Detecting strongly connected components in large distributed state spaces

S.M. Orzan, J.C. van de Pol

REPORT SEN-E0501 JANUARY 2005

ISSN 1386-369X

Detecting strongly connected components in large distributed state spaces

ABSTRACT
Detecting cycles in a state space is a key task in verification algortihms like LTL/CTL model checking and, less well known, reduction modulo branching bisimulation. This paper focuses on the problem of finding cycles (strongly connected components) in very large distributed state spaces. We present a collection of state space transformations meant as building blocks for custom algorithms. We also describe two example algorithms and show that they perform well on practical case studies.

2000 Mathematics Subject Classification: 68N30; 68W15
Keywords and Phrases: verification; strongly connected components; distributed tools

Detecting Strongly Connected Components in Large Distributed State Spaces [*]

Simona Orzan[1,2] Jaco van de Pol[2,1]

S.M.Orzan@tue.nl Jaco.van.de.Pol@cwi.nl

[1]*Dept. of Math. and Comput. Sci., Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands*

[2]*CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*

Abstract

Detecting cycles in a state space is a key task in verification algorithms like LTL/CTL model checking and, less well known, reduction modulo branching bisimulation. This paper focuses on the problem of finding cycles (strongly connected components) in very large distributed state spaces. We present a collection of state space transformations meant as building blocks for custom algorithms. We also describe two example algorithms and show that they perform well on practical case studies.

1 Introduction

A strongly connected component (SCC) of a directed graph is a maximal subgraph in which every vertex is reachable from every other vertex. The problem of decomposing a graph into SCCs is a well known and studied one and has an elegant sequential solution given by Tarjan [24]. The SCC detection problem has applications in many different areas, from data mining to scientific computing to computer-aided design and model checking. Our motivation to study it comes from verification by enumerative model checking and our graphs of interest are state spaces. The SCC detection occurs in at least two stages of the verification process. Firstly, the algorithms for branching bisimulation reduction usually employ a preprocessing step in which the cycles of invisible steps (τs) are eliminated, i.e.

[*]Partially supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW, grant CES.5009.

the SCCs of the τ-subgraph are detected and collapsed. Secondly, SCC detection is useful for LTL model checking: finding counterexamples means finding cycles reachable from the root state that contain some special accepting states.

Efforts towards building algorithms and tools for shared and distributed memory are being made in all segments of the verification process: state space generation [5, 13], equivalence reduction and equivalence checking [21, 6, 7, 16], LTL model checking [4, 8].

In this paper we propose a distributed message-passing solution for the detection of SCCs in state spaces. We describe a collection of algorithmic building blocks that can be combined in various ways heuristically. The algorithms proposed exploit the fact that state spaces, due to being generated as interleavings of parallel processes, are not just random graphs, but usually have a specific structure: there is a special initial state (root); their depth (longest distance between any state and root) and diameter (longest shortest path between any two connected states) are much smaller than the total number of states or transitions. SCCs are usually not very long cycles, but small dense knots, and many larger SCCs are built up of smaller ones. An analysis of typical state space characteristics has recently been made in [20]. These characteristics do not imply any technical restriction on the input of our algorithms. They are correct and will work for any graph.

We state the SCC problem and present the solution in the context of unlabeled graphs The algorithms can immediately be applied to deal with labeled state spaces in which only SCCs with a certain label must be found, for instance the τ-cycles in the case of state spaces.

The sequential very efficient (linear in the input size) algorithm of Tarjan [24] essentially uses depth-first-search and is therefore not likely to have an efficient parallel implementation [22]. If we try to simulate this algorithm in a distributed message-passing environment, the single messages would be very small, and the nodes would most of the time wait for each other. We design our algorithms with distribution in mind, that is such that individual messages can be buffered and sent simultaneously, without introducing deadlocks or very long idle times. This is one of the challenges for verification algorithms on distributed message-passing environments.

Gaining time efficiency is not our [only/primary] interest for using a cluster. In our experience, many verification problems generate state spaces that don't fit in one computer, so we are forced to look for distributed solutions. Fortunately, our experiments show that having multiple processors not only compensates for the extra price of message latencies, but also decreases the total time needed for cycle elimination.

To solve the problem in a parallel/distributed setting, other methods have been explored. For instance, a parallel algorithm for finding SCCs that uses matrix multiplication and needs $\mathcal{O}\ (N^2)$ processors (where N is the number of states) is proposed in [14]. But this solution is not usable on a cluster, where the number of processors available is much smaller than the input size.

A more interesting approach for our application domain is taken in [11] and [17], where a divide-and-conquer algorithm is described, analyzed and implemented. However, that algorithm is aimed at another type of graph: typically much smaller than our state spaces and with large outgoing degrees. The essential observation is that the SCC of any state x is exactly the intersection of its successors and predecessors sets. Our coloring transformation (Section 4.4) also uses this idea, but instead of picking a pivot state and splitting the graph in three independent (no crossing SCCs) parts, we use a set of prioritized colors and split the graph in many parts at once. This rather brute force approach exhibits more parallelism and it works quite well in practice. The trimming step used in [17] is similar to our detection of atomic components.

In the verification world, the problem of detecting SCCs in a distributed graph has so far received attention only in the context of (on-the-fly) LTL/CTL model checking [3, 9]. Like in our approach, in [3] the DFS traversal is abandoned in favor of BFS. But there the search for cycles is done on-the-fly, thus only some of SCCs are detected, and moreover the idea advanced is that BFS alone is not effective in searching for cycles (which is true in the on-the-fly context) and therefore is combined with DFS. The algorithm proposed in [9] is inspired by a symbolic algorithm and it also contains a phase where atomic SCCs are eliminated (namely, those SCCs containing just one state). The problem is again not finding the SCCs, but finding just one reachable accepting cycle. Moreover, the systems on which experiments were performed are rather small (mostly under 1 million states).

We encountered the SCC problem when building the distributed tool for branching bisimulation reduction presented in [7].

Outline. Section 2 introduces the main definitions, motivation and the SCC detection problem from the verification point of view. Our distributed state space transformation routines are described in quite some detail in Section 4. Then, in Section 5, some experiments with two prototype implementations are presented. Conclusions are summarized in Section 6.

2 Preliminaries

We will work with an unlabeled state space $\mathcal{S} \equiv (S, T)$, where S is a set of states and T is a binary relation on S representing the possible transitions between states. We assume that the elements of S are totally ordered. This is a reasonable assumption, since most state space generation tools identify the states by assigning them natural numbers. T^* denotes the reflexive transitive closure of T and T^{-1} the binary relation inverse to T. The *incoming degree* of a state is the number of transitions that end in that state. Conversely, the *outgoing degree* of a state is the number of transitions originating in that state.

The SCC detection problem is to find a representative function $\mathsf{scc} : S \to S$ such that $\forall x, y : \mathsf{scc}(x) = \mathsf{scc}(y)$ iff $(x, y) \in T^* \cap {T^*}^{-1}$.

SCC detection as verification problem. Our intended use of the SCC detection algorithm is as preprocessing step for two verification algorithms: branching bisimulation reduction/equivalence and LTL model checking.

The preprocessing phase in the *branching bisimulation* algorithm consists of merging the states that can reach each other on invisible paths, since they have the same (branching bisimilar) behavior. In other words, it consists in collapsing the SCCs in the state space obtained from the original one by ignoring the visible transitions. Although observations on sequential implementations show that the preprocessing phase is a big advantage, the distributed algorithm in [7] avoided using it because a distributed cycle elimination algorithm that can handle very large instances seemed to be a difficult and challenging problem in itself. This is our main motivation to study the SCC problem.

For the *LTL model checking* algorithm the interesting information is whether an accepting state belongs to a cycle, no matter what labels that cycle might contain. In this case a useful preprocessing step is to detect the SCCs of the state space obtained from the original one by ignoring the labels and to mark all the states situated on a cycle. This procedure consists of computing scc, then performing an extra test for self-loops.

Sequential SCC detection. The classical approach to the detection of strongly connected components is the Tarjan algorithm [24], which is based on depth-first traversal and solves the problem in linear time. We take this algorithm, well explained and proved correct in [2], as a primitive named `SCCTarjan` ($\mathcal{S}$).

3 Distribution

In this section, we prepare the descriptions of the transformations from Section 4. We explain the parallel computation model, the distribution of the state space and the data structures and communication primitives underlying the distributed implementations.

Framework. Our target architecture is a cluster whose nodes are connected by a high bandwidth network (Distributed Memory Machine). The number of nodes available is much smaller than the problem size ($W << \mathtt{size}(S)$). We assume that both the nodes and the network are reliable (no nodes failure, no message loss) and that the order of messages between nodes is preserved.

Distribution of the state space. The state space $\mathcal{S}$ is distributed to the W machines as follows:

$$S = S_0 \cup \cdots \cup S_{W-1} \text{ and } \forall i \neq j : S_i \cap S_j = \emptyset.$$

$$T = \bigcup_{0 \leq i,j < W} T_{ij}, \text{ where } T_{ij} = \{(x,y) \in T \mid x \in S_i \text{ and } y \in S_j\}$$

The node (or: processor, machine, worker) i *owns* the states S_i and the transitions $(\forall j)T_{ij}$. The state spaces are produced in this format by the distributed generation tool [5] from the μCRL toolset. We also assume a globally known hash function $\mathsf{worker} : S \rightarrow \{0 \cdots W-1\}$ that indicates to which worker every state belongs. In our implementations the states are identified by pairs (worker, offset) and the worker function is just a projection. We call transitions that cross worker boundaries (i.e. in T_{ij} with $i \neq j$) *cross transitions*. The performance of most algorithms on distributed state spaces is influenced by the number of cross transitions. We work with a random balanced distribution, that ensures about the same number of states to every worker but does not try to minimize the number of cross transitions. Of course, this could be improved; for instance, in [18] a method is proposed to optimize the number of cross transitions, based on abstract interpretation.

Basic data structures: sets and lists. The basic data structures used are *sets* and *lists*. On sets, the usual set union, intersection and difference ($\cup$, $\cap$, $-$) are defined. Lists are sequences $X = \langle x_1.x_2.\cdots.x_n\rangle$. We write $\langle\ \rangle$ for the empty list and $X[i]$ for the element at position i in the list X . Pairs of lists of equal size are used to implement relations in a form suitable to distribution. For example, the relation $\{(1,1),(2,1),(2,5)\}$ is represented as $(\langle 1.2.2\rangle, \langle 1.1.5\rangle)$. Two workers can now store the two parts $\langle 1.2.2\rangle$ and $\langle 1.1.5\rangle$ and as long as the order is not altered, the relation is implicitly kept by the index. Take the set of transitions between two workers, T_{ij}. The information about the source states are stored by i, and about the destination states by j. When some information about the destination states, for instance, needs to reach the corresponding source states, the information about *all* destination states is sent, in the storage order. This ensures that at the recipient worker the information about each destination reaches the corresponding source. Note that, for this approach to be advantageous, we need algorithms that actually do change a lot on their half of the transition list before requesting a transfer. The penalty of sending large buffers does not pay off if only a small number of updates is made. Our algorithms are designed to fit to this communication scheme.

To simplify the presentation of the algorithms later on, the following notations, operations and predicates for lists are considered:

$X.Y$	list concatenation
(X, Y)	pair of lists with the same length
$X - x$	remove all occurrences of x in X
$X + x$, $(X, Y) + (x, y)$	$X.\langle x \rangle$, respectively $(X + x, Y + y)$
$x \in X$	there is at least an occurrence of x in X
$(x, y) \in (X, Y)$	there is a position i such that $X[i] = x$ and $Y[i] = y$
for $x \in X$, **for** $(x, y) \in (X, Y)$	for every occurrence of x in X, or (x, y) in (X, Y).

The function scc extends naturally to sets and lists as follows:

$$\mathsf{scc}(S) \overset{\text{def}}{=} \{\mathsf{scc}(x) \mid x \in S\} \text{ and } \mathsf{scc}(\langle\ \rangle) \overset{\text{def}}{=} \langle\ \rangle \text{ and } \mathsf{scc}(\langle x \rangle.X) \overset{\text{def}}{=} \langle \mathsf{scc}(x) \rangle.\mathsf{scc}(X)$$

Distributed data structures. The worker i maintains the following data:

- S_i = the set of owned states.
- $\mathsf{scc}(S_i)$ = the current scc mapping of the owned states. scc is initialized as identity.
- $T_{ij} = (\texttt{Source}_{ij}, \texttt{Dest}_{ij})$, for every worker j, including itself. The set of transitions with the state source owned by worker i and destination owned by worker j is implemented as a pair of lists, one containing the source states ($\texttt{Source}_{ij}$) and the other the destinations ($\texttt{Dest}_{ij}$). The order is the same for both. So, $(x, y) \in T_{ij}$ if and only if there is an index p s.t. x is the pth element in $\texttt{Source}_{ij}$ and y the pth element in $\texttt{Dest}_{ij}$.
- $\texttt{finalT}_{ij}$ = the transitions that are definitely in the final set of transitions, but possibly with another numbering. More precisely, if $(x, y) \in \texttt{finalT}_{ij}$ then $(\mathsf{scc}(x), \mathsf{scc}(y))$, with scc the final mapping, will be a transition in the SCC-reduced state space.

Throughout the transformations, the current state space (S, T), the final set of transitions (`finalT`) and the current scc values are treated as global variables. Every transformation expects them to have a special form, expressed by a *precondition*, and modifies them such that a *postcondition* is ensured. We make the convention that all sets or lists occurring in the pseudocode descriptions that are not listed as input, are considered initialized as $\emptyset$ and $\langle\ \rangle$, respectively.

Communication primitives. Processes communicate by executing nonblocking send and receive operations, as well as two more complex communication patterns:

- **SEND** m **TO** i - message m to worker i.
- **RECEIVE** m **FROM** i - message m from worker i.

- **DBSUM**$(x_i\ ,\ xall)$ - for an arbitrary set of local values $x_0 \cdots x_{W-1}$, collectively compute their sum into a global value $xall$ and store a copy of the result on each worker. This is useful when all the workers are executing a loop and the exit condition depends on a global value.

- **REQ-REP** j **:** $B\ :=\ f(B)$ - the worker executing this pattern (i) sends a buffer (request) B to the worker j (not necessarily $\neq i$) and expects j to do the same, i.e. send to i a buffer B'. Upon receiving B', i creates a list C' of the same length as B', where if $B'[t] = x$ then $C'[t] = f(x)$. Then it sends C' to j and waits for a similar reply from j, i.e. a list C. In the end, i replaces B with the newly received list C and j replaces B' with C'. Thus, the effect of the request-reply action is $B\ :=\ f(B)$ and $B'\ :=\ f(B')$.

To implement these primitives, we relied on the LAM/MPI (Message Passing Interface) library [23]. Occasionally, in the actual implementations we also used MPI primitives that are more powerful than simple sends and receives. An example is MPI_AlltoAll, that transfers data, in parallel, from every worker to every worker in a careful order so as to avoid deadlocks. To keep the presentation simple, we abstract away from these details, and base the exposition of the distributed procedures only on the primitives above.

4 Distributed state space transformations

In Sections 4.1- 4.5, we introduce the various building blocks for SCC detection. An algorithm consists of a particular combination of these basic building blocks. Section 4.6 gives two (extreme) examples of such algorithms.

4.1 Identification of atomic components

We start with a simple building block, which turns out to be quite effective in practice. In fact, this procedure is sufficient if the state space contains no cycles.

Usually, a state space will contain a lot of states that do not connect via cycles with any other state. They are SCCs on their own (*atomic SCCs*). We describe a simple procedure that discovers some of them: mark all the states with incoming degree 0 as atomic components and remove them from the current state space, together with their outgoing transitions; repeat until all states have at least one incoming transition. Note that after a number of iterations there will be more than one state with incoming degree 0.

elim-atomic-fwd
Postcondition: $\forall x \in S \exists y \in S$ s.t. $(y, x) \in T$.
(i.e. all atomic components $\{x\}$ reachable from a start node (node with incoming degree 0) have been identified and removed)

```
 1  /* compute all the incoming degrees */
 2  for $x \in S_i$ do $indegree[x] := 0$ enddo
 3  for all workers $j$ do
 4      SEND $\texttt{Dest}_{ij}$ TO $j$ || RECEIVE $\texttt{Dest}_{ji}$ FROM $j$
 5      for $x \in \texttt{Dest}_{ji}$ do $indegree[x] := indegree[x] + 1$ enddo
 6  enddo
 7  /* loop: eliminate all the states without predecessors */
 8  /* and their outgoing transitions */
 9  while there still are states with $indegree$ 0 do
10      for all workers $j$ : $B_{ij} := \emptyset$
11      for $x \in S_i$ : $indegree[x] = 0$ do
12          $S_i := S_i - \{x\}$
13          for $(x, y) \in T_{ij}$ do
14              $T_{ij} := T_{ij} - (x, y)$
15              $\texttt{finalT}_{ij} := \texttt{finalT}_{ij} + (x, y)$
16              $B_{ij} := B_{ij} + y$
17          enddo
18      enddo
19      for all workers $j$ do
20          SEND $B_{ij}$ TO $j$ || RECEIVE $B_{ji}$ FROM $j$
21          for $x \in B_{ji}$ do $indegree[x] := indegree[x] - 1$ enddo
22      enddo
23  enddo
```

Figure 1: Forward BFS pass in order to identify atomic components and final transitions (worker i)

Correctness argument The states with incoming degree 0 are for sure atomic components, otherwise they would be reachable from other states. Their scc will not change anymore and it will also not influence the scc of other states, therefore they can be further ignored. The transitions originating in atomic SCCs obviously lead to states in other SCCs, thus they do not influence the scc function and can also be ignored (removed).

Distributed implementation Fig.1 shows how to find the atomic components distributedly. Workers begin by computing together (steps 2-6) the incoming degrees of all states, *indegree*. As already said, transitions T_{ij} are stored by worker i as a pair of lists $(\texttt{Source}_{ij}, \texttt{Dest}_{ij})$. The incoming degree of an arbitrary state $x \in S_j$ is the number of transitions that have x as destination state and it is easily computed by counting the occurrences of x in all lists $\texttt{Dest}_{ij}$. To this end, every list $\texttt{Dest}_{ij}$ is sent to worker j (step 4), where the number of occurrences is updated (step 5). In the second part (steps 9- 23), all states without incoming transitions are marked as atomic SCCs. Further, any transition with an atomic SCC as source will also be removed (steps 14, 15). Then the destination state of such a transition has to have its incoming degree updated: steps 19-22.

Complexity In steps 4-5, there will be one message for every pair of workers, and the destination state of every transition will be transfered once. Therefore, the message complexity (i.e., total number of messages sent) of computing the incoming degrees is $\mathcal{O}\ (W^2)$ and the bit complexity (i.e., total size of all messages sent) $\mathcal{O}\ (\texttt{size}(T))$. The time complexity, under the balanced distribution assumption, is $\mathcal{O}\ ((\texttt{size}(T) + \texttt{size}(S))/W)$. The total size of the buffers being exchanged in the **while** loop is at most $\texttt{size}(T)$. As for the total number of messages: in the worst case, every buffer gets always only one transition, which leads to a message complexity of $\mathcal{O}\ (\texttt{size}(T))$.

In order to detect as many such atomic components as possible, this procedure should be executed with regard to both forward and backward transitions. We have discussed only the forward pass. The backward pass can be implemented by reversing the graph (see steps 2 -4 in the heads-off routine, Fig.3) and calling the forward pass (with the subtle difference that the transitions marked as final should be reversed again).

Note that this procedure is sound, but not complete. The states placed "in between" two cycles will never get the degree 0 as long as the cycles are still in place.

4.2 Partial SCC detection

The Tarjan DFS algorithm can be exploited in a distributed environment as well, in several ways. One possibility is to let it perform on the local subgraph (S_i, T_{ii}) of each processor, in order to find and collapse the local components. For each component, one of the states, say x, is chosen as representative and all the others (y) are identified with it by means of the scc function: $\mathsf{scc}(y) := x$. Then all transitions have to be renamed from (x, y) to $(\mathsf{scc}(x), \mathsf{scc}(y))$. At the other extreme, `SCCTarjan` can be applied on the whole state space by a distinguished worker that first collects the state space parts from all the other workers, then computes scc and sends it back to the workers. This is of course only possible when the state space is sufficiently small – possibly after a series of other transformations.

A good idea for when the graph is not small enough and the elimination of local components does not shrink it substantially, is an intermediate approach: use

```
collapse-partial (SOME)
Postcondition:
        $(\forall x, y \in S_{\mathtt{SOME}})\ (\ (x,y) \in {T_{\mathtt{SOME}}}^{*} \cap {T_{\mathtt{SOME}}}^{*-1}$ iff $\mathsf{scc}(x) = \mathsf{scc}(y)\ )$

 1  randomly pick a manager $\mathtt{M} \in \mathtt{ALL}$
 2  /* send the local graph to the manager */
 3  if $i \in \mathtt{SOME}$ then SEND $S_i, \bigcup_{j \in \mathtt{SOME}} T_{ij}$ TO $\mathtt{M}$ fi
 4  /* act as manager, if necessary */
 5  if $i = \mathtt{M}$ then
 6          for $i \in \mathtt{SOME}$ do RECEIVE $S_i, \bigcup_{j \in \mathtt{SOME}} T_{ij}$ FROM $i$ enddo
 7          SCCTarjan $(S_{\mathtt{SOME}}, T_{\mathtt{SOME}}, \mathsf{scc\_}p)$
 8          for $i \in \mathtt{SOME}$ do SEND $\mathsf{scc\_}p(S_i)$ TO $i$ enddo
 9  fi
10  /* get the new scc */
11  if $i \in \mathtt{SOME}$ then RECEIVE $\mathsf{scc}(S_i)$ FROM $\mathtt{M}$ fi
12  update
```

```
collapse-partial-all ($\mathtt{SOME}_1 \cdots \mathtt{SOME}_m$)
Precondition: $(\forall i, j$ with $1 \le i < j \le m)\ \mathtt{SOME}_i \cap \mathtt{SOME}_j = \emptyset$

        collapse-partial$(\mathtt{SOME}_1, \mathsf{scc}) \,||\, \cdots \,||\,$ collapse-partial$(\mathtt{SOME}_m, \mathsf{scc})$
```

Figure 2: Partial SCC detection

more managers and apply the transformation above on disjoint subsets of workers, in parallel. This way, the managers get a smaller global graph (this may make the difference between not feasible and feasible). Moreover, the chance of finding components is higher than when collapsing locally. By repeatedly collapsing SCCs on random small sets of workers, we hopefully arrive at a global graph that is small enough to be further reduced on one worker.

Distributed implementation Fig.2 shows two variants of the collapse-partial transformation. Let SOME be the subset of workers under consideration and let $\mathcal{S}_{\mathtt{SOME}} = (S_{\mathtt{SOME}}, T_{\mathtt{SOME}})$ be the subgraph induced by the states owned by workers in SOME. The idea is to simply send $\mathcal{S}_{\mathtt{SOME}}$ to a special worker M (step 3), that will locally compute the scc function (step 7) and send it back (step 8) to the workers in SOME. Note that the assigned scc values may not be the final answers; they can be overwritten in future transformations that consider larger parts of the graph. Since disjoint subsets of workers generate disjoint subgraphs, the partial SCC reduction can be executed in parallel on more subsets of workers (**collapse-partial-all**).

Complexity The number of messages used in **collapse-partial** is $\mathcal{O}\ (\texttt{size}(\texttt{SOME}))$, with a total size of $\mathcal{O}\ (\texttt{size}(S_{\texttt{SOME}}) + \texttt{size}(T_{\texttt{SOME}}))$. The time complexity is also $\mathcal{O}\ (\texttt{size}(S_{\texttt{SOME}}) + \texttt{size}(T_{\texttt{SOME}}))$.

4.3 Update

The two routines above (identify atomic SCCs and collapse-partial) only assign values to the scc function, without actually replacing x with $\mathsf{scc}(x)$, that is without renaming the transitions from (x, y) to $(\mathsf{scc}(x), \mathsf{scc}(y))$. This is the done by a simple `update` routine: first the destination states of all transitions (let y be such a state) are replaced by their new representative ($\mathsf{scc}(y)$), then the source states (x) replaced by $\mathsf{scc}(x)$ and moved to their respective new owners, and finally the updated transitions are moved to their new owners. The implementation is rather straightforward and we do not give a pseudocode for it.

4.4 Reducing the problem by coloring

With a certain state coloring, a partition of the set of states can be achieved, such that if x and y are in the same SCC then x and y have the same color; this splits the SCC problem in smaller disjoint instances. The coloring procedure starts with an initial color function $c : S \to \mathbf{N}$ satisfying a property that we call *safety*, which ensures that whenever two states are colored the same they must be in the same SCC:

$$\forall x, y \in S \ \text{ if } c(x) = c(y) \text{ then } (x, y) \in T^* \cap T^{*-1}$$

If there is no a priori information available that allows the fast construction of a safe initial coloring, we can choose the identity function, which trivially satisfies the condition. We assume an order on the colors, $<$. The coloring procedure consists in successively modifying c until no modification is possible anymore. At each modification step, every state x passes its color to every successor y for which $c(x) < c(y)$. When the coloring is done, the transitions with the source and destination colored differently are removed, because they cross SCCs. The result is a set of disconnected and smaller state spaces, each of them uniformly colored. Note also that every small state space has one or more special states that kept their initial color – let us call them *roots*.

Properties of the final coloring. Let $(S^{start}, T^{start}), \mathit{col}^{start}$ be a state space and an initial color function and let $(S, T), \mathit{col}$ be the state space and color function after the colorify action. Let us also define a set $\texttt{Roots} = \{x \in S \mid \mathit{col}^{start}(x) = \mathit{col}(x)\}$. The following facts are true:

- every SCC in T^{start} (and T) is painted uniformly by *col*
  Proof: at the end of the painting procedure, $col[x] \geq col[y]$ for every transition (x, y). This means that $col[x] \geq col[y]$ for any x, y s.t. $(x, y) \in T^*$. If two states x and y are in the same strongly connected component then there are paths in T^* from x to y and from y to x. Thus $col[x] \geq col[y]$ and $col[y] \geq col[x]$, hence equal.

- if $(x, y) \in T^*$ then $col(y) = col(x)$
  Proof: At the end of the coloring procedure, all the transitions (x, y) with $col(x) \neq col(y)$ are eliminated.

- if col^{start} is safe then: if $x \in \texttt{Roots}$ then $col(y) = col(x)$ iff $(x, y) \in T^*$
  Proof: Since col^{start} is safe, all the states z with $col^{start}(z) = col^{start}(x)$ must be on a cycle with x. If $col(y) = col(x)$ then there is a path (possibly empty) from one of these states to y, because the colors propagate only on paths. It follows that there is also a path from x to y. The other direction was proved at the previous point.

Computing scc **from the final coloring.** These observations justify the claim that the final coloring partitions S into subsets $S^0 \cdots S^{n-1}$ such that any strongly connected component from the initial graph is completely contained in one of the subgraphs induced $(S^0, T^0) \cdots (S^{n-1}, T^{n-1})$. The subgraphs are actually the forward reachability sets of a few selected states (roots). Moreover, solving the problem of detecting the strongly connected components in the initial graph reduces to solving it for the n subgraphs. The final scc mapping is simply the union of the scc sub-mappings thus resulted.

Distributed implementation. Fig.3(up) shows the distributed procedure **colorify** that takes a color function *col* on the states of a graph and modifies it repeatedly until it stabilizes, that is until $col[x] \geq col[y]$ for every transition (x, y). The modifying step identifies the transitions (x, y) that do not conform to this condition and copies the color of the parent to the child.

The colorify routine finishes in about $\texttt{size}(T) * diameter$ steps, which is quite reasonable for state spaces, that usually have a very small diameter.

Heads off. Optionally, we can exploit the colors somewhat more, by finding and extracting some SCCs. Pick a root x. Since the initial coloring was safe, the states wearing x's color are precisely those reachable from x. Thus, the states belonging to the SCC of x are those that are colored the same as x and can reach x.

```
**colorify** ($col^{start}$)                                                    returns $col$
*Precondition:* (safe) if $col^{start}(x) = col^{start}(y)$ then $(x,y) \in T^* \cap T^{*-1}$
*Postcondition:* $\forall x, y \in S\ \forall (x,y) \in T\ col(x) = col(y)$
    $col := col^{start}$
    /* loop:the color of the parent propagates */
    /* to the child, if the color of the child is weaker */
    **DBSUM**($\texttt{size}(S_i)$ , $totalC$)
    $Changed := S_i$
    **while** $totalC > 0$ **do**
        $newC := \emptyset$
        **for** all workers $j$ **do**
            $B_{ij} := \{(y, col[x]) \mid (x,y) \in T_{ij}$ and $x \in Changed\}$
            **SEND** $B_{ij}$ **TO** $j$ || **RECEIVE** $B_{ji}$ **FROM** $j$
            **for** $(y,c) \in B_{ji}$ **do**
                **if** $(c < col[y])$ **then** $col[y] := c$; $newC := newC \cup \{y\}$ **fi**
            **enddo**
        **enddo**
        $Changed := newC$; **DBSUM**($\texttt{size}(Changed)$ , $totalC$)
    **enddo**
    /* mark as final all the transitions between */
    /* states of different colors */
    **for** all workers $j$ **do**
        **REQ-REP** $j$ **:** $\texttt{Dest}_{ij} := col(\texttt{Dest}_{ij})$
        **for** $(x,y) \in T_{ij}$ s.t. $col[x] \neq col[y]$ **do**
            $T_{ij} := T_{ij} - \{(x,y)\}$; $\texttt{finalT} := \texttt{finalT} \cup \{(x,y)\}$
        **enddo**
    **enddo**
    return $col$
```

```
**heads-off** ($col$, $\texttt{Roots}$)
*Precondition:* $\forall (x,y) \in T\ col(x) = col(y)$
        $\wedge\ \forall x \in S\ \exists$ a unique $r \in \texttt{Roots}\ col(x) = col(r)$
    /* reverse the transitions */
    **for** all workers $j$ **do**
        **SEND** $\texttt{Source}_{ij}, \texttt{Dest}_{ij}$ **TO** $j$; **RECEIVE** $\texttt{Dest}_{ij}, \texttt{Source}_{ij}$ **FROM** $j$
    **enddo**
    /* paint the roots with their old color */
    **for** $x \in S_i$ **do** $c[x] := \texttt{size}(S) + 1$ **enddo**
    **for** $r \in \texttt{Roots}$ **do** $c(r) := col(r)$ **enddo**
    $c :=$ **colorify**$(c)$
    **for** $x \in S_i$ **do**
        **if** $c(x) = col(x)$ **then** $\texttt{scc}(x) :=$ the unique $r \in \texttt{Roots}$ s.t. $col(r) = col(x)$ **fi**
    **enddo**
    /* reverse the transitions again */
    **for** all workers $j$ **do**
        **SEND** $\texttt{Source}_{ij}, \texttt{Dest}_{ij}$ **TO** $j$; **RECEIVE** $\texttt{Dest}_{ij}, \texttt{Source}_{ij}$ **FROM** $j$
    **enddo**
    update
```

Figure 3: (Worker i) Graph coloring (up) and elimination of root components (down)

The corresponding routine heads-off, described in Fig.3, gets as input a coloring of the state space together with a set of *roots* (one root per color) with the property that every root can reach all (and only) the states painted in its color. This means that the states that are reachable from their root also on backward paths, form the root's strongly connected component. An easy way to compute the backward reachable states is reversing the state space (steps 2- 4) and coloring it again, with an initial color function that leaves all the non-root states unpainted. The nodes that get painted in this new coloring round are in their root's SCC and can be marked as such – and removed from S. In the end, the state space gets back to the original orientation (13- 15).

4.5 Eliminate reflexive and multiple transitions.

As a result of other transformations, transitions of the form (x, x) and multiple occurrences of the same transition can appear, that have no influence on the SCC. Eliminating these reduces the size of the graph. Since for every state all the outgoing transitions are kept on the same worker, this is a simple local operation and requires no network communication. Therefore, we will not explain it.

4.6 Two overall procedures

Note that every transformation eliminates some of the states and transitions, either by collapsing SCCs or by proving that certain states are atomic or certain transitions are definitely between different components. When discovered, the atomic states are thrown away and the transitions between SCCs are stored in the set `finalT`. After a number of transformations, the set of transitions left in the state space will drop to $\emptyset$. At that moment, scc and `finalT` define the reduced graph, which is the initial one modulo the strong connectivity equivalence relation. But it is possible that scc of some states does not hook them directly to their head of component, but via some intermediate states. To get the final scc definition, a *flattening* phase must be performed, at the end of which $\forall x \in S : \mathsf{scc}(\mathsf{scc}(x)) = \mathsf{scc}(x)$. The distributed implementation of this phase uses just one BFS pass of the graph. This is possible because throughout all the transformations, the following invariant is preserved:

$$\forall x \in S\ \exists \text{ a unique } y \in S \text{ s.t.} \mathsf{scc}(y) = y \wedge \mathsf{scc}(\mathsf{scc}(\cdots \mathsf{scc}(x))) = y.$$

After flattening, the state space without cycles is:

$$S^{\mathsf{scc}} = \{\mathsf{scc}(x) \mid x \in S\}, T^{\mathsf{scc}} = \{(\mathsf{scc}(x), \mathsf{scc}(y)) \mid (x, y) \in \mathtt{finalT}\}$$

```
CE1 - groups:
  elim-atomic
  $groupsize = 1$
  while $groupsize < W$ do
      partition ALL in groups of size (at most) $groupsize$ : $\mathtt{ALL} := \bigcup_{0 \le i < W/groupsize} \mathtt{SOME}_i$
      if $\exists i : \sum_{j \in \mathtt{SOME}_i} \mathtt{size}(T^j) > \mathtt{MAX}$
       then $ERROR$ : group too large
       else
         collapse-partial-all $(\mathtt{SOME}_0 \cdots \mathtt{SOME}_m)$
       fi
      $groupsize := 2 * groupsize$
  enddo

CE2 - colors:
  while $(T \neq \emptyset)$ do
      elim-atomic
      $c$ := colorify $(\mathsf{Self})$
      $\mathtt{Roots} := \{x \in S \mid c(x) = x\}$
      heads-off $(c, \mathtt{Roots})$
  enddo
```

Figure 4: Two overall procedures.

We implemented two SCC reduction algorithms based on the transformations proposed, one using mainly the collapse-partial method, the second using colors (Fig.4). For the first one, a constant MAX is needed to specify the maximum load (in number of transitions) that a worker can handle.

CE1 - groups This algorithm is aimed at speed. It uses a series of collapse-partial-all calls to reduce quickly the size of the distributed graph. The series begins with finding, in parallel, the SCCs on the subgraphs local to every worker (level 0, collapse-partial-all with groupsize 1). Then the groups of workers double in size every step, until only one group including all the workers is considered. If at any step the maximum load is reached, the algorithm stops with an error. This approach will work well for relatively small state spaces and for dense ones, with many small cycles inside of larger ones.

CE2 - colors This algorithm uses only the color-based transformations. Self denotes the identity function, $\mathsf{Self}(x) = x$. The algorithm repeatedly colors the state space starting with Self as initial color function and extracts the head components. Note that, with Self as initial color, every color gets a unique root. This algorithm may in general be slower, but it always terminates successfully, because its memory usage stays more-or-less constant and, moreover, the buffers can be restricted to a convenient size (while in the CE1 case, the manager has to be able to simultaneously store the local graphs of several workers in its memory).

5 Experiments

We evaluated the performance of the two prototypes on a series of large distributed state spaces generated for the verification of a system for lifting trucks [15] (*lift5*, *lift6*), a cache coherence protocol [19] (*cache*), some instances of the Sokoban game [1] (*screen.706*, *screen.801*, *screen.1*) and a sliding window protocol [12] (*swp_piggy*). We also included two state spaces without invisible cycles from the VLTS [10] benchmark (*vasy_8082*, *vasy_4338*). The problem sizes and some other relevant structural characteristics, together with the reduction times are summarized in Fig.5. The times of the distributed algorithms were recorded on a cluster of 4 dual AMD Athlon MP 1600+ nodes with 2G memory each, running Linux and connected by Gigabit Ethernet. For the tests with the sequential algorithm, one of the cluster machines was used. The CE1 and CE2 columns show the runtimes of the two algorithms, not including the I/O operations.

Although the goal of these distributed algorithms is to allow handling state spaces that do not fit in the memory of one machine, and not necessarily an improved performance, nice speedups w.r.t. the sequential tool can be observed on the larger state spaces. For the two smallest examples, the communication cost is too large and leads to a slowdown. The low runtime of both CE algorithms on the case studies with a high percentage of atomic SCCs (even much lower than the sequential Tarjan algorithm!) is mainly due to elim-atomic, which isolates in one pass all the states which are atomic SCCs. Note that in most cases CE1 is faster than CE2. However, this is not true for the Sokoban screens, that have a very small depth, which is very much to the advantage of CE2. The number of iterations in CE2 depends on the diameter of the graph, which is usually rather small for state spaces. In fact, in the experiments above this number was at most 5.

In order to justify that the cycle elimination algorithms can be useful as preprocessing step for branching bisimulation reduction, we also show the runtimes of a distributed branching bisimulation reduction tool [7] on the original state spaces and on the SCC-reduced state spaces. The important observation here is that the cycle reduction times are usually much smaller than the branching bisimulation reduction times. This means that although the cycle elimination step is not always advantageous, it is also not harmful and could be always done, just in case it might provide an important gain (like for the Sokoban screens). Moreover, the current branching bisimulation algorithm is not optimized for the case when the input state space is guaranteed not to contain cycles. There is much space for improving in this direction and then the small penalty paid for eliminating the cycles would completely pay off.

6 Conclusions

We designed and implemented distributed procedures for state space transformations, which, combined in various ways, solve the problem of detecting strongly connected components in large state spaces.

Given that the best single-threaded solution for SCC detection is not efficiently parallelizable, we proposed two (orthogonal) heuristics that approach the problem by reducing it to smaller instances, solvable in parallel. The first idea is to use the Tarjan sequential SCC detection algorithm on groups of workers. The local graphs of several workers get sent to a manager, where the combined subgraph is solved sequentially. Depending on the size of the workers' group, this procedure ranges from solving all local subgraphs in parallel to solving the whole global graph. Due to memory limitations, this approach is not always successful. Therefore we also proposed an alternative solution, based on a state coloring strategy which helps to compute and intersect forwards and backwards reachability sets. This is done by exclusively employing BFS traversals, which are well suitable for parallel implementations, especially since the state spaces usually have a small height.

A third heuristics that proved very helpful uses the observation that if a state is on a non-trivial cycle then it must be reachable from at least one other state placed on a cycle. Thus, the states without parents cannot be on cycles. They are their own (atomic) SCC, and so are also all the states reachable only from atomic SCC. Removing these states is quite cheap and the state space usually becomes significantly smaller. The same operation applied on the reversed state space reduces it even more. The strongest effect of eliminating atomic components can be seen on state spaces without cycles, when the SCC detection finishes in one pass, thus linear time.

The reasonable performance of our algorithms demonstrates that cycle elimination on very large distributed state spaces is feasible and useful. Since SCC detection is one of the most popular graph problems, it is probable that applications for these algorithms can be found outside the verification area as well.

Future work. The obvious further step is to optimize the distributed branching bisimulation algorithm of [7] under the assumption of absence of cycles and integrate the SCC detection as preprocessing phase there. Another attractive continuation is to implement an SCC detection algorithm that combines the methods from CE1 and CE2. This algorithm would use the technique of partitioning by colors only until the graph pieces are small enough to be collapsed by the groups

state	**size of the state space**			**no. atomic**	**size of**	**Runtimes (s)**			**Runtimes BB (s)**	
space	size(S) (in 10^6)	size(T) (in 10^6)	τs (in 10^6)	**SCCs** (in % of S)	**largest SCC**	**seq.**	**CE1**	**CE2**	**after CE**	**original**
screen.706	1.2	2.7	2.4	6.6	38	4.4	7.7	12.4	9.6	106.8
lift5	2.1	8.7	3.8	98.9	165	17.8	8.6	14.7	79	86
vasy_4338	4.3	15.6	3.1	100.0	1	31.74	6	6	82	82
vasy_8082	8.0	42.9	2.5	100.0	1	103.9	11	11	27	27
screen.801	20.7	49.7	44.9	4.3	50	145.7	172	112.5	43.6	2819.2
swp_piggy	9.6	53.4	30.9	10.5	45	197.3	125	237	122	341
cache	7.8	59.1	22.8	99.5	248	153.6	45	47	1331	1394
screen.1	29.9	72.3	65.9	1.2	50	-	210	121	36.7	180 000
lift6	33.9	165.3	74.1	99.8	486	-	160	305	930	1039

Figure 5: Some case studies: size, structure, reduction times

technique. Further, using the distribution method presented in [18] could lead to a significant performance improvement, especially for the partial SCC detection.

References

[1] Sokoban. `http://www.cs.ualberta.ca/~games/Sokoban/`.

[2] A. Aho, J. Hopcroft, and J. Ullman. *Data structures and algorithms*. Addison-Wesley, 1983.

[3] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model checking. In *Proceedings ASE'03*, pages 106–115. IEEE Computer Society, 2003.

[4] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL model-checking in SPIN. In *Proceedings SPIN'01*, volume 2057 of *LNCS*, pages 200–216, 2001.

[5] S.C.C. Blom, I. van Langevelde, and B. Lisser. Compressed and distributed file formats for labeled transition systems. In *Proceedings PDMC'03*, volume 89 of *ENTCS*, 2003.

[6] S.C.C. Blom and S.M. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. In *Proceedings PDMC'02*, volume 68 of *ENTCS*, 2002.

[7] S.C.C. Blom and S.M. Orzan. Distributed branching bisimulation reduction of state spaces. In *Proceedings PDMC'03*, volume 89 of *ENTCS*, 2003.

[8] B. Bollig, M. Leucker, and M. Weber. Parallel model checking for the alternation free μ-calculus. In *Proceedings TACAS'01*, volume 2031 of *LNCS*, pages 543–558, 2001.

[9] I. Černá and R. Pelánek. Distributed explicit fair cycle detection (set based approach). In *Proceedings SPIN'03*, volume 2648 of *LNCS*, pages 49–73, 2003.

[10] CWI/SEN2 and INRIA/VASY. The VLTS benchmark. `http://www.inrialpes.fr/vasy/cadp/resources/benchmark\_bcg.html`.

[11] L.K. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In *Proceedings Irregular'00*, volume 1800 of *LNCS*, pages 505–512, 2000.

[12] W. Fokkink, J.F. Groote, J. Pang, B. Badban, and J.C. van de Pol. Verifying a sliding window protocol in μCRL. In *Proceedings AMAST'04*, LNCS, 2004.

[13] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proceedings SPIN'01*, volume 2057 of *LNCS*, pages 217–234, 2001.

[14] H. Gazit and L. Miller. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Information Processing Letters*, 28:61–65, 1988.

[15] J.F. Groote, J. Pang, and A.G. Wouters. Analyzing a distributed system for lifting trucks. *Journal of Logic and Algebraic Programming*, 55(1–2):21–56, 2003.

[16] C. Joubert and R. Mateescu. Distributed on-the-fly equivalence checking. In *Proceedings PDMC'04*, ENTCS, 2004.

[17] W. C. McLendon III, B.A. Hendrickson, S.J. Plimpton, and L. Rauchwerger. Identifying strongly connected components in parallel. In *Proceedings SIAM PP01*, 2001.

[18] S.M. Orzan, J.C. van de Pol, and M. Valero Espada. A state space distribution policy based on abstract interpretation. In *Proceedings PDMC'04*, ENTCS, 2004.

[19] J. Pang, W.J. Fokkink, R. Hofman, and R. Veldema. Model checking a cache coherence protocol for a Java DSM implementation. In *Proceedings FMPPTA'03*, 2003.

[20] R. Pelánek. Typical structural properties of state spaces. In *Proceedings SPIN'04*, volume 2989 of *LNCS*, 2004.

[21] S. Rajasekaran and I. Lee. Parallel algorithms for relational coarsest partition problems. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):687–699, 1998.

[22] J.H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[23] J.M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings of the 10th European PVM/MPI Users' Group Meeting*, volume 2840 of *LNCS*, 2003.

[24] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.