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

V. Gyuris† A. P. Sistla‡

January 10, 1997

1 Introduction

The state explosion problem is one of the major bottlenecks in temporal logic model checking. Symmetry based techniques have been proposed in [CFJ93, ES93] for combating the state explosion problem. In these methods the reachability graph is collapsed by identifying states that are equivalent under symmetry, and model checking is performed on the reduced graph. Although the initial methods of [CFJ93, ES93] 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 [ES95]. This method constructs an *Annotated Quotient Structure* (AQS), computes a product graph and explores the product for existence of fair paths that violate the correctness property.

In this paper, we present an on-the-fly model checking algorithm that checks for correctness under fairness and that exploits symmetry. The algorithm explores the product graph at the same time as it constructs it and terminates early if it finds an incorrect fair computation. Thus, it may terminate early even before it completes the construction of the product graph. Furthermore, it only saves the states in the partially constructed product graph but not the edges. The paper develops additional theory that leads to new novel techniques that make the on-the-fly model-checking feasible. Our algorithm also uses another technique, called *state symmetry* [ES93], for reducing the running time as well as memory usage. The on-the-fly model checking algorithm has been implemented and experimental results indicate substantial improvement in performance compared to the original method.

The fairness that we consider in this paper is the traditional weak fairness (a computation is said to be weakly fair if every process is either infinitely often disabled or is executed infinitely often). The algorithm for fairness, given in [ES95], works as follows. It constructs the AQS $\overline{M}$ and then computes the product $\overline{B}$ of $\overline{M}$ and A where A is the automaton that accepts exactly all the incorrect computations. It explores $\overline{B}$ checking for existence of "subtly fair" and "final" strongly connected components; in order to check if a strongly connected component $\overline{C}$ is subtly fair, it resolves $\overline{C}$ into a threaded graph C^* and then checks if each of the connected components in C^* is "obviously fair", i.e. contains a good node. The nodes of the threaded graph are of the form $(\overline{v}, j)$ where $\overline{v}$ is a node in $\overline{B}$ and j is a process index.

Our on-the-fly algorithm works as follows. It incrementally constructs $\overline{B}$ as and when required, and at the same time runs a modified algorithm for computing the strongly connected components

*This work is partly supported by the NSF grants CCR-9623229, CCR-9633536

†Mathematics, Statistics and Computer Science Department, The University of Illinois at Chicago, USA
‡Electrical Engineering and Computer Science Department, The University of Illinois at Chicago, USA

(scc) using depth first search [HU74]. During the depth first search, with each vertex on the stack, it maintains a partition vector. If $\overline{u}$ is a state in $\overline{B}$ belonging to the scc $\overline{C}$ then the partition vector associated with $\overline{u}$ captures information about the strongly connected components of the threaded graph C^* (intuitively, if the i^{th} and j^{th} components of the partition vector are equal then it indicates that the nodes $(\overline{u}, i)$ and $(\overline{u}, j)$ are in the same scc in C^*). Whenever, an edge to an already visited node in the same scc is explored then the corresponding partition vectors are merged, and whenever a new vertex in the same scc is visited then the corresponding partitions are combined. After each combine operation or merge operation, it is checked if all the components in the partition vector are good (i.e. weakly fair), and if a final automaton state has been encountered in the strongly connected component; if so, the algorithm terminates indicating that an incorrect fair computation has been found.

In addition to the above, our algorithm also uses state symmetry [ES93] to reduce the size of the AQS that is constructed. A permutation $\pi \in Aut\, M$ on process indices is a state symmetry of a representative state $\overline{s}$, if $\pi(\overline{s}) = \overline{s}$. The state symmetry is exploited in different forms in our algorithms. When checking the correctness of the execution of a single process, in [ES93], $\overline{B}$ is constructed as the product $\overline{M} \times I \times A$ where A is the automaton. However, we construct $\overline{B}$ as the product $\overline{M \times I} \times A$; the new product can have potentially fewer states. State symmetry is also used to reduce the number of edges in $\overline{B}$ and $\overline{M}$; especially, parallel edges between the same nodes are eliminated. This reduce memory usage. (for example, in any state $\overline{s}$ of $\overline{M}$, if many processes are in the same local state then it is enough if we execute one of them and store the corresponding transitions).

Our paper is organized as follows. Section 2 contains notation and preliminaries. In section 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 showing 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* and $Sym\, I$ denote the set of all permutations on I. We consider a system $P = ||_{i \in I} K_i$ of processes running parallel. Each process K_i is a set of transitions. We assume that all variables of P are indexed by indices from I that denote the processes that 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 can 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 s'.

2.2 Annotated Quotient Structure, Threaded Graph

For a permutation π, we define the action of π on the set of global states. For a global state s, we define the global state $\pi(s)$ as follows; for every variable $v_{i_1,\ldots,i_k}$, its value in the state s is given to the variable $v_{\pi(i_1),\ldots,\pi(i_k)}$ in the state $\pi(s)$; of course, if $v_{\pi(i_1),\ldots,\pi(i_k)}$ is not a variable then

$\pi(s)$ is undefined. We say that π is an *automorphism* of the indexed global state graph M if the action of π is defined on S, $\pi(s_0) = s_0$ 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 $\pi(s) = t$. M can be compressed using G as follows.

The *annotated quotient structure* (AQS) for M is $\overline{M} = \langle \overline{S}, \overline{R}, s_0 \rangle$ where $\overline{S}$ is a set of *representative states* that contains exactly one state from each equivalence class of S/G, $\overline{R} = \{\overline{s} \xrightarrow{\pi,i} \overline{t} \ : \ \pi \in G, \overline{s}, \overline{t} \in \overline{S}, \overline{s} \xrightarrow{i} \pi(\overline{t}) \in R\}$.

In order to check correctness of a program under fairness, processes need to be traced in the compressed graph $\overline{M}$, that is, $\overline{M}$ needs to be partially unwound. This leads to the notion of a threaded graph. The *threaded graph* M^* corresponding to $\overline{M}$ is $\langle \overline{S} \times I, R^*, s_0 \rangle$ where $R^* = \{(\overline{s}, k) \xrightarrow{i} (\overline{t}, l) \ : \ \overline{s} \xrightarrow{\pi,i} \overline{t} \in \overline{R}$ for some $\pi \in G, l = \pi^{-1}(k)\}$ (for further detail see [ES95]).

2.3 Strongly Connected Subgraphs

An infinite computation of the program determines a strongly connected subgraph of M reachable from the initial state. (Take the set of those states that are visited infinitely often.) The converse of the statement also holds. This shows that studying infinite computations boils down to the study of reachable strongly connected components (*scc* for short) of M.

Let C be a subgraph of M. Define $\overline{C} = \{\overline{s} : s \in C\}$ to be the image of C in $\overline{M}$. Further, define C^* to be the subgraph $\overline{C} \times I$ of M^*.

Lemma 2.1 *1. If C is a scc of M then $\overline{C}$ is a scc of $\overline{M}$.*

2. If $\overline{C}$ is a scc of $\overline{M}$ then there is a scc D of M with $\overline{D} = \overline{C}$.

3. If $\overline{C}$ is a scc of $\overline{M}$ then C^* is a disjoint (not connected) union of scc of M^*.

2.4 Fairness

In this paper we consider the traditional weak fairness. A computation is said to be *weakly fair* if every process is either infinitely often disabled or executed. This condition leads to the following definitions on M.

Definition 2.2 A *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 executed in C if there are states s, t in C such that $s \xrightarrow{i} t \in R$.)

Definition 2.3 A *scc* C^* of M^* is weakly fair if there is a state $(\overline{s}, k)$ in C^* such that process k is disabled in $\overline{s}$ or C^* has an edge of type $(\overline{s}, k) \xrightarrow{k} (\overline{t}, l)$.

Lemma 2.4 *Let C be a scc of M. Then C is weakly fair if and only if C^* consists of weakly fair sccs of M^*.*

2.5 Model Checking

Let A be an automaton that accepts exactly the 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 a weakly fair *scc* with a final automaton state then P has an weakly fair incorrect computation.

Let $i_1^0, \ldots, i_k^0$ be the indices that are referred to in A. In [ES95], it is shown that we can take the smaller product $\overline{B}_0$ of $\overline{M}$, I^k and A and perform the search for a *scc* in $\overline{B}_0$. Formally, $\overline{B}_0 = \langle \overline{S} \times I^k \times A, \overline{R}_{\mathrm{p}}, \overline{s}_{\mathrm{p}} \rangle$ where $\overline{s}_{\mathrm{p}} = (s_0, i_1^0, \ldots, i_k^0, a_0)$, $\overline{R}_{\mathrm{p}}$ consists of edges $(\overline{s}, i_1, \ldots, i_k, a) \xrightarrow{\pi, l} (\overline{t}, j_1, \ldots, j_k, a')$ such that $\overline{s} \xrightarrow{\pi, l} \overline{t} \in \overline{R}$, $j_1 = \pi^{-1}(i_1), \ldots, j_k = \pi^{-1}(i_k)$, and the automaton A moves from state a to a' on the input gained from $\overline{s}$ after simultaneously interchanging the indices $i_1, \ldots, i_k$ with $i_1^0, \ldots, i_k^0$ respectively. We can define the threaded graph version B_0^* corresponding to $\overline{B}_0$ analogously to that of $\overline{M}$. The analogue of lemmas 2.1 and 2.4 will hold to B_0^*.

Theorem 2.5 *P satisfies the complement of the property defined by A if and only if there is no scc $\overline{C}$ of $\overline{B}_0$ such that $\overline{C}$ contains a final automaton state and C^* consists of weakly fair sccs.*

3 On-the-Fly Model Checking

The original algorithm constructed $\overline{B}_0$ together with the threaded graphs C^*. This method can be improved by applying the following three new ideas.

First, note that the symmetry of $M \times I^k$ was not fully utilized. Instead of compressing M to $\overline{M}$ and then taking the product with $I^k \times A$, we can first compress $M \times I^k$ to $\overline{M \times I^k}$ and then take the product with A. This product graph is denoted by $\overline{B}$. It is possible for two states of the form $(\overline{s}, i_1, \ldots, i_k)$ and $(\overline{s}, j_1, \ldots, j_k)$ to be equivalent and be represented by a single state in $\overline{M \times I^k}$. Thus, $\overline{M \times I^k}$ is possibly smaller than $\overline{M} \times I^k$.

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 $\overline{B}$. Specially, no edges need to be stored.

Finally, the third idea is to use the symmetry of a single global state. Up till this point we did not change the number of edges, $\overline{B}$ has as many edges as $\overline{B}_0$. 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 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 $M \times I$

In subsection 2.2 we defined an equivalence relation on S. Now, we extend it to $S \times I$: $\overline{u} = (\overline{s}, i)$ and $\overline{v} = (\overline{t}, j)$ are equivalent if there is a permutation $\pi \in G$ such that $\pi(\overline{u}) = \overline{v}$, that is, $\pi(\overline{s}) = \overline{t}$, and $\pi(i) = j$. Let $\overline{S}_{\mathrm{n}}$ be a set of *representative* states that contains exactly one state from each equivalence class. The *annotated quotient structure with tracked indices* (AQSI) of $M \times I$ is $\overline{M \times I} = \langle \overline{S}_{\mathrm{n}}, \overline{R}_{\mathrm{n}}, \overline{s}_{\mathrm{n}} \rangle$ where $\overline{R}_{\mathrm{n}} = \{(\overline{s}, i) \xrightarrow{\pi, l} (\overline{t}, j) : \overline{s} \xrightarrow{l} \pi(\overline{t}) \in \overline{R}, j = \pi^{-1}(i)\}$, and $\overline{s}_{\mathrm{n}}$ is (s_0, i_0). In a state $(\overline{s}, i)$, i is called the *tracked process.*

Let $\overline{B}$ be the product of $\overline{M \times I}$ and A. In the actual algorithm presented below we will construct $\overline{B}$ from $\overline{M}$ (instead of $\overline{M \times I}$) and A. The reason for that inconvenience is that we do not want to store $\overline{M \times I}$. During the construction of the product graph $\overline{B}$, we can locally construct $\overline{M \times I}$ from the stored $\overline{M}$. This is done in Command 7 of the algorithm presented below. To ensure that $\overline{S}_{\mathrm{n}}$ and $\overline{S}$ are closely related we adopt the obvious convention that $(\overline{s}, i) \in \overline{S}_{\mathrm{n}}$ implies $\overline{s} \in \overline{S}$.

3.2 Partitions

As explained earlier, our on-the-fly model-checking algorithm will explore the $\overline{B}$ simultaneously as it constructs it. During this process, in order to analyze the threaded graph without explicitly constructing it, we will maintain a partition on I with each $\overline{B}$ node on the stack; this partition will be represented as n-vector.

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.

Let $\overline{r} = (\overline{s}, l, a)$ be a state in $\overline{B}$ and $\overline{C}$ be the *scc* of $\overline{B}$ that contains $\overline{r}$. By the analogue version of lemma 2.1, C^* is a collection of disjoint *scc*s. Thus, if a node in C^* is reachable from both $(\overline{r}, i)$ and $(\overline{r}, j)$ then $(\overline{r}, i)$ and $(\overline{r}, j)$ are in the same *scc* of C^*. Now we define the equivalence relation $\overset{*}{\sim}_{\overline{r}}$ on I as follows:

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

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

It can easily be shown that if $\overline{r}$ and $\overline{r}'$ are nodes in the same *scc* in $\overline{B}$ then the partitions of $\overset{*}{\sim}_{\overline{r}}$ and $\overset{*}{\sim}_{\overline{r}'}$ are not exactly equal, but are equal up to renaming by a permutation in G. This property makes it difficult to directly maintain the partitions of $\overset{*}{\sim}_{\overline{r}}$. This problem can be overcome if we use a common referential base. The perfect nominee for this is the initial state $\overline{s}_{\mathrm{p}}$ of $\overline{B}$.

Let T be a spanning tree of $\overline{B}$ with root $\overline{s}_{\mathrm{p}}$ (in the actual algorithm, we will take T to be the tree induced by the depth first search of $\overline{B}$). For each node $\overline{u} \in \overline{B}$, let $\pi_{\overline{u}}$ denote the product of the permutations on the unique $\overline{s}_{\mathrm{p}} \rightsquigarrow \overline{u}$ path in T. Now, with each state $\overline{r}$ in $\overline{B}$, we define an equivalence relation $\sim_{\overline{r}}$ on I as follows.

$$i \sim_{\overline{r}} j \quad \text{if} \quad \pi_{\overline{r}}^{-1}(i) \overset{*}{\sim}_{\overline{r}} \pi_{\overline{r}}^{-1}(j).$$

Now, it can be shown that, if $\overline{r}$ and $\overline{r}'$ are states in the same *scc* of $\overline{B}$ then $\sim_{\overline{r}}$ and $\sim_{\overline{r}'}$ are equal. Intuitively, if $i \sim_{\overline{r}} j$ then it indicates that processes i and j go through the same sequence of local states in the *scc* containing $\overline{r}$.

Next, we show how to compute $\sim_{\overline{r}}$ by exploring $\overline{B}$ using depth first search. For each edge $e = \overline{u} \xrightarrow{\pi, i} \overline{v}$ in $\overline{B}$, let π_e denote the permutation $\pi_{\overline{u}} \circ \pi \circ \pi_{\overline{v}}^{-1}$. 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 3.1 *If e is an edge in the scc containing $\overline{r}$, then $\pi_e(i) = j$ implies $i \sim_{\overline{r}} j$.*

Let ρ be a permutation on I. Then ρ defines a unique partition on I, called the *orbit partition of* ρ, in the following way: i and j are in the same class of the partition if $\rho^n(i) = j$ for some $n \geq 0$. Now claim 3.1 is equivalent to the following:

If e is an edge in the scc of $\overline{r}$, then the orbit partition of π_e is smaller than or equal to $\sim_{\overline{r}}$.

The following stronger result gives us a method to compute the partition of $\sim_{\overline{r}}$.

Theorem 3.2 *$\sim_{\overline{r}}$ is the smallest partition that contains the orbit partition of π_e, for every edge e of $\overline{C}$.*

The next theorem is a necessary and sufficient condition for checking if a class of $\sim_{\overline{r}}$ is weakly fair.

Theorem 3.3 *A class K of the partition $\sim_{\overline{r}}$ is weakly fair if and only if there is an $i \in K$ and $\overline{u} \in \overline{C}$ such that $\pi_{\overline{u}}^{-1}(i)$ is disabled in $\overline{u}$ or it is executed (that is, there is an edge $\overline{u} \overset{\pi,\pi_{\overline{u}}^{-1}(i)}{\longrightarrow} \overline{v}$ in $\overline{C}$).*

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

3.3 The Algorithm

The algorithm is a modification of the strongly connected component computation using depth first search presented e.g. in [HU74]. For each vertex $\overline{u} = (\overline{s}, l, a)$ of the product graph $\overline{B}$, we maintain the following information.
$\overline{u}$`.dfnum` is a unique id (or *depth first number*) of the node, used for the strongly connected component computation.
$\overline{u}$`.lowlink` is the id of a reachable node lower than $\overline{u}$ itself.
$\overline{u}$`.onstack` is a flag indicating that $\overline{u}$ is still on stack.
$\overline{u}$`.perm` is the vector $\pi_{\overline{u}}$ as defined in the previous subsection.
$\overline{u}$`.partition` is an approximation of $\sim_{\overline{u}}$.
$\overline{u}$`.status` is a vector of flags that indicate which partition classes are known to be weakly fair.
$\overline{u}$`.final` is a flag that indicates if $\overline{u}$ is in a *scc* that contains a final automaton state. This information is propagated down on the depth first tree.

The variables $\overline{u}$`.dfnum`, $\overline{u}$`.lowlink` and $\overline{u}$`.onstack` are maintained as in the algorithm given in [HU74]. Command 3 initializes $\overline{u}$`.status` and $\overline{u}$`.final` appropriately. The two "for" loops in commands 5 and 6, generate the successors of the $\overline{B}$ state $\overline{u}$. In command 7, we construct the product state $\overline{v}$. To do this, we consider the state $(\overline{t}, \pi^{-1}(l))$ of $\overline{M \times I}$ and find an already generated state of the form $(\overline{t}, l')$ and a permutation ρ such that $(\overline{t}, \pi^{-1}(l))$ is equivalent to $(\overline{t}, l')$ under ρ; If no such equivalent state has already been generated, then l' is taken as $\pi^{-1}(l)$ and ρ is taken as the identity permutation. Here the same equivalence checking function can be used as in the construction of $\overline{M}$. (It is to be noted that we need this equivalence checking since we are constructing $\overline{B}$ to be the product $\overline{M \times I}$ 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 [ES95], then we do not need this equivalence checking, and in this case we may have more states in the resulting $\overline{B}$.)

In command 8, we check if the edge $\overline{u} \to \overline{v}$ is a non-tree edge (i.e. $\overline{v}$ has already been visited) and $\overline{v}$ is in the same *scc* as $\overline{u}$. In this case, it merges the orbit partition of π_e with $\overline{u}$`.partition`. Commands 11 thru 14 are executed if the edge $\overline{u} \to \overline{v}$ is a tree edge, i.e. $\overline{v}$ is constructed (and hence visited) for the first time. In command 11, $\overline{v}$`.perm` is set; in command 12, DF-search is invoked on $\overline{v}$. If $\overline{v}$ and $\overline{u}$ are in the same *scc* (indicated by the condition in command 13) then the partitions are combined, the *status* vector of $\overline{u}$ is updated and other updates are carried out. After processing the edge $\overline{u} \to \overline{v}$, in command 15, we check if the partially explored *scc* containing $\overline{u}$ is weakly fair and has a final state; if so the algorithm exits with an "yes" answer indicating a fair computation accepted by the automaton A is found. In command 17, after detecting the *scc*, we pop all the states of the *scc* from the stack.

ON-THE-FLY MODEL CHECKING

M1. Construct $\overline{M}$.
M2. Construct A.
M3. Set $\overline{u}$ to be the initial state $\overline{s}_{\mathrm{p}}$ of $\overline{B}$.
Set $\overline{u}$`.perm` to be the identity permutation.
Set the depth-first-counter to zero.
Conduct DF-Search($\overline{u}$).

DF-Search($\overline{u}$) (Note that $\overline{u} = (\overline{s}, l, a)$).
1. Push $\overline{u}$ in the stack, set $\overline{u}$`.onstack`.
Set $\overline{u}$`.dfnum` and $\overline{u}$`.lowlink` to the depth-first-counter.
Increase the depth-first-counter.
2. Initialize $\overline{u}$`.partition` to be the identity partition.
3. Initialize $\overline{u}$`.status` with the information on disabled processes stored in the AQS state $\overline{s}$.
Set $\overline{u}$`.final` if a is a final automaton state.
4. (Idle command. Later modification will use it.)
5. For each automaton transition $a \rightarrow a'$ that is enabled in $\overline{s}$ do
6. For each AQS edge $e = \overline{s} \xrightarrow{\pi, i} \overline{t}$ do
7. Find an already generated AQSI state of the form $(\overline{t}, l')$ that is equivalent to $(\overline{t}, \pi^{-1}(l))$ and
find the permutation $\rho \in G$ that maps $(\overline{t}, \pi^{-1}(l))$ to $(\overline{t}, l')$.
Let $\overline{v} = (\overline{t}, l', a')$.
8. If $\overline{v}$ is already constructed and $\overline{v}$`.onstack` is set do
9. Merge $\overline{u}$`.partition` with the orbit partition of π_e.
Update $\overline{u}$`.status` using that process i was executed.
Update $\overline{u}$`.lowlink` to be the minimum of $\overline{u}$`.lowlink` and $\overline{v}$`.lowlink`.
10. If $\overline{v}$ is not constructed yet then do
11. Set $\overline{v}$`.perm` $=$ $\overline{u}$`.perm` $\circ\, \pi \circ \rho$.
12. Conduct DF-Search($\overline{v}$).
13. If $\overline{v}$`.onstack` is still set then do
14. Combine $\overline{v}$`.partition` to $\overline{u}$`.partition`.
Combine $\overline{v}$`.status` to $\overline{u}$`.status`.
Update $\overline{u}$`.status` using that process i was executed.
Update $\overline{u}$`.lowlink` to be the minimum of $\overline{u}$`.lowlink` and $\overline{v}$`.lowlink`.
Set $\overline{u}$`.final` if $\overline{v}$`.final` is set.
15. If all the partition classes are weakly fair (use $\overline{u}$`.status`) and $\overline{u}$`.final` is set then exit with *Yes* answer.
16. If $\overline{u}$`.dfnum` $=$ $\overline{u}$`.lowlink` then do
17. Pop all elements above $\overline{u}$ (inclusive) from the stack and mark the popped vertices off–stack.
18. Return with *No* answer.

Theorem 3.4 *The algorithm described above outputs* Yes *if and only if the original program has a weak fair computation that is accepted by A. The algorithm runs in time $O(|\overline{M \times I}| \cdot |A| \cdot n)$.*

3.4 State Symmetry, Partition Initialization

Let $\overline{u} = (\overline{s}, l, a)$ be a vertex of the product graph $\overline{B}$. Process i and j are called $\overline{u}$*–equivalent*, denoted by $i \overset{*}{\approx}_{\overline{u}} j$, if there is a permutation $\rho \in G$ such that $\rho(i) = j$ and $\rho(\overline{u}) = \overline{u}$. Intuitively, $i \overset{*}{\approx}_{\overline{u}} j$ shows that process i and j are identical in state $\overline{u}$. Let $\overline{u} \xrightarrow{\pi, l} \overline{v}$ be an edge of $\overline{B}$. Then $\overline{u} \xrightarrow{\rho\circ\pi, \rho(l)} \overline{v}$ is also an edge yielding that $(\overline{v}, \pi^{-1}(i))$ is a successor of both nodes $(\overline{u}, i)$ and $(\overline{u}, j)$ in the threaded graph B^*. Therefore, $(\overline{u}, i)$ and $(\overline{u}, j)$ are in the same *scc* of B^*. Hence, $i \overset{*}{\sim}_{\overline{u}} j$.

Lemma 3.5 *The partition* $\overset{*}{\approx}_{\overline{u}}$ *is smaller than* $\overset{*}{\sim}_{\overline{u}}$.

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

2′. Initialize $\overline{u}$`.partition` to be $\approx_{\overline{u}}$.

3.5 Parallel edges

Let $e = \overline{u} \xrightarrow{\pi, l} \overline{v}$ and $e' = \overline{u} \xrightarrow{\pi', l'} \overline{v}$ be edges in $\overline{B}$. We call e and e' *parallel* if there is a permutation $\rho \in G$ such that $\rho(\overline{u}) = \overline{u}, \rho \circ \pi = \pi', \rho(l) = l'$. Surely, being parallel is an equivalence relation on the edges. Let $\overline{R}^r_{\mathrm{p}}$ be a set of *representative edges* that contains at least one edge from each parallel class. Having the partitions 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 was considered. It is reflected in the next lemma.

Lemma 3.6 *If* $\overline{u}$ *is in a scc of* $\overline{B}$ *then* $\sim_{\overline{u}}$ *is the smallest partition that contains* $\approx_{\overline{v}}$ *(the initial value of* $\overline{v}$`.partition`*) for every* $\overline{v}$ *in the scc of* $\overline{u}$ *as well as the orbit partition of* π_e *for every edge* $e \in \overline{R}^r_{\mathrm{p}}$.

These ideas can be applied as follows. From each class of $\approx_{\overline{u}}$ pick a representative process and call it the *leader* of that class. Put $\overline{R}^r_{\mathrm{p}} = \{\overline{u} \xrightarrow{\pi, l} \overline{v} \in \overline{R}_{\mathrm{p}} \ : \ l \text{ is a leader}\}$. Since every edge is parallel to one that was caused by a leader process, this $\overline{R}^r_{\mathrm{p}}$ is a satisfactory set of representative edges. We introduce the new vector $\overline{u}$`.leader` of flags. The next improvement in the algorithm is the introduction of Command 4 and the modification of Command 6.

4. Initialize $\overline{u}$`.leader`.
5. For each automaton transition $a \to a'$ that is enabled in $\overline{s}$ do

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

3.6 Redundant edges in $\overline{M}$

In this subsection we show that the edges of $\overline{M}$ can be stored in a more compact way. For that we define state equivalence for every AQS state $\overline{s}$.

$$i \approx_{\overline{s}} j \text{ if there is a } \mu \in G \text{ with } \mu(\overline{s}) = \overline{s}, \mu(i) = j.$$

Note that in subsection 3.4 we introduced state equivalence for $\overline{B}$ states. The state equivalence relation for $\overline{u} = (\overline{s}, l, a)$ was denoted by $\overset{*}{\approx}_{\overline{u}}$. We recall that $i \overset{*}{\approx}_{\overline{u}} j$ if there is a $\mu \in G$ with $\mu(\overline{s}) = \overline{s}, \mu(l) = l$ and $\mu(i) = j$. It follows that $i \overset{*}{\approx}_{\overline{u}} j$ implies $i \approx_{\overline{s}} j$.

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

Now we can present the last improvement to our algorithm.

M1′. Construct $\overline{M}$ with the following modifications. When a new node $\overline{s}$ is created, compute $\approx_{\overline{s}}$. Define a vector $\overline{s}$`.repr` that, for every index i, points to a representative of the $\approx_{\overline{s}}$-class of i. By the construction of the edges of $\overline{M}$, store only those edges that are caused by a representative process. (So $\overline{s} \xrightarrow{\pi,i} \overline{t}$ is stored if $\overline{s}$`.repr[`i`]` $= i$.)

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

11′ Set $\overline{v}$`.perm` $=$ $\overline{u}$`.perm` $\circ\, \mu \circ \pi \circ \rho$.

4 Implementation

We have developed a prototype of the on-the-fly model checker implementing the above presented algorithm.

We have used the implemented system to check for the correctness of a resource controller example. In this example, we have a resource controller and many user processes. Each user process communicates with the controller through shared variables. For this system, we were able to check many properties including the mutual exclusion property, and the liveness property that every user process that requests a resource will eventually access the resource.

We ran both our new model checker and as the old model checker that is an implementation of the results presented in [ES95] on the resource controller example with various number of users. We found a dramatic improvement in all performance measures of the new model checker. The performance statistics we obtained are given in table 1 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, b are the numbers for the cases when we ran the old and new model checkers respectively.

The liveness property that we checked is not satisfied by the resource controller example. Both the 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; this is because we used state symmetry to eliminate many parallel edges. The number of product states explored in the on-the-fly system is much smaller since it terminated early, (also the number of over all product states in the new model checker is smaller because we used $\overline{M \times I} \times A$ instead of $\overline{M} \times I \times A$; however this is not shown here). On the other hand the original model checker constructed the whole $\overline{B}$ before checking for an incorrect fair computation.

For the mutual exclusion property, both the model checkers indicated that the resource controller satisfies the mutual exclusion property. In this case, early termination does not come into effect. Further more, since we do not track any process (because 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 are much smaller in $\overline{M}$ as well as $\overline{B}$ due to the effect of state symmetry. The over all CPU time as well as the memory usage is substantially smaller for the new model checker.

5 Conclusions

In this paper, we have presented an on-the-fly model-checking system that exploits symmetry and checks for correctness under fairness. The algorithm explores the product graph while at the same it checks for incorrect computations. It also exploits state symmetry to reduce the time as well as the memory usage. We have implemented the system, and have presented experimental results that show dramatic improvements in the above performance measures.

As part of future work, it will be interesting 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

[CES83] Clarke, E. M., Emerson, E. A., and 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 1993, Austin, Texas, Also appeared in *ACM TOPLAS*, April 1986

[CFJ93] Clarke, E. M., Filkorn, T., Jha, S. Exploiting Symmetry in Temporal Logic Model Checking, 5th International Conference on Computer Aided Verification, Crete, Greece, June 1993.

[ES93] Emerson, E. A., and Sistla, A. P., Symmetry and Model Checking, 5th International Conference on Computer Aided Verification, Crete, Greece, June 1993

[ES95] Emerson, E. A., and Sistla, A. P., Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata-theoretic Approach, 7th International Conference on Computer Aided Verification, Leige, Belgium, July 1995.

[HU74] A. V. Aho, J. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley 1974.

Liveness 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	146 / 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	479 / 42

Mutual Excl. 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	151 / 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	160 / 44

Table 1: Statistics for the model checker