Effective Marking Equivalence Checking in Systems with Dynamic Process Creation[*]

Łukasz Fronc

IBISC, Université d'Evry-Val d'Essonne
23 boulevard de France,
91037 Évry, France
`lfronc@ibisc.univ-evry.fr`

The starting point of this work is a framework allowing to model systems with dynamic process creation, equipped with a procedure to detect symmetric executions (*i.e.*, which differ only by the identities of processes). This allows to reduce the state space, potentially to an exponentially smaller size, and, because process identifiers are never reused, this also allows to reduce to finite size some infinite state spaces. However, in this approach, the procedure to detect symmetries does not allow for computationally efficient algorithms, mainly because each newly computed state has to be compared with every already reached state.

In this paper, we propose a new approach to detect symmetries in this framework that will solve this problem, thus enabling for efficient algorithms. We formalise a canonical representation of states and identify a sufficient condition on the analysed model that guarantees that every symmetry can be detected. For the models that do not fall into this category, our approach is still correct but does not guarantee a maximal reduction of state space.

1 Introduction

The problem of detecting symmetries during the construction of state spaces has been widely explored [9]. Given a formalism and its operational semantics, the state space is built starting from an initial state. We repeat the computation of new states for every discovered state and we aggregate them in order to reach a fixed point, which is the set of all reachable states. To add abstraction to our formalism we can define equivalence classes of states, two states in the same class are said to be symmetric. Each time a state symmetric to an already visited one is found, it is considered as already known, also it means that if the abstraction was well defined, then the behavior resulting from further exploration of this state leads to an already analysed behaviour and thus may be omitted. The resulting state space corresponds to the quotient graph of equivalence classes of states (symmetric states) and can be exponentially smaller than the original graph [2], or more drastically infinite state spaces can be reduced to finite ones if the number of equivalence classes is finite. Reductions by symmetries have been implemented in a variety of tools and proved to be a successful technique [7, 14, 1].

Here we focus on symmetries induced by dynamic process creation in multi-threaded computing systems. The goal of formalisms using this paradigm is to allow processes to be created at runtime. This ability contributes to the size explosion of state spaces, in particular the order of created processes, despite being irrelevant, introduces large amounts of interleaving. Moreover processes may be created endlessly which would lead to infinite state spaces if states cannot be

---

identified as equivalent. In the formalism we focus on, process names are irrelevant, it is the relations between them that matter and not their identities. For example, let us consider an `http` server, each time it receives a request for the home page of a website, it creates a new thread which will handle the request. Now suppose that the request has been handled but the server receives another one for the same home page, then it creates a *new* thread which will handle the new request. These two threads have different identities however they answer the same kind of requests in same conditions, the resulting behaviour will be the same modulo thread identifiers, they are clearly symmetric. This example well illustrates the nature of symmetries we want to detect.

The current work is based on the approach developed in [10, 11, 6] which presents a method to detect symmetries introduced by process identifiers. The approach in these works reduces the detection of symmetries to computation of graph isomorphisms. Each newly discovered state is transformed into a graph, if this graph is isomorphic to any graph corresponding to a visited state then the two states are symmetric. The main problem with this approach is that each time we discover a state, we have to look for an isomorphism with all the visited states.

In this paper we present an effective way to detect symmetries without computing a graph isomorphism. It provides representations for markings that allow direct comparison of states, with a hash table for example, instead of requiring a comparison with all visited states. Let us consider the following state space exploration algorithm, similar algorithms can be found in [4, 5]. We compute the state space of a model starting with an initial state in a set *todo*. For each state s in set *todo*, we check if the state has already been visited using a function VISITED. If the state was not visited yet then we add it to a set *done* and add all of its successor states (by calling for example a function SUCCS(s)) to set *todo*. At the end of execution the set of reachable states is the set *done*. Symmetry detection can be performed in function VISITED. If we use the graph isomorphism approach, we have to check for an isomorphism with all states in set *done*, whereas using canonical representation we will just test if the state *is* in set *done*. The former approach is more expensive due to the involved loop and the isomorphism computation even if this operation may be fast with distinct graphs [13, 12]. With the latter we can perform the test in constant time using a hash table for example, indeed the representation being canonical we can build a hash function for table lookup and then use a few comparisons.

The method presented here does not detect all symmetries whereas the graph isomorphism method does. However, we trust that we detect a large amount of symmetries in systems where completeness is not achieved and the trade-off between completeness and efficiency is interesting. Moreover, we provide a sufficient condition for maximal reductions, *i.e.*, for completeness.

This paper is structured as follows: first we introduce process identifiers and recall the definition of the Petri net models we use, next we introduce the data structure we will use to represent markings, then we present the theoretical basis for symmetry detection with our representation, and finally we discuss canonisation of the representation and a sufficient condition to achieve completeness. Due to space limitation, proofs have been omitted but a version of this paper including the proofs is available as a technical report [3].

2   Process identifiers and Petri nets

2.1   Process identifiers

Addressing systems with multiple threads or multiple processes is a tedious task. When the number of processes is fixed, each process can be represented as a subnet in the model. This

approach is limited, indeed the number of processes is fixed and thus no new processes can be created. In real life applications we create processes at runtime and so we need to reason about this dynamic process creation and the resulting behaviour.

The problem of reasoning about this kind of systems has lead to the development of different formalisms and methods [15]. Here, we focus on an implementation in terms of Petri nets [8] which allows for dynamic process creation and is mainly based on [6]. First we refine the definition of process identifiers that will be used through this paper, the definition is compatible with the one from [6]. The only difference is the introduction of an empty pid that helps the formal definition of pids. Next, we also add some useful operations to handle pids.

Definition 2.1 (process identifiers) *Process identifiers (pids) are elements of $\mathbb{P} \stackrel{\mathrm{df}}{=} (\mathbb{N}^+)^\star$, the set of tuples over non-zero natural numbers. $\mathbb{P}$ equipped with the tuples concatenation operation and its identity element, the empty tuple $\langle \rangle$, is a monoid. We denote $a_1.a_2.\cdots.a_n$ the process identifier $\langle a_1, a_2, \ldots, a_n \rangle$.*

Definition 2.2 *Let π be a pid. We define the length, the prefix and the set of subpids of π as:*

- *$length(\pi) = n$ if $\exists n > 0$ such that $\pi = \langle a_1, \ldots, a_n \rangle$ and 0 otherwise;*

- *$prefix(\pi) = \langle a_1, \ldots, a_{n-1} \rangle$ if $\exists n > 0$ such that $\pi = \langle a_1, \ldots, a_n \rangle$ and $\langle \rangle$ otherwise;*

- *$subpid(\pi) = \{\pi\} \cup subpid(prefix(\pi))$ if $length(\pi) > 0$ and $\emptyset$ otherwise.*

Now as in [6] we define the operations that can be used to compare process identifiers in a model. The following operations are the *only ones* that can be used in the model to compare pids, any other operation is forbidden.

Definition 2.3 (operations on process identifiers) *Two pids $\pi, \pi' \in \mathbb{P}$ can be compared using the equality and the following operations:*

- *$\pi \lhd_1 \pi'$ iff $\exists a \in \mathbb{N}^+$ such that $\pi.a = \pi'$;*

- *$\pi \lhd \pi'$ iff $\exists a_1, \ldots, a_n \in \mathbb{N}^+$ such that $\pi.a_1.\cdots.a_n = \pi'$;*

- *$\pi \Cap_1 \pi'$ iff $\exists \pi'' \in \mathbb{P}, \exists i \in \mathbb{N}^+$ such that $\pi'' \neq \langle \rangle,\ \pi = \pi''.i$ and $\pi' = \pi''.(i+1)$;*

- *$\pi \Cap \pi'$ iff $\exists \pi'' \in \mathbb{P}, \exists i, j \in \mathbb{N}^+$ such that $\pi'' \neq \langle \rangle,\ \pi = \pi''.i,\ \pi' = \pi''.j$ and $i < j$.*

Intuitively, $\pi \lhd_1 \pi'$ means that π is the parent of π', $\pi \lhd \pi'$ means that π is an ancestor of π', $\pi \Cap \pi'$ means that π is a younger sibling of π', *i.e.*, was spawned before π and have the same parent as π, finally $\pi \Cap_1 \pi'$ means that π is the younger sibling of π' spawned just before π'.

The last refinement on pids is the introduction of a total order between pids. This order will be important when addressing the detection of symmetries and canonical forms.

Definition 2.4 (ordering) *The set of all pids $\mathbb{P}$ is totally ordered with a hierarchical order, i.e., pids are ordered by length and if they have the same length, the lexicographic order on tuples is used.*

2.2 Coloured Petri nets

The models we address are (coloured) Petri nets which allow dynamic process creation. A formal definition was given in [10, 11] then refined in [6], we base ourselves on the refined one. Here, we recall the definition and some requirements. Let $\mathbb{V}$ be a set of variables, $\mathbb{D}$ a set of data values

and $\mathbb{E}$ a set of expressions such that $\mathbb{V} \cup \mathbb{D} \subseteq \mathbb{E}$. We assume that $\mathbb{E}$ contains Boolean expressions. A binding is a partial function $\beta : \mathbb{V} \to \mathbb{P} \cup \mathbb{D}$. The application of a binding β is extended to denote $\beta(e)$, the evaluation of an expression e under β. The evaluation of an expression under a binding is naturally extended to sets and multisets of expressions. Data values, variables, syntax for expressions, possibly typing rules etc. denote the colour domain of a Petri net.

Definition 2.5 (Petri nets) *A Petri net is a tuple (S, T, ℓ) where S is a finite set of places, T, disjoint from S, is a finite set of transitions, and ℓ is a labelling function such that:*

- *for all $s \in S$, $\ell(s)$ is the type of s, a Cartesian product $X_1 \times \cdots \times X_k$ ($k \geq 1$), where each X_i is $\mathbb{P}$ or $\mathbb{D}$, this type denotes the values that s may contain;*

- *for all $t \in T$, $\ell(t) \in \mathbb{E}$ is the guard of t, i.e., a condition for its execution;*

- *for all $(x, y) \in (S \times T) \cup (T \times S)$, $\ell(x, y)$ is a multiset over $\mathbb{E}$ and defines the arc from x to y.*

A marking *of a Petri net is a map that associates to each place $s \in S$ a multiset of values from $\ell(s)$. We denote by* **Mrk** *the set of all markings. From a marking M, a transition t can be fired using a binding β and yield a new marking M', which is denoted by $M[t, \beta\rangle M'$, iff:*

- *there are enough tokens: for all $s \in S$, $M(s) \geq \beta(\ell(s, t))$;*

- *the guard is satisfied: $\beta(\ell(t))$ is true;*

- *place types are respected: for all $s \in S$, $\beta(\ell(t, s))$ is a multiset over $\ell(s)$;*

- *M' is M with tokens consumed and produced according to the arcs: for all $s \in S, M'(s) = M(s) - \beta(\ell(s, t)) + \beta(\ell(t, s))$.*

The following requirements are adapted from [6] which generalised [10, 11] in two ways: first by using transitions systems in general instead of Petri nets in particular; second, by relaxing the constraints. We have preferred to stay with Petri nets to extend our previous works [5, 4]. A Petri N net respecting all the following requirements is called a *thread Petri net* (or *t-net*).

1. The set of places of N contains a unique generator place s_η having type $\mathbb{P} \times \mathbb{N}$. The generator place stores tokens $\langle \pi, i \rangle$ where i is the counter of child threads already spawned by π. Thus the next threads created by π will have pids $\pi.(i+1)$, $\pi.(i+2)$, etc. We say that π is *generative* at a marking M if there is a $n \in \mathbb{N}$ such that $\langle \pi, n \rangle \in M(s_\eta)$.

2. We assume that the initial marking M_0 of N is such that the generator place contains exactly one token, $\langle \langle 1 \rangle, 0 \rangle$, and all the other places are empty or contain data values.

3. For each transition $t \in T$, the annotation on the arc from the generator place to t is a set of the form $\ell(s_\eta, t) \stackrel{\text{df}}{=} \{\langle p_1, c_1 \rangle, \ldots, \langle p_k, c_k \rangle\}$ where $k \geq 0$ and all p_i's and c_i's are distinct pid and counter variables. The annotation on the arc from t to the generator place is a set:

$$\ell(t, s_\eta) \stackrel{\text{df}}{=} \left\{ \begin{array}{l} \langle p_1, c_1 + n_1 \rangle, \ldots, \langle p_m, c_m + n_m \rangle, \\ \langle p_1.(c_1 + 1), 0 \rangle, \ldots, \langle p_1.(c_1 + n_1), 0 \rangle, \\ \ldots \\ \langle p_k.(c_k + 1), 0 \rangle, \ldots, \langle p_k.(c_k + n_k), 0 \rangle \end{array} \right\}$$

where $m \leq k$, and $n_j \geq 0$ for all j. An empty arc annotation means that the arc is absent. Below we denote by Π_t the set of all newly created pids $p_i.(c_i + j)$ used in $\ell(t, s_\eta)$.

4. For each transition $t \in T$ and each place $s \in S \setminus \{s_\eta\}$, the annotation on the arc from s to t is a multiset of vectors built from variables and data values, and the annotation on the arc from t to s is a multiset of vectors built from expressions involving data variables and data values as well as elements from $\Pi_t \cup \{p_1, \ldots, p_m\}$.

5. For each transition $t \in T$, $\ell(t)$ is a computable Boolean expression, built from the variables occurring in the annotations of arcs adjacent to t and data values. The usage of pids is restricted to comparisons of the elements from $\Pi_t \cup \{p_1, \ldots, p_k\}$ using the operators from $\{=, \lhd_1, \lhd, \pitchfork_1, \pitchfork\}$

For each reachable marking M we define the corresponding *state* $q_M \stackrel{\text{df}}{=} (\sigma_M, \eta_M)$ given by $\sigma_M \stackrel{\text{df}}{=} \{(s, M(s)) \mid s \in S \setminus \{s_\eta\}\}$ and $\eta_M \stackrel{\text{df}}{=} \{\pi \mapsto k \mid \langle \pi, k \rangle \in M(s_\eta)\}$. Given a state q_M we define the following notions:

- for each generative pid π in q_M, *i.e.*, $\pi \in dom(\eta_M)$, the next pid to be created is given by $next_{q_M}(\pi) \stackrel{\text{df}}{=} \pi.(\eta_M(\pi) + 1)$;

- the next-pids of q_M are $nextpid_{q_M} \stackrel{\text{df}}{=} \{next_{q_M}(\pi) \mid \pi \in dom(\eta_M)\}$;

- pid_{q_M} is the set of all pids involved in σ_M, *i.e.*, pids from s_η and all data places.

Definition 2.6 (state equivalence) *Two states q_M and $q_{M'}$ are equivalent if there is a bijection $h : (pid_{q_M} \cup nextpid_{q_M}) \to (pid_{q_{M'}} \cup nextpid_{q_{M'}})$ such that for all relations $\prec \in \{\lhd_1, \lhd\}$ and $\curlywedge \in \{\pitchfork_1, \pitchfork\}$:*

1. $h(dom(\eta_{q_M})) = dom(\eta_{q_{M'}})$;

2. $\forall \pi \in dom(\eta_{q_M}), h(next_{q_M}(\pi)) = next_{q_{M'}}(h(\pi))$;

3. $\forall \pi, \pi' \in pid_{q_M} : \pi \prec \pi'$ iff $h(\pi) \prec h(\pi')$;

4. $\forall \pi, \pi' \in pid_{q_M} \cup nextpid_{q_M} : \pi \curlywedge \pi'$ iff $h(\pi) \curlywedge h(\pi')$;

5. $\sigma_{q_{M'}}$ is σ_{q_M} after replacing each pid π by $h(\pi)$.

We denote this by $q_M \sim_h q_{M'}$, or simply $q_M \sim q_{M'}$.

Two reachable markings M and M' of a Petri net are equivalent if and only if their respective states q_M and $q_{M'}$ are equivalent. This is denoted by $M \sim_h M'$ or simply $M \sim M'$.

This equivalence relation ensures that markings contain the same data tokens and pids are related through h. State (or context) equivalence guarantees that h preserves the relations between pids for $\lhd_1$, $\lhd$, $\pitchfork_1$ and $\pitchfork$. So if two markings are equivalent then they differ in pids but these pids have the same relations among them and thus differ only in names; names being not relevant the states can be assimilated.

Theorem 2.1 *Let M and M' be h-equivalent reachable markings of a t-net N, and t be a transition such that $M[t, \beta\rangle \tilde{M}$. Then $M'[t, h \circ \beta\rangle \widetilde{M'}$, where $\widetilde{M'}$ is a marking such that $\tilde{M} \sim_{\tilde{h}} \widetilde{M'}$ for a bijection $\tilde{h}$ coinciding with h on the intersection of their domains.*

As stated in [10, 11, 6] the above result captures a truly strong notion of marking similarity. So, if two markings are equivalent their futures are the same modulo pid renaming. To compute state equivalence, the approach proposed in [10, 11, 6] was first to build a graph then compute graph isomorphisms. Computing these isomorphisms is an expensive step and was hardly conceivable in a tool since we have to compute it for all visited states each time a new state is discovered. The contribution of this paper is to provide a representation for markings to detect this equivalence. This is presented in following sections but before moving on we add some notations for tokens.

Definition 2.7 (owned, shared, active, and referenced pids) *Let $v = \langle x_1, \ldots, x_n \rangle$ be a token. If $x_1 \in \mathbb{P}$ then x_1 is the* owner *of v, otherwise if $x_1 \in \mathbb{D}$ we say that v is* shared*. We denote by $pid(v)$ the owner of v if defined. If a pid π appears in a token v at a marking M then we say that π is* active *at M. Finally we define the set of* referenced *pids of v as the set of all pids appearing in v except $pid(v)$.*

3   Pid-trees

A pid-tree is a representation of a marking where tokens are associated to process identifiers. The intention is to classify them by ownership. The root of the tree will contain data tokens that are shared between different processes and each of the remaining nodes will contain tokens that belong to a particular process. To identify these processes each child node is prefixed with a fragment of pid. Each path in the tree leads to a node that contains a marking where all tokens belong to the process which pid is the concatenation of all the fragment pids along the path; we say that this pid labels the path. A path can be seen as a concatenation of pids that lead to a, possibly empty, marking. A formal definition of paths is given later in this section and ownership constraints will be added when exposing the construction of the tree. Pid-trees are formally defined by the following definition.

Definition 3.1 (pid-tree) *The set Ξ of* pid-trees *is defined by $\langle M, C \rangle \in \Xi$ if $M \in Mrk$ and $\exists n \in \mathbb{N}$ such that $C = \langle \langle a_1, t_1 \rangle, \ldots, \langle a_n, t_n \rangle \rangle \in (\mathbb{P} \times \Xi)^n$ and C satisfies:*

- $\forall i \in \{1, \ldots, n\}$, $a_i \neq \langle \rangle$;
- $\forall i, j \in \{1, \ldots, n\}$ *if $i \neq j$ then $a_i \neq a_j$, $a_i \not\subseteq subpid(a_j)$ and $a_j \not\subseteq subpid(a_i)$.*

We denote $M \xrightarrow{a_1, \ldots, a_n} \langle t_1, \ldots, t_n \rangle$ the pid-tree $\langle M, \langle \langle a_1, t_1 \rangle, \ldots, \langle a_n, t_n \rangle \rangle \rangle$. We will also use the notation M_t to denote the marking M in $t = \langle M, C \rangle$.

Having tuples in the definition of children nodes allows to perform a syntactical ordering of them. The conditions in the second point of definition 3.1 ensure that each pid associated with a node is unique, but more than that, it ensures that we cannot have two children prefixed by the same pid fragment. For example $\emptyset \xrightarrow{1,1} \langle t_1, t_2 \rangle$ is an illegal pid-tree as well as $\emptyset \xrightarrow{1,1.2} \langle t_1, t_2 \rangle$ since $1 \in subpid(1.2)$. A pid-tree can be represented graphically, see Figure 1, each node is labelled with a marking and each arc with a pid fragment. In Figure 1(a), the process of pid 1 does not own any tokens so the corresponding node is associated to an empty marking, but the process of pid 1.2.1.3 owns some tokens so the corresponding node is associated with marking M_3.

To check the inclusion of pid-trees we have to start from the root, first we check the inclusion of markings and then we check the inclusion of children that must be labelled with the same pid fragments. It is important to notice that the inclusion operation on pid-trees preserves the root, respects marking inclusion and does not denote a subtree relation. The formal definition follows.

Definition 3.2 (inclusion) *Let $t = \langle M, \langle \langle a_1, t_1 \rangle, \ldots, \langle a_n, t_n \rangle \rangle \rangle \in \Xi$ be a pid-tree. A pid-tree $t' = \langle M', \langle \langle a'_1, t'_1 \rangle, \ldots, \langle a'_m, t'_m \rangle \rangle \rangle \in \Xi$, with $m \leq n$, is included in t, noted $t' \subseteq t$, if $M' \leq M$ and for all $i \in \{1, \ldots, m\}$, there is a $j \in \{1, \ldots, n\}$ such that $a'_i = a_j$ and $t'_i \subseteq t_j$.*

For example, the pid-trees in Figure 1(b) and 1(c) are included in the pid-tree in Figure 1(a) if $M'_1 \leq M_1$ and $M'_4 \leq M_4$. The pid-tree in Figure 1(d) is not included is any other pid-trees in

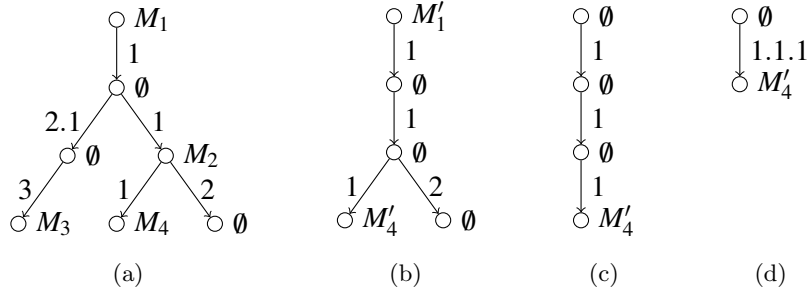(a)   (b)   (c)   (d)

Figure 1: The pid-tree $M_1 \xrightarrow{1} \langle \emptyset \xrightarrow{2.1,1} \langle \emptyset \xrightarrow{3} M_3, M_2 \xrightarrow{1,2} \langle M_4, \emptyset \rangle \rangle \rangle$ is shown in Figure 1(a); the pid-tree $M_1 \xrightarrow{1} \emptyset \xrightarrow{1} \emptyset \xrightarrow{1,2} \langle M'_4, \emptyset \rangle$ is shown in Figure 1(b); Figures 1(c) and 1(d) show two pid-trees that represents $path(1.1.1, M'_4)$ and are respectively $\emptyset \xrightarrow{1} \emptyset \xrightarrow{1} \emptyset \xrightarrow{1} M'_4$ and $\emptyset \xrightarrow{1.1.1} M'_4$. We assume that markings M_i and M'_i are not empty.

Figure 1. The pid-tree $\emptyset \xrightarrow{1} \emptyset$ is included in pid-trees in Figures 1(a), 1(b) and 1(c) whereas $\emptyset \xrightarrow{2} \emptyset$ is not included in any of pid-trees in Figure 1, which is straightforward to understand if we remember that we have to preserve the root.

The definition of subtrees is similar to the usual definition on trees, however we add a localisation hint for the subtree in order to differentiate two subtrees that are structurally equal but at different positions in the tree. Thus a subtree will be a pair formed of a pid and a pid-tree.

Definition 3.3 (subtrees) *Let $t_0 = M \xrightarrow{a_1,\ldots,a_n} \langle t_1, \ldots, t_n \rangle \in \Xi$ be a pid-tree. A pid-tree t is a subtree of t_0 at π, noted $\langle \pi, t \rangle \in Trees(t_0)$, if:*

- *$\pi = \langle \rangle$ and $t = t_0$; or*

- *$\exists a_i \in \{a_1, \ldots, a_n\}$ such that $\pi = a_i.\pi'$ and $\langle \pi', t \rangle \in Trees(t_i)$.*

We denote $Trees(t)$ the set of all subtrees of t.

Using the definition of subtrees, we can retrieve the set of all pids in a pid-tree which corresponds to all the localisations in the pid-tree.

Definition 3.4 *Let $t \in \Xi$ be a pid-tree. We define $pid(t) = \{ \pi \mid \langle \pi, t' \rangle \in Trees(t) \}$.*

Now that we have set up some basic blocks to handle pid-trees, we give a formal definition of paths. A path is a *linear* pid-tree with all markings empty except for the leaf one.

Definition 3.5 (paths) *Let $\pi \in \mathbb{P}$ be a pid, and $M \in Mrk$ a marking. A* path *labeled by π and decorated by M is a pid-tree t such that $\{\pi\} \subseteq pid(t) \subseteq subpid(\pi)$, $\langle \pi, \langle M, \langle \rangle \rangle \rangle \in Trees(t)$ and all other markings in t are empty. We denote $path(\pi)$ the path $path(\pi, \emptyset)$ and $paths(T)$ the set of all paths in a pid-tree T (with respect to $\subseteq$).*

As the definition suggests, there may be different pid-tree representations of a path, indeed these representations will differ in the length of the path and the pid fragments on arcs. For example the pid-trees on Figures 1(c) and 1(d) represents the same path $path(1.1.1, M'_4)$.

A path denotes an ownership relation between a process and some tokens. Intuitively $path(\pi, M)$ represents the fact that all tokens that appear in M belong to the process of pid π. We say that π labels this path. Moreover saying that a subtree t is at π means that the path $path(\pi)$ leads to t.

When addressing process identifiers, their names are irrelevant, so we introduce an opaque description for localisations in pid-trees. We could use pids to locate nodes but we need to abstract them so we use relative paths which describe the localisation of a subtree independently of the labels appearing on the edges. A relative path can be seen as a normalized form of a pid based on its position inside the tree.

Definition 3.6 (relative paths) *Let $t \in \Xi$ be a pid-tree. A relative* path $\widetilde{path}(\pi, t)$ *of* $\pi \in pid(t)$ *is the tuple defined as:*

- $\langle i \rangle$ *if* $\pi = a_i$ *and* $t = M \xrightarrow{a_1,\dots,a_n} \langle t_1,\dots,t_n \rangle$ *where* $1 \le i \le n$;
- $\langle i, i_1, \dots, i_m \rangle$ *if* $\pi = a_i.\pi'$, $t = M \xrightarrow{a_1,\dots,a_n} \langle t_1,\dots,t_n \rangle$ *and* $\langle i_1,\dots,i_m \rangle = \widetilde{path}(\pi', t_i)$ *where* $1 \le i \le n$.

Finally, since we have the ability to consider the ordering of children of a node, we introduce labelling functions that will be used to compare and order these children.

Definition 3.7 *Let $(E, \le)$ be a partially ordered set. A labelling function ℓ is a partial function from $\mathbb{P} \times \Xi$ to E.*

Definition 3.8 *Let $t \in \Xi$ be a pid-tree and ℓ a labelling function. The pid-tree t is ordered with respect to ℓ if:*

- $t = \langle M, \langle \rangle \rangle$ *or*
- $t = M \xrightarrow{a_1,\dots,a_n} \langle t_1,\dots,t_n \rangle$, $\ell(a_1, t_1) \le \dots \le \ell(a_n, t_n)$ *and* $t_1,\dots,t_n$ *are ordered with respect to ℓ.*

4   Checking marking equivalence

To apply theorem 2.1 and reduce state spaces we will need to detect marking equivalence. We check marking equivalence by mapping markings to pid-trees and comparing these trees. If the trees are equivalent, markings will be isomorphic. However the representation we use is tightly linked to the relations used to compare pids in the models. More precisely the representation we use embeds the relations $\lessdot_1$, $\lessdot$, $\Cap_1$ and $\Cap$.

Intuitively a pid-tree embeds the relation $\lessdot$ by construction because it is a hierarchy of pids. In this section we need to include the relation $\Cap$ into pid-trees as well as $\lessdot_1$ and $\Cap_1$. To do so, we will use a specific labelling and ordering on child nodes.

Definition 4.1 (sibling ordering) *Let $t \in \Xi$ be a pid-tree. The pid-tree t is sibling ordered if it is ordered with the labelling function $\ell : \mathbb{P} \times \Xi \to \mathbb{P}$ defined as:*

$$\ell(a_i, t') = a_i$$

Where $\mathbb{P}$ is equipped with the hierarchical order on pids.

Using the sibling ordering we can define the representation of markings that will be used to detect symmetries. The representation is described in terms of three rules that have to be obeyed when building the pid-tree. The resulting pid-tree is not unique, as shown in Figure 2, but the definition gives all the theoretical requirements to guarantee the marking equivalence.

Definition 4.2 (pid-tree representation of markings) *Let M be a reachable marking of a t-net N. We build a representation of M as a sibling ordered pid-tree and $repr(M)$ denotes the set of all such representations. $R(M) \in repr(M)$ if for each place $s \in N$ which type is $X_1 \times \dots \times X_n$, and for each token $v = \langle x_1,\dots,x_n \rangle \in M(s)$, for $X_i \in \{\mathbb{D}, \mathbb{P}\}$, we apply exactly one of the following rules:*

- **generator token rule:** *if $s = s_\eta$ and $v = \langle \pi, i \rangle$ then we have $path(\pi) \subseteq R(M)$ and $path(\pi.(i+1)) \subseteq R(M)$, i.e., $path(next_{q_M}(\pi)) \subseteq R(M)$.*

- **shared token rule:** *if $X_1 = \mathbb{D}$ then we have $path(\langle \rangle, \{(s,v)\}) \subseteq R(M)$, and for each x_i such that $X_i = \mathbb{P}$ we have $path(x_i) \subseteq R(M)$.*

- **owned token rule:** *if $X_1 = \mathbb{P}$ then we have $path(x_1, \{(s,v)\}) \subseteq R(M)$, and for each x_i such that $X_i = \mathbb{P}$ we have $path(x_i) \subseteq R(M)$.*
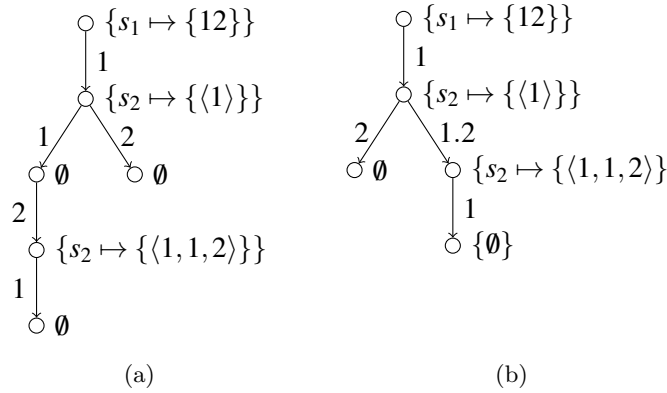

Figure 2: Two sibling ordered pid-trees representing the marking $M = \{s_1 \mapsto \{12\}, s_2 \mapsto \{\langle 1 \rangle, \langle 1,1,2 \rangle\}, s_\eta \mapsto \{\langle 1,1 \rangle, \langle \langle 1,1,2 \rangle, 0 \rangle\}\}$ with $\ell(s_1) = \mathbb{N}$, $\ell(s_2) = \mathbb{P}$ and $\ell(s_\eta) = \mathbb{P} \times \mathbb{N}$.

Checking the equality on sibling ordered pid-trees will not offer any advantages over comparing markings for symmetry detection. To capture symmetries we need to abstract pids because their values are irrelevant and only the relations between pids matter. This leads to the definition of an equivalence relation of sibling ordered pid-trees.

Definition 4.3 (sibling ordered pid-tree equivalence) *Let T and T' be two sibling ordered pid-trees, T and T' are equivalent if:*

1. *the function $h : pid(T) \to pid(T')$ such that $h = \{ (\pi, \pi') \mid \pi \in pid(T), \ \pi' \in pid(T') \ and \ \widetilde{path}(\pi, T) = \widetilde{path}(\pi', T') \}$ is a bijection*

2. *for each pair of subtrees $\langle \pi, t \rangle \in Trees(T)$ and $\langle \pi', t' \rangle \in Trees(T')$ such that $\pi' = h(\pi)$ and $t = M_t \xrightarrow{a_1,\dots,a_n} \langle t_1, \dots, t_n \rangle$, $t' = M_{t'} \xrightarrow{a'_1,\dots,a'_{n'}} \langle t'_1, \dots, t'_{n'} \rangle$ we have:*

   (a) *trees have the same structure, i.e., $n = n'$, which is implied by the definition of h.*

   (b) *for each place s different from s_η, $M_{t'}(s)$ can be obtained from $M_t(s)$ by replacing each pid π occurring in the tuples of $M_t(s)$ by $h(\pi)$;*

   (c) *for each a_i we have*
      - *$length(a_i) = length(a'_i) = 1$; or*
      - *$length(a_i) > 1$ and $length(a'_i) > 1$.*

   (d) *for each a_i and a_{i+1} such that $prefix(a_i) = prefix(a_{i+1})$ and $1 \leq i \leq n-1$ we have $prefix(a'_i) = prefix(a'_{i+1})$ and*
      - *$a_{i+1} - a_i = a'_{i+1} - a'_i = 1$; or*
      - *$a_{i+1} - a_i > 1$ and $a'_{i+1} - a'_i > 1$.*

We will denote this by $T \sim_h T'$ or simply by $T \sim T'$.

The function h in the definition above is a bijection if and only if two pid-trees have the same structure because pids are related through h if and only if their relative paths are equal. The second point states first that trees have the same structure, the markings only differ in pids and these pids are related through h, then checks condition on pid fragments. Length conditions ensure that $\lhd_1$ and $\lhd$ are preserved, indeed if the length of a fragment is equal to one then the parent and the child are in $\lhd_1$ otherwise the parent and the child are in $\lhd$. Offset conditions for fragments with same prefix ensure that $\Cap_1$ and $\Cap$ are preserved. If prefixes are equal then the children are siblings. If offsets are equal to one then the corresponding pids are in $\Cap_1$ and if offsets are greater than one the pids are in $\Cap$.

It has to be stressed that we cannot use $length(a_i) = length(a'_i)$ instead of $length(a_i) > 1$ and $length(a'_i) > 1$ because the the number of inactive pids between them and their parents does not matter as long as $\lhd$ is preserved. We qualify a pid as *inactive* if it does not appear in the marking (including the generator place) used to build the tree. The same remark applies for offsets and $\Cap$.

The equivalence is illustrated in Figure 3. Pid-trees in this figure are valid representations of markings, however we do not explain here how we choose the pid-tree we want among valid representations, this is discussed in section 5. Pid-trees T_1 (Figure 3(a)) and T_2 (Figure 3(b)) are equivalent if $h = \{1 \mapsto 1,\ 1.1 \mapsto 1.4,\ 1.3 \mapsto 1.7,\ 1.2.1 \mapsto 1.3.1,\ 1.2.2 \mapsto 1.3.2,\ 1.1.1 \mapsto 1.4.2\}$ satisfies marking, length and prefix conditions. Marking conditions are satisfied: $h(\langle 1 \rangle) = \langle 1 \rangle$ and $h(\langle 1.1,\ 1.2.1,\ 1.2.2 \rangle) = \langle 1.4,\ 1.3.1,\ 1.3.2 \rangle$. Prefix and length conditions are satisfied as well:

- we have $prefix(1) = prefix(3) = \langle \rangle$, $prefix(4) = prefix(7) = \langle \rangle$, we check offsets: $3 - 1 > 1$ and $7 - 4 > 1$.

- we have $prefix(2.1) = prefix(2.2) = \langle 2 \rangle$, $prefix(3.1) = prefix(3.2) = \langle 3 \rangle$, we check offsets: $2 - 1 = 2 - 1 = 1$.

Pid-trees T_3 (Figure 3(c)) and T_4 (Figure 3(d)) are not equivalent with others in Figure 3 despite they have the same structure, prefix conditions can be checked to convince ourselves.

Intuitively the length conditions guaranties that $\lhd_1$ is preserved, and the prefix conditions guaranties that the distances between pids with respect to relation $\Cap_1$ are correct and thus replaced pids have same relations with their siblings and ancestors.

Now we will see that we can define a restriction of h to $pid_{q_M} \cup nextpid_{q_M}$ that is also a bijection and that we can extract the original marking from the pid-tree representation. This is denoted by the following two propositions. First we bound the pids appearing in a representation, then we state that different restrictions are bijections as well.

Proposition 4.1 *Let M be a reachable marking of a t-net, q_M its state and $R(M)$ its representation. Then*

$$pid_{q_M} \cup nextpid_{q_M} \subseteq pid(R(M)) \subseteq subpid_{q_M} \cup nextpid_{q_M}$$

Where $subpid_{q_M} = \{subpid(\pi) \mid \pi \in pid_{q_M}\}$.

Proposition 4.2 *Let M and M' be two reachable markings of a t-net, $q_M, q_{M'}$ their respective states and $R(M), R(M')$ their representations. If $R(M) \sim_h R(M')$ then:*

1. *the restriction $h_1 : pid_{q_M} \to pid_{q_{M'}}$ of h is a bijection;*

2. *the restriction $h_2 : nextpid_{q_M} \to nextpid_{q_{M'}}$ of h is a bijection.*

3. *the restriction $h_3 : pid_{q_M} \cup nextpid_{q_M} \to pid_{q_{M'}} \cup nextpid_{q_{M'}}$ of h is a bijection.*

(a) Pid-tree T_1, a representation of $M_1 \cup M_2 \cup \{s_\eta \mapsto \{\langle 1,2 \rangle, \langle 1.1,0 \rangle\}\}$.

(b) Pid-tree T_2 representation of $M_1 \cup M_3 \cup \{s_\eta \mapsto \{\langle 1,6 \rangle, \langle 1.4,1 \rangle\}\}$.

(c) Pid-tree T_3 representation of $M_1 \cup M_4 \cup \{s_\eta \mapsto \{\langle 1,4 \rangle, \langle 1.4,1 \rangle\}\}$.

(d) Pid-tree T_4 representation of $M_1 \cup M_5 \cup \{s_\eta \mapsto \{\langle 1,5 \rangle, \langle 1.4,1 \rangle\}\}$.
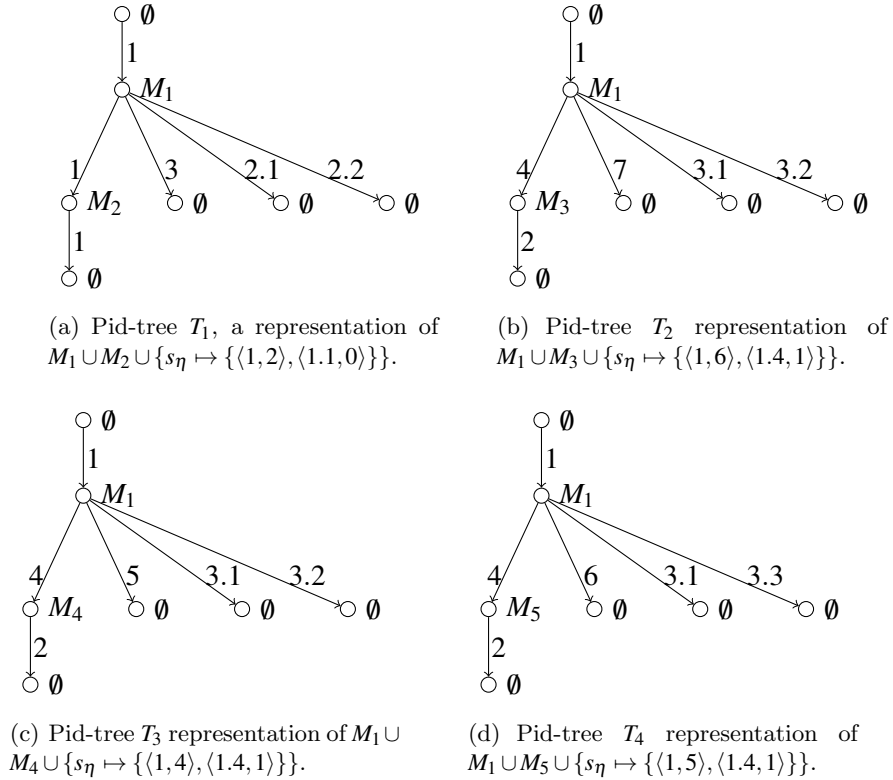
Figure 3: Four sibling ordered pid-trees where $M_1 = \{s_1 \mapsto \{\langle 1 \rangle\}\}$, $M_2 = \{s_2 \mapsto \{\langle 1.1, 2.1, 2.2 \rangle\}\}$, $M_3 = \{s_2 \mapsto \{\langle 1.4, 3.1, 3.2 \rangle\}\}$, $M_4 = \{s_2 \mapsto \{\langle 1.4, 3.1, 3.2 \rangle\}\}$ and $M_5 = \{s_2 \mapsto \{\langle 1.4, 3.1, 3.3 \rangle\}\}$.

The following proposition links all tokens in marking (except tokens from s_η) with tokens in the tree.

Proposition 4.3 *Let M be a reachable marking of a t-net N. Then for each place s in N such that $s \neq s_\eta$ we have:*

$$M(s) = \left(\bigcup_{\langle \pi, t \rangle \in Trees(R(M))} M_t(s) \right)$$

The next theorem states that two equivalent representations of markings imply an equivalence of markings. It leads to potential reduction in state spaces due to symmetries considering theorem 2.1.

Theorem 4.1 *Let M_1 and M_2 be two reachable markings of a t-net. If $R(M_1)$ and $R(M_2)$ are equivalent then $M_1 \sim M_2$.*

It is worth noting that at this stage, markings do not have a unique representation. However in the following section we will introduce a sufficient condition to detect all symmetries within a system.

5   Discussing canonisation

As shown in previous section, there are different representations of a marking. These representations differ in the number of paths they contain. But as shown in proposition 4.1 the number of paths in a pid-tree can be bounded by the following equation:

$$pid_{q_M} \cup nextpid_{q_M} \subseteq pid(R(M)) \subseteq subpid_{q_M} \cup nextpid_{q_M}$$

In next two subsections we will explore two canonical representations of sibling ordered pid-trees. The first corresponding to the upper bound of the representation and the second one to the lower bound.

5.1   Expanded pid-tree representation

At this point we have a representation of markings as sibling ordered pid-trees and a pid-tree equivalence relation that implies an equivalence of markings which lead to the detection of symmetries. However using the rules we provide one can build different pid-trees that respect the definition 4.2, indeed the definition of a path (definition 3.5) allows to strip or expand branches inside the tree.

Definition 5.1 (expanded pid-trees) *A pid-tree $T \in \Xi$ is expanded if for all $\langle \pi, t \rangle$ in $Trees(T)$ such that $t = M \xrightarrow{a_1,\ldots,a_n} \langle t_1, \ldots, t_n \rangle$ we have $length(a_i) = 1$ for $1 \leq i \leq n$.*

In an expanded pid-tree all pids associated to children nodes are of length 1. Intuitively we can see an expanded pid-tree as the biggest representation of a pid-tree, no more nodes can be added to the tree without breaking the property about bounds. Pid-trees in Figures 1(b) and 1(c) are expanded pid-trees but pid-trees in Figures 1(a) and 1(d) are not.

Proposition 5.1 *Let M be a reachable marking of a t-net N. There is a unique expanded pid-tree in $repr(M)$.*

5.2   Stripped pid-tree representation

Now we will minimize the tree by removing all inactive pids. When a path labelled by π is added to the tree we potentially add all the subpids of π which may be inactive. Removing all inactive pids lead to stripped pid-trees.

Definition 5.2 *A stripped pid-tree is a pid-tree T such that for all $path(\pi) \in paths(T)$ we have π active.*

This second canonical form will lead to more symmetry detection. This may seem counterintuitive since we remove data from the tree, but removing these data also removes constraints we add on pids. These constraints are introduced by inactive pids that do not appear in the marking and removing them lead to a better characterisation of the relations between pids. Because inactive pids do not appear in the original marking, they should not impact on symmetry detection. Stripping a pid-tree will remove these pids. An example is given in Figure 4.

Proposition 5.2 *Let M be a reachable marking of a t-net N. There is a unique stripped pid-tree in $repr(M)$.*

Even if removing inactive pids allows to provide a better characterisation of markings, some symmetries may still miss. An example is given in Figure 4(e) and 4(f), the trees are not equivalent but the corresponding states are, because pids 1.1 and 1.2 are inactive the relation between them can be released (they cannot be compared within the model) and thus 1.1.1 and 1.2.1 can exchange roles. So we now introduce a sufficient condition to detect all symmetries with pid-trees.

(a) Pid-tree T_1 (b) Pid-tree T_2 (c) Stripped version of T_1

(d) Stripped version of T_2

$M_1 \stackrel{\text{df}}{=} \{(1.1.1, 1.2.1)\}$
$M_2 \stackrel{\text{df}}{=} \{(1.2.1, 1)\}$

(e) A stripped pid-tree.

$M_2' \stackrel{\text{df}}{=} \{(1.1.1, 1)\}$
$M_1' \stackrel{\text{df}}{=} \{(1.2.1, 1.1.1)\}$
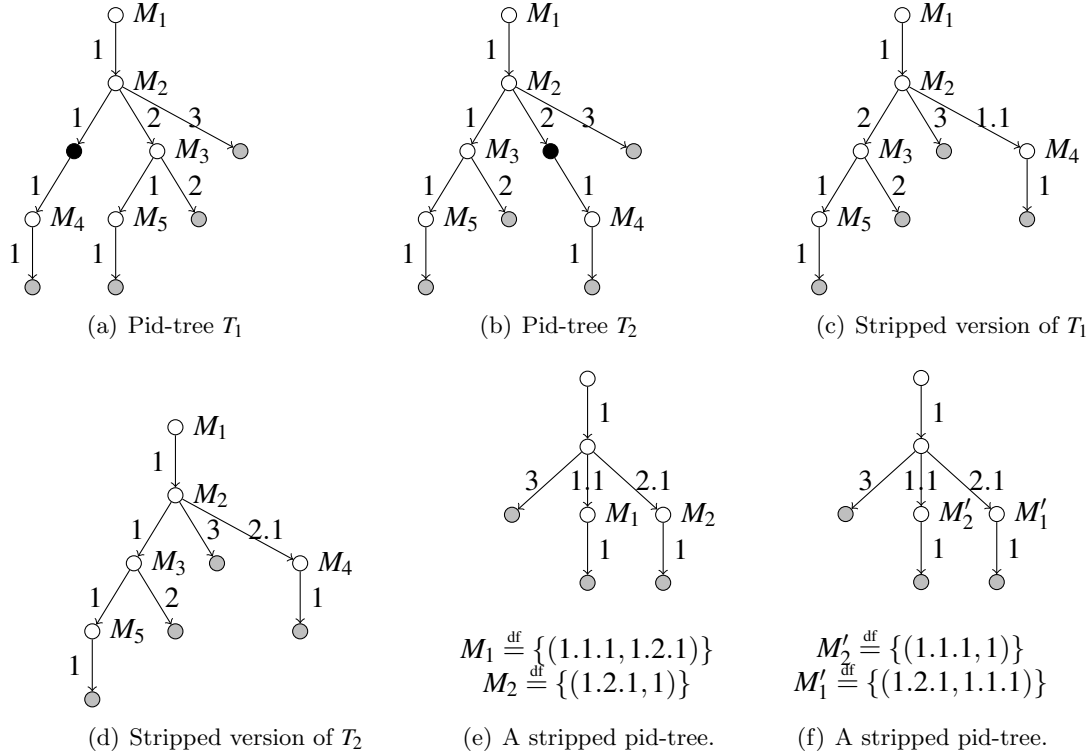
(f) A stripped pid-tree.

Figure 4: Two pid-trees which are not equivalent (4(a), 4(b)) but become equivalent when transformed into stripped pid-trees (4(c), 4(d)). The pids-trees in Figure 4(e) and 4(f) are not equivalent but denote equivalent states. Most of markings have been omitted, black nodes represent inactive pids, gray nodes represent next pids.

5.3   Sufficient condition for a unique representation

To start with, we introduce the notion of *clean marking*. A marking is *clean* if for every active pid, or next-pid, its parent is active, except for the pid 1 "the father of all pids". This notion have several consequences. First, there will be a unique representation of a marking, *i.e.*, $repr(M) = \{R(M)\}$. The second consequence is that we will have:

$$pid_{q_M} \cup nextpid_{q_M} = pid(R(M)) = subpid_{q_M} \cup nextpid_{q_M}$$

So the stripped and expanded representations of the marking are equal.

Proposition 5.3 *Let M be a reachable clean marking of a t-net. Then there is a unique pid-tree representation of M.*

Now we show that the bijection used to check pid-tree equivalence is exactly the bijection used to check equivalence between markings.

Proposition 5.4 *Let M_1, M_2 be two reachable markings in a t-net. If M_1, M_2 are clean and $M_1 \sim M_2$ then there is a bijection h defined as*

$$h = \{(\pi, \pi') \mid \pi \in pid(R(M_1)),\ \pi' \in pid(R(M_2))\ and\ \widetilde{path}(\pi, R(M_1)) = \widetilde{path}(\pi', R(M_2))\}$$

such that $M_1 \sim_h M_2$.

This proposition states that paths inside the pid-tree representation perfectly characterise the bijection between states corresponding to clean markings. It implies that if our pid-trees are equivalent then the bijection used to check pid-tree equivalence also satisfies the state equivalence requirements.

Theorem 5.1 *Let M_1 and M_2 be two reachable markings. If M_1 and M_2 are clean then $R(M_1) \sim R(M_2)$ iff $M_1 \sim M_2$.*

When addressing clean markings we detect all symmetries in state spaces. It may seem a big constraint to work with clean markings, however it can be ensured by restriction at modelling level. One can for example force a process to wait its children to finish before dying; in these cases every reachable marking will be clean.

Moreover, when addressing arbitrary systems, we can use, for example, stripped pid-trees to detect potential symmetries with the guarantee that for every discovered clean marking, we will detect all symmetries.

6   Conclusion

We have developed a theoretical package for symmetries detection in Petri nets with dynamic process creation. This approach is not complete however it leads to effective algorithms that can be implemented in a real-life model-checker. We have also shown its limits and discussed its application and the completeness of symmetry detection. This has been made using a tree data structure, the *pid-trees*, and an equivalence relation among them. The development of this package have been done in a general scope which results in potential symmetry detection. Then we have looked for canonisation with *expanded* and *stripped* pid-trees in order to get a unique representation of a marking. Finally we expressed a sufficient condition to detect all symmetries within a model. Using pid-trees leads to time efficient algorithms to detect symmetries during state space exploration, which is not the case when resorting to graph isomorphism computation [10, 11, 6].

The different canonical forms we have introduced allow to implement the approach and detect symmetries. Even if all symmetries are not detected we believe that many of them are captured. We will try to confirm this statement in future works. We intend to implement our reductions by symmetries within *Neco* net compiler [4, 5]. It will lead to case studies and several benchmarks to validate our intuition about expected performances, however we will not be able to test every existing model. This is why the sufficient condition we have introduced in section 5.3 is important. It guarantees that we will detect all symmetries in a class of models, moreover if the condition is not satisfied the method remains valid even if we lose this guarantee of full detection.

We will also explore possibilities given by the space between our two bounds, expanded and stripped pid-trees, using more precise labelling functions to capture symmetries, and address systems without relations ⋔₁ and ⋔. For example, pid-trees in Figures 4(e) and 4(f) become equivalent using a labelling function which takes in account the size, or contents, of markings.

The trade-off between complexity of the computation and completeness was the main motivation in this work. Indeed developing a complete approach with an exponential complexity in time would not be usable in a real life model-checker where the number of discovered states is almost always exponential in the size of the model. Even if reduction by symmetries may exponentially reduce the size of the state space, such an expensive approach will remain not usable due to the number of states as explained in the introduction. Thus partial methods like this one should be considered in practice even if they do not capture every symmetry.

References

[1] D. Bosnacki, D. Dams & L. Holenderski (2000): *Symmetric Spin.* In: *SPIN Model Checking and Software Verification*, *LNCS* 1885, Springer Berlin / Heidelberg, doi:10.1007/10722468_1.

[2] E. Clarke, E. Emerson, S. Jha & A. Sistla (1998): *Symmetry Reductions in Model Checking.* In: *Computer Aided Verification*, *LNCS* 1427, Springer Berlin / Heidelberg, doi:10.1007/BFb0028741.

[3] Ł. Fronc (2012): *Effective Marking Equivalence Checking.* Technical Report, IBISC. Available at `http://goo.gl/mvBor`.

[4] L. Fronc & F. Pommereau (2011): *Optimizing the Compilation of Petri Net Models.* In: *Proc. of SUMo'11*, 726, CEUR.

[5] L. Fronc & F. Pommereau (2011): *Towards a Certified Petri Net Model-Checker.* In: *APLAS*, *LNCS* 7078, Springer, doi:10.1007/978-3-642-25318-8_24.

[6] Klaudel H., Koutny M., Pelz E. & Pommereau F. (2010): *State Space Reduction for Dynamic Process Creation.* Sci. Ann. Comp. Sci. 20.

[7] M. Hendriks, G. Behrmann, K. Larsen, P. Niebert & F. Vaandrager (2004): *Adding Symmetry Reduction to Uppaal.* In: *Formal Modeling and Analysis of Timed Systems*, *LNCS* 2791, Springer Berlin / Heidelberg, doi:10.1007/978-3-540-40903-8_5.

[8] K. Jensen & Kristensen L.M. (2009): *Coloured Petri Nets - Modelling and Validation of Concurrent Systems.* Springer, doi:10.1007/b95112.

[9] T. Junttila (2003): *On the Symmetry Reduction Method for Petri Nets and Similar Formalisms.* PhD. Thesis, Espoo, Finland.

[10] H. Klaudel, M. Koutny, E. Pelz & F. Pommereau (2008): *Towards Efficient Verification of Systems with Dynamic Process Creation.* LNCS 5160, Springer, doi:10.1007/978-3-540-85762-4_13.

[11] H. Klaudel, M. Koutny, E. Pelz & F. Pommereau (2009): *An Approach to State Space Reduction for Systems with Dynamic Process Creation.* In: *ISCIS*, IEEE Digital Library, doi:10.1109/ISCIS.2009.5291864.

[12] B. McKay: *Nauty.* Available at `http://cs.anu.edu.au/people/bdm/nauty/`.

[13] B. McKay (1981): *Practical Graph Isomorphism.* In: *Manitoba Conference on Numerical Mathematics and Computing*, Congressus Numerantium, Utilitas Mathematica Pub.

[14] K. McMillan (1993): *Symbolic Model Checking.* Kluwer, doi:10.1007/978-1-4615-3190-6.

[15] A. Miller, A. Donaldson & M. Calder (2006): *Symmetry in Temporal Logic Model Checking.* ACM Computing Surveys 38(3), doi:10.1145/1132960.1132962.