra Presentations

In practice, the parameterisation of the pNet model by some specific action algebra is realised by the definition of a many-sorted algebra presentation. It will be used to check the well-formedness of a pNet system, and to define the translation of the pNet semantics into the SMT engine input language (see [6]).

Definition 1 An *algebra presentation* is a triple $\mathcal{P} = \langle Sorts, Constrs, Ops \rangle$, where
- $Sorts$ is a set of data sorts.
- $Constrs$ is a set of *constructor operators*: for each $Con \in Constrs$, $arity(Con) = n \in \mathbb{N}$ is its arity and $Con : (sel_1, sort_1), \ldots, (sel_n, sort_n) \rightarrow sort$ is its signature with the associated selectors. For each argument, the pair $(sel_i, sort_i)$ defines an auxiliary operator of name sel_i with signature $sel_i : sort \rightarrow sort_i$.
- Ops is a set of *auxiliary operators*, with their signatures in the form: $Op : sort_1, \ldots, sort_n \rightarrow sort$.
- $Constrs(sortname)$ and $Sels(sortname)$ are, respectively, the sets of constructors and selectors of the sort $sortname$.

Constructors of arity 0 are called *constants*, and denoted by $Consts(\mathcal{P})$.

In Table 1 we give an algebra presentation. Sorts `Bool` and `Int` are predefined with standard operators. Sort `Action` is similar, with a constructor `Synchro` denoting a synchronised action, i.e. an "internal" action that cannot be further synchronised with the environment. It also comes with an overloaded `FUN` constructor, used to build action expressions with arguments, which will be instantiated to the required sorts for a given pNet.

The definition of an algebra presentation and a set of variables $\mathcal{V}$ fix the term algebra elements $\mathcal{T}_{\Sigma,\mathcal{V}}, \mathcal{B}_{\mathcal{V}}$, and $\mathcal{A}_{\mathcal{V}}$.

2.2 Running example: the Enable operator

To illustrate our definitions and results, we will use a small running example, coming from the specification language LOTOS [30]. Suppose we have two unspecified processes P and Q. In the Enable expression "$P >> Q$", an `exit` statement within P terminates the current process and passes the control to Q, carrying a value x that is captured by the `accept` statement of Q. In the next section we will define pNet encodings of the Enable operator. Let us start by defining the algebra presentation that will be used in these pNets.

Example 1 (The "Enable" Operator) In Fig. 1 we give two possible pNet encodings of the "Enable" operator, which are state-oriented and data-oriented, respectively. The first one is called *EnableState*, and uses a controller *CState* with three simple control actions, l, r and δ, to synchronise the behaviour of P and Q. Here we use a simple action algebra, containing two constructors $\delta(x)$ and $acc(x)$, for any possible data type of the variable x, corresponding to the LOTOS statements `exit` and `accept`. To construct the algebra presentation, we extend the triple $\mathcal{P} = \langle Sorts, Constrs, Ops\rangle$ shown in Table 1 with four new action constant constructors l, r, δ and acc.

This shows an example of the generic `FUN` constructor used to construct expressions (of the Action sort, or of any data sort) in any algebra presentations. Here `FUN` will be instanciated e.g. as, $\delta(x)$ in the form of $\texttt{FUN}_{Action,Data}(\delta, x)$ where x is a variable of type *Data*. Note that the same action constructor δ is also used as a constant action in the controller *CState*.

The *EnableData* pNet uses the same algebra presentation.The main difference between *EnableState* and *EnableData* lies in the construction of their controllers *CState* and *CData* that we will discuss in the next section.

2.3 The (Open) pNets Core Model

A PLTS is a labelled transition system with variables, which 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 transitions targeting this state. Variables of the initial state must be initialized. 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 constants, *op* over operators, and x over variable names. Action terms are given by the following grammar:

$$\begin{array}{rll} \alpha \in \mathcal{A} ::= & a(p_1, \ldots, p_n) & \text{action terms} \\ p_i ::= & \mathit{Expr} & \text{parameters} \\ \mathit{Expr} ::= & \mathit{Value} \mid x \mid \mathit{op}(\mathit{Expr}_1, \ldots, \mathit{Expr}_n) & \text{expressions} \end{array}$$

Definition 2 (PLTS) Given a term algebra $\mathcal{T}_{\Sigma,\mathcal{V}}$, a PLTS is a tuple $\langle S, s_0, \rightarrow\rangle$, where

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

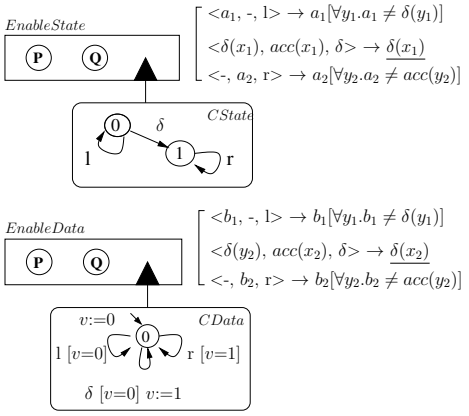

Fig. 1 Two pNet encodings for Enable

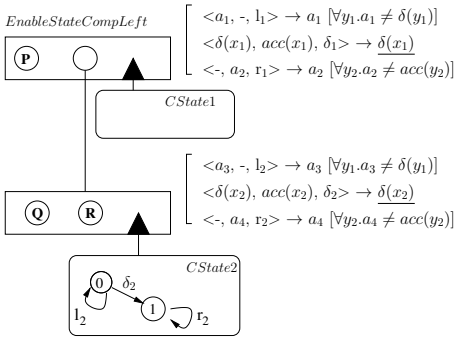

Fig. 2 Composed pNet for "$P >> (Q >> R)$"

Example 2 (PLTS of the Controllers in the "Enable" Operator)
The PLTS of the controller $CState$ from Example 1 is $\langle S, s_0, \rightarrow \rangle$, where:

$$S = \{0, 1\}; s_0 = 0; \rightarrow = \{0 \xrightarrow{l} 0, 0 \xrightarrow{\delta} 1, 1 \xrightarrow{r} 1\}.$$

And the PLTS of the controller $CData$ is $\langle S', s'_0, \rightarrow \rangle$ where:

$$S' = \{0\}; s'_0 = 0; vars(0) = \{v\}; \texttt{init}\ \{v := 0\};$$

$$\rightarrow = \{0 \xrightarrow{l,[v=0]} 0, 0 \xrightarrow{\delta,[v=0],\{v:=1\}} 0, 0 \xrightarrow{r,[v=1]} 0\}.$$

The PLTSs contained in $EnableStateCompLeft$ are similar to the PLTS $CState$, but their action constants have been renamed, ensuring disjoint state variable sets.

Now, we define pNet nodes as constructors for hierarchical 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 (i.e. unknown in the environment).

A composite pNet consists of a set of sub-pNets, each exposing a set of actions. The relation between the actions of a pNet and those of its sub-pNets is given by *synchronisation vectors*, which synchronise one or several internal actions, and expose a single resulting global action.

Definition 3 (pNets) A pNet is a hierarchical structure where leaves are PLTSs and holes: $pNet \triangleq PLTS \mid \langle pNet_i^{i \in I}, J, SV_k^{k \in K} \rangle$, with I, J, K potentialy infinite, where

- $pNet_i^{i \in I}$ is the family of sub-pNets.
- J is a set of indexes, called *holes*. I and J are *disjoint*: $I \cap J = \emptyset$, $I \cup J \neq \emptyset$.
- $SV_k^{k \in K}$ is a set of synchronisation vectors, where $K \in \mathcal{I}_\mathcal{V}$ with $\mathcal{I}_\mathcal{V}$ being the set of indexed sets whose ranges depend on the variables in $\mathcal{V}$. It is required that each SV_k is a pair $(\alpha_l^{l \in I_k \uplus J_k} \rightarrow \alpha'_k,\ g_k)$, where $vars(\alpha'_k) \subseteq \bigcup_{l \in I_k \uplus J_k} vars(\alpha_l)$, $\alpha'_k \in \mathcal{A}_\mathcal{V}$, $I_k \subseteq I$, and $J_k \subseteq J$. The global action of a vector SV_k is α'_k. The Boolean expression g_k, with $vars(g_k) \subseteq \bigcup_{l \in I_k \uplus J_k} vars(\alpha_l)$, is a guard associated to the vector.

Example 3 (pNet of the "Enable" Operator) Consider the pNet *EnableState* shown in Fig. 1, upper part. It has two holes P and Q, and one PLTS $CState$. Note that our graphical syntax presents pNet nodes and synchronisation vectors as finite arrays, where the index sets follow the positions in the arrays. In this example we use *synchronised actions*, to express actions that cannot be further synchronised, similar to the silent action τ of CCS. We denote them as any action expression but with the text underlined, e.g. $\underline{\delta(x_1)}$. Such synchronised actions do not play any special role for defining the semantics of pNets, but as one can expect, they will be crucial for defining weak equivalences. The pNet can be given as follows:

- $pNet = \{CState\}$;
- $J = \{P, Q\}$;
- $SV =$ as listed on the right side of each pNet node.

Consider the first vector of *EnableState*, written as $\langle a_1, \text{-}, l\rangle \rightarrow a_1[\forall y_1.a_1 \neq \delta(y_1)]$. It denotes a possible behaviour of $P >> Q$, for any possible action of the hole P that respects the vector predicate, that is, any action that does not match $\delta(y_1)$, for any value of the variable y_1. Remark that the data-oriented encoding in pNet *EnableData* has the same synchronisation vectors, up to the usual alpha conversion. Only the controller $CState$ has been replaced by $CData$.

Example 4 (pNet of the Composition of "Enable" Operators) Now consider the pNet $EnableStateCompLeft$ in Fig. 2; the processes P, Q and R are three holes of the pNet. It has been obtained by replacing the second hole in pNet *EnableState* by another pNet representing $Q >> R$. Technically, the action constants in the PLTS $CState1$ and $CState2$ have been renamed to avoid clashes, and the variables in the vectors are also given different names.

3 Operational Semantics of Open pNets

The semantics of an open pNet will be defined as an open automaton, which is an automaton where each transition represents the composed transitions of several LTSs with the actions of some holes; a transition occurs if its predicates hold, and can involve a set of state modifications. Each state of an open automaton has a set of *state variables* that can be assigned by incoming transitions.

Definition 4 (Open transitions) An *open transition* (OT) over a set J of holes and a set of states $\mathcal{S}$ is a structure of the form:

$$\frac{\{\xrightarrow{\beta_j}_j\}^{j\in J}, Pred, Post}{s \xrightarrow{\alpha} s'},$$

where $s, s' \in \mathcal{S}$, the β_j, α are action expressions, β_j is an action of the hole j, and α is the resulting action of the OT. *Pred* is a predicate over the variables of the terms, labels, and states s_i, β_j, s, α. *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 is a variable of s' or s'_i, whereas e_k is an expression over other variables of the open transition.

Definition 5 (Open automaton) An *open automaton* is a tuple $\langle J, \mathcal{S}, s_0, \mathcal{T}\rangle$, where

- J is a set of indices,
- $\mathcal{S}$ is a set of states and $s_0 \in \mathcal{S}$ the initial state,

- $\mathcal{T}$ is a set of open transitions and, for each $t \in \mathcal{T}$, there exist J' with $J' \subseteq J$, such that t is an open transition over J' and $\mathcal{S}$.

The *states* and the shape of *predicates* in the transitions of an open automaton representing the semantics of a pNet have the following specific structure.

States of open pNets. A state of an open pNet is a tuple of the states of its leaves. We denote tuples in structured states in the form $\triangleleft \ldots \triangleright$. For any pNet p, let $\langle S_i, s_{i0}, \rightarrow_i \rangle^{i \in L}$ be the set of PLTSs at its leaves, then $States(p) = \{\triangleleft s_i^{i \in L} \triangleright \mid \forall i \in L. s_i \in S_i\}$. A PLTS could be its own single leave: $States(\langle S, s_0, \rightarrow \rangle) = \{\triangleleft s \triangleright \mid s \in S\}$. The initial state is defined as: $InitState(p) = \triangleleft {s_{i0}}^{i \in L} \triangleright$.

Predicates. Let $\langle \overline{pNet}, J, SV_k^{k \in K} \rangle$ be a pNet. Consider a synchronisation 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:

$$MkPred(SV_k, a_i^{i \in I}, b_j^{j \in J}, v) \Longleftrightarrow SV_k = (a_i)^{i \in I}, (b_j)^{j \in J} \rightarrow v.$$

Example 5 (Open transitions) Transition OT_2 of the pNet $EnableStateCompLeft$ (shown in Figure 2) is generated by combining the first vector of its root node with the second vector of its second node. The global states show that controller $CState1$ is changing from state 0 to 1.

$$OT_2 = \frac{\{\xrightarrow{\delta(x_2)}_P \quad \xrightarrow{acc(x_2)}_Q\} \quad [\forall y_1. acc(x_2) \neq \delta(y_1)]}{\triangleleft 0, 0 \triangleright \xrightarrow{\delta(x_2)} \triangleleft 1, 0 \triangleright}$$

If we use the data-oriented pNet $EnableData$ from Figure 1 to build a similar $EnableDataCompLeft$ expression "$P >> (Q >> R)$", the corresponding open transition would be:

$$OT_2' = \frac{\{\xrightarrow{\delta(x_2)}_P \quad \xrightarrow{acc(x_2)}_Q\} \quad [v_1 = 0 \wedge v_2 = 0 \wedge \forall y_1. acc(x_2) \neq \delta(y_1)] \quad \{v_1 \leftarrow 1\}}{\triangleleft 0, 0 \triangleright \xrightarrow{\delta(x_2)} \triangleleft 0, 0 \triangleright}$$

where v_1 and v_2 are respectively the state variables of the first and second $EnableData$ pNets. Indeed the global state space of this version is reduced to $\triangleleft 0, 0 \triangleright$, and the state change is replaced by the assignment "$v_1 \leftarrow 1$".

Structural Semantic Rules. The semantics of pNets in terms of open automata has been defined in [27], in the form of two structural rules, one for PLTSs, and the other for pNet nodes. These rules are slightly improved, adding guards in the synchronisation vectors and syntax for universal quantifier in the guards (see [42])[1].

In [27] we also proved the following result, which ensures the termination of our semantic construction algorithm. Note that this theorem and its proof were part of previous versions of this work, when Algorithm 1 did not include the fixed point refinement computation. Proof of termination of the fixed point itself will be discussed as part of the proof of Theorem 3.

[1] For convenience, we provide these rules and the proof of the finiteness theorem in Appendix A

Theorem 1 (Finiteness) *Let pnet* $= \langle \overline{pNet}, \overline{S}, SV_k^{k \in K} \rangle$ *be an open pNet with leaves* $l_i^{i \in I}$ *and holes* $h_j^{j \in J}$*, where the sets* I *and* J *are finite. Suppose the synchronisation vectors of all pNets included in pnet are finite, and* l_i *has a finite number of state variables for each* $i \in I$*. Then the semantics of pnet is an open automaton with finitely many states and transitions.*

Notice that all the elements of such pNets and open automata are symbolic, and they can represent many classes of unbounded systems.

4 Generation of Open Automata

In this section we describe an algorithm implementing the pNet semantics, its new "on-the-fly" version, and the interaction with the Z3 SMT solver. Under the hypotheses of Theorem 1 both algorithms terminate.

Algorithm 1 Open Automaton Generation

Input: A pNet P (can be a PLTS, but not a hole)
1: Initialize sets $U = \{InitState(P)\}$ and $E = \emptyset$, for unexplored and explored global states, respectively; $L = \emptyset$ for the resulting OTs;
2: **while** !$isEmpty(U)$ **do**
3: Choose S in U; remove S from U, add S to E;
4: $OTs = MakeTransitions(P, S)$;
5: **for** each $OT \in OTs$ **do**
6: Add OT to L;
7: Let S' be the target state of OT; add S' to the unexplored states if necessary:
8: **if** $(S' \notin U \cup E)$ **then** Add S' into U;
9: **end for**
10: **end while**
11: $L1 = filterUnsatTransitions(L, Assignments)$;
12: $OA_1^0 = makeReachableSubAutomaton(InitState(P), L1)$;
13: **repeat** $OA_1^n = RefineReachableSubAutomaton(OA_1^{n-1})$
14: **until** $OA_1^n = OA_1^{n-1}$; \\ Fixed point reached
15: **return** OA_1^n;

Algorithm 1 starts with an open pNet and builds its set of open transitions. Its main loop is a classical residual algorithm: starting from the initial global state, it picks a state in an unexplored set, computes all possible OTs, and adds the target states in the unexplored set, until this set is empty.

The inside loop (*MakeTransitions* method) applies recursively the semantic rules following the structure of the pNet. When applying a rule to a PLTS at the leaves, we simply take the PLTS transitions of the corresponding local state and use the semantic rules to build the OT. When applying a rule to a pNet node we use two methods, *combining* and *matching*, to generate the open transitions in a hierarchical manner, as shown in Algorithm 2. This method directly manages the holes of the node, so *MakeTransitions* is never called on a hole.

Before the SAT checking step, the set L may contain symbolic transitions that do not have valid ground instantiations. This is not logically incorrect, but is naturally not an optimal representation. So (line 11) we prune these UNSAT transitions: at the end of the open automaton generation, the predicate of each OT is translated

into SMTlib assertions (see Section 4.2), and then checked for satisfiability (function *filterUnsatTransitions*). Then we compute (lines 12-14) the set of reachable transitions and states to build the final open automaton (more details in Section 4.3).

Algorithm 2 *MakeTransitions*() for a pNet node

Input: a pNet node P with subnets $\overline{sn}$ and holes $\overline{hole}$; a global state S.
1: Initialize empty list l and set L for sub-transitions and transitions, respectively;
2: **for** each *Subnet* in $\overline{sn}$ **do** \\ Recursively apply the semantic rules on the subnets
3: Store *MakeTransitions*(*Subnet*, S) in l;
4: **end for**
5: $\overline{comb} = Combining(l)$;
6: **for** each $sv \in SV$ and each $comb \in \overline{comb}$ **do**
7: $ot = Matching(sv, \overline{comb}, \overline{hole})$;
8: **if** (ot is defined) **then** Store ot in L; \\ if *Matching*() succeeds
9: **end for**
10: **return** L;

Combining. The combining method enumerates all the possible behaviours of the subnets as all the possible combinations of their open transitions. Assume that there is a collection of n subnets. We denote by $\overline{ot}_i$ the set of open transitions of the i-th subnet (obtained in line 3 of the algorithm); "$-$" means that the subnet is not involved. The combination $\overline{comb}$, a set of n-tuples, is the cartesian product:

$$\overline{comb} = (\{-\} \cup \overline{ot}_1) \times (\{-\} \cup \overline{ot}_2) \times \cdots \times (\{-\} \cup \overline{ot}_n).$$

Matching. The *Matching* method builds the OTs of a pNet node from those of its subnets. For each synchronisation vector and each possible combination of behaviours of the subnets, as generated by the *Combining* method, it builds the corresponding open transition. Here, we only detail the construction of the predicate. Suppose that $sv = \left((a'_i)^{i\in I}(b'_j)^{j\in J} \rightarrow v'\right) \in SV$ is a synchronisation vector, G_k is its guard, and $comb = (ot_i)^{i\in[1,n]} \in \overline{comb}$ is a tuple of open transitions such that, for each $i \in [1,n]$, either $ot_i = -$, or the result action of ot_i is a_i. Suppose further that $hole = (b_j)^{j\in J}$ is the hole behaviour and v is a fresh variable, representing the result action of the OT under construction. Then we can build the predicate:

$$MkPred(sv, comb, \overline{hole}, v) = (\forall i \in I, a_i = a'_i) \wedge (\forall j \in J, b_j = b'_j) \wedge (v = v') \wedge G_k.$$

Filtering. While matching a vector with a combination tuple, *Matching* tries to filter out some incompatibilities as early as possible. If some of the subnet actions are marked inactive in the vector while some of the behaviours at the corresponding positions in the chosen combination are active, the matching would fail and the algorithm can filter it.

The matching would also fail if the active actions in the vector and the active behaviours in the chosen combination are not matched by pattern-matching or unification. Such case will be checked later, together with the guards collected from synchronisation vectors and PLTS transitions, using the satisfiability check in the SMT engine.

4.1 Management of State Variables and Assignments

In a PLTS, there may be several incoming transitions that assign potentially different (symbolic) values to a state variable. To handle such cases, the algorithm manages a list of assignment expressions for each PLTS state. The only assignments for state variables that happen at the PLTS level are as follows: every time the $MakeTransitions$ method is applied on a PLTS, it collects the expressions from the $Post$ in each transition.

For a global state in the open automaton, the set of state variables (which may be used in an outgoing transition) is the disjoint union of the sets of state variables of the individual PLTS states constituting this global state. The final assignment sets are assembled when building the open automaton from the reachable transitions, see below.

4.2 Pruning the Unsatisfiable Results

Our matching/filtering strategy builds some transitions where the predicates express incompatible constraints. Such constraints come from the mismatching of the actions by pattern-matching or unification, including the guards collected from PLTSs and synchronisation vectors, which are all present in the predicates. Even if having an unsatisfiable (symbolic) transition would not be incorrect, we choose to minimize the open automaton (i.e. its number of transitions and states), by checking the predicates for satisfiability.

Example 6 (Unsat OT)

$$OT_{19} = \frac{\{\xrightarrow{a_1}_P \quad \xrightarrow{acc(x_2)}_R\} \quad [a_1 = \delta(x_2) \wedge \forall y_1.a_1 \neq \delta(y_1)]}{\triangleleft 2, 0 \triangleright \xrightarrow{\delta(x_2)} \triangleleft 3, 0 \triangleright}$$

Here we display an unsatisfiable open transition from the open automaton of Fig. 2. It shows the case where processes P and R "try" to communicate directly, but this is made impossible by the combination of chosen synchronisation vectors and their guards. This mismatch is materialised by the predicate fragment "$a_1 = \delta(x_2) \wedge \forall z.a_1 \neq \delta(z)$". In Section 4.5 we will show how this is encoded and checked using the SMT engine.

Checking satisfiability requires some symbolic computation on the action expressions and the predicates, which may depend on the specific theory of the action algebra datatypes. The "Modulo Theory" part of SMT solvers is important here, as the solvers can use specific properties of each data type in the action algebra.

In Algorithm 1, functions *filterUnsatTransitions* (in line 11) *RefineReachableSubAutomaton* (in line 13) make use of the Z3 engine to implement the satisfiability checking, for all the OTs computed that far. For a transition

$$\frac{\{\xrightarrow{\beta_j}_j\}^{j \in J}, Pred, Post}{s \xrightarrow{\alpha} s'}$$

we need to build a predicate encoding both the OT predicate and the information available on the possible assignments of the initial state s. This is expressed as

$$Pred_l \wedge \bigwedge_{v \in vars(s)} \{ \bigvee_{Assign \in assigns(v,s)} Assign \}$$

where each *Assign* represents one of the possible (symbolic) values of variable v.

Each call to the SMT engine will return either SAT or UNSAT results, or may fail due to time bounds. Also, the accuracy of a "SAT" result may depend on the characteristics of the specific theory used to axiomatise the data domains and operators. Such theories may be intrinsically undecidable, or may not be properly handled by the SMT engine (e.g. for recursive datatypes). However, while building the open automaton, which is a symbolic representation of the pNet behaviour, it is not logically incorrect to keep potentially unsatisfiable transitions, so we will treat undecidable results, and time bounds failures, as SAT. So the open automaton may have more transitions and states than the theoretical minimum.

This is not a problem in itself when constructing the automaton, but a similar limitation will arise later when using open automata as the input of model checking or equivalence checking algorithms. Note that decidability results for these will be dependent on the decidability of the SMT theories used, as is proved e.g. in [27], for bisimulations.

4.3 Building the Open Automaton

As the removal of the unsatisfiable transitions may entail some transitions without preceeding transitions, the set $L1$ might contain unreachable transitions. To remove those transitions, the last step of the algorithm (lines 12-14) builds the final open automaton via a traversal of $L1$, starting at the initial state $InitState(p)$ as defined in page 9. More precisely, this is a residual algorithm building a graph from the reachable transitions in $L1$. Each state in the graph is decorated with a set of its local variables, and a list of their possible assignments, gathered during the traversal using the "Post" parts of the OTs.

A further reduction may now be gained by using the information from these sets of possible assignments: adding this information when checking satisfiability of transitions leaving a given state may result in more UNSAT transitions. The set of possible assignments computed by the algorithm is an over approximation of possible values of the variables of such state, so narrowing it means we get a better approximation, and potentially more information, which can be used in the SAT checking of subsequent transitions. In turn, this may reduce the possible assignments of the corresponding target states. In Algorithm 1, we iterate this procedure (lines 13-14), each iteration reducing the set of transitions and the sets of assignments; the final open automaton is the fixed point of this iteration.

In practice, this could be expensive as it requires several traversals of the whole set of open transitions, and it is not incorrect to leave some UNSAT OTs in the result. Empirically, on real life systems, the unreachable transitions constitute a small part of the ultimate open transitions, very little will be gained to compute the fixed point. So we would only perform one step of iteration. Further algorithms (e.g. model-checking or equivalence checking) will be able to use the full information from the predicates and the assignment sets, even if some spurious OTs are left here.

4.4 Computing Reachable State Space on the Fly

The basic algorithm introduced above potentially explores some areas of the global state space that will not be reachable after checking the OTs' predicates for satisfiability. Incorporating the satisfiability check inside the residual algorithm will definitely save some of these useless computation. More precisely, it will still create, and submit to Z3, the OTs at the "border" of the state space, but not the transitions issued from non-reachable global states. We use this idea to create our "smart" algorithm.

But there is more to this. Due to the symbolic nature of the transitions, reachability is not as easy to define as in standard models, as illustrated by the discussion above. By avoiding building the OTs coming from some unreachable states, we also avoid adding the corresponding assignments to the variables of their target states. But we *cannot* use the assignment information in the "on-the-fly" SAT checking, because they cannot be complete before exploring (at least) all the reachable state space. We first need to compute an over-approximation of the reachable state space and assignment sets; this corresponds to the automaton OA_2^0 in Algorithm 3, line 13. So the smart algorithm proceeds in two steps: during the first step, the SAT checking is done without using any assignment set information (line 5). Only the OTs found SAT from this check will be added to L, and the corresponding target global state added to U if necessary. This ensures that all explored global states are effectively reachable by a chain of SAT OTs.

Algorithm 3 "Smart" Reachable Open Automaton Generation

Input: A pNet P (cannot be a hole)
1: Initialize sets $U = \{InitState(P)\}$ and $E = \emptyset$, for unexplored and explored global states, respectively; $L = \emptyset$ for the resulting OTs;
2: **while** !$isEmpty(U)$ **do**
3: Choose S in U; remove S from U, add S to E;
4: $OTs = MakeTransitions(P, S)$;
5: **for** each $OT \in OTs$ **do** Check satisfiability of OT using the SMT solver;
6: **if** $SAT(OT)$ **then**
7: {Add OT to L;
8: Let S' be the target global state of OT
9: **if** $(S' \notin U \cup E)$ **then** Add S' into U; }
10: **end for**
11: **end while**
12: $L2$=$filterUnsatTransitions(L, Assignments)$;
13: OA_2^0=$makeReachableSubAutomaton(InitState(P), L2)$;
14: **repeat** OA_2^n=$RefineReachableSubAutomaton(OA_2^{n-1})$
15: **until** $OA_2^n = OA_2^{n-1}$;
16: **return** OA_2^n;

At the end of the residual loop, we do another traversal of the automaton, this time using the assignment information collected that far (line 12), and compute the final reachable sub-automaton iteratively (lines 13-15), in the same way as in Algorithm 1. Naturally, this step is only useful if there are some assignments.

Now we will prove that the smart algorithm computes essentially the same open automata as those of the original algorithm.

The easy case is when the system does not contain assignments to state variables. This does not mean "data-independency": we can still have guards in the PLTS transitions and in the synchronisation vectors, making the behaviour depending on the

values of input variables. But the open automaton needs not to collect the set of possible values dealing with the values of local variables, so the final automaton is AO_2^0 as computed in line 13.

Theorem 2 (Correctness without assignments) *For a pNet satisfying the finiteness conditions of Theorem 1 and containing no assignment to state variables, the smart algorithm, without a final reachability computation, computes the same open automaton as that of Algorithm 1 up to a renaming of variables in the transition labels.*

Proof Let pn be a pNet. The set of states managed by the two algorithms both are subsets of the cartesian product of the PLTS states, as defined in page 9. In a similar way, the set of state variables attached to each state are also equal. Comparing transitions requires a little more care, because within the $MakeTransitions$ function, fresh variables are created everytime a synchronisation vector is used (this occurs within the $MkPred$ and *matching* functions). However, given a global state, the call of $MakeTransitions$ will always return exactly the same set of OTs, up to a renaming of these fresh variables. When this is not ambiguous, we will speak of equality (of transitions, and of open automata) without the phrase "up to a renaming of fresh variables".

Let us first define some notations.

- $L0$: the original set of transitions by Algorithm 1 (i.e. before line 11),
- $L1_{sat}$: the original set of transitions by Algorithm 1, after satisfiability check (i.e. after line 11),
- $L1_{reach}$: the final set of OTs of Algorithm 1, as computed by line 12 (as OA_1^0).
- $L2_{sat}$: the original set of transitions by Algorithm 3 (i.e. before line 13),
- $L2_{reach}$: the final set of OTs of Algorithm 3, as computed by line 13. Here also $OA_2^n = OA_2^0$.
- $States : \overline{OT} \to \overline{State}$: the function that computes the set of target states of a set of OTs.
- $Rename : L1_{reach} \to L2_{reach}$: a function that keeps state names of transitions but renames fresh variables in the transition labels. In the sequel, transitions l_i will range over $L0$ set (and $L1_*$), and l'_i will range over $L2_*$ sets. Of course renaming preserves satisfiability.

We now prove that $L1_{reach} = L2_{reach}$ and $States(L1_{reach}) = States(L2_{reach})$.

By definition of the algorithms, we have:

$$L1_{reach} \subseteq L1_{sat} \subseteq L0 \qquad \text{and} \qquad L2_{reach} \subseteq L2_{sat} \ .$$

Let $S = \lhd\overline{s}\rhd = \lhd\{s_i\}^{i\in Leaves(pn)}\rhd$ range over the global states $States(L0)$, The only difference between the main loops of the algorithms is that Algorithm 3 only retains the satisfiable OTs (lines 6-7), so Algorithm 3 computes fewer transitions than Algorithm 1. Formally, we have that $L2_{sat} \subseteq Rename(L0)$ (up to a renaming of fresh variables), and as a consequence $States(L2_{reach}) \subseteq States(L0)$, i.e.,

$$\forall \lhd \overline{u}\rhd \in States(L2_{reach}), \exists \lhd \overline{t}\rhd \in States(L0), \lhd\overline{t}\rhd = \lhd\overline{u}\rhd \ .$$

For a global state S, and a set of OTs L, define its distance $|S|_L$ as the length n of (one of the) shortest chain $S_0 \xrightarrow{l_1} S_1 \xrightarrow{l_2} ...S_{n-1} \xrightarrow{l_n} S$ from the initial state S_0, with $l_k \in L$ for each $k \in [1..n]$.

We will prove by induction on the distance from the original state that $\forall S \in L0.\forall n \in \mathbb{N}.S \in States(L2_{reach}) \wedge |S|_{L2_{reach}} = n \Rightarrow S \in States(L1_{reach})$, and vice-versa.

The initial property is trivial, for $n = 0$, both state sets of distance zero are reduced to $\{S_0\}$.

Suppose our property is true for distance n, and suppose $|S|_{L2_{reach}} = n+1$, then there exists a chain $S_0 \xrightarrow{l'_1} S_1 \xrightarrow{l'_2} ...S_n \xrightarrow{l'_{n+1}} S$ with all $l'_k \in L2_{reach}$. In particular, since $L2_{reach} \subseteq L2_{sat}$, we have that $l'_{n+1} \in L2_{sat}$ and its predicate is satisfiable.

By induction hypothesis, we get $S_n \in States(L1_{reach})$, so during the call of *makeReachableSubAutomaton* in Algorithm 1, line 12, there exists an OT $S_n \xrightarrow{l_{n+1}} S$ in $L1_{sat}$, with $l'_{n+1} = Rename(l_{n+1})$; this OT will be checked and found SAT, so $S \in States(L1_{reach})$.

Conversely, if $|S|_{L1_{reach}} = n+1$, we get a similar chain in $L1_{reach}$, with the transition $S_n \xrightarrow{l_{n+1}} S$. Then $S_n \in States(L1_{reach})$ and l_{n+1} is satisfiable. By induction, we have $S_n \in States(L2_{reach}) = States(L2_{sat})$, so there exists $l'_{n+1} = Rename(l_{n+1})$ that will have been checked on the fly, in line 6 of Algorithm 3. Then the transition $S_n \xrightarrow{l'_{n+1}} S$ will be added to $L2_{sat}$, and thus $S \in States(L2_{reach})$.

Note that $L0$ is finite. Therefore, we can conclude that $L1_{reach} = L2_{reach}$ and $States(L1_{reach}) = States(L2_{reach})$. □

General case with assignments. In the presence of assignments, we need to collect all of them before using the assignment information to refine the SAT checking. Still, the smart strategy will explore a reduced state space, though not necessarily the minimal one. In the following theorem, both algorithms are considered in their complete version, meaning the reachability (with assignments) step is iterated up to fixed points.

Theorem 3 (Correctness with assignments) *In the general case, if a pNet satisfies the finiteness conditions of Theorem 1 and contains some assignments to state variables, the smart algorithm computes the same open automaton as that of Algorithm 1 up to a renaming of variables in the transition labels.*

Proof Define the OT sets $L0, L1_{sat}, L2_{sat}, L2_{reach}$ and the function *States* as before, and:

- $L1^*_1$: the final set of OTs of Algorithm 1, as computed by an iterative application of *makeReachableSubAutomaton* (line 14) up to a fixed point.
- $L2^*_2$: the final set of OTs of Algorithm 3, as computed by an iterative application of *makeReachableSubAutomaton* (line 15) up to a fixed point.
- $Assigns : \overline{OT} \rightarrow \overline{Assign}$: the function of computing the set of assignments of (the variables of the states of) a given set of OTs.
- $AssignsVar : Vars \times \overline{OT} \rightarrow \overline{Assign}$: the function of computing the set of assignments of a specific state variable in a transition.

The structure implicitly computed by both algorithms is a triple $Aut = \langle \overline{OT}, \overline{State}, \overline{Assign} \rangle$.

We already know that $L2_{sat} \subseteq L1_{sat}$ (up to a renaming), because every OT that will be checked positively (using assignments) in line 12 of Algorithm 3 will also be tested in Algorithm 1 (as $L2_{sat} \subseteq L0$), and with a larger set of assignments.

Consequently, the initial triples of the refinement loop are:

$$Aut^0_1 = \langle L1_{sat}, States(L1_{sat}), Assigns(L1_{sat}) \rangle$$
$$Aut^0_2 = \langle L2_{sat}, States(L2_{sat}), Assigns(L2_{sat}) \rangle$$
$$\text{with } Aut^0_2 \subseteq Aut^0_1, \text{ component-wise and up to a renaming.}$$

Based on each triple structure Aut, we define the function $Refine : Aut \to Aut$ modelling the function $makeReachableSubAutomaton(S0, OTs)$. It uses the assignments of the OTs remaining from the previous iteration to refine the computed reachable space:

$$Aut^n = Refine(Aut^{n-1}) = \langle L^n, States(L^n), Assigns(L^n)\rangle$$

where $L^n = Reach(L^{n-1} - L^{n-1}_{unsat})$, and L^{n-1}_{unsat} is the set of unsatisfiable transitions checked in this iteration. We have $L^n \subseteq L^{n-1}$, and consequently, their states and assignments may be reduced:

$$States(L^n) \subseteq States(L^{n-1}) \wedge Assigns(L^n) \subseteq Assigns(L^{n-1}) \ .$$

In each iteration k, consider a transition $l \in L^k$ and define

$$Pred(l, L^{k-1}) = Pred_l \wedge \bigwedge_{v \in vars(pn)} \{ \bigvee_{Assign \in AssignsVar(v, L^{k-1})} Assign\} \ .$$

Checking the satisfiability of l means checking the satifiability of $Pred(l, L^{k-1})$. Note that this is equivalent to the predicate defined in Section 4.2, because we consider all state variables of the pNet, rather than those of state s, but assignments of other variables will not change the satisfiability.

There are two reasons why an OT would disappear during one application of $Aut^k = Refine(Aut^{k-1})$: either its predicate, checked with the set of $Assign(L^{k-1})$, becomes UNSAT; or this OT is at the end of a sequence starting from a state in $s_0 \in State(L^k)$, the next OT becomes UNSAT, and the following states in the sequence are not reachable using satisfiable OTs in Aut^k. More formally, there is a chain

$$s_0 \xrightarrow{ot_1} s_1 \xrightarrow{ot_2} ...s_n \xrightarrow{ot_{n+1}} s_{n+1}$$

with $Pred(ot_1, L^{k-1})$ being UNSAT, and all states $s_i, i \in [1..n]$, are unreachable in Aut^k.

In each iteration, the number of triples decreases, thus there exists a sequence

$$Aut^n = Refine(Aut^{n-1}) \quad s.t. \ |Aut^n| < |Aut^{n-1}|.$$

The iteration terminates when Aut^n contains no more unsatisfiable transition, reaching the fixed point $Aut^n = Refine(Aut^{n-1})$. The initial structure is finite by Theorem 1, so the iteration always terminates.

Now we prove that the triples at the end of the fixed point computation in both algorithms are equal. Let Aut^*_1 and Aut^*_2 be two fixed points with $Refine(Aut^*_1) = Aut^*_1$ and $Refine(Aut^*_2) = Aut^*_2$. We want to prove $L^*_1 = L^*_2$.

We already have $Aut^0_2 \subseteq Aut^0_1$. We will proceed in two steps: first prove by induction that $Hyp^k = (L^k_2 \subseteq L^k_1)$ is preserved by each application of *Refine* (meaning that the refine function in Algorithm 1 cannot go "faster" than in Algorithm 3), then prove that when the fixed points are reached, they are the same.

Suppose Hyp^{k-1} holds and let us prove the validity of Hyp^k. Consider an OT $l \in L^k_2$, but $l \notin L^k_1$. We have

$$l \in L^k_2 \implies l \in L^{k-1}_2 \implies l \in L^{k-1}_1$$

so the transtion l has been removed from L^{k-1}_1 during the last iteration $Refine^k_1$. This can be caused by:

– either $Pred(l, L_1^{k-1})$ is UNSAT, because of some assignment $u \in Assigns(L_1^{k-1})$. But $l \in L_2^k$, so its predicate $Pred(l, L_2^{k-1})$ is satisfiable; this is a contradiction, because $Assigns(L_2^k) \subseteq Assigns(L_1^k)$ (having more possible assignments cannot reduce satisfiability).
– or there is a sequence $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} ...s_n \xrightarrow{l} s_{n+1}$, with $s_0 \in States(L_1^{k-1})$, $Pred(l_1, L_1^{k-1})$ is UNSAT, and all further states are unreachable in L_1^{k-1}. But as in the previous case, we would have l_1 in L_2^k and not in L_1^k, reaching a similar contradiction.

Now consider the fixed points of the $Refine$ iterations. From the initial inclusion $L_2^0 \subseteq L_1^0$ and our invariant, we get the inclusion $L_2^* \subseteq L_1^*$.

Suppose there exists an OT l in L_1^* that is not in L_2^*. The fact that $l \in L_1^*$ means that $Pred_l$, with the information contained in $Assigns(L_1^*)$, is satisfiable, but also all the transitions $\{l_i\}_{i\in[1..n]}$ in a sequence $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} ...s_n \xrightarrow{l_{n+1}} s_{n+1}$ leading to this $l_{n+1} = l$.

Clearly all these $\{l_i\}_{i\in[1..n]}$ are also satisfiable and reachable in L_2^0 during the "smart" traversal, which does not use assignments. Now consider the chain of assignment sets $\{Assigns(L_2^i)\}_{i\in[0..k]}$, used during the iteration of $Refine$ by Algorithm 3; we argue that the specific instantiation of variables that makes $Pred(l, L_1^*)$ satisfiable is contained in $Assigns(L_2^0)$, and that if it is removed during one of the $Refine$ step in Algorithm 1, then it is removed similarly in Algorithm 3. So we get a contradiction.

Therefore, we conclude that $L_2^* = L_1^*$. □

4.5 Translation to SMTlib and Satisfiability Check

We check the satisfiability of each open transition using the SMT solver Z3. In this section, we describe the translation of the algebra presentation, of assignments, and of the predicates.

Our implementation submits satisfiability requests to Z3 using its Java API. Here, for readability, we show the Z3 code using its SMTlib input language. Note that in the previous sections, the OTs were displayed in a simplified, human readable form. The input and output of our tool, and also the generated SMTlib fragments, are slightly more difficult to read, in particular because of structured names for the fresh variables generated by the algorithms, allowing for traceability of the result [42].

Figure 3 shows the translation of Example 6, page 12, in the SMTlib syntax. It contains the declaration of the relevant part of the LOTOS action algebra sorts and constructors, then the declaration of variables, and finally the predicate to be checked, encoded as a set of assertions.

Here the behaviour of the hole "P" (a_1 in Example 6), after matching with the synchronisation vectors, appears as a variable "|a1:sva_SVA0:15:1|", and "R" ($acc(x_2)$) as "(ACT delta |x:sva_SV4:1:2|)". Line 16 requires that $a_1 = \delta(x_2)$ to match the second synchronisation vector of the root pNet node, and lines 17-18 that a_1 does not match $\delta(x_1)$ for any x_1. We also display the diagnosis ("SAT" or "UNSAT") generated by Z3.

For simplicity, we have only shown here a use-case with boolean conditions, so we only require the basic SAT capability of the Z3 engine. It should be clear that more involved examples will require reasoning about predicates of various data types. Z3 provides theories for reasonning about integers, bitstrings, enumerated types, arrays, etc. We also need to deal with some universal quantification on free variables occuring

```
(declare-datatypes () (
(Action (ACT (|Action:arg:0| Action)(|Action:arg:1| Int))
  (Synchro (|Action:arg:2| Action))
  (l1 )(r1 )(d1 )(l2 )(r2 )(d2 )(delta )(acc ) )))
(declare-const |a1:sva_SV0:15:1| Action)
(declare-const |x:sva_SV0:15:1| Int)
(declare-const |x:sva_SV4:1:2| Int)
(declare-const |:ra:C2:14:1| Action)
(declare-const |:ra:C1:151:1| Action)
; checking OT:19
(assert(and
  (= d2 |:ra:C2:14:1|)
  (= |:ra:C2:14:1| d2)
  (= l1 |:ra:C1:151:1|)
  (= |:ra:C1:151:1| l1)
  (= |a1:sva_SV0:15:1| (ACT delta |x:sva_SV4:1:2|))
  (forall ((|\|x:sva_SV0:15:1\|| Int))
    (not (= |a1:sva_SV0:15:1| (ACT delta |\|x:sva_SV0:15:1\||))))))
(check-sat)
```

```
unsat
```

Fig. 3 Checking one open transition in Z3

in the guards of synchronisation vectors. For other data structures in such models, like records, or recursive structures, the user will have to develop her own theories. Decidability of these theories will condition on the completeness of the satisfiability check, and the decidability of model-checking or bisimulation checking.

Production of the SMT-lib code To build the input submitted to Z3 for each OT, we translate the algebra presentation, the predicates and the variable assignments into Z3 (Java-API) calls.

Translation of action algebra presentation. In [41], we define the translation of an algebra presentation into SMTlib declarations (`declare-datatypes` and `declare-fun`). We also formalise a condition of well-formedness of pNets to ensure that the generated code is correct and will not raise runtime errors in the SMT engine. Note that the `declare-datatypes` command comprises both the action constructors from Table 1 and also the constant action names from our example. In addition, we will include the axiomatisation of any required functions and predicates of the presentation data-types.

Translation of open transitions. In [42] we formally define all steps of the translation of each open transition, including:

- to collect all variables in the transition, and declare them (using `declare-const`),
- to check the well-formedness and correct typing of expressions,
- to translate the predicate into a conjunction of assertions,
- to translate the state-variable assignments into a disjunctive assertion, if the assignments are present in the source state.

This translation ensures that no runtime error will occur in the SMT engine.

Figure 3 shows the decomposition of the predicate into a set of asserts, each encoding an elementary equality, inequality, or a guard. The result (SAT or UNSAT) of the final `check-sat` command in the translation is then decoded.

5 Use-Cases

In this section we present the results of open automata construction for several examples, starting with three constructions from Section 2 using the Enable operator. Then we give detailed results for the Failure Controller that was presented in [41], and show how this sort of BIP architectures can be composed together. Finally, we give some figures illustrating the benefits of the smart algorithm on these examples.

5.1 Enable operator

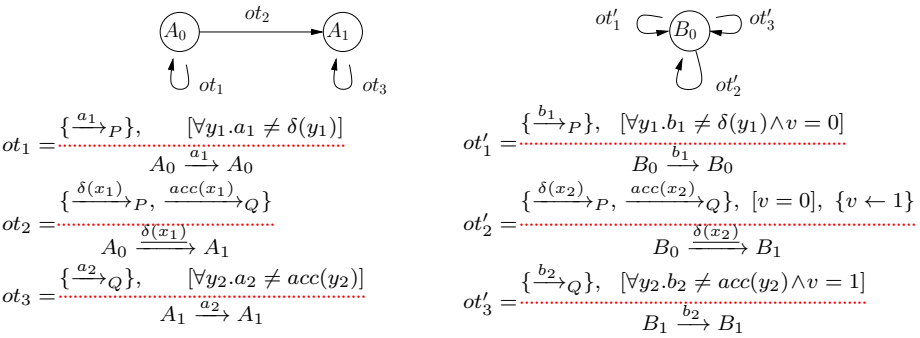

Fig. 4 The two open automata

The open automata generated for the two encodings of Enable in Figure 1 are displayed in Figure 4. The initial transition of *EnableData* is transformed to the initialisation of the state variables, which is $\{v \leftarrow 0\}$, for its open automata. Note that although these automata look very similar to the PLTS controllers from Figure 1, their nature is very different: they represent the behaviour of the whole system, including the holes P and Q. Their transitions do not rely on a particular implementation of the controllers, but only encode the relations between the actions of the holes. We have shown in [27] that the two open automata are equivalent with respect to a notion of (symbolic) FH-bisimulation.

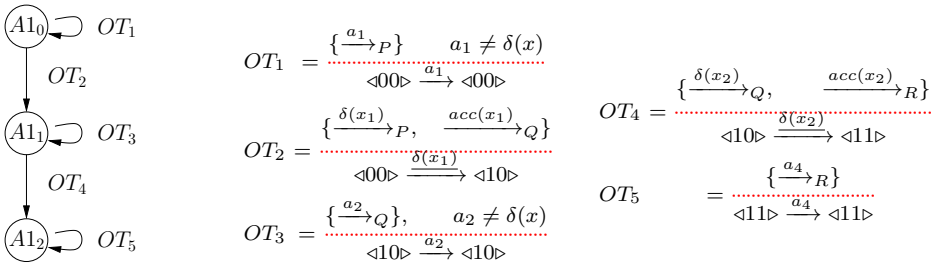

Fig. 5 Open automaton for the term "$P >> (Q >> R)$"

Now let us revisit the Enable composition from Figure 2. Its open automaton is shown in Figure 5. Building the symmetric system "$(P >> Q) >> R$" in a similar

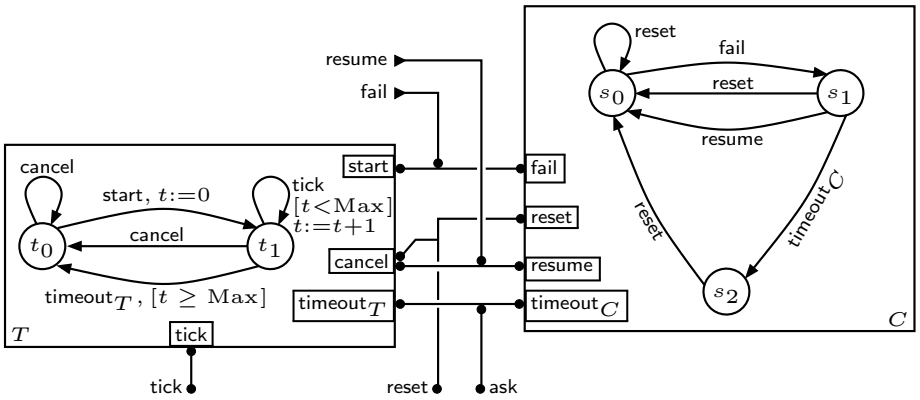

Fig. 6 The BIP specification of the Failure Monitor architecture

way will allow us to prove that they are bisimilar (the Enable operator is associative), as we have shown in [28]. This approach allows us to study the equational properties of many operators in classical process algebras or parallel languages, even when their semantics depends on data properties.

5.2 BIP Failure Monitor Architecture

In this section, we present a variation of the Failure Monitor architecture from the CubETH nanosatellite on-board software case-study [37] realised using BIP.

The architecture-based design process in BIP takes as input a set of components providing basic functionality of the system and a set of temporal properties that must be enforced in the final system. For each property, a corresponding architecture is identified and applied to the model, thereby potentially introducing additional coordinator components and modifying the connectors that define synchronisation patterns among ports of components.

Figure 6 shows this modified version of the Failure Monitor architecture used in [37].[2] The architecture consists of 2 *coordinator components* (C and T, for *Control* and *Timer*, respectively), 5 *dangling ports* (fail, resume, tick, reset and ask) and 5 connectors. The *behaviour* of BIP components is specified by finite automata extended with local data variables. Transitions of these automata are labelled with the ports of the corresponding components, Boolean guards and update functions on local variables. Although this is not visible in the figure, below we will assume that the dangling ports fail and resume belong to the operand component, whereas the dangling ports tick, ask and reset represent actions of the environment.

[2] The original CubETH case study [37] focused on the possibility of assembling a model of a complex software system in a systematic way by applying architectures to discharge individual system requirements. Here, we are more interested in the properties of the Failure Monitor architecture by itself. For that reason, we provide a modified version, in particular decomposing the coordinator into a Controller and a Timer, altering them to allow more flexibility in the acceptable behaviours. In other words, this modification enforces essentially the same properties, while discarding less acceptable behaviours.

Based on the attributes of the connected ports, which may be either *triggers* (triangles in Figure 6) or *synchrons* (bullets in Figure 6), connectors specify the synchronisations (also called *interactions*) between the transitions of the individual components. Intuitively, triggers can initiate interactions, whereas synchrons can only join. If a connector does not have triggers, all the involved ports must synchronise [9]. For instance, the two ports $T.\mathsf{start}$ and $C.\mathsf{fail}$ are always synchronised, since they belong to the same binary sub-connector, where they are both synchrons. In particular, this means that whenever the transition $s_0 \xrightarrow{\mathsf{fail}} s_1$ is fired, so is the transition $t_0 \xrightarrow{\mathsf{start}, t:=0} t_1$, initialising the timer. The binary connector $T.\mathsf{start} \bullet\!\!-\!\!\bullet C.\mathsf{fail}$ is a sub-connector of a hierarchical connector, where the dangling port $B.\mathsf{fail}$ is a trigger. Thus, the above interaction can only happen together with $B.\mathsf{fail}$, forming a ternary interaction. On the contrary, being a trigger, the port $B.\mathsf{fail}$ can fire alone, forming a singleton interaction. The composition semantics of BIP systems consists in firing exactly one interaction, enabled through at least one of the top-level connectors, at each execution round.

Notice that the model in Figure 6 allows reset to happen in any state. However, it is implied that, when the coordinator C is in state s_2, this reset is "asked for" by the architecture (it follows the firing of ask), whereas in other states the corresponding transition reflects an "external" reset. Although this can be formalised in the model using data variables, we have chosen not to do so for the sake of simplicity.

Intuitively, application of the Failure Monitor architecture ensures that, whenever a failure is registered in the operand component, the system will be reset, unless a resumption is registered within Max time units. Formally, this statement is decomposed into several properties, among which 1) the safety property *"A reset is never asked for unless a failure occurs"* specified using CTL as

$$\mathtt{A}\,[\neg \mathsf{ask}\ \mathtt{W}\ \mathsf{fail}] \tag{1}$$

and 2) the liveness property *"in the absence of an external reset, a reset can always be asked for after a non-transient failure"*:

$$\mathtt{AG}\,\big(\mathsf{fail} \rightarrow \mathtt{E}\,[(\neg \mathsf{reset} \wedge \neg \mathsf{resume})\ \mathtt{U}\ \mathsf{ask}]\big) \tag{2}$$

Additional properties are provided in [42]. Once we have generated the open automaton of the architecture, such properties can be verified by model checking tools, which is beyond the scope of the current work.

Figure 7 shows a pNet encoding of the above Failure Monitor architecture. This encoding is structural: each coordinator component is encoded as a PLTS; dangling ports belonging to the operand component are encoded as actions of the hole; connectors of the BIP model are encoded as synchronisation vectors, with ports representing the actions of the environment used as global actions. Each connector that does not involve triggers is trivially encoded by a synchronisation vector comprising the same ports ($SV2$, $SV3$ and $SV4$). In order to encode the semantics of the connectors involving triggers, we use (in $SV0$ and $SV1$) the classical approach where non-participation of a port in an interaction is simulated by an additional loop transition [38]. To this end, we consider an action algebra involving additional Boolean variables: the effective transitions carry the value *true* (e.g. $s_0 \xrightarrow{\mathsf{fail}(true)} s_1$), whereas the added loops carry the value *false* (e.g. $s_2 \xrightarrow{\mathsf{fail}(false)} s_2$). In the synchronisation vectors, we use these variables to encode the connector structure as a Boolean predicate. For example, $SV0$ encodes the connector discussed above: the predicate $(b1 = b2) \wedge (b1 \vee b2 \Rightarrow b0)$

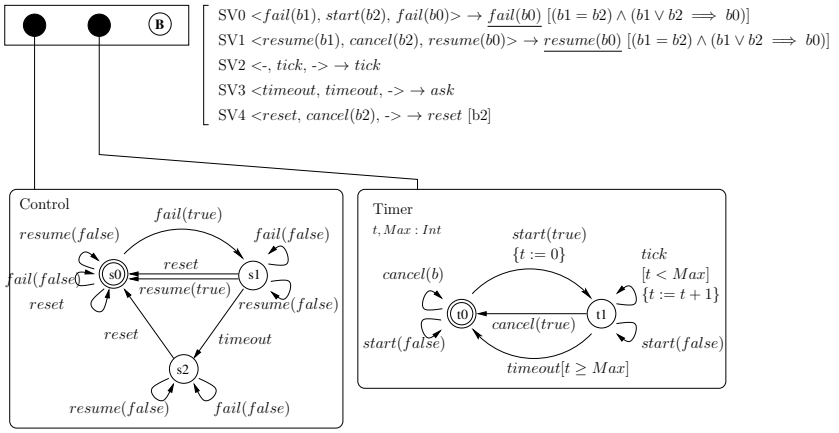

Fig. 7 pNet encoding of the Failure Monitor architecture

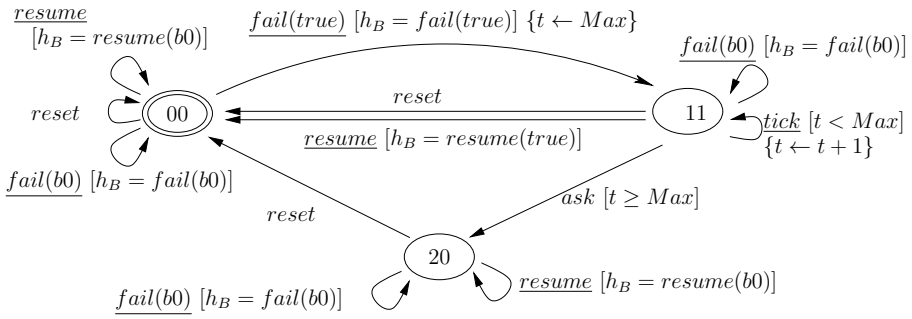

Fig. 8 Open automaton for the Failure Monitor architecture

means that the "true" transitions $C.\mathsf{fail}$ and $T.\mathsf{start}$ can only fire together ($b1 = b2$) and whenever one of them fires, $B.\mathsf{fail}$ must fire also ($b1 \vee b2 \Rightarrow b0$). This encoding can be systematically obtained for any hierarchical BIP connector [10].

Computed open automaton. For this example, the tool (using Algorithm 1) builds 408 open transitions, whereof 396 are detected unsatisfiable by Z3. The resulting open automaton, with 3 reachable global states (out of the possible 6) and 12 open transitions, is shown in Figure 8.

The satisfaction of the safety property (1) could be established by applying symbolic model checking techniques. However, in this example, it is obvious by inspection of the open automaton. The satisfaction of the liveness property relies on the observation that the only loop starting in the state $\triangleleft s1, t1 \triangleright$ and involving neither *reset*, nor *resume* or *ask*, is the self-loop $fail(b0)$. By further inspecting the synchronisation vector $SV0$, we observe that this self-loop corresponds to the transitions $fail(false)$ and $start(false)$ in the two PLTSs. Recall that these transitions (carrying the value *false*) are introduced by the encoding. That is, none of the two coordinating components are involved in the joint transition. Therefore, under resonable scheduling assumptions, the transition *ask* will eventually be fired.

5.3 Composing Two Failure Monitor BIP Architectures

In this section, we use a combination of two Failure Monitor architectures shown in Figure 9 as a slightly larger example to show how the approach scales up. In this example, each of the two instances of the architecture is applied to a corresponding operand component. Hence, the dangling ports fail and reset are duplicated accordingly. On the contrary, the environment actions tick, ask and reset are shared. For a detailed presentation of the architecture composition, we refer the reader to [3].

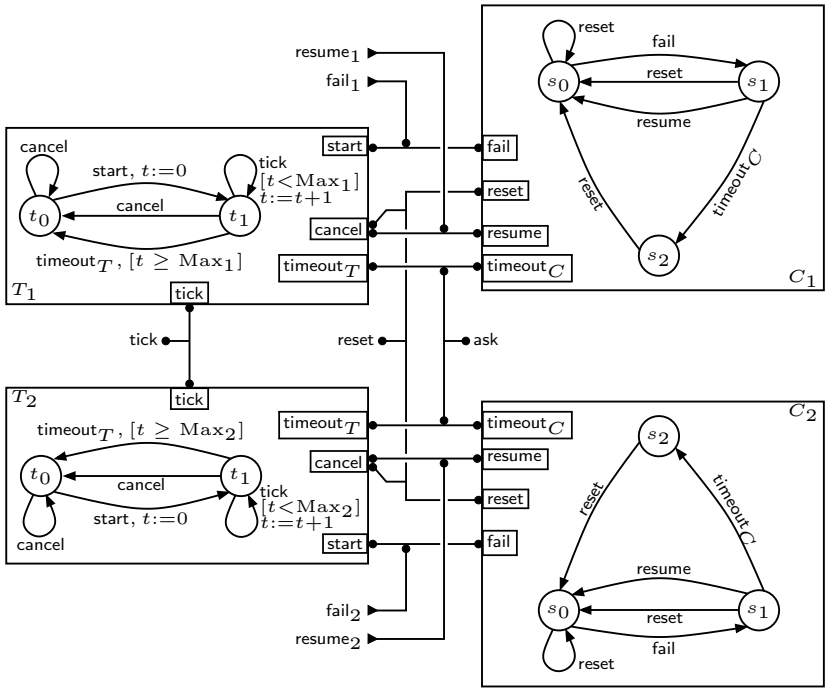

Fig. 9 The BIP specification of two composed Failure Monitor architectures

In Figure 10 we show the open automaton computed for the combined architecture of Figure 9. Comparing the guards of the transitions tick and ask leaving state 1111, we see that ask can never be taken, unless Max1 = Max2. This illustrates the *interference* phenomenon discussed in [3], showing that the liveness property (2) is not preserved when two copies of the Failure Monitor architecture are composed together.[3] In this example, the interference is caused by the strong synchronisation of the tick ports of the two architectures. It can be avoided by replacing the tick $\bullet\!\!-\!\!\bullet T$.tick connector in Figure 6 by tick $\blacktriangleright\!\!-\!\!\bullet T$.tick and applying the so-called *maximal progress* whereby, whenever possible, the interaction $\{\mathsf{tick}, T.\mathsf{tick}\}$ is preferred to the firing of tick alone. However, this fix goes beyond the scope of the current paper.

[3] One of the main results of [3] states that in a system obtained by an application of several architectures, liveness properties are preserved if these architectures are *pair-wise non-interfering*. However, no results were provided to check whether two architectures are interfering or not.

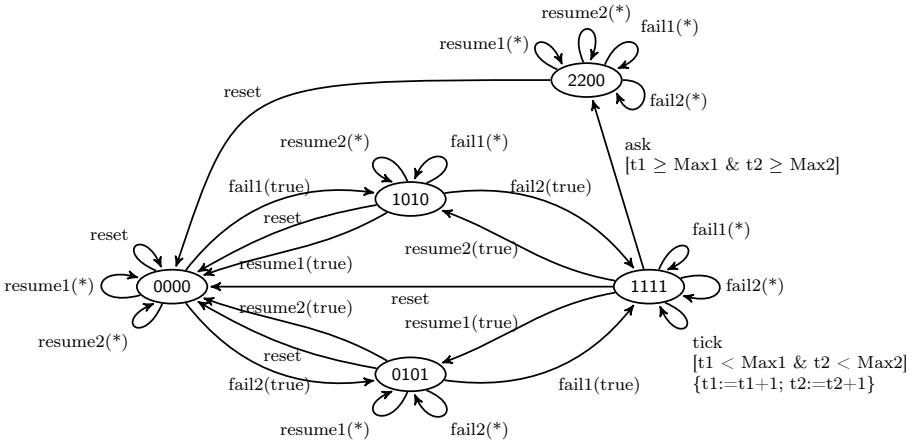

Fig. 10 The resulting automaton

5.4 Benefits from the Smart Algorithm

In each cell of the following table, we display the number of OTs whose predicates were found SAT/UNSAT by the SMT engine, and the overall time spent for the execution of the whole algorithm.

All tests are run on HP EliteBook 840 laptop under Windows 10 x64, with an Intel quad-Core i7-6600U processor, and 32Gb of RAM. The software platform is VerCors[4], running on top of Eclipse Equinox, with Java 1.8.

Use-case	Brute force	Smart algorithm
Enable operator	3/6 OTs	3/6 OTs
State-based	125 ms	110 ms
Enable operator	3/6 OTs	3/6 OTs
Data-based	141 ms	205 ms
Enable Composition	5/55 OTs	5/42 OTs
State-based	359 ms	297 ms
FailureTimer	12/396 OTs	12/192 OTs
	1220 ms	687 ms
FailureTimer X 2	25/8964 OTs	25/1255 OTs
	18088 ms	3266 ms

In the basic cases (*EnableState*, *EnableData*) there is no benefit, and even an overhead in the case of *EnableData*, because of the extra step of checking SAT at the end of the algorithm.

But as soon as there are more OTs in the system and not all states are reachable, these tests show a significant benefit of the "smart" strategy, in the considered use-cases, with 50% of time saving for the *FailureTimer*, over to 80% for the compo-

[4] Available at `https://team.inria.fr/scale/software/vercors/`.

sition, depending on the nature of the examples, but increasing significantly with their complexity.

6 Conclusion and Discussion

The formal definitions and properties of the open pNets model were given in [27]. In the current work we describe an implementation of the model and its semantics construction, including its interaction with the Z3 SMT engine. The implementation has two parts: the first is an algorithm that builds all possible combinations of synchronisations through the pNet hierarchical structure. The result is a so-called open automaton, whose transitions contain predicates relating the actions of the pNet holes and controllers. Some of the open transitions obtained at this step may contain predicates which do not represent any possible concrete instantiations. In the second part of the tool we use the SMT solver Z3 to check the satisfiability of the predicate in each open transition. To this end, we encode into Z3 the representations of the action algebra and the predicates before submitting them to the Z3 solver. We have used a running example based on two possible encodings of the Enable operator of LOTOS. The automata computed with the algorithm can be used to prove equational properties of the operator.

Then we define a "smart" version of the algorithm that explores, as much as possible, only the reachable part of the open automaton. The handling of assignments makes the reachable computation more challenging. We show that the result is equivalent to the original algorithm.

We validate the approach on a larger example, based on a BIP architecture taken from an earlier nanosatellite case study [37]. This example shows that open-automata-based semantics can be instrumental in verifying the properties enforced by the architectures through an encoding into open pNets. In order to evaluate the potential of our approach to scale to larger systems, we also apply the proposed techniques to a BIP architecture obtained by combining two instances of the Failure Monitor architecture. Our results show that, even though, the performance of the smart algorithm degrades more than linearly, execution times remain reasonably small. It should be noted that the aim of our approach, particularly in the case of BIP architectures, is to allow symbolic verification of design patterns to show that they do impose corresponding properties. Safety of the system is obtained compositionally, whereas liveness is verified by checking *pair-wise* non-interference. In both cases, the main characteristic of the approach is compsitionality, so we can compute the behavior and analyse the properties on *small open systems*, and use these properties independently when composing bigger systems.

The results presented in this paper open a number of avenues for future work. Naturally, our next goals after the generation of the open automata will be to model-check logical properties, and to check equivalences of pNets. While model-checking open automata seems easy to define, equivalence checking is more challenging. In [27], we have already found the FH-bisimulation to be a suitable definition. But weak equivalences, or refinements, will definitely be useful when comparing different pNets with different structures. For both model-checking and bisimulation, SMT methods will be the basis for comparing open transitions.

BIP architectures in this paper and in the CubETH case study [37] involved data in the BIP components behaviour but not in the interactions among components. We

have shown in [8] how to overcome this limit, allowing data transfer in BIP architecture connectors and providing an encoding of such architectures in terms of open pNets. To push the study of BIP architecture verification using open pNets even further, one can observe that there is a certain "regularity" in the structure of the composed architecture in Figure 9. Indeed, both architectures in Figures 6 and 9 are instances of the same *architecture style* [36] for 1 and 2 monitored processes, respectively. The question then naturally arises of whether some version of open pNets (e.g. extended to accommodate a parameterised number of sub-pNets) can be used to verify certain architectural styles without instantiating them with fixed parameter values. Although this problem is known to be undecidable in the general case (see, e.g. [31]), it would be interesting to study the interplay of existing techniques, such as well structured transition systems [21], with open pNet extensions.

References

1. Alberti, F., Ghilardi, S., Pagani, E., Ranise, S., Rossi, G.P.: Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. JSAT **8**(1/2), 29–61 (2012). URL `https://satassociation.org/jsat/index.php/jsat/article/view/93`
2. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of sat/smt solvers to coq through proof witnesses. In: International Conference on Certified Programs and Proofs, pp. 135–150. Springer (2011)
3. Attie, P., Baranov, E., Bliudze, S., Jaber, M., Sifakis, J.: A general framework for architecture composability. Formal Aspects of Computing **18**(2), 207–231 (2016)
4. Baranov, E., Bliudze, S.: Offer semantics: Achieving compositionality, flattening and full expressiveness for the glue operators in BIP. Science of Computer Programming **109**(0), 2–35 (2015). DOI 10.1016/j.scico.2015.05.011
5. Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: Computer aided verification. Springer (2011)
6. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017). Available at `www.SMT-LIB.org`
7. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software **28**(3), 41–48 (2011). DOI 10.1109/MS.2011.27
8. Bliudze, S., Henrio, L., Madelaine, E.: Verification of concurrent design patterns with data. In: H. Riis Nielson, E. Tuosto (eds.) Coordination Models and Languages, pp. 161–181. Springer International Publishing, Cham (2019)
9. Bliudze, S., Sifakis, J.: The algebra of connectors—Structuring interaction in BIP. IEEE Transactions on Computers **57**(10), 1315–1330 (2008). DOI 10.1109/TC.2008.26
10. Bliudze, S., Sifakis, J.: Causal semantics for the algebra of connectors. Formal Methods in System Design **36**(2), 167–194 (2010). DOI 10.1007/s10703-010-0091-z
11. Bruni, R., de Frutos-Escrig, D., Martí-Oliet, N., Montanari, U.: Bisimilarity congruences for open terms and term graphs via tile logic. In: C. Palamidessi (ed.) CONCUR 2000, pp. 259–274. Springer, Berlin (2000)
12. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Verification of data-aware processes via array-based systems (extended version). CoRR **abs/1806.11459** (2018). URL `http://arxiv.org/abs/1806.11459`
13. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: A. Biere, R. Bloem (eds.) CAV, pp. 334–342. Springer, Cham (2014)
14. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The kind 2 model checker. In: S. Chaudhuri, A. Farzan (eds.) Computer Aided Verification, pp. 510–517. Springer International Publishing, Cham (2016)
15. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. CoRR **abs/1310.6847** (2013). URL `http://arxiv.org/abs/1310.6847`
16. De Simone, R.: Higher-level synchronising devices in MEIJE-SCCS. Theoretical Computer Science **37**, 245–267 (1985)

17. Déharbe, D.: Integration of smt-solvers in b and event-b development environments. Science of Computer Programming **78**(3), 310–326 (2013)
18. Déharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: Integrating smt solvers in rodin. Science of Computer Programming **94**, 130–143 (2014)
19. Deng, Y., Fu, Y.: Algorithm for verifying strong open bisimulation in full π calculus. Journal of Shanghai Jiaotong University **E-5**(2), 147–152 (2001)
20. Feng, Y., Deng, Y., Ying, M.: Symbolic bisimulation for quantum processes. ACM Transactions on Computational Logic **15**(2), 1–32 (2014)
21. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science **256**(1), 63 – 92 (2001). DOI 10.1016/S0304-3975(00)00102-X
22. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT model checking of array-based systems. In: Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, 2008, pp. 67–82 (2008). DOI 10.1007/978-3-540-71070-7_6. URL `https://doi.org/10.1007/978-3-540-71070-7_6`
23. Hennessy, M., Lin, H.: Symbolic bisimulations. Theoretical Computer Science **138**(2), 353–389 (1995). DOI 10.1016/0304-3975(94)00172-F. URL `http://dx.doi.org/10.1016/0304-3975(94)00172-F`
24. Hennessy, M., Rathke, J.: Bisimulations for a calculus of broadcasting systems. Theoretical Computer Science **200**(1-2), 225–260 (1998). DOI 10.1016/S0304-3975(97)00261-2. URL `http://dx.doi.org/10.1016/S0304-3975(97)00261-2`
25. Henrio, L., Kulankhina, O., Liu, D., Madelaine, E.: Verifying the correct composition of distributed components: Formalisation and Tool. In: FOCLASA, no. 175 in EPTCS. Rome, Italy (2014). URL `https://hal.inria.fr/hal-01055370`
26. 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). IEEE (2015)
27. Henrio, L., Madelaine, E., Zhang, M.: A Theory for the Composition of Concurrent Processes. In: Formal Techniques for Distributed Objects, Components, and Systems (FORTE), vol. LNCS-9688. Heraklion, Greece (2016). URL `https://hal.inria.fr/hal-01432917`
28. Henrio, L., Madelaine, E., Zhang, M.: A theory for the composition of concurrent processes – extended version. Rapport de recherche RR-8898, INRIA (2016)
29. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
30. ISO: Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organisation for Standardization, Geneva, Switzerland (1989). URL `citeseer.ist.psu.edu/338220.html`
31. Konnov, I.V., Kotek, T., Wang, Q., Veith, H., Bliudze, S., Sifakis, J.: Parameterized systems in BIP: design and model checking. In: 27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada, *LIPIcs*, vol. 59, pp. 30:1–30:16 (2016). DOI 10.4230/LIPIcs.CONCUR.2016.30
32. Larsen, K.G.: A context dependent equivalence between processes. Theoretical Computer Science **49**, 184–215 (1987)
33. Li, Z.: Theories and algorithms for the verification of bisimulation equivalences in value-passing CCS and π-calculus. Ph.D. thesis, Changsha Institute of Technology (1999)
34. Lin, H.: Symbolic transition graph with assignment. In: U. Montanari, V. Sassone (eds.) Concur'96, *LNCS*, vol. 1119, pp. 50–65. Springer, Heidelberg (1996)
35. Lin, H.: Model checking value-passing processes. In: 8th Asia-Pacific Software Engineering Conference (APSEC'2001). Macau (2001)
36. Mavridou, A., Baranov, E., Bliudze, S., Sifakis, J.: Architecture diagrams: A graphical language for architecture style specification. In: Proceedings 9th Interaction and Concurrency Experience (ICE), *EPTCS*, vol. 223, pp. 83–97 (2016). DOI 10.4204/EPTCS.223.6
37. Mavridou, A., Stachtiari, E., Bliudze, S., Ivanov, A., Katsaros, P., Sifakis, J.: Architecture-based design: A satellite on-board software case study. In: 13th Int. Conf. on Formal Aspects of Component Software (FACS 2016) (2016)
38. Milner, R.: Calculi for synchrony and asynchrony. TCS **25**(3), 267–310 (1983). DOI 10.1016/0304-3975(83)90114-7
39. Milner, R.: Communication and Concurrency. Int. Series in Computer Science. Prentice-Hall, Englewood Cliffs, New Jersey (1989). SU Fisher Research 511/24
40. Milner, R.: Communicating and mobile systems - the Pi-calculus. Cambridge University Press (1999)

41. Qin, X., Bliudze, S., Madelaine, E., Zhang, M.: Using SMT engine to generate symbolic automata. In: 18th International Workshop on Automated Verification of Critical Systems (AVOCS 2018). Electronic Communications of the EASST (2018)
42. Qin, X., Bliudze, S., Madelaine, E., Zhang, M.: Using SMT engine to generate Symbolic Automata – Extended version. Rapport de recherche RR-9177, INRIA (2018)

A Semantic Rules of pNets

To facilitate the understanding of the two algorithms, we briefly recall the semantic rules of pNets, more details can be found in [27].

We build the semantics of an open pNet as an open automaton where LTSs are the PLTSs at the pNet leaves, and the states of the automaton are structured. To build an open transition we first projects the global state into states of the leaves, then apply 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 synchronisation vector are considered local: for a given pNet expression, we must have fresh local variables for each occurrence of a vector. 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.

The reader may notice that the structure of OTs produced by these rules is richer than the one from the definitions in Section 4, as they contain information about the leaves transitions in each OT. This is also the case in the implementation, and provides us with tracability and debugging features in the tool.

Definition 6 (Operational semantics of open pNets) The semantics of a pNet p is an open automaton $A = \langle Leaves(p), J, \mathcal{S}, s_0, \mathcal{T} \rangle$ where:

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

The first rule (**Tr1**) for a PLTS p checks that the guard is verified and transforms assignments into post-conditions:

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

The second rule (**Tr2**) 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. For example, in the rule, L is the index set of the leaves of the open pNet, L_k is the index set of the leaves of one subnet, thus all L_k are disjoint subsets of L.

Tr2:

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

In rule TR2, the generated predicate is composed of the conjunction of the predicates of the subnets' OTs, with the additional part encoding the application of the chosen synchronisation

vector. In [27] this last part is defined as:

$$MkPred(SV_k, a_i^{i\in I}, b_j^{j\in J}, v) \Leftrightarrow SV_k = (a_i)^{i\in I}, (b_j)^{j\in J} \rightarrow v$$

Within Algorithm 1, these subsets have been computed by the *Combining* method and passed as arguments to the *Matching* method (cf. Section 4)

To have some practical interest, it is important to know when Algorithm 1 terminates. The following theorem shows that if an open pNet has finite synchronisation sets, finitely many leaves and holes, and each PLTS at its leaves has a finite number of states and (symbolic) transitions, then Algorithm 1 terminates and the operational semantics of the open pNet is a finite automaton.

Theorem 4 (Finiteness) *[27] Let pnet* $= \langle \overline{pNet}, \overline{S}, SV_k^{k\in K} \rangle$ *be an open pNet with leaves* $l_i^{i\in I}$ *and holes* $h_j^{j\in J}$*, where the sets* I *and* J *are finite. Suppose the synchronisation vectors of all pNets included in pnet are finite, and* l_i *has a finite number of state variables for each* $i \in I$*. Then the semantics of pnet is an open automaton with finitely many states and transitions.*

Proof The possible set of states of the open automaton is the cartesian product of the states of the pNet $PLTS_i^{i\in I}$, which is finite by hypothesis. So the top-level residual loop of Algorithm 1 terminates provided each iteration terminates. The enumeration of open-transitions in line 5 of Algorithm 1 is bounded by the number of applications of rule Tr2 on the structure of the pNet tree. Since a finite number of synchronisation vectors are applied at each node, the number of global open transitions is finite. Similarily, if the number of transitions of each PLTS is finite, rule Tr1 is applied a finite number of times. Therefore, each internal loop of Algorithm 1 terminates, and we obtain finitely many deduction trees and open transitions. □