Checking Extended CTL properties Using Guarded Quotient Structures

A. Prasad Sistla, Xiaodong Wang, Min Zhou

***Abstract*— We extend CTL logic to a logic called COUNT CTL ($CCTL$) for specifying properties of concurrent programs with large number of processes. We present a model checking algorithm for symmetric or partially symmetric systems when their correctness specification is given in $CCTL$. The model-checking algorithm employs Guarded Quotient Structures introduced in [9]. The GQS structures can be succinct representations for the reachability graphs of partially symmetric or even asymmetric systems. Our algorithm exploits state symmetries for fast evaluation. The algorithm is top down in nature, and automatically incorporates formula decomposition and sub-formula tracking.**

I. Introduction

Recently there has been much interest in symmetry based methods for containing the state explosion problem in model checking. The early symmetry based methods, introduced in [1], [8], [3], exploit the symmetries in the system to identify states that are equivalent under symmetry and construct a quotient structure. The model checking is carried out on the quotient structure. This method can primarily be used for verifying symmetric properties specified in temporal logic, i.e. properties in which the atomic propositions have same truth values on equivalent states. A later approach introduced in [4] constructs an Annotated Quotient Structure (AQS) and unwinds it partially to verify a temporal property. This method can be used for checking both symmetric and asymmetric properties under various notions of fairness. In [7] the AQS method is further extended to check correctness under fairness on-the-fly. These methods have been implemented in the SMC model checker [10].

In [5], [6], the method based on quotient structures is extended to verify symmetric properties of partially symmetric and asymmetric systems also. In [9], the AQS based method is extended to verify symmetric and asymmetric properties of partially symmetric and asymmetric systems as well. This method is based on constructing a Guarded Quotient Structure (GQS). It can be used to verify asymmetric properties of such systems as well. This method works as follow. In order to model check a property for a program $\mathcal{K}$, another program $\mathcal{K}'$ is considered so that the later program has lot more symmetries. Usually, $\mathcal{K}'$ is obtained from $\mathcal{K}$ by simple transformations (such as ignoring process priorities, etc.). Formally, if G and H are the global state graphs of $\mathcal{K}, \mathcal{K}'$ respectively, then they have the same set of states, but H has more edges. Further, the set of symmetries of H is a super set of the set of symmetries of G. The GQS is constructed by first constructing the AQS of H and by adding edge conditions to the AQS. By unwinding GQS with respect to its edge conditions, we can get G from GQS. In [9], a model checking method for CTL^* employing the GQS is presented. In that work, two optimization methods called formula decomposition and sub formula tracking were introduced.

This paper is supported in part by the NSF grants CCR-9988884, CCR-0205365.

In this paper, we extend the branching time temporal logic CTL to a new logic called COUNT CTL ($CCTL$) for specifying properties of concurrent programs. $CCTL$ has all the path quantifiers and temporal operators of CTL and the fair path quantifiers of $Fair\_CTL$ [11] and allows an additional construct, called $COUNT$, which is a function that returns the number of processes that satisfy a given property in a given state. For example, if $C(i)$ is an atomic proposition denoting that process i is in critical section and M is a set of processes, then $COUNT(i, M, C(i))$ gives the number of processes in M that are in the critical section in the state. $COUNT$ can be nested with temporal operators. With $COUNT$, $CCTL$ can specify that a property should hold for some processes (or for all processes) belonging to a class; that is, it can express process quantifiers. $CCTL$ is more expressive than the logic $ICTL$ considered in earlier papers [2]. For example, it can express uniformly the property that the number of processes satisfying property P equals the number of processes satisfying property Q. This property cannot be expressed in $ICTL$ uniformly.

We consider the model checking problem for partially symmetric and asymmetric systems when the correctness specification is given in $CCTL$. We employ GQS for the model checking purpose. We assume that the GQS has already been constructed from the concurrent program. Our model checking algorithm exploits state symmetries of a particular state to evaluate the $COUNT(i, M, \phi(i))$. It first partitions the set of processes M over which i ranges into equivalence classes. Instead of checking $\phi(i)$ with every i in M, we check $\phi(i)$ for those i that corresponds to representatives of equivalence classes. The evaluation is returned by summing up the cardinalities of the equivalence classes when $\phi(i)$ holds with i instantiated with its representative.

Our model-checking algorithm employs lazy evaluation. That is, we invoke the algorithm on the main formula, which invokes on its sub-formulas only if and when it is needed to evaluate their truth values. For example, when invoked to check a formula of the form $g \wedge h$ at a state s, it invokes on g ; the sub-formula h is checked at state s only if g is determined to be satisfied at s.

Our model-checking algorithm is top-down in its approach. The algorithm works inductively over the structure of $CCTL$

formula and thus employs formula decomposition in a seamless manner. With formula decomposition, the algorithm can minimize the GQS unwinding. GQS is unwound with respect to the process ids in sub-formulas instead of all those process ids in the main formula; This reduces the complexity of the algorithm. Formula decomposition was introduced earlier in [9] to check CTL^* properties. Unlike in that paper, formula decomposition is naturally incorporated into our algorithm.

The algorithm described in the paper has been implemented by employing the GQS constructed from the SMC model-checker. Experimental results giving the effectiveness of our method are given.

The rest of the paper is organized as follows. Section 2 introduces the background and notations. Section 3 gives the definition of $CCTL$. Section 4 describes the algorithm. Section 5 describes the implementation and some experimental results. Section 6 concludes the paper with discussion of related work.

II. Background and Notation

We consider a concurrent program $\mathcal{K}$ consisting of n processes. We denote the process ids by the integers $0, ..., n-1$ and let I denote the set $\{0, 1...., n-1\}$. The processes in the concurrent program communicate through variables. We call these as program variables. The name of a program variable is given by an identifier subscripted with the names of processes that share the variable. For example, $u_{1,2}$, $u_{2,3}$ are variables shared by processes $1, 2$ and by processes $2, 3$ respectively. A state s of the program is a mapping associating values to the program variables. Let $G = (S, E)$ be the reachability graph of the concurrent program.

Let $Sym\ I$ be the set of all permutations π on I. $Sym\ I$ forms a group with functional composition ($\circ$) being the group operation. Our convention is that $\pi_b \circ \pi_a$ is evaluated right-to-left: first apply π_a, then π_b. Let Id denote the identity permutation and π^{-1} the inverse of π. For any indexed object b, such as a state, a variable, or a formula, whose definition depends on I, we can define the notion of permutation π acting on b, by simultaneously replacing each occurrence of index $i \in I$ by $\pi(i)$ in b to get the result $\pi(b)$. For a variable $u_{i,j}$, $\pi(u_{i,j})$ is $u_{\pi(i),\pi(j)}$. For a state s, $\pi(s)$ is the state where, for all program variables x, $\pi(s)(x) = s(\pi^{-1}(x))$. For a set C of states, let $\pi(C) = \{\pi(s) : s \in C\}$. Similarly, for a set of edges F, we let $\pi(F) = \{(\pi(s), \pi(s')) : (s, s') \in F\}$.

A permutation π is called an auto morphism of the graph $G = (S, E)$ if $\pi(G) = G$, i.e., $\pi(S) = S$ and $\pi(E) = E$. The set of all auto morphisms of G forms a group and let $\mathcal{G}$ denote this group. As has been shown in earlier works [1], [3], [8], $\mathcal{G}$ induces an equivalence relation $\equiv_{\mathcal{G}}$ on the set of states S given by $s \equiv_{\mathcal{G}} t$ if there exists a $\pi \in \mathcal{G}$, such that $\pi(s) = t$. The above works also construct a quotient structure $QS(G, \mathcal{G})$ to model check symmetric properties of G (i.e. properties specified in CTL^* or mu-calculus where the atomic propositions have the same truth values in all equivalent states). In [4], [7], the authors construct an annotated quotient structure $AQS(G, \mathcal{G})$ that can be unwound to check any property specified in CTL^* (the atomic propositions need not have the same truth values in equivalent states). The idea of the later work is to unwind the $AQS(G, \mathcal{G})$ partially to model check for the required property.

In [5], [6], the method based on the quotient structure is extended to check symmetric properties of systems with less symmetry or no symmetry. In [9] the method based on the AQS is extended to handle systems with less or no symmetry. The later work is based on constructing a Guarded Quotient Structure (GQS). This is done by considering another graph $H = (S, F)$ which has the same set of states as G and satisfies the following conditions: (i) $F \supseteq E$, i.e., it has all the edges of G and possibly more; (ii) its set of automorphisms is a superset of $\mathcal{G}$. We let $\mathcal{H}$ denote the set of automorphisms of H. Usually we choose H so that $\mathcal{H}$ is much larger than $\mathcal{G}$. The equivalence relation $\equiv_{\mathcal{H}}$ is extended to the edges in F in the obvious way, i.e. for $e, e' \in F$, $e \equiv_{\mathcal{H}} e'$ if there exists a permutation $\pi \in \mathcal{H}$ such that $e' = \pi(e)$. Let $class(e, \mathcal{H})$ be the set of edges in the equivalence class of e. The GQS of H with respect to $\mathcal{H}$ is denoted by $GQS(H, \mathcal{H}, G)$ and is a triple $(\overline{V}, \overline{F}, C)$ defined as follows: $\overline{V} \subseteq S$ is a set of states that contains one representative for each equivalence class of $\equiv_{\mathcal{H}}$; $\overline{F} \subseteq \overline{V} \times \overline{V} \times \mathcal{H}$ is a set of labeled edges such that, for every $\overline{s} \in \overline{V}$ and $t \in S$ such that $(\overline{s}, t) \in F$, there exists an element $(\overline{s}, \overline{t}, \pi) \in \overline{F}$ such that $\pi(\overline{t}) = t$; C is a function that associates a condition $C(e)$ with each labeled edge e in $\overline{F}$; $C(e)$ denotes an edge condition such that the set of edges in $class(e, \mathcal{H})$ that satisfy $C(e)$ is exactly the set of edges $class(e, \mathcal{H}) \cap E$. The edge conditions $C(e)$ are specified by a propositional condition on the program variables.

Fig. 1 shows the reachability graph G of a 2-process mutual exclusion algorithm where process 1 has higher priority. The nodes of G are elements belonging to $\{N_1, T_1, C_1\} \times \{N_2, T_2, C_2\}$. We also consider each node of G to be a two element set. For any such node s, if $N_i \in s$ or $T_i \in s$ or $C_i \in s$ (for $i = 1, 2$) this intuitively denotes that process i is in the non-critical section or in the trying section or in the critical section, respectively. We add an edge from the node (T_1, T_2) to (T_1, C_2) to make it symmetric and obtain H. The GQS corresponding to H is shown in Fig. 1. In the GQS only two edges have non-trivial guards. Here $Flip$ is the permutation which interchanges processes 1 and 2; it defines an automorphism on the nodes of H that maps a node $\{D_i, E_j\}$ (where D, E are any of the symbols N, T, C and $1 \leq i, j \leq 2$) to the node $\{D_{Flip(i)}, E_{Flip(j)}\}$. Here id is the identity permutation defining the identity automorphism. For any F in $\{N_1, N_2, T_1, T_2, C_1, C_2\}$, we let F-nodes denote the set of nodes in H that contains the element F. The edge predicate $T_1 \wedge C_1'$ denotes the set of edges in H from a T_1-node to a C_1-node; it is expressed as a formula stating that the current state satisfies T_1 and that the next state satisfies C_1 (the clause C_1' states that C_1 should be satisfied in the next state). There is only one edge labeled by this predicate: this edge is from the node (T_1, T_2) to the node (C_1, T_2).

In many situations, the $GQS(H, \mathcal{H}, G)$ can be constructed directly from the concurrent program description so that it is more succinct than $AQS(G, \mathcal{G})$. In [9], the authors developed a method for checking a property, specified in CTL^*, of the program by unwinding $GQS(H, \mathcal{H}, G)$ appropriately.

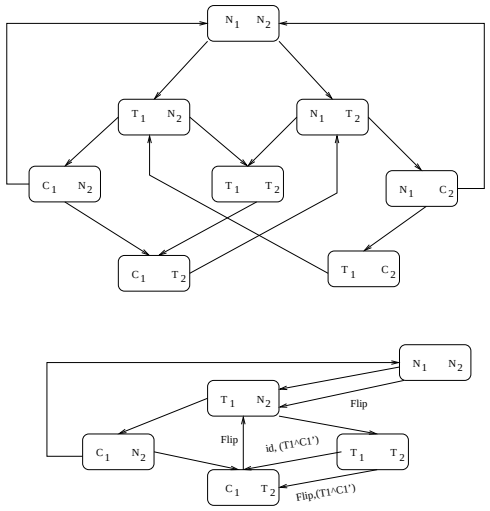

Fig. 1. Glocal Transition Graph and Guarded Annotate Quotient Structure

They also presented optimizing techniques involving formula decomposition and sub formula tracking. The method given there was implemented in the system $PSMC$. The system only checks properties of the form $E(p)$ where p is a linear temporal formula. It did not implement the full CTL^*. It did not implement formula decomposition and sub-formula tracking. The experiments given in [12] employed a high level formula decomposition that was manually carried out.

III. $CCTL$ Logic

In this section we define the syntax of the $CCTL$ logic. $CCTL$ formulas use process variables that range over process ids of the system. They only use program variables in which all the subscripts are process variables, i.e. no process ids are used. $CCTL$ formulas also use constants belonging to the domains of program variables, a function symbol $COUNT$, sets of process ids and comparison operators in $\{=, <, >, >=, <=\}$. In addition to path quantifiers over all paths, $CCTL$ employs path quantifier E_{fair} which quantifies over fair paths. We use standard weak process fairness in [11], i.e., a path is fair if every process is either executed or disabled infinitely often.

Formally, $CCTL$ formulas are defined as follows:

⟨ **formula** ⟩ :: ⟨ atomic formula ⟩ |
⟨ count-term ⟩ ⟨ comp-operator ⟩ ⟨ count-term ⟩ |
⟨ formula ⟩ ∧ ⟨ formula ⟩ | ¬ ⟨ formula ⟩ |
EX (⟨ formula ⟩) | $E_{fair}X$ (⟨ formula ⟩) |
EG (⟨ formula ⟩) | $E_{fair}G$ (⟨ formula ⟩) |
E (⟨ formula ⟩ U ⟨ formula ⟩) |
E_{fair} (⟨ formula ⟩ U ⟨ formula ⟩)

⟨ **count-term** ⟩ :: $COUNT(i, M, \langle formula \rangle)$ |
⟨ constant ⟩

The syntax of the formulas is easily understood from the above BNF notation.

An atomic formula is any of the following: the constant $True$; a binary variable x which is also called an atomic proposition; it is of the form $x \,\rho\, y$ where x, y are program variables or constants and ρ is a comparison operator; it is of the form $i = j$ where i, j are process variables.

A count-term is a term of the form $COUNT(i, M, \phi)$ where ϕ is a $CCTL$ formula and M is a set of process ids and i is a process variable. A count-term can also be an integer constant. Unless otherwise stated, throughout the paper, a count term will refer to a non-constant count-term. The set M of process ids may be either explicitly given or may be specified by a symbolic name which is bound to a set of process ids by a separate command. We say that every occurrence of i in ϕ is bound.

An occurrence of a process variable i in a formula is said to be free if it is not a bound occurrence. We assume that all the occurrences of a process variable are free or all its occurrences are bound in the same count-term. If this property is not satisfied, we can obtain another formula, by renaming the process variables, that satisfies the above condition. For a formula ϕ, let $free\_var(\phi)$ denote the set of process variables appearing free in ϕ. An evaluation for ϕ is a partial function from the $free\_var(\phi)$ to I, the set of process ids. For a count-term u, we define $free\_var(u)$ and evaluation for u, exactly on the same lines as that for a formula. For a count-term u and evaluation f for u, we let $val(s, u, f)$, as defined below, denote the value of the term u in the state s with respect to the evaluation f.

Now we define the semantics of a $CCTL$ formula. The semantics of a formula ϕ are defined in a global state graph $G = (S, E)$ with respect to an evaluation for ϕ in an inductive manner. We denote the satisfaction of ϕ in a state s in G with respect to an evaluation f by $G, s, f \models \phi$. Since G is understood here, we simply write $s, f \models \phi$. The satisfaction relation $\models$ and the function val (i.e., the value of count-terms) are defined mutually inductively. For a subscripted program variable x and an appropriate evaluation f, let $f(x)$ denote the program variable obtained by substituting process variables as given by f for process ids. For an atomic formula of the form $x \,\rho\, y$ where x, y are subscripted program variables, $s, f \models x \,\rho\, y$ if the values of the program variables $f(x), f(y)$ in the program state s are related by ρ. For an atomic formula of the form $i = j$, $s, f \models i = j$ if $f(i) = f(j)$. The satisfaction of $CCTL$ formulas of the form $g \wedge h$, $\neg g$, $E(gUh), EX(g)$ and $EG(g)$ are defined in the standard way as they are defined for CTL formulas. The satisfaction of $E_{fair}(gUh)$, $E_{fair}X(g)$, $E_{fair}G(g)$ are all defined naturally by considering only fair paths. For example, $E_{fair}(gUh)$ is satisfied at a state s if there exists a fair path from s on which g continues to be satisfied until h is satisfied. For a formula of the form $u \,\rho\, v$ where u, v are count-terms, $s, f \models u \,\rho\, v$ if $val(s, u, f)$ and $val(s, v, f)$ are related by the comparison operator ρ, e.g., the values of the two count-terms in the state s are equal if ρ is the equality operator.

For any count-term u, state s, evaluation f for u, we define the value $val(s, u, f)$ as follows. If u is a constant then $val(s, u, f)$ is the constant itself. Let u be the count-term $COUNT(i, M, g)$. For any process id c, let f_c denote the evaluation for g such that $f_c(i) = c$ and for any $j \in dom(f)$ and $j \neq i$, $f_c(j) = f(j)$. We define $val(s, u, f)$ to be the

number of distinct values of c in M such that $s, f_c \models g$.

$CCTL$ can be used to express universal and existential process quantifiers ranging over a set M of process ids. For example, the property $\forall i \in M\ (h(i))$ can be expressed as $COUNT(i, M, h) = COUNT(i, M, True)$.

Other standard CTL temporal operators such as $A(\phi_1 U \phi_2)$, $AG(\phi_1)$, $AX(\phi_1)$, $A_{fair}(\phi_1 U \phi_2)$, $A_{fair}G(\phi_1)$ and $A_{fair}X(\phi_1)$ can all be expressed in $CCTL$. For example, $A_{fair}(\phi_1 U \phi_2) \equiv \neg(E_{fair}(\neg\phi_2 U(\neg\phi_1 \wedge \neg\phi_2)) \vee E_{fair}G(\neg\phi_2))$.

IV. Algorithm

In this section, we present the algorithm for checking $CCTL$ formulas using the guarded quotient structure. First, we need the following definition. Recall that an evaluation, for a $CCTL$ formula or for a count-term, is a partial function whose domain is the set of free variables in the formula or the count-term, respectively.

Recall that H is the reachability graph of the concurrent program and $\mathcal{H}$ is its set of automorphisms. We make the assumption that the set of process ids in every count term, appearing in the formula we want to check, is invariant under the permutations in $\mathcal{G}$. That is, for every count term of the form $COUNT(i, M, g)$ and for every $\pi \in \mathcal{G}$, $\pi(M) = M$. Here let $\pi(M)$ be the set $\{\pi(c) : c \in M\}$.

Definition 1: Let f, f' be evaluations, then f' is an extension of f if $dom(f') \supseteq dom(f)$ and $\forall i \in dom(f)\ f'(i) = f(i)$.

First, we consider the problem of evaluating the value of a count term $COUNT(i, M, \phi)$ in a particular state s. As indicated in the introduction, we exploit the state symmetries (see [3], [7]) in G to evaluate the count term efficiently. For a state s in H, let $Aut(s)$ denote the set of all $\pi \in \mathcal{G}$ such that $\pi(s) = s$. We call the permutations in $Aut(s)$ as symmetries of the state s.

Definition 2: Let u be a count term of the form $COUNT(i, M, \phi)$, s be a state in S and f be an evaluation for u. We define an equivalence relation among processes in M as follows:

$$c_1 \approx_{s,f} c_2 \text{ iff } \exists \pi \in Aut(s), \forall v \in dom(f)$$
$$\pi(f(v)) = f(v),\ \pi(c_1) = c_2$$

.

Definition 3: Let u, s and f be as given in definition 2. For any process c, let f_c be an evaluation for ϕ which is an extension of f such that $f_c(i) = c$.

Now, we have the following theorem. It states that if c, d belong to the same equivalence class of $\approx_{s,f}$ then (s, f_c) satisfies ϕ iff (s, f_d) satisfies ϕ.

Theorem 1: Let u be the count-term $COUNT(i, M, \phi)$ and f be an evaluation for u. Let c and d be process ids in M such that $c \approx_{s,f} d$. Then

$$s, f_c \models \phi \text{ iff } s, f_d \models \phi$$

Proof: By a simple straightforward induction on the structure of ϕ, it is easy to see that for any $\pi \in \mathcal{G}$, $\pi(\phi) = \phi$ (recall that $\pi(\phi)$ is obtained from ϕ by replacing the range M of every count term by $\pi(M)$; since we assumed that for every such M, $\pi(M) = M$, it follows that $\pi(\phi) = \phi$). Since $c \approx_{s,f} d$, there exists a $\pi \in Aut(s)$ such that $\pi(c) = d$ and for $v \in dom(f), \pi(f(v)) = f(v)$. It is not difficult to see that $\pi(f_c) = f_d$. Since $\pi \in \mathcal{G}$, it follows that $s, f_c \models \phi$ iff $\pi(s), \pi(f_c) \models \pi(\phi)$. Since $\pi(s) = s$, $\pi(\phi) = \phi$ and $\pi(f_c) = f_d$, it follows that $s, f_c \models \phi$ iff $s, f_d \models \phi$. □

We take the following approach, called *quantifier elimination*, for evaluating u in the state s with respect to f. From theorem 1, it is easy to see that for each equivalence class of $\approx_{s,f}$, it is enough if we pick one representative c, and check if $s, f_c \models \phi$. Let $C_1, ..., C_k$ be the equivalence classes of $\approx_{s,f}$. Let j_r, for $1 \leq r \leq k$, be a representative from the class C_r. We compute the value of the term u in s with respect to f (i.e., the value $val(u, s, f)$) to be the sum of the cardinalities of the sets C_r such that $s, f_{j_r} \models \phi$. Thus we need to make only k different checks for computing the value of u, instead of n checks in the naive approach.

Description of the Algorithm

In order to check if a formula ϕ is satisfied at a state in G, we consider an expanded graph H (as given in section II). We construct a guarded quotient structure $GQS(H, \mathcal{H}, G)$. Here we assume that this structure is already constructed as given in [9]. Recall that the edges in $GQS(H, \mathcal{H}, G)$ are annotated with permutations and are also associated with edge conditions which act as guards.

Consider a path $s_0, s_1, ..., s_l$ in the guarded quotient structure. Let $\pi_1, ..., \pi_l$ be the permutations labeling the corresponding edges, and $e_1, ..., e_l$ be the edge conditions of the edges respectively. Let π'_i for $i = 1, ..., l$ be the composition of the permutations $\pi_1, ..., \pi_i$ from left to right in that order. Also let $t_0, ..., t_l$ be a sequence of states such that $t_0 = s_0$ and for $i = 1, ..., l$ $t_i = \pi'_i(s_i)$. From the construction of the $GQS(H, \mathcal{H}, G)$, it is the case that $t_0, ..., t_l$ is a path in H but may not be a path in G. If the edge conditions $e_1, ..., e_l$ are satisfied by the edges $(t_0, t_1), ..., (t_{l-1}, t_l)$ then the above path is also a path in G. In order to evaluate if (s_0, f) satisfies ϕ in G, we change the evaluation as we traverse along a path instead of unwinding GQS directly. For example, suppose that we want to check $AG(c_r)$, where c_r is a subscripted binary variable, with respect to the evaluation f_0 where $f_0(r) = 1$ (here AG is the derived CTL operator denoting invariance). We traverse along the path by checking c_r at each successive node s_i with respect to the evaluation f_i where $f_i(r) = (\pi'_i)^{-1}(r)$. Thus we change the evaluation instead of unwinding $GQS(H, \mathcal{H}, G)$. It is to be noted that $f_i = (\pi_i)^{-1}(f_{i-1})$ for $i > 0$. Thus successive evaluations can be obtained, from the evaluation at the previous node, by applying the inverse of the permutation along the edge.

We check the edge conditions as we traverse along a path. We can traverse an edge only if the corresponding edge condition is satisfied. This is straightforward if we use the graph H; simply evaluate the edge condition on the edge and traverse it only if it is satisfied. With $GQS(H, \mathcal{H}, G)$, we accomplish it by tracking the process ids that appear in all the edge conditions by changing it along the path. Suppose all the edge conditions refer to only process 0. As we traverse along a path, we change this process according to the permutations

along the path. Let k_i denote this process when we reach node s_i. Initially, k_0 is set to 0. When we reach node s_i, we set k_i to be $(\pi'_i)^{-1}(0)$. To determine, if edge (s_i, s_{i+1}) can be traversed, we replace process 0 with process k_i in the edge condition e_{i+1} and evaluate this new edge condition. Again note that $k_i = (\pi_i)^{-1}(k_{i-1})$. Thus, successive values of k can be obtained by applying the inverse of the permutation labeling the edge. We use $\vec{k}$ to denote the vector of the process ids that appearing in all edge conditions. Each process id in $\vec{k}$ will change along the path in the same way as explained.

The main procedure $check(\phi, f, \vec{k}, s)$ checks if $s, f \models \phi$ using the process identities in the vector $\vec{k}$ as the corresponding process ids in the edge conditions. As indicated above, f is the evaluation for ϕ which has been changed all along from the initial state to s. The procedure associates two data structures $L(s)$ and $marked(s)$ with each state s in the $GQS(H, \mathcal{H}, G)$. $L(s)$, at the end, contains all triples $(\phi, f, \vec{k})$ such that $s, f \models \phi$ using the process ids in $\vec{k}$ in the edge conditions. $marked(s)$ contains those triples $(f, \vec{k}, \phi)$ such that $EUCheck$ or $EGCheck$ procedure has been invoked with $\phi, f, \vec{k}, s$ as parameters.

As indicated earlier, our model-checking algorithm uses lazy evaluation and works in top-down fashion. Initially $check$ is invoked on the main formula at the state where we want to determine its truth value with an evaluation. The initial values of the parameter $\vec{k}$ are the process ids that appear in the edge conditions.

In order to check with fairness, we introduce a new atomic formula $Exists\_fair\_path$. For a state s and an empty evaluation f, $G, s, f \models Exists\_fair\_path$ iff there exists a fair path in G starting from the state s. Note that a fair path is defined as in section III. Now it is easy to see that $E_{fair}X(\phi_1)$ is equivalent to $EX(\phi_1 \wedge Exists\_fair\_path)$ and $E_{fair}(\phi_1 U \phi_2)$ is equivalent to $E(\phi_1 U(\phi_2 \wedge Exists\_fair\_path))$. We replace the sub-formulas of the above form by the corresponding equivalent formulas and model check for them. To model check the equivalent sub-formulas, we need to give an algorithm for the procedure $check(Exists\_fair\_path, f, \vec{k}, s)$. For this we give the algorithm $efpCheck$ which adopts the method for checking existence of fair paths given in [7]. In order to handle sub-formulas of the form $E_{fair}G(\phi_1)$, we give the algorithm $EfGCheck$ for the procedure $check(E_{fair}G(\phi_1), f, \vec{k}, s)$. The same method given in [7] is adopted for this algorithm.

The $check$ procedure given in table I first verifies if the state s has already been labeled with $(\phi, f, \vec{k})$ or with $(\neg\phi, f, \vec{k})$ where ϕ, f, $\vec{k}$ are the parameters for this invocation of check procedure and returns appropriate truth condition. Subsequently, $check$ works inductively on the structure of ϕ. If ϕ is $\phi_1 \wedge \phi_2$ then $check$ is invoked on ϕ_1 first and, if ϕ_1 holds, on ϕ_2 using the evaluations f', f'' respectively; here f', f'' are restrictions of f to the free variables of ϕ_1 and ϕ_2 respectively. The cases when $\phi = EG(\phi_1)$ and $\phi = E(\phi_1 U \phi_2)$ are handled by procedure $EGCheck$ and $EUCheck$ respectively. The case when $\phi = (COUNT(i, M, \phi_1) = c)$ is handled as explained at the beginning of the section. Formulas, such as $(COUNT(i, M, \phi_1) = COUNT(i, M, \phi_2))$, can be handled similarly with evaluation f restricted to f' and f'' accordingly as in the case of $\phi_1 \wedge \phi_2$.

Given a GQS and a CTL formula ϕ, check if $s \models \phi$

TABLE I

CHECK PROCEDURE

Algorithm $check\ (\phi, f, \vec{k}, s)$
1. If $(\phi, f, \vec{k}) \in L(s)$, then return true
2. If $(\neg\phi, f, \vec{k}) \in L(s)$, then return false
3. Switch(ϕ)
- case ϕ is an atomic formula:
  - if s satisfi es $\phi[f]$
    - then return true,
    - else return false;
  - break
- case $\phi = \neg\phi_1$:
  - $flag \leftarrow \neg check(\phi_1, f, \vec{k}, s)$;
  - break
- case $\phi = (\phi_1 \wedge \phi_2)$:
  - $flag \leftarrow check(\phi_1, f', \vec{k}, s) \wedge$ $check(\phi_2, f'', \vec{k}, s)$;
  - break
- case $\phi = EX\phi_1$:
  - if there is at least one edge from s
  - $(s \xrightarrow{\pi, e(\vec{c})} t)$ such that
    - $(s, \pi(t)) \models e(\vec{k}/\vec{c})$ and
    - $check(\pi^{-1}(\phi_1), \pi^{-1}(f), \pi^{-1}(\vec{k}), t)$
    - then $flag \leftarrow true$
    - else $flag \leftarrow false$
  - break
- case $\phi = EG(\phi_1)$:
  - $flag \leftarrow EGcheck(s, EG(\phi_1), f, \vec{k})$;
- case $\phi = E_{fair}G(\phi_1)$:
  - $flag \leftarrow EfGcheck(s, EG(\phi_1), f, \vec{k})$;
  - break
- case $\phi = E(\phi_1 U \phi_2)$:
  - $flag \leftarrow EUcheck(s, E(\phi_1 U \phi_2), f, \vec{k})$;
  - break
- case ϕ is $(COUNT(i, \phi_1) = c)$:
  - $sum \leftarrow 0$;
  - for every equivalence class x of $\approx_{s,f}$
    - if $\exists j \in x$ such that
    - $(\neg\phi_1, f_j, \vec{k}) \in L(s)$
      - then continue;
    - if $\exists j \in x$ such that
    - $(\phi_1, f_j, \vec{k}) \in L(s)$,
      - then $sum \leftarrow sum + |x|$,
      - continue;
    - choose some $j \in x$
    - if $Check(\phi_1, f_j, \vec{k}, s)$
      - then $sum \leftarrow sum + |x|$;
  - if $sum = c$
    - then $flag \leftarrow True$;
    - else $flag \leftarrow False$;
  - break;
4. If $flag$, then add $(\phi, f, \vec{k})$ to $L(s)$, return true
5. If $\neg flag$, then add $(\neg\phi, f, \vec{k})$ to $L(s)$, return false

In the procedures $EUCheck$ given in table II, the $GQS(H, \mathcal{H}, G)$ is traversed appropriately. In the beginning of $EUCheck$, a new mark is stored with s to indicate that $EUCheck$ has been invoked on s with the input parameters. In the for loop of the procedure $EUCheck$, if $(\pi^{-1}(f), \pi^{-1}(\vec{k}), \pi^{-1}(\phi)) \in marked(t)$ but $(\pi^{-1}(\phi), \pi^{-1}(f), \pi^{-1}(\vec{k})) \notin L(t)$ implies that either there is a cycle and the eventuality is not fulfilled, or the sub-formula has been checked and has been found to be not satisfied. $EGCheck$ given table III is similar as $EUCheck$ and thus

can be easily understood.

Procedure $efpCheck$ is given in table IV. It is based on the standard algorithm which finds out the maximum strongly connected components in a directed graph. Like $EUcheck$ procedure, a new mark is created and stored with s when $efpCheck$ is invoked on s for first time with the parameters. For each state s on which $efpCheck$ is invoked, it is associated with a $partition$ array. After the mark is created, this array is initialized such that $s.partition[i].enabled$ equals to $True$ if process i has a transition enabled in s. Otherwise, $s.partition[i].enabled$ is initialized to $False$. For all processes i, $s.partition[i].executed$ is initialized to $False$. For each edge $s \xrightarrow{\pi,e(\vec{c})} t$ from s, it is classified either as a non-tree-edge or a tree-edge according to the method in [7]. If $s \xrightarrow{\pi,e(\vec{c})} t$ is a tree-edge, $efpCheck$ is invoked on t and $s.partition$ is updated accordingly after that. If $s \xrightarrow{\pi,e(\vec{c})} t$ is a non-tree-edge, $s.partition$ is updated according to the edge. After all edges $s \xrightarrow{\pi,e(\vec{c})} t$ from s have been checked and there is no fair path found from t, $s.partition$ is examined to determine if there is a fair path from s and the procedure returns accordingly.

Algorithm $EfGCheck(s, EfG(\phi_1), f, \vec{k})$ can be adopted from $efpCheck$ with some minor changes. It can be looked as applying $efpCheck$ over a reduced graph of G where each state satisfies ϕ_1. This reduced graph need not to be constructed explicitly. Instead, it can be constructed implicitly by avoiding those states that do not satisfy ϕ_1. This algorithm is omitted from this paper due to its similarity to $efpCheck$.

TABLE II
EUCHECK ALGORITHM

Algorithm $EUCheck(s, E(\phi_1 U \phi_2), f, \vec{k})$
- $\phi \leftarrow E(\phi_1 U \phi_2)$;
- add $(f, \vec{k}, \phi)$ to $marked(s)$;
- if $check(\phi_2, f, \vec{k}, s)$, then return true;
- if $\neg check(\phi_1, f, \vec{k}, s)$, then return false;
- for each edge from s ($s \xrightarrow{\pi,e(\vec{c})} t$) where $(s, \pi(t)) \models e(\vec{k}/\vec{c})$
  - if $(\pi^{-1}(f), \pi^{-1}(\vec{k}), \phi) \in marked(t)$ and $(\phi, \pi^{-1}(f), \pi^{-1}(\vec{k})) \in L(t)$ then return true;
  - if $(\pi^{-1}(f), \pi^{-1}(\vec{k}), \phi) \notin marked(t)$ then flag $\leftarrow$ $check(\phi, \pi^{-1}(f), \pi^{-1}(\vec{k}), t)$; if flag then return true
- return false

The worst case complexity of the algorithm can be shown to be $O(N \cdot |f| \cdot c^n)$ where N is the number of nodes plus edges in $GQS(H, \mathcal{H}, G)$, n is the number of processes and c is the depth of nesting of process quantifiers plus number of distinct proceed ids appearing in the edge conditions. This worst case complexity assumes that there is no state symmetry at all. If there is state symmetry then this would perform much better.

V. EXAMPLE

The mutual exclusion protocol given in table V, as an example, will be used to illustrate the concepts and structures in this paper. The example is given by extending the input language in [12]. The extension allows multiple priority specifications

TABLE III
EGCHECK ALGORITHM

Algorithm $EGCheck(s, EG(\phi_1), f, \vec{k})$
- $\phi \leftarrow EG(\phi_1)$;
- add $(f, \vec{k}, \phi)$ to $marked(s)$;
- if $\neg check(\phi_1, f, \vec{k}, s)$, then return false;
- for each edge from s ($s \xrightarrow{\pi,e(\vec{c})} t$) where $(s, \pi(t)) \models e(\vec{k}/\vec{c})$
  - if $(\pi^{-1}(f), \pi^{-1}(\vec{k}), \phi) \in marked(t)$ and $(\phi, \pi^{-1}(f), \pi^{-1}(\vec{k})) \in L(t)$ then return true;
  - if $(\pi^{-1}(f), \pi^{-1}(\vec{k}), \phi) \notin marked(t)$ then flag $\leftarrow$ $check(\phi, \pi^{-1}(f), \pi^{-1}(\vec{k}), t)$; if flag then return true
- return false

TABLE IV
EFPCHECK ALGORITHM

Algorithm $efpCheck(s, Exist\_Fair\_Path, f, \vec{k})$
- add $(f, \vec{k}, Exist\_Fair\_Path)$ to $marked(s)$ true;
- Initialize $s.partition$;
- for each edge from s ($s \xrightarrow{\pi,e(\vec{c})} t$) where $(s, \pi(t)) \models e(\vec{k}/\vec{c})$
  - if $(\pi^{-1}(f), \pi^{-1}(\vec{k}), Exist\_Fair\_Path) \in marked(t)$ and $(Exist\_Fair\_Path, \pi^{-1}(f), \pi^{-1}(\vec{k})) \in L(t)$ then return true;
  - if $(\pi^{-1}(f), \pi^{-1}(\vec{k}), efp) \notin marked(t)$ and $s \xrightarrow{\pi,e(\vec{c})} t$ is non-tree-edge
    - Construct $partition_e$ for this edge;
    - Combine $s.partition$ with $partition\_e$;
    - Update $s.partition$'s execution bits;
  - if $(\pi^{-1}(f), \pi^{-1}(\vec{k}), Exist\_Fair\_Path) \notin marked(t)$ and $s \xrightarrow{\pi,e(\vec{c})} t$ is tree-edge
    - if ($efpCheck(Exist\_Fair\_Path, \pi^{-1}(f), \pi^{-1}(\vec{k}), t)$) return true;
    - else Combine $t.partition$ with $s.partition$; Update $s.partition$'s execution bits;
- if $s.partition$ indicates there is a fair path from s return true;
- else return false;

to be specified with transitions in the same process module. The formula to be check is specified with $CCTL$.

The mutual exclusion protocol consists of a controller process module and a client process module. The controller process module in this example has only one controller process which controls the resource allocation such that only one client process can hold the resource at a time. The client process module consists of several client processes. The client processes request for the resource through request channel (implemented with shared variables $request[controller, client]$). The controller process acknowledges one request from the process with highest priority through reply channel (implemented with shared variables $reply[controller, client]$) if the resource is available. The client process to which the resource is granted changes its status ($lk[k] = 2$) to hold the resource. It then releases the resource ($buzy[cl] = 0$) and changes back its status ($lk[k] = 0$).

The $CCTL$ formula given at the end of the input program asserts that no two client processes can hold the resource at

the same time. Note that the universal quantifier used in the $CCTL$ formula is a short-cut defined in section III.

AQS is constructed first by ignoring the priority specifications on transitions; it then adds edge conditions to the AQS to reflect the priorities and obtains the GQS. The $CCTL$ property is checked inductively against the GQS. During checking the property, quantifier elimination described in section IV is employed to evaluate $COUNT$ term. For example, $check$ procedure is invoked on the initial state s_0 to evaluate $\forall i \in client\, \forall j \in client(i \neq j \rightarrow AG(lk[i] \neq 2 \vee lk[j] \neq 2))$. Since no client process has requested for the resource at s_0, $Aut(s_0)$ consists of all the permutations over process ids of client processes. According to the definition 2, all client processes form an equivalence class. Instead of checking $\forall j \in client(i \neq j \rightarrow AG(lk[i] \neq 2 \vee lk[j] \neq 2))$ with i instantiated to every client process, we choose an arbitrary client process k as representative for all client processes and check $\forall j \in client(i \neq j \rightarrow AG(lk[i] \neq 2 \vee lk[j] \neq 2))$ with an evaluation which instantiates i with k.

TABLE V
MUTUAL EXCLUSION PROTOCOL

```
Program

Module controller = 1;
Module client = 13;

lc[controller]=0;
lk[client]=0;
request[controller, client]=0;
reply[controller, client]=0;
buzy[controller]=0;

i of controller;

PriorityClass pclass1:client = (0);
PriorityClass pclass2:client = (1-12);

cl of controller :
{
lc[cl] == 0 &  request[cl, k] == 1 &
     ALL(i: reply[i,k] == 0) ->
 reply[cl, k] = 1,
 buzy[cl] = 1 , lc[cl] = 1
(Priority pclass1:pclass2);

lc[cl] == 1 & buzy[cl] == 0  ->
 lc[cl] = 0 ;
}

k of client :
{
lk[k] == 0  ->
 ALL(cl: request[cl, k] = 1) , lk[k] = 1;

lk[k] == 1 & reply[cl, k] == 1 ->
 lk[k] = 2;

lk[k] == 2 & reply[cl, k] == 1 ->
 reply[cl, k] = 0, ALL(i: request[i, k] = 0),
 buzy[cl] = 0 , lk[k] = 0;
}

Evaluation

Formula
```

$\forall i \in client\, \forall j \in client(i \neq j \rightarrow AG(lk[i] \neq 2 \vee lk[j] \neq 2))$

VI. IMPLEMENTATION

We have implemented the model-checking algorithm as the extension of the SMC model-checker [10]. This tool has been applied to the mutual exclusion protocol as well as industry level protocol such as cache coherency protocol. We observed significant performance improvement when checking some useful properties over these protocols.

We check the mutual exclusion property $\forall i \in client\, \forall j \in client(i \neq j \rightarrow AG(lk[i] \neq 2 \vee lk[j] \neq 2))$ with mutual exclusion protocol. Here $lk[j] \neq 2$ denotes that client j is not in critical section. For cache coherence protocol, we check the property $\forall i \in client\, \forall j \in client(i \neq j \rightarrow AG(cache[i] \neq exclusive \vee cache[j] \neq exclusive))$ asserting that no two clients can hold the cache line in exclusive mode simultaneously. Here $cache[i] \neq exclusive$ denotes that client i does not hold the cache line exclusively. The experimental results are presented in table VI. The column $quant\_elim$ indicates if our approach of quantifier elimination through state symmetry is employed. Without quantifier elimination, the protocols are checked in the naive approach. $mark\#$ gives the number of marks generated in the experiments. Recall that a mark is generated when $EUCheck$ or $EGCheck$ is invoked for the first time on a state with the parameters. The running time in column $time(s)$ is given by running the experiments on a Intel Pentium M 1.3G PC. In some experiments, we encounter stack overflow. This is indicated with * in the table. Our tool shows performance improvement for both cases. While with cache coherence protocol of 4 clients, our tool runs 20-30 percentage faster by utilizing quantifier elimination through state symmetry, we got much more performance improvement with mutual exclusion protocol of more than 10 clients. This shows that this tool is especially useful when verifying properties with quantifiers over a large set of processes.

TABLE VI
EXPERIMENT RESULTS

protocol	client#	quant_elim	mark#	time(s)
Mutual Exclusion	10	yes	208	0.02
	10	no	3780	1.6
	20	yes	448	0.12
	20	no	*	*
Cache Coherence	4	yes	96712	5.7
	4	no	115344	6.9

VII. CONCLUSION AND RELATED WORK

We have presented algorithm to check $CCTL$ formula using GQS without unwinding it completely. The algorithm exploits state symmetries. We used state symmetries earlier [7], [3] to reduce memory requirements. Here, for the first time, we use them to model-check for complex properties, using the $COUNT$ functions and process quantifiers, efficiently.

The algorithm uses formula decomposition and sub-formula tracking naturally and implicitly. The formula decomposition is used in the sense that when we invoke the $check$ procedure on a sub-formula ϕ we only track the process ids required for it. Similarly sub-formula tracking is used implicitly.

REFERENCES

[1] Clarke, E. M., Filkorn, T., Jha, S.: *Exploiting Symmetry in Temporal Logic Model Checking.* CAV93, LNCS **697** Springer-Verlag, 1993.

[2] Browne, M., Clarke, E. M., Grumberg, O, *Reasoning about networks with many identical finite-state processes*, Inf. Comput., 1989, (Vol. 81), 13–31.

[3] Emerson, E. A., Sistla, A. P.: *Symmetry and Model Checking.* CAV93, LNCS **697** Springer-Verlag, 1993; journal version appeared in Formal Methods in System Design, 9(1/2),1996, pp 105-130.

[4] Emerson, E. A., Sistla, A. P.: *Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata-theoretic Approach.* CAV95, LNCS **939** Springer-Verlag, 1995.

[5] Emerson E. A., Trefler R., *From Symmetry to Asymmetry: New techniques for Symmetry Reduction in Model-checking*, Proc. of CHARME 1999.

[6] Emerson E. A., Havlicek J. W., *Virtual Symmetry Reductions*, Proc. of LICS 2000.

[7] Gyuris, V., Sistla, A. P.: *On-the-Fly Model Checking under Fairness that Exploits Symmetry.* CAV97, LNCS **1254** Springer-Verlag, 1997.

[8] Ip, C. N., Dill, D. L.: *Better Verification through Symmetry.* Formal Methods in System Design **9** 1/2, pp41–75, 1996.

[9] Sistla A. P., Godefroid P.,*Symmetry and Reduced Symmetry in Model Checking*,CAV01, LNCS **2102** Springer-Verlag, 2001.

[10] Sistla A. P., Gyuris V., Emerson E. A., *SMC: A Symmetry based Model Checker for Verification of Safety and Liveness Properties*, ACM Transactions on Software Engineering Methodologies, Vol 9, No 2, pp 133-166, April 2000.

[11] E. Allen Emerson, Chin-Laung Lei, *Temporal Reasoning Under Generalized Fairness Constraints* , STACS**1986** , pp21-36

[12] Sistla A. P., Godefroid P.,*Symmetry and Reduced Symmetry in Model Checking* , To appear in TOPLAS.