# Symbolic Bisimulation for Open and Parameterized Systems

Zechen Hou, Eric Madelaine

# Symbolic Bisimulation for Open and Parameterized Systems

Zechen Hou
East China Normal University
China
51174500089@stu.ecnu.edu.cn

Eric Madelaine
Inria Sophia Antipolis Mediterannee
France
eric.madelaine@inria.fr

## Abstract

*Open Automata* (OA) are symbolic and parameterized models for open concurrent systems. Here *open* means partially specified systems, that can be instantiated or assembled to build bigger systems. An important property for such systems is "compositionality", meaning that logical properties, and equivalences, can be checked locally, and will be preserved by composition. In previous work, a notion of equivalence named *FH-Bisimulation* was defined for open automata, and proved to be a congruence for their composition. But this equivalence was defined for a variant of open automata that are intrinsically infinite, making it unsuitable for algorithmic treatment.

We define a new form of equivalence named *StrFH-Bisimulation*, working on finite encodings of OAs. We prove that StrFH-Bisimulation is consistent and complete with respect to the FH-Bisimulation.

Then we propose two algorithms to check *StrFH-Bisimulation*: the first one requires a (user-defined) relation between the states of two finite OAs, and checks whether it is a StrFH-Bisimulation. The second one takes two finite OAs as input, and builds a "weakest StrFH-bisimulation" such that their initial states are bisimilar. We prove that this algorithm terminates when the data domains are finite. Both algorithms use an SMT-solver as a basis to solve the proof obligations.

***CCS Concepts*** • **Theory of computation → Logic and verification**; *Process calculi*; • **Computing methodologies → Theorem proving algorithms**; **Model verification and validation**; • **Hardware** → *Equivalence checking*.

***Keywords*** Strong Bisimulation, Open Systems, SMT Solver


*PEPM '20, January 20, 2020, New Orleans, LA, USA*

ACM ISBN 978-1-4503-7096-7/20/01…$15.00
https://doi.org/10.1145/3372884.3373161

**ACM Reference Format:**
Zechen Hou and Eric Madelaine. 2020. Symbolic Bisimulation for Open and Parameterized Systems. In *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '20), January 20, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3372884.3373161

## 1 Introduction

How to reduce the state space of system models, being a challenging area of formal verification, attracts the interest of many researchers approaching this problem from different angles.

Some approaches focus on the structure of the system models. This is the case with the proposal of open systems, which provide a way to represent incomplete systems, like parallel skeletons, or process algebra operators. The idea is to define (small) open systems for which you can prove crucial properties, then assemble these systems to build full applications, while preserving the properties.

Some approaches are dealing with semantics of the system models in a symbolic way. This provides a way to use abstract (parameterized) terms to describe the states or transitions, manipulating explicitly potentially unbounded data domains, and thus decreasing drastically the model size.

Bisimulation is also an important approach, in which attention on the equivalence between different systems, allows for hierarchical model minimization, enhancing the practical capabilities for further analysis, typically model-checking.

The approach of [11] offers a methodology using open, symbolic, and parameterized models, endowed with a notion of symbolic bisimulation. It defines a new behavioural specification formalism called "Open parameterized Networks of Synchronized Automata (pNet)" for distributed, synchronous, asynchronous or heterogeneous systems. The pNet model has a hierarchical and tree-like structure that gives it a strong ability for describing and composing complex systems. The symbolic and open aspects of pNets give them the potential to use small state space to represent large systems. Figures 1 show examples of pNets inspired from [11], and that we will use as running examples in this paper. We won't go here into the details of the pNet model, because each pNet will generate a corresponding open automaton as it's operational semantics, and all the further analysis and verification are defined on this open automaton.

Current work define open automata with some *semantic* flavor, in which each automaton has an interesting "Closure" property under substitution. These *Semantic Open Automata* are endowed with a notion of symbolic bisimulation called *FH-Bisimulation*, and it is proved in [12] that this equivalence is a congruence for pNet composition, that allows for compositional verification methods. But this definition cannot be used as such in algorithms and tools, because all the semantic open automaton are infinite, according to their closure property.

Another important inspiration for our work was from the seminal research in [14] on Symbolic Transition Graphs with Assignements (STGA). STGA share with open automata the ability to manipulate explicitly symbolic expressions in transitions, guards and assignments. They are endowed with a notion of symbolic bisimulation, and [15] defined an "On-the-fly" algorithm for computing this bisimulation, although this was limited to *data-independent* systems. Also STGAs were addressing only closed systems, so open automata are significantly different.

Our main goal in this work is to define an alternative bisimulation relation, suitable for finitely represented open automata, as can be generated from the semantic rules of open pNets, and to define algorithms checking or generating this relation.

Our main contributions in this paper are the following:

**Contribution 1:** We propose a new *structural* Bisimulation equivalence called StrFH-Bisimulation between open automata that are not necessarily closed under substitution. This allows us to address the finite systems that can be computed from pNets. We define a correspondence between such "finite" and "semantic" open automata, and prove that StrFH-Bisimulation is correctly and completely corresponding to FH-Bismulation. Thus, we generalize the interesting properties of FH-Bisimulation to StrFH-Bisimulation. Details of this Bisimulation equivalence and its properties are in section 3.

**Contribution 2:** The StrFH-Bisimulation we mentioned above, is defined between a kind of open and symbolic systems with parameterized action, value-passing and assignments, all these features will bring great challenges. Our second contribution is proposing an SMT-solver based algorithm that can check if a relation is a StrFH-Bisimulation between input automata. The details of this approach are in Section 4.

**Contribution 3:** We go further than above approach, and devise a new *on-the-fly* algorithm that generates a weakest StrFH-Bisimulation between two given open automata. We prove that the result of the algorithm is correct and indeed the weakest. The algorithm terminates whenever both the (symbolic) open-automata and the data-domains are finite. Details of this algorithm are in Section 5. Note that this is definitely better than the "data-independence" constraint of the STGA algorithm. If used in association with some abstract interpretation of the data domains, we can address a large set of infinite systems.

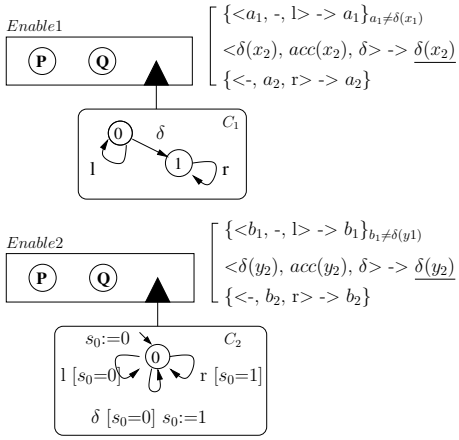



**Figure 1.** Two pNet encodings for Enable

Detailed proofs of the theorems are available in the extended version of this paper [13].

## 2 Background

### 2.1 Notations

#### 2.1.1 Term algebra

Our models rely on a notion of parameterised actions, that are symbolic expressions using data types and variables. As our model aims at encoding the low-level behaviour of possibly very different programming languages, we do not want to impose one specific algebra for denoting actions, nor any specific communication mechanism. So we leave unspecified the constructors of the algebra that will allow building expressions and actions. Moreover, we use a generic *action interaction* mechanism, based on (some sort of) unification between two or more action expressions, to express various kinds of communication or synchronisation mechanisms.

Formally, we assume the existence of a term algebra $\mathbb{T}$, where $\Sigma$ is the signature of the data and action constructors. Within $\mathbb{T}$, we distinguish a set of expressions $\mathbb{E}$, including a set of boolean expressions $\mathbb{E}$ ($\mathbb{B} \subseteq \mathbb{E}$). We let $e_i$ range over expressions ($e_i \in \mathbb{E}$). On top of $\mathbb{E}$ we build the action algebra $\mathbb{A}$, with $\mathbb{A} \subseteq \mathbb{T}, \mathbb{E} \cap \mathbb{A} = \emptyset$; naturally action terms will use data expressions as subterms.

Let $a$ range over action labels, $op$ be operators, and $x_i$ range over variable names.

The set of actions is defined as:

| | | | |
|---|---|---|---|
| $\alpha \in \mathbb{A}$ | ::= | $a(p_1, \ldots, p_n)$ | action terms |
| $p_i$ | ::= | $?x \mid e_i$ | parameters (input variable or expression) |
| $e_i$ | ::= | $Value \mid x \mid op(e_1, .., e_n)$ | Expressions |

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

Bound variables are the variables quantified by a quantifier $\forall$ or $\exists$, and other variables are free variables. The function $vars(t)$ identifies the set of free variables in a term $t \in \mathbb{T}$, and $iv(t)$ returns the name of its input variables (without the '?' marker). Action algebras can encode naturally usual point-to-point message passing calculi (using $a(?x_1, ..., ?x_n)$ for inputs, $a(v_1, .., v_n)$ for outputs), but it also allows for more general synchronisation mechanisms, like gate negotiation in Lotos, or broadcast communications.

#### 2.1.2 Substitutions

We denote $(x_k \leftarrow e_k)^{k \in K}$ a substitution, also ranged by $\sigma$, where $(x_k)^{k \in K}$ is a set of variables and $(e_k)^{k \in K}$ is a set of expressions. The application of the substitution to an expression is denoted $e'\{\!\{(x_k \leftarrow e_k)^{k \in K}\}\!\}$, the operation replaces in parallel all free occurrences of the variables $x_k^{k \in K}$ by the expression $e_k^{k \in K}$ in $e'$. We define $dom(\sigma)$ as the domain of substitution $\sigma$, ranged by variables, and $codom(\sigma)$ the set of variables in the right hand side expressions in $\sigma$. For example, if $\sigma = (x \leftarrow e_1, y \leftarrow e_2)$ then $dom(\sigma)$ denotes the variable set $\{x, y\}$, and $codom(\sigma)$ is $vars(e_1) \cup vars(e_2)$.

We write $\uplus$ the union operator between substitutions. Suppose $\sigma_1$ and $\sigma_2$ are two substitutions, then $\{\!\{\sigma_1 \uplus \sigma_2\}\!\}$ means applying the substitution $\sigma_1$ and $\sigma_2$ parallel. If there are conflict between $\sigma_1$ and $\sigma_2$, preference is given to the substitution in $\sigma_1$. Formally, suppose a substitution $\sigma_{inter}$ where $dom(\sigma_{inter}) = dom(\sigma_1) \cap dom(\sigma_2)$ and $\sigma_{inter} \subseteq \sigma_2$, we have: $\{\!\{\sigma_1 \uplus \sigma_2\}\!\} = \{\!\{\sigma_1 \uplus (\sigma_2 \setminus \sigma_{inter})\}\!\}$.

We write $\otimes$ the application operator between substitutions, which means apply the second substitution to the first. Formally: $(x_k \leftarrow e_k)^{k \in K} \otimes \sigma = (x_k \leftarrow e_k\{\!\{\sigma\}\!\})^{k \in K}$.

We write $\odot$ the composition operator between substitutions. When there are two substitution applied sequentially, we can use this operator to compose these two substitutions as one substitution. Formally, $e'\{\!\{\sigma_1 \odot \sigma_2\}\!\} = (e'\{\!\{\sigma_1\}\!\})\{\!\{\sigma_2\}\!\}$

Supposing $\sigma = (x_k \leftarrow e_k)^{k \in K}$, it's easy to see we can derive that:

$$\begin{aligned}(\sigma \otimes \sigma_1) \otimes \sigma_2 &= (x_k \leftarrow e_k\{\!\{\sigma_1\}\!\})^{k \in K} \otimes \sigma_2 \\ &= (x_k \leftarrow e_k\{\!\{\sigma_1\}\!\}\{\!\{\sigma_2\}\!\})^{k \in K} \\ &= (x_k \leftarrow e_k\{\!\{\sigma_1 \odot \sigma_2\}\!\})^{k \in K} \\ &= \sigma \otimes (\sigma_1 \odot \sigma_2)\end{aligned}$$

#### 2.1.3 Valuation

For a set $V = (x_k)^{k \in K}$ of variables, we denote $\rho(V) = (x_k \leftarrow v_k)^{k \in K}$ a valuation defined on $V$, where $v_k$ is an element in data domain of $x_k$. It's easy to see that a valuation is a specific substitution.

For example, let $fv$ be a set of free variables $fv = \{x, y\}$ with $x$ an Integer variable and $y$ a Boolean variable, then $\{x \leftarrow 1, y \leftarrow True\}$ is a valuation defined on $fv$.

### 2.2 Open Automaton

Open Automata[12] are semantic symbolic models representing the operational semantics of open data-dependant systems. In[12] a set of structural operational semantics rules is introduced, which defines the behavioural semantics of Open pNets in terms of Open Automata.

Each open automaton consists of a set of *States* and a set of *Open Transitions*, each with its (disjoint) set of *state variables*, and *Open Transitions* relates the behavior of holes (encoding the environment) with the behavior of the system. This section will show the formal definition of an open automaton.

**Definition 2.1. Open Automaton:** An open automaton is a structure $A = \langle J, \mathcal{S}, s_0, \mathcal{T} \rangle$ where:

- $J$ is a set of holes indices,
- $\mathcal{S}$ is a set of states and $s_0$ is a state among $\mathcal{S}$,
- $\mathcal{T}$ is a set of open transitions and for each t $\in \mathcal{T}$ there exists $J'$ with $J' \subseteq J$, such that t is an open transition over J' and $\mathcal{S}$.

**Definition 2.2. Open Transition:** An open transition of an open automaton $\langle J, \mathcal{S}, s_0, \mathcal{T} \rangle$ is a structure of the form

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

where $J' \subseteq J$, $s, s' \in \mathcal{S}$ and $\beta_j$ is an action of the hole $j$. Holes (represented above by their indices) are used to represent unspecified subsystems, that are parts of the environment. These unspecified subsystems can have any possible behaviour, but their interaction with the system is specified by the predicate *Pred*. $\alpha$ is an action expression denoting the resulting action of this open transition. We call *source variables* of an OT the union of all variables in the terms $\beta_j$ and $\alpha$, and the state variables of $s$. *Pred* is a predicate over the source variables. *Post* is a set of variable substitutions represented as $(x_k \leftarrow e_k)^{k \in K}$ where $x_k$ are state variables of $s'$, and $e_k$ are expressions over the source variables. Open transitions are identified modulo logical equivalence on their predicate. In the algorithms, this will be checked using a combination of predicate inclusions.

Suppose $OT$ is an open transition starting from state $s$ to $t$. We denote the set of all the free variables in open transition $OT$ as $vars(OT)$, and $vars(s)$ is set of all the state variables of state $s$. Last, $fv_{OT}$ denotes the set of variables in the open transition $OT$ besides all the state variables. More precisely, $fv_{OT} = vars(OT) \setminus \big(vars(s) \cup vars(s')\big)$.

These definitions will be widely used in subsequent sections.

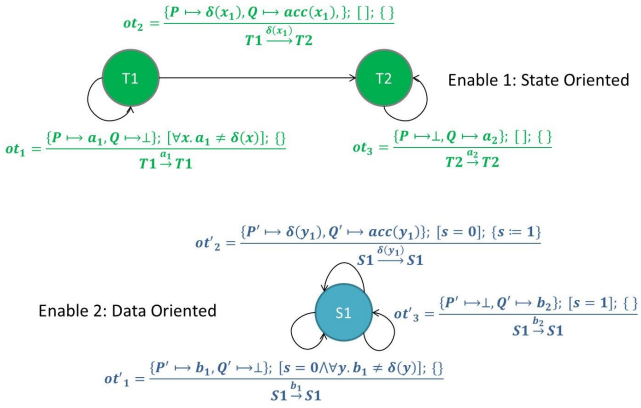



**Figure 2.** The 2 open automata derived from pNets in Fig. 1

Figure 2 shows two open automata generated by the pNets in Figure 1. Here we don't need to figure out the process of this generation, and just need to understand that, for any system modeled by a pNet, we can generate an open automaton, representing the behavioral semantics of the original pNet, and any property we proved on this open automaton is also owned by original pNets.

This provides us a clear way to verify the property of systems. We can use some open data-dependent modeling language, like pNet, which are more readable and easier to use, to model the systems we are interested in. Then, we can use some operational rules to construct an open automaton, which is less readable but easier for operation and verification, from the original system we modeled.

### 2.3 Semantic Open Automata

Previous researches were taking a semantics and logical understanding of these automata: *Semantic Open Automata* are closed under a simple form of refinement that allows to refine the predicate, or substitute any free variable by an expression. Formally:

**Definition 2.3. Semantic Open Automata:** A semantic open automaton is an open automaton $oa = \langle J, \mathcal{S}, init, \mathcal{T} \rangle$, such that for any open transition $ot \in \mathcal{T}$:

Let $\sigma$ be any substitution such that $dom(\sigma) \cap (vars(s) \cup vars(s')) = \emptyset$, let $pred$ be any predicate on $vars(ot)$, then

$$\frac{\beta\{\sigma\}, Pred\{\sigma\} \wedge pred\{\sigma\},\ Post \otimes \sigma}{s \xrightarrow{\alpha\{\sigma\}} s'} \in \mathcal{T}$$

In fact, the reason why this definition deserves our attention is, it's closely related to the nature of symbolic systems. In semantics, symbolic attribution implies each variables in system can represent a huge set of values. In terms of it, any open transition in an open automaton, can represent large sets of *ground open transitions*, in which all the variables are valuated with constants, as long as these constants satisfy the predicate of original transition. Thus, thanks to the above "closure" definition, for any open transition $OT$ in a semantic open automata $oa$, not only $OT$ can represent large sets of ground open transitions, but also for any subset of these ground open transitions, one can always find an open transition $OT'$ in $oa$, which can represent this subset.

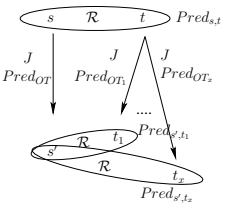
With this definition, researchers defined a relation between semantic open automata, called FH-Bisimulation [12], which have an interesting "Composability" property with respect to pNet composition, allowing compositional reasoning on open systems.

### 2.4 FH-Bisimulation

Bisimulations are equivalence relations between transition systems, where systems behave in the same way in the sense of one system simulates the other, in terms of the *actions* they do, not of their internal state.

FH-Bisimulation is a kind of strong bisimulation relation between *Semantical Open automata*, introduced in . FH-Bisimulation can be formally defined in the following way:

**Definition 2.4. FH-Bisimulation:** Suppose that $A_1 = \langle J, \mathcal{S}_1, init_1, \mathcal{T}_1 \rangle$ and $A_2 = \langle J, \mathcal{S}_2, init_2, \mathcal{T}_2 \rangle$ are two semantic open automata with identical activated holes, with disjoint state variables. Then $\mathcal{R}$ is an FH-Bisimulation iff for any triple $(s, t | Pred_{s,t}) \in \mathcal{R}$ where $s \in \mathcal{S}_1 \wedge t \in \mathcal{S}_2$, we have the following:



- For any open transition $OT \in \mathcal{T}_1$ starting from $s$

$$\frac{\beta_j^{j \in J'},\ Pred_{OT},\ Post_{OT}}{s \xrightarrow{\alpha} s'}$$

there exists a set of open transitions $OT_x^{x \in X} \subseteq \mathcal{T}_2$ starting from $t$

$$\frac{\beta_{j_x}^{j_x \in J'_x},\ Pred_{OT_x},\ Post_{OT_x}}{t \xrightarrow{\alpha_x} t'_x}$$

such that $J' \subseteq J,\ J'_x \subseteq J,\ \forall x. J' = J'_x,\ \{s', t'_x | Pred_{s',t'_x}\} \in \mathcal{R}$ and

$$Pred_{s,t} \wedge Pred_{OT} \Longrightarrow \bigvee_{x \in X} (\forall j \in J'. \beta_j = \beta_{j_x} \wedge \alpha = \alpha_{j_x} \wedge Pred_{OT_x} \wedge Pred_{s',t'_x}\{Post_{OT} \uplus Post_{OT_x}\})$$

that means each open transition starting from $s$ in $\mathcal{T}_1$ can be covered by a set of transitions $OT_x^{x \in X}$ starting from $t$ in $\mathcal{T}_2$.

- and symmetrically, for any open transition starting from $t$ in $\mathcal{T}_2$, should be covered by a set of transitions starting from $s$ in $\mathcal{T}_1$.

The word *covered* here means that each *ot* on one side, representing symbolically a set of "ground" transitions, can be covered on the other side by several open transitions on the other sides, each implementing transitions for a subset of *ot* instantiations, and eventually leading to different though still equivalent, states.

The (slightly complicated) implication means that for each instantiation fulfilling $Pred_{s,t} \wedge Pred_{OT}$, it is possible to find an instantiation of one of the corresponding $OT_x$, satisfying the corresponding property.

It maybe a little unclear also why we need $Pred_{s,t}$ in Triples. In fact, recall the name of FH-Bisimulation, here "FH" is a short cut of "Formal Hypotheses", showing the fact that FH-Bisimulation is a relation based on a certain hypotheses. Thus, $Pred_{s,t}$ in Triples is a formal way to describe this hypotheses, and in particular express the relations between the state variables of the two open automata.

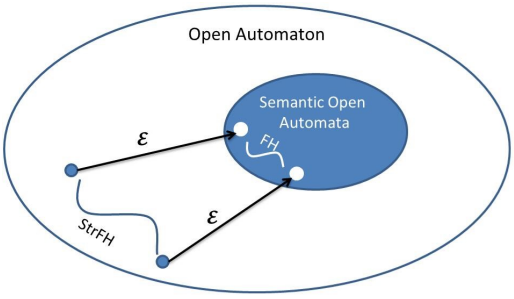
In [2]'s work, it is proved that FH-Bisimulation is an equivalence, which is reflexive, symmetric and transitive. And as mentioned before, FH-Bisimulation also has a powerful property of *Composability*: Given two semantic open automata such that there exist a FH-Bisimulation between them, if we compose these two automata with another semantic open automata (technically, using an open pNet), then the two result automata is still FH-Bisimulation.

And that's an important motivation of our work. Obviously, any semantic open automaton is infinite, thus it's hard to do any verification of FH-Bisimulation on semantic open automata. What we want, is to generalize this FH-Bisimulation to a relation between all the open automata, not only between the "infinite" subset of open automata, and thus it's become possible for us to do any verification on this relation. At next section, we will discuss how we do that.

## 3 StrFH-Bisimulation

As mentioned before, the *closure* property of semantic open automata bring the important composability properties, but also does not allow implementing FH-Bisimulation algorithms between them. So we want to generalize this FH-Bisimulation to a relation between all the open automata, including the finite ones generated from the pNets semantic rules[12].

In this section, we first give the definition of our new relation *StrFH-Bisimulation*, which is defined between open automata. Then we define an Expansion function $\mathcal{E}$, such that for any open automata we can build a corresponding semantic open automata. Finally, with the above definitions, we will show how our relation is consistent and complete with respect to the FH-Bisimulation. From following figure, we can easily have a first intuition about what all these definitions are:




### 3.1 StrFH-Bisimulation

We provide a new relation, which is between open automata, called StrFH-Bisimulation, defined formally as:

**Definition 3.1. StrFH-Bisimulation:** Let $StrA_1 = \langle J, \mathcal{S}_1, init_1, \mathcal{T}_1 \rangle$ and $StrA_2 = \langle J, \mathcal{S}_2, init_2, \mathcal{T}_2 \rangle$ be two open automata with identical activated holes, and disjoint state variables. Then $\mathcal{R}$ is an StrFH-Bisimulation iff for any triple $(s, t | Pred_{s,t}) \in \mathcal{R}$, where $s \in \mathcal{S}_1 \wedge t \in \mathcal{S}_2$ and we have the following:

- For any open transition $OT \in \mathcal{T}_1$ starting from $s$

$$\frac{\beta_j^{j \in J'},\ Pred_{OT},\ Post_{OT}}{s \xrightarrow{\alpha} s'}$$

there exist a set of open transitions $OT_x^{x \in X} \subseteq \mathcal{T}_2$ starting from $t$

$$\frac{\beta_{j_x}^{j_x \in J'_x},\ Pred_{OT_x},\ Post_{OT_x}}{t \xrightarrow{\alpha_x} t'_x}$$

such that $J'_x \subseteq J,\ \forall x. J' = J'_x,\ \{s', t'_x | Pred_{s',t'_x}\} \in \mathcal{R}$ and

$$\forall fv_{OT}.\{Pred_{s,t} \wedge Pred_{OT} \Longrightarrow \bigvee_{x \in X} [\exists fv_{OT_x}.(\forall j \in J'.\beta_j = \beta_{j_x} \wedge \alpha = \alpha_{j_x} \wedge Pred_{OT_x} \wedge Pred_{s',t'_x}\{\!|Post_{OT} \uplus Post_{OT_x}|\!\})]\} \quad (1)$$

that means open transition start from $s$ in $\mathcal{T}_1$ can be covered by a set of transitions $OT_x^{x \in X}$ starting from $t$ in $\mathcal{T}_2$

- and symmetrically, for any open transition starting from $t$ in $\mathcal{T}_2$ can be covered by a set of transitions starting from $s$ in $\mathcal{T}_1$.

There may be another confusing term here: $\forall fv_{OT}.\varphi$, for a set of variables $fv_{OT}$. This means that for any valuation of all the variables in $fv_{OT}$, the inside formula is true. For example, for an open transition $OT$, if $fv_{OT} = \{x, y, z\}$, then $\forall fv_{OT}.\varphi$ means $\forall x, y, z.\varphi$.

### 3.2 Expansion

Here, we define a Expansion function $\mathcal{E}$, such that for any open automaton, we can use this function to get a corresponding semantic open automaton.

**Definition 3.2. Expansion:** Let $OA$ be the set of all open automata, then we define an Expansion Function $\mathcal{E}$ :

$$\mathcal{E} : OA \rightarrow OA$$

where for any open automata $A = \langle J, \mathcal{S}, s_0, \mathcal{T} \rangle$, $\mathcal{E}(A) \in OA$, we can have $\mathcal{E}(A) = \langle J, \mathcal{S}, s_0, \mathcal{T}' \rangle$.

The only difference between $A$ and $\mathcal{E}(A)$ is on transitions set, where $\mathcal{T}'$ is the smallest set of open transitions such that: for any open transition $OT \in \mathcal{T}$

$$\frac{\beta_j^{j\in J},\ Pred_{OT},\ Post_{OT}}{s \xrightarrow{\alpha} s'}$$

let's denote

$$OT\{\!|\sigma, pred|\!\} = \frac{\beta_j^{j\in J}\{\!|\sigma|\!\},\ Pred\{\!|\sigma|\!\} \wedge pred\{\!|\sigma|\!\},\ Post \otimes \sigma}{s \xrightarrow{\alpha\{\!|\sigma|\!\}} s'}$$

Then for any substitution $\sigma$ where $dom(\sigma) \subseteq fv(OT)$, for any additional predicate $pred$, we have:

$$OT\{\!|\sigma, pred|\!\} \in \mathcal{T}'$$

Obviously, in this definition, after applying function $\mathcal{E}$, the result we have is an open automaton. The following theorem states that this result $\mathcal{E}(A)$ is a semantic open automaton.

**Theorem 3.3.** *For any open automaton A, the expanded open automaton $\mathcal{E}(A)$ is a semantic open automaton.*

We give a formal proof for this theorem in Appendix **??**. Briefly, the proof will show that all the open transitions in the open automaton generated by Expansion Function, meet the "Closure" property of Semantic Open Automata.

***Note:*** by construction, the states of $\mathcal{E}(A)$ are same as the states of $A$. As a consequence, all relations we construct as triple sets $\mathcal{R} = \{(s, t | Pred_{s,t})\}$, are both relations on $A$ and $\mathcal{E}(A)$.

### 3.3 Correct Correspondence

Here we want to prove that StrFH-Bisimulation correctly corresponds to FH-Bisimulation, as expressed by:

**Theorem 3.4.** ***Correctness:*** *For any two open automata $A_1$ and $A_2$, construct the two corresponding semantic open automata $\mathcal{E}(A_1)$ and $\mathcal{E}(A_2)$. Then we have: Suppose Triple Set $\mathcal{R}$ is a StrFH-Bisimulation between $A_1$ and $A_2$, then $\mathcal{R}$ is also a FH-Bisimulation between $\mathcal{E}(A_1)$ and $\mathcal{E}(A_2)$.*

We give a formal proof for Theorem 3.4 in [13], Appendix B.1. Briefly, the intuition of the proof is: for any open transition $OT'$ in $\mathcal{E}(A_1)$, supposing it's original open transition is $OT$ in $A_1$, we know that $OT$ can be covered with a set of open transition $OT_x^{x\in X}$ in $A_2$. From this set, We can always construct a set of *ground* open transitions $\{OT'_{x,y}\}^{x\in X \wedge y\in Y}$ belonging to $\mathcal{E}(A_2)$ such that this set of open transitions covers the $OT'$. And symmetrically, for any open transition in $\mathcal{E}(A_2)$.

### 3.4 Complete Correspondence

Here we want to prove that StrFH-Bisimulation completely corresponds to FH-Bisimulation, as expressed by:

**Theorem 3.5.** ***Completeness:*** *For any two open automata $A_1$ and $A_2$, construct the two corresponding semantic open automata $\mathcal{E}(A_1)$ and $\mathcal{E}(A_2)$. Then we have: Suppose Triple Set $\mathcal{R}$ is a FH-Bisimulation between $\mathcal{E}(A_1)$ and $\mathcal{E}(A_2)$, then $\mathcal{R}$ is also a StrFH-Bisimulation between $A_1$ and $A_2$.*

We give a formal proof for Theorem 3.5 in [13], Appendix B.2. The intuition of the proof is: for any open transition $OT$ in $A_1$, applying any (ground) valuation $\rho_i$ defined on $fv_{OT}$ to $OT$, the result can be represented by an open transition $OT'_i$, which belongs to $\mathcal{E}(A_1)$. We know that $OT'_i$ can be covered by a set of open transitions $\{OT'_{x,i}\}^{x\in X}$ in $\mathcal{E}(A_2)$, corresponding to open transitions $\{OT_{x,i}\}^{x\in X}$ in $A_2$ (may contain duplicate open transition). Collecting all the open transitions $\{OT_{x,y}\}^{x\in X}$ constructed with each valuation $\rho_y$, and we can finally get a larger set $\{OT_{x,y}\}^{x\in X \wedge y\in Y}$. We will prove $OT$ is covered by this larger set. And symmetrically for any open transition in $A_2$.

### 3.5 Running Example

Recall the two open automata in Figure 2, both automata are generated from enable model in Figure 1, respectively by pNets *Enable1* and *Enable2*. It's easy to see, the former automaton, we name it as $A_1$, uses different states to represent before and after activation (the $\delta$ transition); and the latter automaton, we name it as $A_2$, uses different values of a state variable $s$ to represent that. It's a very typical situation that two equivalent systems essentially express the same meaning in different ways.

Here we provide these two automata as a running example, to illustrate why Triple Set $\{(T_1, S_1 | s = 0), (T_2, S_1 | s = 1)\}$ is a StrFH-Bisimulation between these two automata. Details are presented in [13], Appendix B.3.

## 4 Checking Algorithm

As we mentioned before, in the formal definition, for any two open automata $\langle J_1, \mathcal{S}_1, init_1, \mathcal{T}_1 \rangle$, $\langle J_2, \mathcal{S}_2, init_2, \mathcal{T}_2 \rangle$, a StrFH-Bisimulation $\mathcal{R}$ is a set of triples $\{(s, t | Pred_{s,t})\}$.

Given two open automata and a set of Triples, it seems that following the definition, we can easily verify if the given relation is a StrFH-Bisimulation. But for that, we need to check whether a first order logic formula is a tautology or not, and due to the fact that first order logic is undecidable, and that we want to use this approach with realistic data

types and expressions in the pNets, it may becomes very hard.

Thanks to the development of Satisfiability Modulo Theory (SMT) technology, which has a great ability for solving satisfiable problems of first order logic[6], we have a practical way to do this checking: for any proof obligation, we can generate its negation, and use an SMT-solver to check if it is satisfiable. If yes, thus the original proof obligation is not a tautology. This does not break the undecidability problem, but gives us a semi-decidable method: our StrFH-Bisimulation check is decidable whenever satisfiability of the first-order formulas used in the relation is decidable by our SMT encoding. In practice, this last property depends on the axiomatisation of the data domains and operators used in a particular use case.

In this section we will first explain a basic algorithm, which checks whether a given Triple Set is a StrFH-Bisimulation between two input open automata. Note that, in this algorithm, the user needs to provide the Triple Set, which may be difficult for some complicated systems. In next section we will give another algorithm which can automatically compute a StrFH-Bisimulation for given open automata.

### 4.1 Checking Algorithm

From two open automaton $A_1 = \langle J_1, \mathcal{S}_1, init_1, \mathcal{T}_1\rangle$ , $A_2 = \langle J_2, \mathcal{S}_2, init_2, \mathcal{T}_2\rangle$, and a set of triples $\mathcal{R} = \{(s, t|Pred_{s,t})\}$, our algorithm will output whether the given Triple Set is a StrFH-Bisimulation between $A_1$ and $A_2$ or not. We explain this algorithm in two parts:

#### 4.1.1 Traverse Triples

The main function is *TraverseTriples*. It this function, we will traverse all the Triples, and check whether all of them meet the definition of StrFH-Bisimulation. For each Triple $(s, t|Pred_{s,t})$, we will check both directions: first, check if all the open transitions from $s$ can be covered by the open transitions from $t$ under the $Pred_{s,t}$; then symmetrically, check if all the open transitions from $t$ can be covered by the open transitions from $s$ under the same predicate. The algorithm is the same for both, thus we only show the first direction.

Let Triple $(s, t|Pred_{s,t})$ be the one we are checking. For each open transition $OT$ from state $s$, let $s'$ be it's target. Then, we will filter all the open transitions starting from $t$, and collect a set of open transitions $OTSet$ and a set of formulas $PredSet$, such that for any open transition $OT_x \in OTSet$, we have:

- $OT$ has the same active holes as $OT_x$
- let $t'_x$ be the target of transition $OT_x$, then there exists a Triple $(s', t'_x|Pred_{s',t'_x})$ in $\mathcal{R}$

and each predicate formula $Pred_{s',t'_x}$ is the predicate formula of Triple $(s', t'_x|Pred_{s',t'_x})$, corresponding to the open transition $OT_x$.

Then, we call function *Verify*$(OT, Pred_{s,t}, OTSet, PredSet)$. In that function, will generate and check the proof obligation (that is Formula (1) from Definition 3.1), and return a boolean value as result. If this result is false, means proof obligation is not a tautology, thus Triple $(s, t|Pred_{s,t})$ doesn't meet the definition, and Triple Set $\mathcal{R}$ won't be a StrFH-Bisimulation between $A_1$ and $A_2$, so function *TraverseTriples* will return false.

If all the Triples pass this check, it means all the Triples in $\mathcal{R}$ meet the definition, thus we can say $\mathcal{R}$ is a StrFH-Bisimulation between $A_1$ and $A_2$ and return true.

**Algorithm 1** Verify StrFH-Bisimulation with Given Triple Set

**input:** $A_1$ and $A_2$ are two open automata, where $A_1 = \langle J_1, \mathcal{S}_1, init_1, \mathcal{T}_1\rangle$, $A_2 = \langle J_2, \mathcal{S}_2, init_2, \mathcal{T}_2\rangle$. $\mathcal{R}$ is a set of triples, where $\mathcal{R} = \{(s, t|Pred_{s,t})\}$.
**output:** a boolean value, means whether $\mathcal{R}$ is a StrFH-Bisimulation between $A_1$ and $A_2$

```
1:  function TraverseTriples(A1, A2, R)
2:      for each Triple (s, t|Pred_{s,t}) in R do
3:          for each Open Transition OT from state s, suppose it's target is s' do
4:              OTSet ← an empty set for Open Transitions
5:              PredSet ← an empty set for Formula
6:              for each Open Transition OT' from state t, suppose it's target is t' do
7:                  if there exists a Triple (s', t'|Pred_{s',t'}) ∈ R and OT have same activated holes as OT' then
8:                      OTSet.add(OT')
9:                      PredSet.add(Pred_{s',t'})
10:                 end if
11:             end for
12:             if OTSet is empty then
13:                 return False
14:             end if
15:             res ← Verify(OT, OTSet, Pred_{s,t}, PredSet)
16:             if res is False then
17:                 // means proof obligation proofOb is not a tautology.
18:                 return False
19:             end if
20:         end for
21:     end for
22:     return True
23: end function
```

#### 4.1.2 Generate and Verify Proof Obligation

Then we come to the details of function *Verify*. For the input open transition $OT$, open transition set $OTSet$, formula $Pred_{s,t}$ and formula set $PredSet$, without losing generality,

we can suppose that:

$$OT = \frac{\beta_j^{j\in J'},\ Pred_{OT},\ Post_{OT}}{s \xrightarrow{\alpha} s'}$$

and for any open transition $OT_x \in OTSet$ where $x \in X$ is the index of $OTSet$, we have:

$$OT_x = \frac{\beta_j^{j\in J'_x},\ Pred_{OT_x},\ Post_{OT_x}}{t \xrightarrow{\alpha_x} t'_x}$$

and have a corresponding predicate $Pred_{s',t'_x} \in PredSet$. Thus, the proof obligation by definition as following:

$$\begin{aligned}\forall fv_{OT}.\{&Pred_{s,t} \wedge Pred_{OT} \Longrightarrow \\ &\bigvee_{x\in X}[\exists fv_{OT_x}.(\forall j \in J'.\beta_j = \beta_{j_x} \wedge \alpha = \alpha_x \\ &\wedge Pred_{OT_x} \wedge Pred_{s',t'_x}\{\!|Post_{OT} \uplus Post_{OT_x}|\!\})]\}\end{aligned}$$

and note that if negate this proof obligation, then we will get:

$$\begin{aligned}\exists fv_{OT}.\{&Pred_{s,t} \wedge Pred_{OT} \wedge \\ &\bigwedge_{x\in X}[\forall fv_{OT_x}.(\exists j \in J'.\beta_j \neq \beta_{j_x} \vee \alpha \neq \alpha_x \\ &\vee \neg Pred_{OT_x} \vee \neg Pred_{s',t'_x}\{\!|Post_{OT} \uplus Post_{OT_x}|\!\})]\}\end{aligned} \quad (2)$$

Thus, we can check the satisfiability of this Formula 2 with SMT-solver, if this Formula 2 is unsatisfiable, means the original proof obligation is a tautology, and function will return True; otherwise, return False.

The checking of proof obligation is strongly depending on SMT-solver, as long as SMT solver can determine all the generated proof obligation, then we can say our algorithm can determine StrFH-Bisimulation problem on given automata and Triple Set. If there exists a proof obligation can't be solved by SMT-solver, our algorithm will return *FAILED* and terminate.

Thus, our algorithm succeeds whenever the theories (over the data domains, expressions, and predicates) used in the SMT engine are decidable.

### 4.2 Correctness and Termination

#### 4.2.1 Correctness

The algorithm will traverse all the given triples, check whether all the triples satisfied definition. It's easy to see that, this period is closely corresponding to the definition of StrFH-Bisimulation, so we can say: Given two open automata $A_1$, $A_2$ and a set of triples $\mathcal{R}$, then $\mathcal{R}$ is a StrFH-Bisimulation between $A_1$ and $A_2$ iff the result of applying $A_1$, $A_2$ and $\mathcal{R}$ into this checking algorithm is *True*, when the case is decidable.

#### 4.2.2 Termination

It's easy to see that, for each Triple $(s, t|Pred_{s,t})$, as long as the open transitions starting from $s$ and starting from $t$ are finite, the algorithm will generate finite proof obligations according to this Triple. For each proof obligation, as long as it can be decided by SMT-solver, it will terminate and give a result.

Thus, we can say, as long as we input a finite Triple Set, and each Triple corresponding to finite open transitions(even inputted open automata are infinite), and the case is decidable, our algorithm will terminate.

**Algorithm 2** Generate and Verify Proof Obligation

**input:** $OT$ is an open transition, $OTSet$ is a set of open transitions, $Pred$ is a formula, $PredSet$ is a set of formulas.
**output:** *result*, a boolean value, means whether the proof obligation generated from input is a tautology or not.

```
1:  function VERIFY(OT, OTSet, Pred, PredSet)
2:      negateOb ← generate negation of proof obligation
3:      use SMT-solver to check if negateOb is satisfiable
4:      if SMT-solver doesn't terminate then
5:          // negateOb is not decidable by SMT-solver
6:          return FAILED
7:      end if
8:      if negateOb is sat then
9:          // negateOb is satisfiable, means original proof
        obligation is not a tautology
10:         return False
11:     else
12:         return True
13:     end if
14: end function
```

## 5 Generating Weakest StrFH-Bisimulation and StrFH-Bisimilar

In the previous section, we gave a checking algorithm, which accepts two open automata and a relation (given as a Triple set), and outputs whether the input Triple set is a StrFH-Bisimulation between the two input open automata.

Obviously, this algorithm has a strong need of a reasonable input. If user failed to find out or correctly encode a proper Triples Set, then algorithm can only get the result that a incorrect Triple set is not a StrFH-Bisimulation, but couldn't answer whether there exists a StrFH-Bisimulation between the two input automata.

And in fact, between any two open automata, there always exists many worthless StrFH-Bisimulation relations. For example, suppose there is a Triple set, where for each Triple $(s, t|Pred_{s,t})$ in this set, $Pred_{s,t}$ is always unsatisfiable. Obviously, this Triple set meet the definition of StrFH-Bisimulation, but it makes no sense.

For these reasons, in this section, we will first give a new algorithm, which can accept two open automata, and output the weakest StrFH-Bisimulation between the given open automata; Then, we will define a new property called StrFH-Bisimilar. If two open automata are StrFH-Bisimilar, it means

there exists a meaningful StrFH-Bisimulation between them; From the weakest StrFH-Bisimulation generated, we can check if given open automata are StrFH-Bisimilar. We give a formal proof of our algorithm's correctness, prove a (partial) termination property for this algorithm.

### 5.1 Generate Weakest StrFH-Bisimulation

We will first give a function called *GenerateWeakestStrFH*. The input of function *GenerateWeakestStrFH* are two open automata $A_1 = \langle J_1, \mathcal{S}_1, init_1, \mathcal{T}_1 \rangle$ and $A_2 = \langle J_2, \mathcal{S}_2, init_2, \mathcal{T}_2 \rangle$. It's output is a Triple set $TripleSet$, which is a StrFH-Bisimulation between $A_1$ and $A_2$. Here we will first describe the procedure of this function, and its pseudo-code in algorithm 3. In this section, we will show the fact that the output is a StrFH-Bisimulation between given open automata. Later in the next section, we will show why it's result is the weakest StrFH-Bisimulation.

**Algorithm 3** Generate Weakest StrFH-Bisimulation

**input:** $A_1$, $A_2$ two open automata, where $A_1 = \langle J_1, \mathcal{S}_1, init_1, \mathcal{T}_1 \rangle$ and $A_2 = \langle J_2, \mathcal{S}_2, init_2, \mathcal{T}_2 \rangle$.
**output:** $TripleSet$, a Triple's set, which is a Weakest StrFH-Bisimulation between two open automata

```
1:  function GenerateWeakestStrFH(StateStack)
2:      StateStack ← an empty stack for StatePair
3:      TripleSet ← an empty set for Triple
4:      PushReachableStatePairs(init_1, init_2, StateStack)
5:      for each StatePair (s, t) in StateStack do
6:          add a Triple (s, t|True) into TripleSet
7:      end for
8:      while StateStack is not empty do
9:          (s, t) ← StateStack.pop()
10:         let i be the index of Triple (s, t|Pred_{s,t}) in the TripleSet
11:         UpdatePredicate(i, TripleSet)
12:         if predicate of Triple (s, t|Pred_{s,t}) has been changed then
13:             for each pre-StatePair (s_pre, t_pre) of (s, t) do
14:                 if StateStack doesn't contains this StatePair (s_pre, t_pre) then
15:                     push StatePair (s_pre, t_pre) into StateStack
16:                 end if
17:             end for
18:         end if
19:     end while
20: end function
```

First let us give some definitions: We denote $(s, t)$ as a StatePair, where $s$ and $t$ are states from different open automata.

We denote $(s_{pre}, t_{pre})$ as a pre-StatePair of $(s, t)$ where there exists an open transition from $s_{pre}$ to $s$, and exists an open transition from $t_{pre}$ to $t$.

We denote $(s_{next}, t_{next})$ as a next-StatePair of $(s, t)$ where there exists an open transition from $s$ to $s_{next}$, and exists an open transition from $t$ to $t_{next}$.

We denote $(s', t')$ as a reachable StatePair of $(s, t)$ iff there exists a path from $(s, t)$ to $(s', t')$, where each step in this path is a StatePair move to it's next-StatePair.

At the very beginning, function will construct an empty StatePair stack, named $StateStack$. Function will also construct an empty Triple set, named $TripleSet$.

In fact, here we hope $TripleSet$ become the final output of the function, which means after the function terminating, $TripleSet$ become an StrFH-Bisimulation between $A_1$ and $A_2$, and any Triple in $TripleSet$ will meet the requirement: all the proof obligations generated from it are tautologies. And $StateStack$ here exists as a register, temporarily cache all StatePairs whose corresponding Triples may not satisfy this requirement.

To initialize $StateStack$ and $TripleSet$, function will call a sub-function *PushReachableStatePairs*, inputting two states: $init_1$ and $init_2$ and an empty StatePair stack: $StateStack$. It's a recursive function, and after it finished, $StateStack$ will contain all the StatePairs which are reachable from $(init_1, init_2)$. Then, for each StatePair $(s, t)$ in $StateStack$, function will add a corresponding Triple $(s, t|True)$ into a Triple set $TripleSet$.

After above initialization, function will enter a loop that: pops a StatePairs from $StateStack$ and updates the corresponding Triple, loops until the stack $StateStack$ becomes empty.

In each loop, let $(s, t)$ be the StatePair just popped, function will find the corresponding Triple $(s, t|Pred_{s,t})$ in the $TripleSet$, and call function *UpdatePredicate* to update this Triple. After processed by function *UpdatePredicate*, Triple $(s, t|Pred_{s,t})$ will definitely meet the requirement: all the proof obligations generated from this Triple are tautology.

Finally, function will check if $(s, t|Pred_{s,t})$ has been updated after calling *UpdatePredicate*. If not, the function will continue to pop next StatePair. If changed, we can't simply go to next StatePair, because the change of $Pred_{s,t}$ may change the proof obligations of other Triples, we need to push their corresponding StatePairs into $StateStack$ again.

In more detail, let $\{(s_{pre_x}, t_{pre_x})\}^{x \in X}$ be all pre-StatePairs of $(s, t)$. Due to the fact that $Pred_{s,t}$ has been changed to $newPred_{s,t}$, the proof obligation of Triple $(s_{pre_x}, t_{pre_x}|Pred_{pre_x})$ will also be changed, and we need to update this Triple again. So, we will push StatePair $(s_{pre_x}, t_{pre_x})$ into $StateStack$ , and the function will definitely pop this StatePair and update corresponding Triple at some later time.

We know that, at the beginning, $StateStack$ contains all the reachable StatePairs. Each time a StatePair is popped, the corresponding Triple will be update so that it will meet the current requirement. On the other hand, after updating, if this updating changes other Triples' proof obligation, make these Triples may no longer meet the requirement, then their corresponding StatePair will be pushed into stack

again. Thus, we can see, if any Triple which does not conclusively meet the requirement, it's corresponding StatePair will be pushed into stack. As long as this function *GenerateWeakestStrFH* terminates, which means the stack becomes empty, all the Triples in $TripleSet$ will meet the requirement, and $TripleSet$ is definitely a StrFH-Bisimulation between $A_1$ and $A_2$.

#### 5.1.1 Sub-Functions

Recall that there are two sub-functions during generating the weakest StrFH-Bisimulation. Here we will give a short summary for these two sub-functions, detailed description and pseudo-code can be seen in [13], Appendix C.1.

Sub-function *PushReachableStatePairs* is a recursive function, a variant of DFS (depth first search). Every time it visits a StatePair $(s, t)$, it will pop $(s, t)$ into stack, and call the same function with next-StatePair of $(s, t)$ recursively. Thus, after function *PushReachableStatePairs* terminates, the stack will contain all the reachable StatePairs from initial input $(init_1, init_2)$.

Sub-function *UpdatePredicate* will generate and verify the proof obligation of given Triples, using the SMT engine, like what we did in Section 4.1.2. If the proof obligation is not a tautology, the function will update the predicate with the conjunction of original predicate and simplified proof obligation. So that after this updating, the new proof obligation generated by this updated predicate will definitely be a tautology.

#### 5.1.2 Weakestness

The output of function *GenerateWeakestStrFH* will be the weakest StrFH-Bisimulation between two given open automata. Remember that for any pair of states $(s, t)$, there can be only one triple $(s, t|Pred_{s,t})$ in a FH-bisimulation relation. Then the weakest StrFH-bisimulation between 2 automata is the one in which each Triple is the weakest:

**Definition 5.1. Weakest Triple:** Triple $(s, t|Pred_{s,t})$ is a weakest Triple between open automata $A_1$ and $A_2$, meaning that $A_1$ and $A_2$ contains states $s$ and $t$ respectively, and for any predicate formula $Pred_{new}$, which can't imply the predicate $Pred_{s,t}$, any Triple Set containing Triple $(s, t|Pred_{new})$ won't be a StrFH-Bisimulation between $A_1$ and $A_2$.

**Theorem 5.2.** ***Weakest StrFH-Bisimulation:*** *For a StrFH-Bisimulation $\mathcal{R}$ between open automata $A_1$ and $A_2$, all the Triples in $\mathcal{R}$ are Weakest Triples, meaning $\mathcal{R}$ is a Weakest StrFH-Bisimulation.*

**Theorem 5.3.** ***Weakestness:*** *For any pair of open automata $A_1$ and $A_2$, the TripleSet computed by algorithm is the weakest StrFH-Bisimulation between $A_1$ and $A_2$.*

We give a detailed proof of these two theorems in [13], Appendix C.2.

Proof of theorem 5.2 relies on the fact that a *weakest* Triple can be either (strictly) logically weaker, or incomparable, to the original Triple, but in both cases having at least one weakest Triple contradicts definition 5.1.

Proof of theorem 5.3 is by mathematical induction over the set of Triplpes: for all Beginning Triples (we define it in proof), we prove that they satisfy Definition 5.1 after updating. Then, let $(s, t|Pred_{s,t})$ be a general Triple, suppose all the other Triple satisfy Definition 5.1, then $(s, t|Pred_{s,t})$ will also satisfy Definition 5.1.

### 5.2 Checking StrFH-Bisimilar

With above functions, we can finally check if two open automata are StrFH-Bisimilar. First let us give a formal definition for StrFH-Bimilar:

**Definition 5.4. FH-Bisimilar:** Let $A_1 = \langle J_1, \mathcal{S}_1, init_1, \mathcal{T}_1 \rangle$ and $A_2 = \langle J_2, \mathcal{S}_2, init_2, \mathcal{T}_2 \rangle$ be two open automata. $A_1$ and $A_2$ are FH-Bisimilar if and only if there exists a StrFH-Bisimulation $\mathcal{R}$ between $A_1$ and $A_2$, such that:

- for any Triple $(s, t|Pred_{s,t})$ in $\mathcal{R}$, formula $(Pred_{s,t} \implies False)$ is not a tautology, or we can say $Pred_{s,t}$ is satisfiable.
- there must exists a predicate $Pred_{init}$ such that Triple $(init_1, init_2|Pred_{init}) \in \mathcal{R}$

In this definition, all the predicates are satisfiable, meaning the hypothesis of this Bisimulation are satisfiable; and there exists a initial Triple in this Bisimulation, meaning this Bisimulation will become effective as long as both two open automata start from initial state at same time.

**Algorithm 4** Check StrFH-Bisimilar

**input:** $A_1$, $A_2$ two open automata, where $A_1 = \langle J_1, \mathcal{S}_1, init_1, \mathcal{T}_1 \rangle$ and $A_2 = \langle J_2, \mathcal{S}_2, init_2, \mathcal{T}_2 \rangle$.
**output:** *result*, a boolean value, means whether $A_1$ and $A_2$ are StrFH-Bisimilar

```
1:  function CheckStrFH-Bisimilar(A1, A2)
2:      TripleSet ← GenerateWeakestStrFH(StateStack)
3:      for each Triple (s, t|Pred_{s,t}) ∈ TripleSet do
4:          delete this Triple if Pred_{s,t} is unsatisfiable
5:      end for
6:      if TripleSet contains Triple (init1, init2, Pred_init) then
7:          return True
8:      else
9:          return False
10:     end if
11: end function
```

After we generated a weakest StrFH-Bisimulation between $A_1$ and $A_2$ called $\mathcal{R}$, in which any Triple has weakest predicate, we can easily check if $A_1$ and $A_2$ are StrFH-Bisimilar by: first, deleting all the Triples with unsatisfiable Predicate

formula, and get a new Triple set $\mathcal{R}_{new}$; then, if $\mathcal{R}_{new}$ contains Triple $(init_1, init_2|Pred_{init})$, we can say $A_1$ and $A_2$ are StrFH-Bisimilar. Algorithm 4 shows the pseudo-code of this procedure.

### 5.3 Algorithm Correctness

Proving this algorithm is correct also contains two parts: *Correctness* and *Completeness*.

**Theorem 5.5.** ***Correctness:*** *Inputting two open automata $A_1$ and $A_2$, as long as the result of algorithm is $True$, means $A_1$ and $A_2$ are StrFH-Bisimilar.*

**Theorem 5.6.** ***Completeness:*** *Inputting two open automata $A_1$ and $A_2$, if $A_1$ and $A_2$ are StrFH-Bisimilar, then the result of algorithm is $True$ as long as the problem is decidable by our algorithm.*

#### 5.3.1 Correctness

We give a formal proof for Theorem 5.5 in [13], Appendix D.1. Key point is: after deleting all the unsatisfiable Triples in the generated StrFH-Bisimulation, the result is still a StrFH-Bisimulation.

#### 5.3.2 Completeness

We give a formal proof for Theorem 5.6 in [13], Appendix D.2. Key point is: Contrapositive of this Theorem will be easier to prove. With the Theorem 5.3, it's easy to see that, as long as the function return $False$, it's impossible to find a StrFH-Bisimulation between given automata which contains a initial Triple $(init_1, init_2|Pred_{init})$ and $Pred_{init}$ is satisfiable.

### 5.4 Conditional Termination

In this section, we give a conditional termination result for algorithm 4: for any two given open automata, if they are finite, and data domain of all variables in given automata are finite, and the StrFH-Bisimilar problem for these two automata is decidable, then our algorithm will terminate.

Noting that it's may not the most general hypothesis ensuring termination, but it's a very reasonable hypothesis, due to the fact that many widely used realistic systems meet this requirement.

*Proof.* Suppose there are two finite open automata $A_1$ and $A_2$.

Considering the function *GenerateWeakestStrFH*, because $A_1$ and $A_2$ are finite, then we will generate a finite $StateStack$, and thus have a finite $TripleSet$.

And for each time we call function *UpdatePredicate* with an input Triple $(s, t|Pred_{s,t})$, due to the fact that both two automata are finite, we will only generate finite proof obligation.

On the other hand, denoting $\{\rho_x\}^{x\in X}$ as set of all possible valuations on all variables. If the data domain of all variables in $A_1$ and $A_2$ are finite, we know that $\{\rho_x\}^{x\in X}$ is definitely finite. For any Predicate Formula $Pred_{s,t}$, we denote $\{\rho_y\}^{y\in Y}$ as a set of valuations (defined on all variables, the same below) which can satisfy $Pred_{s,t}$, we know $\{\rho_y\}^{y\in Y}$ is finite, and $Y \subseteq X$. Every times we change the predicate formula $Pred_{s,t}$ to $newPred_{s,t}$ during function *UpdatePredicate*, we know that:

$$newPred_{s,t} = Pred_{s,t} \wedge po$$

where $po$ is the proof obligation. And $Pred_{s,t}$ is being updated implies the fact that there exists some valuations $\rho_{y'}^{y'\in Y}$ which can't satisfy $po$. Thus, if we denote $\{\rho_z\}^{z\in Z}$ as the set of valuations which can satisfy proof obligation $po$, it's easy to see that the set of valuations which can satisfy $newPred_{s,t}$ equals to $\{\rho_y\}^{y\in Y} \cap \{\rho_z\}^{z\in Z}$, and it's definitely smaller than $\{\rho_y\}^{y\in Y}$. Thus we come to the conclusion that this kind of change must be finite, because this updating is a decrement within a finite field.

Moreover, for any Triple $(s, t|Pred_{s,t})$, only when there exists at least one next-Triple $(s_{next}, t_{next}|Pred_{next})$ of it has changed predicate, function *UpdatePredicate* will be called with inputting this Triple $(s, t|Pred_{s,t})$.

Due to the fact that next-Triples of any Triple are finite, and the changes of predicates are also finite, we know that there are always finitely many calls to function *UpdatePredicate* with any Triple. Thus, function *GenerateWeakestStrFH* will definitely terminate, and obviously function *CheckStrFH-Bisimilar* will also terminate. □

### 5.5 Running Example

In [13], Appendix D.3, we also provide the open automata in Fig.2 as a running example for our algorithm, to illustrate it's procedure.

## 6 Related Work

To the best of our knowledge, there is no other research works on the Bisimulation Equivalence between such complicated (open, symbolic, parameterized, with loops and assignments) system models, with providing fully complete algorithms. For the sake of completeness, we give a brief overview of other Symbolic Bisimulation researches.

One line of research is providing Symbolic Bisimulation for different model or language. In Calder's work [5], they define a symbolic semantic for full LOTOS, with a symbolic bisimulation over it; Borgstrom et al., Delaune et al. and Buscemi et al. provide symbolic semantic and equivalence for different variants of pi calculus respectively [3, 4, 7], and later in 2012 Liu's work provided symbolic bisimulation for full applied pi calculus [16]. The most recent work, Feng et al. provide a symbolic bisimulation for quantum processes [10]. All the above works are based on models quite different from ours, and most of them have no available implementation.

Another line of research is devising algorithms for computing symbolic Bisimulation Equivalence. In Dovier's work, they provide a rank-based algorithm, layering the input model to compute bisimulation [8]. Baldan et al. also focus on open systems, using a logical programming language *Prolog* to model the systems and compute Bisimulation [1]. Wimmer et al. present a signature-based approach to compute Bisimulation, implemented by using BDDs [18]. Lin [15] presents a symbolic bisimulation between symbolic transition graph with assignments (STGA); as mentioned in the *Introduction*, this work brought us lots of inspiration, but they had a strong "data-independence" constraint that our approach significantly overcomes.

Going further than computing symbolic bisimulation, there are several works on devising approach to minimize the system by symbolic bisimulation. The well-known partition refinement algorithm, which first devised by Paige et al. [17], is unsuitable for symbolic models. Thus Bonchi et al. devised a *symbolic partition refinement* algorithm to compute bisimilarity and redundancy at same time [2]; D'Antoni et al. provided a Forward Bisimulation for minimizing the nondeterministic symbolic finite automata [9]. The above approaches, due to the difference of system models they use, are not applicable to our issues, but we are still inspired a lot from these related work.

## 7 Conclusion

Based on previous research, we have proposed StrFH-Bisimulation, a new structural Bisimulation equivalence between *open automata* which are symbolic, open and parameterized system models and can address all the finitely represented systems computed from pNets. By proving that StrFH-Bisimulation correctly and completely corresponds to FH-Bisimulation, we generalize the interesting properties of FH-Bisimulation to StrFH-Bisimulation.

After that, we proposed two algorithms for checking and computing StrFH-Bisimulation: the first one requires a (user-defined) relation and two finite open automata, to check whether the given relation is a StrFH-Bisimulation between the given automata; the second one go further, for any two given open automata, it computes the *weakest StrFH-Bisimulation* $\mathcal{R}$ between them, which means all the generated conditions in $\mathcal{R}$ are the (logicaly) weakest. We provided a nontrivial proof to show the "weakestness" of our algorithm, and show that this algorithm terminates when the data domains are finite.

We have started implementing these algorithms, and need to evaluate their practical performances on realistic use cases. There are also several interesting directions to further develop our work. One idea is about computing weakest StrFH-Bisimulation algorithm. Our preliminary termination conditional is certainly not the best we can get. While it is clear that we cannot get unconditional termination, it would be interesting to find a more liberal condition or to improve the algorithm to handle more practical cases.

Second idea is to develop a minimization algorithm from our StrFH-Bisimulation. Minimizing system models is the most common way to use a bisimulation equivalence, especially in the case of hierarchical models like pNets, where minimization can be used in a compositional way.

Noting that we only developed the theory of symbolic bisimulation between open automata for *Strong Bisimulation*. Devising a symbolic bisimulation for *Weak Bisimulaiton*, which takes invisible or internal moves into account, would also be interesting, and some of our colleagues are already working on that.

## Acknowledgments

This work is partially supported by the National Key Research and Development Project 2017YFB1001800, and the National Natural Science Foundation of China (61972150, 61572195, 61672229, 61832015).

## References

[1] Paolo Baldan, Andrea Corradini, Hartmut Ehrig, and Reiko Heckel. 2001. Compositional modeling of reactive systems using open nets. In *International Conference on Concurrency Theory*. Springer, 502–518.

[2] Filippo Bonchi and Ugo Montanari. 2009. Minimization algorithm for symbolic bisimilarity. In *European Symposium on Programming*. Springer, 267–284.

[3] Johannes Borgström, Sébastien Briais, and Uwe Nestmann. 2004. Symbolic bisimulation in the spi calculus. In *International Conference on Concurrency Theory*. Springer, 161–176.

[4] Maria Grazia Buscemi and Ugo Montanari. 2008. Open bisimulation for the concurrent constraint pi-calculus. In *European Symposium on Programming*. Springer, 254–268.

[5] Muffy Calder and Carron Shankland. 2001. A symbolic semantics and bisimulation for full LOTOS. In *International Conference on Formal Techniques for Networked and Distributed Systems*. Springer, 185–200.

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[7] Stéphanie Delaune, Steve Kremer, and Mark Ryan. 2007. Symbolic bisimulation for the applied pi calculus. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 133–145.

[8] Agostino Dovier, Raffaella Gentilini, Carla Piazza, and Alberto Policriti. 2002. Rank-based symbolic bisimulation:(and model checking). *Electronic Notes in Theoretical Computer Science* 67 (2002), 166–183.

[9] Loris DâĂŹAntoni and Margus Veanes. 2017. Forward bisimulations for nondeterministic symbolic finite automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 518–534.

[10] Yuan Feng, Yuxin Deng, and Mingsheng Ying. 2014. Symbolic bisimulation for quantum processes. *ACM Transactions on Computational Logic (TOCL)* 15, 2 (2014), 14.

[11] Ludovic Henrio, Eric Madelaine, and Min Zhang. 2015. pnets: An expressive model for parameterised networks of processes. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 492–496.

[12] Ludovic Henrio, Eric Madelaine, and Min Zhang. 2016. A Theory for the Composition of Concurrent Processes. In *36th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE) (Formal Techniques for Distributed Objects, Components, and Systems)*, Elvira Albert and Ivan Lanese (Eds.), Vol. LNCS-9688. Heraklion, Greece, 175–194. https://hal.inria.fr/hal-01432917

[13] Zechen Hou, Eric Madelaine, and Jing Liu. 2019. *Symbolic Bisimulation for Open and Parameterized Systems - Extended version.* Research Report RR-9304. UCA - Inria Sophia Antipolis Mediterannee. 47 pages. https://hal.inria.fr/hal-02376147

[14] Huimin Lin. 1996. Symbolic transition graphs with assignment. In *International Conference on Concurrency Theory*. Springer, 50–65.

[15] Huimin Lin. 1998. âĂIJOn-the-fly instantiationâĂİ of value-passing processes. In *Formal Description Techniques and Protocol Specification, Testing and Verification*. Springer, 215–230.

[16] Jia Liu and Huimin Lin. 2010. A complete symbolic bisimulation for full applied pi calculus. In *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 552–563.

[17] Robert Paige and Robert E Tarjan. 1987. Three partition refinement algorithms. *SIAM J. Comput.* 16, 6 (1987), 973–989.

[18] Ralf Wimmer, Marc Herbstritt, Holger Hermanns, Kelley Strampp, and Bernd Becker. 2006. Sigref–a symbolic bisimulation tool box. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 477–492.