View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Directory of Open Access Journals

Scientific Annals of Computer Science vol. 20, 2010
"Alexandru Ioan Cuza" University of Iaşi, Romania

State Space Reduction for Dynamic Process Creation

Hanna KLAUDEL[1], Maciej KOUTNY[2],
Elisabeth PELZ[3], Franck POMMEREAU[1]

Abstract

Automated verification of dynamic multi-threaded computing systems is severely affected by problems relating to dynamic process creation. In this paper, we describe an abstraction technique aimed at generating reduced state space representations for such systems. To make the new technique applicable to a wide range of different system models, we express it in terms of general labelled transition systems.

At the heart of our technique is an equivalence relation on system states based on a suitable isomorphism between their component parts and relationships between component process identifiers. In addition, the equivalence takes into account new process identifiers which can be derived from those present in the states being compared, in effect performing a limited lookahead.

Applying state space reduction based on such a state equivalence may produce a finite representation of an infinite state system while still allowing to validate essential behavioural properties, *e.g.*, freedom from deadlocks. We evaluate the feasibility of the proposed method through extensive experiments. The results clearly demonstrate that the new state space reduction technique can be implemented in an efficient way.

We also describe how the new state equivalence relation can be implemented for a class of high-level Petri nets supporting dynamic thread creation.

Keywords: multi-threaded systems, state equivalence, state space computation, state space reduction.

[1]IBISC, Université d'Évry, Tour Évry 2, 523 place des terrasses, 91000 Évry, France, email: `{klaudel,pommereau}@ibisc.univ-evry.fr`

[2]School of Computing Science, Newcastle University, Newcastle upon Tyne, NE1 7RU, United Kingdom, email: `maciej.koutny@ncl.ac.uk`

[3]LACL, Université Paris Est, 61 avenue du général de Gaulle, 94010 Créteil, France, email: `pelz@univ-paris12.fr`

1 Introduction

In a multi-threaded programming paradigm, sequential code can be run repeatedly in concurrent threads of execution, interacting through shared data and/or rendez-vous communication. The presence of thread identifiers in state descriptions usually accelerates the state space explosion, especially when new threads may be created dynamically. However, thread identifiers are arbitrary (anonymous) symbols whose sole role is to ensure a consistent execution of each thread. The exact identity of an identifier is irrelevant, and what only matters are the *relationships* between such identifiers, *e.g.*, parenthood or siblinghood. As a result, (sets of) identifiers may often be replaced by other (sets of) identifiers without changing the resulting execution in an essential way. This creates a possibility of identifying equivalent executions, which must be addressed by any verification and/or simulation approach to multi-threaded programming schemes.

We aim at an abstraction technique for generating reduced state space representations for multi-threaded systems with dynamic process creation. To make it applicable to a wide range of different system models, we consider in this paper a general model-independent (behavioural) framework of labelled transition systems. We formulate conditions allowing one to identify behaviourally equivalent system states and, since state equivalence is required to be preserved over system evolutions, to identify equivalent executions. The equivalence relation on system states is based on an isomorphism between their component parts and relationships between the component process identifiers. Moreover, one takes into account new process identifiers which can be derived from those present in the states being compared, in effect performing a limited lookahead. Applying state space reduction based on such a state equivalence may even in some cases produce a finite representation of an infinite state system while still allowing to validate essential behavioural properties, *e.g.*, freedom from deadlocks.

Another distinguishing feature of the proposed state equivalence 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 or indirect descendant of another thread. The approach easily adapts to what is usually allowed and this can be a crucial point to maximise the degree of state space reduction. As shown later on, using fewer operations usually leads to better reduction, because process identifiers are more likely to be equivalent if there are fewer possibilities to compare them.

We evaluate the feasibility of the proposed method through extensive

experiments in order to show that the new state space reduction technique can be implemented in an efficient way. We also present a concrete system model based on high-level Petri nets, in which the key property required of the state equivalence relation — its preservation over system evolution — holds thanks to mild *syntactic restrictions* placed upon the structure of the net. In this way, the paper prepares the ground for future applications of the new state space reduction technique.

The approach presented in this paper systematises and extends our earlier work initiated in [11, 12]: in particular, the Petri net implementation of our approach has been dramatically simplified, all the proofs are now provided, and we now consider an additional isomorphism algorithm and draw conclusions based on a rigorous analysis of our experimental measurements.

> **Running example.** Let us consider a server system in which a bunch of threads listen for connections from clients requesting some calculation. Figure 1 shows a message sequence chart of a typical session. Whenever a new request arrives, a listener thread creates a handler thread to process the request. The handler calls an auxiliary function to perform the required calculation and then sends the answer back to the client. Terminated handlers are collected asynchronously by the thread that created them. The client part is depicted for the sake of clarity but will not be considered in the model to keep things simpler.
>
> This example illustrates two standard ways of calling a subprogram: either asynchronously by creating a thread, or synchronously by calling a function. In our setting, both these methods amount to creating a new thread, the only difference is that a function call is modelled by creating a thread and immediately waiting for its termination.

The paper is organised as follows. We first characterise the class of labelled transition systems for which it is feasible to apply identification of states inspired by the marking equivalence of [11]. We then provide experimental results about the effort needed to verify the state equivalence which reaffirms our initial hypothesis that this can be done in an efficient way. Finally, we show how the state equivalence can be treated in a class of high-level Petri nets supporting dynamic thread creation. More precisely, for each net in this class the generated labelled transition system satisfies all the requirements formulated in the general setting.

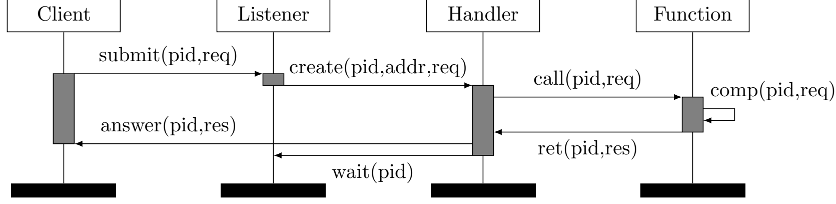


Figure 1: Running example: sequence diagram of a session.

2 Multi-thread modelling

We denote by $\mathbb{D}$ the set of *data values*, and by $\mathbb{P} \stackrel{\text{df}}{=} \{\pi, \pi', \pi'', \ldots\}$ a disjoint set of *process identifiers*, (or *pids* for short) 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. A possible way of implementing dynamic pid creation—and one adopted in this paper—is to consider them as finite sequences of positive integers (written down as dot-separated strings, for example, 1.2 or 2.3.1). In modelled systems, it is not allowed to *decompose* a pid (*e.g.*, to extract the parent pid of a given pid) which is considered to be an atomic value (a black box), nor it is allowed to use concrete pid values (literals).

To compare pids within Boolean expressions, we use a set of binary relations on pids, $\Omega_{pid} \stackrel{\text{df}}{=} \{=, \lessdot_1, \lessdot, \pitchfork_1, \pitchfork\}$, such that:

- $=$ is the equality on pids;
- $\pi \lessdot_1 \pi'$ (which holds if there is a positive integer i such that $\pi.i = \pi'$) means that π is the parent of π' (*i.e.*, thread π created thread π');
- $\pi \lessdot \pi'$ means that π is an ancestor of π' (*i.e.*, $\lessdot$ is the transitive closure $\lessdot_1^+$);
- $\pi \pitchfork_1 \pi'$ (which holds if there is a pid π'' and a positive integer i such that $\pi = \pi''.i$ and $\pi' = \pi''.(i+1)$) means that π is a sibling of π' and π was created immediately before π' (*i.e.*, after creating π, the parent π'' of π and π' did not create any other thread before creating π'); and
- $\pi \pitchfork \pi'$ means that π is an elder sibling of π' (*i.e.*, $\pitchfork$ is the transitive closure $\pitchfork_1^+$).

It is assumed that only the operators in Ω_{pid} can be used to compare pids.

The above scheme has several advantages: it is deterministic and can be distributed to support concurrent generation of pids; it is simple and easy to implement; and it may be bounded by restricting, *e.g.*, the length of the pids, or the maximal number of pid instances spawned by each thread.

3 Transition systems and state equivalence

In this paper, a *labelled transition system* LTS provides a description of all reachable states of some multi-threaded system, operating over some finite set of locations $\mathbb{L}$, together with transitions between these states. As usual, there is a distinguished *initial* state from which all other states can be reached. Locations may correspond intuitively to the variables of the system under analysis, or to any other similar "data holder". Each state $q \stackrel{\text{df}}{=} (\sigma_q, \eta_q)$ is composed of a pair of mappings:

- a mapping σ_q from $\mathbb{L}$ to finite multisets of *vectors* (tuples) of data and pids;
- a mapping η_q which for each *active* thread (*i.e.*, one which belongs to the domain of η_q) gives the number of threads it has already created.

Given a state q as above, we define the following:

- For each active pid π in q the next pid created (also called *next-pid*) is given by $next_q(\pi) \stackrel{\text{df}}{=} \pi.(\eta_q(\pi) + 1)$;
- The next-pids of q are $nextpid_q \stackrel{\text{df}}{=} \{next_q(\pi) \mid \pi \in dom(\eta_q)\}$;
- pid_q is the set of all pids involved in σ_q; and
- $(G_q, H_q) \stackrel{\text{df}}{=} (pid_q, \eta_q)$ is the *thread configuration* of q.

Each transition $q \xrightarrow{t} q'$ of an LTS is labelled by t which can be the name of a command or action (possibly internal) together with the actual parameters which may include pids present in pid_q.

Assumption 1 *When moving from a state q to q' along a transition $q \xrightarrow{t} q'$, the pids present in q' must either be present in q or be newly created using the information provided by η_q (*i.e., $pid_{q'} \subseteq pid_q \cup nextpid_q$*). Moreover, any pid active at q' must be active in q or be a newly created one (*i.e., $dom(\eta_{q'}) \subseteq dom(\eta_q) \cup nextpid_q$*).*

Not all potential states can be considered as valid. For example, one should prohibit the generation of an already existing pid. This can be achieved by requiring that no pid involved in σ_q can be derived as a future child of an active thread.

Assumption 2 *Each state q reachable from the initial one generates a* consistent *thread configuration (or ct-configuration) (G_q, H_q) which means that:*

- *$dom(H_q) \subseteq G_q$,* i.e., *each active pid is present in the state; and*
- *for all $\pi \in dom(H_q)$ and $\pi' \in G_q$, if $\pi.k$ is a prefix of π' then $k \leq \eta_q(\pi)$,* i.e., *pids present in q cannot be created again.*

In the rest of the section we will introduce the notion of equivalence for reachable states of an LTS. First, however, we look at our example.

> **Running example.** An initial fragment of the LTS for the server system is illustrated in Figure 2, where:
>
> - $\mathbb{I} = \{1, 2\}$;
> - $\mathbb{L} = \{L, H, F\}$ are the locations;
> - 1, 2, 1.1, 1.2, 2.1, 2.2, 1.1.1 and 2.1.1 are pids; and
> - 2009, 0 and *addr* are data items.
>
> It portrays two alternative execution branches corresponding to the same scenario played by two different threads. In this scenario, a server calculates whether a given year is a leap year. A listener receives a data *req* = 2009, creates a handler, and passes to it *req* together with client's address *addr*. The handler calls a function passing *req* to it, the function calculates the result *res* = 0, and returns it to the handler. The handler passes on the result to the client (not modelled). Finally, the listener terminates the handler.
>
> Looking at Figure 2, one can note that the two branches,
>
> $$q_0 \xrightarrow{t_1} q_1 \xrightarrow{t_2} \cdots \xrightarrow{t_5} q_5 \quad \text{and} \quad q_0' \xrightarrow{t_1'} q_1' \xrightarrow{t_2'} \cdots \xrightarrow{t_5'} q_5' \ ,$$
>
> are intuitively equivalent since the roles of pids 1 and 2 can be swapped to obtain one from the other. Moreover, q_0 is equivalent to q_5 and q_5'. For example, q_5 is the same as q_0 except for the value of $\eta(1)$.

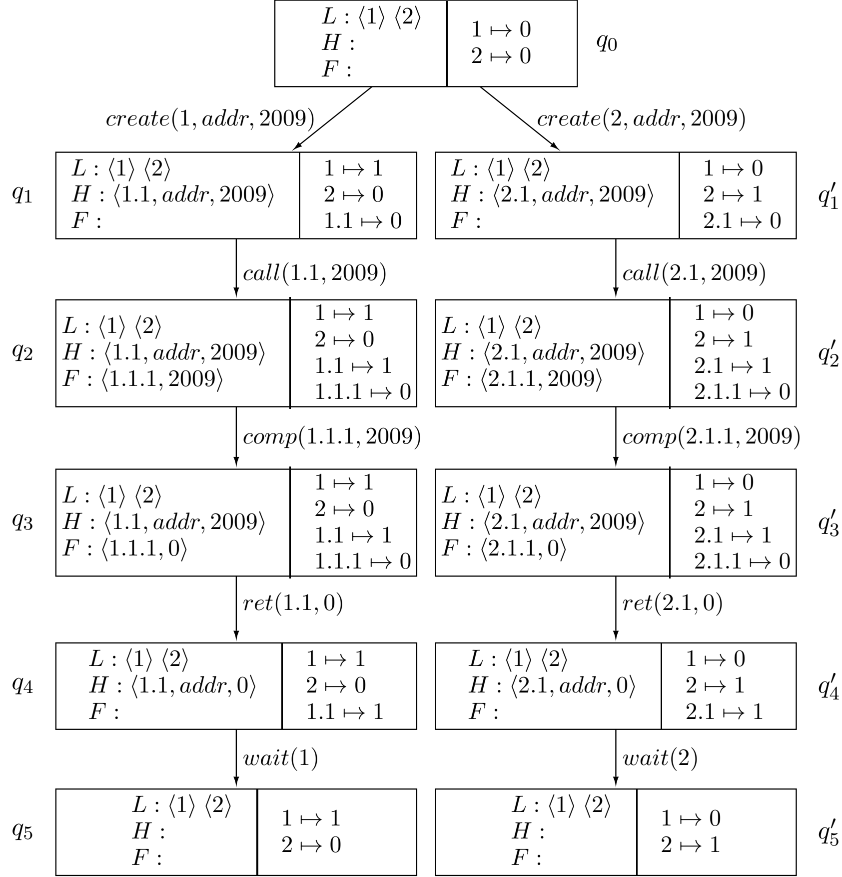


Figure 2: A possible LTS for the running example: σ_q is shown in the left and η_q in the right part of each state.

We can now proceed with the formal definition of states equivalence. It should be stressed that our formalisation is more elaborate than the equivalence relations such as that defined in abstract terms in [8]. The reason is that our definition not only looks at the actual components of two states being compared, but also 'looks ahead' including potential future component pids which can be created from the existing ones. Clearly, such a lookahead strategy needs to be carefully designed in order not to make the whole approach infeasible.

Definition 1 *Two states q and q' defining ct-configurations are* equivalent *if there is a bijection $h : (pid_q \cup nextpid_q) \to (pid_{q'} \cup nextpid_{q'})$ such that for all relations $\prec \in \{\lhd_1, \lhd\}$ and $\curlywedge \in \{\pitchfork_1, \pitchfork\}$:*

- $h(dom(\eta_q)) = dom(\eta_{q'})$;
- $\forall \pi \in dom(\eta_q)$, $h(next_q(\pi)) = next_{q'}(h(\pi))$;
- $\forall \pi, \pi' \in pid_q$: $\pi \prec \pi'$ *iff* $h(\pi) \prec h(\pi')$;
- $\forall \pi, \pi' \in pid_q \cup nextpid_q$: $\pi \curlywedge \pi'$ *iff* $h(\pi) \curlywedge h(\pi')$;
- $\sigma_{q'}$ *is* σ_q *after replacing each pid* π *by* $h(\pi)$.

We denote this by $q \sim_h q'$ (or $q \sim q'$).

The lookahead feature in the above definition stems from the fact that the mapping h used to relate the two states operates on the pids in $pid_q \cup nextpid_q$ and $pid_{q'} \cup nextpid_{q'}$, and so takes into account pids which are *not* present in the states being compared. In contrast, a similar alternative mapping h_{alt} in the approach of, *e.g.*, [8] would be of the form $h_{alt} : pid_q \to pid_{q'}$. Now, the argument in support of our approach is very strong because no mapping h_{alt} could in general provide a satisfactory solution. To prove that we do need the next-pids (*i.e.*$nextpid_q$ and $nextpid_{q'}$) in Definition 1, we provide a simple example.

> **Counterexample.** Let q and q' be states such that $pid_q = \{1, 1.1\}$, $\eta_q(1) = 1$, $pid_{q'} = \{5, 5.1\}$ and $\eta_{q'}(5) = 3$. Then consider a bijection $h_{alt} : pid_q \to pid_{q'}$ such that $h_{alt}(1) = 5$ and $h_{alt}(1.1) = 5.1$. Everything is fine as far as preserving the parenthood/siblinghood of the corresponding pids is concerned. Let us now imagine that the two corresponding active threads, 1 and 5, created new pids, leading respectively to new states r and r' such that $pid_r = \{1, 1.1, 1.2\}$ and $pid_{r'} = \{5, 5.1, 5.4\}$. Then the two new states are no longer equivalent, because $1.1 \pitchfork_1 1.2$ yet $5.1 \not\pitchfork_1 5.4$. This violates the preservation of the immediate siblinghood relation of the corresponding pids, and so the two new states are not equivalent. By including next-pids in the definition of our mapping h, we avoid the problem as q and q' are no longer equivalent states.

To be of use for state space reduction, the equivalence relation introduced in Definition 1 should be preserved through possible executions.

Assumption 3 *If $q \sim_h q'$ and $q \xrightarrow{t} r$ then $q' \xrightarrow{h(t)} r'$ and $r \sim_{h'} r'$ where h and h' coincide on the intersection of their domains. And similarly for $q' \xrightarrow{t'} r'$.*
Note: $h(t)$ denotes t with each occurrence of $\pi \in pid_q$ replaced by $h(\pi)$.

The above assumption (which needs to be demonstrated for each concrete system model such as that described in Section 6) means that $\sim$ behaves like a strong bisimulation relation and can therefore be regarded as sufficient, *e.g.*, for the purpose of state reduction for deadlock detection. Moreover, the fact that the bijection h is preserved on the retained pids over the corresponding transitions means that it is also sufficient if one deals with properties of individual (abstracted) threads over sequences of states, making it compatible with the unfolding based verification technique [10].

4 Checking state equivalence

Checking state equivalence proceeds in two phases. First, candidate states are mapped to layered labelled directed graphs (or LGs), and then the LGs are checked for graph isomorphism.

LGs are constructed as follows. The first layer is labelled by locations, the second layer by (abstracted) vectors, the third layer by (abstracted) active pids and the fourth layer by (abstracted) next-pids. The arcs are of two sorts: those going from the container object towards the contained object (locations contain vectors which contain pids), and those between the vertices of the third and fourth layers reflecting the relationship between the corresponding pids through the comparisons in $\Omega^{\star}_{pid} \stackrel{\mathrm{df}}{=} \{\lhd_1, \ldots, \lhd_k\}$, where $\Omega^{\star}_{pid}$ is Ω_{pid} without the equality relation and without any relation that is not needed in the model (*i.e.*, the concrete system model generating an LTS does not use such a relation). Figures 3 and 4 show examples with $\Omega^{\star}_{pid} = \{\lhd_1\}$. The abstraction mapping $\lfloor\,\rfloor \colon \mathbb{D} \cup \mathbb{P} \to \mathbb{D} \cup \{\epsilon\}$ is defined as the identity on $\mathbb{D}$ and as a constant mapping ϵ on $\mathbb{P}$, extended component-wise to vectors.

Let $q \stackrel{\mathrm{df}}{=} (\sigma_q, \eta_q)$ be a state of an LTS. Then the corresponding *labelled graph* representation

$$LG(s) \stackrel{\mathrm{df}}{=} (V; A, A_{\lhd_1} \ldots, A_{\lhd_k}; \lambda) \ ,$$

where V is the set of vertices (which is composed of location names, vectors and pids), $A, A_{\lhd_1}, \ldots, A_{\lhd_k}$ are sets of arcs and λ is a labelling on vertices and arcs, is defined as follows:

1. First layer: for each location $\ell \in \mathbb{L}$ such that $\sigma_q(\ell) \neq \varnothing$, ℓ is a vertex in V labelled by itself, *i.e.*, ℓ.

2. Second layer: for each location $\ell \in \mathbb{L}$ and for each vector $v \in \sigma_q(\ell)$, v is a vertex in V labelled by $\lfloor v \rfloor$ and $\ell \longrightarrow v$ is an unlabelled arc in A.

3. Third layer: for each vertex v of the second layer and for each pid π in v at the position n (in the vector), π is an ϵ-labelled vertex in V and there is an arc $v \xrightarrow{n} \pi$ in A.

4. Fourth layer: for each active pid π, its potential next child, $next_q(\pi)$, is a vertex in V labelled by ϵ.

For all vertices π, π' of the third and fourth layers, and for all $1 \leq j \leq k$, there is an arc $\pi \xrightarrow{\lhd_j} \pi'$ in $A_{\lhd_j}$ iff $\pi \lhd_j \pi'$, *i.e.*, $A_{\lhd_j}$ defines the graph of the relation $\lhd_j$ on $V \cap \mathbb{P}$. There is no other vertex nor arc in $LG(s)$.

In diagrams, we do not show arcs for relations that can be deduced from the depicted ones. For instance, if there exists a path $x \xrightarrow{\lhd_1} y \xrightarrow{\lhd_1} z$, then we do not depict the arc $x \xrightarrow{\lhd} z$ nor $x \xrightarrow{\lhd} y$ nor $y \xrightarrow{\lhd} z$.

Theorem 1 *Let q_1 and q_2 be two reachable states. Then $LG(q_1)$ and $LG(q_2)$ are isomorphic iff $q_1 \sim q_2$.*

Proof: (sketch) Let $LG_i \stackrel{\text{df}}{=} LG(q_i) = (V_i; A_i, A_{i_{\lhd_1}}, \ldots, A_{i_{\lhd_k}}; \lambda_i)$ for $i \in \{1, 2\}$.

($\Rightarrow$) Let $h\colon V_1 \to V_2$ be an isomorphism between LG_1 and LG_2, such that $\forall v \in V_1$, $\lambda_1(v) = \lambda_2(h(v))$ and $\forall u, v \in V_1$, $\lambda_1((u, v)) = \lambda_2((h(u), h(v)))$. The states $q_i \stackrel{\text{df}}{=} (\sigma_{q_i}, \eta_{q_i})$ can easily be obtained from LG_i: (i) vertices of the second layer represent the vectors associated to locations which are vertices of the first layer; this allows one to obtain σ_{q_i}; and (ii) vertices of the fourth layer allow to retrieve η_{q_i}, *i.e.*, for each such vertex $\pi.j \in V_i$, there is an active thread π in q_i and $\eta_{q_i}(\pi) = j - 1$. Moreover, all pids (present as vertices of the third and fourth layer) are related through h, which is always the identity on data. So, the state equivalence follows.

($\Leftarrow$) Let $q_1 \sim_h q_2$. By definition, LG_1 and LG_2 only differ by the identity of some vertices (their number, arcs and labelling being identical). By definition of the state equivalence, h is the identity on data and relates

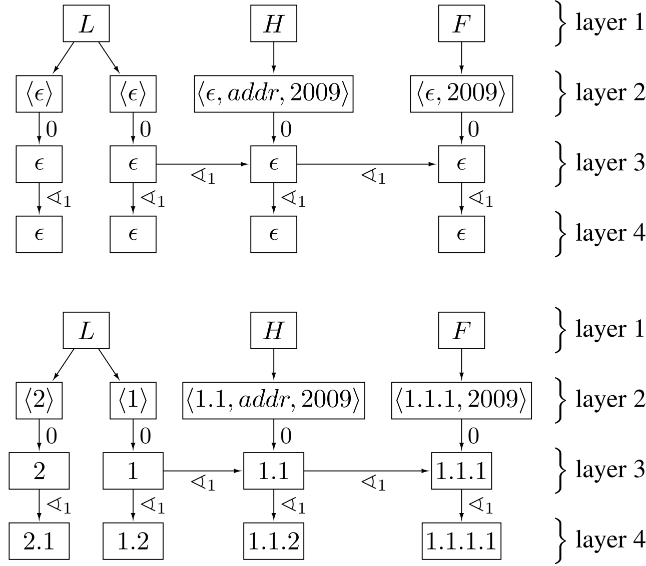


Figure 3: LG of q_3 and below its version with explicit vertex names included to improve readability.

pids between q_1 and q_2. So h relates in the same way the identities of vertices in V_1 and V_2. $\square$

Running example. In our server example, the siblinghood relation is not needed to compare pids. Indeed, the only requirement is that a parent thread waits for one of its children to terminate. So it is not necessary to consider $\pitchfork_1$ nor $\pitchfork$ in $\Omega^{\star}_{pid}$. After taking this into account, the identification of equivalent states in the LTS of Figure 2 leads to a reduced state space. First, the LGs of q_3 and q_3' are clearly isomorphic (see Figure 3). The same holds for all pairs q_i, q_i' for $1 \leq i \leq 5$. Thus, only one of the two execution branches would be present in a reduced representation, which shows how symmetric executions can be identified.

Note that considering also the siblinghood relation for $LG(q_3)$ and $LG(q_3')$ would add extra arcs, *e.g.*, from the node of pid 1 towards that of pid 2. This would result in losing the isomorphism of the two LGs. Consequently, in order to increase the reduction

rate, it is important to keep in $\Omega^{\star}_{pid}$ only those relations that actually used by the system model generating an LTS.

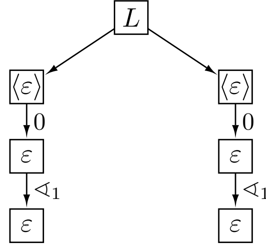


Figure 4: LG of q_0.

Let us then consider the initial state q_0 whose LG is represented in Figure 4. It is easy to check that it is isomorphic to the LGs of q_5 and q'_5. Thus, the infinite behaviour that is present in each execution branch can be reduced to a loop because we consistently abstract away the information about newly generated pids.

5 Experimental results

As the complexity of checking graph isomorphism is in general unknown, it is essential to evaluate how efficient in practice it can be verified in the case of checking the state equivalence defined in the previous section. We therefore devised a systematic and thorough series of experiments aimed at estimating the complexity of checking the isomorphism of graphs generated by two states. We started by observing the following:

> *Checking isomorphism will, in general, be very efficient when the states being compared differ significantly,* e.g.*, when the number of non-empty locations or pids involved is different.*

The reason is that in such cases the non-equivalence can be detected, *e.g.*, in linear time w.r.t. the number of locations. In particular, the experiments should not consider pairs of independently generated states, as they are likely to differ by a wide margin. Consequently, the adopted testing methodology consisted in generating a random state q and then comparing it with a number of similar (both equivalent or non-equivalent) states obtained through a number of transformations modifying the original state q.

The above methodology stems from our overall aim not to provide a verification tool but instead a method to be incorporated into such a tool. Using a set of verification case studies would not be appropriate for our purpose because this could introduce an undesirable bias. Indeed, any case study would necessarily yield states of a particular shape that could influence the efficiency of isomorphism checking. Furthermore, one would expect that any set of case studies contains balanced subsets of favourable, neutral and unfavourable cases. Considering randomly generated states can therefore be viewed as considering an arbitrarily diversified set of case studies. Moreover, as explained above, we have considered only similar states and thus only the less favourable cases from the point of view of our approach. As a consequence, our results can be seen as concerning the worst cases of any specific case study. Finally, running effective case studies would imply much more than checking isomorphisms, which would also introduce bias since the overall performance would then be influenced by many more (difficult to control) factors.

To illustrate transformations used in our experiments, let us consider the following example with two locations, each location comprising two vectors:

$$\begin{array}{lcll} L & : & \langle\, 2{:}2, 2 \,\rangle & \langle\, 3, 2.1{:}0 \,\rangle \\ L' & : & \langle\, 1{:}0, 3, 4{:}0 \,\rangle & \langle\, 2.2{:}0 \,\rangle \end{array}$$

where each pid is followed by a colon and the number of threads it has already created. The transformations we applied were as follows:

- Transform component vectors so that the resulting state is equivalent:

$$\begin{array}{lcll} L & : & \langle\, 2.3.2.5{:}5, 2 \,\rangle & \langle\, 3, 2.3.2.5.4{:}3 \,\rangle \\ L' & : & \langle\, 2.3.2.4{:}3, 3, 2.3.2.7{:}3 \,\rangle & \langle\, 2.3.2.5.5{:}3 \,\rangle \end{array}$$

- Exchange vectors between locations:

$$\begin{array}{lcll} L & : & \langle\, 3, 2.1{:}0 \,\rangle & \langle\, 1{:}0, 3, 4{:}0 \,\rangle \\ L' & : & \langle\, 2.2{:}0 \,\rangle & \langle\, 2{:}2, 2 \,\rangle \end{array}$$

- Exchange components within vectors:

$$\begin{array}{lcll} L & : & \langle\, 2, 2{:}2 \,\rangle & \langle\, 2.1{:}0, 3 \,\rangle \\ L' & : & \langle\, 4{:}0, 3, 1{:}0 \,\rangle & \langle\, 2.2{:}0 \,\rangle \end{array}$$

- Exchange data and/or pid components between vectors:

$$\begin{array}{lcll} L & : & \langle\, 2{:}2, 3\,\rangle & \langle\, 2, 2.1{:}0\,\rangle \\ L' & : & \langle\, 1{:}0, 3, 4{:}0\,\rangle & \langle\, 2.2{:}0\,\rangle \end{array}$$

$$\begin{array}{lcll} L & : & \langle\, 2.1{:}0, 2\,\rangle & \langle\, 3, 2{:}2\,\rangle \\ L' & : & \langle\, 4{:}0, 3, 2.2{:}0\,\rangle & \langle\, 1{:}0\,\rangle \end{array}$$

$$\begin{array}{lcll} L & : & \langle\, 2.1{:}0, 3\,\rangle & \langle\, 2, 2{:}2\,\rangle \\ L' & : & \langle\, 4{:}0, 3, 2.2{:}0\,\rangle & \langle\, 1{:}0\,\rangle \end{array}$$

- Replace pids:

$$\begin{array}{lcll} L & : & \langle\, 26.25.25{:}0, 2\,\rangle & \langle\, 3, 32.32.33{:}1\,\rangle \\ L' & : & \langle\, 35.35.34{:}0, 3, 32.32.33{:}1\,\rangle & \langle\, 32.32.33.1{:}0\,\rangle \end{array}$$

- Increment the count of created pids:

$$\begin{array}{lcll} L & : & \langle\, 2{:}4, 2\,\rangle & \langle\, 3, 2.1{:}0\,\rangle \\ L' & : & \langle\, 1{:}2, 3, 4{:}4\,\rangle & \langle\, 2.2{:}3\,\rangle \end{array}$$

All these transformations were randomised with suitable parameters in order to control, *e.g.*, the number of exchanged pids.

In order to check graph isomorphism, we used NetworkX [5] implementing the VF2 [4] algorithm, as well as Sage [17] implementing the Nauty [14] algorithm that is usually regarded as the most efficient one. We carried out more than two millions of comparisons, on the basis of which we observed that the computation time:

- deteriorates with the state size and with the percentage of pids in vectors (w.r.t. data); and
- improves with the increase of distinct data values.

This should not be surprising. In particular, the presence of data leads to labelled nodes that can be quickly matched when comparing two graphs. We further observed that Nauty is more efficient than VF2, and so the analysis below is based on the results obtained for Nauty.

The experimental results are depicted in Figure 5 which shows the computation time t with respect to the number p of distinct pids in the system. We can clearly identify there two distinct components (groupings of measurements):

- the lower part which looks linear, or slightly sub-linear; and
- the upper part which visually fits in-between $O(p^2)$ and $O(p \log p)$ curves.

Each component was analysed separately using the same method.

We first extracted the components as vectors of indexed pairs (p_i, t_i) with t_i being the observed computation time for a given number p_i of pids (see also Figure 5):

$$C_1 \stackrel{\mathrm{df}}{=} \left[(p_i, t_i) \mid (1 \leq i \leq k_1) \wedge (p_i \geq 200) \wedge (t_i < \tfrac{20\,p_i}{1800})\right]$$
$$C_2 \stackrel{\mathrm{df}}{=} \left[(p_i, t_i) \mid (1 \leq i \leq k_2) \wedge (p_i \geq 200) \wedge (t_i > \tfrac{20\,p_i}{1800})\right],$$

where $t_i = \frac{20\,p_i}{1800}$ corresponds to the straight dotted line in Figure 5, and $p_i \geq 200$ is where the two components are clearly separated. We also ensured that both C_1 and C_2 are ordered so that $p_i \leq p_j$ whenever $i < j$. As it turned out, C_1 comprises 78% of the observations for $p \geq 200$, while only 22% are in C_2.

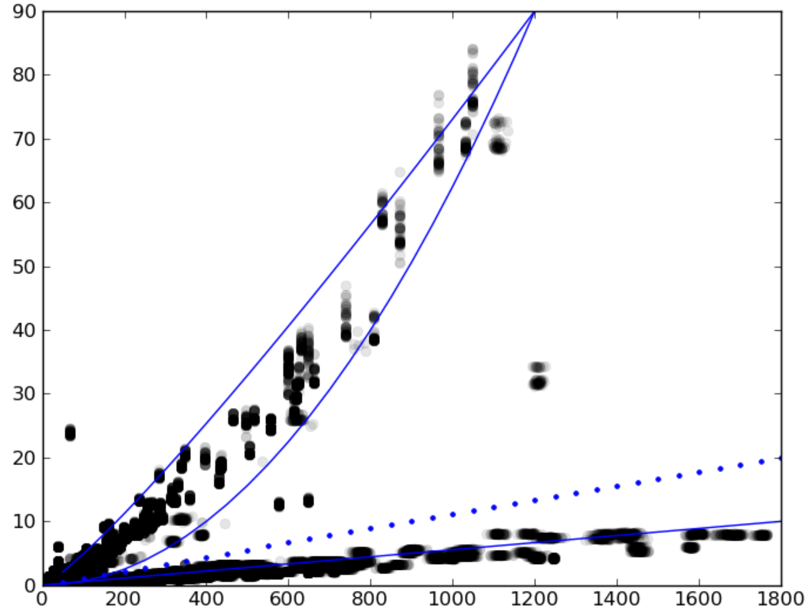
Let us assume that within the components C_1 and C_2, t can respectively be characterised by $O(p)$ and $O(p \log p)$. To verify this assumption, we computed two more vectors:

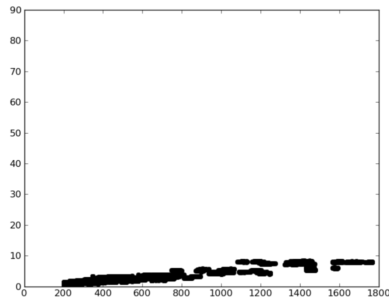
$$D_1 \stackrel{\mathrm{df}}{=} \left[\tfrac{t_i}{p_i} \mid (p_i, t_i) \in C_1\right] \quad \text{and} \quad D_2 \stackrel{\mathrm{df}}{=} \left[\tfrac{t_i}{p_i \log p_i} \mid (p_i, t_i) \in C_2\right].$$

By drawing a histogram of each D_j we checked that its values were grouped around 0.0045 for D_1, and 0.0075 for D_2. To gain more confidence about this distribution, we sub-divided each D_j into successive segments (corresponding to growing p_is) and drawn their histograms. Then, taking D_1 as an example, we have that:

- if $t = O(p)$ then the histograms should be similar for each segment;
- if $t > O(p)$ then the histogram of each segment should be right-shifted w.r.t. that of the previous segment; and
- if $t < O(p)$ then the histogram corresponding of each segment should be left-shifted w.r.t. that of the previous segment.

Figure 6 show the histograms for D_1 and D_2 with 4 segments. We can now make the following observations (which turned out to be the same for other numbers of segments that we tried):





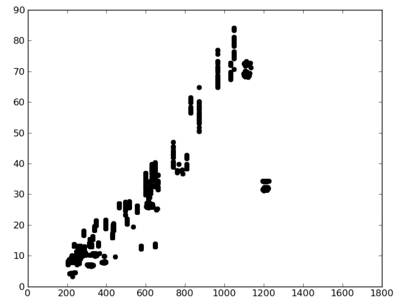


Figure 5: Top: observed computation times t w.r.t the number p of unique pids in compared states (darker points correspond to more frequent observations). The straight line is $O(p)$, the upper curve is $O(p \log p)$ and the lower curve is $O(p^2)$. Bottom: two components, C_1 (left) and C_2 (right).

- the left column confirms that $t = O(p)$ within C_1;
- in the middle column, there is a progressive right-shift of the histograms when we go down the column, and so $t > O(p \log p)$ within C_2; and
- the right column showing the histograms for $D'_2 \stackrel{\text{df}}{=} \left[\frac{t_i}{{p_i}^2} \mid (p_i, t_i) \in C_2\right]$ confirms that $t < O(p^2)$ within C_2.

This provides a full justification of the earlier visual observation made for Figure 5. Moreover, since in the right column the shifting is faster than in the middle one, it follows that for D_2 t is closer to $O(p \log p)$ than $O(p^2)$.

As a result, we are in a position to conclude that the cost of checking state equivalence is very low for states that differ considerably (in practice, a majority of the compared pairs), and is still very good for states that are equivalent or similar: for 78% of them, it is linear with respect to the number p of unique pids in the state, and for the remaining 22%, the computation time is slightly higher than $O(p \log p)$.

Moreover, it should be noted that discovering equivalent states allows one to limit the state space exploration. Potential reduction may also allow to analyse systems with infinite state spaces, or state spaces which are too large to fit into the computer's memory, and thus would have been intractable regardless of the computation time.

6 Petri net implementation

In this section, we will introduce a class of high-level Petri nets[4] for which the approach described above always works. More precisely, the syntactic restrictions imposed on these nets will imply Assumptions 1-3, and so for each net in this class the generated labelled transition system will satisfy all the requirements formulated in the general setting.

The kind of (finite) high-level Petri net we have in mind is a tuple $N \stackrel{\text{df}}{=} (S, T, \Lambda, M_0)$ which consists of a set S of *places*, a set T of *transitions* (disjoint from S), a *labelling* Λ of places, transitions and arcs (in $(S \times T) \cup (T \times S)$), and an *initial marking* M_0 (which is a mapping that associates to each $s \in S$ a finite multiset of values in $\Lambda(s)$) such that:

[4] We assume that the reader is familiar with the basic notions concerning high-level Petri nets [13].

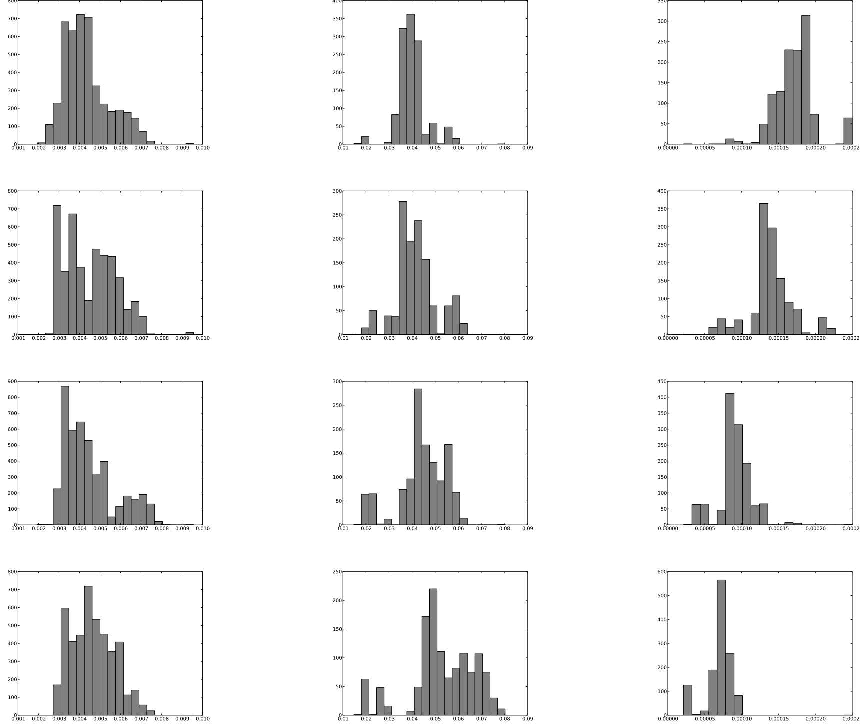


Figure 6: Left column: the j-th row from the top ($0 \leq j \leq 3$) shows the distribution of $\frac{t_i}{p_i}$ for $(p_i, t_i) \in D_1$ such that $\frac{j}{4}k_1 < i \leq \frac{j+1}{4}k_1$ (*i.e.*, the j-th segment of D_1). Middle and right columns: distributions for $\frac{t_i}{p_i \log p_i}$ and $\frac{t_i}{{p_i}^2}$ respectively, for $(p_i, t_i) \in D_2$.

1. For each place $s \in S$, the *type* of s, $\Lambda(s)$, is a Cartesian product $X_1 \times \cdots \times X_k$ ($k \geq 1$), where each X_i is $\mathbb{P}$ or $\mathbb{D}$.

2. The set of places of N contains a unique *generator* place s_η having the type $\mathbb{P} \times \mathbb{N}$. It is used by the underlying scheme to implement the mapping η that is needed when new threads are spawned. For each active thread identified by π, the generator place stores a token $\langle \pi, i \rangle$ where i is the integer *counter* of the child threads already spawned by π. Thus the next threads to be created by π will have the pids $\pi.(i+1)$, $\pi.(i+2)$, etc.

3. We assume that the initial marking M_0 of N is such that the generator place contains exactly one token, and all other places are empty. This implies that $\mathbb{I}$ is reduced to a single element.[5]

4. For each transition $t \in T$, the annotation on the arc from the generator place to t is a set of the form

$$\Lambda(s_\eta, t) \stackrel{\text{df}}{=} \{\langle p_1, c_1\rangle, \ldots, \langle p_k, c_k\rangle\}$$

   where $k \geq 0$ and all the 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 of the form:

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

   where $m \leq k$, and $n_j \geq 0$ for all j.[6] An empty arc annotation means that the arc is absent.
   Below we denote by Π_t the set of all the pid expressions $p_i.(c_i + j)$ used in $\Lambda(t, s_\eta)$.

5. 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 $\Pi_t \cup \{p_1, \ldots, p_m\}$.

6. For each transition $t \in T$, $\Lambda(t)$ is a computable Boolean expression, called the *guard* of t, build 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 Ω_{pid}.

[5] This is not a restriction in practice since any desired marking can be obtained by initially firing a special transition which inserts the required tokens into the places of the net N.

[6] Note that the first row $\langle p_1, c_1 + n_1\rangle, \ldots, \langle p_m, c_m + n_m\rangle$ corresponds to those pids $p_1, \ldots, p_k$ which remain active after the firing of t, and the remaining rows correspond to newly created pids. If $n_j = 0$ then the whole row $\langle p_j.(c_j + 1), 0\rangle, \ldots, \langle p_j.(c_j + n_j), 0\rangle$ is absent, *i.e.*, no child pids were created for p_j.

Notice that the transitions of such nets manipulate the pids in a controlled way. In particular, they may create new pids only from existing and active ones, and only following a precise creation scheme (ensured by the connection to the place s_η).

A *binding* β of a transition $t \in T$ is a mapping from the variables occurring in the guard of t and the annotations of arcs adjacent to t to concrete values of the corresponding types. We use $\beta(e)$ to denote the result of evaluating an expression e under the binding β. Then, t is *enabled* at a marking M if there is a binding β such that the following hold:

- for all $s \in S$, $\beta(\Lambda(s,t)) \leq M(s)$, *i.e.*, there are enough tokens in the input places of t;
- $\beta(\Lambda(t,s))$ is a multiset over $\Lambda(s)$, *i.e.*, the types of the output places of t are respected; and
- $\beta(\Lambda(t))$ evaluates to *true*.

Such an enabled transition t may *fire* producing a new marking M' such that, for all $s \in S$, we have:

$$M'(s) \stackrel{\text{df}}{=} M(s) - \beta(\Lambda(s,t)) + \beta(\Lambda(t,s)) \; .$$

We denote this by $M[t, \beta\rangle M'$. Then a marking M is *reachable* if it can be derived from M_0 by firing a finite sequence of transitions.

In [11] we also introduced a class of high-level Petri nets aimed at modelling systems with dynamic process creation. However, the syntactic rules there were over-complicated and so it was one of our objectives to simplify the constraints imposed on suitable Petri nets. The resulting class supports much more refined way of modelling systems.

We will now present results validating our claim that the just defined class of high-level Petri nets satisfies Assumptions 1-3. To start with, in this particular case, states correspond to reachable markings of N. For each such a marking M, the corresponding state $q_M \stackrel{\text{df}}{=} (\sigma_M, \eta_M)$ is given by:

$$\begin{array}{rcl} \sigma_M & \stackrel{\text{df}}{=} & \{(s, M(s)) \mid s \in S\} \\ \eta_M & \stackrel{\text{df}}{=} & \{\pi \mapsto k \mid \langle \pi, k \rangle \in M(s_\eta)\} \; , \end{array}$$

and the generated ct-configuration is $ctc_M \stackrel{\text{df}}{=} (G, H)$, where $H = \eta_M$ and G is the set of all pids occurring in the marking M. Note that η_M is a well defined function since no pid occurs more than once in s_η which can be

shown in a similar way as Theorem 2 (see the Appendix). As a result, q_M and ctc_M are also well defined.

We may now observe that Assumption 1 is guaranteed by the restrictions on the form of annotations on the arcs adjacent to a single transition (in particular items 4 and 5 in the definition above). As to the remaining two general assumptions, we have the following.

Theorem 2 (Petri net rendering of Assumption 2) *Let M be a reachable marking of N. Then, ctc_M is a ct-configuration.*

Let M and M' be reachable markings such that $q_M \sim_h q_{M'}$. Then the two markings are also *h-equivalent*, $M \sim_h M'$.

Theorem 3 (Petri net rendering of Assumption 3) *Let M and M' be h-equivalent reachable markings of N, and t be a transition such that $M[t, \beta\rangle\widetilde{M}$. Then $M'[t, h \circ \beta\rangle\widetilde{M}'$, 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. Moreover, the result still holds if Ω_{pid} is restricted to any of its subsets which includes pid equality.*

Both theorems are proved in the Appendix (note that neither of these proofs was previously published).

7 Comparison with other approaches

A variety of methods such as those in [7, 2, 9] (see, *e.g.*, [16] for a recent survey) address the state explosion problem by exploiting, in particular, system symmetries in order to avoid searching parts of the state space which are equivalent to those that have already been explored. Some methods, such as [1, 6, 15], have been implemented in widely used verification tools and proved to be successful in the analysis of, *e.g.*, communication protocols and distributed systems. The method proposed in this paper actually focuses on abstracting thread identifiers, while symmetries are addressed indirectly. So, in addition to reducing symmetric executions, it can also cope with infinite repetitive behaviours. Moreover, it should be noted that symmetry reduction techniques rely on the computation of a canonical representation of each state. It has been shown in [3] that this is as hard as checking graph isomorphism. So, our method, while capturing symmetry reductions, is not computationally more complex.

It is worth noting that our approach could be in principle combined with any framework that offers symmetry reductions. To this end, one would basically need to encode explicitly into the states of the model the information captured by the layered graphs we generate for checking states equivalence. In particular, this would require to encode and maintain the relation between active pids and the pids of their next siblings, as well as the comparisons considered in $\Omega^{\star}_{pid}$. This would greatly complicate the modelling but would ensure that symmetry discovery is consistent with the constraints we require in Assumptions 1-3.

When applied to Petri net markings, our approach can be more specifically related to that developed in [9] (which, incidentally, covers those in [7, 2]) where a general framework was proposed aimed at reduced state space exploration through the utilisation of symmetries in data types. More precisely, [9] defines three classes of primitive data types. Two of these, ordered (for which symmetries are not considered) and cyclic (which are finite) cannot lead to reductions based on pids. The remaining one, unordered, can in principle be used for reduction even for infinite domains. An interesting aspect of the approach developed in [9] is that, in principle, it allows a definition of data types taking into account various comparisons between pids. This could alleviate the encoding of the layer graphs within the model as suggested above. Only the encoding and maintaining of the relation between active pids and their next siblings would then be necessary.

Finally, any reduction method based on identifying equivalent Petri net markings falls into the category of *equivalence reduction* as defined in [8]. This method requires an equivalence specification to be consistent [8, def. 1], *i.e.*, preserved along the executions of a system, which in our case has been split over Assumptions 1-3. However, it is important to note that [8] defines a general notion but does not provide any help in finding suitable equivalence relations and proving their consistency with respect to state space reduction. On the contrary, it is even stressed in [8] that this is a particularly difficult task, which, still according to [8], explains why equivalence reductions are not often used and why less general symmetry reductions are preferred.

8 Concluding remarks

In this paper we presented an abstraction technique for generating reduced state space representations of dynamic multi-threaded computing systems, aimed specifically at alleviating problems resulting from dynamic process

creation. To make the new technique applicable to a wide range of different system models, we based it at the behavioural level on general labelled transition systems which have, in particular, to satisfy Assumptions 1–3. Such an approach is practical as labelled transition systems of the this kind can be obtained through introducing mild syntactic restrictions on the system model generating them, as shown in Section 6. That is, the introduced class of high-level Petri nets does generate labelled transition systems satisfying the conditions required in first part of this paper. What is more, similar restrictions can be introduced for other system models which support explicit thread creation and manipulation.

The proposed technique is based on an equivalence relation between system states. It takes into account new process identifiers which can be derived from those present in the states being compared, in effect performing a limited lookahead. We demonstrated that checking state equivalence can be implemented in a very efficient way by reusing well known algorithms for graph isomorphism and their publicly available implementations.

Acknowledgements

We would like to thank the reviewers for their constructive comments. In particular, we would like to thank the reviewer who encouraged us to evaluate more closely the approaches developed in [8] and [9]. This research was supported by the French ANR project AUTOCHEM, and the British projects RAE&EPSRC DAVAC and EPSRC VERDAD.

References

[1] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric Spin. *International Journal on Software Tools for Technology Transfer*, 4(1):92–106, 2002.

[2] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. A Symbolic Reachability Graph for Coloured Petri Nets. *Theoretical Computer Science*, 176:39–65, 1997.

[3] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry Reductions in Model Checking. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, vol. 1427 of *Lecture Notes in Computer Science*, pp. 147–158, 1998.

[4] 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*, 26:1367–1372, 2004

[5] A.Hagberg, D. Schult, P. Swart. NetworkX, High Productivity Software for Complex Networks. `http://networkx.lanl.gov`

[6] M. Hendriks, G. Behrmann, K.G. Larsen, P. Niebert, and F.W. Vaandrager. Adding Symmetry Reduction to UPPAAL. In *Proceedings of the First International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, vol. 2791 of *Lecture Notes in Computer Science*, pp. 46–59, 2004.

[7] C. Norris Ip and David L. Dill. Better Verification Through Symmetry. *Formal Methods in System Design*, 9:41–75, 1996.

[8] J. B. Jørgensen and L. M. Kristensen. Verification of Coloured Petri Nets Using State Spaces with Equivalence Classes. *LINCOM Studies in Computer Science*, 1:17–34, 2003.

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

[10] V. Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings.* PhD Thesis, School of Computing Science, University of Newcastle, 2003.

[11] H. Klaudel, M. Koutny, E. Pelz, and F. Pommereau. Towards Efficient Verification of Systems with Dynamic Process Creation. In *Proceedings of the 5th International Colloquium on Theoretical Aspects of Computing (ICTAC'08)*, vol. 5160 of *Lecture Notes in Computer Science*, pp. 186–200, 2008.

[12] H. Klaudel, M. Koutny, E. Pelz, and F. Pommereau. An Approach to State Space Reduction for Systems with Dynamic Process Creation. In *Proceedings of the 24th International Symposium on Computer and Information Sciences (ISCIS'09)*, pp. 543–548, 2009.

[13] K. Jensen and L.M. Kristensen *Coloured Petri Nets: Modelling and Validation of Concurrent Systems.* Springer, 2009.

[14] B. D. McKay. Practical Graph Isomorphism. *Congressus Numerantium*, 30: 45–87, 1981.

[15] K. McMillan. *Symbolic Model Checking.* Kluwer Academic, 1993.

[16] A. Miller, A. Donaldson, and M. Calder. Symmetry in Temporal Logic Model Checking. *ACM Computer Surveys*, 38(3):Article 8, 2006.

[17] W. Stein. Sage Mathematics Software. The Sage Group (2007) http://www.sagemath.org

A Additional results and proofs of Theorems 2 and 3

In what follows, the function H occurring in a ct-configuration $ctc = (G, H)$ is often treated as a set of pairs. We first observe that being a ct-configuration is unaffected by pid deletion.

Proposition 1 *Let (G, H) be a ct-configuration. If $dom(H) \subseteq G' \subseteq G$ and $H' \subseteq H$, then both (G', H) and (G, H') are ct-configurations.*

Proof: Follows directly from the definition of a ct-configuration. $\square$

The next result captures the change of a ct-configuration $ctc = (G, H)$ after the spawning of a single new thread. The idea is that we first select π in the domain of H and then replace the unique (π, i) in H by $(\pi, i + 1)$, and respectively add $\pi.(i + 1)$ and $(\pi.(i + 1), 0)$ to G and H, leading to:

$$ctc^{\pi} \stackrel{\text{df}}{=} \big(G \cup \{\pi.(i+1)\}\, ,\, H \setminus \{(\pi, i)\} \cup \{(\pi, i+1), (\pi.(i+1), 0)\,\} \big)\ .$$

In the proofs below we use CTC1 and CTC2 to respectively refer to the first and second part of the definition of a ct-configuration.

Proposition 2 *The ctc^{π} above is a ct-configuration. Moreover, $\pi.(i+1) \notin G$.*

Proof: Let $ctc^{\pi} = (G', H')$. We first observe that, by CTC2 for ctc,

$$\pi.(i + 1).\tau \notin G, \text{ for all } \tau. \qquad (*)$$

Hence the second part of the result holds. Moreover, by $(*)$ and CTC1 for ctc together with $dom(H') = dom(H) \cup \{\pi.(i + 1)\}$, we obtain that H' is a function.

It is clear that ctc^{π} satisfies CTC1 since ctc satisfies CTC1 and we have $G' = G \cup \{\pi.(i + 1)\}$ and $dom(H') = dom(H) \cup \{\pi.(i + 1)\}$.

To show that CTC2 is also satisfied, we proceed as follows. Since *ctc* satisfies CTC2, if the same is not true of ctc^{π}, then at least one of the following three cases must hold.

Case 1: $\pi.(i+1) = \pi'.k.\tau$ and $k > m$, for some $(\pi', m) \in H \setminus \{(\pi, i)\}$. If τ is empty, then $\pi = \pi'$, producing a contradiction with the choice of (π', m) and H being a function. Hence $\tau = \tau'.n$, and so we have that $\pi = \pi'.k.\tau' \in G$, producing a contradiction with CTC2 for *ctc*.

Case 2: $k > i+1$, for some $\pi.k.\tau \in G$. Then also $k > i$, producing a contradiction with CTC2 for *ctc*.

Case 3: $\pi.(i+1).\tau \in G$, for some τ. Then we immediately obtain a contradiction with $(*)$. □

We can now prove that Assumption 1 holds for the class of Petri nets introduced in this paper.

Sketch of the proof of Theorem 2: Follows from the fact that ctc_{M_0} is a ct-configuration and from the definition of annotations on arcs adjacent to a transition t. More precisely, they imply that if $M[t, \beta\rangle M'$ then $ctc_{M'}$ can be derived from ctc_M by zero or more applications of Proposition 2 followed by at most two applications of Proposition 1. □

We now turn to the properties of equivalent markings. In what follows, two ct-configurations such as in the definition of h-equivalent states will also be called *h-equivalent*. Moreover, $next(H) = \{\pi.(i+1) \mid (\pi, i) \in H\}$, for every ct-configuration (G, H). We can then show that h-equivalence of ct-configurations is preserved by coherent deletions of pids.

Proposition 3 *Let (G, H) and (G', H') be h-equivalent ct-configurations, and $dom(H) \subseteq \widetilde{G} \subseteq G$ and $\widetilde{H} \subseteq H$. Moreover, let $\widehat{G} = h(\widetilde{G})$, and let $\widehat{H}$ be obtained from H' by deleting each (π, i) such that $\pi \notin h(dom(\widetilde{H}))$.*

- *$(\widetilde{G}, H)$ and $(\widehat{G}, H')$ are $\widetilde{h}$-equivalent ct-configurations, where $\widetilde{h}$ is h restricted to $\widetilde{G} \cup next(H)$.*
- *$(G, \widetilde{H})$ and $(G', \widehat{H})$ are $\widetilde{h}$-equivalent ct-configurations, where $\widetilde{h}$ is h restricted to $G \cup next(\widetilde{H})$.*

Proof: By Proposition 1, $(\widetilde{G}, H)$, $(\widehat{G}, H')$, $(G, \widetilde{H})$ and $(G', \widehat{H})$ are all ct-configurations. Then the result holds as in each case the new bijection is a restriction of h. □

Proposition 4 *Let $ctc = (G, H)$ and $ctc' = (G', H')$ be h-equivalent ct-configurations, and let $(\pi, i) \in H$ and $(\pi', j) \in H'$ be such that $\pi' = h(\pi)$. Then $(G, H)^{\pi}$ and $(G', H')^{\pi'}$ are $\widetilde{h}$-equivalent ct-configurations, for some extension $\widetilde{h}$ of h.*

Proof: Let $\widetilde{ctc} = (\widetilde{G}, \widetilde{H}) = (G, H)^{\pi}$ and $\widehat{ctc} = (\widehat{G}, \widehat{H}) = (G', H')^{\pi'}$. From the assumptions we made and the fact that ctc and ctc' are h-equivalent, it follows that $h(\pi.(i+1)) = \pi'.(j+1)$, $\widetilde{G} = G \uplus \{\pi.(i+1)\}$ and $\widehat{G} = G' \uplus \{\pi'.(j+1)\}$ as well as:

$$\begin{aligned} next(\widetilde{H}) &= next(H) \setminus \{\pi.(i+1)\} \cup \{\pi.(i+2)\} \\ next(\widehat{H}) &= next(H') \setminus \{\pi'.(j+1)\} \cup \{\pi'.(j+2)\} . \end{aligned}$$

We then extend h to $\widetilde{h}$ by adding $\widetilde{h}(\pi.(i+2)) \stackrel{\text{df}}{=} \pi'.(j+2)$. Then $\widetilde{h}$ is a bijection which follows from $\pi.(i+2) \notin G$ and $\pi'.(j+2) \notin G'$ which in turn follows from CTC2 for ctc and ctc', respectively. One can easily check that $\widetilde{ctc}$ and $\widehat{ctc}$ are $\widetilde{h}$-equivalent ct-configuration. □

We can now prove that Assumption 3 holds for the class of Petri nets introduced in this paper.

Sketch of the proof of Theorem 3: We first observe that t with the binding $h \circ \beta$ is enabled at the marking M'. In particular, the guard of t evaluates to *true* (as in the case of β and M) since the pids can only be involved in comparisons using the operators in Ω_{pid} and the ct-configurations ctc_M and $ctc_{M'}$ are h-equivalent. We then observe that the existence of a suitable bijection $\widetilde{h}$ satisfying $\widetilde{M} \sim_{\widetilde{h}} \widetilde{M'}$ comes from zero or more applications of Proposition 4 followed by at most two applications of Proposition 3. □