, , 1–23 ()

On-the-Fly Model checking under Fairness that Exploits Symmetry*

VIKTOR GYURIS viktor@math.uic.edu
Mathematics, Statistics and Computer Science Department, The University of Illinois at Chicago, USA

A. PRASAD SISTLA sistla@eecs.uic.edu
Electrical Engineering and Computer Science Department, The University of Illinois at Chicago, USA

Editor: E. A. Emerson

Abstract. An on-the-fly algorithm for model checking under fairness is presented. The algorithm utilizes symmetry in the program to reduce the state space, and employs novel techniques that make the on-the-fly model checking feasible. The algorithm uses state symmetry and eliminates parallel edges in the reachability graph. Experimental results demonstrating dramatic reductions in both the running time and memory usage are presented.

Keywords: Model checking, State explosion, Symmetry reduction, Automata, Verification

1. Introduction

The state explosion problem is one of the major bottlenecks in temporal logic model checking. Many techniques have been proposed in the literature [6, 5, 9, 8, 13, 11, 12, 16, 17] for combating this problem. Among these, symmetry based techniques have been proposed in [5, 9, 13]. In these methods the state space of a program is collapsed by identifying states that are equivalent under symmetry and model checking is performed on the reduced graph. Although the initial methods of [5, 9] could only handle a limited set of liveness properties, a more generalized approach for checking liveness properties under various notions of fairness has been proposed in [10]. This method, however, does not facilitate early termination, it supplies an answer only after the construction of all the required data structures is complete.

Many traditional model checking algorithms ([3, 11, 12, 17]) use on-the-fly techniques to avoid storing the complete state space in the main memory. However, none of these techniques employ symmetry. [13] uses on-the-fly techniques together with symmetry for model checking. There the focus is on reasoning about a simple but basic type of correctness, i.e., safety properties expressible in the temporal logic CTL by an assertion of the form AG$\neg$`error`.

In this paper, we present an on-the-fly model checking algorithm that checks for correctness under weak fairness and that exploits symmetry. (A computation is said

* A preliminary version of this paper appeared in the Proceedings of the 9th International Conference on Computer Aided Verification held in Haifa, Israel in June 1997. The work presented in this paper is partially supported by the NSF grants CCR-9623229 and CCR-9633536

to be weakly fair if every process is either infinitely often disabled or is executed infinitely often). This work is an extension of the work presented in [10]. Here we develop additional theory leading to novel techniques that make the on-the-fly model-checking feasible. We not only exploit the symmetry between different states, but also take advantage of the symmetric structure of each individual state; this allows us to further reduce the size of the explored state space.

The other major improvement is gained by breaking the sequential line of the algorithm. The original algorithm constructed three data structures (the reduced state space, the product graph and the threaded graph — details are given below) one after the other and performed a test on the last one. We eliminated the construction of the third data structure by maintaining some new dynamic information; the algorithm constructs parts of the first structure only when it is needed in the construction of the second; finally we store only the nodes in the second structure. This on-the-fly construction technique and up-to-date dynamic information maintenance facilitates early termination if the program does not satisfy the correctness specification and allows us to construct only the minimal necessary portion of the state space when the program satisfies the correctness specification. The on-the-fly model checking algorithm has been implemented and experimental results indicate substantial improvement in performance compared to the original method.

The algorithm, given in [10], works as follows. It assumes that the system consists of a set I of processes that communicate through shared variables. Each variable is associated with a subset of I, called the *index set*, that denotes the set of processes that share the variable. Clearly, the index set of a local variable consists of a single process only. A state of the system is a mapping that associates appropriate values to the variables. A permutation π over the set I of processes, extends naturally to a permutation over the set of variables and to the states of the system. A permutation π is an automorphism of the system if the reachability graph of the system is invariant under π (more specifically, if $s \rightarrow t$ is an edge in the reachability graph then $\pi(s) \rightarrow \pi(t)$ is also an edge and vice versa). Two states are equivalent if there is an automorphism of the system that maps one to the other. Factoring with this equivalence relation compresses the reachable state space.

The original method consists of three phases. First it constructs the reduced state space. Then it computes the product of the reduced state space and the finite state automaton that represents the set of incorrect computations. It explores the product graph checking for existence of "fair" and "final" strongly connected components; these components correspond to fair incorrect computations. Checking if a strongly connected component is fair boils down to checking if it is fair with respect to each individual process. This is done by taking the product of the component and the index set I. The result is called *the threaded graph resolution* of the component. A path in the threaded graph corresponds to a computation of the system with special attention to one designated process. Fairness of a strongly connected component in the product graph is checked by verifying that each of the strongly connected components of its threaded graph are fair with respect to the designated process of that component.

Our on-the-fly algorithm has two layers: the reduced state space and the product graph construction. The successors of a node in the reduced state space are constructed only when the product graph construction requests it. The product graph construction is engined by a modified algorithm for computing strongly connected components (*scc*) using depth first search (see [2]). During the depth first search, with each vertex on the stack it maintains a partition vector of the process set I. The partition vector associated to a product state u captures information about the threaded graph of the strongly connected component of u in the already explored part of the product graph. Intuitively, if processes i and j are in the same partition class then it indicates that the nodes (u, i) and (u, j) are in the same *scc* of the threaded graph. This reveals that the infinite run of the system corresponding to the strongly connected component of u in the already explored part of the product graph is either fair with respect to both processes i and j or it is not fair with respect to any of the mentioned processes. The partition vectors are updated whenever a new node or an edge to an already constructed node is explored. The correctness of the above algorithm is based on new theory that we develop as part of this paper; this thoery connects the partition vectors of the algorithm with the strongly connected components of the threaded graph.

A permutation $\pi \in Aut\, M$ on processes is a *state symmetry* of a state s, if $\pi(s) = s$ (state symmetry was originally introduced in [9]). Suppose that π maps process i to j. In that case, transitions ignited by process i are in one-to-one correspondence with those caused by process j. Hence, we can save space and computation time by considering only those that belong to process i. This is one of the forms *state symmetry* is exploited in our algorithm. Another way is the initialization of the partition vector with the state symmetry partition. If π maps process i to j then the threads corresponding to process i and j are certainly in the same situation: they are either both fair or none is fair.

Our paper is organized as follows. Section 2 contains notation and preliminaries. In Sect. 3 we develop the necessary theory and present the on-the-fly algorithm. We describe various modifications of the algorithm that take state symmetry into consideration. Section 4 presents experimental results showing the effectiveness of our algorithm and dramatic improvements in time as well as memory usage. Section 5 contains concluding remarks.

2. Preliminaries

2.1. Programs, Processes, Global State Graph

Let I be a set of *process indices.* We consider a system $P = ||_{i \in I} K_i$ of processes running in parallel. Each process K_i is a set of transitions. We assume that all variables of P are indexed by a subset of I indicating which processes share the variable. A system P, that meets the above description, is called an *indexed transition system* (or briefly *program*).

A *global state* of an indexed transition system is an assignment of values to the variables. We assume that each variable can take only a finite number of values.

This assumption ensures that the number of global states of the system is also finite. We define an indexed graph M on the set of global states that captures the behavior of the program. The *indexed global state graph* is $M = \langle S, R, s_0 \rangle$ where S is the set of global states, s_0 is the initial state, and $R \subseteq S \times I \times S$ is the transition relation, i.e., $s \xrightarrow{i} t \in R$ if there is a transition in process i that is enabled in state s and its execution leads to state t.

2.2. Strongly Connected Subgraphs and Weak Fairness

Infinite paths in M starting from the initial state denote computations of P. An infinite path p in M is *weakly fair* if for each process i, either i is disabled infinitely often in p or it is executed infinitely often in p. Unless otherwise stated, we only consider weak fairness throughout the paper. The implementation, however, is capable of handling strong fairness. (An infinite execution of the program is *strongly fair* if every process that is enabled infinitely often is executed infinitely often.)

A *strongly connected subgraph* of a graph is a set of nodes such that there is a path between every pair of nodes passing only through the nodes of the subgraph. A *strongly connected component* (*scc* for short) is a maximal strongly connected subgraph.

The set of states that appear infinitely often in an infinite computation of a finite state program P forms a strongly connected subgraph of M. Many properties of infinite computations, such as fairness, are in one-to-one correspondence with properties of the associated strongly connected components of M. Therefore, all of our efforts will be directed towards finding an *scc* of M with certain properties.

We can formulate weak fairness as a condition on *scc*'s.

Definition 1. An *scc* C of M is weakly fair if every process is either disabled in some state of C or executed in C. (Process i is disabled in state s if all of its transitions are disabled in s; process i is executed in C if there are states s, t in C such that $s \xrightarrow{i} t \in R$.)

For a given program P, we are interested in checking the absence of fair and incorrect computations. Here we assume that incorrectness is specified by a Buchi automaton A; the set of computations accepted by A is exactly the set of incorrect computations. The problem of checking whether the program P has any incorrect weakly fair computation can be decided by looking at the product graph B_0 of M and A. If B_0 has an *scc* whose A projection contains a final automaton state and whose M projection is weakly fair then P has a weakly fair incorrect computation. Here it is sufficient if we construct the product of the reachable part of M and A.

2.3. Annotated Quotient Structure

The previously suggested method, i.e., of analyzing the reachable part of M, can be very expensive since, for many systems, the reachable part of M is huge. For systems that exhibit a high degree of symmetry, the state space can be reduced by identifying states that are equivalent under symmetry and by constructing the quotient structure as given below.

Denote the set of all permutations on I by $Sym\,I$, let $\pi \in Sym\,I$. π induces an action on the set of variables and on the set of global states in the following way. For every variable $v_{i_1,\ldots,i_k}$, its image under π is $\pi(v_{i_1,\ldots,i_k}) = v_{\pi(i_1),\ldots,\pi(i_k)}$. Of course, $\pi(v_{i_1,\ldots,i_k})$ may not be a variable of the program. We say that π *respects the set of variables* if the image of every variable is a variable of the program. Assume that π has that property. The image of a global state s under π is defined to be the global state $\pi(s)$ that satisfies that the value of $v_{i_1,\ldots,i_k}$ in s is the same as the value of $\pi(v_{i_1,\ldots,i_k})$ in $\pi(s)$ for every variable $v_{i_1,\ldots,i_k}$ of the program. We say that π is an *automorphism* of the indexed global state graph M if π respects the variables, $\pi(s_0) = s_0$, and $s \xrightarrow{i} t \in R$ exactly when $\pi(s) \xrightarrow{\pi(i)} \pi(t) \in R$. The set of automorphisms of M is denoted by $Aut\,M$. Certainly, $Aut\,M$ is a subgroup of $Sym\,I$.

Given any subgroup G of $Aut\,M$, we can define an equivalence relation on S. State s is equivalent to t if there is a $\pi \in G$ such that $\pi(s) = t$. Using G, M can be compressed to a smaller structure $\overline{M} = \langle \overline{S}, \overline{R}, s_0 \rangle$, called the *annotated quotient structure (AQS) for M*, as follows.

- $\overline{S}$ is a set of *representative states* that contains exactly one state from each equivalence class of S/G, in particular, it contains s_0 itself from s_0's class.

- $\overline{R}$ is a set of triples $s \xrightarrow{\pi,i} t$ denoting edges between representative states annotated with permutations from G and with process indices. To define $\overline{R}$ formally, with each state $t \in S$, we define t_{rep} to be the unique representative state of the equivalence class of t; with each such t, we also associate a canonical permutation $\pi_t \in G$ such that $t = \pi_t(t_{\mathrm{rep}})$. Now, we define $\overline{R} = \{s \xrightarrow{\pi_t,i} t_{\mathrm{rep}} \ : \ s \in \overline{S}, t \in S \text{ and } s \xrightarrow{i} t \in R\}$.

Remark. In many cases it is useful to allow multiple initial states to capture nondeterminism of the program. In [10] M is defined to have a set S_0 of initial states and it is required that for every automorphism π and $s \in S_0$ $\pi(s) \in S_0$. Their concept of multiple initial state can be simulated in our system by introducing a new, fully symmetric initial state s_0 and for every $s \in S_0$ and $i \in I$ an edge $s_0 \xrightarrow{i} s$.

2.4. Checking for Fairness and the Threaded Graph

We briefly outline the approach taken in [10] for checking if a given concurrent program P exhibits a fair computation that is accepted by an automaton A. Assume that the automaton A refers to variables whose index set involve processes $c_0, \ldots, c_{k-1}$ only; that is, A specifies a property of the executions of these processes only. These processes are called *global tracked processes.* If we traverse in the compressed $\overline{M}$ these global tracked processes are represented by different sets in each state of the path. Formally, the *local tracked processes* in a state t after passing through the path p are $\pi^{-1}(c_0), \ldots, \pi^{-1}(c_{k-1})$ where π is the product of the permutations found on the edges of p. For example, if the first edge of the path is $s_0 \xrightarrow{\pi,i} t$ then, since t is only a representative of the real successor state, the global tracked processes are represented by the processes $\pi^{-1}(c_0), \ldots, \pi^{-1}(c_{k-1})$ in t. After some steps in the path we may return to state t again but at that time we encounter a different set of local tracked processes. This property makes $\overline{M}$ not feasible for model checking purposes. $\overline{M}$ is too compact, we need a less compressed version of M where the set of local tracked processes in a given state t does not depend on the path that lead to t from the initial state. We need to unwind $\overline{M}$ partially; the threaded graph construction captures this unwinding.

Let $H = \langle V, E \rangle$ be any graph whose edges are labeled with permutations of a set I, and possibly with other marks. The *k-threaded graph* $H^{k\text{-thr}}$ corresponding to H is $\langle V \times I^k, E^{k\text{-thr}} \rangle$ where $E^{k\text{-thr}}$ consists of edges of the form $(s, i_0, \ldots, i_{k-1}) \xrightarrow{\pi,\lambda} (t, j_0, \ldots, j_{k-1})$ with $s \xrightarrow{\pi,\lambda} t \in E$ and $j_l = \pi^{-1}(i_l)$ for all $l \in k$. Note that if H has labels on its edges (denoted by λ in the previous line) other than the permutations of a I then $H^{k\text{-thr}}$ inherits them. The 1-threaded graph corresponding to H is denoted by H^{thr}. The second component in a state (s, i) of H^{thr} is called the *designated* process. The following simple example depicts these concepts. Here, and throughout the paper, *id* denotes the identity permutation, π_{ij} the transposition that interchanges i and j.

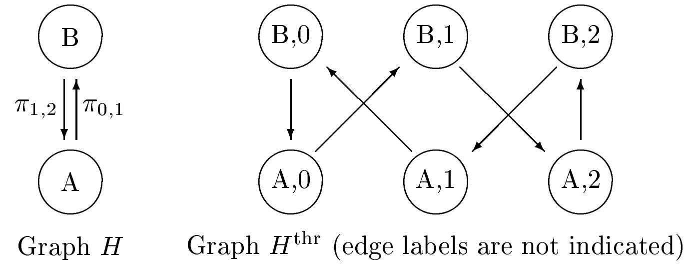

Figure 1. The threaded graph construction

The original algorithm first constructs the annotated quotient structure $\overline{M}$ corresponding to P. In the second step, a product graph $\overline{B}_0 = \overline{M}^{k\text{-thr}} \times A$ is constructed. Each state of this product graph is of the form $(s, i_0, \ldots, i_{k-1}, a)$; $(s, i_0, \ldots, i_{k-1}, a) \xrightarrow{\pi,u} (t, j_0, \ldots, j_{k-1}, a')$ is an edge of $\overline{B}_0$ if $(s, i_0, \ldots, i_{k-1}) \xrightarrow{\pi,u} (t, j_0, \ldots, j_{k-1})$ is an edge of $\overline{M}^{k\text{-thr}}$ and the automaton A has a transition from state a to a' on the input consisting of the program state obtained by simultaneously replacing index c_l by i_l for each $l \in k$.

In the third step the product graph $\overline{B}_0$ is checked for existence of fair strongly connected components (these are called subtly fair *scc*s in [10]). This checking is done by constructing the threaded graph resolution of every *scc* of $\overline{B}_0$. Every *scc* of the threaded graph is checked if it is fair with respect to the designated process.

In Sect. 3 below we show that, using techniques based on new and deeper theoretical results, the above method can be considerably enhanced.

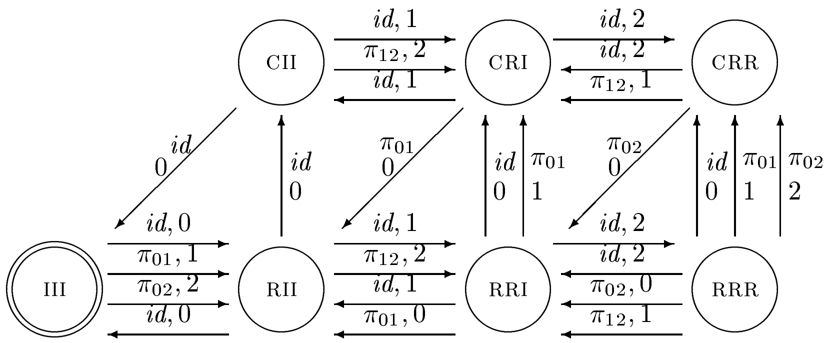

Figure 2. The AQS $\overline{M}$ for the simplified Resource Controller

```
StartingState A0;
FinalState    A1;

STATE[1] != C -> A0 : A0;
STATE[1] == C -> A0 : A1;
True          -> A1 : A1;
```

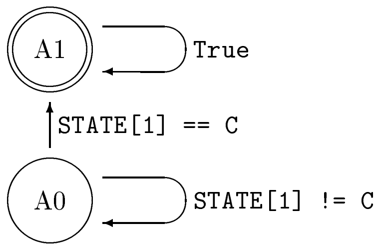

Figure 3. The automata A

2.5. *The Simplified Resource Controller Example*

To illuminate our general concepts we present the instructive example of the simplified *Resource Controller.*

The program consists of a server and 3 client processes running in parallel. Each client is either in idle (I), request (R) or critical (C) state. The variable `STATE[c]` indicates the status of client `c` (`c` $= 1, 2, 3$). Clients can freely move between idle and request state. The server may grant the resource to a client by moving it to critical state, provided that that client is in request state and no client is in critical state yet.

For this simple example, M has 20 states (all combinations of values I, R and C except those that contain more than one C.) As a contrast, $\overline{M}$ has only 7 states. (as illustrated in Fig. 2.)

The initial state is the one marked with III in the lower left side of the figure. All three process are in idle (I) state. Any of them can move to request state (R), hence it has three successors in the state space: RII, IRI and IIR. All of these states are equivalent, we chose RII to be the representative of them. The three edges departing from III correspond to the three enabled transitions. Edge III $\xrightarrow{\pi_{01},1}$ RII for example indicates that process 1 has an enabled transition and the execution of that transition leads to state $\pi_{01}^{-1}(\textsc{rii}) = \textsc{iri}$. Similarly, state RRI represents 3 states: itself, RIR and IRR.

Suppose that we want to check the (obviously false) property that *Client 1 never gets to critical state.* The negation of that property can be captured by the automaton given in Figure 3; this automaton states that Client 1 eventually gets to a critical state. The global tracked process is 1. The product graph $\overline{B}_0$ has $7 \cdot 3^1 \cdot 2 = 42$ states. A depth first search on $\overline{B}_0$ reveals that it has a strongly connected subgraph that is weakly fair and contains a final automaton state.

3. Utilizing State Symmetry

The original algorithm, briefly described in subsection 2.4, constructed $\overline{B}_0$ together with the threaded graph $\overline{B}_0^{\mathrm{thr}}$. This method can be improved by applying the following three new ideas.

In constructing $\overline{M}^{k\text{-thr}}$ our goal was to define a less compressed version of $\overline{M}$ with the property that if we visit a state t multiple times by an infinite path then we encounter the same set of local tracked processes. In that sense, we can link the set of local tracked processes to that state. The k-threaded graph unwinding of $\overline{M}$ was not the optimal solution. We can define an equivalence relation on $M \times I^k$ that usually results in greater compression: $\overline{M \times I^k}$ still has the desired property, and it is smaller than $\overline{M}^{k\text{-thr}}$ in cases where the program exhibits some symmetry. (It is possible for two states $(s, i_1, \ldots, i_k)$ and $(s, j_1, \ldots, j_k)$ of $\overline{M}^{k\text{-thr}}$ to be equivalent and be represented by a single state in $\overline{M \times I^k}$.)

The second improvement is the application of an on-the-fly algorithm. Here we incrementally construct $\overline{B}$ and simultaneously explore it. By this exploration we analyze the threaded graphs without constructing them. If the partially explored $\overline{B}$ contains a required subgraph then the algorithm immediately exits saving further computation time. Because of the on-the-fly nature of the algorithm, we do not need to store the complete $\overline{B}$. Specifically, no edges need to be stored.

Finally, the third idea is to use the symmetry of a single global state. Using state symmetry we can reduce the number of edges by eliminating the redundant parallel ones. Such redundant parallel edges can be removed from $\overline{M}$ also. This results in further reduction in memory usage.

For keeping the presentation simple, we assume that we are tracking only one process. Doing so, we do not loose generality. All the results, that are presented below, apply (with the obvious modifications) to the case with many tracked processes. In the actual implementation of the algorithm given below, we used the general case.

3.1. *Compressing* $\boldsymbol{M} \times \boldsymbol{I}$

In Subsect. 2.3 we defined an equivalence relation on S. Now, we extend it to $S \times I$ as follows. We say that $u = (s, i)$ and $v = (t, j)$ are equivalent if there is a permutation $\pi \in G$ such that $\pi(u) = v$, that is, $\pi(s) = t$ and $\pi(i) = j$. Obviously, this equivalence relation partitions the set $S \times I$ into a set of equivalence classes. Let $\overline{S}_{\rm aqsi}$ be a set of *representative* states that contains exactly one state from each equivalence class. To ensure that $\overline{S}_{\rm aqsi}$ and $\overline{S}$ are closely related we adopt the convention that $(s,i) \in \overline{S}_{\rm aqsi}$ implies $s \in \overline{S}$, that is, $\overline{S}_{\rm aqsi} \subseteq \overline{S} \times I$. This containment may be strict as it is possible for two states of the form (s,i) and (s,j) to be equivalent. The *annotated quotient structure of M with a tracked index* (AQSI) is $\overline{M \times I} = \langle \overline{S}_{\rm aqsi}, \overline{R}_{\rm aqsi}, s_0 \rangle$ where $\overline{R}_{\rm aqsi} = \{(s,i) \xrightarrow{\pi,l} (t,j) : s \overset{\pi,l}{\to} t \in \overline{R}, j = \pi^{-1}(i)\}$. In a state (s,i), i is the *local tracked process.* Note that the indicated initial state s_0 is formaly not an element of the set of states $\overline{S}_{\rm aqsi}$, no specific tracked process is assigned with it. This seemingly unnatural definition was adopted because we did not want to encorporate information on the automaton A into the definition of $\overline{M \times I}$.

Since $\overline{S}_{\rm aqsi} \subseteq \overline{S} \times I$, $\overline{M \times I}$ can be considerably smaller than $\overline{M}^{\rm thr}$. In the best case, we may achieve a reduction in the number of nodes and number of edges by a factor of n and n^2, respectively. (Here, n is the size of I.)

In our simplified Resource Controller example $\overline{M}^{\rm thr}$ has $7 \cdot 3 = 21$ states. To calculate the size of $\overline{S}_{\rm aqsi}$ note first that in state III all 3 processes are in the same local state, hence (III, 0), (III, 1) and (III, 2) are equivalent in $M \times I$ so only one of them needs to be included in $\overline{S}_{\rm aqsi}$. Similarly, only one of (RRR, 0), (RRR, 1) and (RRR, 2) is in $\overline{S}_{\rm aqsi}$. If s is any of RII, RRI, CRR or IIC then two of $(s,0)$, $(s,1)$ and $(s,2)$ are equivalent, hence only two need to be stored in $\overline{S}_{\rm aqsi}$. Finally, in the case of $s =$ CRI all processes are in different local state, therefore, all $(s,0)$, $(s,1)$ and $(s,2)$ should be in $\overline{S}_{\rm aqsi}$. This counting shows that $\overline{S}_{\rm aqsi}$ contains only 13 representative states.

Let $\overline{B}$ be the product of $\overline{M \times I}$ and A. Formally, $\overline{B} = \langle \overline{S}_{\rm aqsi} \times A, \overline{R}_{\rm pr}, s_{\rm pr} \rangle$ where $s_{\rm pr} = (s, i'_0, a_0)$ for some i'_0 with the property that (s, i'_0) is the representative of (s, i_0) in $\overline{S}_{\rm aqsi}$. Recall that i_0 is the process being tracked by the automaton A. $\overline{R}_{\rm pr}$ consists of edges $(s,i,a) \xrightarrow{\pi,l} (t,j,a')$ such that (s,i) and (t,j) are in $\overline{S}_{\rm aqsi}$, $(s,i) \xrightarrow{\pi,l} (t,j) \in \overline{R}_{\rm aqsi}$, and the automaton A moves from state a to a' on the input gained from s after replacing all occurrences of index i with i_0.

Definition 2. Let C be an *scc* of $\overline{B}$. An *scc* D of $C^{\rm thr}$ is weakly fair if there is a state (u,k) in D such that process k is disabled in u or D has an edge of type $(u,k) \overset{k}{\to} (v,l)$.

Using slightly modified versions of arguments found in [10] we deduce that $\overline{B}$ contains all the information needed to decide if the program P satisfies the complement of the property given by A. This is stated in the following theorem which can be proved exactly on the same lines as theorem 3.3 and lemma 3.8 of [10] by using our new definition of $\overline{B}$.

THEOREM 1 *P satisfies the complement of the property defined by A if and only if there is no scc C of $\overline{B}$ such that C contains a final automaton state and every scc of $C^{\rm thr}$ is weakly fair.*

4. On-the-Fly Model Checking

The main contribution of the present paper lies in showing that we can search for an *scc* of $\overline{B}$ without requiring the complete $\overline{B}$ to be previously constructed. $\overline{B}$ can be explored *while* we are constructing it in an on-the-fly manner.

As it was mentioned earlier, $\overline{B}$ is constructed to be the product of $\overline{M \times I}$ and A. One of the first improvements is that we do not construct $\overline{M \times I}$ but only the smaller $\overline{M}$. By the construction of $\overline{B}$ we implicitly create $\overline{M \times I}$ and store it as part $\overline{B}$. This is based on the observation that, after a careful choice of representative states, $\overline{M \times I}$ is a threaded graph resolution of $\overline{M}$. By the construction, each of the nodes of the threaded graph are checked for equivalence against all the nodes stored already (implicitely, as part of $\overline{B}$). The new node is stored only if it is the first one in its equivalence class. This is done in command 6 of the algorithm presented below.

In our implementation we can control the way $\overline{M}$ is constructed. In the default (and most efficient) case the successors of an $\overline{M}$ state together with the edges leading to them are created (stored) when they are first needed in the construction of $\overline{B}$. If we want to avoid storing the edges of $\overline{M}$ we can use the second option that recreates them temporarily each time when a $\overline{B}$ state construction requires it. As a third possible option, the implementation allows us to construct $\overline{M}$ in advance. This can be usefull when the program is tested against multiple correctness properties.

The second component, used by the construction of $\overline{B}$, is the automaton A representing the correctness property. As A is small in size compared to other data structures involved, its construction in an on-the-fly manner is not motivated.

After these general introductory lines, let us turn to presenting the actual algorithm. As explained earlier, our on-the-fly model-checking algorithm explores $\overline{B}$ simultaneously as it constructs it. During this process, in order to analyze the threaded graph without explicitly constructing it, we maintain a partition of I with each $\overline{B}$ node on the stack. This partition indicates which processes are known to be in the same strongly connected component of the threaded graph.

4.1. Partitions

First, we would like to adopt the following conventions concerning partitions. We identify equivalence relations and the corresponding partitions on a given set. In that sense, we say that a partition contains another partition if the equivalence relation corresponding to the first partition is a superset of the equivalence relation corresponding to the second partition. The join of two partitions is the smallest partition containing both.

The following two lemmas prove some important properties of *scc*s in $\overline{B}$.

LEMMA 1 *Let C be an scc in $\overline{B}$. Then the following properties hold.*

- *If r, r' are nodes in C and there is a path from (r,i) to (r',j) in C^{thr} then there is a path from (r',j) to (r,i) as well.*

- *The sccs in C^{thr} are disjoint, i.e., no two distinct sccs are connected by a path in C^{thr}.*

Proof: The first part of the lemma is proved as follows. Assume that r, r' are nodes in C and there is a path from (r,i) to (r',j) in C^{thr}. This means that there exists a path p from r to r' in C such that $j = \pi^{-1}(i)$ where π is the product of all the permutations on the path p. Since C is an *scc*, there exists a path p' from r' to r in C. Let π' be the product of the permutations on p'. There exists an $n > 0$ such that $(\pi \circ \pi')^n$ is the identity permutation. Now, consider the path $(p \circ p')^n$. This path creates a cycle in C^{thr} starting from (r,i) back to (r,i) and passing through (r',j); obviously, this cycle contains a path from (r',j) to (r,i). The second part of the lemma follows trivially from the first part. ■

Let $r = (s,l,a)$ be a state in $\overline{B}$ and C be the *scc* of $\overline{B}$ that contains r. We define the equivalence relation $\overset{*}{\sim}_r$ on I as follows:

$$i \overset{*}{\sim}_r j \quad \text{if} \quad (r,i) \text{ and } (r,j) \text{ are in the same component of } C^{\mathrm{thr}}.$$

It is easy to see that a class of the partition $\overset{*}{\sim}_r$ identifies a unique component of C^{thr}, and every component of C^{thr} is identified by a class of $\overset{*}{\sim}_r$. Thus, we will use these partitions to represent the *scc*s of C^{thr}. A class of $\overset{*}{\sim}_r$ is called weakly fair if the corresponding *scc* of C^{thr} is weakly fair. Note that the tracked process l in r always forms a class of size 1.

Suppose that if r and r' are nodes in the same *scc* in $\overline{B}$. The partitions of $\overset{*}{\sim}_r$ and $\overset{*}{\sim}_{r'}$ are not equal in most cases, but fortunately one can be obtained from the other by a permutation belonging to G. This problem motivates the use of a common referential base. A possible nominee for this is the initial state s_{pr} of $\overline{B}$.

Assume that we have explored $\overline{B}$ in a depth first manner starting from the initial state, until all reachable states have been visited. Let T be the resulting depth first spanning tree. Now, for each state $u \in \overline{B}$, let π_u denote the product of the permutations on the unique $s_{\mathrm{pr}} \to u$ path in T. Now, for each state r in $\overline{B}$, we define an equivalence relation $\sim_r$ on I as follows.

$$i \sim_r j \quad \text{if} \quad \pi_r^{-1}(i) \overset{*}{\sim}_r \pi_r^{-1}(j).$$

LEMMA 2 *If r, r' are nodes in the same scc C of $\overline{B}$ then $\sim_r = \sim_{r'}$.*

Proof: To prove the lemma, it is enough if we show that, for every i,j, $i \sim_r j$ implies $i \sim_{r'} j$ and vice versa. We show this by proving that, for every i, $(r, \pi_r^{-1}(i))$ and $(r', \pi_{r'}^{-1}(i))$ are in the same *scc* in C^{thr}. This will automatically imply the following: for every i and j, if $i \sim_r j$, i.e., there is a path in C^{thr} from $(r, \pi_r^{-1}(i))$ to $(r, \pi_r^{-1}(j))$, then there is also a path from $(r', \pi_{r'}^{-1}(i))$ to $(r', \pi_{r'}^{-1}(j))$ and hence $i \sim_{r'} j$.

To show that $(r, \pi_r^{-1}(i))$ and $(r', \pi_{r'}^{-1}(i))$ are in the same *scc* in C^{thr}, we take the following approach. Let u be the root of the *scc* C, i.e., u is the first node in C that was visited during the depth-first search that induced the forest F. Let T be the tree in F that contains u and r, and let the initial state s_{pr} be the root of T. It can be shown that the unique path in T from s_{pr} to r passes through u (see [7]). Hence, there exists a path in C^{thr} from $(u, \pi_u^{-1}(i))$ to $(r, \pi_r^{-1}(i))$. Hence, by Lemma 1, we see that these two nodes are in the same *scc* in C^{thr}. By a similar argument, we see that $(u, \pi_u^{-1}(i))$ and $(r', \pi_{r'}^{-1}(i))$ are in the same *scc*. This shows that $(r, \pi_r^{-1}(i))$ and $(r', \pi_{r'}^{-1}(i))$ are in the same *scc*. ■

Intuitively, $i \sim_r j$ indicates that the threads of (s_{pr}, i) and (s_{pr}, j) enter the same *scc* of the threaded graph after they passed r. To illustrate these concepts consider the subgraph of the simplified Resource Controller example depicted in Figure 4 below. The tree edges are denoted by boldface arrows.

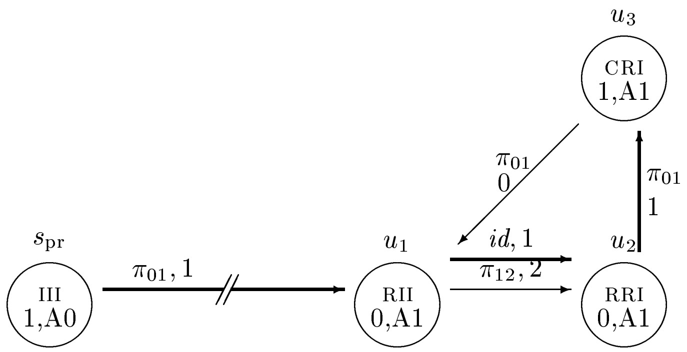

Figure 4. A strongly connected subgraph of the product graph $\overline{B}$

The nodes u_1, u_2 and u_3 form a strongly connected subgraph. In u_1 we have $1 \overset{*}{\sim}_{u_1} 2$ because $(u_2, 2)$ is an immediate successor of both $(u_1, 1)$ and $(u_1, 2)$ in the threaded graph. Hence, $\overset{*}{\sim}_{u_1}$ is $\boxed{0}\boxed{12}$. Similarly, $\overset{*}{\sim}_{u_2}$ is $\boxed{0}\boxed{12}$, but $\overset{*}{\sim}_{u_3}$ is $\boxed{1}\boxed{02}$. Using that $\pi_{u_1} = \pi_{01}, \pi_{u_2} = \pi_{01}$ and $\pi_{u_2} = id$ we compute $\sim_{u_1} = \sim_{u_2} = \sim_{u_3} = \boxed{1}\boxed{02}$.

Returning to the general argument, we show how to compute $\sim_r$ by exploring $\overline{B}$ using depth first search. For each edge $e = u \xrightarrow{\pi, i} v$ in $\overline{B}$, let π_e denote the permutation $\pi_u \circ \pi \circ \pi_v^{-1}$. Note that if e is an edge of T then π_e is the identity permutation. The permutation π_e satisfies the following property.

Claim 1 *If e is an edge in the scc C containing r, then $\pi_e(i) = j$ implies $i \sim_r j$.*

Proof: Since T is a depth first search tree, it enters an *scc* of the graph in a unique vertex. According to Lemma 2, $\sim_r = \sim_{r'}$ for all r, r' in the same *scc*, therefore, we may assume that r is the root of the *scc* that contains e. Hence, there are paths p_1 and p_2 on T from r to both endpoints u, v of e. Let π_1, π_2 be the products of all the permutations on the paths p_1 and p_2 respectively. It should be easy to see that $\pi_1 = \pi_r^{-1} \circ \pi_u$ and $\pi_2 = \pi_r^{-1} \circ \pi_v$.

Now, to demonstrate that $i \sim_r j$, it is enough if we show that there is a path in $C^{\rm thr}$ from $(r, \pi_r^{-1}(j))$ to $(r, \pi_r^{-1}(i))$. By substituting $\pi_u \circ \pi \circ \pi_v^{-1}(i)$ for j and replacing $\pi_r^{-1} \circ \pi_u$ by π_1, we get $\pi_r^{-1}(j) = \pi_1 \circ \pi \circ \pi_v^{-1}(i)$. Let π'' denote the permutation $\pi_1 \circ \pi \circ \pi_2^{-1}$. Since $\pi_v = \pi_r \circ \pi_2$, it follows that $\pi_v^{-1} = \pi_2^{-1} \circ \pi_r^{-1}$. Hence $\pi_r^{-1}(j) = \pi'' \circ \pi_r^{-1}(i)$.

Since r and v are in the same *scc*, there exists a path p' from v to r; now, the path $p_2 \circ p'$ is a cycle and there exists an $n > 0$ such that $(p_2 \circ p')^n$ is also a cycle and the product of all the permutations on this cycle is the identity permutation. This big cycle can be written as $p_2 \circ p'_2$ where p'_2 is the path $p' \circ (p_2 \circ p')^{n-1}$. Now it should be obvious that the product of all the permutations on p'_2 equals π_2^{-1}.

Finally, consider the cycle $p_1 \circ e \circ p'_2$ in C. The product of all permutations on this cycle equals $\pi_1 \circ \pi \circ \pi_2^{-1}$, which is π''. This cycle creates a path in $C^{\rm thr}$ from $(r, \pi'' \circ \pi_r^{-1}(i))$ to $(r, \pi_r^{-1}(i))$. Since $\pi_r^{-1}(j) = \pi'' \circ \pi_r^{-1}(i)$, it follows that there is a path in $C^{\rm thr}$ from $(r, \pi_r^{-1}(j))$ to $(r, \pi_r^{-1}(i))$. ■

Let ρ be a permutation on I. We define the *orbit relation of* ρ to be the reflexive, transitive closure of the binary relation $\{(i, \rho(i)) \ : \ i \in I\}$. Obviously, the orbit relation of ρ is an equivalence relation; we define the *orbit partition of* ρ to be the partition induced by the orbit relation of ρ. Now Proposition 1 can be reformulated as: *If e is an edge in the scc of r, then the orbit partition of π_e is smaller than or equal to (i.e. a subset of) $\sim_r$.* The following stronger result characterizes $\sim_r$.

THEOREM 2 *$\sim_r$ is the join of all orbit partitions of π_e, where e ranges over the edges of the strongly connected component of r, i.e. it is the smallest equivalence relation containing the orbit relation π_e, for each edge e in the scc containing r.*

Proof: We need to show that $i \sim_r j$ implies that there are processes $i_0, i_1, \ldots, i_n$ and edges $e_0, \ldots, e_{n-1}$ in the *scc* of r such that $\pi_{e_k}(i_{k+1}) = i_k$ for $k = 0, \ldots, n-1$ and $i_0 = i, i_n = j$. This would indicate that (i_k, i_{k+1}) is in the orbit relation of π_{e_k}; hence, (i, j) is in the smallest equivalence relation containing the orbit relations of all the edges in the *scc*.

The relationship $i \sim_r j$ implies $\pi_r^{-1}(i) \overset{*}{\sim}_r \pi_r^{-1}(j)$. Therefore, $(r, \pi_r^{-1}(i))$ and $(r, \pi_r^{-1}(j))$ are in the same *scc* of the threaded graph. Let $(r, \pi_r^{-1}(i)) = (r_0, l_0) \overset{e_0}{\to} (r_1, l_1) \overset{e_1}{\to} \ldots \overset{e_{n-1}}{\to} (r_n, l_n) = (r, \pi_r^{-1}(j))$ be a path that connects them. Take i_k to be $\pi_{r_k}(l_k)$. Certainly, $i_0 = i, i_n = j$. Let π'_k be the permutation labeling the edge e_k (note that π'_k is different from π_{e_k}). Now we have $\pi_{e_k}(i_{k+1}) = \pi_{r_k} \circ \pi'_k \circ \pi_{r_{k+1}}^{-1}(i_{k+1})$. Since, $i_{k+1} = \pi_{r_{k+1}}(l_{k+1})$, i.e. $l_{k+1} = \pi_{r_{k+1}}^{-1}(i_{k+1})$, we get $\pi_{e_k}(i_{k+1}) = \pi_{r_k} \circ \pi'_k(l_{k+1})$. Substituting $l_k = \pi'_k(l_{k+1})$ (from the definition of the threaded graph), we get $\pi_{e_k}(i_{k+1}) = \pi_{r_k}(l_k) = i_k$. This completes the proof. ■

In our illustrating example in Figure 4 we can compute π_e for the 4 edges in the component $\{u_1, u_2, u_3\}$. $e_1 = u_1 \overset{id,1}{\longrightarrow} u_2$ and $e_2 = u_2 \overset{\pi_{01},1}{\longrightarrow} u_3$ are tree edges so $\pi_{e_1} = \pi_{e_2} = id$. Examining $e_3 = u_3 \overset{\pi_{01},0}{\longrightarrow} u_1$ we find that $\pi_{e_3} = \pi_{u_3} \circ \pi_{01} \circ \pi_{u_1}^{-1} = id \circ \pi_{01} \circ \pi_{01}^{-1} = id$. In a similar way, we compute for $e_4 = u_1 \overset{\pi_{12},2}{\longrightarrow} u_2$ that $\pi_{e_4} = \pi_{u_1} \circ \pi_{12} \circ \pi_{u_2}^{-1} = \pi_{02}$. The orbit partition of π_{e_1}, π_{e_2} and π_{e_3} is $\boxed{0}\boxed{1}\boxed{2}$

while that of π_{e_4} is $\boxed{1}\boxed{02}$. The join of these partitions is $\boxed{1}\boxed{02}$ that coincides with $\sim_{u_1} = \sim_{u_2} = \sim_{u_3}$.

The next theorem follows immediately from Definition 2 and is a necessary and sufficient condition for checking if a class of $\sim_r$ is weakly fair. Let C be the *scc* of r in $\overline{B}$.

THEOREM 3 *A class K of the partition $\sim_r$ is weakly fair if and only if there is an $i \in K$ and $u \in C$ such that process $\pi_u^{-1}(i)$ is disabled in u or it is executed (that is, there is an edge $u \stackrel{\pi,\pi_u^{-1}(i)}{\longrightarrow} v$ in C).*

We have gathered together all the necessary tools to present the on-the-fly algorithm.

4.2. The Algorithm

Our algorithm is a modification of the strongly connected component computation using depth first search presented e.g. in [2].

For each vertex $u = (s, l, a)$ of the product graph $\overline{B}$, we maintain the following information.

- u.`dfnum` is a unique id (or *depth first number*) of the node, used for the strongly connected component computation.q
- u.`lowlink` is the id of a reachable node lower than u itself.
- u.`onstack` is a flag indicating that u is still on stack.
- u.`perm` is the vector π_u as defined in the previous subsection.
- u.`partition` is an approximation of $\sim_u$.
- u.`status` is a vector of flags that indicate which partition classes are known to be weakly fair.
- u.`final` is a flag that indicates if u is in an *scc* that contains a final automaton state. This information is propagated down on the depth first tree.

The variables u.`dfnum`, u.`lowlink` and u.`onstack` are maintained as in the algorithm given in [2]. u.`perm` is set when u is created, while u.`partition`, u.`status` and u.`final` are updated every time an edge to a successor state of u is explored.

On-tye-fly Model Checking
M1. Set the depth-first-counter to zero.
M2. Set $u = s_{\mathrm{pr}}$ (the initial state of $\overline{B}$).
    Set u.`perm` to be the identity permutation.
    Conduct **DF-Search**(u).
M3. Exit with a *No* answer.

DF-Search(u) *(Note that $u = (s, l, a)$.)*

1. Push u on to the stack, set u.`onstack`.
   Set u.`dfnum` and u.`lowlink` to the depth-first-counter.
   Increase the depth-first-counter.
2. Initialize u.`partition` to be the identity partition.
3. Initialize u.`status` with the information on disabled processes stored in the AQS state s.
   Set u.`final` if a is a final automaton state.
4. (Idle command. Later modification will use it.)
5. For each AQS edge $e = s \xrightarrow{\pi,i} t$ do
6. $(l', \rho) =$ **FindEquiv**$(t, \pi^{-1}(l))$.
7.   For each automaton transition $a \rightarrow a'$ that is enabled in s do
8.   Set $v = (t, l', a')$.
9.   If v is already constructed and v.`onstack` is set then do
10.   Compute the join of u.`partition` and the orbit partition of π_e and store it in u.`partition`.
                Update u.`status` using that process i was executed.
                Set u.`lowlink` to the minimum of u.`lowlink` and v.`lowlink`.
11.   If v is not constructed yet then do
12.   Set v.`perm` $=$ u.`perm` $\circ\, \pi \circ \rho$.
13.   Conduct **DF-Search**(v).
14.   If v.`onstack` is still set then do
15.   Compute the join of u.`partition` and v.`partition` and store it in u.`partition`.
                    Combine v.`status` to u.`status`.
                    Update u.`status` using that process i was executed.
                    Set u.`lowlink` to the minimum of u.`lowlink` and v.`lowlink`.
                    Set u.`final` if v.`final` is set.
16.   If all the partition classes are weakly fair (use u.`status`) and u.`final` is set then exit with *Yes* answer.
17. If u.`dfnum` $=$ u.`lowlink` then do
18.   Pop all elements above u (inclusive) from the stack and mark the popped vertices off–stack.

In command M2, we construct the initial state of $\overline{B}$ and invoke DF-search on this vertex. In DF-search, the algorithm may exit with a *Yes* answer if a fair and final *scc* is discovered. If none of the recursively called invocations of DF-search exit with a *Yes* answer, the algorithm outputs a *No* answer and exits in command M3.

DF-search works as follows. Commands 1–3 initialize variables appropriately. The two "for" loops in commands 5 and 7 generate the successors of the $\overline{B}$ state u.

In command 6, we invoke the routine **FindEquiv** to find the equivalent representative of $(t, \pi^{-1}(l))$ in $\overline{S}_{\text{aqsi}}$. The returned (l', ρ) has the property that $(t, \pi^{-1}(l))$ and (t, l') are equivalent under the permutation ρ and the later belongs to $\overline{S}_{\text{aqsi}}$. This equivalence test becomes very easy if we store the state symmetry partition $\overset{*}{\approx}_t$ of the underlying $\overline{M}$ state t. For definition and details consult Subsect. 4.4 below. (It is to be noted that we need this equivalence checking since we are constructing

$\overline{B}$ to be the product of $\overline{M \times I}$ and the automaton A, and we are doing this using $\overline{M}$ and A. However, if we want to construct $\overline{B}$ to be $\overline{M} \times I \times A$, as in [10], then we do not need this equivalence checking, and in this case we may have more states in the resulting $\overline{B}$. For this, command 6 needs to be changed to set l' to $\pi^{-1}(l)$ and to set ρ to the identity permutation.)

In command 9, we check if $u \to v$ is a non-tree edge and u, v are in the same *scc*; this is accomplished by testing that v has already been constructed (i.e. visited) and v is still on stack. If the test is passed, the orbit partition of π_e is joined with u.`partition` and the result is stored in u.`partition`. Commands 12 through 15 are executed if the edge $u \to v$ is a tree edge, i.e., v is constructed (and hence visited) for the first time. In command 12, v.`perm` is set; in command 13, DF-search is invoked on v. If v and u are in the same *scc* (indicated by the condition in command 14) then the partitions are joined, u.`status` is updated and other updates are carried out. After processing the edge $u \to v$, in command 16, we check if the partially explored *scc* containing u is weakly fair and has a final state; if so the algorithm exits with a *Yes* answer indicating a fair computation accepted by the automaton A is found. In command 18, after detecting the *scc*, we pop all the states of the *scc* from the stack.

THEOREM 4 *The algorithm described above outputs* Yes *if and only if the original program has a weakly fair computation that is accepted by A.*

Proof: The proof relies on the theory developed in Subsect. 4.1 and on the correctness of the strongly connected component algorithm of [2].

Suppose that the algorithm halts with a *Yes* answer. At termination the stack contains a strongly connected subgraph of the product graph. That subgraph is weakly fair with respect to all processes because u.`partition`'s classes are all weakly fair and u.`partition` is an approximation of (it is smaller than) $\sim_u$ yielding that $\sim_u$ is weakly fair itself. This subgraph defines a fair run of the program that is accepted by the automaton.

If the program terminates with a *No* answer then it explored the entire $\overline{B}$ graph and found that none of the strongly connected components are satisfactory (they either lack a final automaton state or are not fair with respect to one of the processes). Hence, the original program had no fair run that is accepted by the automaton.

■

To analyze to complexity of the algorithm, we use the following notation. If K is a graph or an automaton, then $|K|$ denotes the number of nodes and $E(K)$ the number of edges or transitions. Execution of commands M1, M2 and M3 together takes $O(|I|)$ time. Commands 1–4, 12–15, 17 and 18 are executed once for each $\overline{B}$ node. The number of $\overline{B}$ nodes is at most $|\overline{M \times I}| \cdot |A|$ and a single execution of the above listed commands takes $O(|I|)$ time. Thus these commands contribute $O(|\overline{M \times I}| \cdot |A| \cdot |I|)$ to the over all complexity.

Now consider the execution of the commands 8–11 and 16. Every time these commands are executed, the triple $(e, l, a \to a')$ has a different value. Hence these commands are executed no more than $E(\overline{M}) \cdot |I| \cdot E(A)$ times. Each execution of commands 8 and 16 takes $O(|I|)$ time. We have implemented an algorithm for joining the two partitions mentioned in command 10; this algorithm uses graph data structures and has complexity $O(|I|)$. Commands 9 and 11 require checking if the node v has already been constructed. In our implementation, with each $\overline{M}$ state s, we maintain a linked list of all $\overline{B}$ states whose first component is s; obviously, the length of this list is at most $|I|\cdot|A|$ and searching this list takes $O(|I|\cdot|A|)$. Thus, we see that execution of commands 8–11 and 16 contributes $O(E(\overline{M})\cdot|I|^2\cdot E(A)\cdot|A|)$ to the over all complexity.

Finally consider command 6. Each time it is executed, the triple (e,l,a) has a different value. Hence, the number of times it is executed is bounded by $E(\overline{M})\cdot|I|\cdot|A|$. Thus command 6 contributes $O(E(\overline{M})\cdot|I|\cdot|A|\cdot x)$ to the over all complexity, where x denotes the complexity of a single execution of command 6.

From the above analysis, we see that the over all complexity of the algorithm is $O(E(\overline{M})\cdot|I|\cdot|A|\cdot(E(A)\cdot|I|+x))$.

Note that, in the most general case, checking for state symmetry in command 6 can have exponential complexity, and hence the value of x can be exponential. However, in our implementation we only checked for restricted forms of symmetries, namely for those symmetries that swap two processes, and we also used the state symmetry partitions generated during the construction of $\overline{M}$ (see next subsections). This implementation has complexity $O(|I|)$. Hence, for this implementation, the over all complexity of the algorithm is $O(E(\overline{M})\cdot|I|^2\cdot|A|\cdot E(A))$.

It is to be noted that if we do not invoke the equivalence check in command 6, as explained earlier, then we will be constructing $\overline{B}$ as $\overline{M}\times I\times A$, and exploring it. In this case the over all complexity will also be $O(E(\overline{M})\cdot|I|^2\cdot|A|\cdot E(A))$.

4.3. State Symmetry in $\overline{\boldsymbol{B}}$, Partition Initialization, Parallel Edges

This subsection is devoted to showing that the equivalence relation $\sim_u$ (defined in Subsect. 4.1, computed in u.`partition`) can be computed more efficiently than presented in the basic algorithm. Improvements can be achieved by sophisticated initialization of $\sim_u$ and by considering only a portion of the edges in command 5.

Let $u=(s,l,a)$ be a vertex of the product graph $\overline{B}$. Processes i and j are called *u–equivalent*, denoted by $i \overset{*}{\approx}_u j$, if there is a permutation $\rho\in G$ such that $\rho(i)=j$ and $\rho(u)=u$. $\overset{*}{\approx}_u$ is called the *local state symmetry partition* at u. Intuitively, $i \overset{*}{\approx}_u j$ shows that processes i and j are interchangeable in state u. Let $u \xrightarrow{\pi,l} v$ be an edge of $\overline{B}$. Then $u \xrightarrow{\rho\circ\pi,\rho(l)} v$ is also an edge yielding that $(v,\pi^{-1}(i))$ is a successor of both nodes (u,i) and (u,j) in the threaded graph $\overline{B}^{\mathrm{thr}}$. From lemma 4, it follows that (u,i) and (u,j) are in the same *scc* of $\overline{B}^{\mathrm{thr}}$. Hence, $i \overset{*}{\sim}_u j$. This proves the next lemma.

LEMMA 3 *The partition $\overset{*}{\approx}_u$ is smaller than $\overset{*}{\sim}_u$ for every state u of the product graph $\overline{B}$.*

This fact allows an improvement to the algorithm. First we need to project down $\overset{*}{\approx}_u$ to the common referential base. We define $i \approx_u j$ if $\pi_u^{-1}(i) \overset{*}{\approx}_u \pi_u^{-1}(j)$. Command 2 in DF-Search can be changed to

$2'$. Initialize u.`partition` to be $\approx_u$.

Now we describe how state symmetry can be used to remove parallel edges. Let $e = u \xrightarrow{\pi,l} v$ and $e' = u \xrightarrow{\pi',l'} v$ be edges in $\overline{B}$. We say that e and e' are *parallel* if there is a permutation $\rho \in G$ such that $\rho(u) = u, \rho \circ \pi = \pi', \rho(l) = l'$. Surely, being parallel is an equivalence relation on the edges. Let $\overline{R}^r_{\mathrm{pr}}$ be a set of *representative edges* that contains at least one edge from each parallel class. When the partitions are initialized as presented in command $2'$, the orbit partition of $\pi_{e'}$ does not give any new information after the orbit partition of π_e has been considered. It is reflected in the next lemma.

LEMMA 4 *If r is in an scc of $\overline{B}$ then $\sim_r$ is the smallest partition that contains $\approx_v$ (the initial value of v.`partition`) for every v in the scc of r as well as the orbit partition of π_e for every edge $e \in \overline{R}^r_{\mathrm{pr}}$.*

Proof: Let $e = u \xrightarrow{\pi,l} v$ be an edge of the *scc* of r whose representative is $e' = u \xrightarrow{\rho\circ\pi,\rho(l)} v$ in $\overline{R}^r_{\mathrm{pr}}$. Suppose $\pi_e(i) = j$. We show that (i,j) is contained in the join of the orbit partition of $\pi_{e'}$ and $\approx_u$. Using the definition of π_e, we have $\pi_u \circ \pi \circ \pi_v^{-1}(i) = j$ so $\pi_u \circ (\rho^{n-1} \circ \pi_u^{-1} \circ \pi_u \circ \rho) \circ \pi \circ \pi_v^{-1}(i) = j$ where ρ^n is the identity permutation. Let $l = \pi_u \circ \rho \circ \pi \circ \pi_v^{-1}(i)$. Now $\pi_{e'}(i) = l$ and $\pi_u \circ \rho^{n-1} \circ \pi_u^{-1}(l) = j$. The later implies $\rho^{n-1} \circ \pi_u^{-1}(l) = \pi_u^{-1}(j)$. Hence $\pi_u^{-1}(l) \overset{*}{\approx}_u \pi_u^{-1}(j)$ giving $l \approx_u j$. Summing up, $\pi_e(i) = j$ implies that there is an l such that (i,l) is in the orbit partition of e' while (l,j) is in $\approx_u$. Therefore, the orbit partition of e is contained in the join of $\approx_u$ and the orbit partition of e'.

This proves that the smallest partition that contains $\approx_v$ for every v in the *scc* of r as well as the orbit partition of π_e for every edge $e \in \overline{R}^r_{\mathrm{pr}}$ actually contains the orbit partition of all edges in the *scc* of r. Using Theorem 2 we conclude that it contains $\sim_r$ as well. The other direction follows from Proposition 1 and Lemma 3.

■

These ideas can be applied as follows. From each class of $\approx_u$ pick a representative process and call it the *leader* of that class. Put $\overline{R}^r_{\mathrm{pr}} = \{u \xrightarrow{\pi,l} v \in \overline{R}_{\mathrm{pr}} : l \text{ is a leader}\}$. Since every edge is parallel to one that was caused by a leader process, this $\overline{R}^r_{\mathrm{pr}}$ is a satisfactory set of representative edges. We introduce the new vector u.`leader` of flags. The next improvement in the algorithm is the introduction of command 4 and the modification of command 5.

4. Initialize u.`leader`.

5'. For each AQS edge $e = s \xrightarrow{\pi,i} v$ if u.`leader[`i`]` is set do

4.4. State Symmetry in $\overline{M}$

In this subsection, we show how state symmetry can also be used to reduce the number of edges of $\overline{M}$ that are generated and stored. Let μ be a state symmetry of s, i.e., $\mu(s) = s$. Now, if $s \xrightarrow{\pi,j} t$ is an edge of $\overline{M}$, then $s \xrightarrow{\mu\circ\pi,\mu(j)} t$ is also an edge of $\overline{M}$. This simple observation shows that we need not store both $s \xrightarrow{\pi,j} t$ and $s \xrightarrow{\mu\circ\pi,\mu(j)} t$ provided that μ can be efficiently computed for s.

The above idea is employed by first introducing, for each AQS state s, an equivalence (called *state equivalence*) relation $\overset{*}{\approx}_s$ among process indices defined as follows.

$$i \overset{*}{\approx}_s j \text{ if there is a } \mu \in G \text{ with } \mu(s) = s, \mu(i) = j.$$

Note that in Subsect. 4.3 we introduced local state symmetry partition for $\overline{B}$ states; the local state symmetry partition of a $\overline{B}$ state $u = (s, l, a)$ is denoted by $\overset{*}{\approx}_u$. (Note that the subscript distinguishes the two notations.) We recall that $i \overset{*}{\approx}_u j$ if there is a $\mu \in G$ with $\mu(s) = s, \mu(l) = l$ and $\mu(i) = j$. Observe that $i \overset{*}{\approx}_u j$ implies $i \overset{*}{\approx}_s j$. Therefore, the local state symmetry partition for a $\overline{B}$ state is usually smaller than the state equivalence relation of the underlying $\overline{M}$ state. This is caused by the fact that a state symmetry permutation of a $\overline{B}$ state fixes not only the underlying $\overline{M}$ state but the tracked processes as well. Nevertheless, having $\overset{*}{\approx}_s$ in hand, $\overset{*}{\approx}_u$ can be easily computed.

Unfortunately, the problem of computing $\overset{*}{\approx}_s$ can be a difficult task since it is equivalent to the graph isomorphism problem. (With any given graph H, we can associate a program P and a state s of P in a straightforward way. P has as many processes as many nodes H has; for each pair of processes v, w, P has a variable $a[v, w]$ indexed by v and w; $a[v, w]$ takes value 1 if $v \to w$ is an edge of H, otherwise it is 0. Now $v \overset{*}{\approx}_s w$ exactly when H has an automorphism that maps node v to w. That later problem is equivalent to the graph isomorphism problem.) In many important special cases the symmetry detection can be performed efficiently. In general, however, only approximating solutions are available.

Let $s \xrightarrow{\pi,j} t$ be an edge of $\overline{M}$, $\mu(s) = s$. Then $s \xrightarrow{\mu\circ\pi,\mu(j)} t$ is an edge as well. This simple observation shows that we need not store both $s \xrightarrow{\pi,j} t$ and $s \xrightarrow{\mu\circ\pi,\mu(j)} t$ provided that μ can be efficiently computed for s.

We are ready to present the last improvement to our algorithm. In the construction of $\overline{M}$ (not shown in the algorithm) we do the following modifications. When a new node s is created, we compute $\overset{*}{\approx}_s$. A vector s.`repr` is defined such that, for every index j, it points to a representative of the $\overset{*}{\approx}_s$-class of j. By the construction of the edges of $\overline{M}$, we store only those edges that are caused by a representative process. (So $s \xrightarrow{\pi,j} t$ is stored if $s.\texttt{repr[}j\texttt{]} = j$.)

In our original algorithm commands 5', 12 should be changed to 5" and 12' below, respectively.

5″. For each stored AQS edge $e = s \xrightarrow{\pi,j} t$ and each process i with $s.\mathtt{repr}[i] = j$ and $u.\mathtt{leader}[i]$ is set, compute some $\mu \in G$ with $\mu(s) = s$ and $\mu(j) = i$, and then do

12′ Set $v.\mathtt{perm} = u.\mathtt{perm} \circ \mu \circ \pi \circ \rho$.

As it has been pointed out earlier, in general, computing $\overset{*}{\approx}_s$ is computationally hard. However, we have implemented a method where we only look for state symmetries, i.e. permutations, which only interchange two process indices; computing all such symmetries and the corresponding equivalence relation $\overset{*}{\approx}_s$ can be done efficiently. The same approach is employed in computing the state symmetries in $\overline{B}$. Also note that in step 5" of the algorithm, it is enough if we find one permutation μ satisfying the given property; we do not have to compute all such permutations. Since, for the case of state symmetry, we are restricting the class of permutations to be those that only interchange two process indices, step 5" can also be implemented efficiently.

Concluding this section, we illustrate the concept of state symmetry and redundant edges by showing $\overline{M}$ of our simplified Resource Controller after deleting all redundant edges.

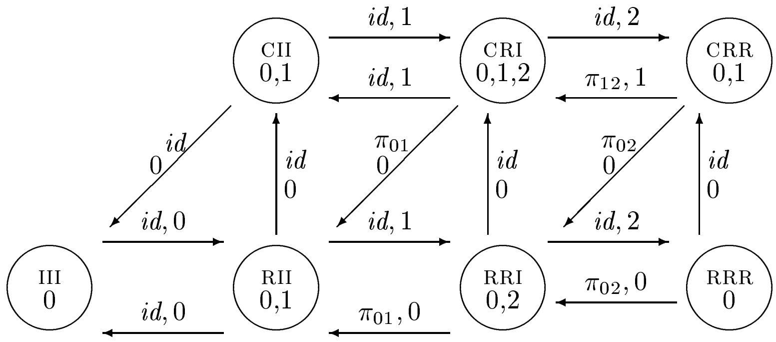

Figure 5. The AQS $\overline{M}$ without redundant edges

Consider Figure 5. The top row in each circle shows the local states of the three processes while the bottom row lists the representative processes. The compressed $\overline{M}$ has 16 edges while the original had 27.

5. Implementation

We have developed a prototype of the on-the-fly model checker implementing the above presented algorithm. We have used efficient approximation techniques to check equivalence of two states when generating the AQS and also in the main algorithm where we had to check for equivalence of $\overline{B}$ states. For the case of complete symmetries, as in the Resource Controller and Readers/Writers examples, this approximation algorithm will indicate two states to be equivalent whenever they are equivalent. For other types of symmetry, these approximation methods may sometime indicate two states to be inequivalent although they are equivalent. In such cases, we may not get maximum possible reduction in the size of the state space; however, our algorithm will still correctly indicate if the concurrent system satisfies the correctness specification or not.

We used the implemented system to check for the correctness of the *Resource Controller* example, the *Readers Writers* example, and the *Ethernet Protocol* with various number of users.

We contrasted our new system with the old model checker that implements the results presented in [10] on the Resource Controller example. We checked many properties including the liveness property that every user process that requests a resource will eventually access the resource, and the mutual exclusion property. Dramatic improvement was detected in all performance measures as indicated in Table 1 and 2 below. The product graphs constructed by the old and new model checker are referred to as $\overline{B}_0$ and $\overline{B}$ respectively. Each statistic is given as a/b where a and b are the numbers corresponding to the old and new model checker respectively.

Table 1. Statistics for checking a liveness property

Eventually Access	10	50	100
AQS states	38 / 38	198 / 198	398 / 398
AQS transitions	235 / 107	6175 / 1567	24850 / 5642
AQS const. time (sec)	0 / 0	16 / 5	149 / 39
Explored $\overline{B}_0/\overline{B}$ states	385 / 91	9943 / 91	39893 / 91
$\overline{B}_0/\overline{B}$ const. time (sec)	0 / 0	19 / 0	330 / 0
Total memory used (kbyte)	31 / 13	1219 / 216	6878 / 830
Total CPU time used (sec)	0 / 0	37 / 6	481 / 42

Table 2. Statistics for checking a safety property

Mutual Exclusion Prop.	10	50	100
AQS states	38 / 38	198 / 198	398 / 398
AQS transitions	235 / 107	6175 / 1567	24850 / 5642
AQS const. time (sec)	0 / 0	16 / 5	149 / 39
Explored $\overline{B}_0/\overline{B}$ states	38 / 38	198 / 198	398 / 398
$\overline{B}_0/\overline{B}$ const. time (sec)	0 / 0	1 / 0	7 / 2
Total memory used (kbyte)	11 / 10	481 / 221	2903 / 857
Total CPU time used (sec)	0 / 0	18 / 6	158 / 44

The liveness property we checked is not satisfied by the Resource Controller. Both model checkers found a fair incorrect computation. From the table, we see that the number of AQS states are the same, while the number of AQS transitions is much smaller in the new model checker due to the use of state symmetry. The number of product states explored in the on-the-fly system is much smaller since it terminated early. On the other hand the original model checker constructed the whole $\overline{B}_0$ before checking for an incorrect fair computation.

For the mutual exclusion property, both model checkers indicated that the Resource Controller satisfies that property. In this case, early termination does not come into effect. Furthermore, since we do not track any process (mutual exclusion is a global property), the number of states explored in $\overline{B}_0$ and $\overline{B}$ are the same. However, the number of transitions is much smaller in $\overline{M}$ as well as in $\overline{B}$ due to the effect of state symmetry. The over all CPU time and the memory usage are substantially smaller for the new model checker.

6. Conclusions

In this paper, we have presented an on-the-fly model checking system that exploits symmetry (between states as well as inside a state) and checks for correctness under fairness.

Symmetry based reduction has been shown to be a powerful tool for reducing the size of the state space in a number of contexts. For example, such techniques have been employed in the Petri-net community [14, 15] to reduce the size of the state space explored. Such techniques have also been used in protocol verification [1, 16] and in hardware verification [13] and in temporal logic model checking [5, 9, 10]. There have been on-the-fly model checking techniques [3, 11, 12, 17] that employ traditional state enumeration methods. Some of them [11, 12, 17] also use other types of state reduction techniques. To the best of our knowledge, ours is the first approach that performs on-the-fly model checking under fairness for the full range of temporal properties and that exploits symmetry.

As part of future work, we plan to explore techniques to automatically detect symmetries and integrate these techniques with the model checker. Also, algorithms for checking equivalence of global states under other types of symmetry need to be further explored.

References

1. Aggarwal S., Kurshan R. P., Sabnani K. K.: "A Calculus for Protocol Specification and Validation", *Protocol Specification, Testing and Verification III*, H. Ruden, C. West (ed's), pp19–34, North-Holland 1983.
2. Aho, A. V., Hopcroft, J., Ullman, J. D.: "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1974.
3. Bhat, G., Cleaveland, R., Grumberg, O.: "Efficient On-the-Fly Modelchecking for CTL", Proceedings of the *International Conference on Logic in Computer Science*, San Diego, California, 1995.
4. Clarke, E. M., Emerson, E. A., Sistla, A. P.: "Automatic Verification of Finite State Concurrent Programs Using Temporal Logic: A Practical Approach", Proceedings of the *ACM Symposium on Principles of Programming Languages*, January 1983, Austin, Texas, Also appeared in *ACM TOPLAS*, April 1986.
5. Clarke, E. M., Filkorn, T., Jha, S.: "Exploiting Symmetry in Temporal Logic Model Checking", Proceedings of the *5th International Conference on Computer Aided Verification*, Crete, Greece, June 1993.

6. Cleaveland, R.: "Analyzing Concurrent Systems using the Concurrency Workbench, Functional Programming, Concurrency, Simulation, and Automated Reasoning", pp129–144, LNCS **693** Springer-Verlag, 1993.
7. Cormen T. H., Leiserson C. E., Rivest R. L.: "Introduction to Algorithms", The MIT Press 1990.
8. Dams, D., Grumberg, O., Gerth, R.: "Generation of Reduced Models for checking fragments of CTL", CAV93. LNCS **697** Springer-Verlag, 1993.
9. Emerson, E. A., Sistla, A. P.: "Symmetry and Model Checking", Proceedings of the *5th International Conference on Computer Aided Verification*, Crete, Greece, June 1993.
10. Emerson, E. A., Sistla, A. P.: "Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata-theoretic Approach", Proceedings of the *7th International Conference on Computer Aided Verification*, Leige, Belgium, July 1995.
11. Godefroid, P.: "Partial-Order Methods for the Verification of Concurrent Systems", LNCS **1032** Springer-Verlag, 1996.
12. Holzmann, G.J., Peled, D.: "The State of SPIN", 8th Intl. Proceedings of the *Conference on Computer Aided Verification*, July 1996.
13. Ip, C. N., Dill, D. L.: "Better Verification through Symmetry", *Intl. Symposium on Computer Hardware Description Languages and their Application*, April 1993. Also in *Formal Methods in System Design* **9** 1/2, pp41–75, 1996.
14. Jensen, K.: "Colored Petri Nets: Basic Concepts, Analysis Methods, and Practical Use", vol. 2: Analysis Methods, EATCS Monographs, Springer-Verlag, 1994.
15. Jensen, K., Rozenberg, G. (eds.): "High-level Petri Nets: Theory and Application", Springer-Verlag, 1991.
16. Kurshan, R. P.: "Testing Containment of omega-regular Languages", Bell Labs Tech. Report 1121-861010-33 (1986); conference version in R. P. Kurshan, "Reducibility in Analysis of Coordination", pp19–39 LNCS **103** Springer-Verlag, 1987.
17. Kurshan, R. P.: "Computer Aided Verification of Coordinated Processes: The Automata Theoretic Approach", Princeton Univerity Press, Princeton NJ, 1994.