Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Distributed Graph-Based State Space Generation

Stefan Blom and Gijs Kant and Arend Rensink

12 pages

Guest Editors: Juan de Lara, Daniel Varro
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Distributed Graph-Based State Space Generation

Stefan Blom and Gijs Kant and Arend Rensink*

s.c.c.blom@utwente.nl kant@cs.utwente.nl rensink@cs.utwente.nl
Formal Methods and Tools Group
Department of Computer Science
University of Twente, The Netherlands

Abstract: LTSMIN provides a framework in which state space generation can be distributed easily over many cores on a single compute node, as well as over multiple compute nodes. The tool works on the basis of a vector representation of the states; the individual cores are assigned the task of computing all successors of states that are sent to them. In this paper we show how this framework can be applied in the case where states are essentially graphs interpreted up to isomorphism, such as the ones we have been studying for GROOVE. This involves developing a suitable vector representation for a canonical form of those graphs. The canonical forms are computed using a third tool called BLISS.

We show that the time performance of the resulting system scales well (i.e., close to linear) with the number of cores. We also report surprising statistics on the memory consumption, which imply that the vector representation used to store graphs in LTSMIN is more compact than the representation used in GROOVE.

Keywords: Graph Transformation, Symmetry Reduction, State Space Generation, Distributed Computing, GROOVE, LTSMIN

1 Introduction

For the last two years, the development in modern computer processors has been to put more cores on a single processor, rather than to speed up individual cores. To benefit from this development, it is therefore important to find ways to utilise the power of parallel processing. So far, there is no general way to achieve this for arbitrary applications.

In the context of graph transformation, this topic has been investigated by Bergmann et al. in [BRV09] for the tool VIATRA2. The core functionality of VIATRA2 is to compute a sequence of transformations, controlled by a predefined set of rules, as fast as possible. The paper proposes parallellisation of the matching algorithm.

In this paper, we address parallellisation of GROOVE [Ren04], which differs from other graph transformation tools in that it aims at *complete state space exploration* for a given set of rules, rather than computing a single sequence — where a state equates to a graph. One of the most important aspects of GROOVE, furthermore, is that states are compared *modulo isomorphism*; that is, two graphs are considered to represent the same state if they are isomorphic. Though

* any projects?

checking graph isomorphism is thought to be non-polynomial, the resulting reduction in state space size can more than make up for the cost of isomorphism checking; see, e.g., [CPR08].

At the core of our solution lies the LTSMIN framework [BPW10], which is specifically designed to enable distributed state space exploration. To use LTSMIN, an application has to:

1. Provide a serialisation of states in the form of fixed-length *state vectors*. State vectors are minimised to so-called *index vectors* (see [BLPW08]), which can be efficiently stored and transmitted.
2. Be able to generate all successors of a given source state, where both the source state and the successor states are communicated in the form of such a state vector.

LTSMIN will then run parallel copies of this application on every available core; the copies communicate using message passing, so that this works equally well with parallel and distributed cores. This method of parallellisation is particularly promising for GROOVE because the time-intensive step of isomorphism checking is done concurrently for many states.

In the case of GROOVE, Step 2 is present by default, but Step 1 is challenging. It is not enough to "flatten" graphs to a vector representation of some kind: in order to reduce the state space up to graph isomorphism, we have to make sure that the representative vector is the same for isomorphic graphs. For this purpose, we can make use of an existing tool called BLISS [JK07], which computes *canonical graphs* based on the principles developed in [McK81]. The LTSMIN vector representing a graph is thus the "flattening" of its canonical form.

We have experimented with this combination of LTSMIN, GROOVE and BLISS. In this paper we report two results:

- For larger cases, the time performance of the parallellised system scales well (though not linearly) with the number of processors. On a single core the setup is a good deal less efficient than GROOVE, but a system with eight or more cores easily outperforms the stand-alone version of GROOVE.
- Given a good vectorisation of the canonical form, the memory performance of GROOVE is also a good deal better than that of the stand-alone version. This is surprising given the fact that, in contrast to GROOVE, the data structures that LTSMIN uses in its tree compression and central state store are not at all optimised towards the storage of graphs. The gain is large enough to make us consider moving to the compressed vector representation even in the stand-alone, sequential version.

We introduce GROOVE in Section 2 and the relevant features of the LTSMIN framework in Section 3, especially the canonical graph vector representation. In Section 4 we report and analyze the outcome of the experiments. Section 5 draws conclusions and discusses future work.

2 Graph-based state space generation

Graph transformation is a declarative formalism, based on a set of *rules* that are applied to *graphs*. In the context of this paper, graphs are edge-labelled, with labels drawn from a global set Lab; moreover, nodes are drawn either from a set of node identities Node, or from the set of primitive data values $\mathsf{Val} = \mathsf{Bool} \cup \mathsf{Int} \cup \mathsf{Real} \cup \mathsf{String}$.

Definition 1 (graph, isomorphism) A graph G is a tuple $\langle V, E \rangle$ where $V \subseteq \mathsf{Node}$ is a finite set of nodes and $E \subseteq V \times \mathsf{Lab} \times (V \cup \mathsf{Val})$ is a finite set of edges. We use $src(e)$, $tgt(e)$ and $lab(e)$ to denoted the source, target and label of an edge e. The set of all graphs is denoted Graph. Graphs G, H are *isomorphic*, denoted $G \cong H$, if there exists a bijection $f : V_G \to V_H$ such that $(f(v), a, \bar{f}(w)) \in E_H$ if and only if $(v, a, w) \in E_G$, where $\bar{f} = f \cup id_{\mathsf{Val}}$. We sometimes write $f(G) = H$.

There is no need to precisely define rules; we merely formalise their actions upon graphs. A rule is an object r that can be applied to a *host graph* G if there exists a so-called *match* m for r in G (not formalised here, either). The rule and the match together determine a transformation of G, formally expressed by a derivation relation $G \xrightarrow{r,m} H$, where H is called the *target graph*. This derivation relation is well-defined and deterministic modulo isomorphism:

- $G \xrightarrow{r,m} H$ and $G' \cong G$ implies $G \xrightarrow{r,m} H'$ for some $H' \cong H$.
- $G \xrightarrow{r,m} H_1$ and $G \xrightarrow{r,m} H_2$ implies $H_1 \cong H_2$.

Using these concepts we define the graph transition system generated by a set of rules.

Definition 2 (graph transition system) The graph transition system (GTS) for a set of rules R and a start graph S is given by $\langle Q, \to, S \rangle$, where $\to$ is the derivation relation restricted to Q, and Q is the smallest set of graphs such that (i) $S \in Q$, and (ii) $H \in Q$ for all $G \in Q$, $r \in R$ and $G \xrightarrow{r,m} H$.

The GTS is a labelled transition system as used in many verification methods, in particular model checking [BK09]. Unfortunately, the GTS can easily be infinite, and even when finite can grow extremely large even for small start graphs — a phenomenon called *state space explosion*. One way to combat state space explosion is through *symmetry reduction* (see, e.g., [CJEF96]). In the case of graphs, symmetries show up as isomorphisms; the state space can be reduced by collapsing all isomorphic states, or in other words, taking the quotient of the GTS under $\cong$. The following algorithm generates this quotient $\langle Q, T, S \rangle$ (where T is the set of transitions).

```
1  let Q := {S},  T := ∅,  F := {S}        (F is the collection of fresh states)
2  while F ≠ ∅
3  do choose G ∈ F        (which G is chosen depends on the structure of F)
4     let F := F \ {G}
5     for G --r,m--> H
6     do if ∃H' ∈ Q : H' ≅ H
7        then let H := H'
8        else let Q := Q ∪ {H},  F := F ∪ {H}
9        endif
10       let T := T ∪ {(G, r, m, H)}
11    endfor
12 endwhile
```

The crux is in Line 6, which tests for *membership up to isomorphism*: given a graph H and a set of graphs Q, find $H' \in Q$ such that $H' \cong H$. Testing $H' \cong H$ for given graphs H, H' is believed to be non-polynomial in $|H|$ (see [Wei02]), and clearly membership up to isomorphism generalises the

pairwise test. However, we have shown in [Ren07, CPR08] that the gain by symmetry reduction can be huge, and hence can be worthwhile despite its complexity. We now discuss two ways to implement membership modulo isomorphism.

Graph certificates. The current implementation of GROOVE, as reported in [Ren07], uses *certificates* to obtain a data structure for Q allowing a membership-up-to-isomorphism test that performs well in many practical cases.

A *node certifier* is a function $nc\colon \mathsf{Graph} \to \mathsf{Node} \rightharpoonup \mathsf{Nat}$, which for every graph G results in a function $nc_G\colon V_G \to \mathsf{Nat}$ with the property that $nc_G = nc_H \circ f$ for all isomorphisms f from G to H. A *graph certifier* is a function $gc\colon \mathsf{Graph} \to \mathsf{Nat}$ such that $G \cong H$ implies $gc(G) = gc(H)$. An easy example of a node certifier is to count the number of incident edges ($nc_G\colon v \mapsto |\{e \in E_G \mid src(e) = v \vee tgt(e) = v\}|$ for all $v \in V_G$). Every node certifier nc gives rise to a graph certifier $gc\colon G \mapsto \sum_{v \in V_G} nc(v)$.

GROOVE currently implements Q as a map $\mathsf{Nat} \to \mathbf{2}^{\mathsf{Graph}}$ such that $n \mapsto \{G \in Q \mid n = gc(G)\}$. Finding $H' \in Q$ such that $H' \cong H$ comes down to searching $Q(gc(H))$, which for a good graph certifier gc is almost always either empty or a singleton set. Moreover, pairwise testing $H' \cong H$ for the $H' \in Q(gc(H))$ is made easier by using a node certifier.

Canonical forms. One can take the idea of graph certifiers one step further by also requiring that $gc(G) = gc(H)$ implies $G \cong H$. This is the idea behind the concept of *canonical forms*.

A *graph canoniser* is a function $can\colon \mathsf{Graph} \to \mathsf{Node} \rightharpoonup \mathsf{Node}$, which for every graph G results in an injective function $can_G\colon V_G \to \mathsf{Node}$ such that $G \cong H$ if and only if $can_G(G) = can_H(H)$. (Note that, in combination with a hash function $hash\colon \mathsf{Node} \to \mathsf{Nat}$, this gives rise to a node certifier $nc = hash \circ can$.) Q can then be implemented as a set of canonical form graphs. Obviously, computing canonical forms is as complex as testing for isomorphism; nevertheless, in practice the complexity often turns out to be bearable. In particular, the algorithm developed by McKay [McK81] as implemented in the tools NAUTY [McK09] and BLISS [JK07] does well in practice.

We have used BLISS in our experimentation. There is a discrepancy in that BLISS uses node-labelled rather than edge-labelled graphs; however, our graphs can be converted to BLISS graphs without loss of information, though with a slight blowup due to the need to encode edge labels in some way. Another noteworthy property is that the canonical forms produced by BLISS always map to an initial fragment of Nat; that is, $can_G(V_G) = \{0, \ldots, |V_G| - 1\}$ for all graphs G.

3 The LTSMIN framework

LTSMIN is meant to be used as a module in a tool chain that enables state space generation on a parallel or distributed systems, consisting of many independent cores. The modular design ensures that the framework can be used for a variety of formalisms. The communication between LTSMIN and the application that uses it, hereafter called the *user module*, is through an interface called PINS, for *Partitioned Next-State function*. We will briefly explain the underlying concepts.

To run an application on top of the LTSMIN framework, an LTSMIN client as well as a copy of the user module is started up in parallel on every core. These copies communicate by message passing, so that it does not matter (from the protocol view) whether cores are on a single machine or distributed over different machines. State space exploration then proceeds as follows:

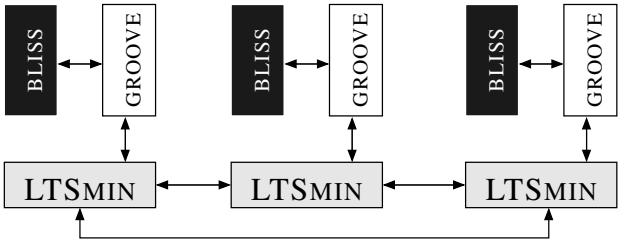


Figure 1: 3-core configuration of LTSMIN with GROOVE +BLISS as user module.

- LTSMIN defines a function that associates a fixed core with each state, on the basis of the state's vector representation. When a state is generated, it is sent to the associated core for further processing. The exploration is kicked off by sending the initial state to the appropriate core.
- Each core keeps a store of all states sent to it so far, remembering also whether the states are closed (i.e., already fully explored) or fresh.
- Upon reception of a state, a core adds it to its state store, marking it as fresh if it was not already in the store.
- Each core computes the successor of each fresh state, and sends the successors to their associated cores. This computation is done by the user module.

An example configuration with GROOVE and BLISS is depicted schematically in Figure 1.

3.1 State vectors and tree compression

The central concept enabling the modularity of LTSMIN is the *state vector*. Every state has to be presented as a vector $\langle p_1, \ldots, p_n \rangle$ for fixed n. The nature of the elements p_i actually does not matter, as these are immediately mapped to table indices for each position. That is, for $i = 1, \ldots, n$ LTSMIN builds up an injective mapping $t_i : P_i \to \mathsf{Nat}$, where P_i is the set of all values encountered so far at position i and Nat is a finite fragment of natural numbers; e.g., that fragment which can be represented in 32 bits. Every state vector $\langle p_1, \ldots, p_n \rangle$ is then converted to an *index vector* $\langle t_1(p_1), \ldots, t_n(p_n) \rangle \in \mathsf{Nat}^n$. The function associating a core with each state is computed as a hash on the index vector, modulo the number of available cores.

The tables $(t_i)_{1 \le i \le n}$ are duplicated in the system. It is essential that all workers use the same tables, but they are impossible to build beforehand, as it is unknown which values will be encountered at each position. For this reason, the LTSMIN framework also has the task of distributing the tables over the workers, which in turn means that all the t_i are replicated over all LTSMIN clients. On the other hand, the tables also need to be known on the side of the user module, since this is where the coding of state vectors to and from index vectors actually takes place. Thus, in a system with c cores, all t_i are replicated $2c$ times.[1]

Index vectors are further compressed using so-called *tree compression* (see [BLPW08]): without going into details, this comes down to repeatedly grouping neighbouring positions of the

[1] This description is actually still slightly simplified with respect to the implementation: there the encoding of the *outgoing* states may be different from that of the *incoming* ones; the former is then local to each core, and the LTSMIN clients translate the local to the global encoding.

index vector and building a new table of all combinations of values at those positions that are found during exploration. All these tables together form what is called the *leaf database*.

The success of the method crucially depends on finding a state vector representation that has as few values at each vector position as possible; i.e., each of the P_i should be small. This does not contradict a huge overall state space size: for $\max_i |P_i| = m$, the number of states that can potentially be represented is m^n. In the worst case, for one or more i $|P_i|$ approaches the total state space size, and hence so does the size of the leaf database; the advantage of this compression method is then completely lost.

3.2 Serialising canonical form graphs

We will now describe the steps necessary to use GROOVE as a user module in the LTSMIN framework. The main difficulty is to find a suitable state vector representation. This is entirely up to the user module: LTSMIN gets to see the state vectors only after they have been produced, and treats the values in the P_i as completely unstructured.

It is absolutely necessary that the state vectors uniquely represent states. This means that, if we want to benefit from symmetry reduction, we have to put graphs into canonical form *before* communicating them to LTSMIN. Moreover, as explained above, the vector representation should ideally have only few possible values at each slot.

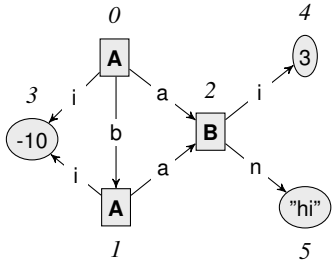
In Section 2 we have explained that the canonical form computed by BLISS essentially assigns a sequence number from 0 to $|V|-1$ to each node of a graph G. This imposes a total ordering $\leq$ on V; we will use $\vec{v} = v_0 \cdots v_k$ to denote the ordered sequence of nodes in V. Furthermore, we use the natural total orders on the primitive values Int, String, Bool and Real and we assume a total order on Lab (for instance, the alphabetical ordering). This also gives rise to a lexicographical ordering on edges. In the sequel, $ord(X)$ for a set X with an implicit order will denote the ordered vector of X-elements, and $\vec{x} \restriction_I$ for a sequence $x \in X^*$ and an index set $I \subseteq \{0,\ldots,|\vec{x}|-1\}$ will denote the sequence of elements at positions I.

The vector $\vec{p}_G$ representing G will consist of n slots, of which the first contains a sequence of *node colours*, one for each node, in the order imposed by the canonical form; the second to fifth contain the sets of primitive values from Val used as target nodes, seperated per primitive type; and the remaining slots contain outgoing edges for the individual nodes. If $k > n-5$ (where $k = |V_G|$ and $n = |\vec{p}|$) then nodes are "wrapped around", e.g. for $n = 12$ slot p_5 would be used for $v_0, v_7, v_{14}, \ldots$. Formally this is defined by

$$p_i = \begin{cases} color_G(v_0)\cdots color_G(v_k) & \text{if } i = 0 \\ \mathsf{X}_G & \text{if } i = 1+j \text{ and } \mathsf{X} = (\mathsf{Int}\ \mathsf{String}\ \mathsf{Bool}\ \mathsf{Real}) \restriction_j \\ out_G(w_0)\cdots out_G(w_m) & \text{if } i = 5+j \text{ and } \vec{w} = \vec{v} \restriction \{l \mid l = j \bmod (n-5)\} \end{cases}$$

where $color_G(v)$ denotes the colour and $out_G(v)$ the outgoing edges of v, defined as follows:

$$\begin{aligned} out_G(v) &= \{(a, can_G(w)) \mid (v,a,w) \in E_G, v \neq w\} \\ self_G(v) &= \{a \mid (v,a,v) \in E_G\} \\ \mathsf{X}_G &= ord(\mathsf{X} \cap tgt(E_G)) \qquad \text{for } \mathsf{X} = \mathsf{Int}, \mathsf{String}, \mathsf{Bool}, \mathsf{Real} \\ color_G(v) &= \begin{cases} self_G(v) & \text{if } v \in \mathsf{Node} \\ (\mathsf{X}, i) & \text{if } v = \mathsf{X}_G \restriction_i \end{cases} \end{aligned}$$



$p_0 = \{\mathbf{A}\}\ \{\mathbf{A}\}\ \{\mathbf{B}\}\ (\mathsf{Int},1)\ (\mathsf{Int},0)\ (\mathsf{String},0)$ list of nodes
$p_1 = -10\ 3$ list of Ints
$p_2 =$ "hi" list of Strings
$p_3 = \varepsilon$ (the empty sequence) lisf of Bools
$p_4 = \varepsilon$ lisf of Reals
$p_5 = \{(\mathsf{a},2),(\mathsf{b},1),(\mathsf{i},4)\}\ \emptyset$ edges of *0* and *4*
$p_6 = \{(\mathsf{a},2),(\mathsf{i},4)\}\ \emptyset$ edges of *1* and *5*
$p_7 = \{(\mathsf{i},3),(\mathsf{n},5)\}$ edges of *2*
$p_8 = \emptyset$ edges of *3*

Figure 2: An example graph with $|V| = 6$, represented by a state vector with $|\vec{p}| = 9$. The canonical node numbers are in italic. Node labels **A**, **B** are self-edges; oval nodes are data values.

An example state vector is shown in Figure 2. As related above, this is translated to an index vector together with a set of tables $t_0, \ldots, t_n$, so that a value at position i which recurs in another state vector at the same position is encoded by the same index. For instance, if the graph in Figure 2 is modified by $i := 3$ in node 0, only slot 5 of the state and index vectors would change (namely to $\{(\mathsf{a},2),(\mathsf{b},1),(\mathsf{i},3)\}\ \emptyset$) and only t_5 might have to be updated with this new value.

4 The experiments

We have carried out experiments based on three rule systems with varying characteristics.

le A leader election protocol. In this case there is a fixed number of nodes representing network nodes and a varying number of nodes representing messages. The number of nodes is an upper limit on the number of messages. This case study has been used for the GraBaTs 2009 tool contest (see http://is.tm.tue.nl/staff/pvgorp/events/grabats2009).

unflagged-platoon A protocol for forming car platoons. In this model there is always a fixed number of nodes. The behaviour shows extensive symmetries; reduction modulo isomorphism shrinks the state space by many orders of magnitude. This case study has been used for the Transformation Tool Contest 2010 (see http://planet-research20.org/ttc2010).

append A model of list appenders that concurrently add a value to the same list. In this case the number of nodes grows in each step. The maximal number of nodes equals the number of appenders plus 1 times the number of elements in the list. There is hardly any nontrivial isomorphism in the transition system.

The experiments have been performed on a cluster consisting of 8 compute nodes with 4 dual Intel E5520 CPUs each and 24GB RAM, for a total of 8 cores per compute node and 64 cores in total. GROOVE 4.0.1 has been used with a Sun Java 1.6.0 64-bit VM with a maximum of 2GB of memory for each core. For computing canonical forms we used BLISS 0.50. We used LTSMIN 1.5, with an added dataflow module to facilitate the communication between LTSMIN and GROOVE. For all experiments, the combined system was given a time limit of 4 hours. The

Table 1: Results for the largest start graphs where both GROOVE (sequential) and LTSMIN (1, 8 and 64 cores) were able to generate the state-space. The memory usage shown is the average per core. The last column shows the number of elements in the global leaf database for LTSMIN.

Grammar/ Start State	States/ Transitions	Tool	Cores	Time (s)	Speedup	Mem (MB)	Leaf db
le	3.724.544	GROOVE		5.128		2.751	
start-7p	16.956.727	LTSMIN	1	–	–	–	
			8	2.005	2,6	52	
			64	307	16,7	52	1.819
unflagged-platoon	1.580.449	GROOVE		1.016		1.259	
start-10	10.200.436	LTSMIN	1	6.621	0,2	120	
			8	889	1,1	54	
			64	156	6,5	54	8.534
append	261.460	GROOVE		202		372	
append-4-list-8	969.977	LTSMIN	1	3.285	0,1	99	
			8	352	0,6	76	
			64	92	2,2	70	69.147

state vector size for the first two cases was chosen such that $n \geq k+5$, hence no slot needs to encode the edges of more than one node; however, this is not the case for the third case.

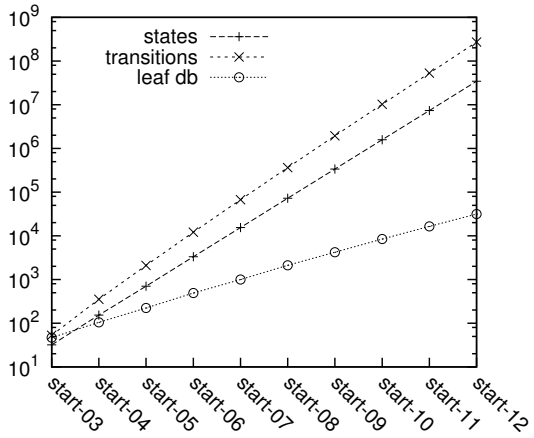
We have compared the performance of the distributed setting with the default, sequential implementation of GROOVE, running on the same machine but with a memory upper bound of 20GB. For the leader election with start state `start-7p` a different machine with 60GB of memory has been used, because with 20GB of memory the result could not be calculated.

Where there are no values in the table or figures of this section, either the time limit of 4 hours was exceeded or there was not enough memory.

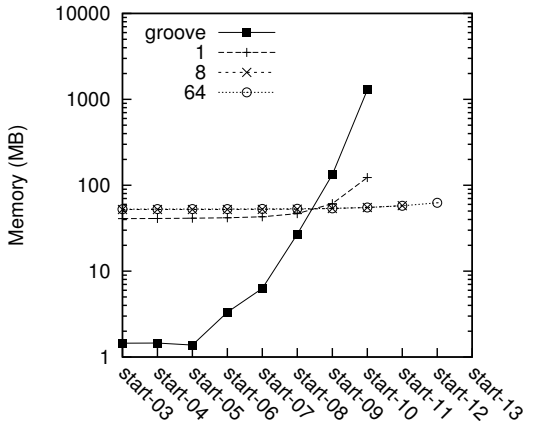
Global results. Table 1 shows some global results for the three cases, using the largest start graphs for which the sequential setting could compute the entire state space. We can observe that with 8 cores, the distributed setting starts to outperform the sequential, and also that the speedup increase from 1 to 8 cores and from 8 to 64 cores is sizeable, though below the optimal value of 8. Furthermore, the leaf database of the append rule system grows much larger than for the others, despite the fact that the state space is much smaller. This is a consequence of the fact that the graph size outgrows the vector size for this case.

Time and memory distributions. For the car platooning case, more detailed results are shown in Figures 3 and 4. First of all, Figure 3a shows that even though the size of the problem grows exponentially, the number of elements of the leaf value database of LTSMIN does not. This is also reflected by the per-core memory usage of LTSMIN (Figure 3b), which seems hardly to grow, in contrast to the more than exponentially growing memory usage of GROOVE.

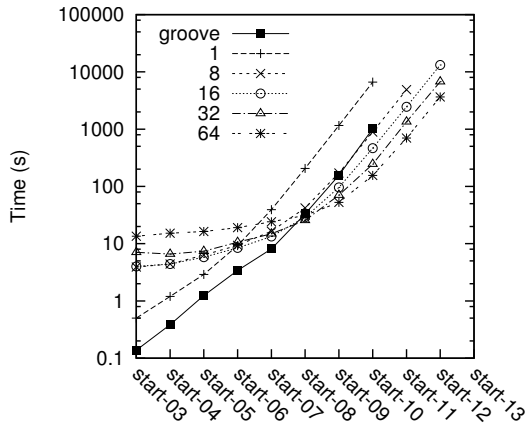
Figures 3c and 3d show the execution time respectively the speedup of LTSMIN with different numbers of cores compared to GROOVE. The execution time of LTSMIN with one core is much worse than GROOVE, but the speedup is growing fast. For the start states with 11 and 12 cars,



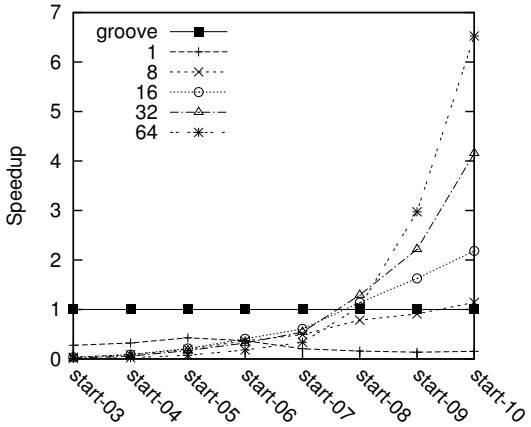
(a) Number of states, transitions and leaf values.



(b) Memory usage for GROOVE and LTSMIN (per core).



(c) Execution time.



(d) Speedup compared to GROOVE.

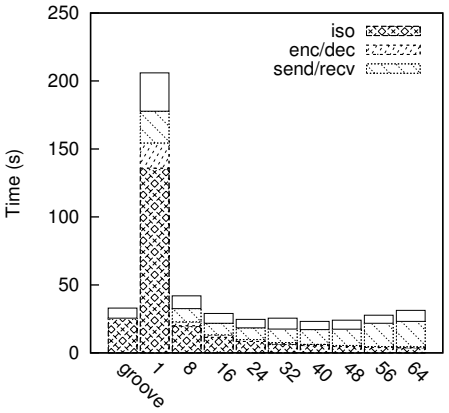
Figure 3: Figures for the car platooning case for different start states.

GROOVE cannot generate the state-space within 4 hours, but LTSMIN with 8 respectively 16 or more cores can.

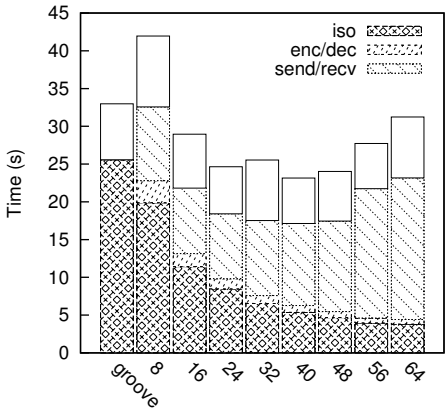
Execution time decomposition. Figure 4 shows how the execution time is built up. For smaller start states, the communication between cores (labelled "send/recv" in the figure) is a major factor in the computation time for LTSMIN, but for larger start states most of the time is spent on computing canonical forms (isomorphism reduction). As the number of cores grows, however, the communication again starts to play a larger relative rule — which is to be expected since this is the only task that is *not* parallellised; indeed, the communication overhead grows more than linearly with the number of cores.

Analysis. For lack of space we cannot include all results in this paper, but the trends for the leader election and append rule systems are very similar to the ones reported above for car platooning. Based on these results, we come to the following observations:

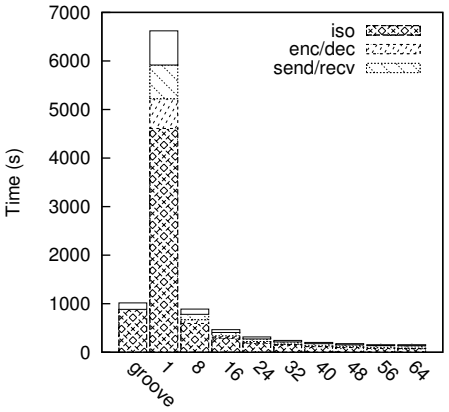
- The chosen serialisation of graphs works well in the reported cases. The total number



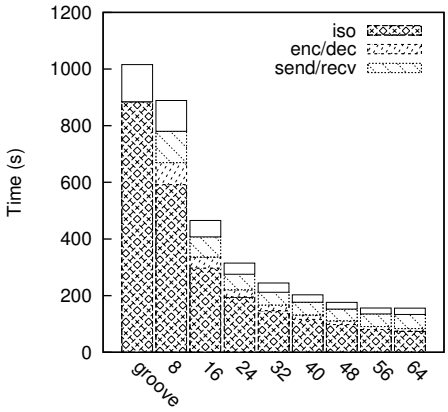
(a) Execution times for `start-08`.



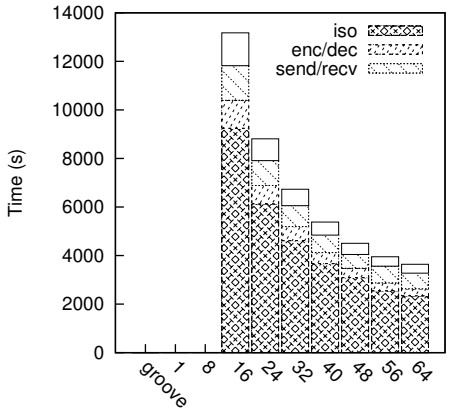
(b) Execution times for `start-08` without LTSMIN 1.



(c) Execution times for `start-10`.



(d) Execution times for `start-10` without LTSMIN 1.



(e) Execution times for `start-12`.

Legend

iso	Computing canonical forms, including the conversion of GROOVE to BLISS and back
enc/dec	Encoding and decoding of GROOVE graphs into state vectors and back
send/recv	Waiting for the next assignment from LTSMIN
rest	Matching in GROOVE

Figure 4: Decomposed execution times for the car platooning case for different numbers of cores.

of values stored, the so called *leaf nodes* of the tree compression database, is orders of magnitude smaller than the total number of states. Indeed, the number of states grows

exponentially with the problem size for all three modelled systems, but the number of leaf nodes grows less than exponentially. Although the canonical form calculation can renumber nodes in an unpredictable manner, which in the worst case could blow up the number of leaf values, apparently the different states are really a combinatorial result of the different parts of the vector. This is especially true for the leader election and car platooning cases, where the number of nodes is a priori bounded and the vector size can be chosen to accomodate this; in the append case, where the state vector representation has to reuse slots for multiple nodes, the results are less spectacular, though still quite good.

- The memory performance of the distributed LTSMIN solution is better than that of the sequential GROOVE system. This is a direct consequence of the success of the serialisation, but it deserves a separate mention. GROOVE uses dedicated data structures, which store only the difference (delta) between successive graphs; nevertheless, the very general tree compression algorithm of LTSMIN turns out to beat this hands down. This came as a big surprise to us, and is reason to reconsider the data structures of GROOVE.

- The time performance of the distributed LTSMIN solution scales well with the number of cores, especially for larger start graphs. The performance of a single core is quite bad compared to GROOVE, taking in the order of 8-10 times as much time, but the distributed system with 8 or more cores is faster. For the largest cases that GROOVE still can compute, we get speedups up to 16 (for 64 cores); moreover, the LTSMIN solution continues to scale well for larger start graphs, which GROOVE on its own cannot cope with at all any more.

- The canonical form computation in the LTSMIN-based system lasts as much as 5 times longer than isomorphism checking in stand-alone GROOVE. As the certificate-based solution of GROOVE uses the same underlying technique as BLISS' canonical form computation (namely, repeated partition refinement), there is no obvious reason for this performance penalty; we hypothesize that it is a consequence of the required encoding of edge-labelled GROOVE graphs as node-labelled BLISS graphs, which increases the graph size. It therefore seems interesting to reimplement the BLISS algorithm for edge-labelled graphs. Given the fact that isomorphism checking is a major fraction of the total time, we expect that this may further improve the distributed performance.

5 Conclusion

We showed a successful way of parallellising graph-based state space generation, using a combination of three tools: GROOVE, BLISS and LTSMIN. A nontrivial step is the encoding of arbitrary graphs into fixed-sized state vectors. We concluded that the resulting system scales well with the number of cores, and has a surprisingly good memory performance — so good, in fact, that it might be worth replacing the current GROOVE data structures. We also observed that a further performance gain can probably be made by reimplementing the functionality of BLISS in order to take advantage of the structure of edge-labelled graphs.

An interesting question raised in the course of this work is whether isomorphism checking is a good idea at all. Omitting the canonical graph computation would ensure that rules have only

local effect on the state vector, giving rise to nontrivial (in)dependencies between transitions. This in turn would allow more of the functionality of LTSMIN to be used, namely the symbolic storage of states. Though there are examples where symmetry reduction has a huge payoff, the same is true, to an even larger degree, for symbolic representations. This is a subject for future investigation.

Bibliography

[BK09] C. Baier, J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2009.

[BLPW08] S. Blom, B. Lisser, J. van de Pol, M. Weber. A Database Approach to Distributed State Space Generation. In Cerná and Haverkort (eds.), *Parallel and Distributed Methods in verifiCation (PDMC)*. Electr. Notes Theor. Comput. Sci. 198, pp. 17–32. Elsevier, 2008.

[BPW10] S. Blom, J. van de Pol, M. Weber. LTSMIN: Distributed and Symbolic Reachability. In *Computer-Aided Verification (CAV)*. LNCS 6174. 2010. To appear.

[BRV09] G. Bergmann, I. Ráth, D. Varró. Parallelization of Graph Transformation Based on Incremental Pattern Matching. In Boronat and Heckel (eds.), *Graph Transformation and Visual Modeling Techniques (GT-VMT)*. Electr. Comm. of the EASST 18. 2009.

[CJEF96] E. M. Clarke, S. Jha, R. Enders, T. Filkorn. Exploiting Symmetry in Temporal Logic Model Checking. *Formal Methods in System Design* 9(1/2):77–104, 1996.

[CPR08] P. Crouzen, J. C. van de Pol, A. Rensink. Applying Formal Methods to Gossiping Networks with mCRL and Groove. *ACM SIGMETRICS performance evaluation review* 36(3):7–16, December 2008.

[JK07] T. Junttila, P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *9th Workshop on Algorithm Engineering and Experiments*. Pp. 135–149. SIAM, 2007. See http://www.tcs.hut.fi/Software/bliss/.

[McK81] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium* 30:45–87, 1981.

[McK09] B. D. McKay. NAUTY User's Guide (Version 2.4). Nov. 2009. See http://cs.anu.edu.au/~bdm/nauty/nug.pdf.

[Ren04] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz et al. (eds.), *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. LNCS 3062, pp. 479–485. Springer Verlag, 2004.

[Ren07] A. Rensink. Isomorphism Checking in GROOVE. In Zündorf and Varró (eds.), *Graph-Based Tools (GraBaTs)*. Electr. Comm. of the EASST 1. September 2007.

[Wei02] E. W. Weisstein. Isomorphic Graphs. From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/IsomorphicGraphs.html, 2002.