Towards Efficient Verification of Systems with Dynamic Process Creation

Hanna Klaudel, Maciej Koutny, Elisabeth Pelz, Franck Pommereau

Towards Efficient Verification of Systems with Dynamic Process Creation

Hanna Klaudel[1], Maciej Koutny[2], Elisabeth Pelz[3], and Franck Pommereau[3]

[1] IBISC, University of Evry, bd F. Mitterrand, 91025 Evry, France
hanna.klaudel@ibisc.fr
[2] SCS, Newcastle University, Newcastle upon Tyne, NE1 7RU, UK
maciej.koutny@newcastle.ac.uk
[3] LACL, University of Paris Est, 61 av. du général de Gaulle, 94010 Créteil, France
{pelz,pommereau}@univ-paris12.fr

Abstract. Modelling and analysis of dynamic multi-threaded state systems often encounters obstacles when one wants to use automated verification methods, such as model checking. Our aim in this paper is to develop a technical device for coping with one such obstacle, namely that caused by dynamic process creation.
We first introduce a general class of coloured Petri nets—not tied to any particular syntax or approach—allowing one to capture systems with dynamic (and concurrent) process creation as well as capable of manipulating data. Following this, we introduce the central notion of our method which is a marking equivalence that can be efficiently computed and then used, for instance, to aggregate markings in a reachability graph. In some situations, such an aggregation may produce a finite representation of an infinite state system which still allows one to establish the relevant behavioural properties. We show feasibility of the method on an example and provide initial experimental results.
Keywords: Petri nets, multi-threaded systems, marking symmetries, state-space generation.

1 Introduction

Multi-threading is a programming feature with an ever increasing presence due to its central role in a broad range of application areas, including web services, business computing, virtual reality, pervasive systems, and networks-on-a-chip. Given this and the widely acknowledged complexity of multi-threaded designs, there is a growing demand to provide methods supporting the highest possible confidence in their correctness. In a multi-threaded (or multi-process) programming paradigm, sequential code can be executed repeatedly in concurrent threads interacting through shared data and/or rendezvous communication. In this paper, we consider a Petri net model that captures such a scheme in a general fashion: programs are represented by Petri nets, and the active threads are identified by differently coloured tokens which use, in particular, thread identifiers. Such programs and their corresponding representation in coloured Petri

nets may be obtained compositionally from algebras of Petri nets (*e.g.*, [19]) which ensures that several behavioural properties of the resulting nets may be validated automatically and/or syntactically (*i.e.*, by construction). The corresponding class of nets may also be characterised using a suitable combination of structural properties. The latter approach is used in this paper in order to avoid dealing with a concrete net algebra.

The presence of thread identifiers in net markings has the potential of accelerating the state space explosion, and so poses an additional threat for the efficiency of verification. However, thread identifiers are arbitrary (anonymous) symbols whose sole role is to ensure a consistent (*i.e.*, private or local) execution of each thread. The exact identity of an identifier is basically irrelevant, and what matters are the *relationships* between such identifiers, *e.g.*, being a thread created by another thread. As a result, (sets of) identifiers may often be swapped with other (sets of) identifiers without changing the resulting execution in any essential way. Moreover, an infinite state system can sometimes be reduced to a finite representation which in turn allows one to model check the relevant behavioural properties (*e.g.*, mutual exclusion or deadlock freeness). This leads to the problem of identifying symmetric executions, which must be addressed by any reasonable verification and/or simulation approach to multi-threaded programming schemes.

In this paper, we propose a method that contributes towards an efficient verification approach for multi-threaded systems modelled using a class of coloured Petri nets. At its core lies a marking equivalence that identifies global states which have essentially isomorphic future behaviour up to renaming of thread identifiers. The equivalence can be computed efficiently and then it may be used to aggregate nodes in a marking graph, or to find cut-offs during the unfolding of a Petri net [14]. The proposed method is complemented with a generation scheme for concrete values of thread identifiers that is both distributed and concurrent.

An important feature of the method is that it is parameterised by a set of operations that can be applied to thread identifiers. For instance, it may or may not be allowed to test whether one thread is a direct descendant of another thread, and the proposed method takes this into account.

Context and related works. The difficulty of reasoning about the behaviour of multiple threads operating on shared data has motivated the development of a variety of formalisms and methods for the modelling and detecting various kinds of errors, *e.g.*, data races, deadlocks and violations of data invariants.

In proof based methods, such as the recent approaches in [11,23], the model is described by means of axioms, and properties are theorems to be verified using a theorem prover. These techniques have the advantage of being applicable to infinite state systems, but the use of theorem provers can be a deeply technical task which is hard to automate.

As an alternative approach, model checking techniques (see [5]) allow one to achieve a high degree of confidence in system correctness in an essentially automatic way. This is done by exhaustively checking a finite system model for

violations of a correctness requirement specified formally as, *e.g.*, a temporal logic formula [16]. However, this makes model checking sensitive to the state explosion problem, and so it may not be well suited to tackle real-life systems. A variety of methods (see, *e.g.*, [18] for a recent survey) address the state explosion problem by exploiting, for instance, symmetries in the system model, in order to avoid searching parts of the state space which are equivalent to those that have already been explored. Several techniques have been implemented in widely used verification tools, such as [3,13,17], and proved to be successful in the analysis of complex communication protocols and distributed systems.

When dealing with multi-threaded systems, model checking typically involves manual definition of models using low-level modelling means, such as pseudo programming languages, Petri nets or process algebras. Some recent methodologies (*e.g.* [24,6,1]) allow one to verify (Java or C) program invariants by combining model checking and abstract interpretation [8], while others (*e.g.* [9]) propose dedicated high-level input languages (a combination of multiset rewriting and constraints) allowing one to use verification techniques employing symbolic representations of infinite state spaces. In the domain of coloured Petri nets, extensive work has been conducted to make model checking efficient through the use of symbolic reachability graph constructions [4,25], and by exploiting various kinds of partial order reductions [10,14].

Being general purpose techniques rather than designed specifically for multi-threaded systems, the above approaches do not exploit explicitly symmetries related to thread identifiers. An example of work which addresses expressivity and decidability aspects of various extensions of P/T Petri nets allowing, in particular, fresh name generation and process replication is [22]. However, it only allows equality tests on process identifiers, and does not deal with aspects related to the efficiency of verification.

Outline of the paper. After introducing basic concepts concerning thread identifiers, we present a class of Petri nets used to model multi-threaded systems and establish some relevant properties of their reachable markings. We then define an equivalence relation on markings abstracting from the identities of thread identifiers and discuss its main features. The paper ends with a procedure for checking the equivalence, supported by an example and some initial experimental results. All proofs and auxiliary results are provided in the technical report [15].

2 Process identifiers

We denote by $\mathbb{D}$ the set of *data values* which, in particular, contains all integers. We then denote by $\mathbb{V}$ the set of variables such that $\mathbb{V} \cap \mathbb{D} = \varnothing$. The set $\mathbb{P}$, disjoint with $\mathbb{D} \cup \mathbb{V}$, is the set of *process identifiers*, (or *pids*) that allow one to distinguish different concurrent threads during an execution. We assume that there is a set $\mathbb{I} \subset \mathbb{P}$ of *initial* pids, *i.e.*, threads active at the system startup. To keep the formal treatment simpler, we assume throughout this paper that at the beginning there is just one active thread, and so $|\mathbb{I}| = 1$. This is a harmless

restriction since any non-trivial initial marking (with several active threads) can be created from a restricted one by firing an initialisation transition.

Operations on process identifiers. It is possible to check whether two pids are equal or not since different threads must be distinguished. Other operations may also be applied to thread identifiers, in particular:

- $\pi \lhd_1 \pi'$ checks whether π is the parent of π' (*i.e.*, thread π spawned thread π' at some point of its execution).
- $\pi \lhd \pi'$ checks whether π is an ancestor of π' (*i.e.*, $\lhd$ is $\lhd_1^+$).
- $\pi \pitchfork_1 \pi'$ checks whether π is a sibling of π' and π was spawned immediately before π' (*i.e.*, after spawning π, the parent of π and π' did not spawn any other thread before spawning π').
- $\pi \pitchfork \pi'$ checks whether π is an elder sibling of π' (*i.e.*, $\pitchfork$ is $\pitchfork_1^+$).

Throughout the paper, we will denote by Ω_{pid} the set of the four relations introduced above, as yet informally, together with the equality. In particular, only the operators in Ω_{pid} can be used to compare pids in the annotations used in Petri nets. Crucially, it is not allowed to *decompose* a pid (to extract, for example, the parent pid of a given pid) which is considered as an atomic value (or black box), and no literals nor concrete pid values are allowed in Petri net annotations (*i.e.*, in guards and arc labels) involving the pids.

The resulting formalism is rich while still being decidable. Indeed, it can be shown that the monadic second order theory of $\mathbb{P}$ equipped with $\lhd_1$ and $\pitchfork_1$ can be reduced to the theory of binary trees equipped with the left-child and right-child relation which, in turn, has been shown to be exactly as expressive as the tree automata [21]. Having said that, many simple extensions of the formalism based on pids (de)composition are undecidable.

Thread implementation. We assume that there exists a function ν generating the i-th child of the thread identified by a pid π, and that there is no other way to generate a new pid. In order to avoid creating the same pid twice, each thread is assumed to maintain a count of the threads it has already spawned.

A possible way of implementing dynamic pids creation—adopted in this paper—is to consider them as finite strings of positive integers written down as dot-separated sequences. Then we take $\mathbb{I} \stackrel{\text{df}}{=} \{1\}$ and, for all π and i, we set $\nu(\pi, i-1) \stackrel{\text{df}}{=} \pi.i$ (*i.e.*, the i-th pid generated from π is $\pi.i$, and $i-1$ is the number of pids generated so far from π). With such a representation, the relations in Ω_{pid} other than equality are given by:

- $\pi \lhd_1 \pi'$ iff $(\exists i \in \mathbb{N}^+)$ $\pi.i = \pi'$
- $\pi \lhd \pi'$ iff $(\exists n \geq 1)\ (\exists i_1, \ldots, i_n \in \mathbb{N}^+)$ $\pi.i_1.\cdots.i_n = \pi'$
- $\pi \pitchfork_1 \pi'$ iff $(\exists \pi'' \in \mathbb{P})\ (\exists i \in \mathbb{N}^+)$ $\pi = \pi''.i \ \wedge\ \pi' = \pi''.(i+1)$
- $\pi \pitchfork \pi'$ iff $(\exists \pi'' \in \mathbb{P})\ (\exists i < j \in \mathbb{N}^+)$ $\pi = \pi''.i \ \wedge\ \pi' = \pi''.j$

Such a scheme has several advantages: (i) it is deterministic and allows for distributed generation of pids; (ii) it is simple and easy to implement without

re-using the pids; and (iii) it may be bounded by restricting, *e.g.*, the length of the pids, or the maximum number of children spawned by each thread.

3 Coloured Petri nets

We start with a general definition of coloured Petri nets and their dynamic behaviour. More details about this particular formalism and, in particular, variable bindings and operations on multisets, can be found in [2].

Definition 1 (Petri net graph). *A* Petri net graph *is a tuple (S, T, ℓ) where S is a finite set of* places*, T is a finite set of* transitions *(disjoint from S), and ℓ is a* labelling *of places, transitions and arcs (in $(S \times T) \cup (T \times S)$) such that:*

- *For each place $s \in S$, $\ell(s)$ is a Cartesian product of subsets of pids and data, called the* type *of s.*
- *For each transition t, $\ell(t)$ is a computable Boolean expression, called the* guard *of t.*
- *For each arc α, $\ell(\alpha)$ is a finite set of tuples of values and/or variables.* $\diamond$

Since we allow tuples as token values, it is possible to represent complex data structures in a flattened form (as Cartesian products). In what follows, the set of all finite tuples beginning with a value or variable x will be denoted by $\mathbb{T}_x$.

Definition 2 (Petri net and its behaviour). *A* marking *M of a Petri net graph (S, T, ℓ) is a mapping that associates with each $s \in S$ a finite multiset of values in $\ell(s)$. A* Petri net *is then defined as $N \stackrel{\text{df}}{=} (S, T, \ell, M_0)$, where M_0 is the* initial *marking.*

A transition $t \in T$ is enabled *at a marking M if there exists a binding $\sigma : \mathbb{V} \to \mathbb{D}$ such that $\sigma(\ell(t))$ evaluates to true and, for all $s \in S$, $\sigma(\ell(s,t)) \leq M(s)$ and $\sigma(\ell(t,s))$ is a multiset over $\ell(s)$. (In other words, there are enough tokens in the input places, and the types of the output places are being respected.)*

An enabled t may fire *producing the marking M', defined for all $s \in S$ by $M'(s) \stackrel{\text{df}}{=} M(s) - \sigma(\ell(s,t)) + \sigma(\ell(t,s))$. We denote this by $M[t, \sigma\rangle M'$.*

We also denote $M_0 \to^* M$ if M is produced from M_0 through the firing of a finite sequence of transitions, i.e.*, if the marking M is* reachable *(from M_0).* $\diamond$

We will use a specific family of Petri nets respecting structural restrictions detailed below. Throughout the rest of this section, N is as in definition 2.

Assumption 1 (places) *The set of places is partitioned into a unique* generator *place s_{gen}, a possibly empty set of* data *places S_{data}, and a nonempty set of* control-flow *places S_{flow},* i.e.*, $S \stackrel{\text{df}}{=} \{s_{gen}\} \uplus S_{data} \uplus S_{flow}$. It is assumed that:*

1. *The generator place s_{gen} has the type $\mathbb{P} \times \mathbb{N}$.*
2. *Each control-flow or data place s has the type $\mathbb{P} \times \mathbb{P}^{k_s} \times \mathbb{D}_s$ where $\mathbb{D}_s \subseteq \mathbb{D}^{m_s}$, for some $k_s, m_s \geq 0$.* $\diamond$

The typing discipline for places ensures that we can talk about each token $\langle \pi, \cdots \rangle$ being *owned* by a thread, the pid π of which is the first component of the token. A pid is *active* at a marking if it owns a token in the generator place.

Data places store, for different threads, tuples of data and/or pids owned by currently and previously active threads.

Control-flow places indicate where the control of active threads resides. When a control-flow place s is such that $k_s + m_s \geq 1$, the information following the pid of the owner provides the status of the execution; for instance, allowing one to find out whether an exception has been raised (like in [19]).

The generator place s_{gen} is needed by the underlying scheme for the dynamic creation of fresh pids. For each active thread π, it stores a *generator* token $\langle \pi, i \rangle$ where i is the number of threads already spawned by π. Thus the next thread to be created by π will receive the pid $\pi.(i+1)$.

Assumption 2 (initial marking) *The initial marking is such that:*

1. *All data places are empty.*
2. *The generator place contains exactly one token,* $\langle 1, 0 \rangle$.
3. *There is exactly one control-flow place that is non-empty, its type is* $\mathbb{P}$ *and it contains exactly one token,* $\langle 1 \rangle$. $\diamond$

Firing a transition t captures a progression of one or several threads which meet at a rendezvous. Below, the threads entering the rendezvous belong to a finite non-empty set $\mathcal{E} \subset \mathbb{V}$. Some of them (in the set $\mathcal{X} \subseteq \mathcal{E}$) may exit the rendezvous, others may terminate (in $\mathcal{E} \setminus \mathcal{X}$), and new ones may be created (in the set $\mathcal{N}$). Each of the created threads is spawned by one of the threads entering the rendezvous. Without lost of generality, if all the entering threads terminate, we assume that at least one is created (in order to ensure that each transition has at least one output arc to a control-flow place).

Each thread e entering the rendezvous creates $k_e \geq 0$ children. Their pids are generated using the generator place s_{gen} that holds a counter $g \in \mathbb{N}$ for e (as for all active pids). This counter for e stored in s_{gen} is incremented by k_e when the transition fires, and the pids of the generated threads are $e.(g+1), \ldots, e.(g+k_e)$.

At the same time, e may access data using the *get* operation, which consumes a token from a data place, or the *put* operation, which produces a token and inserts it into a data place. If the tokens involved are owned by e, this corresponds to data management for e. For example, getting and putting the same value into the same data place corresponds to reading, while getting one value and putting another one into the same data place corresponds to an update (the computation of the new value may be expressed through the guard of t). If the tokens involved are not owned by e, this corresponds to asynchronous communication through shared variables. In such a case, the put operation corresponds to the sending of a message, while the get corresponds to the receiving of a value deposited by another thread.

The purely syntactic restrictions on arcs and guards given below ensure that pids are not treated as (transformable) data. Markings are not involved, and so each thread will be identified by the variable bound to an actual pid at firing

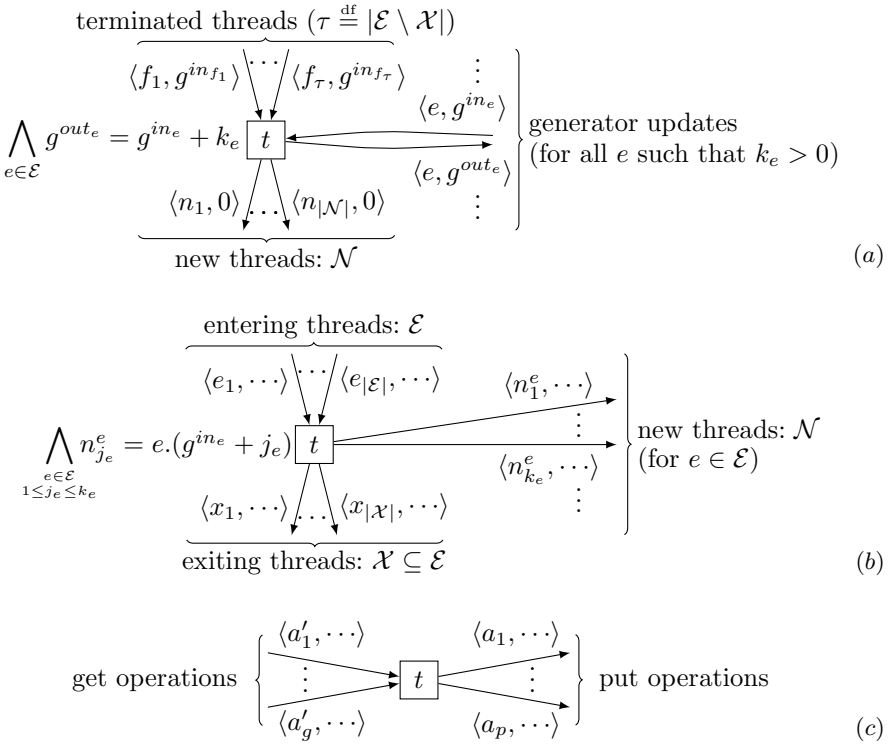

Fig. 1. Parts of the guard and the shape of arcs for assumption 3. In (a) the arcs are connected to the generator place, in (b) to control-flow places, and in (c) to data places. The a_i's and a'_j's are variables.

time. This will not cause any confusion as each active pid will always appear only once in exactly one control-flow place.

Assumption 3 (transitions, arcs and guards) *For each transition $t \in T$, the following specifies all the arcs, arc annotations and guard components.*

1. *The sets of threads $\mathcal{E}$, $\mathcal{X}$ and $\mathcal{N}$ are defined as:*

$$\mathcal{E} \stackrel{\text{df}}{=} \{e \mid s \in S_{flow} \wedge \ell(s,t) \cap \mathbb{T}_e \neq \varnothing\} ,$$
$$\mathcal{X} \stackrel{\text{df}}{=} \{x \mid s \in S_{flow} \wedge \ell(t,s) \cap \mathbb{T}_x \neq \varnothing\} \cap \mathcal{E} ,$$
$$\mathcal{N} \stackrel{\text{df}}{=} \{n \mid s \in S_{flow} \wedge \ell(t,s) \cap \mathbb{T}_n \neq \varnothing\} \setminus \mathcal{E} \;=\; \biguplus_{e \in \mathcal{E}} \{n^e_1, \ldots, n^e_{k_e}\} .$$

   It is assumed that $\mathcal{E}$, $\mathcal{X}$ and $\mathcal{N}$ are subsets of $\mathbb{V}$ and $\mathcal{E} \neq \varnothing \neq \mathcal{X} \cup \mathcal{N}$.
2. *t is connected to the control-flow places as shown in figure 1(b), where:*
   - *For each $e \in \mathcal{E}$, there exists exactly one control-flow place s such that $\ell(s,t) \cap \mathbb{T}_e \neq \varnothing$. Moreover, $|\ell(s,t) \cap \mathbb{T}_e| = 1$.*
   - *For each $x \in \mathcal{X}$, there exists exactly one control-flow place s such that $\ell(t,s) \cap \mathbb{T}_x \neq \varnothing$. Moreover, $|\ell(t,s) \cap \mathbb{T}_x| = 1$.*

- *For each $n \in \mathcal{N}$, there exists exactly one control-flow place s such that $\ell(t,s) \cap \mathbb{T}_n \neq \varnothing$. Moreover, $|\ell(t,s) \cap \mathbb{T}_n| = 1$.*

3. *t is connected to the generator place s_{gen} as shown in figure 1(a), where:*
   - *For each $f \in \mathcal{E} \setminus \mathcal{X}$, $\ell(s_{gen},t) \cap \mathbb{T}_f = \{\langle f, g^{in_f}\rangle\}$ where $g^{in_f} \in \mathbb{V}$.*
   - *For each $n \in \mathcal{N}$, $\ell(t,s_{gen}) \cap \mathbb{T}_n = \{\langle n, 0\rangle\}$.*
   - *For each $e \in \mathcal{E}$ with $k_e > 0$, $\ell(s_{gen},t) \cap \mathbb{T}_e = \{\langle e, g^{in_e}\rangle\}$ and $\ell(t,s_{gen}) \cap \mathbb{T}_e = \{\langle e, g^{out_e}\rangle\}$ where $g^{in_e}, g^{out_e} \in \mathbb{V}$.*
   - *For each $e \in \mathcal{E}$ with $k_e = 0$, $\ell(s_{gen},t) \cap \mathbb{T}_e = \ell(t,s_{gen}) \cap \mathbb{T}_e = \varnothing$.*
4. *There is no restriction on how t is connected to the data places. As illustrated in figure 1(c), each put operation corresponds to a tuple in the label of an arc from t to a data place while each get operation corresponds to a tuple in the label of an arc from a data place to t.*
5. *The variables occurring in the annotations of the arcs adjacent to t can be partitioned into* pid variables *and* data variables*, as follows: for each place $s \in S$ which has the type $\mathbb{P} \times \mathbb{P}^{k_s} \times D_1 \times \cdots \times D_{m_s}$, for each tuple $\langle x_0, x_1, \ldots, x_{k_s}, y_1, \ldots, y_{m_s}\rangle \in \ell(s,t) \cup \ell(t,s)$, the x_i's are pid variables and the y_j's are data variables. In other words, locally to each transition, a variable cannot be used simultaneously for pids and data.*
6. *The guard of t is a conjunction of the formulas corresponding to:*
   - *The creation of the new pids:* $\bigwedge_{e \in \mathcal{E}, 1 \leq j_e \leq k_e} n^e_{j_e} = e.(g^{in_e} + j_e)$.
   - *The updating of counters of spawned threads:* $\bigwedge_{e \in \mathcal{E}} g^{out_e} = g^{in_e} + k_e$.
   - *A Boolean formula expressing a particular firing condition and data manipulation, where only the operations from Ω_{pid} are allowed on pid variables.* ◇

Finally, any N obeying the above assumptions is a *thread Petri net* (or *t-net*).

4 Properties of reachable markings

We want to capture some useful properties of t-net behaviours. First, we introduce control safeness and consistent thread configurations which will be used to characterise pids occurring in reachable t-net markings.

Definition 3 (control safe markings). *A t-net marking M is* control safe *if, for each pid $\pi \in \mathbb{P}$, one of the following holds:*

- *There is exactly one token owned by π in the generator place and exactly one token owned by π in exactly one of the control-flow places (note that this unique place may contain tokens not owned by π).*
- *Tokens owned by π (if any) appear only in the data places.* ◇

Control safeness ensures that each thread is sequential, and that there is no duplication of control-flow tokens.

Definition 4 (ct-configuration). *A* consistent thread configuration *(or ct-configuration) is a pair* $ctc \stackrel{\text{df}}{=} (G, H)$*, where* $G \subset \mathbb{P} \times \mathbb{N}$ *and* $H \subset \mathbb{P}$ *are finite sets. Assuming that* $pid_G \stackrel{\text{df}}{=} \{\pi \mid \langle \pi, i \rangle \in G\}$ *and* $pid_{ctc} \stackrel{\text{df}}{=} pid_G \cup H$*, the following are satisfied, for all* $\langle \pi, i \rangle \in G$ *and* $\pi' \in pid_{ctc}$*:*

1. $\langle \pi, j \rangle \notin G$*, for every* $j \neq i$*.*
2. *If* $\pi \lhd \pi'$ *then there is* $j \leq i$ *such that* $\pi.j = \pi'$ *or* $\pi.j \lhd \pi'$*.*

We also denote $nextpid_{ctc} \stackrel{\text{df}}{=} \{\pi.(i+1) \mid \langle \pi, i \rangle \in G\}$*.* ◇

Intuitively, G represents tokens held in the generator place, pid_G comprises pids of active threads, and H keeps record of all the pids that might occur in the data tokens of some reachable t-net marking.

Definition 5 (ctc of a marking). *Given a reachable t-net marking* M*, we define* $ctc(M) \stackrel{\text{df}}{=} (M(s_{gen}), H)$*, where* H *is the set of all the pids occurring in the tokens held in the data and control-flow places at* M*.* ◇

We can now characterise reachable markings of t-nets.

Theorem 1. *Let* M *be a reachable t-net marking.*

1. M *is control safe.*
2. $ctc(M)$ *is a ct-configuration.* ◇

Knowing that all reachable t-net markings are control safe will allow us to identify those which admit essentially the same future behaviour. We start with an auxiliary definition at the level of ct-configurations (see [15] for its soundness).

Definition 6 (isomorphic ct-configurations). *Two ct-configurations,* $ctc = (G, H)$ *and* $ctc' = (G', H')$*, are* h-isomorphic*, denoted by* $ctc \sim_h ctc'$*, if there is a bijection* $h : (pid_{ctc} \cup nextpid_{ctc}) \to (pid_{ctc'} \cup nextpid_{ctc'})$ *such that:*

1. $h(pid_G) = pid_{G'}$*.*
2. *For all* $\langle \pi, i \rangle \in G$ *and* $\langle h(\pi), j \rangle \in G'$*,* $h(\pi.(i+1)) = h(\pi).(j+1)$*.*
3. *For* $\prec$ *in* $\{\lhd_1, \lhd\}$ *and* $\pi, \pi' \in pid_{ctc}$*:* $\pi \prec \pi'$ *iff* $h(\pi) \prec h(\pi')$*.*
4. *For* $\curlywedge$ *in* $\{\pitchfork_1, \pitchfork\}$ *and* $\pi, \pi' \in pid_{ctc} \cup nextpid_{ctc}$*:* $\pi \curlywedge \pi'$ *iff* $h(\pi) \curlywedge h(\pi')$*.* ◇

We now can introduce the central notion of this paper.

Definition 7 (marking equivalence). *Let* M *and* M' *be reachable markings of a t-net such that* $ctc(M) \sim_h ctc(M')$*. Then* M *and* M' *are* h-isomorphic *if:*

- *For each control-flow or data place* s*,* $M'(s)$ *can be obtained from* $M(s)$ *by replacing each pid* π *occurring in the tuples of* $M(s)$ *by* $h(\pi)$*.*
- $h(\{\pi \mid \langle \pi, i \rangle \in M(s_{gen})\}) = \{\pi' \mid \langle \pi', i' \rangle \in M'(s_{gen})\}$*.*

We denote this by $M \sim_h M'$ *or simply by* $M \sim M'$*.* ◇

The equivalence $M \sim_h M'$ means that pids are related through h, and data in tokens in control-flow and data places remain unchanged. As far as the generator tokens are concerned, the only requirement is that they involve h-corresponding pids.

As shown in [15], $\sim$ is an equivalence relation. It follows from the next result that it captures a truly strong notion of marking similarity.

Theorem 2. *Let M and M' be h-isomorphic reachable markings of a t-net, and t be a transition such that $M[t,\sigma\rangle\widetilde{M}$. Then $M'[t, h \circ \sigma\rangle\widetilde{M'}$, where $\widetilde{M'}$ is a marking such that $\widetilde{M} \sim_{\widetilde{h}} \widetilde{M'}$ for a bijection $\widetilde{h}$ coinciding with h on the intersection of their domains.* $\diamond$

Moreover, the above result still holds if Ω_{pid} is restricted to any of its subsets that includes pid equality.

5 Checking marking equivalence

We check marking equivalence in two steps. First, markings are mapped to three-layered labelled directed graphs, and then the graphs are checked for isomorphism.

The three-layered graphs are constructed as follows. Layer-I nodes are labelled by places, layer-II by (abstracted) tokens and layer-III by (abstracted) pids. The arcs are of two kinds: those going from the container object toward the contained object (places contain tokens which in turn contain pids), and those between the vertices of layer-III reflecting the relationship between the corresponding pids through the comparisons in Ω_{pid} other than equality, denoted below as $\lhd_j$ (see figure 4).

Definition 8 (graph representation of markings). *Let M be a reachable marking of a t-net N. The corresponding* graph representation

$$R(M) \stackrel{\text{df}}{=} (V; A, A_{\lhd_1}, \ldots, A_{\lhd_\ell}; \lambda) \ ,$$

where V is the set of vertices, A, $A_{\lhd_1}$, ..., $A_{\lhd_\ell}$ are sets of arcs and λ is a labelling of vertices and arcs, is defined as follows:

1. *Layer-I: for each control-flow or data place s in N such that $M(s) \neq \varnothing$, s is a vertex in V labelled by s.*
2. *Layer-II: for each control-flow or data place s, and for each token $v \in M(s)$, v is a vertex in V labelled by $\lfloor v \rfloor$ (which is v with all pids replaced by epsilon's) and there is an unlabelled arc $s \longrightarrow v$ in A.*
   Note: separate copies of node v are created for different occurrences of v in case $M(s)(v) > 1$.
3. *Layer-III:*
   - *for each vertex v added at layer-II, for each pid π in v at the position n (in the tuple), π is an ε-labelled vertex in V and $v \stackrel{n}{\longrightarrow} \pi$ an arc in A.*

- *for each token* $\langle \pi, i \rangle \in M(s_{gen})$, $\pi.(i+1)$ *(that is, the potential next child of* π*) is a vertex in* V *labelled by* ε.
- *for all vertices* π, π' *added at layer-III, for all* $1 \leq j \leq \ell$, *there is an arc* $\pi \xrightarrow{\lhd_j} \pi'$ *in* $A_{\lhd_j}$ *iff* $\pi \lhd_j \pi'$ *(that is,* $A_{\lhd_j}$ *defines the graph of the relation* $\lhd_j$ *on* $V \cap \mathbb{P}$*).*

4. *There is no other vertex nor arc in* $R(M)$. ◇

To gain efficiency, $R(M)$ may be optimised by removing some vertices and arcs, *e.g.*, each subgraph $\langle \pi \rangle \xrightarrow{0} \pi$ can be replaced by π.

Theorem 3. *Let* M_1 *and* M_2 *be two reachable markings of a t-net.* $R(M_1)$ *and* $R(M_2)$ *are isomorphic iff* $M_1 \sim M_2$. ◇

5.1 Example

In order to illustrate the proposed approach, we consider a simple server system with a bunch of threads waiting for connections from clients (not modelled). Whenever a new connection is made, a handler is spawned to process it. The handler performs some unspecified computation and then calls an auxiliary function. Terminated handlers are awaited for by the thread that spawned them. The example illustrates two typical ways of calling a subprogram: either asynchronously by spawning a thread, or synchronously by calling a function. In our setting, both ways result in creating a new thread, the only difference is that a function call is modelled by spawning a thread and immediately waiting for it. In order to simplify the presentation, data is not being modelled, only the control-flow. Moreover, for this particular example, we can take Ω_{pid} without the relations $\lhd$ and $\pitchfork$.

The whole system is modelled by the Petri net depicted in figure 2. The main process corresponds to the transitions *init*, *spawn* and *wait*:

- Upon firing *init*, the initial thread 1 terminates and creates k children that carry out the actual spawning/waiting for handler threads. The place s_1 holds pairs (pid, counter) in order to allow each thread to remember the number of handlers it has spawned.
- *spawn* creates one handler child and increments the counter. The maximum number of active children is bound by m due to the guard $c < m$.
- *wait* terminates one of the children (this is verified by the guard) and decrements the counter.

A handler process corresponds to the transitions *comp*, *call* and *ret*: *comp* models the computation performed by the handler; *call* creates one child in order to start an instance of the function; immediately after that *wait* awaits for its termination. The function itself is modelled by a single transition *fun*. The net is parameterised by two constants k and m, and so we denote it by $N_{k,m}$.

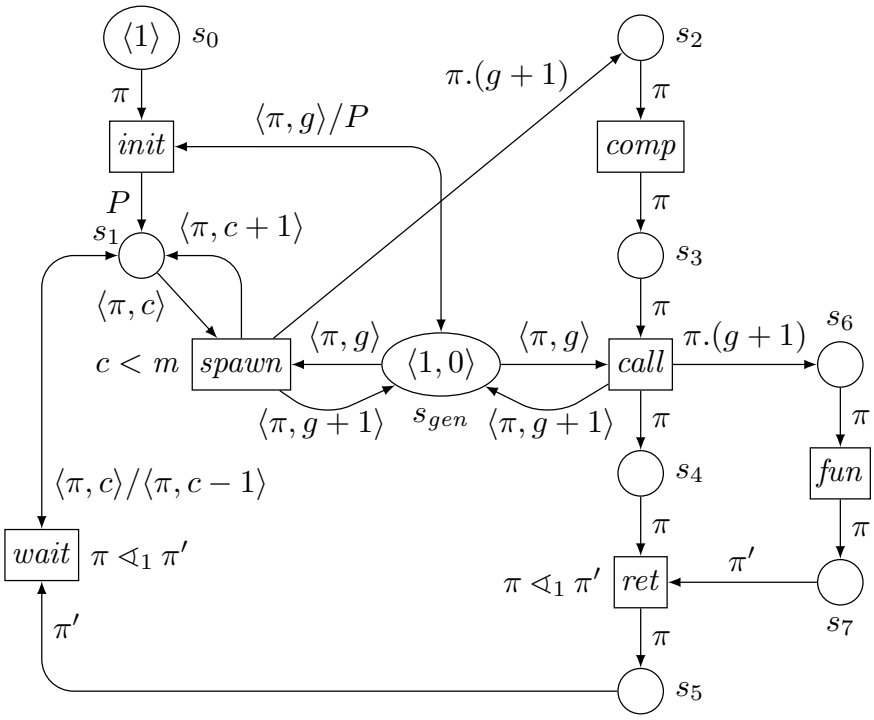

Fig. 2. The example Petri net, where $P \stackrel{\text{df}}{=} \{\langle\pi.(g+1), 0\rangle, \ldots, \langle\pi.(g+k), 0\rangle\}$. An arc with an arrow at both sides and labelled by a/b denotes that a is consumed and b is produced. All places but s_{gen} are control-flow ones. Places s_{gen} and s_1 have type $\mathbb{P} \times \mathbb{N}$ and all the other places have type $\mathbb{P}$. The angle brackets around singletons and true guards are omitted. Finally, we may write an expression E on an output arc as a shorthand for a fresh variable y instead of E, together with the condition $y = E$ in the guard of the adjacent transition.

Bounding the executions. Our approach allows to find a finite state space of the system by detecting loops in the behaviour, *i.e.*, parts of the execution that are repeated with new pids. This can be illustrated using $N_{1,1}$: its state space is infinite if we use the standard Petri net transition rule. But if we identify markings that are equivalent, it only has 7 states, as shown in figure 3.

The overall behaviour is clearly looping but, without using marking equivalence, there is no cycle in the state space. Indeed, the execution of *wait* produces the marking $\{s_1 : \langle 1.1, 0\rangle; s_{gen} : \langle 1.1, 1\rangle\}$ instead of $\langle 1.1, 0\rangle$ in s_{gen} that was created by the firing of *init*. From here, a second execution of *spawn* would produce a new pid 1.1.2 instead of 1.1.1 that was used in the first loop. By incorporating the proposed marking equivalence, the exact values of pids are abstracted as well as the marking of the generator place, which allows to detect a state which is basically the same and thus to stop the computation.

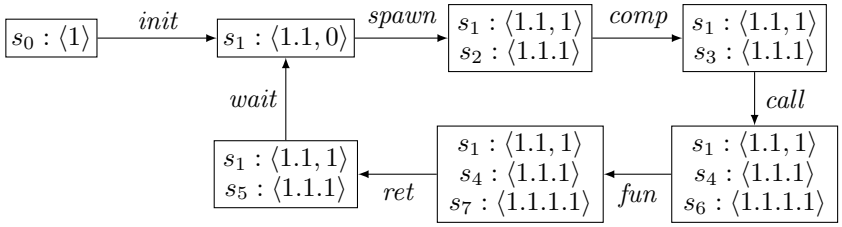

Fig. 3. The state graph of $N_{1,1}$ where the marking of s_{gen} has not been represented.

Handling symmetries. Another advantage of our approach can be illustrated using $N_{2,1}$: two main threads are started and each can have at most one active handler child. This system exhibits symmetric executions since the behaviour of both threads is concurrent but is interleaved in the marking graph. For instance, the state space has a diamond when the two threads spawn concurrently one child each. The markings corresponding to the intermediate states of the diamond are depicted in figure 4. Because the relation $\pitchfork_1$ has been taken into account, the two markings are clearly not equivalent. But, when this relation is not considered, the markings become equivalent, as shown on the right of figure 4. In the state spaces such diamonds are removed and only one interleaving preserved.

5.2 Experimental results

We have implemented a prototype of the proposed method using SNAKES [20] and NetworkX [12], for the Petri net and graph part, respectively. The latter implements VF2 [7] that is considered to be one of the fastest algorithms for checking graph isomorphism. We have generated several state spaces, using various values of k and m and considering various Ω_{pid}'s. The global execution times are not relevant since our implementation is not yet optimised. However, we measured the time spent on computing the graph isomorphism (this part is implemented efficiently) with respect to the size of the graphs representing t-net markings (measured as the product of the number a of arcs and the number v of vertices in the union of the graphs being compared). The result shows a progression that appears to be linear (see figure 6 in [15]). This suggests that the heuristics in VF2 are efficient for the kind of graphs involved in the checking of marking equivalence. Considering that $a \leq v^2$, the experimentally observed performance appears to be at worst v^3 or, equivalently, $a^{3/2}$.

6 Conclusions

Working within the context of coloured Petri nets, we proposed a technical device for coping with dynamic and concurrent creation of processes capable of manipulating data encountered, *e.g.*, in multi-threaded systems. The method introduced in this paper defines and efficiently exploits an equivalence relation on markings with essentially isomorphic future behaviours. It can be used, in

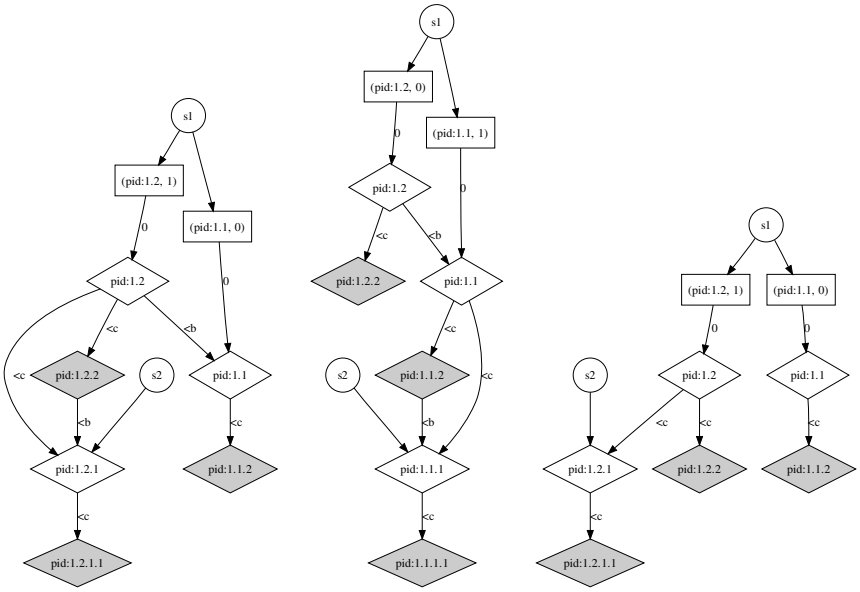

Fig. 4. Graph representations of states of $N_{2,1}$. On the right, the same states reduce to isomorphic ones when $\pitchfork_1$ is not taken into account (and so only one is depicted). The circle, square and diamond vertices depict respectively layer-I, layer-II and layer-III vertices. Gray vertices are those added on the basis of s_{gen}. $\lhd_1$ is depicted by `<c` and $\pitchfork_1$ by `<b`.

particular, to aggregate nodes in a state graph. As demonstrated by the initial experiments, this may produce efficiently a finite representation of an infinite state systems that is reduced with respect to symmetric executions.

Acknowledgements. We would like to thank Alexis Bes, Patrick Cegielski, Christian Laforest and Victor Khomenko for their comments on the earlier versions of this paper. This research was supported by NSFC Grant 60433010.

References

1. T.Ball, S.Chaki and S.K.Rajamani: *Parameterized Verification of Multithreaded Software Libraries*. Proc. TACAS'01, Lecture Notes in Computer Science 2031, Springer (2001) 158–173
2. E.Best et. al: *M-Nets: An Algebra of High-Level Petri Nets, with an Application to the Semantics of Concurrent Programming Languages*. Acta Informatica 35 (1998) 813–857
3. D.Bosnacki, D.Dams and L.Holenderski: *Symmetric Spin*. International Journal on Software Tools for Technology Transfer 4 (2002) 92–106
4. G.Chiola, C.Dutheillet, G.Franceschinis and S.Haddad: *A Symbolic Reachability Graph for Coloured Petri Nets*. Theoretical Computer Science 176 (1997) 39–65

5. E.Clarke, O.Grumberg and D.Peled: *Model Checking*. MIT Press (2000)
6. J.C.Corbett et. al: *Bandera: Extracting Finite-state Models from Java Source Code*. Proc. ICSE'00, ACM (2000) 439–448
7. L.P.Cordella, P.Foggia, C.Sansone and M.Vento: *A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs*. IEEE Transactions on Pattern Analysis and Machine Intelligence 26 (2004) 1367–1372
8. P.Cousot and R.Cousot: *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. Proc. POPL'77, ACM (1977) 238–252
9. G.Delzanno: *Constraint-based Automatic Verification of Abstract Models of Multithreaded Programs*. Journal of Theory and Practice of Logic Programming 7 (2007)
10. S.Evangelista: *High Level Petri Nets Analysis with Helena*. Proc. ICATPN'05, Lecture Notes in Computer Science 3536, Springer (2005) 455–464
11. C.Flanagan, S.N.Freund, S.Qadeer and S.A.Seshia: *Modular Verification of Multithreaded Programs*. Theoretical Computer Science 338 (2005) 153–183
12. A.Hagberg, D.Schult and P.Swart: *NetworkX, High Productivity Software for Complex Networks*. `http://networkx.lanl.gov`
13. M.Hendriks et.al: *Adding Symmetry Reduction to Uppaal*. Proc. FORMATS'03, Lecture Notes in Computer Science 2791, Springer (2003) 46–59
14. V.Khomenko: *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD Thesis, School of Computing Science, University of Newcastle (2003)
15. H.Klaudel, M.Koutny, E.Pelz and F.Pommereau: *Towards Efficient Verification of Systems with Dynamic Process Creation*. LACL Technical Report (2008) `http://lacl.univ-paris12.fr`
16. Z.Manna and A.Pnueli: *The Temporal Logic of Reactive and Concurrent Systems Specification*. Springer (1991)
17. K.McMillan: *Symbolic Model Checking*. Kluwer Academic (1993)
18. A.Miller, A.Donaldson and M.Calder: *Symmetry in Temporal Logic Model Checking*. ACM Comput. Surv. 38 (2006)
19. F.Pommereau: *Versatile Boxes, a Multi-Purpose Algebra of High-Level Petri Nets*. Proc. DADS/SCSC'07, SCS/ACM (2007)
20. F.Pommereau: *Quickly Prototyping Petri Net Tools with Snakes*. Proc. PNTAP'08, ACM Digital Library (2008)
21. M.O.Rabin: *Decidability of Second-order Theories and Automata on Infinite Trees*. Transactions of the American Mathematical Society 141 (1969)
22. F.Rosa-Velardo and D.de Frutos-Escrig: *Name Creation vs. Replication in Petri Net Systems*. Proc. ICATPN'07, Lecture Notes in Computer Science 4546, Springer (2007) 402–422
23. R.F.Stärk: *Formal Specification and Verification of the C# Thread Model*. Theoretical Computer Science 343 (2005) 482–508
24. S.D.Stoller: *Model-Checking Multi-threaded Distributed Java Programs*. Proc. SPIN'00, Lecture Notes in Computer Science 1885, Springer (2000) 224–244
25. Y.Thierry-Mieg, C.Dutheillet and I.Mounier. *Automatic Symmetry Detection in Well-Formed Nets*. Proc. ICATPN'03, Lecture Notes in Computer Science 2679, Springer (2003) 82–101