Distributed State Space Generation for Graphs up to Isomorphism

Gijs Kant

24 August 2010

Master's thesis

Department of Computer Science

UNIVERSITY OF TWENTE.

Summary

In order to achieve reliability of systems, formal verification techniques are required. We model systems as graph transition systems, where states are represented by graphs and transistions between states are defined by graph transformation rules. For formal verication, often the state space of the system has to be generated, i.e., the set of all reachable states. The main problem with generating the state space is its size, which tends to grow exponentially with the size of the modelled system. This results in enormous computation time and memory usage already for relatively small systems. Recent developments in hardware design aim towards multi-core and multi-processor architectures. In order to benefit from these we require distributed tools: the task of state space generation needs to be split up into subtasks that can be distributed to multiple cores, called workers.

In this report we present a tool for *distributed state space generation* for graph-based systems. The tool uses LTSMIN [23] for efficiently distributing the states over the workers and for storing the state space, and instances of GROOVE [28] for computing successor states by applying graph transformation rules. Because LTSMIN uses fixed sized state vectors to represent states, a serialisation from graphs to state vectors and its reverse are required. We define two such serialisation functions: one that partitions the graph by its nodes (node vector) and one that uses edge labels to partition the graph (label vector).

In graph transformation systems a powerful symmetry reduction can be achieved by using *isomorphism* checking [27]. Isomorphic states can be merged, resulting in a reduced transition system that is bisimilar to the transition system without reduction, which implies that they satisfy the same semantic properties [29]. In our distributed tool we compute a canonical form for each computed successor graph, which enables us to also distribute the isomorphism reduction to the workers. For computing canonical forms, BLISS (described in [16]) is used together with a conversion that is needed because GROOVE uses a slightly different graph formalism (edge labelled graphs) than BLISS (coloured graphs) does.

We have performed experiments to investigate the time and memory performance of the distributed tool, based on LTSMIN, with the two different serialisation functions, compared to the sequential version of GROOVE for three different case studies. The experiments show that the node vector encoding is better than the label vector encoding both in memory usage and execution time.

The distributed setup with the node vector serialisation results in orders of magnitude less memory usage than sequential GROOVE for the very symmetric cases. This is suprising because the storage in GROOVE is optimised for graphs, storing only the differences (deltas) between state graphs instead of the complete graphs themselves.

The execution time for LTSMIN with one core is much worse than that of GROOVE, because GROOVE uses a canonical hashcode, which often prevents that isomorphism checking has to be used, and some kind of partial order reduction for graph transition systems that is not applicable in the distributed setting. Also, the conversion to coloured graphs, needed for computing canonical forms using BLISS, blows up the size of the graphs. However, for the larger models the distributed solution scales well. For all reported case studies there are start graphs for which GROOVE runs out of memory or is not capable of finishing within the time limit, while the distributed tool still can generate the state space. In one of the cases a speedup of 16 is achieved with 64 workers in the largest case for which also GROOVE could generate the state space within the time limit. For the very symmetric cases most of the time is spent on isomorphism checking in GROOVE. In the distributed setting, the speedup is explained by the time spent on canonical form computation that decreases linearly as the number of workers increases. As the number of workers increases, however, the communication overhead also grows.

Contents

1

Introduction

Computer systems and software are everywhere around us. Internet applications and databases are widely used for communication and administrative tasks. We are more and more dependent on the availability of such systems for economic transactions, communication, etc. Computer systems are also used in safety-critical situations, for controlling power plants, airplanes, cars, etc. These systems become more and more complex, especially with the ongoing trend towards distributed systems and parellel software. Therefore we need reliable systems and for that we need techniques to verify systems and to assist in building reliable, dependable systems.

In software engineering there are several ways to establish correctness and reliability of the software product. We consider one of them, which is to formally specify the behaviour of the system and verify that the system satisfies some set of formal properties, e.g., that it is free of deadlocks and that no illegal state can be reached. The properties can be expressed as formulae in a temporal logic, such as Computational Tree Logic (CTL) or Linear Temporal Logic (LTL). Verifying that a specification satisfies such a formula is called *model checking*. This kind of verification is usually done by exploring the *state space* of the specified system, i.e., the set of all reachable states (from some given initial state). The state space is represented as a *Labelled Transition System* (LTS), where each transition represents an action that is possible from a certain state and to which state the action leads. Checking correctness properties can be done while generating the LTS (on-the-fly) or afterwards if the LTS is stored somewhere.

We model systems as *graph transformation systems*, where *states* are represented by *graphs* and *transistions* between states are defined by *graph transformation rules*. Modelling systems as graph transformation systems has certain advantages. Modelling states as graphs provides an elegant way of expressing entities and relations between entities. Graph transformation systems have a solid formal basis, the semantics of graph transformation rules are well defined. Various tools exist that support graph-based modelling of systems. For instance, the tool Groove [28] can be used for graph-based model checking [27]. In Groove, directed edge labelled graphs are used to model states. See, e.g., [12] for examples of modelling and analysis using Groove. In Chapter 2 the basic concepts related to graphs and graph transition systems are presented.

The main problem with model checking is the size of the transition system, which tends to grow exponentially with the size of the specified system. This results in enormous computation time and memory usage already for relatively small systems. Many kinds of techniques have been developed and are still being developed to try to cope with this complexity of model checking. There are very powerful reduction techniques that reduce the size of the transition system without changing its semantic properties, such as Partial Order Reduction (see, e.g., [2]).

Recent developments in hardware design aim towards multi-core and multi-processor architectures. In order to benefit from these we require distributed tools: the task of state space generation for graph transformation systems needs to be split up into subtasks that can be distributed to multiple cores or machines, called *workers*. The algorithms in Groove are written to run on one machine. As far as we know, no other tools are currently available that provide distributed state space generation for graph transformation systems. Tools already exist for distributed state space generation for states modelled as vectors of data which, however, cannot be used directly for graphs. One of these is the LTSMIN toolset [23].

In this report we present a tool for *distributed state space generation* for graph-based systems. The tool uses LTSMIN for efficiently storing the state space and distributing the states over the workers, and instances of GROOVE for computing successor states by applying graph transformation rules. Because LTSMIN uses fixed sized state vectors to represent states, a serialisation from graphs to state vectors and its reverse are needed for using LTSMIN to communicate graphs. We define two such serialisation functions for graphs: one that partitions the graph by its nodes (node vector) and one that uses edge labels to partition the graph (label vector).

In state space generation, there are also techniques for *symmetry reduction*, i.e., grouping of 'symmetric' states. Particularly for graph transformation systems a powerful symmetry reduction can be achieved by using *isomorphism* checking [27]. Isomorphic states are structurally equivalent, but the nodes of the graph can have different identities. Because application of transformation rules is based on matching, i.e., finding a subgraph isomorphism, isomorphic states give rise to equivalent transitions with equivalent target states. Therefore, if two states are isomorphic, they are considered to represent the same state and only one of them has to be stored. The resulting reduced transition system is bisimilar to the transition system without reduction, which implies that they satisfy the same properties [29]. This reduction can be established in two ways: (a) when encountering a new state we can search in the set of visited states for an isomorphic state; or (b) we can compute a canonical form of the graph and store that canonical form. A canonical form of a graph G is a graph that is isomorphic to G, such that for a graph H, the canonical forms of G and H are equal if and only if G and H are isomorphic. By definition a set of isomorphic graphs has only one canonical form, which serves as unique representative of that set.

In our distributed tool we compute a canonical form for each computed successor graph, which enables us to also distribute the isomorphism reduction to the workers. Canonical forms for coloured graphs, a slightly different graph formalism from the edge labelled graphs used in GROOVE, can be computed by BLISS (described in [16]), which is an improved version of NAUTY (described in [19]). In Chapter 4 the theory behind these tools is described and a conversion from edge labelled graphs to coloured graphs is presented. An extensive survey of different methods for computing canonical forms is given in [17].

We have performed experiments with several rule systems to compare the distributed setup, based on LTSMIN, to the existing sequential version of GROOVE. The results show that a significant speedup and much better memory usage are achieved by the distributed approach compared to GROOVE.

This report is organised as follows. In the next section we discuss related work. Chapter 2 introduces the concepts and definitions used in the report. The architecture of the distributed state space generation tool and the relation between LTSMIN, GROOVE and BLISS are described in Chapter 3. The algorithms behind graph canonizer BLISS and how BLISS can be applied to edge labelled graphs are described in Chapter 4. We describe the graph serialisations and formally demonstrate their correctness in Chapter 5. The experiments and results are presented in Chapter 6. Chapter 7 concludes the report and contains suggestions for future work.

1.1 Related Work

(Distributed) Graph transformation. Some work has already been done on parallelisation of graph transformation by parallelising graph matching, see [4]. Pattern matching and rule application are parallelised by distributing the storage and updating of the rule match network, the so called *rete* network. However, that work is aimed at performing linear graph transformation rather than generating a complete state space. In that case isomorphism reduction is not an issue.

Distributed state space generation. Several tools exists for distributed model checking based, such as distributed versions of the model checker SPIN [18, 3] and the tool LTSMIN [5, 23]. SPIN uses a process modelling language (Promela) as input. In [30] a serialisation of graph grammars to Promela models (used in SPIN) is reported, but the performance (with normal SPIN) is worse than that of GROOVE. LTSMIN is very modular by design and supports multiple language modules (for input) and multiple model checking tools (for verification). That is why we choose to use LTSMIN, which is described in Section 3.1.

Graph isomorphism. Subgraph matching is known to be a NP-complete problem (Problem GT48 in [11]). For isomorphism checking it is not known if the time complexity is in P or if the problem is NP-complete. It is believed not to be in P [34].

Many algorithms exists that can check if two graphs are isomorphic. Ullmann [33] presented a search tree based algorithm for finding graph or subgraph isomorphisms between two graphs. Messner & Bunke [21, 22] made an optimised version for large graphs. The graph matching algorithms by Cordella et al. [6, 7, 8] also aim at isomorphism checking for pairs of graphs. They use heuristics and efficient data structures that are optimed for matching large graphs. Foggia, Sansone, and Vento compare four one-to-one isomorphism checking algorithms to NAUTY, a tool that computes canonical forms [10]. For many cases NAUTY performs comparable to these algorithms or better. For some cases NAUTY performs worse or is unable to find an answer, whereas some other algorithms are able to find an answer for all test cases.

Computing canonical forms. The NAUTY program by McKay [20] is able to produce a canonical form for directed coloured graphs, which can be used to test for isomorphism between graphs. The algorithm of McKay for computing the canonical form is described in [19] and will be explained in Section 4.2.

The complexity of the algorithm of McKay has been analysed by Miyazaki [24]. Miyazaki shows that NAUTY has an exponential worst case complexity. For some 3-regular graphs (for which canonical labellings can be computed in polynomial time, see [1]) NAUTY has an exponential lower bound. However, in practice the average computation time is much better.

Optimisations of McKay's algorithm for large and sparse graphs have been done by Junttila & Kaski [16] in the tool BLISS [15]. In experiments BLISS is shown to be faster than NAUTY for large and sparse graphs. It uses datatypes that allow more efficient storage and searching than the adjacency matrix that is used in NAUTY. Also other certificates for nodes in the search tree are used and the heuristics for pruning certain subtrees of the search tree are further optimised.

Using canonical forms in model checking. The tool NAUTY has been used in model checking of systems specified in B by the tool PROB [32, 31]. The states of the B model are translated to edge labelled graph representations. These are again converted to a coloured graph representation and compared using NAUTY. In [32] a version of the NAUTY algorithm is used that is adapted to work for edge labelled graphs, but the search tree pruning optimisations of NAUTY are left out. In [31] on the contrary a conversion from edge labelled graphs to vertex coloured graphs is used in combination with the orignal NAUTY algorithm. In both approaches the states of the B model are converted to a graph representation and a canonical form for the state graph is computed in order to be able to store only the canonical forms. States in B models consist of sets containing abstract elements, nested sets and relations between elements. Transitions between states are inferred by operations on that data. Symmetries exist between the abstract elements of the sets. The elements do not have a concrete value, so if their relations to other elements are symmetric, they are interchangeable. Experimentation shows that the symmetry reduction results in faster model checking. However, this has only been tested with small numbers of vertices (< 100).

2

Graph-Based Verification

In this chapter we define concepts that will be used troughout the report, such as graphs, isomorphism of graphs, graph transformation, and graph transition systems.

2.1 Graphs

A *graph* is a structure that consists of a set of *vertices* (or *nodes*) and a set of *edges* (or *arrows*) between vertices. There are many different kinds of graphs, with and without labels or colours, with directed or undirected edges, and with only binary or n-ary edges. We use *directed labelled graphs*, to be defined shortly hereafter. We assume a fixed ordered set of edge labels $\mathscr{L}_{\mathscr{E}}$ and a fixed ordered set of vertex labels $\mathscr{L}_{\mathscr{V}}$. Vertex labels are used for *types* and *flags*, which are unary edges, i.e., edges that are connected to exactly one vertex. A vertex can have at most one type and arbitrary many flags. We also use special nodes representing primitive values, elements of the set $\mathsf{Val} = \mathsf{Int} \cup \mathsf{Bool} \cup \mathsf{String} \cup \mathsf{Double}$. We denote the set of types as $T = \{\mathsf{Int}, \mathsf{Bool}, \mathsf{String}, \mathsf{Double}\}$. We write the primitive values as tuples, e.g., $\langle \mathsf{Int}, 1\rangle$, $\langle \mathsf{Bool}, \mathbf{true}\rangle$ and use functions $type\colon T \times \mathsf{Val} \to T$ and $val\colon T \times \mathsf{Val} \to \mathsf{Val}$ for the type repectively value of the value nodes. E.g., $type(\langle \mathsf{Int}, 1\rangle) = \mathsf{Int}$ and $val(\langle \mathsf{Int}, 1\rangle) = 1$.

Let us define formally what we mean by directed labelled graphs.

Definition 2.1 (Directed labelled graph). A *directed labelled graph* G is a tuple $\langle V, E, c\rangle$ with a finite nonempty ordered set of vertices $V = (0, 1, \ldots, n-1)$ with vertices represented by natural numbers in the range $[0..n-1]$, a set of ordered tuples $E \subseteq V \times \mathscr{L}_{\mathscr{E}} \times V$ representing the binary edges, and a colouring function $c : V \to \mathscr{P}(\mathscr{L}_{\mathscr{V}}) \cup \mathsf{Val}$ that associates a set of vertex labels or a primitive value with each vertex. The edges have associated source and target functions $src, tgt\colon E \to V$ and label function $lab\colon E \to \mathscr{L}_{\mathscr{E}}$. The class of directed labelled graphs is denoted $\mathscr{G}_{\mathscr{L}}$.

Example 2.2. As an example throughout the report we use the *dining philosophers problem*, essentially a problem of *mutual exclusion*. The problem can be briefly described as follows. A number of philosophers (we use three for the example) is sitting around a table, where they are about to eat spaghetti. To the left and to the right of each philosopher there is a fork, i.e., each fork is shared by two philosophers. Each philosopher needs two forks to be able to eat spaghetti. Philosophers are either *thinking*, trying to acquire forks (*waiting*), or *eating*. Figure 2.1 depicts the start state of the problem. The square boxes denote nodes in the graph, with a bold face label for its type and italic face labels for flags. In the figure the nodes are numbered 0..5. These so called *node identities* are shown outside of the nodes or inside the box as *n0, n1*, etc, which is the notation that is being used in Groove. Node identities are only needed to write down the graph on paper in a textual form and for storing them in computer memory. The arrows denote the edges between nodes. The edges are directed, which means that, e.g., the edge $(5, \mathsf{right}, 2)$ has node 5 as source and node 2 as target.

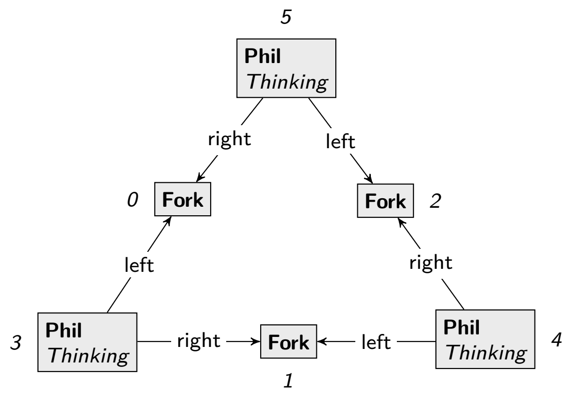


Figure 2.1: Example graph representing a state in the dining philosophers problem.

2.2 Graph Transformation

Graphs can be transformed using graph transformation rules. Without going in to details, we will explain here the basic concepts in graph transformation that are necessary in the context of this report. Graph transformation is defined category theoretically in terms of morphisms and pushouts. A graph *morphism* is a structure preserving mapping from elements in one graph to elements in another graph. This mapping can be a partial mapping. If the mapping is a bijection it is an isomorphism. A *pushout* is a formal definition of the result of combining morphisms. The rule is applied to a graph that is called the *host graph*. A detailed description of graph transformation, morphisms, pushouts and graph transition systems can be found in, e.g., [9].

A graph transformation rule consists of three parts:

Left Hand Side (LHS) The *reader* and *eraser* elements of the rule. These are the elements that are used for finding a *match*, which is a morphism from the LHS of the rule to the host graph. The *eraser* elements will be deleted from the host graph when the rule is applied.

Right Hand Side (RHS) The *reader* and *creator* elements of the rule. There is a morphism between the reader elements of the LHS to the reader elements of the RHS. The *creator* elements are added to the host graph when the rule is applied.

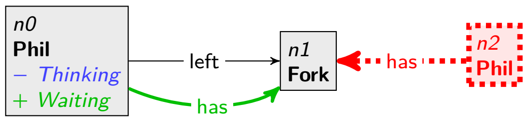
Negative Application Condition (NAC) The *embargo* elements and (some of the) *reader* elements of the rule. There is a morphism between the reader elements of the NAC to (part of) the reader elements of the NAC. If a match (extending the match between host graph and LHS) can be found from the NAC to the host graph, then the rule is *not* applicable.

The transformation takes place in three steps: i) *matching* the reader and eraser elements (the LHS) of the rule to (a subgraph of) the host graph, the graph on which the transformation takes place (i.e., finding a morphism between the host graph and the LHS), ii) trying to extend the match between host graph and LHS with a match between host graph and the NAC, and iii) *applying* the change that is specified by the rule if a match is found for the reader elements, but not for the NAC.

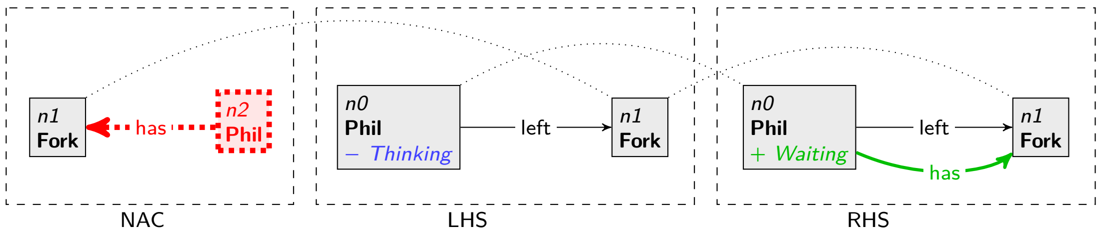
Rule application in GROOVE is done in a Single Pushout (SPO) manner. The most important consequence of that is that when a node is removed also attached edges are removed automatically. We will not further explain pushouts here. See, e.g.,[9] for a more detailed account.

In GROOVE the three parts of rule are combined in a single graph representation. There the black parts of the rule form the *reader* elements, the blue dashed parts form the *eraser* elements, the green thick elements are the *creator* elements, and the red thick dotted elements are the *embargo* elements.

Example 2.3. Figure 2.2 shows a graph transformation rule in the two different representations. The tranformation rule looks in the host graph for a philosopher node and the adjacent fork node to the left of the philosopher (LHS) such that no philosopher already has taken the fork (NAC). The rule adds a has label between the philosopher and the fork representing that the philosopher has taken the fork (RHS). Between the LHS, RHS, and NAC there are implicit morphisms, i.e., nodes *n0* and *n1* in the LHS are mapped to nodes *n0* and *n1* in the RHS repectively and node *n1* in the LHS is mapped to *n1* in the NAC.



(a) The pickupLeft rule in the single graph represenation used in GROOVE.



(b) The pickupLeft rule as three graphs with implicit morphisms.

Figure 2.2: Example graph transformation rule: pickupLeft.

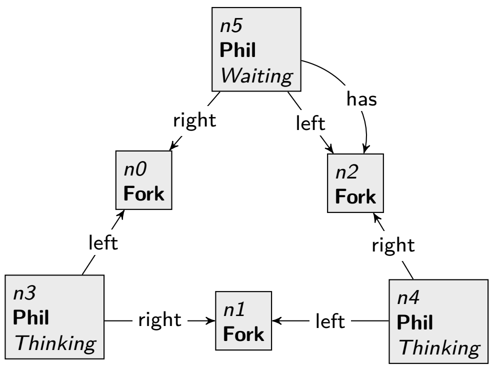


Figure 2.3: Resulting graph from transformation of the graph in Figure 2.1 by the pickupLeft rule in Figure 2.2.

When applying the rule to the graph in Figure 2.1, there are three possible resulting states, because the rule can be applied to three pairs of philosopher and fork nodes. The three resulting states are isomorphic, i.e., only the node identities are different. Isomorphism will be defined in Section 2.4. One of the resulting state graphs is in Figure 2.3.

2.3 Graph Transition Systems

Given a start graph and a set of graph transformation rules (also called a *graph grammar*), we can compute all states reachable from the start state. This results in a graph transition system, as described in [27]. We write $\mathcal{G}$ for a class of graphs, e.g. directed labelled graphs, $\mathcal{R}$ for the class of graph transformation rules that can be applied to graphs in $\mathcal{G}$, and $\mathcal{L}_{\mathcal{R}}$ for the class of transition labels resulting from graph transformation rules.

Definition 2.4 (Graph transition system)**.** A *labelled transition system* is a tuple $K = \langle Q, T, q_0 \rangle$, with a set of state graphs $Q \subseteq \mathcal{G}$, a set of labelled transitions $T \subseteq Q \times \mathcal{L}_{\mathcal{R}} \times Q$, and an initial state $q_0 \in Q$.

The states in the transition system are graphs and there is a transition labelled r, m from q to q', denoted $q \xrightarrow{r,m} q'$ if q' has been generated from q by rule r with match m, where m is a morphism from the left hand side of r to q.

In graph-based systems, usually the labels of self-edges in the transition system, i.e., the matching rules that do not change the state graph, are used to represent atomic properties that hold in the state to which the self-edge is connected. Using these atomic properties for states, from a graph transition system a *Kripke structure* can be constructed, a state machine with an interpretation function that associates each state with the set of atomic propositions that hold in that state. For this Kripke structure, Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) formulae can be specified expressing properties about the state space. Checking if such a formula holds for a system is called *model checking*. See, e.g., [2] for more on model checking, Kripke structures, state space reduction techniques and temporal logics.

In the following we assume the existence of a successor function that computes the set of successor state graphs for a given state graph for some state, based on a set of graph transformation rules:

Definition 2.5 (Successor function)**.** The successor function $succ\colon \mathcal{G} \times \mathcal{P}(\mathcal{R}) \to \mathcal{P}(\mathcal{G})$ computes the set of successor state graphs $succ(G,R)$ for a given state graph $G \in \mathcal{G}$, based on a set of graph transformation rules $R \subseteq \mathcal{R}$.

2.4 Isomorphism and Isomorphism Reduction

Is two graphs have the same structure and only have different node identities they are called *isomorphic*, which is formally defined as follows.

Definition 2.6 (Isomorphism of directed labelled graphs)**.** Let $G = \langle V_G, E_G, c_G \rangle$ and $H = \langle V_H, E_H, c_H \rangle$ be two directed labelled graphs. A bijective function $f : V_G \to V_H$ is called an *isomorphism* if

1) for all $v \in V_G$, $c_G(v) = c_H(f(v))$, and

2) for all $v_1, v_2 \in V_G$ and $l \in \mathsf{Lab}$, $(v_1, l, v_2) \in E_G$ if and only if $\big(f(v_1), l, f(v_2)\big) \in E_H$.

If such a function exists, G and H are called *isomorphic*, denoted $G \cong H$.

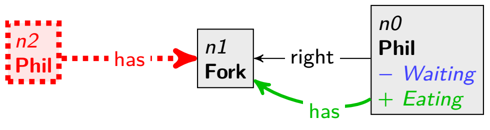
Isomorphism of two graphs implies that the same rules apply on the graphs, resulting in isomorphic successor states: for graphs $G, H \in \mathcal{G}$ and a set of rules $R \subseteq \mathcal{R}$,

$$G \cong H \implies \forall G' \in succ(G,R) \cdot \exists H' \in succ(H,R) \text{ such that } G' \cong H'.$$

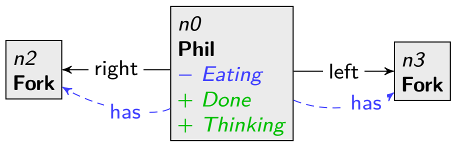
Isomorphism of state graphs is an equivalence that can be used for reduction of the state space. Instead of generating the full state space, we generate a *graph transition system modulo isomorphism*, i.e., isomorphic graphs are considered to represent the same state. In such a reduced transition system isomorphic states are represented by one representative state. The reduced transition system is *bisimilar* to the transition without isomorphism reduction (see [26, 29]). Bisimulation equivalence

Algorithm 2.1 Compute reduced graph-based transition system for $q_0 \in \mathcal{G}$ and a set of rules $R \subseteq \mathcal{R}$.

```
1:  $Q := \{q_0\}$
2:  $T := \emptyset$
3:  $S := \{q_0\}$
4:  while $S \neq \emptyset$ do
5:      Let $q$ be some element of $S$
6:      $S := S \setminus q$
7:      for all $s \in succ(q, R)$ do
8:          if $\exists p \in Q$ such that $s \cong p$ then
9:              $T := T \cup \{(q, p)\}$
10:         else
11:             $Q := Q \cup \{s\}$
12:             $T := T \cup \{(q, s)\}$
13:             $S := S \cup \{s\}$
14:         end if
15:     end for
16: end while
17: return $\langle Q, T, q_0 \rangle$;
```



(a) The pickupRight rule.



(b) The dropForks rule.

Figure 2.4: Two graph transformation rules, used in Example 2.7.

implies that Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) formulae that hold in one transition system also hold in an equivalent system (see, e.g., [2] on bisimulation equivalence).

In Alg. 2.1 it is shown how a graph transition system modulo isomorphism can be derived from a start state q_0 and a set of rules R. This *isomorphism reduction* is achieved by checking for isomorphism instead of checking for equality in line 8. For a set of state graphs S (initially containing only the initial state q_0) the successor states are computed. If a successor state has been visited before, only a transition to that state is added (line 9), otherwise a new state is added to the set of states Q and to S and a transition to the new state is added to the set of transitions (lines 10–13).

Example 2.7. In Figure 2.4 there are two other graph transformation rules: pickupRight and dropForks. The pickupRight rule specifies that if a philosopher is in a *waiting* state, meaning that he already has a left fork, he picks up the fork on his right if that fork is available. The philosopher changes to *eating* state. The dropForks rule specifies that if a philosopher is in an *eating* state he may stop eating and drop the forks on his left and right hand side. The philosopher then changes to *thinking* state again and is marked *done*, meaning that he already has had a meal. The *done* flag can be used to check if every philosopher gets a meal, i.e., that the rule system does not result in *starvation*. Applying Algorithm 2.1 to the rules pickupLeft, pickupRight, and dropForks and the start state in Figure 2.1 results in a labelled transition system consisting of 40 states and 110 transitions, which is shown in Figure 2.5. Without isomorphism reduction the state space consists of 112 states and 314 transitions. The green node represents the start state *s0*, which is the state in Figure 2.1, the orange nodes represent final states, i.e., states from which no transition is possible. This means that all philosophers are a *waiting* state and none can get his right fork. Such a state is called a *deadlock* state. From the LTS it is clear that this rule system is not free of deadlocks. When one inspects the final states (not displayed here), it appears that the rule system is also not free of starvation, i.e., there are final states where not all philosophers have had a meal. Sometimes there are multiple labels on a single transitions, meaning that different transitions lead to an isomorphic state.

Instead of checking for isomorphism for each visited state separately, we can use canonical forms

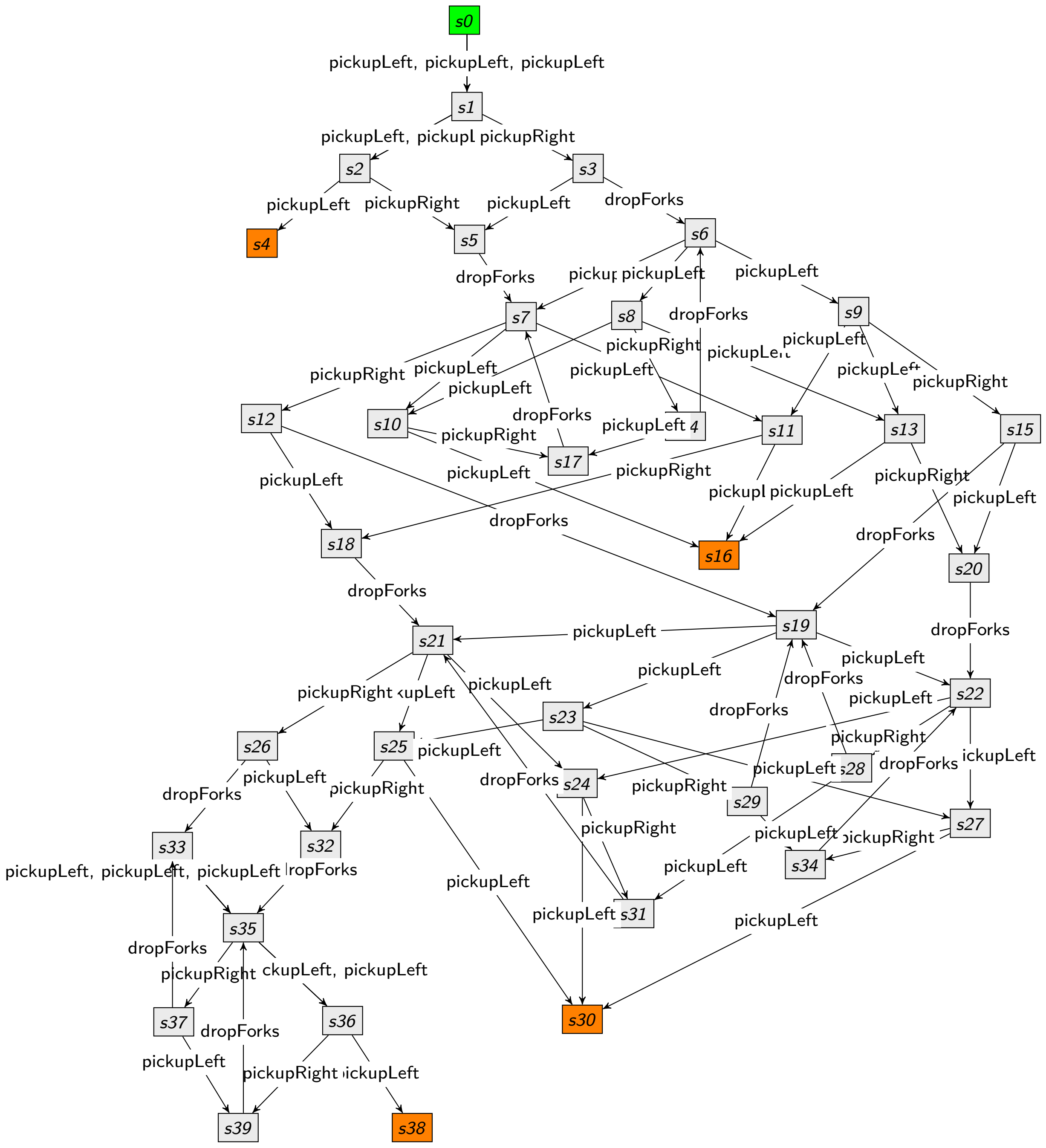


Figure 2.5: Labelled Transition System for the dining philosophers problem with three philosophers.

Algorithm 2.2 Compute the reduced graph-based transition system for $q_0 \in \mathscr{G}$ and a set of rules $R \subseteq \mathscr{R}$ using a canonical representation function *can*.

```
1:  $r_0 := can(q_0)$
2:  $Q := \{r_0\}$
3:  $T := \emptyset$
4:  $S := \{r_0\}$
5:  while $S \neq \emptyset$ do
6:      Let $q$ be some element of $S$
7:      $S := S \setminus q$
8:      for all $s \in succ(q, R)$ do
9:          $r := can(s)$
10:         if $r \notin Q$ then
11:             $Q := Q \cup \{r\}$
12:             $S := S \cup \{r\}$
13:         end if
14:         $T := T \cup \{(q, r)\}$
15:     end for
16: end while
17: return $\langle Q, T, q_0 \rangle$;
```

of successor states and communicate and store those canonical forms. The advantage of that being that isomorphism reduction is achieved independent of the way states are stored. For this we need a canonical representation function that is defined as follows:

Definition 2.8 (Canonical representation and canonical form)**.** A *canonical representation* function $can\colon \mathscr{G} \to \mathscr{G}$ computes an isomorphism invariant graph representative for each graph that is isomorphic to that graph, i.e., $can(G) \cong G$ for all $G \in \mathscr{G}$, such that for every pair of graphs $G, H \in \mathscr{G}$,

$$can(G) = can(H) \text{ if and only if } G \cong H.$$

$can(G)$ is called the *canonical form* of G.

In Alg. 2.2 it is shown how such a canonical representation function can be used in generating a reduced transition system. In line 9 the canonical form is computed. In the next line an equal graph is looked up, instead of an isomorphic graph as in Alg. 2.1. The use of canonical forms may result in a system with different (but isomorphic) states than the system that is generated by Alg. 2.1. However, the two transition systems themselves are isomorphic. Isomorphism of transition systems means that there is a bijective stucture preserving mapping from states in one system to states in the other system. In this case the states that are mapped to each other by the isomorphism of the transition systems are also isomorphic states. Isomorphism of transitions systems is an even stronger equivalence relation than bisimulation. Hence, the properties that hold for one system, also hold for an isomorphic one.

3

Architecture of Distributed Groove

In this chapter we describe the architecture that we use for distributed state space generation for graph transition systems. The task is distributed among several *workers*. Each worker may be a separate machine, but also multiple workers may run on the same machine. For the communication between the workers *message passing* (MPI) is used. Within the architecture a single worker consists roughly of four components.

Groove is used to compute successor states for state graphs, as described in Chapter 2.

LTSmin is used for storing the state space and for facilitating the communication of states to and from other workers.

Bliss is used for computing canonical forms for *coloured graphs*. Coloured graphs are slightly different from the edge labelled graphs used in Groove and will be defined in Chapter 4. A conversion between edge labelled graphs and coloured graphs is used to compute canonical forms for edge labelled graphs (described in the same chapter).

Distributed Groove (DG) connects Groove, LTSmin and Bliss as described below. DG runs within the same Java Virtual Machine (JVM) as Groove and uses Java method calls to communicate with Groove.

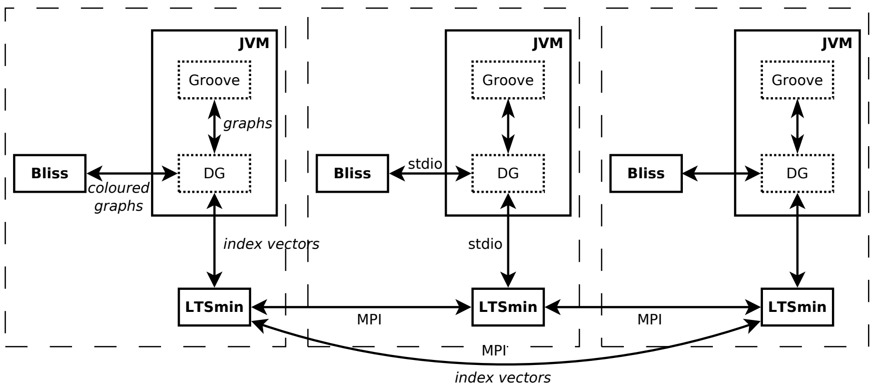


Figure 3.1: Schematic overview of the architecture.

How the tools relate to each other is depicted schematically in Figure 3.1. The workers are indicated by the dashed boxes. Each solid box in the figure represents a separate process, dotted boxes represent different tool instance within a JVM (for Groove and DG). The arrows represent communication between the tool instances.

- DG communicates (synchronously) with Bliss using pipes (standard input/output). Because of this, currently DG and Bliss need to run on the same machine. For each instance of DG a separate process of Bliss is run. DG sends a coloured graph to Bliss and receives its canonical form back. Conversion of graphs is done on the side of DG.

- DG communicates (synchronously) with LTSmin using either pipes (standard input/output) or TCP/IP. For each instance of LTSmin a separate process of DG is running. DG receives a state from LTSmin, computes successor states using Groove, and sends canonical forms of the successor states back to LTSmin. The state graphs are communicated between DG and LTSmin as *state vectors*. This means that on receiving a state, DG needs to decode the state vector to a graph, and on sending successor states, the canonical forms have to be encoded as a state vector. This *serialisation* of graphs is described in Chapter 5. The state vectors are compressed into *index vectors*, i.e., both DG and LTSmin keep a store of values used in the state vectors and communicate a vector of indices of the values instead of a complete vector of values as representation of states. Of course, each new value has to be communicated once between DG and LTSmin, so that they both know which index represents which value. This compression is explained in Section 3.1.1.

- LTSmin takes care of storing the labelled transition system, in which states are compressed even further using *tree compression*, which is explained in Section 3.1.2. The different instances of LTSmin communicate with each other using MPI. LTSmin keeps a global database of values used and uses compressed index vectors for communication between workers. A hash function that is applied to index vectors is used to determine which worker is responsible for which state.

In the following sections we describe the functionality of LTSmin (Section 3.1) and the functionality of DG and Groove in this distributed setting (Section 3.2).

3.1 LTSmin

The LTSmin toolkit [23] contains sequential, symbolic, and distributed tools to generate labelled transition systems for various input languages, such as μCRL, mCRL2, Promela (used in SPIN) and DVE (used in DiVinE). Bisimulation minimisation can be applied on the generated LTSs and several tools can be used to verify properties of the LTSs, such as CADP. LTSmin uses its own file format that is optimised for efficient storage of LTSs and distributed writing to disc. States are represented as state vectors with a fixed length and compression of the state vectors is used for efficient storage and communication of states. The state vectors and index vector encoding are described in Section 3.1.1. A recursive compression technique called *tree compression* that is used in LTSmin is described in Section 3.1.2.

3.1.1 State vectors and index vectors

Definition 3.1 (State vector). A *state vector* is a fixed-length vector of values $\vec{p} = \langle p_0, p_1, \ldots, p_m \rangle$, i.e., a vector of which the length m is fixed. The values can be of arbitrary type and size.

The actual state vector is not communicated, but instead an *index vector*, i.e., a vector containing the indexes of the values in a table.

LTSmin uses a global store P_j for each of the parts of the vector.[1] The complete so called *leaf database* of values consists of the stores $P_0, P_1, \ldots, P_m$. Now the index vector of a state vector $\langle p_0, p_1, \ldots, p_m \rangle$ is $\langle i_0, i_1, \ldots, i_m \rangle$, where i_j is the index of the value p_j in store P_j.

This technique is used in, e.g., the SPIN model checker [14] and in LTSmin [23]. It is based on the assumption that the number of different values of a certain part of the vector is relatively small with respect to the total number of different state vectors. For example, if multiple processes are modelled and local variables are stored in separate parts, then these parts are unchanged when an action of another processes is executed. Most changes are then *local* to the process. If two parts are never changed by the same action or transition, they are called *independent*. The total number of states is of combinatorial nature, i.e., the total number of states that can be encoded is the product of the number of values of the different parts: $|P_0| \cdot |P_1| \cdot \cdots \cdot |P_m|$. The more locality there is in the state vectors, the more space can be saved by storing the values of the parts separately and using an index vector,

[1]Actually, LTSmin uses a store for each *type* (not further defined here), but the description here is accurate for how LTSmin is used in this setting.

containing the indexes of the values in the database as vector parts. The total number of values to be stored is not the product, but the sum of the number of values of the parts: $|P_0| + |P_1| + \cdots + |P_m|$.

Example 3.2. Let $m = 4$ and let the leaf database contain the following values:

i	P_0	P_1	P_2	P_3
0	$\langle 4,5\rangle$	7	"a"	**49**
1	$\langle \mathbf{1},\mathbf{2}\rangle$	1337	"b"	50
2	$\langle 4,2\rangle$	**0**	"aa"	51
3	$\langle 9,5\rangle$	50	"ab"	52
4	$\langle 1,1\rangle$	2	**"aba"**	53

Now, value vector $\langle\langle 1,2\rangle, 0, \text{"aba"}, 49\rangle$ is encoded as index vector $\langle 1,2,4,0\rangle$.

For the examples we used (tuples of) integers and string for the values in the leaf database, but in LTSMIN any serializable type of values may be used in the leaf database. The indexes are in the range of natural numbers that can be encoded in 32 bits.

3.1.2 Tree Compression

These index vectors can also be partitioned into parts, which are then values for a next level of indexing. Parts of the index vector are stored again in a store, and the index of these parts in the store are used in the compressed vector. This is called *recursive indexing* [13] or *tree compression* [5].

Example 3.3. We explain the tree compression in LTSMIN by an example. For the index vector $\langle 1,2,4,0\rangle$ from the previous example, tree compression is applied in the following way. First the vector is split into two parts: $\langle 1,2\rangle$ and $\langle 4,0\rangle$. For the first part, the index is looked up in store I_0, for the second part in store I_1:

i	I_0	I_1
0	$\langle 0,0\rangle$	$\langle 0,0\rangle$
1	$\langle 1,1\rangle$	$\langle 1,1\rangle$
2	$\langle \mathbf{1},\mathbf{2}\rangle$	$\langle 2,0\rangle$
3	$\langle 2,0\rangle$	$\langle 3,1\rangle$
4	$\langle 0,1\rangle$	$\langle \mathbf{4},\mathbf{0}\rangle$

Tuple $\langle 1,2\rangle$ has index 2 in I_0, $\langle 4,0\rangle$ has index 4 in store I_1, so the vector $\langle\langle 1,2\rangle, 0, \text{"aba"}, 49\rangle$ can be encoded as $\langle 2,4\rangle$.

For larger vectors multiple levels of tree compression can be used. The *root* table consists of pairs of indexes representing states. Depending on the size of the state vectors, an index in a table of the tree compression datastructure points to either a value in the leaf database or to a value in a separate table one level lower. See [5] for an extensive description of tree compression in LTSMIN.

3.1.3 State compression and graphs

In the distributed setting we use canonical forms of state graphs to achieve isomorphism reduction. Canonically numbering nodes such as is done in NAUTY and BLISS renumbers the vertices of the graph in a way that is difficult to predict and for which local changes (adding or removing one edge, renaming a label) may result in a completely different numbered graph. This means that when a straightforward serialisation of graphs is used, already a small change in the state graph may result in large changes in the state vector resulting from that state graph. Therefore it is not immediately clear if there will be any benefit in using the state compression of LTSMIN for graphs. Althought there will perhaps be little locality of transitions, because of the renumbering of nodes, nonetheless the number of values per part may still be much less than the total number of states. It needs to be determined empirically if the state compression of LTSMIN works well for serialised graphs. The experiments in Chapter 6 will show that it depends on which serialisation is used whether the compression works well.

3.2 Groove, DG, and the interface to LTSmin

The DG part of the distributed setup is responsible for computing successor states (using Groove), computing canonical forms (using Bliss), serialising graphs and compressing the resulting state vectors into index vectors. DG and Groove are started with a *graph grammar*, containing a set of graph transformation rules, and a *start graph*, the graph representing the start state.

At initialisation, LTSmin starts N LTSmin workers and N instances of DG. Each instance of DG uses Groove as a library and loads the grammar and the start state that are to be explored. DG communicates to LTSmin the size of the state vectors used, which is either based on the number of nodes in the start graph or on the number of different edge labels used in the system (depending on the particular serialisation used, see Chapter 5). DG computes a canonical form for the start graph using Bliss and sends a serialised form of the canonical start graph to LTSmin.

The functionality of DG in this distributed setting is described in Algorithm 3.1. The following functions are used:

`recvValues` receives new values for the leaf database, i.e., values that have not been received before, and keeps a mapping between the values and their indexes in the leaf database;

`recvIndexVector` reads an index vector representing an open state, to be explored by the current worker;

`decompress`$_V$ decompresses the index vector to a state vector using the mapping between indexes and values of the leaf database;

`decode` decodes state vectors to graphs (see Chapter 5);

`succ` computes successor state graphs for a given graph (see Chapter 2) and returns a list containing for each successor state graph g a tuple $\langle l, g \rangle$ containing a transition label l(which is the name of the transformation rule that resulted in the successor state) and the state graph g;

`can` computes a canonical form of a graph (see Chapter 4);

`encode` encodes a graph to a state vector (see Chapter 5);

`compress`$_V$ and **`compress`**$_L$ compress state vectors and transition labels to index vectors and transition label indexes respectively;

`sendValues` sends newly encountered part values and transition labels to LTSmin;

`sendTransition` sends tuples containing a transition label index and the index vector of a successor state to LTSmin representing a transition.

Algorithm 3.1 Functionality of DG in the distributed setting: receiving states from LTSMIN, computing successor states and canonical forms, and communicating encoded successor states back to LTSMIN.

```
recvValues();
$q_I := \texttt{recvIndexVector}();$
while ($q_I \neq \texttt{null}$) do
    $q_V := \texttt{decompress}_V(q_I);$
    $g_q := \texttt{decode}(q_V);$
    $t := \langle \langle l_1, g_1 \rangle, \langle l_2, g_2 \rangle \ldots, \langle l_r, g_r \rangle \rangle := \texttt{succ}(g_q);$
    for $1 \leq i \leq r$ do
        $c := \texttt{can}(g_i);$
        $s_V := \texttt{encode}(c);$
        $s_I := \texttt{compress}_V(s_V);$
        $l_I := \texttt{compress}_L(l);$
        sendValues();
        $\texttt{sendTransition}(l_I, s_I);$
    end for
    recvValues();
    $q_I := \texttt{recvIndexVector}();$
end while
```

4

Computing a Canonical Form

In the architecture of the distributed state space generation tool, described in the previous chapter, isomorphism reduction of the generated state space is achieved by computing *canonical forms* of the state graphs, as defined in Definition 2.8. This chapter describes how to compute canonical forms of edge labelled graphs. [1]

Various tools exists that compute canonical form for coloured graphs, which is a slightly different graph formalism from the edge labelled graphs as defined in Chapter 2. Coloured graphs and other concepts used in this chapter will be defined in Section 4.1. In [17] an overview is provided of the tools that compute canonical forms of coloured graphs. In this report we use the tool BLISS by Junttila & Kaski [15, 16], which is based on the tool NAUTY by McKay [20, 19]. In Section 4.2 we describe the basic ideas and algorithms behind these tools. The reader that is not interested in the details of computing canonical forms may safely skip that section.

In order to be able to use the existing tools that use coloured graphs, a conversion is required from the edge-labelled graphs used in GROOVE to coloured graphs. The conversion that we use is presented in Section 4.3. A consequence of the conversion used, is that the size of the resulting graph is typically larger than the original graph. The size of the resulting graph is discussed in Section 4.3.2.

4.1 Definitions

We assume an ordered universe of vertex colours C. In BLISS natural numbers are used to represent colours.

Definition 4.1 (Coloured graph)**.** A *directed graph* G is a tuple $\langle V_G, E_G\rangle$ with a finite nonempty set of *nodes* (or *vertices*) V_G and a set of *edges* $E_G \subseteq V_G \times V_G$. The edges have associated source and target functions $src, tgt\colon E_G \to V_G$. A directed graph G is called *coloured* if it has an associated function $c\colon V_G \to \mathrm{C}$. The class of coloured graphs is denoted $\mathcal{G}_{\mathscr{C}}$.

Definition 4.2 (Isomorphism of coloured graphs)**.** Let $G = \langle V_G, E_G, c_G\rangle$ and $H = \langle V_H, E_H, c_H\rangle$ be two coloured graphs. A bijective function $f : V_G \to V_H$ is called an *isomorphism* if for all $v \in V_G$ $c_G(v) = c_H(f(v))$ and for all $v_1, v_2 \in V_G$,

$$(v_1, v_2) \in E_G \iff (f(v_1), f(v_2)) \in E_H.$$

If such a function exists, G and H are called *isomorphic*, denoted $G \cong H$.

Definition 4.3 (Permutation)**.** A *permutation* of a set A is a bijective function $\alpha\colon A \to A$. The image of $a \in A$ under a permutation α is denoted $\alpha(a)$ or a^α. The set of all permutations for a set $\{1, 2, \ldots, n\}$ is denoted S_n.

[1]This chapter is a compressed version of an earlier published technical report [17]. The technical report is partly written in the context of a research topics project and partly in the context of the final project, of which this report is the end product.

A permutation can be represented as a matrix:

$$\alpha = \begin{bmatrix} 1 & 2 & \cdots & n \\ 1^\alpha & 2^\alpha & \cdots & n^\alpha \end{bmatrix} \in S_n.$$

Definition 4.4 (Graph permutation). A *graph permutation* is a vertex permutation $\gamma\colon V \to V$ that associates with each directed coloured graph $G = \langle V, E, c\rangle$ a permuted graph $G^\gamma = \langle V^\gamma, E^\gamma, c^\gamma\rangle$, where

$$\begin{aligned} V^\gamma &= \{v^\gamma \mid v \in V\} = V, \\ E^\gamma &= \{(v_1^\gamma, v_2^\gamma) \mid (v_1, v_2) \in E\}, \text{ and} \\ c^\gamma &= \{(v^\gamma, k) \mid (v, k) \in c\}. \end{aligned}$$

The set of all graph permutations for a set of vertices V is denoted S_V.

For all permutations $\gamma \in S_V$ it holds that $G^\gamma \cong G$. A special subset of S_V is the set of *automorphisms* of G, $\mathrm{Aut}(G) = \{\gamma \in S_V \mid G^\gamma = G\}$.

An important ingredient of the algorithm that will be described in the next section is partition refinement. Vertices of the graph are partitioned in equivalence classes. The initial partition of the vertices is based on the colours of the vertices. Then the partition is refined such that also the number of incoming and outgoing edges from the vertices is taken into account.

Definition 4.5 (Partition). A *partition* π of a set of nodes V is a set $\{W_1, W_2, \ldots, W_r\}$ of nonempty disjoint *cells* $W_i \subseteq V$ whose union is V. A partition with only trivial cells, i.e., cells that contain only one element, is called a *discrete partition*. The partition that contains only one cell, the set V, is called the *unit partition*. The set of partitions of V is denoted $\Pi(V)$. An *ordered partition* of V is a sequence $(W_1, W_2, \ldots, W_r)$ such that the set $\{W_1, W_2, \ldots, W_r\}$ is a partition of V. The set of ordered partitions of V is denoted $\utilde{\Pi}(V)$.

The set of automorphisms for a graph G with vertex partition π is defined as $\mathrm{Aut}(G, \pi) = \{\gamma \in S_V \mid G^\gamma = G \wedge \pi^\gamma = \gamma\}$.

In the following we denote the vertices as natural numbers, i.e., the set of vertices V is the set of numbers $\{1, 2, \ldots, n\} \subseteq \mathbb{N}$ with $n = |V|$.

Definition 4.6 (Partition permutation). If $\pi \in \utilde{\Pi}(V)$ is a discrete ordered partition, we define the *permuted graph* $G(\pi)$, isomorphic to G, by relabelling the vertices of G in the order that they appear in π: given $\pi = (\{i_1\}, \{i_2\}, \ldots, \{i_n\})$ with $\{i_1, i_2, \ldots, i_n\} = \{1, 2, \ldots, n\}$, the permuted graph, denoted $G(\pi)$, is defined as $(G)^\delta$, where the permutation δ is given by

$$\delta = \begin{bmatrix} i_1 & i_2 & \cdots & i_n \\ 1 & 2 & \cdots & n \end{bmatrix} \in S_V.$$

This permutation δ, associated with partition π, is also written as $\overline{\pi}$. This partition permutation provides a relabelling of vertices based on a generated partition of vertices.

Example 4.7. As an example, suppose an isomorphism $\gamma = \{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 1\}$ that maps vertices of graph G to vertices of H, with

$$\begin{aligned} V_G &= V_H = \{1, 2, 3\}, \\ E_G &= \{(1, a, 2), (2, b, 3), (3, c, 1)\}, \text{ and} \\ E_H &= \{(2, a, 3), (3, b, 1), (1, c, 2)\}. \end{aligned}$$

If we use some ordered partition $\pi = (\{i_1\}, \{i_2\}, \{i_3\})$ as a permutation of G, then

$$(G)^{\overline{\pi}} = (H)^{\overline{\pi}^\gamma}.$$

For instance, let $\pi = (\{2\}, \{1\}, \{3\})$. Then

$$\begin{aligned} (E_G)^{\overline{\pi}} &= \{(2, a, 1), (1, b, 3), (3, c, 2)\}, \\ \overline{\pi}^\gamma &= (\{3\}, \{2\}, \{1\}), \\ (E_H)^{\overline{\pi}^\gamma} &= \{(2, a, 1), (1, b, 3), (3, c, 2)\} = (E_G)^{\overline{\pi}}. \end{aligned}$$

Definition 4.8 (Partition refinement). Given partitions π_1, π_2 of some set, π_1 is called a *refinement* of π_2 or *finer* than π_2 (and π_2 is called *coarser* than π_1), denoted $\pi_1 \sqsubseteq \pi_2$, if for all cells $V_i \in \pi_1$ there exists a cell $W_j \in \pi_2$ such that $V_i \subseteq W_j$.

The partition refinement algorithm used in computing the canonical form, to be described in Section 4.2 computes the coarsest stable refinement of a partition. Stability of a partition is based on the numbers of adjacent elements of the members of the cells of the partition.

Definition 4.9 (Number of adjacent elements). Given a directed graph $G = \langle V, E\rangle$ and a partition $\pi \in \Pi(V)$, for an element $v \in V$ and a cell $W \in \pi$, the number of elements of W which are adjacent in G to v is defined as:

$$d(v, W) = |\{w \in W \mid (v, w) \in E \vee (w, v) \in E\}| \tag{4.1}$$

This definition considers edges in both directions. This differs from [20] where only one direction is used, which is related to the data structure used in Nauty, which allows for easy comparison of rows of the matrix, whereas comparing columns is more expensive. In the case of undirected graphs this does not make a difference, but for directed graphs it does.

Definition 4.10 (Stable partition). A partition π is called *stable* for a directed graph G if for every pair of cells $W_i, W_j \in \pi$ the number of adjacent elements in W_j is the same for each element in W_i, i.e., for all vertices $v_1, v_2 \in W_i$ it holds that $d(v_1, W_j) = d(v_2, W_j)$. The set of all stable partitions of a set V is denoted $\Pi_S(V)$.

The stable partition resulting from the partition refinement algorithm is not necessarily a discrete partition, so the result of partition refinement can not immediately be used as permutation of the vertices. Each discrete partition is also stable, but for n vertices there are $n!$ possible permutations and we want to find a unique partition that gives us a canonical relabelling. Therefore we need a search tree that we can search for candidate canonical permutations and we need a way to order the candidate permutations so that we can choose one. The generation of this search tree and the ordering of the permutations are discussed in Section 4.2.3 and 4.2.2.

The partition refinement algorithm consists of iteratively splitting cells of the partition based on the number of adjacent elements of members of cells, until the partition is stable. The splitting of the cells of a partition π is defined with respect to a set S (usually some cell of the partition) and denoted $split(\pi, S)$.

Definition 4.11 (Split). For a partition $\pi \in \Pi(V)$ and a set $S \subseteq V$, a partition $\pi' = split(\pi, S)$ is a refinement of π for which for all $W \in \pi$, for all $W_i, W_j \subseteq W$ with $W_i, W_j \in \pi'$, it holds that

$$\forall v_1 \in W_i, v_2 \in W_j \cdot \big(d(v_1, S) = d(v_2, S) \iff W_i = W_j\big).$$

If $split(\pi, S) \neq \pi$, S is called a *splitter* of π.

4.2 Computing a canonical form of vertex coloured graphs

In this section it is explained how a canonical form of a directed coloured graph can be computed. McKay published an algorithm for finding a unique vertex labelling for isomorphic graphs [19], which is implemented in the tool Nauty [20]. Improvements have been done by Junttila & Kaski in the tool Bliss; the algorithm they describe is used in the remainder of this paper.

The idea is to generate for each graph a set of discrete partitions that can be used as permutation of the vertices of the graph, which results in a relabelled graph. If we have an ordering of the graphs, and if for isomorphic graphs the same set of relabelled graphs is generated, we can choose the minimum or maximum of the set as canonical form.

An easy but inefficient way of generating this set of graphs is generating all possible permutations of the vertices, which results in $|V|!$ permutations of the set of vertices V. An ordering of the graphs can be obtained by representing each graph by a string that is a concatenation of the vertex colours and of the rows of the adjacency matrix (which represents the incident edges in the graph), and use an ordering on the strings.

The tools Nauty and Bliss use far more efficienct algorithms that do not generate all possible vertex permutations, but still result in an equal set of permuted graphs for isomorphic graphs. The algorithms mainly consist of the following two ingredients:

1) A partition refinement algorithm that computes the unique coarsest stable partition for a given graph and initial partition of vertices;

2) An algorithm that generates a search tree of stable partitions with discrete partitions as leaf nodes, of which one is chosen as the relabelling partition permutation leading to the canonical form.

The search tree is generated by first computing a stable partition (which is the root node of the tree) and then splitting one of the cells. For each of the members of the cell a subtree is added, where that member is put in a separate cell. Then each of the resulting partitions is stabilised again. This continues until all branches end in discrete partitions (the leaf nodes).

Because every intermediate partition is stabilised before it is split again, the number of nodes in the tree is reduced. The properties that are used in the partition refinement are isomorphism invariant, so the resulting set of permuted graphs stays equal for isomorphic graphs. This is required in order to be able to compute the canonical form.

In the next section the partition refinement algorithm is explained. In Sections 4.2.3 and 4.2.4 the generation and pruning of the search tree are described. An ordering of coloured graphs is given in Section 4.2.2.

4.2.1 Partition refinement algorithm

The vertices of a graph are partitioned into cells of vertices that are similar. Initially this partition is based on vertex colours, but this partition is refined based on the number of neighbours of vertices in other cells. The partition refinement algorithm and the result it produces are described in this section.

The algorithm computes the unique coarsest stable refinement of a partition. The stability of partitions is defined in terms of numbers of adjacent vertices in the cells of the partition. A partition is stable if for each pair of elements of a cell the number of adjacent vertices is equal for both elements in all of the cells of the partitions (see Definition 4.10).

Suppose we have two isomorphic graphs $G \cong H$, with $G^\alpha = H$. Then if π_1 is a partition of vertices in G and $\pi_2 = \pi_1^\alpha$ is a partition of vertices in H, equivalent to π_1, in the sense that $(G)^{\pi_1} = (H)^{\pi_2}$ (see Example 4.7). Then also the unique coarsest stable refinements of π_1 and π_2 are equivalent. This is the case, because stability of partitions is defined such that it is isomorphism invariant, i.e., does not depend on the particular identities of vertices. It follows then from the uniqueness of the coarsest stable partition refinement and the isomorphism between G and H that the resulting stable partitions are equivalent (in the same sense of equivalence and by the same isomorphism α).

Unique coarsest stable refinement Here we prove that a unique coarsest stable refinement exists for each partition $\pi \in \Pi(V)$ of vertices V for a graph G. We assume that the set V is finite and hence also $\Pi(V)$, the set of all partitions of V, is finite. We start with proving that the set of all partitions of a set forms a lattice (both a least upper bound and a greatest lower bound exists for each set of partitions). Then we prove that the least upper bound preserves stability. From the fact that each discrete partition is stable we can conlude that for each partition there exists a stable refinement (i.e. the discrete partition). It then follows that for each partition there exists a unique coarsest stable refinement.

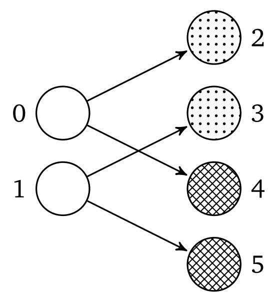
Definition 4.12 (Least upper bound of partitions)**.** For a set of partitions $\Pi \subseteq \Pi(V)$ and an ordering relation $\sqsubseteq$, an *upper bound* is an element $\pi \in \Pi(V)$ such that for all elements $\rho \in \Pi$, $\rho \sqsubseteq \pi$. The *least upper bound* of Π, denoted $\operatorname{lub} \Pi$, is the upper bound π such that for all other upper bounds ρ, $\pi \sqsubseteq \rho$.

The least upper bound of a pair of elements $\pi_1, \pi_2 \in \Pi(V)$ is also called *join*, denoted $\pi_1 \sqcup \pi_2$. For computing this least upper bound we need the following relation.

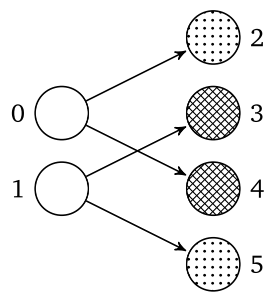
Every partition $\pi \in \Pi(V)$ can be considered as a binary equivalence relation where each pair reflects that two elements are in the same cell:

$$R = \{(s,t) \in V \times V \mid \exists W \in \pi \cdot s,t \in W\} \tag{4.2}$$

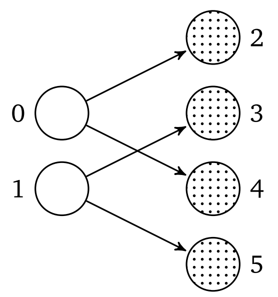
An example of partitions represented by binary relations is shown in Figure 4.1. The other way around, a partition can be derived from a binary relation $R \subseteq V \times V$. The relation can be seen as a graph (an edge between two elements representing that a relation between the elements exists). By taking the maximal connected subgraphs (or components) and regarding the vertices in those subgraphs as the elements of a cell (so, one cell per subgraph) we have a partition of the elements.



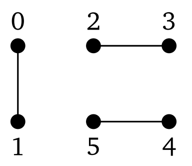
(a) Partition π_1.



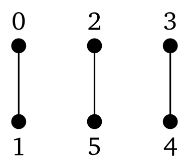
(b) Partition π_2.



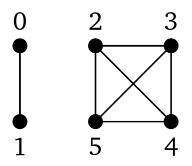
(c) The least upper bound of π_1 and π_2, $\pi_1 \sqcup \pi_2$.



(d) Relation R_1, reflecting the partition $\{\{0,1\},\{2,3\},\{4,5\}\}$.



(e) Relation R_2, reflecting the partition $\{\{0,1\},\{2,5\},\{3,4\}\}$.



(f) The transitive closure of $R_1 \cup R_2$, reflecting the partition $\{\{0,1\},\{2,3,4,5\}\}$.

Figure 4.1: The partitions π_1 and π_2, which are stable, and their least upper bound $\pi_1 \sqcup \pi_2$. The partitions are shown by the colours of the vertices. The partitions can be seen as relations, where elements in the same cell are related. Relation R_1 reflects partitions π_1, R_2 reflects partition π_2, and the transitive closure of R_1 and R_2 reflects the least upper bound $\pi_1 \sqcup \pi_2$.

Proposition 4.13. *Given a set of partitions $\Pi = \{\pi_1, \pi_2, \dots, \pi_r\} \subseteq \Pi(V)$ and their associated binary relations $R_1, R_2, \dots, R_r$, the partition π' formed by the sets of vertices of maximal connected subgraphs of the union $R_1 \cup R_2 \cup \dots \cup R_r$ is the least upper bound of Π.*

Now we show that the least upper bound of two stable partitions is itself stable as well.

Theorem 4.14. *Given two stable partitions $\pi_1, \pi_2 \in \Pi_S(V)$, the least upper bound $\mathrm{lub}\{\pi_1, \pi_2\}$ is also stable.*

Proof. To prove: for all $\pi_1, \pi_2 \in \Pi_S$, $\pi_1 \sqcup \pi_2 \in \Pi_S$.

1) First we observe that because π_1 and π_2 are refinements of their least upper bound, the cells of $\pi_1 \sqcup \pi_2$ are unions of cells in π_1 and unions of cells in π_2.

2) Because the way the least upper bound is constructed there exists for each pair $v, w \in W$, $W \in \pi_1 \sqcup \pi_2$ a path
$$v = v_1, v_2, \dots, v_r = w$$
such that for all pairs $v_i, v_{i+1} (1 \le i < r)$, $\exists W' \in (\pi_1 \cup \pi_2) \cdot v_i, v_{i+1} \in W'$.

3) Because π_1 and π_2 are stable, each pair v_i, v_{i+1} has the same number of neighbouring elements for all cells of either π_1 or π_2, so certainly for unions of cells in π_1 or of cells in π_2.

4) Hence, by induction on r, the same holds for the pair v, w. So, $\pi_1 \sqcup \pi_2$ must also be stable. □

Definition 4.15 (Greatest lower bound of partitions)**.** Given a set of partitions $\Pi \subseteq \Pi(V)$ and an ordering relation $\sqsubseteq$, a *lower bound* is an element $\pi \in \Pi(V)$ such that for all elements $\rho \in \Pi$, $\pi \sqsubseteq \rho$. The *greatest lower bound* of Π, denoted $\mathrm{glb}\,\Pi$, is the upper bound π such that for all other lower bounds ρ, $\rho \sqsubseteq \pi$.

The greatest lower bound of a pair of elements $\pi_1, \pi_2 \in \Pi(V)$ is also called *meet*, denoted $\pi_1 \sqcap \pi_2$.

Proposition 4.16. *Given a set of stable partitions $\Pi = \{\pi_1, \pi_2, \dots, \pi_r\} \subseteq \Pi_S(V)$, there exists a stable greatest lower bound $\mathrm{glb}_S\{\pi_1, \pi_2, \dots, \pi_r\} \in \Pi_S(V)$.*

Proof. Let L be the set of stable lower bounds of Π:

$$L = \{\pi \in \Pi_S(V) \mid \forall \pi' \in \Pi \cdot \pi \sqsubseteq \pi'\}.$$

The least upper bound of this set of lower bounds, $\operatorname{lub} L$, is the stable greatest lower bound of Π, because

1) the least upper bound of a set of stable partitions is itself stable (Theorem 4.14);

2) $\operatorname{lub} L$ is a lower bound of Π, i.e., $\operatorname{lub} L \in L$;

3) $\operatorname{lub} L$ is an upper bound of the set L.

Hence, a stable greatest lower bound exists. □

Proposition 4.17. *Given a set of vertices V, the set of stable partitions $\Pi_S(V)$ forms a lattice under the refinement relation $\sqsubseteq$, $\langle \Pi_S(V), \sqsubseteq \rangle$.*

Proof. Both least upper bounds and greatest lower bounds exist, see Prop. 4.13 and 4.16 repectively. □

The existance of a greatest lower bound enables us to conclude the following.

Theorem 4.18. *For a directed graph G and initial partition $\pi \in \Pi(V)$, there is a* unique coarsest stable refinement*, i.e., a stable partition $\pi' \sqsubseteq \pi$, such that for all other stable partitions $\rho \sqsubseteq \pi$ it holds that $\rho \sqsubseteq \pi'$.*

Proof. Two parts:

1) The discrete partition is stable, so there is always a stable partitions that is a refinement of π.

2) Of the refinements of π there is one which is the coarsest, this is the greatest lower bound of π in $\Pi_S(V)$, given by Prop. 4.16. □

McKay's partition refinement algorithm This unique coarsest stable partition can be computed by applying the partition refinement algorithm presented by McKay, which is shown in Algorithm 4.1. An example of the partition refinement is given in Figure 4.2. The algorithm iterates over the sequence of potential splitters $W \in \alpha$. It searches for cells V for which W is a splitter, i.e., there exists $v_1, v_2 \in V$ for which the number of adjacent elements in W is not equal: $d(v_1, W) \neq d(v_2, W)$. In line 12 the cell V is split into cells X_i, which are ordered by the number of adjacent elements in W. This can be easily done by building an ordered map where each element $v \in V$ is added to the entry with $d(v, W)$ as key. V is replaced by one of the largest cells, the others are added (with their ordering maintained) to the sequence of potential splitters α. An example of this step is shown in Figure 4.3.

Paige & Tarjan [25] published a similar algorithm, with some small differences:

1) McKay uses ordered partitions, i.e., sequences of disjoint cells that form a partition, and Paige & Tarjan use sets of cells for partitions;

2) After splitting a cell into subcells, the algorithm of Paige & Tarjan leafs out the largest subcell when adding subcells to the sequence of splitters. The algorithm of McKay instead replaces the original cell in the sequence of splitters by the largest subcell (if the original cell is still in the queue of splitters, otherwise the largest subcell is left out).

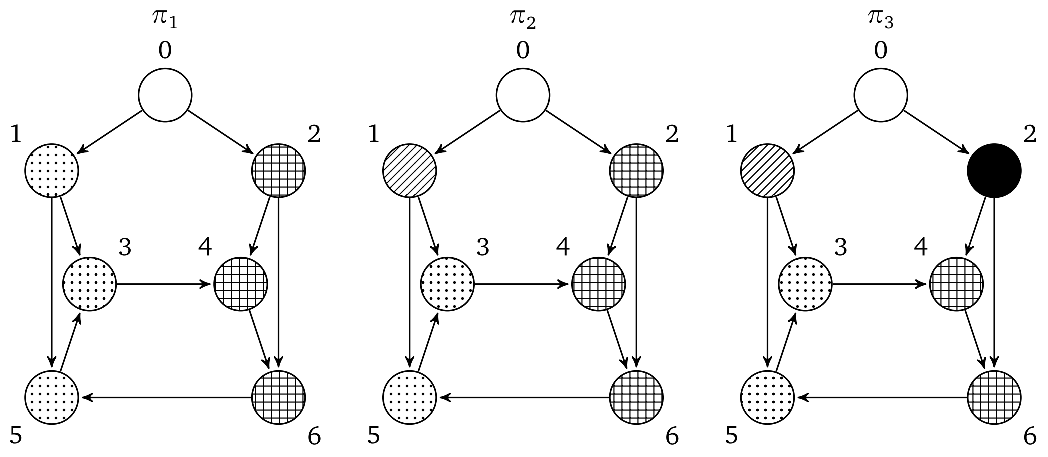
In effect, both algorithms implement a variant of the "process the smaller half" strategy of Hopcroft. Because of this the time complexity of the algorithms is $O(|E| \cdot \log|V|)$ (see [25]);

Proposition 4.19. *Given a graph G and a partition π of the vertices of G, refine(G, π, π) (Alg. 4.1) yields the coarsest stable partition of π for G.*

Proof. This has been proved in [19]. □

Algorithm 4.1 Compute the refinement *refine*(G, π, α) of an ordered partition $\pi \in \underset{\sim}{\Pi}(V)$ for a directed graph $G = \langle V, E \rangle$, given α, a sequence of cells in π that are used as splitters. *refine*(G, π, π) computes the coarsest stable partition of π for G.

```
$\pi' := \pi$
Let $\alpha'$ be a queue, initialised with the elements of $\alpha$
while $\alpha'$ is not empty do
    {Suppose $\alpha' = (W_1, W_2, \ldots, W_q)$ at this point.}
    if $\pi'$ is discrete then
        return $\pi'$;
    end if
    $W := W_1$
    Remove $W_1$ from $\alpha'$
    {Suppose $\pi' = (V_1, V_2, \ldots, V_r)$ at this point.}
    for $k := 1$; $k \le r$; $k{+}{+}$ do
        Define $\pi_k = (X_1, X_2, \ldots, X_s) \in \underset{\sim}{\Pi}(V_k)$ such that for all $v_1 \in X_i, v_2 \in X_j$, we have
          $d(v_1, W) < d(v_2, W) \iff i < j$.
        if $s > 1$ then
            $t := \min\{i \mid 1 \le i \le s \wedge |X_i| = \max\{|X_j| \mid X_j \in \pi_k\}\}$
              {the smallest integer $t$ such that $|X_t|$ is maximum (with $1 \le t \le s$)}
            if $\exists j$ such that $W_j = V_k$ (with $1 \le j \le q$) then
                $W_j := X_t$ {Replace $W_j$ in $\alpha'$ by $X_t$, the largest subcell of $W_j$}
            end if
            for $1 \le i < t$ and $t < i \le s$ do
                Add $X_i$ to the end of $\alpha'$
            end for
            Update $\pi'$ by replacing the cell $V_k$ with the cells $X_1, X_2, \ldots, X_s$ in that order (in situ).
        end if
    end for
end while
return $\pi'$;
```



(a) The initial partitions with cells $\{0\}$, $\{1, 3, 5\}$, and $\{2, 4, 6\}$.

(b) First the first cell is used as a splitter ($m = 1$, $W = \{v_0\}$). The second cell ($k = 2$, $V_k = \{1, 3, 5\}$) is split into two cells, $\{3, 5\}$ and $\{1\}$ (in that order), because v_1 has one incoming edge from 0 and the vertices 3 and 5 have no edges from or to 0.

(c) The cell $V_k = \{2, 4, 6\}$ ($k = 4$) is also split into two cells, $\{4, 6\}$ and $\{2\}$ (in that order), because 2 has one incoming edge from 0 and the vertices 4 and 6 have no edges from or to 0. The resulting partition is stable.

Figure 4.2: An example of partition refinement by Algorithm 4.1. Graph (a) shows the initial partition of the vertices. In (b) and (c) the result of subsequent splitting of cells is shown. The split steps are explained in Figure 4.3.

$$\begin{aligned}
\alpha' &= (\overbrace{\{0\}}^{W}, \{\underline{1}, \underline{3,5}\}, \{2,4,6\}) \\
\pi' &= (\{0\}, \underbrace{\{\underline{1}, \underline{3,5}\}}_{V_k}, \{2,4,6\}) \\
&\qquad\qquad \downarrow \\
X_1 &= \{3,5\}, \quad \forall v \in X_1 \cdot d(v,W) = 0 \\
X_2 &= \{1\}, \quad \forall v \in X_2 \cdot d(v,W) = 1 \\
&\qquad\qquad \downarrow \\
\alpha' &= (\overbrace{\{0\}}^{W}, \overbrace{\{3,5\}}^{X_1}, \{2,4,6\}, \overbrace{\{1\}}^{X_2}) \\
\pi' &= (\{0\}, \underbrace{\{3,5\}}_{X_1}, \underbrace{\{1\}}_{X_2}, \{2,4,6\})
\end{aligned}$$

(a) First step.

$$\begin{aligned}
\alpha' &= (\overbrace{\{0\}}^{W}, \{3,5\}, \{\underline{2}, \underline{4,6}\}, \{1\}) \\
\pi' &= (\{0\}, \{3,5\}, \{1\}, \underbrace{\{\underline{2}, \underline{4,6}\}}_{V_k}) \\
&\qquad\qquad \downarrow \\
X_1 &= \{4,6\}, \quad \forall v \in X_1 \cdot d(v,W) = 0 \\
X_2 &= \{2\}, \quad \forall v \in X_2 \cdot d(v,W) = 1 \\
&\qquad\qquad \downarrow \\
\alpha' &= (\overbrace{\{0\}}^{W}, \{3,5\}, \overbrace{\{4,6\}}^{X_1}, \{1\}, \overbrace{\{2\}}^{X_2}) \\
\pi' &= (\{0\}, \{3,5\}, \{1\}, \underbrace{\{4,6\}}_{X_1}, \underbrace{\{2\}}_{X_2})
\end{aligned}$$

(b) Second step.

Figure 4.3: The split steps done by Algorithm 4.1 for the coloured graph in Figure 4.2. The step with $V_k = \{0\}$ is not shown, because a singleton cell cannot be split (the resulting partition is $\pi_k = (X_1)$, with $X_1 = V_k = \{0\}$). (a) shows the splitting that corresponds to the transition from π_1 to π_2. (b) shows the splitting that corresponds to the transition from the π_2 to π_3, which is a stable partition.

4.2.2 A total ordering on coloured graphs

To determine the minimum of a set of coloured graphs we need a total ordering on $\mathcal{G}_{\mathcal{C}}$. In this section we define an ordering based on the number of vertices, number of edges, the colours of the vertices, and the adjacency matrix, which represents the incident edges. We denote the vertices as natural numbers, i.e., the set of vertices V is the set of numbers $\{1,2,\dots,n\} \subseteq \mathbb{N}$ with $n = |V|$, and use the natural ordering of $\mathbb{N}$ as ordering of the vertices. Now we define the adjacency matrix for coloured directed graphs.

Definition 4.20 (Adjacency matrix)**.** For a coloured directed graph $G = \langle V, E, c\rangle$, the *adjacency matrix* $A(G)$ is a $n \times n$ matrix ($n = |V|$) with for all $i, j \in [1..n]$,

$$A(G)_{i,j} = \begin{cases} 1, & \text{if } (i,j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

The ordering of adjacency matrices is based on concatenating the rows of the matrix ($A(G)_i$), for $1 \leq i \leq n$) which results in a binary number. As ordering of these number the usual natural ordering of numbers is used.

The colours of the vertices are compared as follows. For coloured graphs with n vertices, the colours can be represented as a sequence $(c(1), c(2), \dots, c(n))$. For comparing sequences of n colours we use lexicographical ordering, i.e., first the first elements are compared, if these are equal the second elements are compared, etc., until a difference in colour is found or all elements have been compared. In this section $c_{G,i}$ denotes the i-th elements of this sequence for G: $c_{G,i} = c_G(i)$ for $i \in V_G$. For two graphs $G = \langle V_G, E_G, c_G\rangle$ and $H = \langle V_H, E_H, c_H\rangle$ with $V_G = V_H = \{1,2,\dots,n\}$,

$$\begin{aligned}
&(c_{G,1}, c_{G,2}, \dots, c_{G,n}) < (c_{H,1}, c_{H,2}, \dots, c_{H,n}), \\
&\quad \text{if } c_{G,i} < c_{H,i} \text{ for the smallest } i \in [1..n] \\
&\quad \text{for which } c_{G,i} \neq c_{H,i}.
\end{aligned}$$

For coloured directed graphs we define an order relation $\leq$ as follows.

Definition 4.21 (Order relation $\leq$ on coloured graphs)**.** For all pairs of coloured graphs $G, H \in \mathcal{G}_{\mathcal{C}}$

with $G = \langle V_G, E_G, c_G \rangle$, $H = \langle V_H, E_H, c_H \rangle$, $V_G = \{1, 2, \ldots, n\}$, $V_H = \{1, 2, \ldots, m\}$

$$\begin{aligned}
&G \leq H, \\
&\quad \text{if } n < m \text{ or } \big(n = m \text{ and } (\\
&\qquad |E_G| < |E_H| \\
&\qquad \text{or } \big(|E_G| = |E_H| \text{ and } (\\
&\qquad\quad (c_{G,1}, c_{G,2}, \ldots, c_{G,n}) < (c_{H,1}, c_{H,2}, \ldots, c_{H,n}) \\
&\qquad\quad \text{or } \big(\forall_{1 \leq i \leq n}(c_{G,i} = c_{H,i}) \text{ and } A(G) \leq A(H)\big) \\
&\qquad)\big) \\
&\quad)\big).
\end{aligned}$$

Proposition 4.22. *$(\mathcal{G}_{\mathcal{C}}, \leq)$ is totally ordered.*

Proof. The number of vertices and number of edges of graphs are totally ordered. If the number of vertices is equal for two graphs, then also the corresponding sequences of colours are totally ordered (the cartesian product of a totally ordered set is itself also totally ordered). Because the adjacency matrix can be expressed as a natural number the adjacency matrices are also totally ordered. And because from this information (number of vertices, number of edges, sequence of colours, and adjacency matrix) the graph can be reconstructed (in other words, the information captures all there is to know about the graph), the combination of this information as defined in Def. 4.21 is a total ordering, i.e., without further proof we can say that the relation $\leq$ is

1) reflexive: $\forall G \in \mathcal{G}_{\mathcal{C}}, \quad G \leq G$;
2) antisymmetric: $\forall G, H \in \mathcal{G}_{\mathcal{C}}, \quad G \leq H \wedge H \leq G \implies G = H$;
3) transitive: $\forall G_1, G_2, G_3 \in \mathcal{G}_{\mathcal{C}}, \quad G_1 \leq G_2 \wedge G_2 \leq G_3 \implies G_1 \leq G_3$; and
4) total: $\forall G, H \in \mathcal{G}_{\mathcal{C}}, \quad$ either $G \leq H$ or $H \leq G$. □

4.2.3 Generating a search tree for a canonical relabelling partition

The partition refinement is used in the generation of a search tree that is used to find the discrete partition that will be used for a canonical relabelling of the vertices. The search tree is generated in the following way. Given an initial (refined) partition, a non-trivial (non-singleton) cell W is selected, of which one vertex is chosen that is deleted from the cell and put in a separate (singleton) cell. This is done for each of the vertices in the cell, resulting in $|W|$ different partitions. The new partitions are then refined, and then the same procedure is followed for the resulting partitions. This is repeated until the partitions are discrete. The procedure is shown in Alg. 4.2.

The result of the algorithm is a search tree of which the root node is the refinement of the initial partition, $\pi_1 = \mathit{refine}(G, \pi, \pi)$. The smallest non-trivial cell is selected and for each vertex v in that cell a subtree is added of which the root node is the refinement of the partition in which v is put in a separate singleton cell. An edge between the root node and the subtree is added, labelled v. The subtrees have the same structure. The nodes in the search tree are the intermediate refined partitions, resulting from individualising the non-trivial cells and refining the partitions. The edges of the search tree are labelled with the vertex that is isolated from its cell. The discrete partitions form the leaf nodes of the tree.

The search tree can be interpreted as sequences of traces from the root node to the discrete leaf nodes, defined as follows.

Definition 4.23 (Search tree). A *search tree* $T(G, \pi)$ is the set of all paths $(\pi_1, \pi_2, \ldots, \pi_m)$ that are derived from the directed graph G, an ordered partition π, and a sequence $v_1, v_2, \ldots, v_{m-1}$ where, for $1 \leq i \leq m-1$, v_i is an element of the first non-trivial cell V_k of π_i which has the smallest size:

$$\forall V_j \in \pi_i \cdot k \neq j \implies |V_k| < |V_j| \vee (|V_k| = |V_j| \wedge k < j).$$

The derivation is established in the following way. π_1 is the coarsest stable refinement of π, i.e., $\pi_1 = \mathit{refine}(G, \pi, \pi)$. The successors are defined in terms of their predecessors. π_{i+1} is derived from π_i and v_i by partition refinement such that $\pi_{i+1} = \mathit{refine}(G, \pi_i \downarrow v_i, (v_i))$, where $\pi_i \downarrow v$ is defined for $\pi_i = (V_1, V_2, \ldots, V_r)$ and $v \in V_k \in \pi_i$ as

$$\pi_i \downarrow v = (V_1, \ldots, V_{k-1}, \{v\}, V_k \setminus v, V_{k+1}, \ldots, V_r).$$

Algorithm 4.2 Generate the search tree $T(G,\pi)$ for a directed graph $G = \langle V, E\rangle$ and partition $\pi \in \underset{\sim}{\Pi}(V)$, where the nodes are partitions of V. The root node of the tree is the coarsest stable refinement of π and the leaf nodes are discrete partitions of V. The result is a list of paths in the tree, which is an alternative representation of the tree itself.

```
1: $k := 1$
2: $\pi_1 := refine(G, \pi, \pi)$
3: Let $W_1$ be the first non-trivial cell of $\pi_1$ of the smallest size.
4: Let $\tau$ be a list with only the singleton path $\pi_1$ as an element.
5: while $k \geq 1$ do
6:     if $\pi_k$ is discrete then
7:         $k := k - 1$
8:     end if
9:     if $k \geq 1$ then
10:        if $W_k \neq \emptyset$ then
11:            {The vertex identies are used to order the vertices.}
12:            $v := \min W_k$
13:            $W_k := W_k \setminus v$
               {Suppose $\pi_k = (V_1, V_2, \dots, V_r)$ and $v \in V_i$ at this point.}
14:            $\pi_k{}' := (V_1, \dots, V_{i-1}, \{v\}, V_i \setminus v, V_{i+1}, \dots, V_r)$
15:            $\pi_{k+1} := refine(G, \pi_k{}', (v))$
16:            $k := k + 1$
17:            Add the path $(\pi_1, \pi_2, \dots \pi_k)$ to $\tau$.
18:            Let $W_k$ be the first non-trivial cell of $\pi_k$ of the smallest size.
19:        else
20:            $k := k - 1$
21:        end if
22:    end if
23: end while
24: return $\tau$;
```

The relation between the partitions π_i and the vertices v_i is represented by the following notation:

$$\pi_1 \xrightarrow{v_1} \pi_2 \xrightarrow{v_2} \cdots \xrightarrow{v_{m-1}} \pi_m.$$

Proposition 4.24. *Given a graph G, a stable partition $\pi \in \Pi(V)$ and an element $v \in V$, refine$(G, \pi \downarrow v, (v))$ yields a stable partition and refine$(G, \pi \downarrow v, (v)) \sqsubseteq \pi$.*

Proof. This has been proved in [19]. □

Because π_1 is stable and because of Prop 4.24, all partitions π_i in the search tree have to be stable. Note that for all sequences $(\pi_1, \pi_2, \dots, \pi_m) \in T(G, \pi)$ it holds that

$$\pi_m \sqsubseteq \cdots \sqsubseteq \pi_2 \sqsubseteq \pi_1.$$

We write $X(G,\pi)$ for the set of all leaf nodes of $T(G,\pi)$, i.e., sequences of which the last element is a discrete partition. These ordered discrete partitions can be used as permutation of the vertices of the graph in the sense of Definition 4.6. We write π_λ for the discrete partition of leaf node λ and $\overline{\lambda}$ for the permutation associated with π_λ. For the set of all graphs resulting from the permutations that are generated by the search tree we write

$$P(G,\pi) = \{G^{\overline{\lambda}} \mid \lambda \in X(G,\pi)\}.$$

In [19, Theorem 2.14] it is stated that $T(G^\gamma, \pi^\gamma) = T(G,\pi)^\gamma$, in other words, for every sequence in $T(G,\pi)$ there is an equivalent sequence in $T(G^\gamma, \pi^\gamma)$. We reformulate and prove this property in the following lemma.

Lemma 4.25. *For two isomorphic coloured graphs G and G^γ ($\gamma \in S_V$),*

$$(\pi_1, \pi_2, \dots, \pi_m) \in T(G,\pi) \iff (\pi_1^\gamma, \pi_2^\gamma, \dots, \pi_m^\gamma) \in T(G^\gamma, \pi^\gamma).$$

Proof. First $(\pi_1, \pi_2, \dots, \pi_m) \in T(G,\pi) \implies (\pi_1^\gamma, \pi_2^\gamma, \dots, \pi_m^\gamma) \in T(G^\gamma, \pi^\gamma)$ (by induction). Consider the sequence

$$\pi_1 \xrightarrow{v_1} \pi_2 \xrightarrow{v_2} \cdots \xrightarrow{v_{m-1}} \pi_m.$$

1) Base step: $\pi_1 \in T(G,\pi) \implies \pi_1^\gamma \in T(G^\gamma,\pi^\gamma)$. For the initial partitions π and π^γ it holds that for all vertices $v \in V_G$, if v is in the i-th cell of π then v^γ is in the i-th cell of π^γ.

2) Induction step: If there exist $(\pi_1,\dots,\pi_k,\dots) \in T(G,\pi)$ and $(\pi'_1,\dots,\pi'_k,\dots) \in T(G^\gamma,\pi^\gamma)$ such that $\pi'_i = \pi_i^\gamma$ for $1 \le i \le k$, then π_k is discrete or there exist $(\pi_1,\dots,\pi_k,\pi_{k+1},\dots) \in T(G,\pi)$ and $(\pi'_1,\dots,\pi'_k,\pi'_{k+1},\dots) \in T(G^\gamma,\pi^\gamma)$ such that for $1 \le i \le k+1$, $\pi'_i = \pi_i^\gamma$.

   (a) Assume $(\pi_1,\dots,\pi_k,\dots) \in T(G,\pi)$ and $(\pi'_1,\dots,\pi'_k,\dots) \in T(G^\gamma,\pi^\gamma)$ such that $\pi'_i = \pi_i^\gamma$ for $1 \le i \le k$;

   (b) For the partitions π_k and π_k^γ it holds that for all vertices $v \in V_G$, if v is in the i-th cell of π_k then v^γ is in the i-th cell of π_k^γ;

   (c) This means that (if π_k is not discrete) the cell of π_k that is selected for the k-th iteration of generating $T(G,\pi)$ (the first smallest non-trivial cell), containing v_k, has an equivalent in π_k^γ that will be selected first in generating $T(G^\gamma,\pi^\gamma)$, which contains v_k^γ;

   (d) From this follows that $T(G^\gamma,\pi^\gamma)$ contains a branch starting with $\pi_k^\gamma \xrightarrow{v_k^\gamma}$;

   (e) The position of the cells of the resulting partition $\pi_k \downarrow v_k$ will still be equivalent to $\pi_k^\gamma \downarrow v_k^\gamma$;

   (f) If for the partitions $\pi_k \downarrow v_k$ and $\pi_k^\gamma \downarrow v_k^\gamma$ the partitions are equivalent, then also the stable partitions $\pi_{k+1} = \mathit{refine}(G,\pi_k \downarrow v_k)$ and $\pi_{k+1}^\gamma = \mathit{refine}(G^\gamma,\pi_k^\gamma \downarrow v_k^\gamma)$ are equivalent, i.e., it holds that for all vertices $v \in V_G$, if v is in the i-th cell of π_{k+1} then v^γ is in the i-th cell of π_{k+1}^γ, because partition refinement preserves isomorphism of isomorphic partitions;

   (g) Hence, either π_k and π_k^γ are discrete or there exist $(\pi_1,\dots,\pi_k,\pi_{k+1},\dots) \in T(G,\pi)$ and $(\pi'_1,\dots,\pi'_k,\pi'_{k+1},\dots) \in T(G^\gamma,\pi^\gamma)$ such that for $1 \le i \le k+1$, $\pi'_i = \pi_i^\gamma$.

Similar for the symmetric case: $(\pi_1,\pi_2,\dots,\pi_m) \in T(G,\pi) \Longleftarrow (\pi_1^\gamma,\pi_2^\gamma,\dots,\pi_m^\gamma) \in T(G^\gamma,\pi^\gamma)$. □

A consequence of the this lemma is that for isomorphic graphs with equivalent initial partitions equivalent leaf nodes are generated.

Lemma 4.26. *For two isomorphic coloured graphs G and G^γ ($\gamma \in S_V$),*

$$\pi_X \in X(G,\pi) \iff \pi_X^\gamma \in X(G^\gamma,\pi^\gamma).$$

Proof. $T(G^\gamma,\pi^\gamma) = T(G,\pi)^\gamma$ (Lemma 4.25) implies $X(G^\gamma,\pi^\gamma) = X(G,\pi)^\gamma$. □

The equivalence of the sets of leaf nodes means equivalence of the associated discrete partition, which results in equal graphs when used as permutation of the vertices.

Proposition 4.27. *If two graphs $G = \langle V, E_G, c_G\rangle$ and $H = \langle V, E_H, c_H\rangle$ are isomorphic, i.e., a function $\gamma \in S_V$ exists that maps vertices in G to vertices in H such that $G^\gamma = H$, then the sets of graphs resulting from permutations generated by the search tree contain exactly the same graphs, i.e., $P(G,\pi) = P(G^\gamma,\pi^\gamma)$.*

Proof. By Lemma 4.26. □

So, given a total ordering of graphs (see Section 4.2.2), we can choose the minimum of the set $P(G,\pi)$ as the canonical form of G.

4.2.4 Pruning the search tree

The number of leaf nodes in the search tree grows very large if the graph has a large automorphism group. The worst case complexity of the search tree generation is $O(|V|!)$. Therefore several heuristics need to be used to prune parts of the tree.

By choosing well the parts that we prune, we want to reduce the number of candidate graphs without losing the property of Prop. 4.27, i.e. that two isomorphic graphs will result in the same set of candidate graphs such that one graph is the minimum of both sets (which is the canonical form). There exist several methods to prune (large) parts of the search tree without losing the ability to compute a canonical form. The methods presented in [19] and [16] are based on finding automorphisms and on using leaf certificates and node invariants for nodes of the search tree.

Definition 4.28 (Leaf certificate)**.** For a leaf node $\lambda \in X(G,\pi)$, a *leaf certificate* $C(G,\pi,\lambda)$ is a certificate that maps leaf nodes to some value such that for all leaf nodes $\lambda_1, \lambda_2 \in X(G,\pi)$, and their associated partitions π_{λ_1} and π_{λ_2},

$$C(G,\pi,\lambda_1) = C(G,\pi,\lambda_2) \iff \exists \gamma \in \mathrm{Aut}(G,\pi) \text{ such that } \pi_{\lambda_1}^{\gamma} = \pi_{\lambda_2}.$$

A leaf certificate for which this holds is the combination of the permuted graph and the permuted initial partition: $C(G,\pi,\lambda) = \langle G^{\overline{\lambda}}, \pi^{\overline{\lambda}} \rangle$. The definition implies that if for two leaf nodes λ_1 and λ_2 the leaf certificates are equal, also the associated graphs are equal because there exists an automorphism $\gamma \in \mathrm{Aut}(G,\pi)$, such that

$$G^{\overline{\lambda_2}} = G^{\overline{\lambda_1^{\gamma}}} = (G^{\overline{\lambda_1}})^{\gamma} = G^{\overline{\lambda_1}}.$$

The automorphism implied by the equality of the certificates is determined by the leaf node partitions. If two leaf nodes $\lambda_1, \lambda_2 \in X(G,\pi)$ give rise to the same certificate, there exists an automorphism $\gamma = \overline{\lambda_1}\,\overline{\lambda_2}^{-1} \in \mathrm{Aut}(G,\pi)$. This will be used for detecting automorphisms in the graph during the generation of the search tree.

Definition 4.29 (Node invariant)**.** Given a node $v \in T(G,\pi)$ and the associated stable partition π_v, a *node invariant* $I(G,\pi,\pi_v)$ is an invariant such that for all graph permutations $\gamma \in S_V$,

$$I(G,\pi,\pi_v) = I(G^{\gamma},\pi^{\gamma},\pi_v^{\gamma}).$$

An example of such an invariant is an integer value based on the number of vertices in the cells of the partition π_v, e.g.,

$$I(G,\pi,\pi_v) = \prod_{W \in \pi_v} |W|.$$

Based on such a node invariant I also a new invariant can be defined that combines the invariant values of the nodes in a path of the search tree $v = (\pi_1, \pi_2, \ldots, \pi_l)$:

$$\vec{I}(G,\pi,v) = (I_1, I_2, \ldots, I_l),$$

where $I_i = I(G,\pi,\pi_i)$. This invariant will be used to select which paths are taken in the traversal of the search tree.

Pruning using automorphisms found

During the generation of the search tree, leaf nodes are encountered with associated discrete partitions. If we store the leaf nodes and the leaf certificates for these nodes, we can compute all automorphisms by comparing new certificates with the certificates stored. If for a graph $G = \langle V,E,c \rangle$ and initial partition $\pi \in \Pi(V)$, we discover a leaf node λ_1 with the same certificate as a stored leaf node λ_2, i.e., $C(G,\pi,\lambda_1) = C(G,\pi,\lambda_2)$, then there is an automorphism $\gamma = \overline{\lambda_1}\,\overline{\lambda_2}^{-1} \in \mathrm{Aut}(G,\pi)$. This automorphism induces an orbit partition, i.e., a partition $\{\{v\} \cup \{v^{\gamma}\} \mid v \in V\}$, which can be computed as follows. Let the two associated discrete partitions be $\pi_{\lambda_1} = (\{v_{1,1}\}, \{v_{1,2}\}, \ldots, \{v_{1,n}\})$ and $\pi_{\lambda_2} = (\{v_{2,1}\}, \{v_{2,2}\}, \ldots, \{v_{2,n}\})$. Then there is binary equivalence relation $\{(v_{1,1}, v_{2,1}), (v_{1,2}, v_{2,2}), \ldots, (v_{1,n}, v_{2,n})\}$ with $v_{1,i}^{\gamma} = v_{2,i}$ for all $1 \le i \le n$. The transitive closure of this relation can be represented as a partition, which we call the *orbit partition* Φ.

The orbit partition Φ is stored to be used in pruning parts of the search tree. The stored orbit partition is updated with new orbit partitions by computing the least upper bound of both (see Prop. 4.13 on how to determine the least upper bound of two partitions).

This orbit partition implies that in individualising a cell in the generation of the search tree (line 12 in Alg. 4.2), only one of the members of a cell of the orbit partition has to be considered. If we just have visited a branch in the search tree corresponding to vertex v, then we can skip branches from the same node if they correspond to a vertex that is in the same cell as v in orbit partition Φ.

Moreover, we can prune the whole subtree in which we found λ_1 up to its common ancestor with λ_2, the leaf node we stored earlier. This is visualised in Figure 4.4. There ϕ is the leaf node that is discovered before and λ is the newly discoved leaf node with the same leaf certificate as ϕ. The two leaf nodes having the same certificate implies that the two branches from their common ancestor in the search tree, which is μ in the figure, result from two individualised vertices that are automorphic.

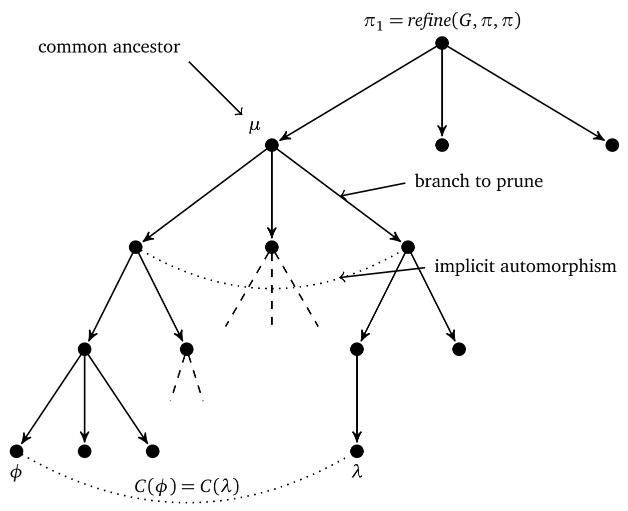


Figure 4.4: Pruning the search tree. The certificate values of the leaf nodes ϕ and λ are equal, so the branch of λ can be pruned from their common ancestor μ.

We can now prune the current subtree under the common ancestor, because the two branches result in the same values. This means we can backtrack to μ.

Because there can be a lot of leaf nodes, we do not want to store all leaf nodes and all leaf certificates. In Bliss two leaf nodes are stored [16]: the first leaf node we discovered, which is called ϕ, and the leaf node that leads to the 'best' candidate sofar, which is called ψ. The orbit partition Φ is stored for automorphisms with the first leaf node ϕ, and another one, Ψ for automorphisms with ψ. In Bliss also the m most recently found automorphisms are stored, where m can be set to some convenient value.

Pruning using leaf certificates and node invariants

Using the discrete partitions associated with the leaf nodes we can permute the vertices of the original graph (as discussed in the previous section), i.e., the partition π_λ associated with leaf node λ is used to generate graph $G^{\overline{\lambda}}$. Doing this for all leaf nodes results in the set of graphs $P(G, \pi)$ of which the minimum can be chosen as the canonical form of G.

However, the leaf certificates and node invariants defined earlier enable to consider a smaller set in the following way. The set of graphs to consider can be limited to the graphs that result from leaf nodes with a minimum leaf certificate:

$$\begin{aligned} P_C(G, \pi) = \{G^{\overline{\lambda}} \mid {} & \lambda \in X(G, \pi) \\ & \wedge C(G, \pi, \lambda) = \min\{C(G, \pi, \nu) \mid \nu \in X(G, \pi)\}\} \end{aligned}$$

This can be taken a step further by reducing the set to only include leaf nodes with a minimum node invariant:

$$\begin{aligned} P_I(G, \pi) = \{G^{\overline{\lambda}} \mid {} & \lambda \in X(G, \pi) \\ & \wedge C(G, \pi, \lambda) = \min\{C(G, \pi, \nu) \mid \nu \in X(G, \pi)\} \\ & \wedge \vec{I}(G, \pi, \lambda) = \min\{\vec{I}(G, \pi, \nu) \mid \nu \in X(G, \pi)\}\} \end{aligned}$$

Because of the iterative nature of the vector $\vec{I}(G, \pi, \lambda)$, parts of the search tree that do not have a minimum node invariant can be pruned early in the search tree. If $\vec{I}(\lambda_1) < \vec{I}(\lambda_2)$ for some nodes λ_1 and λ_2, then also for every descendant λ_1' of λ_1 and descendant λ_2' of λ_2 it holds that $\vec{I}(\lambda_1') < \vec{I}(\lambda_2')$. So, if λ_1 and λ_2 are descendants of the same node ν, only child λ_1 has to be considered, because the subtree from λ_2 will not lead to leaf nodes with a minimum node invariant. Note that the ordering of nodes by the node invariant need not to be complete, so multiple children of a node in the search tree can have the same node invariant.

The better the node invariant function is in discriminating nodes, the larger the part of the search tree that is pruned. Usually this results in a trade-off between time spent on calculating the invariant and the reduction of the search tree that is achieved by using the invariant.

4.3 Conversion from edge-labelled graphs to coloured graphs

In the algorithm explained in the previous section coloured graphs are used, while GROOVE uses edge labelled graphs (even node labels are labelled (self-)edges). In order to be able to use the existing algorithms, the edge labelled graphs have to be converted to coloured graphs.

We want to have a conversion function that preserves isomorphism of graphs, because then checking for isomorphism of the converted graphs yields the same result as checking for isomorphism of the original graphs. This enables the use of existing isomorphism checking algorithms that are built for coloured graphs also for edge labelled graphs.

First we formally define the properties of such functions. Then we present a conversion function τ that converts each edge label into a distinct coloured vertex. For this conversion a mapping between vertex colours and edge labels is maintained. An example is shown in Figure 4.5.

In Section 4.3.2 the difference in size of the resulting graphs from the conversion is discussed.

Definition 4.30 (Isomorphism preserving conversion function τ_L)**.** A function $\tau_L \colon \mathcal{G}_{\mathcal{L}} \to \mathcal{G}_{\mathcal{C}}$ is called an *isomorphism preserving conversion function* from edge labelled graphs to coloured graphs if for all $G, H \in \mathcal{G}_{\mathcal{L}}$,

$$G \cong H \iff \tau_L(G) \cong \tau_L(H).$$

If we have such an isomorphism preserving conversion function τ_L and we want to check if two edge labelled graphs G and H are isomorphic, then it suffices to check if $\tau_L(G)$ and $\tau_L(H)$ are isomorphic. Furthermore, if we want to store canonical forms to check if we have seen an isomorphic graph before, we can store the canonical form of the converted graphs. If *can* is a canonical representation function for coloured graphs, then

$$G \cong H \iff \tau_L(G) \cong \tau_L(H) \iff \mathit{can}(\tau_L(G)) = \mathit{can}(\tau_L(H)).$$

From now on we assume there to be a fixed, ordered set of labels $\mathcal{L} = \{l_1, l_2, \dots, l_k\}$ that is the same for the different graphs that are compared.

4.3.1 Label to vertex conversion τ

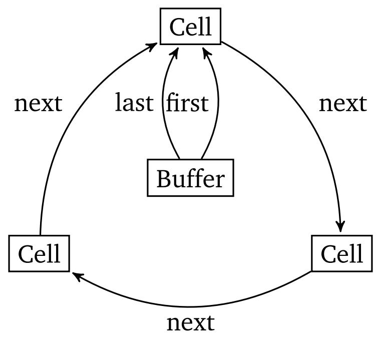
The 'label vertex' conversion creates a label vertex for each edge in the original graph with the colour corresponding to the label of the edge. Also for each vertex in the original graph a vertex is created in the resulting graph and edges are added between these vertices and the label vertices to connect the copies of the source and target vertices to the edges. For this conversion a mapping is maintained from edge labels to vertex colours. One special class of colours is reserved to designate vertices that represent the vertices in the original graph. These colours represent a set of vertex labels, i.e., the labels of the self-edges of the vertex in the original graph. Here again we assume the set of vertex labels and the set of edge labels to be disjoint. See Figure 4.5b for an example of the mapping and Figure 4.5c for the resulting converted graph.

Definition 4.31 ('Label vertex' conversion function τ)**.** $\tau \colon \mathcal{G}_{\mathcal{L}} \to \mathcal{G}_{\mathcal{C}}$ is a conversion function that maps each directed labelled graph $G = \langle V, E \rangle$ to a coloured graph $\tau(G) = G' = \langle V', E', c \rangle$ with

$$\begin{aligned} V' &= V \cup \{(v_1, l, v_2) \in E \mid v_1 \neq v_2\}, \\ E' &= \{(v_1, (v_1, l, v_2)) \mid (v_1, l, v_2) \in E\} \\ &\quad \cup \{((v_1, l, v_2), v_2) \mid (v_1, l, v_2) \in E\}, \\ c &= \{(v, (\mathsf{v}, \lambda_V(v))) \mid v \in V\} \\ &\quad \cup \{((v_1, l, v_2), (\mathsf{e}, l)) \mid (v_1, l, v_2) \in E, v_1 \neq v_2\}, \end{aligned}$$

where $\lambda_V \colon V \to \mathcal{P}(\mathcal{L})$ is the map from vertices to their associated set of vertex labels

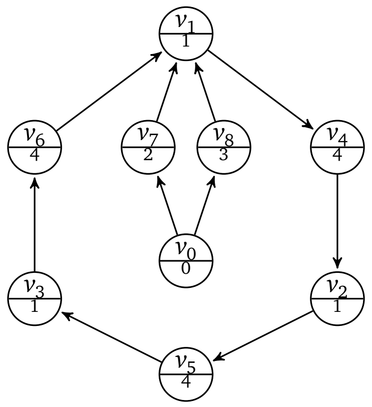
$$\lambda_V = \{(v, \{l \mid (v, l, v) \in E\}) \mid v \in V\}.$$



(a) An edge labelled graph G representing a bounded buffer. In the center is a 'Buffer' vertex, which points to the first and last cells that are occupied. The cells of the buffer, the vertices labelled 'Cell', are connected by 'next' edges.

{Buffer}	$\mapsto$	0
{Cell}	$\mapsto$	1
first	$\mapsto$	2
last	$\mapsto$	3
next	$\mapsto$	4

(b) The mapping between labels in the edge labelled graph and colours in the coloured graph, used by τ. Each edge label and each set of vertex labels that is used in the graph is mapped to a distinct colour.



(c) $\tau(G)$. For each vertex in the original graph there is a copy in this graph. The vertex colours (denoted by the number in the lower part of a vertex) are according to the mapping in (b). For each edge in G a label vertex is created with the colour according to the mapping from edge labels to colours. The copies of the source and target vertices are connected to the label vertices by unlabelled directed edges.

Figure 4.5: An edge labelled graph representing a bounded buffer, and its converted form.

Here, v is used to mark a colour as belonging to a vertex in the original graph and e to mark a colour as representing the label of an edge in the original graph. This is needed to be able to distinguish 'edge' vertices from 'vertex' vertices in the resulting graph.

Proposition 4.32. *For all $G, H \in \mathscr{G}_{\mathscr{L}}$, $G \cong H \iff \tau(G) \cong \tau(H)$, i.e., the conversion function τ is isomorphism preserving.*

Proof. See [17]. $\square$

4.3.2 Size of the converted graphs

For an edge labelled graph $G = \langle V, E \rangle \in \mathscr{G}_{\mathscr{L}}$, number of vertices $n = |V|$, number of non-self-edges $m = |E \setminus E_S|$ (with E_S being the set of self-edges of G), the label vertex conversion τ results in a coloured graph with $n + m$ vertices and $2 \cdot m$ edges. With this method the number of vertices and the number of edges is usually increased by the conversion. When there are a lot of self-edges, however, the number of edges may decrease.

The complexity of computing a canonical form is related to the size of the graph. Because typically the resulting graphs are larger than the original graphs, the computation time required for computing a canonical form may blow up as a result of the intermediate conversion steps. Therefore, an algorithm that works directly on edge labelled graphs is expected to perform better that an algorithm that first converts an edge labelled graphs to a coloured graph and than computes a canonical form of the coloured graph.

5

Serialisation of State Graphs

In this chapter we describe two methods for serialising graphs to state vectors that can be used in LTSmin. In Section 3.1.1 the state vectors as used in LTSmin have been discussed. In LTSmin state vectors are compressed to index vectors (see Section 3.1.2), which only works well if the number of different values for each part of the vector is much smaller than the total number of states. This influences the choice for a serialisation of graphs, because we want to benefit from the efficient state storage in LTSmin.

For the serialisation of graphs to state vectors we formulate the following requirements:

1) We require a globally unique serialisation of state graphs. This is necessary for communicating the states and to be able to decide which worker is responsible for computing successors for the state. From the vector representation we should be able to reconstruct the state graph (or an isomorphic graph).

2) The encoding and decoding should be fast, because otherwise the generated overhead could cancel out the possible gain by distributing the state space generation.

3) The state vector should have a fixed length that is known beforehand. It would be nice if the size of the vector is not too large, but there are no clear guidelines for what size is optimal.

4) The number of different values that each part can have should be small, i.e. they must be far smaller than the total number of states. The idea is that the total number of states comes from the number of combinations of values, not from the number of different values of the parts.

Because we need a globally unique representation of states and we do not want to introduce isomorphism checking in the store in LTSmin we choose to encode canonical forms of state graphs. In this chapter we describe the serialisations and demonstrate their correctness, i.e., that the serialisation is reversible (criterion 1). The speed of the serialisation (criterion 2) and the number of values per part (criterion 4) caused by the serialisation are subject of the experiments in Chapter 6.

5.1 Definitions

We now formally define the serialisation of the state graphs. We use state vectors as defined in Definition 3.1: a vector $\vec{p} = \langle p_0, p_1, \ldots, p_m \rangle$ with a predefined vector length m.

The state vector encoding is then defined as follows:

Definition 5.1 (State vector encoding of a state graph)**.** A *state vector encoding* function *enc* is a function that maps a given graph $G = \langle V, E, c \rangle$ to a sequence $enc(G) = \langle p_0, p_1, p_2, \ldots, p_m \rangle$, for a predefined vector length m.

Definition 5.2 (State graph decoding of a state vector)**.** A *state vector decoding* function *dec* is a function that maps a given state vector $\vec{p} = \langle p_0, p_1, p_2, \ldots, p_m \rangle$ to a graph $G \in \mathcal{G}$.

Definition 5.3 (Reversibility)**.** A state vector encoding function *enc* is *reversible* if a state vector decoding function *dec* exists such that for any graph $G \in \mathscr{G}$,

$$dec(enc(G)) \cong G.$$

Reversibility is defined in terms of isomorphism and not of equality because isomorphic state graphs give rise to equivalent behaviour (see Chapter 2). Actually, one of the deserialisation functions defined in this chapter (dec_L) renumbers nodes such that not necessarily for all G, $dec_L(enc_L(G)) = G$.

Vertices are represented as natural numbers, we assume the universe of labels to be totally ordered, and we use the natural ordering for the primitive values. For tuples of values, e.g., the edges, we use a lexicographical ordering.

We use $ord(S)$ for a set S with an implicit order to denote the ordered vector of elements of S, i.e., $ord(S) = \left(e_1, e_2, \ldots, e_r\right)$ with $r = |S|$, $e_i \in S$ for $1 \leq i \leq r$ and $e_i < e_j \iff i < j$.

For ordering sets of labels we use a bitset encoding and use the natural ordering of numbers. E.g., if $\mathscr{L}_{\mathcal{V}} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$, $ord(\mathscr{L}_{\mathcal{V}}) = (\mathsf{A}, \mathsf{B}, \mathsf{C})$, then a set $L_1 = \{\mathsf{A}, \mathsf{C}\}$ is represented as a sequence of bits $(1 \quad 0 \quad 1)$ and $L_2 = \{\mathsf{B}, \mathsf{C}\}$ is represented as $(0 \quad 1 \quad 1)$. In this case $L_2 < L_1$.

We use $\operatorname{ran} \alpha$ to denote the *range* or *image* of a function or mapping α.

5.2 Node vector serialisation

For defining a serialisation of graphs to vectors with a predefined fixed length, one needs to choose a way to partition elements in the graph such that each part of the graph is encoded in a slot of the vector encoding. One way of partitioning a graph is to assign a slot to each node of the graph. Here we describe a serialisation that is based on such a partitioning. Extra care needs to be taken to encode node labels, value nodes, and graphs with more nodes than there are slots available for encoding nodes, because the size of graphs is not necessarily fixed.

The *node vector serialisation*, as we call it, encodes node labels and primitive values of nodes in the first slot. Then four slots are used to encode the ordered sets of primitive values used in the graph. The remaining slots are used to encode the edges. Specifically, slot i is used to encode the edges with nodes numbered $i - 5 \mod (m - 5)$ as source. So, slot 5 will be used to encode edges from nodes *n0, n4, n8,* etc., if $m = 9$. This way the nodes are “wrapped around”, so that is is possible to use the encoding in a system where the size of graphs is not fixed. Formally the encoding is defined as follows.

Definition 5.4 (Node vector serialisation enc_N)**.** Given a graph $G = \left\langle V_G, E_G, c_G \right\rangle$ and $V_G = \{0, \ldots, n-1\}$, the *state vector encoding* $enc_N(G)$ is a sequence $\left(p_0, p_1, p_2, \ldots, p_m\right)$ with m the predefined vector length, and

$$\begin{aligned}
p_0 &= \left(col_G(0), col_G(1), \cdots, col_G(n-1)\right), \\
p_1 &= ord(\operatorname{ran} c_G \cap \mathsf{Int}), \\
p_2 &= ord(\operatorname{ran} c_G \cap \mathsf{Bool}), \\
p_3 &= ord(\operatorname{ran} c_G \cap \mathsf{String}), \\
p_4 &= ord(\operatorname{ran} c_G \cap \mathsf{Double}), \\
p_i &= \left(out_G(j_0), out_G(j_1), \ldots, out_G(j_r)\right) \\
&\qquad \text{for } 5 \leq i \leq m \text{ with} \\
&\qquad \left(j_0, j_1, \ldots, j_r\right) = ord(\{v \in V_G \mid i - 5 = v \mod (m-5)\}),
\end{aligned}$$

where

$$\begin{aligned}
col_G(v) &= \begin{cases} \langle type(c_G(v)), ind_G(c_G(v)) \rangle & \text{if } c_G(v) \in \mathsf{Val}, \\ c_G(v) & \text{otherwise,} \end{cases} \\
ind_G(c) &= k \text{ such that } val_k = val(c) \text{ with } c \in \mathsf{Val} \text{ and} \\
&\qquad \left(val_0, \ldots, val_k, \ldots, val_q\right) = ord(\operatorname{ran} c_G \cap type(c)), \\
out_G(v) &= ord(\{(l, w) \mid (v, l, w) \in E_G\}).
\end{aligned}$$

Example 5.5. We give an example of this encoding for the graph in Figure 5.1, which represents a state for the dining philosophers problem. In this case the fixed vector length is chosen to be 10,

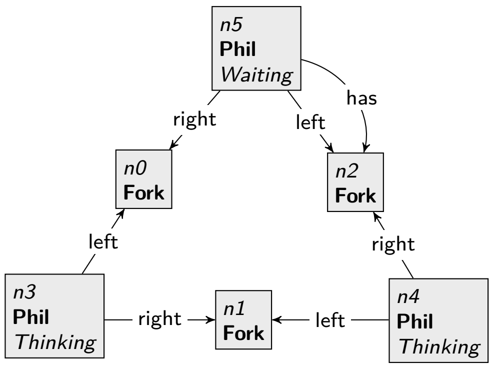


Figure 5.1: An example graph representing a state in the dining philosophers problem.

so wrapping of nodes has to be used. The encoding of the graph is in the following table. For the example we abbreviate **Phil** as **P**, **Fork** as **F**, *Thinking* as *T*, and *Waiting* as *W*.

node labels	p_0	{**F**}, {**F**}, {**F**}, {**P**, *T*}, {**P**, *T*}, {**P**, *W*}
Int	p_1	∅
Bool	p_2	∅
String	p_3	∅
Double	p_4	∅
n0, *n5*	p_5	∅ (⟨has, 2⟩, ⟨left, 2⟩, ⟨right, 0⟩)
n1	p_6	∅
n2	p_7	∅
n3	p_8	(⟨left, 0⟩, ⟨right, 1⟩)
n4	p_9	(⟨left, 1⟩, ⟨right, 2⟩)

The first slot (p_0) contains a sequence with for each node its set of node labels. The four following slots are empty, because this graph does not contain value nodes. Slot p_5 contains outgoing edges for both *n0* and *n5*. Node *n0* has no outgoing edges, which is encoded as ∅. Node *n5* has three outgoing edges: one labelled has with target node *n2*, one labelled left also to node *n2*, and one labelled right to node *n0*.

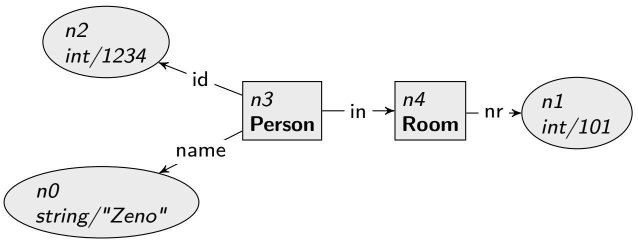


Figure 5.2: An example graph representing a person and a room. Primitive values are represented by attribute nodes.

Example 5.6. We give another example to demonstrate the encoding of attribute nodes. We use the graph in Figure 5.2, which models a person with a *name*, which is a string, an *id*, which is an integer, and a *location* that is modelled by a node labelled **Room**, which has a *number* (an integer). In this case the fixed vector length is chosen to be 10. In this case all node slots are used for only one node. The encoding of the graph is in the following table:

node labels	p_0	⟨String, 0⟩ , ⟨Int, 0⟩ , ⟨Int, 1⟩ , {**Person**} , {**Room**}
Int	p_1	101, 1234
Bool	p_2	∅
String	p_3	"Zeno"
Double	p_4	∅
n0	p_5	∅
n1	p_6	∅
n2	p_7	∅
n3	p_8	(⟨id, 2⟩ , ⟨in, 4⟩ , ⟨name, 0⟩)
n4	p_9	(⟨nr, 1⟩)

In slot p_0 the node types are encoded in the following way: Node *n0* is a value node representing the string "Zeno", which is the first in the sequence of strings used in the graph. Therefore the node is encoded as ⟨String, 0⟩, a tuple of the type of the node and the index of the value of the node within the sequence of values of that type. The sequence of string values is in p_3. Nodes *n1* and *n2* are encoded as ⟨Int, 0⟩ and ⟨Int, 1⟩ respectively, representing the integer values 101 and 1234. Nodes *n3* and *n4* are nodes with sets of node labels {**Person**} and {**Room**}. The slots $p_1, \ldots, p_4$ list the primitive values of types Int, Bool, String, and Double, used in the graph, ordered according to the natural orders of these types. The slots $p_5, \ldots, p_9$ list the edges from the nodes *n0*, …, *n4*.

We now define a deserialisation function dec_N.

Definition 5.7 (Node vector deserialisation dec_N). Given a state vector $s = \langle p_0, p_1, p_2, \ldots, p_m \rangle$, the *state vector decoding* $dec(s)$ is a graph $H = \langle V_H, E_H, c_H \rangle$ with for $n = |p_0|$ and $p_0 = (col_0, col_1, \ldots, col_{n-1})$,

$$V_H = \{0, 1, \ldots, n-1\},$$

$$c_H(v) = \begin{cases} \langle t, a_i \rangle & \text{if } col_v = \langle t, i \rangle \\ & \quad \text{with } (a_0, a_1, \ldots, a_q) = values(t), \\ col_v & \text{otherwise,} \end{cases}$$

$$E_H = \{(v, l, w) \in V_H \times \mathscr{L}_{\mathscr{E}} \times V_H \mid i \in \{5, \ldots, m\} \wedge v = (i-5) + (m-5) \cdot j \\ \wedge (e_0, e_1, \ldots, e_r) = p_i \wedge (l, w) \in e_j\}$$

where

$$values(t) = \begin{cases} p_1 & \text{if } t = \mathsf{Int}, \\ p_2 & \text{if } t = \mathsf{Bool}, \\ p_3 & \text{if } t = \mathsf{String}, \\ p_4 & \text{if } t = \mathsf{Double}. \end{cases}$$

This deserialisation function dec_N is the inverse of enc_N, which means that for any graph G, the deserialisation of its serialised form $dec_N(enc_N(G))$ is equal to G. This is stronger than the reversability that is defined in Definition 5.3. Reversibility of enc_N follows from invertibility.

Proposition 5.8 (Invertibility of enc_N). *The encoding function enc_N is invertible: $dec_N \circ enc_N = \mathrm{id}$ (for any $G \in \mathscr{G}$ and $H = dec_N(enc_N(G))$, $H = G$).*

Proof. For any $G \in \mathscr{G}$ and $H = dec_N(enc_N(G))$, assume $enc_N = \langle p_0, p_1, p_2, \ldots, p_m \rangle$ with $p_0 = (col_0, col_1, \cdots, col_{n-1})$.

1) $V_G = V_H = \{0, 1, \ldots, n-1\}$. The nodes in $V_G = \{0, 1, \ldots, n-1\}$ are represented in $enc_N(G)$ in the first part p_0 by n colours. In the decoded graph $dec_N(enc_N(G)) \cong G$, n nodes are created, based on the size of p_0, resulting in the set of nodes $V_H = \{0, 1, \ldots, n-1\}$.

2) for any $v \in V_G$, $c_G(v) = c_H(v)$:

   (a) either v is a *value node* ($c_G(v) \in \mathsf{Val}$):
   This implies $col_v = \langle type(c_G(v)), ind(c_G(v)) \rangle$ such that $ord(\operatorname{ran} c_G \cap type(c_G(v))) = (val_0, val_1, \ldots, val_q)$ and $val(c_G(v)) = val_i \iff ind(c_G(v)) = i$.
   The sequence $ord(\operatorname{ran} c_G \cap type(c_G(v)))$ is encoded in one of the slots $p_1, \ldots, p_4$ such that $values(type(c_G(v))) = ord(\operatorname{ran} c_G \cap type(c_G(v)))$ (as defined in dec_N).

By dec_N this results in $c_H(v) = \langle t, a_i \rangle$ such that $t = type(c_G(v))$, $i = ind(c_G(v))$, and $\left(a_0, a_1, \ldots, a_q\right) = values(type(c_G(v))) = ord(\operatorname{ran} c_G \cap type(c_G(v)))$. Hence $c_H(v) = c_G(v)$.

(b) or v is a node with a set of *node labels* ($c_G(v) \notin \mathsf{Val}$):
This implies $col_v = c_G(v)$ (by enc_N) and $c_H(v) = col_v$ (by dec_N).

3) for all $v, w \in V_G$ and $l \in \mathscr{L}_{\mathscr{E}}$, $\quad (v,l,w) \in E_G \iff (v,l,w) \in E_H$:

(a) $\implies$: each $(v,l,w) \in E_G$ is encoded as (l,w) into p_i in $out_G(j)$ such that $v = (i-5) + (m-5) \cdot j$, resulting in an edge $(v',l,w) \in E_H$ with $v' = (i-5) + (m-5) \cdot j = v$ (by dec_N).

(b) $\impliedby$: each $(v',l,w') \in E_H$ originates from e_j in p_i such that $(l,w') \in e_j$ and $v' = (i-5) + (m-5) \cdot j$, originating from an edge $(v,l,w) \in E_G$ with $v = (i-5) + (m-5) \cdot j$ and $w = w'$ (by enc_N), so $(v',l,w') = (v,l,w)$. □

Proposition 5.9 (Reversibility of enc_N). *The encoding function enc_N is reversible: for any $G \in \mathscr{G}$ and $H = dec_N(enc_N(G))$, $H \cong G$.*

Proof. This follows directly from Proposition 5.8. □

5.3 Label vector serialisation

Another way of partitioning a graph such that the graph can be encoded into a fixed length vector is to assign a slot to each edge label of the predefined set $\mathscr{L}_{\mathscr{E}}$. Here we describe a serialisation that is based on such a partitioning. Also here care needs to be taken to encode node labels and value nodes.

The *label vector serialisation*, as we call it, encodes nodes as a tuple of type and index. The types and numbers of nodes present in the graph are encoded in the first slot as tuples of the set of node labels or value type and the number of nodes in the graph with that label set or value type. Four slots are used to encode the ordered sets of primitive values used in the graph, as in the node vector serialisation. The remaining slots are used to encode the edges. Specifically, for all edge labels $ord(\mathscr{L}_{\mathscr{E}}) = \left(l_0, l_1, \ldots, l_s\right)$, slot i is used to encode the edges with label l_{i-5}. The source and target vertices of the edges are encoded with their set of node labels or value type and the index of that vertex in the ordered set of vertices with that set of node labels or value type. In this section we denote the set of value types and sets of node labels, together the *types*, as T^*:

$$T^* = \{\mathsf{Int}, \mathsf{Bool}, \mathsf{String}, \mathsf{Double}\} \cup \mathscr{P}(\mathscr{L}_{\mathscr{V}}).$$

Formally the encoding is defined as follows.

Definition 5.10 (Label vector serialisation enc_L). Given a graph $G = \langle V_G, E_G, c_G \rangle$, $V_G = \{0, \ldots, n-1\}$ and sequence of edge labels $ord(\mathscr{L}_{\mathscr{E}}) = \left(l_1, l_2, \ldots, l_q\right)$, the *state vector encoding* $enc_L(G)$ is a sequence $\left(p_0, p_1, p_2, \ldots, p_m\right)$, with $m = q + 5$ the predefined vector length,

$$\begin{aligned}
p_0 &= \left(\langle c_0, num_G(c_0)\rangle, \langle c_1, num_G(c_1)\rangle, \ldots, \langle c_r, num_G(c_r)\rangle\right) \\
&\quad \text{with } \left(c_0, c_1, \ldots, c_r\right) = ord(\{c \mid \exists v \in V_G \text{ such that } c = c_G(v)\}), \\
p_1 &= ord(\operatorname{ran} c_G \cap \mathsf{Int}), \\
p_2 &= ord(\operatorname{ran} c_G \cap \mathsf{Bool}), \\
p_3 &= ord(\operatorname{ran} c_G \cap \mathsf{String}), \\
p_4 &= ord(\operatorname{ran} c_G \cap \mathsf{Double}), \\
p_i &= out_G(l_{i-5}) \text{ for } 5 \le i \le m,
\end{aligned}$$

where

$$\begin{aligned}
num_G(c) &= |\{v \in V_G \mid c_G(v) = c\}|, \\
out_G(l) &= ord(\{\langle col_G(v), ind_G(v), col_G(w), ind_G(w)\rangle \mid (v,l,w) \in E_G\}), \\
col_G(v) &= \begin{cases} type(c_G(v)) & \text{if } c_G(v) \in \mathsf{Val}, \\ c_G(v) & \text{otherwise}, \end{cases} \\
ind_G(v) &= k \text{ such that } v = w_k \text{ with} \\
&\quad \left(w_1, \ldots, w_k, \ldots, w_q\right) = ord(\{w \in V_G \mid c_G(w) = c_G(v)\}).
\end{aligned}$$

Example 5.11. We give an example of this encoding for the graph in Figure 5.1. We assume that the predefined set of edge labels $\mathscr{L}_{\mathscr{E}}$ for the grammar consists of the labels has, left, and right. In the encoding of the edges, the vertices are represented by the set of node labels or value type of the vertex and the index of the vertex in the ordered set of vertices with that set of node labels or value type. In the next table we see this mapping from vertices in the example graph to the tuple resulting from the encoding:

v	$\langle col_G(v), ind_G(v)\rangle$
n0	$\langle\{$**Fork**$\}, 0\rangle$
n1	$\langle\{$**Fork**$\}, 1\rangle$
n2	$\langle\{$**Fork**$\}, 2\rangle$
n3	$\langle\{$**Phil**, *Thinking*$\}, 0\rangle$
n4	$\langle\{$**Phil**, *Thinking*$\}, 1\rangle$
n5	$\langle\{$**Phil**, *Waiting*$\}, 0\rangle$

There are three vertices with $\{$**Fork**$\}$ as set of node labels. The node identity is used for the ordering of the vertices in the ordered set of vertices with that set of node labels. The encoding of the graph is in the following table:

node labels	p_0	$\langle\{\mathbf{F}\}, 3\rangle, \langle\{\mathbf{P}, T\}, 2\rangle, \langle\{\mathbf{P}, W\}, 1\rangle$
Int	p_1	$\emptyset$
Bool	p_2	$\emptyset$
String	p_3	$\emptyset$
Double	p_4	$\emptyset$
has	p_5	$\langle\{\mathbf{P}, W\}, 0, \{\mathbf{F}\}, 2\rangle$
left	p_6	$\langle\{\mathbf{P}, T\}, 0, \{\mathbf{F}\}, 0\rangle, \langle\{\mathbf{P}, T\}, 1, \{\mathbf{F}\}, 1\rangle, \langle\{\mathbf{P}, W\}, 0, \{\mathbf{F}\}, 2\rangle$
right	p_7	$\langle\{\mathbf{P}, T\}, 0, \{\mathbf{F}\}, 1\rangle, \langle\{\mathbf{P}, T\}, 1, \{\mathbf{F}\}, 2\rangle, \langle\{\mathbf{P}, W\}, 0, \{\mathbf{F}\}, 0\rangle$

In slot p_0 the node types are encoded in the following way: the tuple $\langle\{\mathbf{F}\}, 3\rangle$ means that there are three nodes in the graph with the set of node labels $\{\mathbf{F}\}$; $\langle\{\mathbf{P}, T\}, 2\rangle$ that there are two node with the set of node labels $\{\mathbf{P}, T\}$; etc. The slots $p_1, \dots, p_4$ list the primitive values of types Int, Bool, String, and Double, used in the graph, ordered according to the natural orders of these types (in this case there are none). The slots $p_5, \dots, p_7$ list the edges with labels $l_0, \dots, l_2$, where each edge is represented as a tuple $\langle c_1, i_1, c_2, i_2\rangle$, where c_1 is the set of node labels of the source vertex and i_1 the index of that vertex within the ordered set of vertices with that label and c_2 and i_2 similar for the target vertex. The edge $(5, \mathsf{has}, 2)$, for instance, is represented in slot p_5 as $\langle\{\mathbf{P}, W\}, 0, \{\mathbf{F}\}, 2\rangle$, in accordance with the mapping from nodes to tuples of the set of node labels and index shown in the table above.

Example 5.12. We use the graph in Figure 5.2 to demonstrate the encoding of attribute nodes in the label vector serialisation. In the next table we show the mapping from vertices to the tuple of set of node labels or value type and index resulting from the encoding:

v	$c_G(v)$	$\langle col_G(v), ind_G(v)\rangle$
n0	$\langle$String, "Zeno"$\rangle$	$\langle$String, 0$\rangle$
n1	$\langle$Int, 101$\rangle$	$\langle$Int, 0$\rangle$
n2	$\langle$Int, 1234$\rangle$	$\langle$Int, 1$\rangle$
n3	$\{$**Person**$\}$	$\langle\{$**Person**$\}, 0\rangle$
n4	$\{$**Room**$\}$	$\langle\{$**Room**$\}, 0\rangle$

The index for value nodes is based on the natural ordering of the value type. The index for the other vertices is based on the vertex identity. The encoding of the graph is in the following table:

node labels	p_0	$\langle \mathsf{Int}, 2\rangle, \langle \mathsf{String}, 1\rangle, \langle \{\textbf{Person}\}, 1\rangle, \langle \{\textbf{Room}\}, 1\rangle$
Int	p_1	$101, 1234$
Bool	p_2	$\varnothing$
String	p_3	"Zeno"
Double	p_4	$\varnothing$
id	p_5	$\langle \{\textbf{Person}\}, 0, \mathsf{Int}, 1\rangle$
in	p_6	$\langle \{\textbf{Person}\}, 0, \{\textbf{Room}\}, 0\rangle$
name	p_7	$\langle \{\textbf{Person}\}, 0, \mathsf{String}, 0\rangle$
nr	p_8	$\langle \{\textbf{Room}\}, 0, \mathsf{Int}, 0\rangle$

The encoding of numbers of vertices for each type in p_0 is straightforward as in the previous example. The encoding of primitive values in $p_1, \ldots, p_4$ is the same as in the node vector serialisation. The difference with the previous example is in the encoding of value nodes. In the encoding of the edges, the index of vertices with node labels is based on the vertex identity and the index of value nodes is based on the ordering of the values used in the graph. The edge $\langle \{\textbf{Room}\}, 0, \mathsf{Int}, 0\rangle$ in p_8, for instance, represents the edge with label nr from the first vertex with the set of node labels $\{\textbf{Room}\}$ (of which there is only one in this case) to the first value node of type Int, the value 101 in this case.

In order to demonstrate the reversibility of enc_L we need to define a deserialisation function dec_L.

Definition 5.13 (Label vector deserialisation dec_L). Given a state vector $s = \langle p_0, p_1, p_2, \ldots, p_m\rangle$, the *state vector decoding* $dec_L(s)$ is a graph $H = \langle V_H, E_H, c_H\rangle$ with for

$$p_0 = (\langle col_0, num_0\rangle, \langle col_1, num_1\rangle, \ldots, \langle col_r, num_r\rangle),$$

$$\begin{aligned}
V_H &= \{0, 1, \ldots, n-1\} \\
&\quad \text{with } n = |\vec{v}(0)| + |\vec{v}(1)| + \cdots + |\vec{v}(r)|, \\
c_H(v) &= \begin{cases} \langle c, a_i\rangle & \text{if } c \in \{\mathsf{Int}, \mathsf{Bool}, \mathsf{String}, \mathsf{Double}\} \\ & \quad \text{with } \left(a_0, a_1, \ldots, a_q\right) = values(c), \\ c & \text{otherwise,} \end{cases} \\
&\quad \text{with } c \in T^* \text{ and } i \in \mathbb{N} \text{ such that } map(c, i) = v, \\
E_H &= \left\{ \left(map(c_1, i_1), l_{j-5}, map(c_2, i_2)\right) \in V_H \times \mathscr{L}_{\mathscr{G}} \times V_H \mid 5 \le j \le m, \langle c_1, i_1, c_2, i_2\rangle \in p_j \right\} \\
&\quad \text{with } (l_0, l_1, \ldots, l_s) = ord(\mathscr{L}_{\mathscr{G}}),
\end{aligned}$$

where

$$\begin{aligned}
\vec{v}(i) &= (map(c, 0), map(c, 1), \ldots, map(c, q)) \\
&\quad \text{with } c = col_i \text{ and } q = num_i, \\
map(c, i) &= i + \sum_{j=0}^{k-1} (num_j) \quad \text{with } c = col_k, \\
values(t) &= \begin{cases} p_1 & \text{if } t = \mathsf{Int}, \\ p_2 & \text{if } t = \mathsf{Bool}, \\ p_3 & \text{if } t = \mathsf{String}, \\ p_4 & \text{if } t = \mathsf{Double}. \end{cases}
\end{aligned}$$

The interesting part of this conversion from state vector to graph is the node numbering, specified by the function $map\colon T^* \times \mathbb{N} \to \mathbb{N}$. map assigns a unique number (a node identity) in the range $0, \ldots, n-1$ to each combination of 'colour' c and index i with $c = col_j$ and $0 \le i \le num_j$ for each j with $0 \le j \le r$ (i.e., each combination that is used in the state vector encoding). $\vec{v}(i)$ specifies the sequence of nodes with the type col_i.

Proposition 5.14 (Reversability of enc_L). *The encoding function enc_L is reversible: for any $G \in \mathscr{G}$ and $H = dec_L(enc_L(G))$, $H \cong G$.*

Proof. For any $G \in \mathcal{G}$ and $H = dec_L(enc_L(G))$, let there be two functions $g\colon V_G \to T^* \times \mathbb{N}$ and $h\colon T^* \times \mathbb{N} \to V_H$ defined as

$$g = \{v \mapsto \langle c, i\rangle \mid v \in V_G, c = col_G(v), i = ind_G(v)\}$$

with col_G and ind_G as defined in enc_L;

$$h = \left\{\langle c, i\rangle \mapsto v' \mid \exists i, j \cdot 0 \le j \le r, 0 \le i \le num_j, c = col_j, v' = map(c, i)\right\}$$

with $\langle col_0, num_0\rangle, \ldots, \langle col_r, num_r\rangle$ and map as defined in dec_L.

Then $f : V_G \to V_H$ with $f = h \circ g$ is an isomorphism, because

1) for any $v \in V_G$, $c_G(v) = c_H(f(v))$:

   (a) either v is a *value node* ($c_G(v) \in \mathsf{Val}$):
   This implies $col_v = \langle type(c_G(v)), ind(c_G(v))\rangle$ such that $ord(\operatorname{ran} c_G \cap type(c_G(v))) = \left(val_0, val_1, \ldots, val_q\right)$ and $val(c_G(v)) = val_i \iff ind(c_G(v)) = i$.
   The sequence $ord(\operatorname{ran} c_G \cap type(c_G(v)))$ is encoded in one of the slots $p_1, \ldots, p_4$ such that $values(type(c_G(v))) = ord(\operatorname{ran} c_G \cap type(c_G(v)))$ (as defined in dec_L).
   By dec_L, $g(v) = \langle c, i\rangle$ such that $c = type(c_G(v))$, $i = ind(c_G(v))$. c_H is defined such that $c_H(h(\langle c, i\rangle)) = \langle c, a_i\rangle$ with $\left(a_0, a_1, \ldots, a_q\right) = values(c) = ord(\operatorname{ran} c_G \cap c) = ord(\operatorname{ran} c_G \cap type(c_G(v)))$ if $c \in \{\mathsf{Int}, \mathsf{Bool}, \mathsf{String}, \mathsf{Double}\}$, which is the case if $c_G(v) \in \mathsf{Val}$. Hence $c_H(f(v)) = c_G(v)$ if $c_G(v) \in \mathsf{Val}$.

   (b) or v is a node with a set of *node labels* ($c_G(v) \notin \mathsf{Val}$):
   By enc_L, $g(v) = \langle c, i\rangle$ with $c = type(c_G(v))$. c_H is defined such that $c_H(h(\langle c, i\rangle)) = c$ if $c \notin \{\mathsf{Int}, \mathsf{Bool}, \mathsf{String}, \mathsf{Double}\}$, which is the case if $c_G(v) \notin \mathsf{Val}$. Hence $c_H(f(v)) = c_G(v)$ if $c_G(v) \notin \mathsf{Val}$.

2) for all $v, w \in V_G$ and $l \in \mathcal{L}_{\mathcal{E}}$, $(v, l, w) \in E_G \iff \left(f(v), l, f(w)\right) \in E_H$.

   (a) $\Longrightarrow$: each edge $(v, l, w) \in E_G$ is encoded into a tuple $\langle c_1, i_1, c_2, i_2\rangle$ in part p_j such that $g(v) = \langle c_1, i_1\rangle$ and $g(w) = \langle c_2, i_2\rangle$ and $l = l_{j-5}$ (for $ord(\mathcal{L}_{\mathcal{E}}) = l_0, \ldots, l_s$); each tuple $\langle c_1, i_1, c_2, i_2\rangle$ in p_j results in an edge $\left(map(c_1, i_1), l_{j-5}, map(c_2, i_2)\right)$ in E_H, with $h(\langle c_1, i_1\rangle) = map(c_1, i_1)$ and $h(\langle c_2, i_2\rangle) = map(c_2, i_2)$. So, for each $(v, l, w) \in E_G$ there is an edge $\left(h(g(v)), l, h(g(w))\right) = \left(f(v), l, f(w)\right) \in E_H$.

   (b) $\Longleftarrow$: each edge $(v', l, w') \in E_H$ originates from a tuple $\langle c_1, i_1, c_2, i_2\rangle \in p_j$ with $l = l_{j-5}$, $v' = map(c_1, i_1) = h(\langle c_1, i_1\rangle)$ and $w' = map(c_2, i_2) = h(\langle c_2, i_2\rangle)$. Each tuple $\langle c_1, i_1, c_2, i_2\rangle \in p_j$ originates from an edge $(v, l_{j-5}, w) \in E_G$ with $g(v) = \langle c_1, i_1\rangle$ and $g(w) = \langle c_2, i_2\rangle$. So, for each $(v', l, w') \in E_H$ there is an edge $\left(f^{-1}(v'), l, f^{-1}(w')\right) \in E_G$. □

6

Experiments

We have carried out experiments to compare the performance of the distributed tool based on LTSMIN, presented in this work, to the existing sequential version of the graph-based state space generation tool GROOVE. In the distributed setup, GROOVE is used to compute successor states. There are, however, some notable differences between sequential GROOVE and the way GROOVE is used in the distributed setup. These differences are described in Section 6.1.

Two different serialisations are used and compared: the serialisation functions enc_N and enc_L that are described in Chapter 5. Different series of input models have been used, which are described in Section 6.2, together with other experiment parameters. In the experiments both execution time and memory usage have been measured. The results are presented in Section 6.3. Their meaning is interpreted in Section 6.4.

6.1 GROOVE

In this section we describe some features of GROOVE [28] that are not applicable in the distributed setting. These features give the sequential version of GROOVE a head start in the comparison to the distributed tool.

6.1.1 Isomorphism checking

Isomorphism reduction is established in GROOVE by an algorithm that directly checks isomorphism between two graphs. The algorithm is based on iteratively computing so-called node certificates and applying partition refinement (similar to the partition refinement algorithm by Paige & Tarjan [25]). The isomorphism checking is done when a new state is encountered. It has to be checked if the store of visited states contains a graph isomorphic to the newly encountered state. Instead of comparing the new graph to all graphs in the store, the graph is only compared to graphs with the same *canonical hash code*.

Definition 6.1 (Canonical hash function)**.** For a graph $G \in \mathcal{G}$, a *canonical hash function* or *invariant hash function* is a function $hash\colon \mathcal{G} \to \mathbb{N}$ that associates an integer value, called *hash code*, with each graph such that for every pair of graphs $G, H \in \mathcal{G}$,

$$hash(G) = hash(H) \text{ if } G \cong H.$$

The store of visited states is implemented as a hash set. Each visited state q is stored at position $hash(q)$. Because the hash function is canonical, a newly encountered state s only has to be compared to the states at position $hash(s)$. If the hash function is very good (i.e., gives rise to little hash collisions for similar but not isomorphic graphs), this means that the new state has to be compared to only a small number or even no states. Isomorphism checking and the use of a canonical hash function in GROOVE are further described in[26].

The canonical hash code in GROOVE is very cheap to compute compared to isomorphism checking or computing a canonical form. In sequential GROOVE the use of a canonical hash code can often prevent that the more expensive isomorphism checking has to be performed. In the distributed setting, however, always a canonical form has to be computed before communicating a state.

6.1.2 Partial Order Reduction for Graphs

If two rule application on the same graph are independent of each other, i.e., the one rule does not create or remove elements that are used by the other and vice versa, the order in which they are executed does not matter. This is called *local confluence* or *parallel independence* and gives rise to a reduction technique called *partial order reduction*, which is explained for graphs in [26].[1] The idea is as follows. If from state q_0 the transitions $q_0 \xrightarrow{r_1,m_1} q_1$ and $q_0 \xrightarrow{r_2,m_2} q_2$ (derived from the rules r_1 and r_2 and matches m_1 and m_2) are independent of each other, then r_2, m_2 is also applicable to q_1, resulting in the transition $q_1 \xrightarrow{r_2,m_2} q_3$. It then follows that there has to be a transition $q_2 \xrightarrow{r_1,m_1} q_3$. This last rule application does not have to be computed, but is available for free. The partial state space consisting of states q_0, q_1, q_2, and q_3 and the described transitions form a *confluent diamond*, which is depicted below. The inferred transition $q_2 \xrightarrow{r_1,m_1} q_3$ is represented by a dotted line.

$$\begin{tikzcd}
 & q_0 \arrow[dl, "r_1, m_1"'] \arrow[dr, "r_2, m_2"] & \\
q_1 \arrow[dr, "r_2, m_2"'] & & q_2 \arrow[dl, dotted, "r_1, m_1"] \\
 & q_3 &
\end{tikzcd}$$

The same technique can be applied to more than one transition, resulting in larger confluent diamonds.

In sequential GROOVE during state space generation, the complete transition system generated sofar is available in memory. In the distributed setting, however, each worker only has fragments of the transition system available: DG alternatingly receives a state from LTSMIN and sends its successor states back to LTSMIN and the states it receives are not necessarily direct successors of states that it has sent to LTSMIN before, they might as well originate from other workers. Because of this the partial order reduction for graph transition systems is not directly applicable in the distributed setting. Each worker should have efficient access to a larger part of the generated LTS for such a technique to work efficiently.

When a transition is inferred by the partial order reduction in sequential GROOVE, the target state graph does not have to be computed or to be looked up in the store of visited states.

6.2 Experiment setup

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

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`).

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`).

[1]Actually, partial order reduction usually means that of the paths of independent actions resulting in the same state only one is explored, thus reducing the size of the generated state space. In GROOVE all paths are explored, but they are available more easily, thus reducing the required time. See, e.g., [2] for more on the usual partial order reduction for labelled transition systems.

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 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 with the node vector encoding; 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.

6.3 Results

Table 6.1 shows some global results for the three cases, using the largest start graphs for which the sequential setting could compute the entire state space. The distributed setup is referred to as LTSMIN.

Comparison of encodings. From the table it is clear that the size of the leaf database with the label vector encoding (enc_L) is dramatically larger than with the node vector encoding (enc_N) in the car platooning and leader election cases. In the car platooning case the size of the leaf database is even larger than the number of states with the label vector encoding, while it is orders of magnitude smaller for the node vector encoding. Also in the append case the node vector encoding is the better one.
However, the leaf databases of the append rule system grows much larger than for the other cases with the node vector encoding, despite the fact that the state space is much smaller. This is a consequence of the fact that the graph size outgrows the fixed vector size for this case. The difference between the two encodings is much smaller for that case.

The difference in execution time for the different encodings is not very large, but the distributed setup is always faster with the node vector encoding than with the label vector encoding. In the leader election case, LTSMIN with 64 cores and label vector is not able to finish within the time limit, while the same setup with node vector encoding results in a nice speedup. This is probably due to communication overhead caused by the large number of leaf values that have to be communicated between workers with the label vector encoding.

Memory usage. The size of the leaf database is reflected in the memory usage, which is much larger for LTSMIN with the label vector encoding and the sequential version of GROOVE than for LTSMIN with the node vector encoding. Note that although the memory usage in the table is the average per worker, with 8 cores also the total memory usage of LTSMIN with node vector encoding is better than that of GROOVE for the car platooning and leader election cases. In all reported cases the average memory usage per worker is best for LTSMIN with node vector encoding: 23 times better than GROOVE for the car platooning case and even 52 times better for the leader election case (both with 8 cores).

So, in the distributed setup, both memory usage and time performance are better with the node vector encoding than with the label vector encoding. Compared to GROOVE, memory performance is best for the distributed setup with node vector encoding.

Time performance. Time performance of LTSMIN with a single core is quite bad compared to GROOVE, taking in the order of 6-10 times as much time. For the leader election case, the single core LTSMIN even cannot compute the state space within the time limit of four hours, while sequential GROOVE can.

Table 6.1: 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
unflagged-platoon	1.580.449	GROOVE	1	1.016		1.259	
start-10	10.200.436	LTSMIN	1	6.621	0,2	120	
		node vector	8	889	1,1	54	
			64	156	6,5	54	8.451
		LTSMIN	1	6.897	0,1	1.355	
		label vector	8	1.551	0,7	791	
			64	503	2,0	556	1.653.536
le	3.724.544	GROOVE	1	5.128		2.751	
start-7p	16.956.727	LTSMIN	1	–	–	–	
		node vector	8	2.005	2,6	52	
			64	307	16,7	52	1.815
		LTSMIN	1	–	–	–	
		label vector	8	2.820	1,8	888	
			64	–	–	–	1.584.584
append	261.460	GROOVE	1	202		372	
append-4-list-8	969.977	LTSMIN	1	1.912	0,1	105	
		node vector	8	352	0,6	76	
			64	92	2,2	70	69.117
		LTSMIN	1	2.071	0,1	252	
		label vector	8	402	0,5	169	
			64	105	1,9	141	148.977

However, we can observe that with 8 cores, the distributed setting with node vector encoding starts to outperform sequential GROOVE for the car platooning and leader election cases, 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. Only for the append case the 8 core distributed tool is slower than GROOVE. Even with 64 cores we see only a speedup of 2,2 for the append case.

The table shows results for the largest cases where GROOVE finished within the time limit and had enough memory. For the car platooning and leader election cases there are larger start graphs for which the distributed setup could still generate the state space. In the next sections we look in more detail to the various cases, where also the larger start graphs are considered.

6.3.1 Car platooning

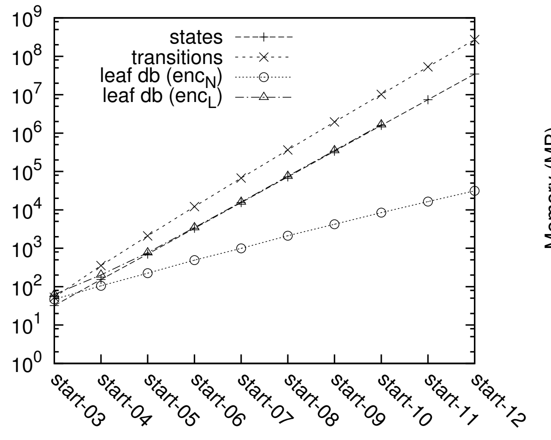
Time and memory distributions. For the car platooning case, more detailed results are shown in Figures 6.1 and 6.2. First of all, Figure 6.1a shows that even though the size of the problem grows exponentially, the number of elements of the leaf value database of LTSMIN does not. This is also reflected by the per-core memory usage of LTSMIN (Figure 6.1b), which seems hardly to grow, in contrast to the more than exponentially growing memory usage of GROOVE. This is caused partly by the initial database size of about 50MB used by LTSMIN, but also when this initial size is not sufficient (e.g., from `start-09` for 1 core and enc_N), the growth is slower than that for GROOVE. Memory usage grows much slower for LTSMIN with enc_N than with enc_L. With 64 workers, for `start-12` the memory usage with enc_N has hardly grown, while for `start-10` memory usage with enc_L is already more than ten times the memory usage with enc_N for the same system.

Figures 6.1c and 6.1d show the execution time respectively the speedup of LTSMIN with the node vector encoding (enc_N) and 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, GROOVE cannot generate the state-space within 4 hours, but LTSMIN with 8 respectively 16 or more cores can. Figure 6.1d also clearly shows that the larger the problem is, the larger is also the speedup. Figures 6.1e and 6.1f show the execution times and speedup for the car platooning case using the label vector encoding (enc_L). The performance with this serialisation is worse and seems to scale less. For 64 cores, the node vector serialisation results in a speedup that is three times as good as that of the label vector serialisation. With both cases there seems to be a limit to the speedup, an optimal number of cores, from which the speedup does not increase anymore. This optimum seems to be larger for larger models, so the larger the model the larger the number of cores that is sensible to use. With node vector encoding the optimum number of cores seems to be 48 for `start-09` and 56 for `start-10`.

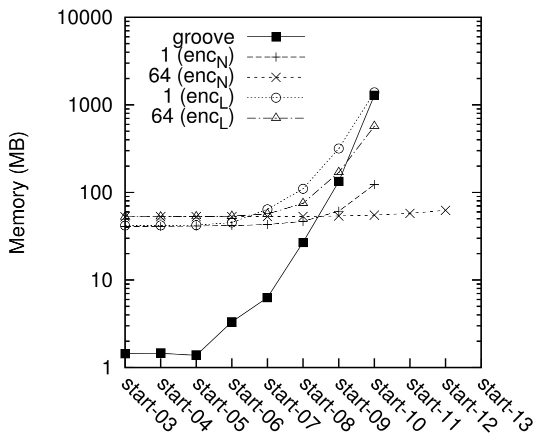
Execution time decomposition. Figure 6.2 shows how the execution time is built up. For smaller start states, the communication between cores (labelled "send/recv" in the figure) is a major factor in the computation time for LTSMIN, but for larger start states most of the time is spent on computing canonical forms (isomorphism reduction). As the number of cores grows, however, the communication again starts to play a larger relative role — 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. Comparing Figures 6.2e and 6.2f shows that the difference in performance between the node vector encoding and the label vector encoding is mainly due to the difference in communication overhead, caused by the leaf values that have to be communicated between the workers.

6.3.2 Leader election

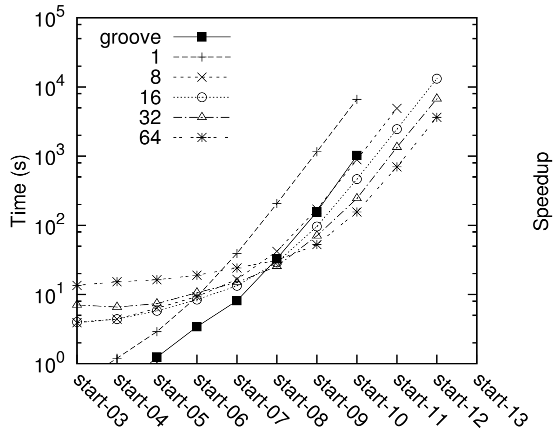
The results for the leader election case are shown in Figures 6.3 and 6.4. The results are similar to those of the car platooning case, which is expected because of the similarities between the two systems: both have a lot of symmetry. Figures 6.3c and 6.3d show the execution time respectively the speedup of LTSMIN with different numbers of cores compared to GROOVE for the leader election case. The speedup seems to be very good for `start-7p` with enc_N, but this could be caused by the different equipment that is used for running the `start-7p` model for GROOVE (see Section 6.2). However, the speedup seems to grow almost linearly, which is promising. As in the car platooning case we see clearly that the node vector encoding (Figures 6.3c and 6.3d) is better than the label vector encoding (Figures 6.3e and 6.3f) in terms of speed. With 56 and 64 workers there are no results for `start-7p`, which is probably due to the time that is spent on communication of values between



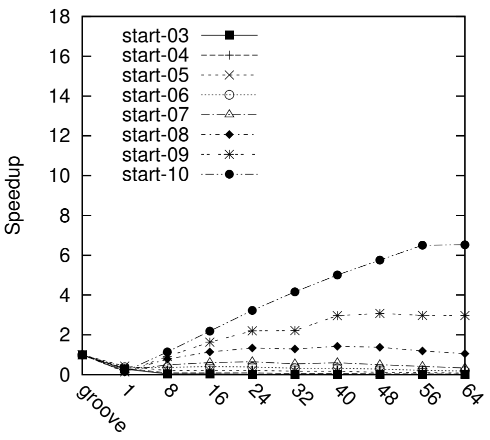
(a) unflagged-platoon. Number of states, transitions and leaf values.



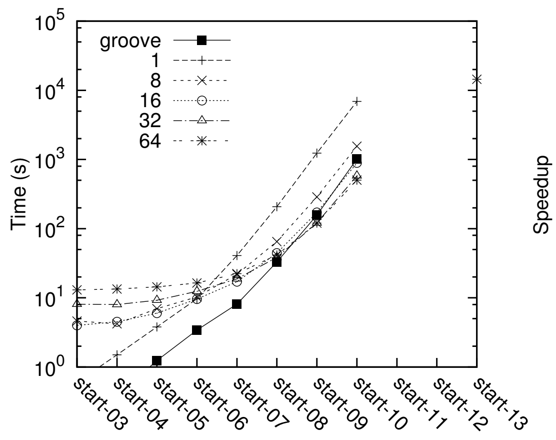
(b) unflagged-platoon. Memory usage for GROOVE and LTSMIN (per core).



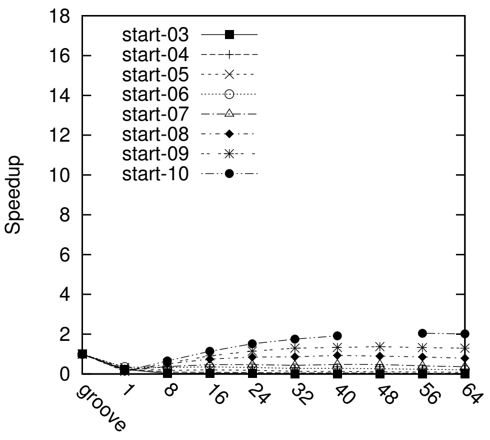
(c) `unflagged-platoon`, enc_N. Execution time.



(d) `unflagged-platoon`, enc_N. Speedup compared to GROOVE.



(e) `unflagged-platoon`, enc_L. Execution time.



(f) `unflagged-platoon`, enc_L. Speedup compared to GROOVE.

Figure 6.1: Figures for the car platooning case for different start states.

workers. For larger models the speedup is also larger. There seems to be an optimal number of workers, above which the speedup does not increase anymore. With the node vector encoding this is 40 for `start-6p`; for `start-7p` it seems that the optimum has not been reached yet. The graphs in Figure 6.4 clearly show that for smaller models more time is spent on sending and receiving, relatively to the time spent on isomorphism checking. Also, the part of the time spent on communication is larger with the label vector encoding than with the node vector encoding.

6.3.3 Append

The results for the append case are in Figures 6.5 and 6.6. Figure 6.5a shows that also in this case the numbers of states and transitions grow exponentially and that the size of the leaf database with the nodevector encoding grows slower (but still about exponentially). In Figure 6.5b we see that also in this case memory usage with the node vector encoding is better than the sequential version of GROOVE, despite the lack of symmetry in the append case and the decreased suitability for the node vector encoding (wrapping around of nodes is required). Figures 6.5c and 6.5d show the execution time respectively the speedup of LTSMIN with different numbers of cores compared to GROOVE for the append case. Here there is hardly any gain in the distributed setting, as is expected because of the lack of symmetry (less time is spent on isomorphism checking) and the need for wrapping around nodes. Even in the best case (node vector encoding and 56 nodes) the speedup is only slightly more than 2. The graphs in Figure 6.6 confirm this view. A relatively small part of the execution time is spent on isomorphism checking. Most of the time is spent on sending and receiving states and values.

6.4 Analysis

Comparison of serialisation functions The experiments show clear differences between the node vector serialisation (enc_N) and the label vector serialisation (enc_L). In all reported cases the size of the leaf database (the total number of distinct values per part of the state vector) is smaller for the node vector encoding than for the label vector encoding.

The smaller leaf database results in lower memory usage for the node vector encoding and also the time performance is (slightly) better. Additional execution time for the label vector encoding is caused by the leaf values that have to be communicated between the nodes. Compared to the memory usage of sequential GROOVE the serialisation results in a much smaller memory footprint per worker, and for up to 8 cores also in smaller total memory usage for the car platooning and leader election cases.

The label vector encoding is designed to work well when edges with a certain label are bound to certain types of source and target nodes. Edges are grouped by their labels and are encoded using the types of source and target. Nonetheless, the encoding appears to be inefficient for the reported cases. Indeed, in the rule systems that we used, most of the edge labels can occur on edges between any type of source and target nodes, which breaks the assumption.

That the node vector encoding works so well is suprising, because it seems very sensitive to the node renumbering caused by the canonical form computation. Small changes in a graph may result in a completely different node numbering, thus resulting in changes in all parts of the state vector. Because of this, probably multiple values in the leaf database encode the same set of edges (but for a different node numbering), leading to redundancy in the leaf database. However, each value can be replicated at maximum m times, because there are m vector parts. This means that the effect duplication on the size of the of the leaf database is at maximum a constant factor m. Its depends on the distribution of values over the tables $P_0, \dots, P_m$, how much values are needed to encode a set of states. With $|P_0| + \cdots + |P_m|$ values, $|P_0| \cdot \cdots \cdot |P_m|$ states can be encoded at maximum. If the values are distributed evenly over the vector parts, $\log_m s$ values can encode s states. In the car platooning and leader election rule systems, the actions on different nodes are relatively independent of each other and edges can occur between almost any combination of edges. Apparently, this causes the values to be nicely distributed over the different parts of the vector, such that the total number of states really is a combinatorial result of the different parts. Because of that the index vector compression has a larger effect than the redundancy in the database.

Comparison of sequential and distributed setup Comparing sequential GROOVE to the distributed tool with node vector serialisation, we come to the following observations:

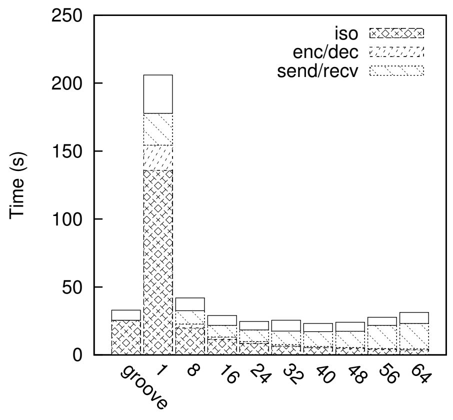
- The chosen node vector serialisation of graphs (enc_N) works well in the reported cases. The total number of values stored, the so called *leaf nodes* of the tree compression database, is orders of magnitude smaller than the total number of states. Indeed, the number of states grows exponentially with the problem size for all three modelled systems, but the number of leaf nodes grows much slower and even less than exponentially for the car platooning and leader election cases. 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 node vector encoding has to reuse slots for multiple nodes, the results are less spectacular, though the serialisation still works quite good (the number of leaf values is much smaller than the number of states).

- 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.

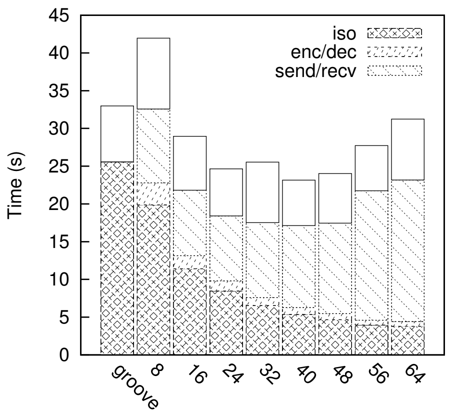
- For the car platooning and leader election cases, the time performance of the distributed LTSMIN solution scales well with the number of cores, especially for larger start graphs. The performance of a single core is quite bad compared to GROOVE, taking in the order of 6-10 times as much time, but the distributed system with 8 or more cores is faster. The bad performance of the single core distributed tool compared to sequential GROOVE is mainly due to the canonical hash code and the partial order reduction that is used in GROOVE, as described in Section 6.1.

  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 anymore. There seems to be, however, a limit to the speedup of the distributed setup. From a certain number of cores the speedup does not increase anymore. For the car platooning case this optimum is 48 for 9 cars and 56 for 10 cars; for the leader election case it is 40 for 6 network nodes. The optimal number of cores is larger for larger models.

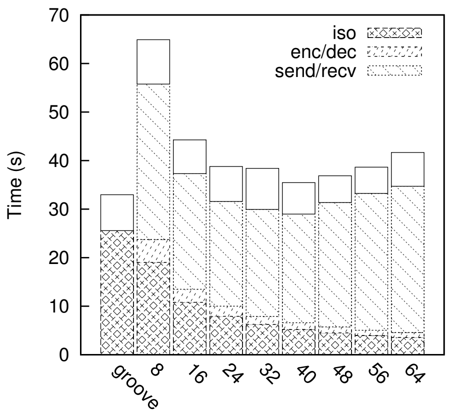
- The canonical form computation in the LTSMIN-based system lasts as much as 5 times longer than isomorphism checking in stand-alone GROOVE. As the certificate-based solution of GROOVE uses the same underlying technique as BLISS' canonical form computation (namely, repeated partition refinement), there 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.



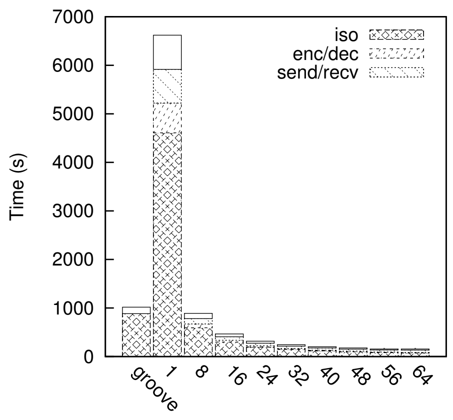
(a) `unflagged-platoon`, `start-08`, enc_N.



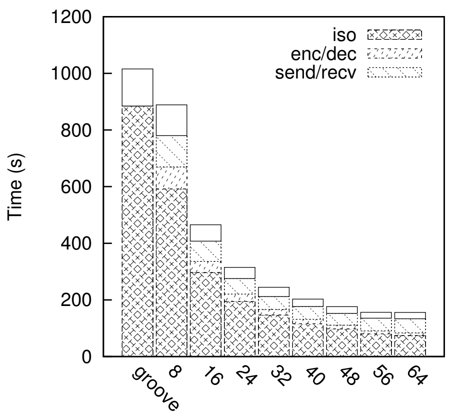
(b) `unflagged-platoon`, `start-08`, enc_N.*



(c) `unflagged-platoon`, `start-08`, enc_L.*



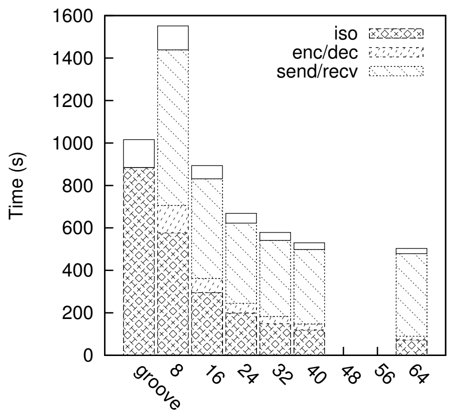
(d) `unflagged-platoon`, `start-10`, enc_N.



(e) `unflagged-platoon`, `start-10`, enc_N.*

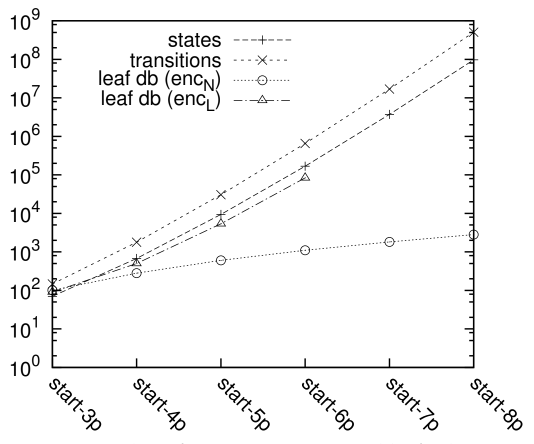
Legend	
iso	Computing canonical forms, including the conversion of GROOVE to BLISS and back
enc/dec	Encoding and decoding of GROOVE graphs into state vectors and back
send/recv	Waiting for the next assignment from LTSMIN
rest	Matching in GROOVE

* LTSMIN with one core is left out, because it is more than five times slower than the fastest after that.

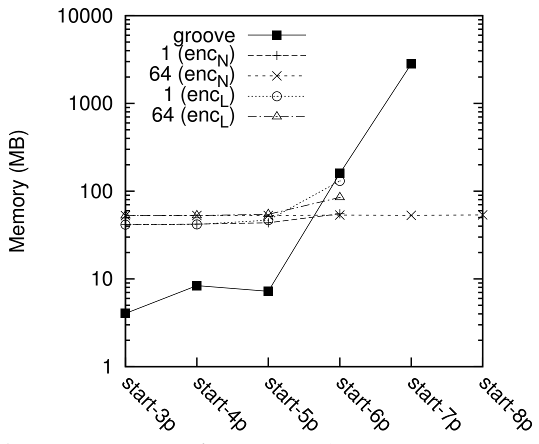


(f) `unflagged-platoon`, `start-10`, enc_L.*

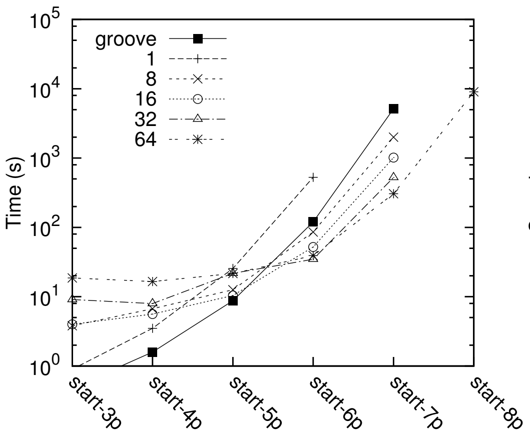
Figure 6.2: Decomposed execution times for the car platooning case for different numbers of cores.



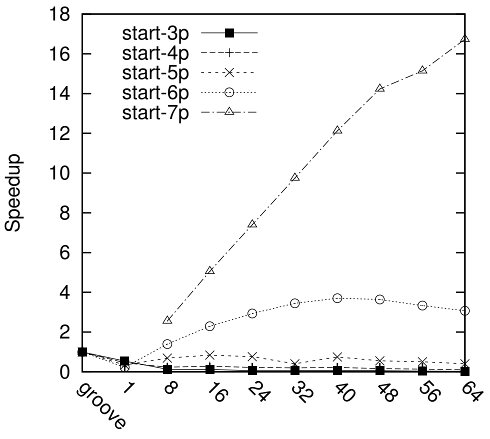
(a) `le`. Number of states, transitions and leaf values.



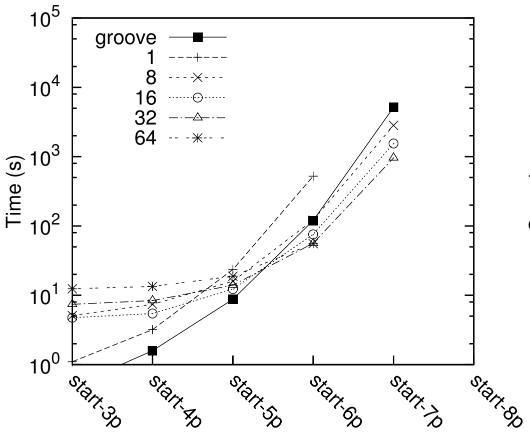
(b) `le`. Memory usage for GROOVE and LTSMIN (per core).



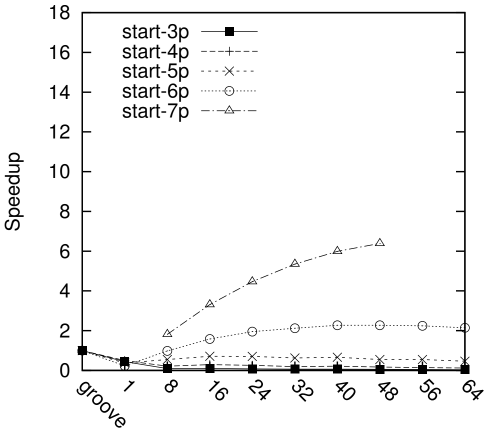
(c) `le`, enc_N. Execution time.



(d) `le`, enc_N. Speedup compared to GROOVE.

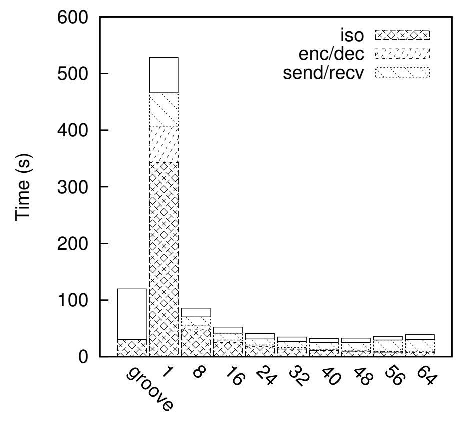


(e) `le`, enc_L. Execution time.

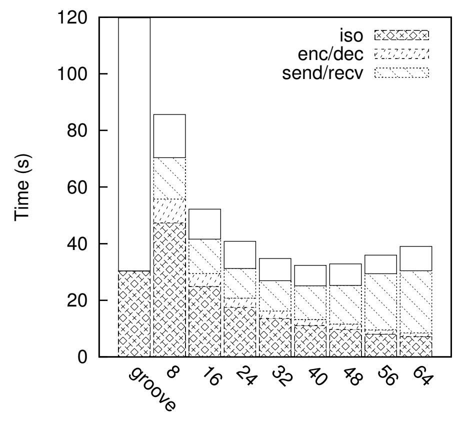


(f) `le`, enc_L. Speedup compared to GROOVE.

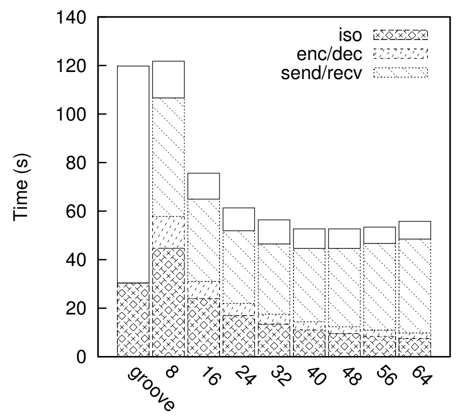
Figure 6.3: Figures for the leader electon case for different start states.



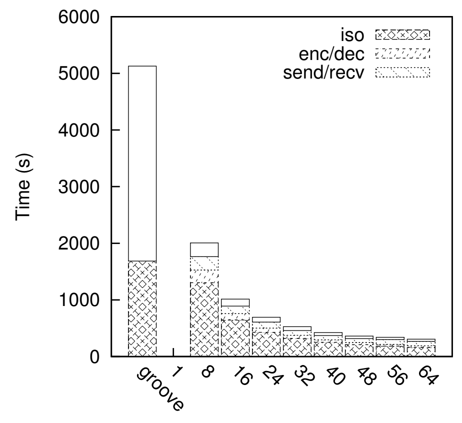
(a) `le`, `start-6p`, *enc*$_N$.



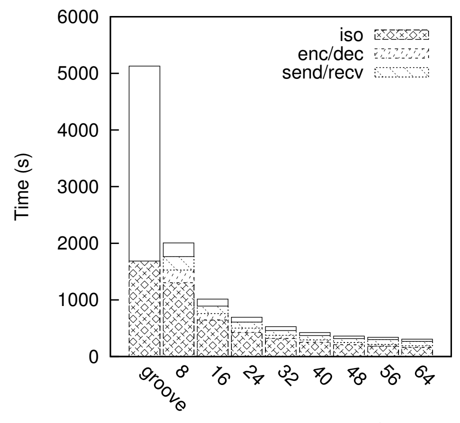
(b) `le`, `start-6p`, *enc*$_N$.*



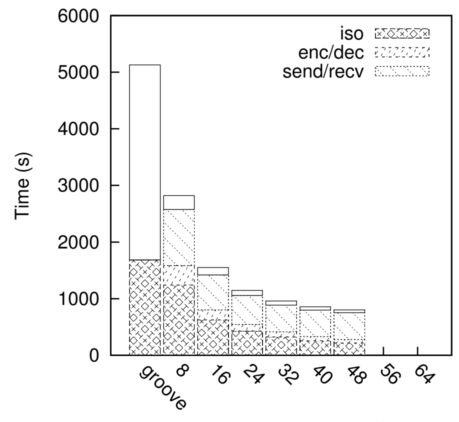
(c) `le`, `start-7p`, *enc*$_L$.*



(d) `le`, `start-7p`, *enc*$_N$.



(e) `le`, `start-7p`, *enc*$_N$.*



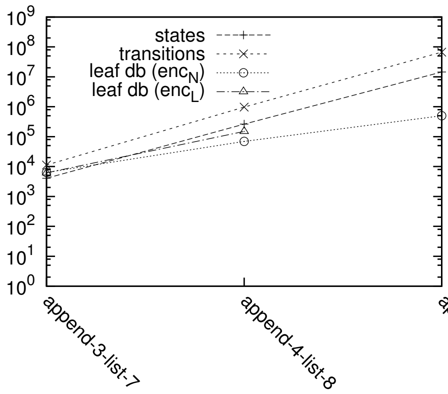
(f) `le`, `start-7p`, *enc*$_L$.*

Legend

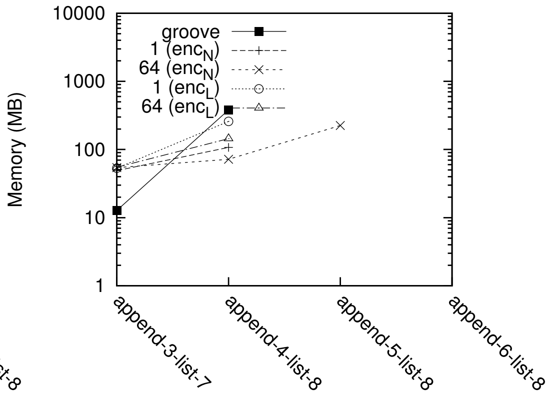
iso	Computing canonical forms, including the conversion of GROOVE to BLISS and back
enc/dec	Encoding and decoding of GROOVE graphs into state vectors and back
send/recv	Waiting for the next assignment from LTSMIN
rest	Matching in GROOVE

* LTSMIN with one core is left out, because it is more than five times slower than the fastest after that.

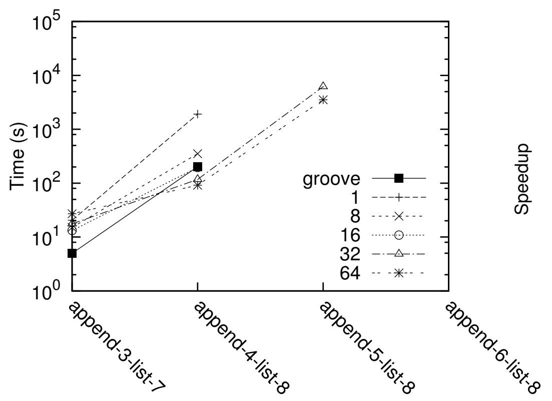
Figure 6.4: Decomposed execution times for the leader election case for different numbers of cores.



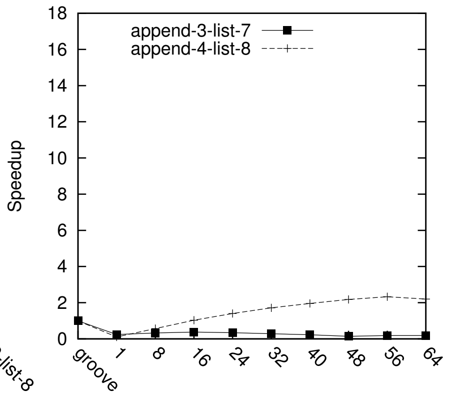
(a) append. Number of states, transitions and leaf values.



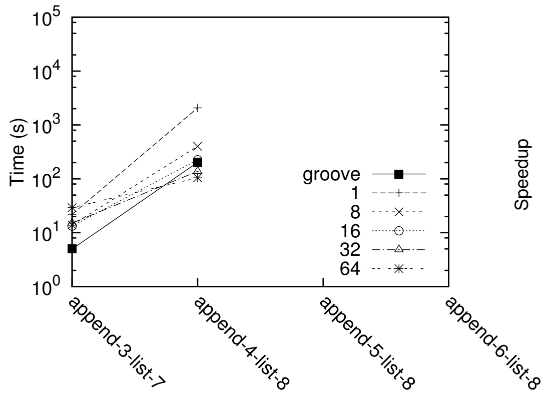
(b) append. Memory usage for Groove and LTSmin (per core).



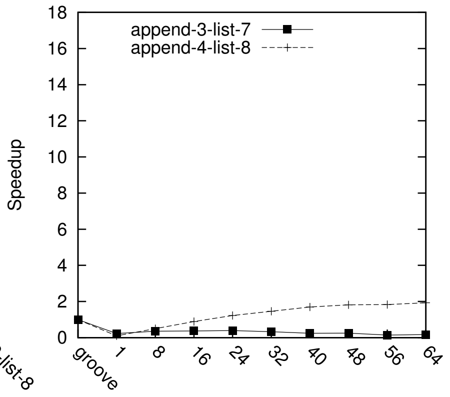
(c) append, enc_N. Execution time.



(d) append, enc_N. Speedup compared to Groove (enc_N).

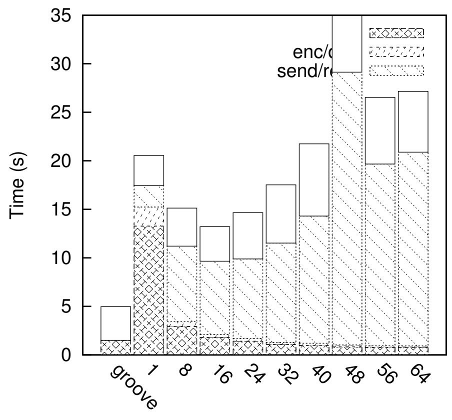


(e) append, enc_L. Execution time.

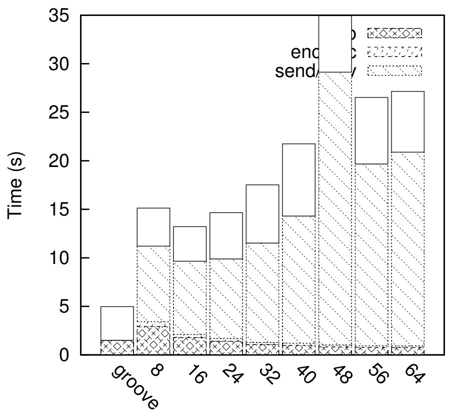


(f) append, enc_L. Speedup compared to Groove.

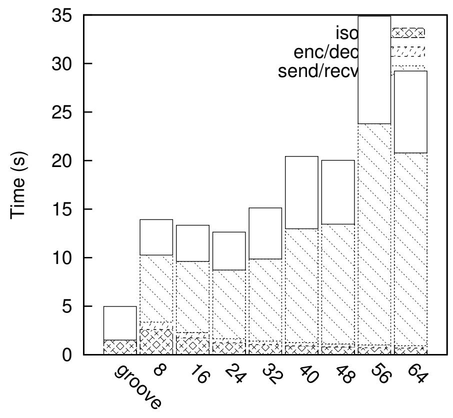
Figure 6.5: Figures for the concurrent append case for different start states.



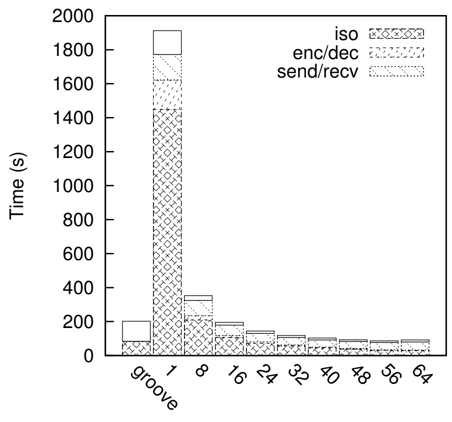
(a) append, `append-3-list-7`, enc_N.



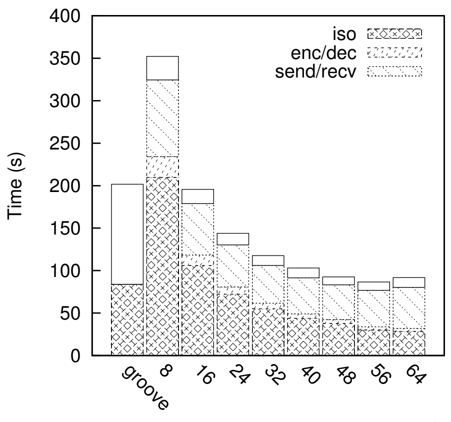
(b) append, `append-3-list-7`, enc_N.*



(c) append, `append-3-list-7`, enc_L.*



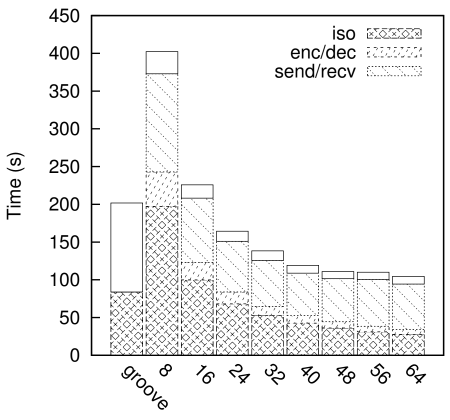
(d) append, `append-4-list-8`, enc_N.



(e) append, `append-4-list-8`, enc_N.*

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

* LTSmin with one core is left out, because it is more than five times slower than the fastest after that.



(f) append, `append-4-list-8`, enc_L.*

Figure 6.6: Decomposed execution times for the concurrent append case for different numbers of cores.

7 Conclusions and Future Work

We developed a tool for distributed state space generation for graph transformation systems. The tool is based on the existing tools (a) GROOVE, used for computing successor states for graphs, (b) LTSMIN, used for communicating states between different instances of the tool (workers) and storing the generated state space, and (c) BLISS, a tool for computing canonical forms of graphs. GROOVE in itself is already capable of generating a graph transition system for a given graph grammar and start graph. However, GROOVE does this only in a sequential manner. LTSMIN uses state vectors as representation of states, but uses compressed vectors for storage and communication between the workers. Specifically, fixed size vectors are used and for the state vector compression in LTSMIN to work well, the different number of values per part of the state vector needs to be orders of magnitude smaller than the total number of states in the generated system. The values used in the state vectors together form the leaf database. In the distributed setup, state graphs need to be serialised to vectors of data. We defined two graph serialisation functions: one that occupies a slot per node in the graph (enc_N) and one the assigns slots to edge labels (enc_L). In GROOVE edge labelled graphs are used to represent states. BLISS uses coloured graphs, which is a slightly different graph formalism. We describe a way to compute canonical forms for edge labelled graphs using BLISS by means of a conversion from edge labelled graphs to coloured graphs.

Experiments have been performed to investigate the time and memory performance of the distributed tool with the two different serialisation functions compared to the sequential version of GROOVE for three different case studies: two with a lot of symmetry in the state graphs (car platooning and leader election) and one without so much symmetry (append).

The experiments show that the node vector encoding enc_N is better than the label vector encoding enc_L in that it results in smaller leaf databases, less memory usage, and shorter execution times.

The distributed setup with node vector serialisation enc_N results in orders of magnitude less memory usage than sequential GROOVE for the very symmetric cases. This is caused by the size of the leaf database that is also orders of magnitude smaller than the total number of states with this serialisation. Also for the case without symmetry, memory usage is better for the distributed setting than for GROOVE. That the memory usage is better for LTSMIN than for GROOVE is suprising because the storage in GROOVE is optimised for graphs, storing only the differences (deltas) between state graphs instead of the complete graphs themselves. The memory usage is already better for LTSMIN with one core, which means that using graph serialisation instead of deltas can be an improvement for sequential GROOVE.

The execution time for the distributed setting with one core is much worse than that of sequential GROOVE. There are three main reasons for this: (1) in GROOVE the expensive isomorphism checking can often be avoided by use of an efficient canonical hashcode, while a canonical form has to be computed for every state graph in the distributed setting; (2) the partial order reduction for graph transition systems that is used in sequential GROOVE is not applicable in the distributed setting; and (3) the conversion to coloured graphs needed for computing canonical forms using BLISS blows up the size of the graphs, increasing the complexity of the task.

For the larger models the distributed solution scales well. For all reported case studies there is a start graph for which sequential GROOVE runs out of memory or is not capable of finishing within the time limit, while the distributed tool still can generate the state space in time and without running out of memory. For the cases with much symmetry, the speedup is particularly good. In the leader election case a speedup of 16 is achieved with 64 workers in the largest case for which also sequential GROOVE could generate the state space within the time limit. This means that the distributed setup works well for solving larger cases and that for larger cases the architecture itself does not cause too much overhead. For the very symmetric cases most of the time is spent on isomorphism checking (computing canonical forms in the distributed tool), which is one of the tasks that is paralellised. Although the method for computing canonical forms can still be improved, it is clear that isomorphism checking for a large number of state graphs can effectively be distributed by using canonical forms. For the append case, however, the speedup is disappointing: only a factor 2 with 64 cores. This is due to the larger leaf database that is needed. The communication of these leaf values between workers is a major factor in the execution time for this case.

There seems to be an optimal number of cores that can be efficiently used in the distributed setup. For the car platooning case, for instance, this optimum is 56 cores for 10 cars. The optimum is higher for larger start graphs.

7.1 Future work

There are several directions of research we would like to explore further.

The existing canonical form computating by BLISS is much slower the combination of the canonical hashcode and the isomorphism checking algorithm in GROOVE. However, the algorithms in GROOVE are based on the same principles as the canonical form algorithm in BLISS, which is iterative partition refinement. Therefore we expect improvements to be possible from reimplementation of the algorithm for edge labelled graphs.

Memory usage appears to be better using serialised graphs in LTSMIN than using deltas in GROOVE. We are interested in applying the tree compression of LTSMIN together with graph serialisation in the state storage of sequential GROOVE. In the distributed setup this serialised form is based on the canonical form of state graphs; in the sequential setting that is not necessary, but an efficiently computable canonical hashcode has to be available for the state vectors.

We have compared two serialisation functions of which the most straightforward one, the node vector encoding, appeared to perform the best. We would like to try other encodings of graphs, such as encodings that do not need the “wrapping around” of nodes as in the node vector encoding. Also a better distribution of nodes over the vector parts might result in a better performance for systems without fixed sized graphs or without much symmetry.

In the experiments we looked at two kinds of models: two very symmetric ones for which the node vector encoding worked out particularly well and the append case. We would like to use a broader set of graph transformation systems for the experiments to see if one encoding is the best overall or if we can find heuristics for which encoding might work well for which kind of graphs.

There are a lot of factors influencing time performance in the distributed setting. Not all of these have been investigated thoroughly. Especially the communication between the workers needs further research. How do network bandwidth and latency affect the communication of states and values? Also, some communication between components currently is synchronously, which may introduce unnessecary waiting. This needs further research and perhaps changing to an asynchronous implementation.

For partial order reduction in graph transition systems, a larger part of the transition system needs to be accessible than is now the case in the distributed setting. It would be interesting to set up an archictecture in which partial order reduction is possible. In such a setup, either each worker has to be responsible for multiple succesive transitions (if they are independent), or there has to be a central store containing the LTS that is efficiently readable for the workers.

Bibliography

[1] L. Babai and E.M. Luks. Canonical Labeling of Graphs. In *Proc. of the 15th Annual ACM Symposium on Theory of Computing*, pages 171–183. ACM, 1983.

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, Mass., 2008.

[3] J. Barnat, L. Brim, and Jitka Stříbrná. Distributed LTL model-checking in SPIN. In M. Dwyer, editor, *Model Checking Software*, volume 2057 of *Lecture Notes in Computer Science*, 2001.

[4] G. Bergmann, I. Ráth, and D. Varró. Parallelization of Graph Transformation Based on Incremental Pattern Matching. In A. Boronat and R. Heckel, editors, *Graph Transformation and Visual Modeling Techniques (GT-VMT)*, volume 18 of *Electronic Communications of the EASST*, 2009.

[5] S.C.C. Blom, B. Lisser, J.C. van de Pol, and M. Weber. A Database Approach to Distributed State-Space Generation. *Journal of Logic and Computation*, 2009.

[6] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance Evaluation of the VF Graph Matching Algortithm. In *Proc. of the International Conference on Image Analysis and Processing*, pages 1172–1177, 1999.

[7] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. An Improved Algorithm for Matching Large Graphs. In *Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.

[8] 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(10):1367–1372, 2004.

[9] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, Berlin, Heidelberg, 2006.

[10] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, 2001.

[11] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[12] A.H. Ghamarian, M.J. de Mol, A. Rensink, Eduardo Zambon, and M.V. Zimakova. Modelling and Analysis Using GROOVE. Technical Report TR-CTIT-10-18, CTIT, University of Twente, Enschede, 2010.

[13] G.J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proc. of the Third International SPIN Workshop*, 1997.

[14] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[15] T. Junttila. bliss: A Tool for Computing Automorphism Groups and Canonical Labelings of Graphs. Online, 2008. `http://www.tcs.hut.fi/Software/bliss/`.

[16] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proc. of the 9th Workshop on Algorithm Engineering and Experiments*, pages 135–149. SIAM, 2007.

[17] G. Kant. Using Canonical Forms for Isomorphism Reduction in Graph-based Model Checking. Technical Report TR-CTIT-10-28, CTIT, University of Twente, Enschede, 2010.

[18] F. Lerda and R. Sisto. Distributed-Memory Model Checking with SPIN. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, 1999.

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

[20] B.D. McKay. nauty 2.4. Online, 2008. `http://cs.anu.edu.au/~bdm/nauty/`.

[21] B.T. Messmer and H. Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition*, 32(12):1979–1998, 1999.

[22] B.T. Messmer and H. Bunke. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):307–323, 2000.

[23] Formal Methods and Tools. LTSmin. Online, 2008. University of Twente, `http://fmt.cs.utwente.nl/tools/ltsmin/`.

[24] T. Miyazaki. The complexity of McKay's canonical labeling algorithm. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 28:239–256, 1997.

[25] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[26] A. Rensink. Isomorphism Checking in GROOVE. In *Proc. of the Third International Workshop on Graph Based Tools (GraBaTs 2006)*, volume 1 of *Electronic Communications of the EASST*, 2007.

[27] A. Rensink. Explicit State Model Checking for Graph Grammars. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 114–132, 2008.

[28] A. Rensink. Groove 4.0.1. Online, 2010. `http://groove.sourceforge.net`.

[29] A. Rensink. Isomorphism Checking for Symmetry Reduction. Technical Report TR-CTIT-10-27, CTIT, University of Twente, Enschede, 2010.

[30] A. Rensink, Á. Schmidt, and D. Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In *Graph Transformations*, volume 3256 of *Lecture Notes in Computer Science*, 2004.

[31] C. Spermann and M. Leuschel. ProB gets Nauty: Effective Symmetry Reduction for B and Z Models. In *Proc. of the 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'08)*, pages 15–22, 2008.

[32] E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry Reduced Model Checking for B. In *Proc. of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE '07)*, pages 25–34, 2007.

[33] J.R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31–42, 1976.

[34] E.W. Weisstein. Isomorphic Graphs. *MathWorld – A Wolfram Web Resource*, 2009. `http://mathworld.wolfram.com/IsomorphicGraphs.html`.