An approach to state space reduction for systems with dynamic process creation

Hanna Klaudel, Maciej Koutny, Elisabeth Pelz, Franck Pommereau

An Approach to State Space Reduction for Systems with Dynamic Process Creation

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

(1) IBISC, FRE 3190 CNRS, Université d'Evry, 91000 Evry, France
(2) School of Computing Science, Newcastle University, NE1 7RU, United Kingdom
(3) LACL, Université Paris Est, 61 av. du général de Gaulle, 94010 Créteil, France

Abstract

Automated verification of dynamic multi-threaded computing systems can be adversely affected by problems relating to dynamic process creation. We therefore investigate — in a general setting of labelled transition systems – a way of reducing the state spaces of multi-threaded systems. At the heart of our method is a state equivalence, which may produce a finite representation of an infinite state system while still allowing to validate the relevant behavioural properties. We demonstrate the feasibility of the method through experiments involving the checking of the proposed state equivalence.

Key words: *multi-threaded systems, state equivalence, state space computation, state space reduction.*

1. Introduction

In a multi-threaded (or multi-process) 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 may accelerate 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 (*i.e.*, private or local) execution of each thread. Hence the exact identity of an identifier is basically irrelevant, and what only matters are the *relationships* between such identifiers. As a result, (sets of) identifiers may often be replaced by other (sets of) identifiers without changing the resulting execution in any essential way. This leads to the problem of identifying equivalent executions, which must be addressed by any reasonable verification and/or simulation approach to multi-threaded programming schemes.

We initiated our investigation in [9] using a specific syntactic system model, viz. high-level Petri nets, resulting in a proposal for identifying markings which are essentially equivalent from the point of view of dynamic process creation. In this paper we consider a much more general (in fact, model-independent or behavioural) framework of labelled transition systems. 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 [9]. We then provide experimental results about the effort needed to verify the state equivalence which reaffirm our initial hypothesis that this can be done in an efficient way.

A distinguishing feature of our 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 or indirect descendant of another thread. The proposed method easily adapts to what is usually allowed.

2. Multi-thread modelling

We denote by $\mathbb{D}$ the set of *data values*, and by $\mathbb{P} = \{\pi, \pi', \pi'', \ldots\}$ the 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 (below $\mathbb{I} \stackrel{\text{df}}{=} \{1, 2\}$). A possible way of implementing dynamic pids creation—and one adopted in this paper—is to consider them as finite sequences of positive integers (written down as dot-separated strings). It is not allowed to *decompose* a pid (*e.g.*, to extract the parent pid of a given pid) which is considered as an atomic value (a black box).

To compare pids, we use a set of binary relations on pids, $\Omega_{pid} \stackrel{\text{df}}{=} \{=, \lhd_1, \lhd, \pitchfork_1, \pitchfork\}$, defined as follows:

- $\pi \lhd_1 \pi'$ (which holds if there is a positive integer i such that $\pi.i = \pi'$) indicates whether π is the parent of π' (*i.e.*, thread π created thread π').
- $\pi \lhd \pi'$ indicates if π is an ancestor of π' (*i.e.*, $\lhd$ is $\lhd_1^+$).

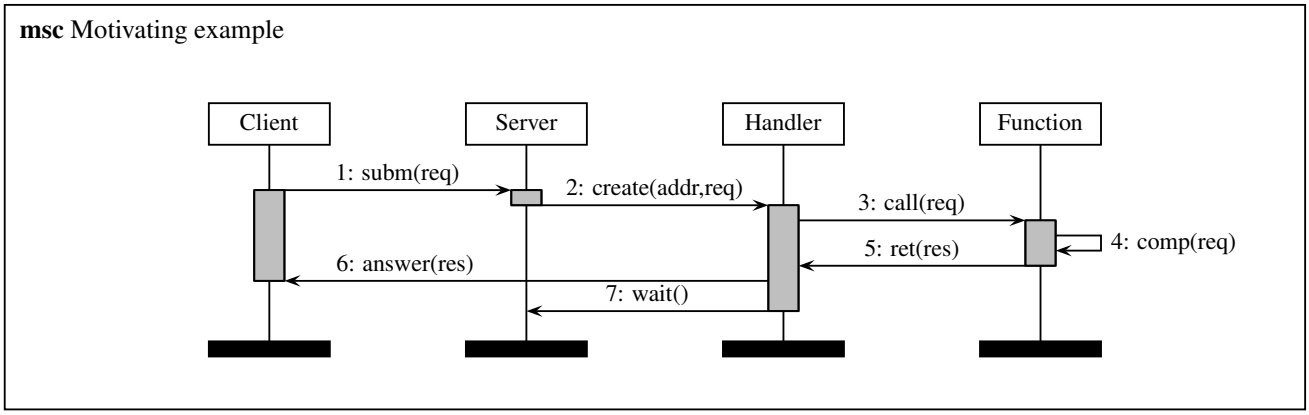

Figure 1. Motivating example: sequence diagram of a session

- $\pi \pitchfork_1 \pi'$ (which holds if there is a pid π'' and a positive integer i such that $\pi = \pi''.i$ and $\pi' = \pi''.(i+1)$) indicates whether 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').

- $\pi \pitchfork \pi'$ indicates whether π is an elder sibling of π' (*i.e.*, $\pitchfork$ is the transitive closure $\pitchfork_1^+$).

For example, pids 1.1 and 1.2 are siblings such that $1.1 \pitchfork_1$ 1.2 and both are children of 1 (*i.e.*, $1 \lhd_1 1.1$ and $1 \lhd_1 1.2$).

Consider a simple server system in which a bunch of threads are waiting for connections from clients requesting some calculation. Figure 1 shows a message sequence chart of a typical session. Whenever a new request arrives, a handler is created to process it. 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.

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.

Figure 2 shows an initial fragment of the state space with two alternative branches corresponding to the same scenario played by two different threads, tuples $\langle 1 \rangle$ and $\langle 2 \rangle$ in S. The client is not modelled. In this scenario, a server calculates whether a given year is a leap year. It receives a data $req = 2009$, creates a handler, and passes req and the client's address $addr$ to it. 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. Finally, the server terminates the handler.

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 local states $\mathbb{L}$, together with transitions between these states. In Figure 2: (i) $\mathbb{L} = \{S, H, F\}$ are the local states; (ii) 1, 2, 1.1, 1.2, 2.1, 2.2, 1.1.1 and 2.1.1 are pids; and (iii) 2009 and $addr$ are data items.

As usual, there is a distinguished *initial* state from which all other states can be reached. Each state $s \stackrel{\text{df}}{=} (\sigma_s, \eta_s)$ is composed of a pair of mappings: (i) a mapping σ_s from $\mathbb{L}$ to finite multisets of *vectors* (tuples) of data and pids (*e.g.*, in the initial node s_0, the local state at S is composed of two vectors: $\langle 1 \rangle$ and $\langle 2 \rangle$); and (ii) a mapping η_s which for each active thread (*i.e.*, one which belongs to the domain of η_s) gives the number of threads it has already created.

Each transition is labelled by the pid of the thread where it took place, and the specific name of an action or command (possibly internal) which effected it together with the actual parameters used.

When moving from a state s to s' along a transition t (or $s \xrightarrow{t} s'$), the pids present in s' as well as those involved in t must either be present in s or be newly created using the information provided by η_s. Similarly, any pid active at s' must be active in s or be a newly created one.

Looking at Figure 2 one can note that the two branches of execution, $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_5$ and $s_0 \rightarrow s'_1 \rightarrow s'_2 \rightarrow \ldots \rightarrow s'_5$, are basically equivalent since the roles of pids 1 and 2 can be swapped to obtain one from the other. Moreover, s_0 is equivalent to s_5 and s'_5. For example, s_5 is the same as s_0 except for the value of $\eta(1)$.

Before proceeding with a formal definition of state equivalence, we make an important observation that 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

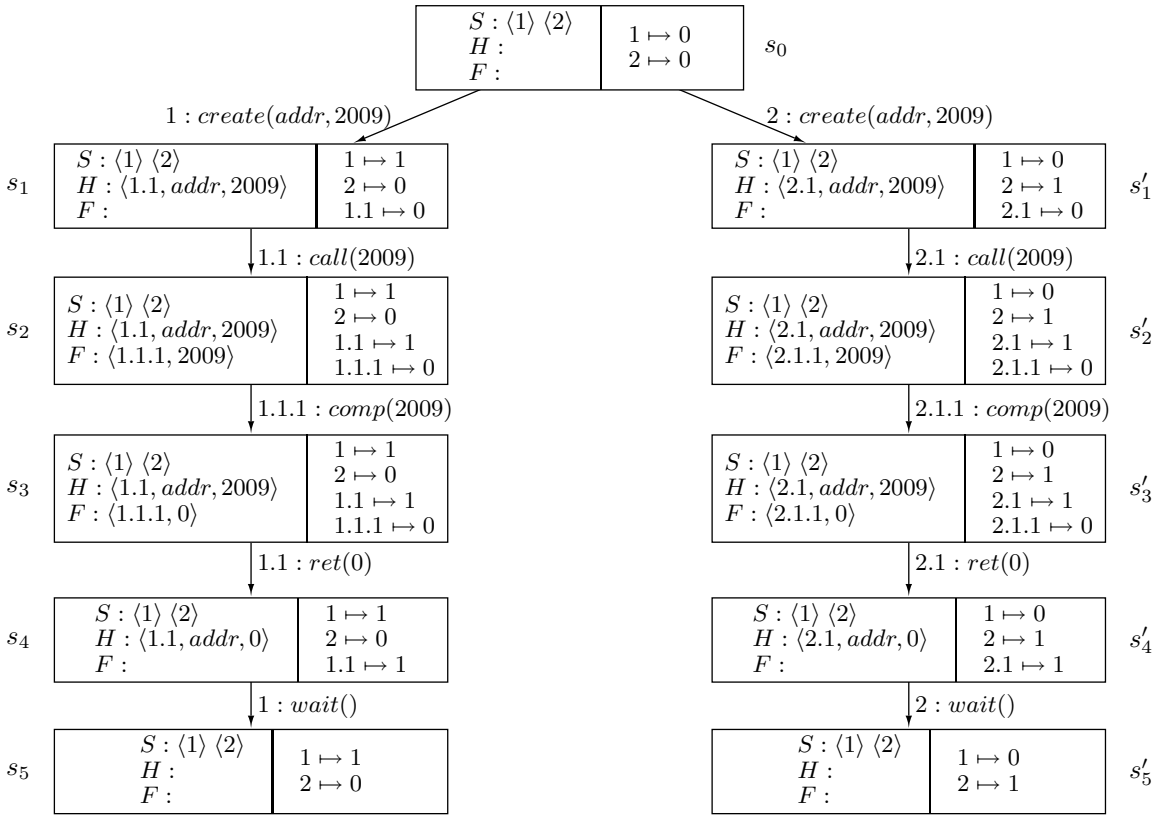

Figure 2. An initial fragment of the state space: σ_s is shown in the left part of each state and η_s in the right part

in σ_s can be derived as the next son of another active thread. More generally, we require that each state $s = (\sigma_s, \eta_s)$ defines what we call a *consistent thread configuration* (or ct-configuration): Assuming that pid_s is the set of all pids involved in σ_s, then: (i) $dom(\eta_s) \subseteq pid_s$, *i.e.*, each active pid is present in a local state; and (ii) for all $\pi \in dom(\eta_s)$ and $\pi' \in pid_s$, if $\pi.k$ is a prefix of π' then $k \leq \eta_s(\pi)$, *i.e.*, all pids in local states had been created in the past.

For each active pid π in s we denote the possibly next created pid by $next_s(\pi) \stackrel{\text{df}}{=} \pi.(\eta_s(\pi) + 1)$ and the set of next pids in s by $nextpid_s \stackrel{\text{df}}{=} \{next_s(\pi) \mid \pi \in dom(\eta_s)\}$.

Definition 3.1 (state equivalence) *Two states s and s' defining ct-configurations, are* equivalent *if there is a bijection $h : (pid_s \cup nextpid_s) \to (pid_{s'} \cup nextpid_{s'})$ such that $h(dom(\eta_s)) = dom(\eta_{s'})$ and for all $\prec \in \{\lhd_1, \lhd\}$ and $\curlywedge \in \{\pitchfork_1, \pitchfork\}$:*

- $\forall \pi \in dom(\eta_s)$, $h(next_s(\pi)) = next_{s'}(h(\pi))$.
- $\forall \pi, \pi' \in pid_s$: $\pi \prec \pi'$ *iff* $h(\pi) \prec h(\pi')$.
- $\forall \pi, \pi' \in pid_s \cup nextpid_s$: $\pi \curlywedge \pi'$ *iff* $h(\pi) \curlywedge h(\pi')$.
- $\sigma_{s'}$ *is* σ_s *after replacing each pid* π *by* $h(\pi)$.

We denote this by $s \sim_h s'$ or $s \sim s'$. $\diamond$

For state space reduction, the above equivalence should be preserved through possible executions, and so we make the following assumption.

Assumption 1 *If $s \sim_h s'$ and $s \xrightarrow{t} r$ then $s' \xrightarrow{h(t)} r'$ and $r \sim_{h'} r'$ where h' is suitably restricted (no newly terminated threads) and extended (created threads added) bijection. And similarly for $s' \xrightarrow{t'} r'$.* $\diamond$

The above assumption (which needs to be demonstrated for each concrete system model as, *e.g.*, in [9]) 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 relevant if one deals with properties of individual (abstracted) threads over sequences of states, making it compatible with the unfolding based verification technique [8].

It is important to see why we need the $nextpid$'s in the

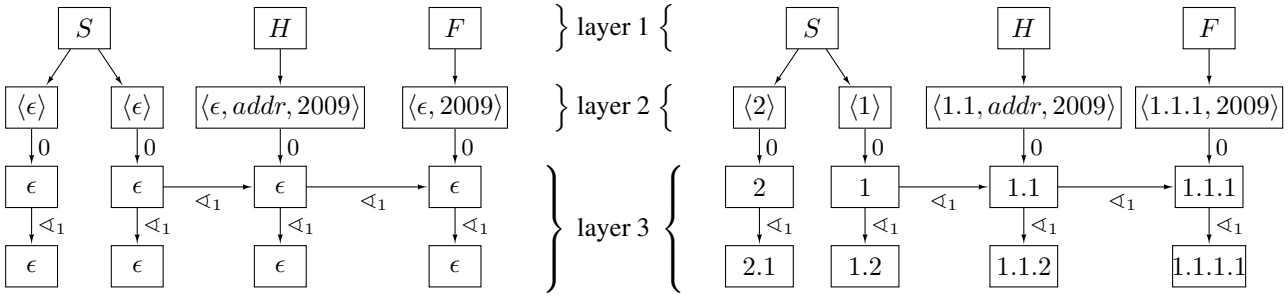

Figure 3. TLG of s_3 and its version with explicit pids (on the right) included to improve readability.

definition of equivalence. Assume that s and s' are such that $pid_s = \{1, 1.1\}$ and $\eta_s(1) = 1$ whereas $pid_{s'} = \{5, 5.1\}$ and $\eta_{s'}(5) = 3$. Then consider a bijection $h : pid_s \rightarrow pid_{s'}$ such that $h(1) = 5$ and $h(1.1) = 5.1$. Everything is fine as far as preserving the fatherhood/brotherhood 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 brotherhood relation of the corresponding pids and so the two new states are not equivalent.

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

TLGs are constructed as follows. The first layer is labelled by local states, the second layer by (abstracted) vectors and the third one by (abstracted) pids. The arcs are of two sorts: those going from the container object towards the contained object (local states contain vectors which contain pids), and those between the vertexes of the third layer reflecting the relationship between the corresponding pids through the comparisons in $\Omega^{\star}_{pid} \stackrel{\text{df}}{=} \{\lhd_1, \ldots, \lhd_k\}$, where $\Omega^{\star}_{pid}$ is Ω_{pid} without the equality relation, without relations that are transitive closures of other relations, 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 $\lfloor\,\rfloor : \mathbb{D} \cup \mathbb{P} \rightarrow \mathbb{D} \cup \{\varepsilon\}$ is defined as the identity on $\mathbb{D}$ and as a constant mapping ε on $\mathbb{P}$, extended component-wise to vectors of values.

Definition 3.2 (TLG) *Let $s \stackrel{\text{df}}{=} (\sigma_s, \eta_s)$ be a state of an LTS. Then the corresponding graph representation $TLG(s) \stackrel{\text{df}}{=} (V;\, A,\, A_{\lhd_1},\, \ldots,\, A_{\lhd_k};\, \lambda)$, where V is the set of vertexes, A, $A_{\lhd_1}$, …, $A_{\lhd_k}$ are sets of arcs and λ is a labelling on vertexes and arcs, is defined as follows:*
- First layer: for each local state $\ell \in \mathbb{L}$ such that $\sigma_s(\ell) \neq \varnothing$, ℓ is a vertex in V labelled by ℓ.
- Second layer: for each local state $\ell \in \mathbb{L}$ and for each vector $v \in \sigma_s(\ell)$, v is a vertex in V labelled by $\lfloor v \rfloor$ and $\ell \longrightarrow v$ is an unlabelled arc in A.
- Third layer:

- *for each vertex v at the second layer, 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 $v \stackrel{n}{\longrightarrow} \pi$ in A;*
- *for each active pid π, its potential next child, $next_s(\pi)$, is a vertex in V labelled by ε; and*
- *for all vertexes π, π' at this layer, for all $1 \leq j \leq k$, there is an arc $\pi \stackrel{\lhd_j}{\longrightarrow} \pi'$ 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 $G(s)$. ◇

In Figures 3 and 4, the S-, H- and F-labelled nodes belong to the first layer, the tuple labelled nodes belong to the second layer and the remaining nodes belong to the third layer.

Theorem 3.3 *Let s_1 and s_2 be two reachable states. Then $TLG(s_1)$ and $TLG(s_2)$ are isomorphic iff $s_1 \sim s_2$.*

Proof (sketch): For $i \in \{1, 2\}$, let us denote $TLG_i \stackrel{\text{df}}{=} TLG(s_i) = (W_i; A_i, A_{i_{\lhd_1}}, \ldots, A_{i_{\lhd_k}}; \lambda_i)$.

($\Rightarrow$) Let $h\colon W_1 \rightarrow W_2$ be an isomorphism between G_1 and G_2, such that $\forall v \in W_1,\ \lambda_1(v) = \lambda_2(h(v))$ and $\forall u, v \in W_1,\ \lambda_1((u, v)) = \lambda_2((h(u), h(v)))$. The states $s_i \stackrel{\text{df}}{=} (\sigma_{s_i}, \eta_{s_i})$ can easily be obtained from G_i: (i) vertexes of the second layer represent the vectors associated to localities, which are vertexes of the first layer; this allows one to obtain σ_{s_i}; and (ii) vertexes of the third layer which have no incoming arc from any vertex of the second layer allow to retrieve η_{s_i}, *i.e.*, for each such vertex $\pi.j \in W_i$, there is an active thread π in s_i and $\eta_{s_i}(\pi) = j - 1$. Moreover, all pids (present as vertexes of the third layer) are related through h, which is always the identity on data. So, the marking equivalence follows.

($\Leftarrow$) By Definition 3.2, h-equivalent states s_i generate graph representations G_i which only differ by the identity of some vertexes (their number, arcs and labelling being identical). By definition of the state equivalence, h is the identity on data and relates pids between s_1 and s_2. So h relates in the same way the identities of vertexes in W_1 and W_2. □

In our motivating example, the brotherhood 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 $\Uparrow_1$ 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 TLGs of s_3 and s'_3 are clearly isomorphic (see Figure 3). Thus, only one of the two execution paths would be represented, which shows how symmetric executions can be identified. Let us first consider the initial state s_0 whose TLG is represented in Figure 4. It is easy to check that the TLGs of s_5 and s'_5 are isomorphic to this one. Thus, the infinite execution that is possible in each branch can be reduced to a loop because we consistently abstract away the information about newly generated pids.

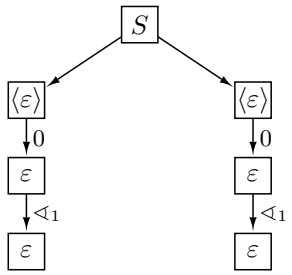

Figure 4. TLG of s_0.

Note that considering also the brotherhood relation for $TLG(s_3)$ and $TLG(s'_3)$ would add one extra arc from the node of pid 1 towards that of pid 2. This would result in losing the isomorphism of the two TLGs. Consequently, in order to achieve the highest reduction rate, it is important to keep in $\Omega^{\star}_{pid}$ only the relations actually used by the system generating an LTS.

4 Experimental results

The complexity of checking graph isomorphism is, in general, not known. In order to evaluate how efficient in practice it turns out to be in our case, we generated random states and measured the time spent on checking isomorphism of the corresponding graphs. We only considered pairs of graphs where the same local states are non-empty. Indeed, pairs that do not match this way can be checked for non-equivalence in linear time with respect to the number of local states, with no need for checking graph isomorphism.

Our methodology has been to generate random states and compare each one with similar ones obtained through all possible transformations based on: (i) pids swapping preserving isomorphism; (ii) shuffling of vectors throughout local states; (iii) shuffling of vector components; (iv) exchange of components between vectors (pids, data or both); (iv) data replacement; (v) pids replacement; and (vi) shifting of integers that compose pids. In order to check graph isomorphism, we used NetworkX [4] that implements the VF2 [3] algorithm.

We carried out more than two millions of comparisons, on the basis of which it may be observed that: (i) computation time is growing with the state and with the percentage of pids in vectors (with respect to data); while (ii) computation time improves with the increase of distinct data values. None of these facts should really be surprising. In particular, the presence of data leads to labelled nodes that can be quickly matched when comparing two graphs. Moreover, we observed that the general evolution of computation time is linear with respect to $p^{2/3}$ where p is the number of distinct pids in the compared states. With respect to p^2, the evolution of time looks clearly sublinear. Hence it is justifiable to estimate that computation time is in mean polynomial (around $p^{3/2}$) with respect to the number of threads in the observed system. Finally, we determined that the computation time looks slightly less than quadratic with respect to the number of vectors in compared states.

As a result, we conclude that the cost of checking state equivalence is very low for states that are different (most of the compared pairs) and stays good for states that are similar. Moreover, it should be noted that discovering equivalent states allows to limit the state space exploration. Thus the time spent in computing equivalences is probably less than the time required to computed the whole state space, which obviously holds for infinite state spaces reduced to finite ones. This reduction may also allow to analyze systems whose complete state space would simply not fit into the computer's memory, and thus would have been intractable regardless of the computation time.

5 Comparison with other approaches

A variety of methods such as those in [2, 6, 7] (see, *e.g.*, [12] for a recent survey) address the state explosion problem by exploiting, in particular, symmetries in the system 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, 5, 11], have been implemented in widely used verification tools and proved to be successful in the analysis of numerous communication protocols and distributed systems. The method proposed in this paper actually focuses on abstracting thread identifiers and symmetries are addressed only indirectly.

Our approach appears to be closely related to that developed in [7] (which, incidentally, covers those in [6, 2]) where a general framework was proposed aiming at reduced state space exploration through the utilisation of symmetries in data types. More precisely, [7] 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 an infinite domain (Sect.7.5 in [7]). However, the approach proposed in [7] does not cover the case of dynamic process creation as considered in this paper. First of all, our approach requires to establish the crucial behavioural consistency condition formulated as Assumption 1 by restricting the kind of global states (through the notion of ct-configuration) for which our consistency condition can work. Moreover, when comparing two states for equivalence, the approach of [7] considers only data actually present in the states. In our case, however, we have to look at additional data (potential sons of the pids present in the states) to ensure the preservation of equivalence over possible behaviours. We believe that approaches like [7] are insufficient for dealing with dynamic process creation and checking relationships between pids such as the brotherhood.

6 Concluding remarks

The presentation in this paper is based at the behavioural level of labelled transition systems which have, in particular, to satisfy Assumption 1. One might question, therefore, the practical usefulness of our approach. Fortunately, as it turns out, labelled transition systems of the 'right' kind can be ensured through introducing mild syntactic restrictions on the system model generating them. For example, in [9] we introduced a class of high-level Petri nets which generate labelled transition systems satisfying the conditions required in this paper. It is our conjecture that this situation is not unique and many other system models which support explicit thread creation and manipulation will lead to the same view.

In our future work we intend to carry out experimentation using the Sage [13] tool that implements the Nauty [10] algorithm which is often acknowledged as the fastest existing algorithm for graph isomorphism. Moreover, some case studies should be performed in order to be able to measure the concrete impact on verification efficiency. In a longer term, we plan to investigate the possibility of merging together the treatments of ordered and cyclic data types of [7] together with our approach.

Acknowledgement

This research was partially supported by: the Davac project, NSFC grant 60433010, and the SPREADS ANR project.

References

[1] D. Bosnacki, D. Dams and L. Holenderski. *Symmetric Spin.* Int. J. Soft. Tools Technol. Transfer 4(1)65-80, 2002.

[2] G. Chiola, C. Dutheillet, G. Franceschini and S. Haddad. *On Well-Formed Coloured Nets and their Symbolic Reachability Graph.* High-Level Petri Nets, Theory and Application, LNCS, Springer 1991.

[3] L.P. Cordella, P. Foggia, C. Sansone, M. Vento. *A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs.* IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, no. 10, 2004.

[4] A. Hagberg, D. Schult and P. Swart. *NetworkX, High productivity software for complex networks.* `http://networkx.lanl.gov`

[5] M. Hendriks, G. Behrmann, K. Larsen, P. Niebert and F. Vaandrager. *Adding symmetry reduction to UPPAAL.* FORMATS'03, LNCS 2791. Springer, 2003.

[6] C.N. Ip and D.L. Dill. *Better verification through symmetry.* FMSD 9, 1996.

[7] T. Junttila. *On the symmetry reduction method for Petri nets and similar formalisms.* PhD Thesis, HUT, Espoo, Finland, 2003.

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

[9] H. Klaudel, M. Koutny, E. Pelz and F. Pommereau. *Towards efficient verification of systems with dynamic process creation.* ICTAC 2008, LNCS 5160. Springer, 2008. Full version: LACL Technical report, 2008. `http://lacl.univ-paris12.fr/pommereau/publis/2008-lacl.html`

[10] B. D. McKay. *Practical Graph Isomorphism.* Congressus Numerantium 30, 1981.

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

[12] A. Miller, A. Donaldson and M. Calder. *Symmetry in temporal logic model checking.* ACM Comput. Surv., 38(3)8. ACM, 2006.

[13] W. Stein. *Sage Mathematics Software.* The Sage Group, 2007. `http://www.sagemath.org`