Electronic Communications of the EASST
Volume 1 (2006)

Proceedings of the
Third International Workshop on Graph Based Tools
(GraBaTs 2006)

Isomorphism Checking in GROOVE

Arend Rensink

11 pages

Guest Editors: Albert Zündorf, Daniel Varró
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Isomorphism Checking in GROOVE

Arend Rensink

Department of Computer Science, University of Twente
P.O.Box 217, 7500 AE Enschede, The Netherlands

Abstract: In this paper we show how isomorphism checking can be used as an effective technique for symmetry reduction in graph-based state spaces, despite the inherent complexity of the isomorphism problem. In particular, we show how one can use *element-based graph certificate mappings* to help in recognising non-isomorphic graphs. These are mappings that assign to all elements (edges and nodes) of a given graph a number that is invariant under isomorphism, in the sense that any isomorphism between graphs is sure to preserve this number. The individual element certificates of a graph give rise to a certificate for the entire graph, which can be used as a hash key for the graph; hence, this yields a heuristic to decide whether a graph has an isomorphic representative in a previously computed set of graphs. We report some experiments that show the viability of this method.

Keywords: Graph Certificates, Isomorphism, GROOVE, Model Checking

1 Introduction

The core activity of any model checker is state space exploration. In case of exiplicit-state model checking, for this purpose it is imperative to be able to detect, as fast as possible, whether the target state of a newly computed transition has been encountered before during exploration. In a setting where the states are graphs, as in the GROOVE tool [10], in fact it is even more interesting to know if an *isomorphic* state has been encountered before: differences that are finer than isomorphism do not matter to any property that we might want to check, and by collapsing isomorphic states one achieves the strongest possible form of symmetry reduction. Not only does this reduce the state space by a factor equal to the degree of symmetry in the problem, but it also relieves us from the burden of choosing clever representatives for fresh nodes or edges that are newly created in a production: the isomorphism of the target graph does not depend on this choice.

The downside of this idea, obviously, is that graph isomorphism is a hard problem, believed not to be polynomial (see, e.g., [14]). The main contribution of this paper is to discuss the palette of choices made in GROOVE to alleviate this problem. The essence is to use *certificates* (often also called *invariants*) to characterise the isomorphism classes of graphs. The necessary definitions are given in Section 2; in Section 3 we discuss the particular algorithm implemented in GROOVE to compute the certificates. In Section 4 we show this has been successful at least to some degree: for a particular (highly symmetrical) example problem, thanks to the automatic isomorphism checking, GROOVE is able to generate answers for larger-sized problems than conventional model checkers. Finally, in Section 5 we discuss the results, including some directions for further improvement.

2 Definitions

We assume a universe Lab of labels, with an injective hash function $hash\colon \mathsf{Lab} \to \mathsf{Nat}$ assigning numbers to labels. We first recall the definitions of graphs and isomorphism.

Definition 1 (graphs and isomorphisms) A *graph* is a tuple $\langle V, E, src, tgt, lab \rangle$ with V a set of nodes and E a set of edges, $src, tgt\colon E \to V$ the source and target function, and $lab\colon E \to \mathsf{Lab}$ the edge labelling function. Given two graphs G, H, an *isomorphism* $f\colon G \to H$ is a pair of bijections $f_V : V_G \to V_H$ and $f_E : E_G \to E_H$ such that $src_H \circ f_E = f_V \circ src_G$, $tgt_H \circ f_E = f_V \circ tgt_G$ and $lab_H \circ f_E = lab_G$.

Let Graph denote the universe of graphs. We recall the following (see, e.g., [14]):

Observation 2 (complexity of isomorphism) *Given two graphs G, H, deciding $G \cong H$ is in NP relative to $|V_G|$, but not known either to be in P or to be NP-complete; it is thought to be neither.*

An important concept in algorithms for isomorphism checking is that of an *invariant* or *certificate*:

Definition 3 (graph certificates) A *graph certificate mapping* is a function $c\colon \mathsf{Graph} \to X$ for some set X such that $G \cong H$ implies $c(G) = c(H)$. $c(G)$ is called the *certificate* of G. c is called *element-based* if $c(G) = (c_V^G, c_E^G)$ with $c_V^G\colon V_G \to Y_V$ and $c_E^G\colon E_G \to Y_E$ for some sets Y_V, Y_E, such that $f : G \cong H$ implies $c_V^G = c_V^H \circ f_V$ and $c_E^G = c_E^H \circ f_E$.

Hence, an element-based graph certificate mapping is one for which the set X of graph certificates consists of individual certificate mappings for the nodes and edges of the graphs. In this paper we concentrate on certificates in Nat, i.e., such that $X = Y_V = Y_E = \mathsf{Nat}$. Straightforward examples are:

- $G \mapsto |E_G|$, yielding the number of edges in a graph;
- $G \mapsto (c_V, c_E)$ where $c_V : v \mapsto |tgt_G^{-1}(v)|$ and $c_E : e \mapsto hash(lab_G(e))$.

Note that, given an element-based certificate mapping c, we can easily derive a certificate mapping $\bar{c}\colon \mathsf{Graph} \to \mathsf{Nat}$, for instance by defining

$$\bar{c}\colon G \mapsto \sum_{v \in V_G} c_V^G(v) + \sum_{e \in E_G} c_E^G(e) \quad . \tag{1}$$

Bisimulation. Our algorithm for isomorphism checking is inspired by *bisimilarity* as defined by Milner [7] and Park [9], but this is due to our own background; in terms of the literature on graph isomorphism, what we are about to define corresponds to the notion of an *equitable partition* (see McKay [6]).

Definition 4 (generalised bisimilarity) Given a graph G, a *generalised bisimulation over G* is a relation $R_G \subseteq (V \times V) \cup (E \times E)$ satisfying the following two properties:

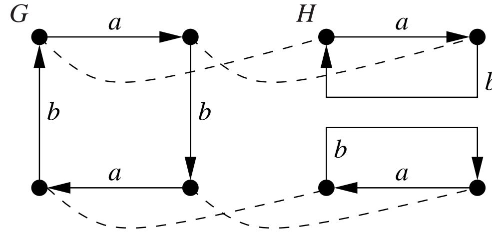


Figure 1: Two bisimilar, non-isomorphic graphs

1. $(v,w) \in R_G$ implies there is a bijection g^v from the incident edges of v to the incident edges of w, such that for all $e \in dom(g^v)$: (i) $src(e) = v$ iff $src(g^v(e)) = w$, (ii) $tgt(e) = v$ iff $tgt(g^v(e)) = w$, and (iii) $(e, g^v(e)) \in R_G$.
2. $(d,e) \in R_G$ implies (i) $lab(d) = lab(e)$ and (ii) $(src(d), src(e)), (tgt(d), tgt(e)) \in R_G$.

Generalised bisimilarity over G, denoted $\sim_G$, is the largest generalised bisimulation over G.

The notion of a largest generalised bisimulation is well-defined because the identity relation is a generalised bisimulation, as is the union of an arbitrary set of generalised bisimulations. The connection between generalised bisimilarity and isomorphism is given by the following proposition, where $G \uplus H$ denotes the disjoint union of G and H:

Proposition 1 *$f : G \cong H$ implies $v \sim_{G \uplus H} f_V(v)$ for all $v \in V_G$ and $e \sim_{G \uplus H} f_E(e)$ for all $e \in E_G$.*

On the other hand, it is *not* the case that generalised bisimilarity completely predicts isomorphism; rather, it gives rise to a strictly coarser relation. To make this statement precise, let us denote $G \sim H$ if there is a bijective mapping h from the elements of G to those of H (not necessarily a morphism) such that $v \sim_{G \uplus H} h(v)$ and $e \sim_{G \uplus H} h(e)$ for all $v \in V$, $e \in E$. Now Proposition 1 can be seen to imply that $G \sim H$ whenever $G \cong H$. However, the inverse does not hold; in other words, $\sim$ is *strictly* coarser than $\cong$. An example is given in Figure 1, which shows two non-isomorphic graphs G and H, with some non-trivially $\sim_{G \uplus H}$-related node pairs connected by dashed lines. Clearly, $G \sim H$ in the sense defined above.

As discussed in the introduction, our business here is state space exploration, where a graph is associated with each state and we want to collapse states with isomorphic associated graphs. The problem to be solved is therefore not just comparing two graphs up to isomorphism, but determining whether we have already encountered any isomorphic variant of a given graph. To be precise, given a graph G and a previously computed, finite set S of graphs, which are guaranteed to be distinct up to isomorphism, we want to find the unique $H \in S$ such that $G \cong H$ if such an H exists, or to compute $S \cup \{G\}$ otherwise. (Clearly, the problem of deciding isomorphism of two given graphs G, H appears as a special case of this, by taking $S = \{H\}$.) In this problem, not only the size of the graphs but also the size of S is a factor in the complexity.

3 The algorithm

The algorithm we use to solve the problem described above consists of the following elements:

1. For a given graph G, we compute a sequence of element certificate mappings (c^i_V, c^i_E) for $i \in \mathsf{Nat}$, mapping to Nat; each such mapping induces a similarity relation $\sim^i \supseteq \sim_G$ over the nodes and edges of G, defined by $v \sim^i w$ if $c^i_V(v) = c^i_V(w)$ and $d \sim^i e$ if $c^i_E(d) = c^i_E(e)$;
2. We stop the computation at the first i for which $|V/{\sim^i}| \leq |V/{\sim^{i-1}}|$, i.e., as soon as the number of cells in the partition induced by the certificate mapping no longer grows;
3. From the ensuing element-based graph certificate mapping c with $(c^G_V, c^G_E) = (c^i_V, c^i_E)$, we derive a certificate mapping $\bar{c}$ as in (1);
4. The set of previously explored graphs S is stored as a hash set with hash values determined by $\bar{c}$;
5. After computing $\bar{c}(G)$ for a new graph G, we determine the set of graphs $Q \subseteq S$ defined by $Q = \{H \mid \bar{c}(H) = \bar{c}(G)\}$. We try to establish $G \cong H$ for all $H \in Q$; if this fails, we add G to S.
6. To test $G \cong H$, we proceed as follows:

   Equal graphs First test $G = H$; if so, obviously $G \cong H$;

   Injective element certificates Otherwise, test if c^G_V and c^G_E are injective; if so, $G \cong H$ if and only if $((c^H_V)^{-1} \circ c^G_V, (c^H_E)^{-1} \circ c^G_E)$ is an isomorphism;

   Complex isomorphism Otherwise, define $\simeq \subseteq (V_G \times V_H) \cup (E_G \times E_H)$ such that $v \simeq w$ iff $c^G_V(v) = c^H_V(w)$ and $d \simeq e$ iff $c^G_E(d) = c^H_E(e)$; if $f : G \cong H$ then $v \simeq f_V(v)$ and $e \simeq f_E(e)$ for all $v \in V_G$, $e \in E_G$, so $\simeq$ gives a good starting point for finding f.

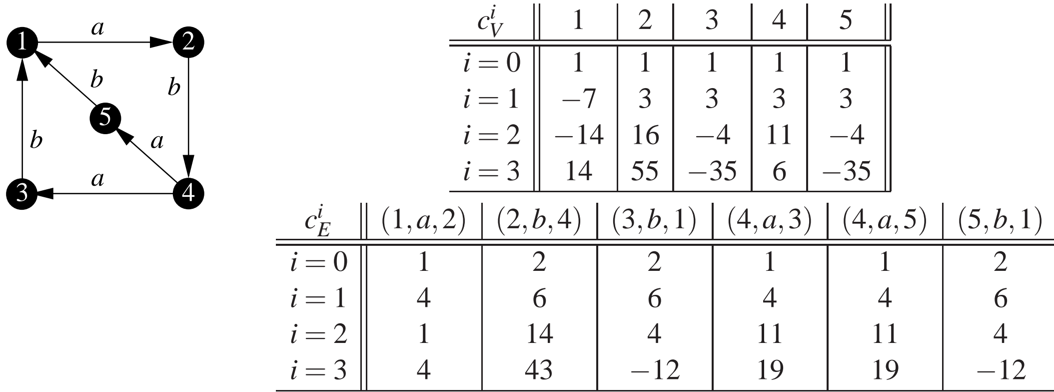
   False positives If all fails, then $\bar{c}(G) = \bar{c}(H)$ whereas $G \ncong H$; we call this a *false positive* of the certificate function $\bar{c}$.

Obviously, if $G \sim H$ but $G \ncong H$ then a false positive is unavoidable; but false positives may also arise in case $G \nsim H$, depending on how well the certificate mapping does in assigning distinct values to elements that are not bisimilar.

Although the requirements on the sequence of element certificate mappings that are listed above are sufficient to guarantee the *correctness* of the algorithm, its *performance* depends strongly on some further “quality criteria”. Desirable properties for the sequence of mappings are:

- Each next similarity relation should refine the previous; i.e., $\sim^{i+1} \subseteq \sim^i$ for all i. (This implies that the termination criterion in Step 2 is satisfied if and only if $V/{\sim^i} = V/{\sim^{i-1}}$, i.e., if and only if the i-th certification mapping induces the *same* partition as the previous one.) Moreover, once the refinement has stabilised ($\sim^{i+1} = \sim^i$ at some i) it should remain stable ($\sim^j = \sim^i$ for all $j > i$). (This ensures that we do not terminate the iteration too early.)

- The sequence of similarity relations should converge to $\sim_G$; i.e., $\sim^i = \sim_G$ for some i. (In combination with the requirement $\sim^i \supseteq \sim_G$ in Step 1, this implies that $|V/{\sim^{i+1}}| \leq |V/{\sim^i}|$ for that i, and hence the termination criterion in Step 2 is fulfilled. If, furthermore, the previous desideratum is also met, we are sure to actually terminate at that i and not before, so the mapping c is *optimal* for our purpose.)



c_V^i	1	2	3	4	5
$i=0$	1	1	1	1	1
$i=1$	−7	3	3	3	3
$i=2$	−14	16	−4	11	−4
$i=3$	14	55	−35	6	−35

c_E^i	$(1,a,2)$	$(2,b,4)$	$(3,b,1)$	$(4,a,3)$	$(4,a,5)$	$(5,b,1)$
$i=0$	1	2	2	1	1	2
$i=1$	4	6	6	4	4	6
$i=2$	1	14	4	11	11	4
$i=3$	4	43	−12	19	19	−12

Figure 2: Example graph and element certificate function, for $hash(\mathtt{a}) = 1$, $hash(\mathtt{b}) = 2$ *and* $newCert(old, lab, srcCert, tgtCert) = old + lab + srcCert + tgtCert$

- The number of iterations until termination should be small, and the computation of each next mapping in the sequence fast. (We recall from Paige and Tarjan [8] that the average-case complexity of computing bisimilarity, which is based on the same principle of partition refinement, is $O(n \log n)$ with $n = |V|$. In analogy, we conjecture that the average number of iterations ideally is $O(\log |V|)$, in which case the complexity of each iteration is $O(|V|)$.)

Our basic strategy is to start c_V^0 at some constant and c_E^0 at the hash value of the edge labels, and to compute the certificates in each next iteration as a function of the previous certificate values for the immediate local context, i.e., the end nodes for the edges and the incident edges for the nodes. In short:

$$c_E^i(e) = \begin{cases} hash(lab(e)) & if \text{ i=0} \\ newCert(c_E^{i-1}(e), lab(e), c_V^{i-1}(src(e)), c_V^{i-1}(tgt(e))) & otherwise \end{cases}$$

$$c_V^i(v) = \begin{cases} 1 & if \text{ i=0} \\ c_V^{i-1}(v) + \sum_{v=src(e)} c_E^i(e) - \sum_{v=tgt(e)} c_E^i(e) & otherwise. \end{cases}$$

where the most important parameter is the function *newCert*, which computes the next edge certificate value from the previous one, the edge label, and the previous certificate values of the source and target nodes. As an aside, it is not difficult to prove that, irregardless of the choice of *newCert*, the above definition results in a valid sequence of element certificate mappings.

As an example, in Figure 2 we show a graph with some internal symmetry and the calculation of c_V^i and c_E^i until $V/\!\sim^i$ converges, which is for $i = 3$, at $V/\!\sim^3 = V/\!\sim^2 = \{\{1\}\{2\},\{3,5\},\{4\}\}$. Note that, for the sake of understandability, here we have taken a particularly simple choice for *newCert*.

It is the task of *newCert* to pick appropriate values so as to meet the desiderata listed above. For instance, ideally *newCert* should be injective (so that the sequence converges, and distinctions

in label or end nodes are taken into account), and everywhere asymmetric in its third and fourth parameter (so that the direction of the edges is taken into account). In practice, however, we have to make do with the finite fragment of integers offered by our programming language of choice; in Java, the type `int` with 32 bits. As a consequence, we cannot even ensure injectivity. The current implementation is as follows:

```
int newCert(int old, Label lab, int srcCert, int tgtCert) {
    int srcShift = 8;
    int tgtShift = (hash(lab) & 0xf) + 1;
    return ((srcCert << srcShift) | (srcCert >>> (32-srcShift)))
        + ((tgtCert << tgtShift) | (tgtCert >>> (32-tgtShift)))
        + old;
}
```

The core algorithm for the computation of (c_V^G, c_E^G), as outlined above (Steps 1–3), is given in Table 3. We assume arrays `int[] src,tgt,lab` encoding the graph with `int vSize, eSize` the node and edge count; the result of the algorithm is given by arrays `int[] vCert,eCert`. To determine the size of the partitions we have implemented a type `IntSet` with methods `void add(int i)` and `int size()`; it works on the basis of a hash set.

Note that the complexity of each iteration is $O(|V|+|E|)$, and the number of iterations is bounded by $|V|$ (since the iteration terminates as soon as the size of the partition stops growing, and the size can obviously grow no larger than $|V|$). As mentioned above, by analogy to the complexity of bisimilarity checking we conjecture that the average number of iterations is $O(\log |V|)$.

4 Results

In this section we provide some statistics that give an impression on how well GROOVE performs with respect to isomorphism checking using the setup described above, and we discuss future work.

Table 4 shows figures from a number of case studies we have undertaken. These were measured on a 3 GHz Pentium IV machine, running the development version of GROOVE in the JVM 1.5.0 with 1,5 GB of memory. The first three case studies were also reported (for an older version of GROOVE) in [13, 11]).

Mutex is a mutual exclusion protocol taken from [3]. It is characterised by relatively small states with a lot of unpredictable symmetries. The figures reported here are for graphs of up to 6 nodes (which can either be processes or resources, in terms of the protocol).

Philosophers is a variant of the well-known dining philosophers example, here computed for a problem size 12. For a precise description see [13]. It is characterised by a 12-fold symmetry, which is quite predictable as no nodes are created or deleted.

Append models a concurrently invoked append method that puts a new element to the tail of a list. For a precise description see [13]. The case is characterised by symmetry that is mainly due to confluence of rules; the graphs are quite dynamic and grow relatively large. The figures reported here are for a list of length 8 and 4 concurrent invocations.

```
int[] tmp = new int[vSize];
int partSize = 1;
// initialise the edge certificates
for (int e = 0; e < eSize; e++)
    eCert[e] = hash(lab[e]);
// initialise the node certificates
for (int v = 0; v < vSize; v++)
    vCert[v] = 1;
do {
    // store the current number of partition cells
    int oldPartSize = partSize;
    // calculate the new edge certificates
    for (int e = 0; e < eSize; e++) {
        eCert[e] = newCert(eCert[e], lab[e], vCert[src[e]],
                                             vCert[tgt[e]]);
        // propagate to the endpoints
        tmp[src[e]] += eCert[e];
        tmp[tgt[e]] -= eCert[e];
    }
    // collection of certificate values, used to compute partition size
    IntSet certSet = new IntSet();
    for (int v = 0; v < vSize; v++) {
        // copy the temporary node certificates to the real ones
        vCert[v] = tmp[v];
        tmp[v] = 0;
        certSet.add(vCert[v]);
    }
    partSize = certSet.size();
    // continue while the number of cells in the partition still grows
} while (partSize > oldPartSize);
```

Table 3: Core of the certificate computation

Gossips models the “gossipping girls” example (see, e.g., [4]). It is characterised by a huge amount of unpredictable symmetry. The graphs are static: no nodes are created or deleted. The figures reported here are for 8 girls.

The results in the table should be interpreted as follows:

- The **average node** and **edge counts** give an indication of the size of the graphs involved.
- The **comparison count** is the number of graphs for which the essential question: “Did we encounter an isomorphic representative of this graph before?” had to be answered.
- The **distinct graph count** is the total number of distinct graphs, i.e., the number of times the *proper* answer to the above question was “no”.
- The **equal graph certificate count** is the number of times the certificate of a new graph had been encountered before, so that the *actual* answer to the above question was “yes” (including false positives).

	Mutex	Philosophers	Append	Gossips
Average node count	5	24	38	16
Average edge count	14	66	114	64
Comparison count	1212724	2873309	116643	12193600
Distinct graph count	515134	347337	31104	2309763
Equal graph certificate count	698664	2526077	82905	9889024
Equal graphs	252402	2076356	36503	3314920
Injective element certificates	436454	449616	42734	7936
Complex isomorphism	8734	0	3668	6560981
False positives	1074	105	0	5187
Isomorphism checking (s)	76	269	21	12802
Isomorphism checking (% of total)	58%	53%	28%	72%
Computing certificates (% of iso time)	86%	65%	62%	15%
Graph equality (% of iso time)	5%	27%	12%	1%
Element certificate (% of iso time)	7%	8%	16%	0%
Complex isomorphism (% of iso time)	2%	0%	10%	84%

Table 4: Isomorphism checking results

- The next lines split up the further analysis of the equal graph certificates; see Section 3. The sum of these four cases equals the total above.
- The number of false positives is listed separately; the following equation should hold

$$\textbf{comparisons} = \textbf{distinct graphs} + \textbf{equal certs} - \textbf{false positives} \; .$$

- The **isomorphism checking** rows indicate the time taken to compute the above.

The results give rise to the following observations:

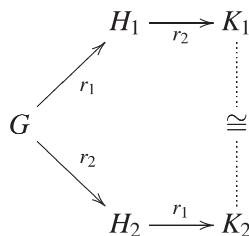
1. Isomorphism checking takes between 25% and 75% percent of the total time for state space exploration — a major fraction.
2. The degree of symmetry in all but the “append” example is sufficient to warrant the time taken for isomorphism checking. For instance, the “gossips” example can be analysed in a traditional model checker, without symmetry reduction, up to size 6 only — which should be contrasted to the size 8 reported here.[1]
3. In most cases, the majority of this time is taken up by computing the graph element certificates. The exception is the “gossips” example, which has a huge amount of non-trivial symmetry.
4. The graph certificates are very accurate in predicting isomorphism, yielding false positives in only $0,2\%$ of cases for the worst case (the “mutex” example).

[1] In [12] we report a further improvement by an order of magnitude, achieved by using quantified transformation rules.

5. Between 30% and 80% of the isomorphic graphs are actually equal.
6. In the great majority of the remaining cases, the graph element certificates are injective (or, in other words, in terms of Definition 4 there are no non-trivial bisimilarities in the graphs) so that they give a fast method for establishing isomorphism. The exception is, once more, the "gossips" example.
7. In the "gossips" example, most of the time (amounting to 51% of the total time for state space exploration) is spent in establishing the actual isomorphisms. There is certainly room for improvement here.
8. In the "append" example, which is the most realistic software model with the least symmetry, the most dynamism and the largest average graph size, the isomorphism check takes 0,18 ms per graph.

As an aside, it may be worth mentioning that even in the smaller case studies, over 50% of the total state space exploration time is taken up by garbage collection. In the "gossips" case this is much worse; we have no precise figures here, but we conjecture that in the order of 90% of the total time is taken up by garbage collection.

Parallel independence. Whatever we do to improve the performance of isomorphism checking, as long as we are dealing with arbitrary graphs (but see below) it will always remain a major bottleneck. The best improvement would be to avoid having to check for isomorphism in the first place. It turns out that this is indeed often possible, namely by taking advantage of parallel independence of productions. Namely, if the applications of r_1 and r_2 in the following diagram are independent, then we can predict beforehand, without having to check anything, that $K_1 \cong K_2$.



We have recently improved GROOVE based on this insight. It turns out that, with respect to Table 4, 50–100% of the *equal* graphs are due to such "confluent diamonds". Since we can then omit even the computation of the certificate, this results in a notable speedup in isomorphism checking, of 25% in the "append" and "gossips" examples to 70% in the "philosophers" example. Our implementation is inspired by [2].

5 Conclusion

Summarising, the contribution of this paper is the following:

- We have set out the variant of the graph isomorphism problem we need to solve for the purpose of symmetry reduction in GROOVE, and described the basic strategy for solving this using graph certificates;

- We have presented an algorithm to compute the certificate values, as implemented in GROOVE;
- We have reported and discussed some experimental results.

We have not yet carried out a sufficiently extensive comparative investigation to know how well our implementation does with respect to either other model checkers that use symmetry reduction (e.g., symmetric SPIN [1] or Murϕ [5]), or with respect to other tools for isomorphism checking (e.g., Nauty [6]); this is clearly a necessary part of our future work.

This paper allows states that can be arbitrary graphs. This is in fact one of the major differences with traditional model checkers such as the aforementioned SPIN and Murϕ, which *a priori* assume a rather rigid structure consisting of processes and data. Though we want to stick to the graph view, we believe that for software models it may be sufficient to use *deterministic* graphs only — that is, graphs for which the out-degree of any node for any given label does not exceed 1. Since isomorphism checking is known to be polynomial and actually sometimes linear for subclasses of graphs satisfying this type of restrictions, the benefit may be enormous. This, too, is future work.

Acknowledgements: The research reported herein was carried out as part of the GROOVE project, funded by NWO (Grant 612.000.314)

Bibliography

[1] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric SPIN. *STTT*, 4(1):92–106, 2002.

[2] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. An event structure semantics for safe graph grammars. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, volume A–56 of *IFIP Transactions*, pages 423–444. IFIP, North-Holland Publishing Company, 1994.

[3] R. Heckel. Compositional verification of reactive systems specified by graph transformation. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 1382 of *Lecture Notes in Computer Science*, pages 138–153. Springer-Verlag, 1998.

[4] C. A. J. Hurkens. Spreading gossip efficiently. *Nieuw Archief voor Wiskunde*, 1(2):208–210, 2000.

[5] R. Iosif. Symmetry reductions for model checking of concurrent dynamic software. *STTT*, 6(4):302–319, 2004.

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

[7] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

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

[9] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings 5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[10] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

[11] A. Rensink. Time and space issues in the generation of graph transition systems. In *International Workshop on Graph-Based Tools (GraBaTs)*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 127–139. Elsevier Science Publishers, 2005.

[12] A. Rensink. Nested quantification in graph transformation rules. In A. Corradinit et al., editor, *International Conference on Graph Transformation (ICGT)*, volume 4178 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 2006.

[13] A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *International Conference on Graph Transformations (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 226–241. Springer-Verlag, 2004.

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