Symmetry and Reduced Symmetry In Model Checking

A. PRASAD SISTLA
University of Illinois at Chicago
and
PATRICE GODEFROID
Bell Laboratories, Lucent Technologies

Symmetry reduction methods exploit symmetry in a system in order to efficiently verify its temporal properties. Two problems may prevent the use of symmetry reduction in practice: (1) the property to be checked may distinguish symmetric states and hence not be preserved by the symmetry, and (2) the system may exhibit little or no symmetry. In this article, we present a general framework that addresses both of these problems. We introduce "Guarded Annotated Quotient Structures" for compactly representing the state space of systems even when those are asymmetric. We then present algorithms for checking any temporal property on such representations, including non-symmetric properties.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods; model checking*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*mechanical verification*

General Terms: Verification, Reliability, Theory

Additional Key Words and Phrases: State space explosion, symmetry reductions, model checking algorithms and tools, temporal logics, formula decomposition

1. INTRODUCTION

In the last few years, there has been much interest in symmetry-based reduction methods for model checking concurrent systems [Ip and Dill 1993; Clarke et al. 1993; Emerson and Sistla 1996, 1997; Gyuris and Sistla 1999; Clarke and Jha 1995]. These methods exploit automorphisms of the system's global

A. P. Sistla's work was supported in part by the National Science Foundation (NSF) grants CCR-9988884 and CCR-0205365. A. P. Sistla's work was partly done while visiting Bell Laboratories.
P. Godefroid's work was supported in part by NSF grant CCR-0341658.
Author's address: A. P. Sistla, Department of computer Science, University of Illinois at Chicago, Chicago, IL 60607; email: sistla@eecs.uic.edu; P. Godefroid, Bell Laboratories, Lucent Technologies, Lisle, IL 60532.

state graph induced by permutations on process indices and variables. Given a correctness property specified by a temporal formula ϕ, existing symmetry-reduction methods for verifying ϕ can be broadly classified into two categories: the first class of methods [Clarke et al. 1993; Emerson and Sistla 1996; Ip and Dill 1993; Clarke and Jha 1995] consider only those automorphisms that preserve the atomic predicates appearing in ϕ, construct a *Quotient Structure* (*QS*), and check the formula ϕ on the *QS* using traditional model-checking algorithms; the second class of methods [Emerson and Sistla 1997] consider all automorphisms induced by process/variable permutations, construct an *Annotated Quotient Structure* (*AQS*), and unwind it to verify the formula ϕ.

In this article, we generalize symmetry-based reduction in several ways. First, the mathematical framework used to formalize symmetry reduction supports *any* automorphism on the system's state graph; for example, automorphisms induced by permutations on variable-value pairs can be considered in addition to those induced by permutations on process indices and variables. Thus, this framework allows for more automorphisms and hence greater reduction.

Second, we introduce the notion of *Guarded Annotated Quotient Structure* (*GQS*) to represent, in a very compact way, the state graph of systems with little or even no symmetry. In a nutshell, a *GQS* is an *AQS* whose edges are also associated with a guard representing the condition under which the corresponding original program transition is executable. Given a program P and its reachability graph G, by adding edges to G (via a transformation of P), we obtain an expanded graph H that has more symmetry than G, and hence can be represented more compactly. A *GQS* for G can be viewed as an *AQS* for H whose edges are labeled with guards in such a way that the original edges of G can be recovered from the representation of H. To verify a temporal formula ϕ, the *GQS* is unwound as needed, by tracking the values of the atomic predicates in ϕ and the guards of the *GQS*, so that only edges in G are considered. The *GQS* of G can be much smaller than its *QS* because it is defined from a larger set of automorphisms: a *GQS* is derived by considering all the automorphisms of H, which exhibits more symmetry than G, including those automorphisms that do not preserve the atomic predicates in ϕ. We show that unwinding the *GQS* on-demand, in order to verify a property ϕ, can be done without ever generating a structure larger than the corresponding *QS*.

Third, we present two new techniques for further optimizing the model-checking procedure using *GQS*s. These techniques minimize the amount of unwinding necessary to check a formula ϕ and may yield an exponential improvement in performance. The first technique, called *formula decomposition*, consists of decomposing ϕ into sets of top-level subformulas that contain correlated atomic predicates; the satisfaction of ϕ can then be checked by checking each set of subformulas separately, which in turn can be done by successively unwinding the *GQS* with respect to only the predicates appearing in the subformulas of that set separately; therefore, unwinding the *GQS* with respect to all the atomic predicates appearing in ϕ simultaneously can be avoided. The second technique, called *subformula tracking*, consists of identifying a good maximal

set of "independent" subformulas of ϕ and unwinding the GQS by tracking these subformulas only. A good maximal independent set contains symmetric or near symmetric subformulas. Thus, subformula tracking allows us to exploit symmetries in the subformulas to contain the size of the unwound structure. This is a generalization of formula symmetry, originally considered in Emerson and Sistla [1996], and it allows us to exploit symmetries in the formula more widely. Formula decomposition and subformula tracking are complementary techniques and can be applied recursively.

We also present an alternative method that does not construct the GQS. Instead, it simply constructs the AQS for the expanded graph H as defined above, and modifies the correctness property to include the guards associated with the edges of the GQS. The modified correctness property asserts that the original correctness property be satisfied on those paths of H that are also paths in G. Note that formula decomposition can also be used with this method, as well as with the traditional method employing the QS.

We call the method involving the construction of the QS from the original graph G the QS-based method and, similarly, the methods involving the construction of the GQS and the AQS from the expanded graph H are called the GQS-based and the AQS-based methods, respectively. We present experimental results comparing these three methods. To do this, we extended the SMC model-checker [Sistla et al. 2000] to handle priorities by using the GQS-based method. Our experiments are performed using a standard resource-controller example and a real-world example, the Fire-wire protocol [IEEE 1995]. These experiments show that the GQS-based method outperforms the other two methods most of the time. They also show that formula decomposition can improve performance significantly.

This article is organized as follows. Section 2 introduces background definitions and notations. Section 3 describes the GQS-based method and the model-checking procedure using it. Section 4 presents the techniques based on formula decomposition and sub-formula tracking. Section 5 describes the extension of the SMC system to handle priorities and presents experimental results comparing the three methods mentioned above. Section 6 contains concluding remarks and discusses related work.

2. BACKGROUND

A Kripke structure K is a tuple $(S, E, \mathcal{P}, L)$ where S is a set of states, $E \subseteq S \times S$ is a set of edges, $\mathcal{P}$ is a set of atomic propositions and $L: S \rightarrow 2^{\mathcal{P}}$ is a function that associates a subset of $\mathcal{P}$ with each state in S. CTL^* is a logic for specifying temporal properties of concurrent programs (e.g., see [Emerson 1990]). It includes the temporal operators U (until), X (next-time) and the existential path quantifier E. Two types of CTL^* formulas are defined inductively: path formulas and state formulas. Every atomic proposition is a state formula as well as a path formula. If p and q are state formulas (respectively, path formulas) then $p \wedge q$ and $\neg p$ are also state formulas (respectively, path formulas). If p and q are path formulas then $p\mathsf{U}q$, $\mathsf{X}p$ are path formulas and $\mathsf{E}(p)$ is a state formula. Every state formula is also a path formula. We use the abbreviation $\mathsf{EF}(p)$ for

$\mathsf{E}(True\mathsf{U}p)$ and $\mathsf{AG}(p)$ for $\neg(\mathsf{EF}\neg p)$. A CTL^* formula is a state formula. CTL is the fragment of CTL^* where all path formulas are of the form $p\mathsf{U}q$ or of the form $\mathsf{X}p$ where p, q are state formulas. CTL^* formulas are interpreted over Kripke structures (e.g, see Emerson [1990] for a detailed presentation of the semantics of CTL^*). The nesting depth of a CTL^* formula is the nesting depth of the path quantifiers within the formula. For example, the nesting depth of the formula $E(P\mathsf{U}Q) \wedge EF(Q)$ is one, while that of $E(P\mathsf{U}E(Q\mathsf{U}R))$ is two.

Let $K = (S, R, \mathcal{P}, L)$ and $K' = (S', R', \mathcal{P}, L')$ be two Kripke structures with the same set of atomic propositions. A bisimulation between K and K' is a binary relation $U \subseteq S \times S'$ such that, for every $(s, s') \in U$, the following conditions are all satisfied: (1) $L(s) = L'(s')$; (2) for every t such that $(s, t) \in R$, there exists $t' \in S'$ such that $(t, t') \in U$ and $(s', t') \in R'$; and (3) for every t' such that $(s', t') \in R'$, there exists $t \in S$ such that $(t, t') \in U$ and $(s, t) \in R$. We say that a state $s \in S$ is bisimilar to a state $s' \in S'$ if there exists a bisimulation U between K and K' such that $(s, s') \in U$. It is well known that bisimilar states satisfy the same CTL^* formulas.

We define a predicate over a set S as a subset of S. Let f be a bijection on S, that is, a one-to-one mapping from S to S. Let C be a predicate over S. Let $f(C)$ denote the set $\{f(x) : x \in C\}$. Let f^{-1} denote the inverse of the bijection f. If f, g are two bijections, then we let fg denote their composition in that order; note that fg is also a bijection. Throughout the article, we use the following identity relating the inverse and composition operators: $(fg)^{-1} = g^{-1}f^{-1}$.

Let $G = (S, E)$ be the reachability graph of a concurrent program where S denotes a set of nodes/states and $E \subseteq S \times S$. An automorphism f of G is a bijection on S such that, for all $s, t \in S$, $(s, t) \in E$ iff $(f(s), f(t)) \in E$. We say that an automorphism f respects a predicate C over S if $f(C) = C$. The set of all automorphisms of a graph forms a group *Aut(G)*. Given a set $P_1, \ldots, P_k$ of predicates over S, the set of automorphisms of G that respect $P_1, \ldots, P_k$ form a subgroup of *Aut(G)*.

Let $\mathcal{G}$ be a group of automorphisms of G. We say that states $s, t \in S$ are equivalent, denoted by $s \equiv_{\mathcal{G}} t$, if there exists some $f \in \mathcal{G}$ such that $t = f(s)$. As observed in Clarke et al. [1993], Emerson and Sistla [1996], and Ip and Dill [1993], $\equiv_{\mathcal{G}}$ is an equivalence relation. A *quotient structure* of G with respect to $\mathcal{G}$ is a graph $(\bar{S}, \bar{E})$ where $\bar{S}$ contains exactly one node in each equivalence class of $\equiv_{\mathcal{G}}$ and $(\bar{s}, \bar{t}) \in \bar{E}$ iff there exists some t such that $t \equiv_{\mathcal{G}} \bar{t}$ and $(\bar{s}, t) \in E$. Each state $\bar{s} \in \bar{S}$ represents all states in S that belong to its equivalence class. Different quotient structures can be defined by choosing different representatives for each equivalence class. However, all these structures are isomorphic. We denote by $rep(s, \mathcal{G})$ the representative element of the equivalence class to which s belongs. In what follows, $QS(G, \mathcal{G})$ denotes the quotient structure obtained by choosing a unique representative for each equivalence class.

A predicate P on the edges of G is a subset of $S \times S$. We say that an edge (s, t) in E, satisfies P if $(s, t) \in P$. Let *True* denote the set $S \times S$. For an edge predicate P and automorphism f on states, let $f(P) = \{(f(s), f(t)) : (s, t) \in P\}$. Given a group $\mathcal{G}$ of automorphisms of G, we can extend the equivalence relation $\equiv_{\mathcal{G}}$ from states in S to edges in E as follows: two edges $e = (s, t)$ and $e' = (s', t')$

are equivalent (written as $e \equiv_{\mathcal{G}} e'$) if there exists some $g \in \mathcal{G}$ such that $s' = g(s)$ and $t' = g(t)$. It is easy to see that $\equiv_{\mathcal{G}}$ on E is an equivalence relation [Godefroid 1999].

3. MODEL CHECKING USING GUARDED ANNOTATED QUOTIENT STRUCTURES

3.1 Guarded Quotient Structures

In this section, we introduce Guarded Annotated Quotient Structures (*GQS*s) as extensions of Annotated Quotient Structures considered in Emerson and Sistla [1996, 1997]. These structures can be defined with respect to arbitrary automorphisms and can compactly represent the state space of systems that contain little symmetry. For example, consider a resource allocation system composed of a resource controller and three identical user processes, named a, b and c. When multiple user processes request the resource at the same time, the controller process allocates it to one of the requesting users according to the following priority scheme: user a is given highest priority while users b and c have the same lower priority. This system exhibits some symmetry since users b and c are "interchangeable". Now consider a similar system but where the three user processes are given equal priority. This system exhibits more symmetry since all three users are now "interchangeable". Thus, the system without priorities has more symmetry than the system with priorities. A guarded annotated quotient structure allows us to verify systems with reduced symmetry (e.g., a system with priorities) by treating these as if they had more symmetry (e.g., a system without priorities) and without compromising the accuracy of the verification results. For instance, in the state graph G of the above resource allocation system with priorities, a state s where all three users have requested the resource has only one outgoing edge (granting the resource to user a). By adding two other edges from s (granting the resource to the two other user processes), the state graph H of the system without priorities can be defined. Since H exhibits more symmetry than G, it can be verified more efficiently using symmetry reduction. Thus, by viewing G as H extended with guards so that G can be re-generated if needed, model checking can be done more efficiently.

Formally, let $H = (S, F)$ be a graph such that $F \supseteq E$ and $Aut(G) \subseteq Aut(H)$, that is, H is obtained by adding edges to $G = (S, E)$ such that every automorphism of G is also an automorphism of H.[1] Let $\mathcal{H}, \mathcal{G}$ be groups of automorphisms of H and G, respectively, such that $\mathcal{H} \supseteq \mathcal{G}$. As indicated earlier, $\equiv_{\mathcal{H}}$ defines equivalence relations on the nodes and edges of H. For any edge $e \in F$, let $Class(e, \mathcal{H})$ denote the set of edges in the equivalence class of e defined by $\equiv_{\mathcal{H}}$. Let $\mathcal{Q} = \{Q_1, \ldots, Q_l\}$ be a set of predicates on S such that each automorphism in $\mathcal{G}$ respects all the predicates in $\mathcal{Q}$. Let $QS(G, \mathcal{G}) = (\bar{U}, \bar{E})$ be the quotient structure of G with respect to $\mathcal{G}$ as defined earlier.

[1] Our results can easily be extended to allow the addition of nodes as well as edges. Note that adding edges/nodes to a graph may sometimes reduce symmetry.

A *Guarded Annotated Quotient Structure* of $H = (S, F)$ with respect to $\mathcal{H}$, denoted by $GQS(H, \mathcal{H})$, is a triple $(\bar{V}, \bar{F}, C)$ where $\bar{V} \subseteq S$ is a set of states that contains one representative for each equivalence class of states defined by $\equiv_{\mathcal{H}}$ on S, $\bar{F} \subseteq \bar{V} \times \bar{V} \times \mathcal{H}$ is a set of labeled edges such that, for every $\bar{s} \in \bar{V}$ and $t \in S$ such that $(\bar{s}, t) \in F$, there exists an element $(\bar{s}, \bar{t}, f) \in \bar{F}$ such that $f(\bar{t}) = t$, and C is a function that associates an edge predicate $C(e)$ with each labeled edge $e \in \bar{F}$ satisfying the two following conditions: (1) let $e = (\bar{s}, \bar{t}, f)$ and $e' = (\bar{s}, f(\bar{t}))$; at the time when we unwind the GQS, we will need to unwind only the edges in $Class(e', \mathcal{H})$ that are edges in G; we chose to encode this information using the predicate $C(e)$ which we define as a predicate that contains all the edges of $Class(e', \mathcal{H})$ that are in E; formally, we thus require that $C(e)$ satisfies the condition: $C(e) \cap Class(e', \mathcal{H}) = E \cap Class(e', \mathcal{H})$; (2) for all $g \in \mathcal{G}$, $g(C(e)) = C(e)$ (i.e., g respects the edge predicate $C(e)$).

For a labeled edge $e = (\bar{s}, \bar{t}, f) \in \bar{F}$, let $Class(e, \mathcal{H})$ simply denote $Class(e', \mathcal{H})$ where $e' = (\bar{s}, f(\bar{t}))$. For the labeled edge $e = (\bar{s}, \bar{t}, f) \in \bar{F}$, $f \in \mathcal{H}$ is called the label of e and denotes an automorphism that can be used to obtain the corresponding original edge in F; the edge predicate $C(e)$ can in turn be used to determine whether this edge is also an edge of G. Labels of edges in $\bar{F}$ and the edge predicate C are used to unwind $GQS(H, \mathcal{H})$ when necessary during model checking, as described later. Note that edge predicates, given by C, that satisfy the above conditions always exist: for instance, taking $C(e) = E$ always satisfies the definition. In practice, a compact representation of an edge predicate C satisfying the conditions above can be obtained directly from the description of the concurrent program. For example, in the case of the prioritized resource allocation system, the edge predicate $C(e)$ is defined as follows: if the labeled edge e denotes the allocation of the resource to a user, then $C(e)$ asserts that if there is a request from user a then a is allocated the resource; for all other labeled edges, $C(e)$ is the predicate *True*. Similarly, the automorphisms labeling edges in $\bar{F}$ can also have succinct implicit representations. For example, any automorphism induced by permutations of n process indices as considered in Emerson and Sistla [1996, 1997], and Gyuris and Sistla [1999] can be represented by an array of n variables ranging over n. Tools like SMC [Sistla et al. 2000] and Murphi [Ip and Dill 1993] include optimized algorithms for representing and manipulating such sets of permutations.

Figure 1 shows the reachability graph G of a 2-process mutual exclusion algorithm where process 1 has 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 noncritical 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 Figure 2. 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

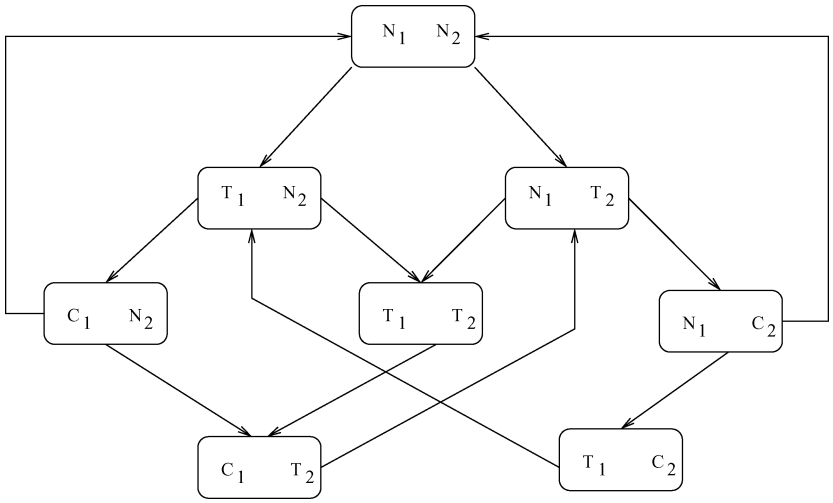

Fig. 1. Global transition graph.

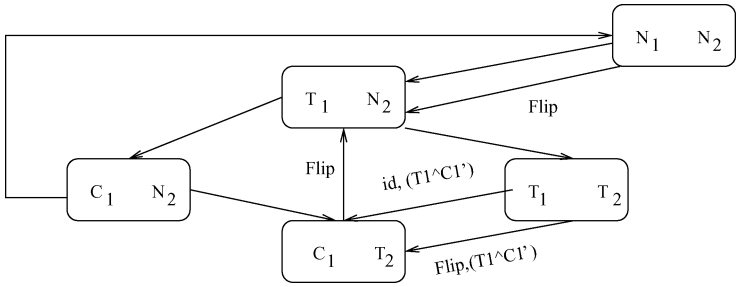

Fig. 2. Guarded annotated quotient structure.

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).

3.2 Relationship between the Different Structures

Given a set $\mathcal{Q}$ of predicates over S that are all respected by the automorphisms in $\mathcal{G}$, we have already define three Kripke structures $K\_Stru(G, \mathcal{Q})$, $QS\_Stru(G, \mathcal{G}, \mathcal{Q})$ and $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$ derived from $G = (S, E)$, $QS(G, \mathcal{G}) = (\bar{U}, \bar{E})$ and $GQS(H, \mathcal{H}) = (\bar{V}, \bar{F}, C)$, respectively. We show that *these three Kripke structures are pairwise bisimilar, and hence can all be used for CTL* model checking*. Since $\mathcal{G}$ is a subgroup of $\mathcal{H}$, each equivalence class of $\equiv_{\mathcal{H}}$ is a union of smaller equivalence classes defined by $\equiv_{\mathcal{G}}$. Thus, the number of equivalence classes of $\equiv_{\mathcal{H}}$ is smaller than those of $\equiv_{\mathcal{G}}$, and $GQS(H, \mathcal{H})$ contains (possibly

exponentially) fewer nodes than $QS(G, \mathcal{G})$. $QS(G, \mathcal{G})$ itself can be much smaller than G.

For each predicate Q_j $(1 \leq j \leq l)$ in $\mathcal{Q}$, we introduce an atomic proposition denoted q_j. Let $\mathcal{X} = \{q_i : 1 \leq i \leq l\}$. Let $K\_Stru(G, \mathcal{Q})$ denote the Kripke structure $(S, E, \mathcal{X}, L)$ where for any $s \in S$, $L(s) = \{q_j : s \in Q_j\}$. The Kripke structure $QS\_Stru(G, \mathcal{G}, \mathcal{Q})$ is given by $(\bar{U}, \bar{E}, \mathcal{X}, M)$ where $M(\bar{s}) = \{q_j : \bar{s} \in Q_j\}$. The following theorem has been proved in Emerson and Sistla [1996], Clarke et al. [1993], and Ip and Dill [1993].

THEOREM 1. *There exists a bisimulation between the structures $K\_Stru(G, \mathcal{Q})$ and $QS\_Stru(G, \mathcal{G}, \mathcal{Q})$ such that every state $s \in S$ is bisimilar to its representative in $\bar{U}$.*

Therefore, any CTL^* formula over atomic propositions in $\mathcal{X}$ is satisfied at a state s in $K\_Stru(G, \mathcal{Q})$ iff it is satisfied at its representative $rep(s, \mathcal{G})$ in $QS\_Stru(G, \mathcal{G}, \mathcal{Q})$.

If C in the structure $GQS(H, \mathcal{H}) = (\bar{V}, \bar{F}, C)$ is implicitly represented by a collection of edge predicates $\Theta_1, \ldots, \Theta_r$, the Kripke structure $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$ is obtained from $GQS(H, \mathcal{H})$ by partially unwinding it and by tracking the node predicates in $\mathcal{Q}$ (i.e., the predicates $Q_1, \ldots, Q_l$) and the edge predicates $\Theta_1, \ldots, \Theta_r$ during this unwinding process. In other words, the unwinding is performed with respect to the predicates $Q_1, \ldots, Q_l$ and $\Theta_1, \ldots, \Theta_r$, not with respect to the states of G, in order to limit the unwinding as much as possible. This partial unwinding is a generalization of the unwinding process described in [Emerson and Sistla 1996, 1997]. Precisely, the Kripke structure $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$ is the tuple $(W, T, \mathcal{X}, N)$ where W, T and N are defined as follows:

—For all $\bar{s} \in \bar{V}$, $(\bar{s}, Q_1, \ldots, Q_l, \Theta_1, \ldots, \Theta_r) \in W$.

—Let $u = (\bar{s}, X_1, \ldots, X_l, \Phi_1, \ldots, \Phi_r)$ be any state in W, $e = (\bar{s}, \bar{t}, f)$ be a labeled edge in $\bar{F}$ and j be the unique integer such that Θ_j is the edge predicate $C(e)$. If the edge $(\bar{s}, f(\bar{t}))$ satisfies the predicate Φ_j, the node $v = (\bar{t}, f^{-1}(X_1), \ldots, f^{-1}(X_l), f^{-1}(\Phi_1), \ldots, f^{-1}(\Phi_r))$ is in W and the edge (u, v) is in T.

—For all $u = (\bar{s}, X_1, \ldots, X_l, \Phi_1, \ldots, \Phi_r) \in W$, $N(u) = \{q_i : \bar{s} \in X_i\}$.

For the sake of brevity, many times we may only be referring to unwinding with respect to the predicates in $\mathcal{Q}$; however, in all these cases it should be understood that the unwinding is being done with respect to edge predicates also.

Consider the example of Figures 1 and 2. Let G be the graph shown in Figure 1, and H be its extension obtained by adding the edge from (T_1, T_2) to (T_1, C_2). $\mathcal{H}$ is the group $\{id, Flip\}$. The predicate $\Theta_1 = T_1 \wedge C_1'$ is the only edge predicate. Let $\mathcal{Q}$ be the set containing the two node predicates T_1-nodes and C_1-nodes, that is, $\Theta_1 = T_1$-nodes and $\Theta_2 = C_1$-nodes. We let $\mathcal{X} = \{T_1, C_1\}$, i.e., $q_1 = T_1$ and $q_2 = C_1$. Note that each node u in GQS_Stru is of the form $(\bar{s}, X_1, X_2, \Phi_1)$ where X_1, X_2 are the tracked values of the node predicates and Φ_1 is the tracked value of the edge predicate. Notice that in $GQS(H, \mathcal{H})$, there are two types of paths from the initial node (N_1, N_2) to all other nodes: those

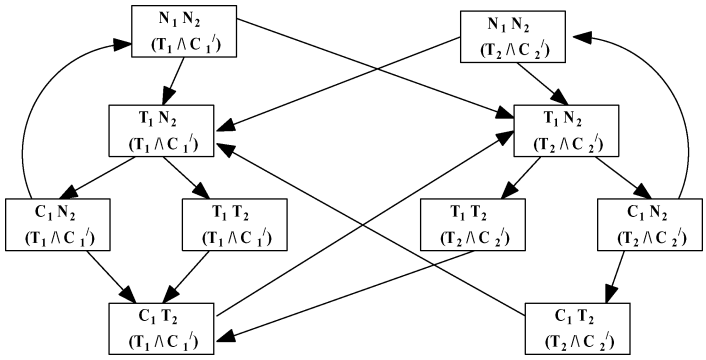

Fig. 3. Unwound guarded quotient structure.

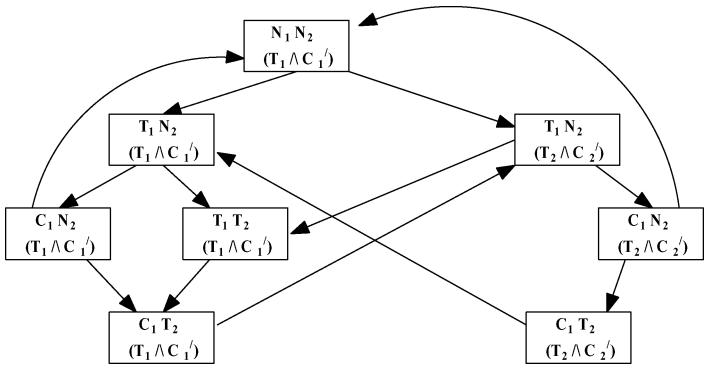

Fig. 4. Reduced structure.

with the product of all permutations on the edges being equal to id and *Flip*, respectively. As a consequence, for every node $\bar{s}$ in $GQS(H, \mathcal{H})$, there will be two nodes in $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$ given by $u_1 = (\bar{s}, T_1\text{-nodes}, C_1\text{-nodes}, T_1 \wedge C_1')$ and $u_2 = (\bar{s}, T_2\text{-nodes}, C_2\text{-nodes}, T_2 \wedge C_2')$. We call u_1, u_2 the $\bar{s}$-nodes. The labels $N(u_1)$ and $N(u_2)$ are defined as follows: $T_1 \in N(u_1)$ iff $T_1 \in \bar{s}$, $T_1 \in N(u_2)$ iff $T_2 \in \bar{s}$; similarly $C_1 \in N(u_1)$ iff $C_1 \in \bar{s}$, $C_1 \in N(u_2)$ iff $C_2 \in \bar{s}$. The $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$ is shown in Figure 3. In each node the tracked edge predicate is shown, but the tracked node predicates are not shown. Notice that each of the (T_1, T_2)-nodes has only one successor. Although $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$ has more nodes than G, it can be further reduced to a smaller structure $Reduced\_Stru(H, \mathcal{H}, \mathcal{Q})$, shown in Figure 4, defined later (see Theorem 3). Note that if we had considered a similar mutual exclusion example for three or more processes then the the size of the corresponding GQS_Stru would be much smaller than the reachability graph G.

The following theorem states that $QS\_Stru(G, \mathcal{G}, \mathcal{Q})$ and $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$ are bisimilar. Recall that $G = (S, E)$, $H = (S, F)$ where $F \supseteq E$, $\mathcal{G}$ is a group of automorphisms of G that respect all the predicates in $\mathcal{Q}$, and $\mathcal{H}$ is a group of automorphisms of H such that $\mathcal{H} \supseteq \mathcal{G}$. Further, $GQS(H, \mathcal{H}) = (\bar{V}, \bar{F}, C)$ and

for every $e \in \bar{F}$ all the automorphisms in $\mathcal{G}$ respect the edge predicate $C(e)$, $QS\_Stru(G, \mathcal{G}, \mathcal{Q}) = (\bar{U}, \bar{E}, \mathcal{X}, M)$ and $GQS\_Stru(H, \mathcal{H}, \mathcal{Q}) = (W, T, \mathcal{X}, N)$.

THEOREM 2. *Given $QS\_Stru(G, \mathcal{G}, \mathcal{Q})$ and $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$ as previously defined, let $Z = \{(s, u) : s \in \bar{U}, u \in W$ and there exists an automorphism $f \in \mathcal{H}$ such that $u = (f(s), f(Q_1), \ldots, f(Q_l), f(\Theta_1), \ldots, f(\Theta_r))\}$. Then, the following properties hold:*

(1) *Z is a bisimulation between $QS\_Stru(G, \mathcal{G}, \mathcal{Q})$ and $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$.*
(2) *If $\mathcal{G}$ is the maximal subgroup of $\mathcal{H}$ that respects $Q_1, \ldots, Q_l$, then no two nodes in $\bar{U}$ are related to the same node in W through Z.*
(3) *For all $u \in W$, there exists a node $s \in \bar{U}$ such that $(s, u) \in Z$.*
(4) *Two nodes $u = (t, X_1, \ldots, X_l, \Phi_1, \ldots, \Phi_r)$ and $u' = (t', Y_1, \ldots, Y_l, \Delta_1, \ldots, \Delta_r)$ of $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$ are related to a single node s of $QS\_Stru(G, \mathcal{G}, \mathcal{Q})$ through Z iff $t = t'$ and there exists some h in $\mathcal{H}$ such that $h(t) = t$ and $X_i = h(Y_i)$ for all $i = 1, \ldots, l$, and $\Phi_j = h(\Delta_j)$ for all $j = 1, \ldots, r$.*

PROOF. First, we show that Z is a bisimulation. Let $(s, u) \in Z$ where $u = (t, X_1, \ldots, X_l, \Phi_1, \ldots, \Phi_r)$. By the definition of Z, we know that there exists an automorphism $f \in \mathcal{H}$ such that $t = f(s)$ and $X_i = f(Q_i)$ for each $i = 1, \ldots, l$ and $\Phi_j = f(\Theta_j)$ for each $j = 1, \ldots, r$. For each $i = 1, \ldots, l$, $q_i \in N(u)$ iff $t \in X_i$ iff $t \in f(Q_i)$ iff $s = f^{-1}(t) \in Q_i$ iff $q_i \in M(s)$. From these observations, we see that $M(s) = N(u)$.

Now, suppose $(s, s') \in \bar{E}$, i.e., it is an edge in $QS\_Stru(G, \mathcal{G}, \mathcal{Q})$. We show that there exists u' such that $(s', u') \in Z$ and (u, u') is an edge in $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$, that is, $(u, u') \in T$. From the definition of $QS(G, \mathcal{G})$, it follows that there exists an automorphism $g \in \mathcal{G}$ such that $(s, g(s')) \in E$. Let e' denote the edge $(s, g(s'))$. Since $f \in \mathcal{H}$, $f(e') = (t, fg(s')) \in F$. From the construction of $GQS(H, \mathcal{H})$, we see that there exist $t' \in \bar{V}$ and $h \in \mathcal{H}$, such that $(t, t', h) \in \bar{F}$ and $h(t') = fg(s')$. Let e denote the labeled edge (t, t', h). Let j be the integer such that Θ_j is the edge predicate $C(e)$. Clearly, $e' \in Class(f(e'), \mathcal{H})$ and hence $e' \in E \cap Class(f(e'), \mathcal{H})$. From the definition of $GQS(H, \mathcal{H})$, we have $\Theta_j \cap Class(f(e'), \mathcal{H}) = E \cap Class(f(e'), \mathcal{H})$. Hence $e' \in \Theta_j$. From this we see that $f(e') = (t, h(t')) \in f(\Theta_j)$, i.e., $(t, h(t')) \in \Phi_j$. From the construction of $GQS\_Stru(G, \mathcal{H}, \mathcal{Q})$, we see that there is an edge from u to u' in T, i.e. $(u, u') \in T$, where $u' = (t', Y_1, \ldots, Y_l, \Delta_1, \ldots, \Delta_r)$ and $Y_i = h^{-1}(X_i)$ for $1 \leq i \leq l$ and $\Delta_j = h^{-1}(\Phi_j)$ for $1 \leq j \leq r$. Now we show that $(s', u') \in Z$. Let $h' = h^{-1} fg$. Also let k be any integer such that $1 \leq k \leq l$. Now, we have $h'(Q_k) = h^{-1} fg(Q_k)$. Since $g \in \mathcal{G}$ and g respects Q_k, it follows that $h'(Q_k) = h^{-1} f(Q_k)$. From the fact that $f(Q_k) = X_k$ and $h^{-1}(X_k) = Y_k$, we see that $h'(Q_k) = Y_k$. Thus, for each $k = 1, \ldots, l$, $h'(Q_k) = Y_k$. Using similar arguments and the fact that every automorphism in $\mathcal{G}$ respects the edge predicates $\Theta_1, \ldots, \Theta_r$, it can be shown that $h'(\Theta_i) = \Delta_i$ for each $i = 1, \ldots, r$. From this, we see that $(s', u') \in Z$.

Now, suppose (u, u') is any edge in $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$, that is, $(u, u') \in T$. We show that there exists an edge $(s, s') \in \bar{E}$ such that $(s', u') \in Z$. Assume that $u' = (t', Y_1, \ldots, Y_l, \Delta_1, \ldots, \Delta_r)$. From the construction of $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$, we see that there exists an automorphism $h \in \mathcal{H}$ such that the labeled edge $e = (t, t', h)$ is in $\bar{F}$ and the following properties are satisfied: $e' = (t, h(t')) \in \Phi_j$ where j

is the unique integer such that Θ_j is the edge predicate $C(e)$; $Y_i = h^{-1}(X_i)$ for $i = 1, \ldots, l$; $\Delta_k = h^{-1}(\Phi_k)$ for $k = 1, \ldots, r$. From the definition of $GQS(H, \mathcal{H})$, we see that $e' \in F$ and $f^{-1}(e') \in Class(e', \mathcal{H})$. Since $e' \in \Phi_j$, we see that $f^{-1}(e') \in f^{-1}(\Phi_j)$. Since $\Phi_j = f(\Theta_j)$, it follows that $f^{-1}(e') \in \Theta_j \cap Class(e', \mathcal{H})$. Since $\Theta_j \cap Class(e', \mathcal{H}) = E \cap Class(e', \mathcal{H})$, it follows that $f^{-1}(e') \in E$. From this observation and the facts that $t = f(s)$, $f^{-1}(e') = (f^{-1}(t), f^{-1}h(t'))$, it follows that $(s, f^{-1}h(t')) \in E$. From the construction of $QS\_Stru(G, \mathcal{G}, \mathcal{Q})$, we see that there exists $s' \in \bar{U}$ and $g \in \mathcal{G}$ such that $(s, s') \in \bar{E}$ and $g(s') = f^{-1}h(t')$. Now, we show that $(s', u') \in Z$. Let $h' = h^{-1}fg$. Clearly, $h'(s') = t'$. Let i be any integer such that $1 \leq i \leq l$. Now, $h'(Q_i) = h^{-1}fg(Q_i)$. Since g respects Q_i, we have $h'(Q_i) = h^{-1}f(Q_i)$. Since $f(Q_i) = X_i$ and $h^{-1}(X_i) = Y_i$, it follows that $(h')(Q_i) = Y_i$. Thus, for each $i = 1, \ldots, l$, we have $(h')(Q_i) = Y_i$. Using similar arguments as in the previous paragraph, it can be shown that $h'(\Theta_k) = \Delta_k$ for each $k = 1, \ldots, r$. From these arguments, we see that $(s', u') \in Z$. Hence, Z is a bisimulation.

Now we prove part (2) of the theorem by contradiction. First, assume that $\mathcal{G}$ is the maximal subgroup of $\mathcal{H}$ such that all automorphisms in it respect $Q_1, \ldots, Q_l$. Contrary to part (2) of the theorem, assume that there exist two distinct elements s, s' in $\bar{U}$ and an element $u \in W$ such that $(s, u) \in Z$ and $(s', u) \in Z$. Let $u = (t, X_1, \ldots, X_l, \Phi_1, \ldots, \Phi_r)$. From the definition of Z, we see that there exist $f, g \in \mathcal{H}$ satisfying the following properties: $t = f(s)$, $t = g(s')$ and $X_i = f(Q_i) = g(Q_i)$ for $i = 1, \ldots, l$. Now, it is easy to see that $s = f^{-1}g(s')$. Furthermore, for each $i = 1, \ldots, l$, $Q_i = f^{-1}g(Q_i)$. Hence, the automorphism $f^{-1}g$ respects $Q_1, \ldots, Q_l$. Clearly, $f^{-1}g \in \mathcal{H}$. Since $\mathcal{G}$ is the maximal subgroup of $\mathcal{H}$ that respect $Q_1, \ldots, Q_l$, it follows that $f^{-1}g \in \mathcal{G}$. Since $s = f^{-1}g(s')$, s and s' belong to the same equivalence class induced by $\equiv_{\mathcal{G}}$. This means that we have two representatives from the same equivalence class in $\bar{U}$. This contradicts our construction where we pick only one representative from each equivalence class.

Now, we prove part (3) of the theorem. Let $u = (t, X_1, \ldots, X_l, \Phi_1, \ldots, \Phi_r)$ be any node in W. From the construction, it is not difficult to see that there exists some $f \in \mathcal{H}$ such that $X_i = f^{-1}(Q_i)$ for each $i = 1, \ldots, l$, and $\Phi_j = f^{-1}(\Theta_j)$ for each $j = 1, \ldots, r$. Now consider $f(t)$. There exists a state $s \in \bar{U}$ and $g \in \mathcal{G}$ such that $g(s) = f(t)$. Let $h = f^{-1}g$. Thus, $t = h(s)$. Now, for each $i = 1, \ldots, l$, we see that $h(Q_i) = f^{-1}g(Q_i)$. Since, $g \in \mathcal{G}$, it respects Q_i for each $i = 1, \ldots, l$. Hence $h(Q_i) = f^{-1}(Q_i) = X_i$. Similarly, it can be shown that $h(\Theta_j) = f^{-1}(\Theta_j) = \Phi_j$ for each $j = 1, \ldots, r$. From this, we obtain that $(s, u) \in Z$.

We prove part (4) of the theorem as follows. Let $u = (t, X_1, \ldots, X_l, \Phi_1, \ldots, \Phi_r)$ and $u' = (t', Y_1, \ldots, Y_l, \Delta_1, \ldots, \Delta_r)$ be two nodes such that $(s, u), (s, u') \in Z$. This implies that there exist f, g in $\mathcal{H}$ such that $t = f(s)$, $t' = g(s)$ and $X_i = f(Q_i)$ and $Y_i = g(Q_i)$ for $i = 1, \ldots, l$, and $\Phi_j = f(\Theta_j)$ and $\Delta_j = g(\Theta_j)$ for $j = 1, \ldots, r$. This implies that $t = fg^{-1}(t')$ and $X_i = fg^{-1}(Y_i)$ for $1 \leq i \leq l$, and $\Phi_j = fg^{-1}(\Delta_j)$ for $j = 1, \ldots, r$. From this, it follows that $t = t'$ since t, t' belong to the same equivalence class of $\equiv_{\mathcal{H}}$ and we have only one representative state from each such equivalence class in $\bar{V}$. By taking $h = fg^{-1}$, we see that part (4) of the theorem holds in one direction. The other direction can be proved in a similar way. □

From the previous theorem, we see that multiple nodes in *GQS_Stru* $(H, \mathcal{H}, \mathcal{Q})$ can be related through Z to a single node in *QS_Stru*$(G, \mathcal{G}, \mathcal{Q})$. Hence, in principle, *GQS_Stru*$(H, \mathcal{H}, \mathcal{Q})$ can have more nodes than *QS_Stru*$(G, \mathcal{G}, \mathcal{Q})$. The following construction can be used to further reduce the number of nodes in *GQS_Stru*$(H, \mathcal{H}, \mathcal{Q})$ so that the reduced structure has never more nodes than *QS_Stru*$(G, \mathcal{G}, \mathcal{Q})$. First, observe that all the nodes in *GQS_Stru*$(H, \mathcal{H}, \mathcal{Q})$ that are related through Z to a single node s in *QS_Stru*$(G, \mathcal{G}, \mathcal{Q})$ can be represented by a single node since they are all bisimilar to each other. The algorithm for generating *GQS_Stru*$(H, \mathcal{H}, \mathcal{Q})$ can thus be modified to apply this reduction to construct a smaller Kripke structure *Reduced_Stru*$(H, \mathcal{H}, \mathcal{Q})$. Nodes in *GQS_Stru*$(H, \mathcal{H}, \mathcal{Q})$ that are related to a single node in *QS_Stru*$(G, \mathcal{G}, \mathcal{Q})$ can be detected by evaluating the condition stated in part (4) of Theorem 2.

Now consider the 2-process mutual-exclusion example whose reachability graph G is given in Figure 1. The corresponding *GQS* and *GQS_Stru* are given in Figures 2 and 3 respectively. The bisimulation relation Z relates the nodes of G and *GQS_Stru* as follows: node (N_1, N_2) (in G) is related to the two (N_1, N_2)-nodes (in *GQS_Stru*), the (T_1, T_2) node to the two (T_1, T_2)-nodes, the (T_1, N_2) node to the (T_1, N_2)-node with the tracked edge predicate $T_1 \wedge C_1'$, the (N_1, T_2) node to the (T_1, N_2)-node with tracked edge predicate $T_2 \wedge C_2'$, etc. The structure *Reduced_Stru* is shown in Figure 4; it identifies the two (N_1, N_2)-nodes and the two (T_1, T_2)-nodes, and hence has the same number of nodes as G.

If $\mathcal{G}$ is the maximal subgroup of $\mathcal{H}$ consisting of all automorphisms of G that respect $Q_1, \ldots, Q_l$, then, from part (2) of the above theorem, we see that no two nodes in *QS_Stru*$(G, \mathcal{G}, \mathcal{Q})$ are related to a single node in *Reduced_Stru*$(H, \mathcal{H}, \mathcal{Q})$ through Z. Hence, Z defines a bijection from $\bar{U}$ to W. Hence, *Reduced_Stru*$(H, \mathcal{H}, \mathcal{Q})$ has the same number of nodes as *QS_Stru*$(G, \mathcal{G}, \mathcal{Q})$. It is also not difficult to see that Z also defines an isomorphism between the two structures. If $\mathcal{G}$ is not the maximal subgroup of $\mathcal{H}$ consisting of all automorphisms of G that respect $Q_1, \ldots, Q_l$, then *Reduced_Stru*$(H, \mathcal{H}, \mathcal{Q})$ has fewer nodes than *QS_Stru*$(G, \mathcal{G}, \mathcal{Q})$.

The following theorem states that we can check the satisfaction of a CTL^* formula by checking its satisfaction in either of the structures *GQS_Stru*$(H, \mathcal{H}, \mathcal{Q})$, *Reduced_Stru*$(H, \mathcal{H}, \mathcal{Q})$.

THEOREM 3. *Let ϕ be a CTL^* formula that only uses atomic propositions in $\mathcal{X}$. Let s be any node in K_Stru$(G, \mathcal{Q})$ and f be an automorphism in $\mathcal{H}$ such that $f(rep(s, \mathcal{H})) = s$. Then ϕ is satisfied at node s in the structure K_Stru$(G, \mathcal{Q})$ iff ϕ is satisfied at node $u = (rep(s, \mathcal{H}), f^{-1}(Q_1), \ldots, f^{-1}(Q_l), f^{-1}(\Theta_1), \ldots, f^{-1}(\Theta_r))$ in the structure GQS_Stru$(H, \mathcal{H}, \mathcal{Q})$ iff ϕ is satisfied at node u in the structure Reduced_Stru$(H, \mathcal{H}, \mathcal{Q})$.*

PROOF. In the definition of the structure *QS_Stru*$(G, \mathcal{G}, \mathcal{Q})$, let s itself be the representative of the equivalence class of $\equiv_{\mathcal{G}}$ to which it belongs. Since the node s in *K_Stru*$(G, \mathcal{Q})$ is bisimilar to the same node s in *QS_Stru*$(G, \mathcal{G}, \mathcal{Q})$, it follows that s satisfies ϕ in the former structure iff s satisfies ϕ in th latter structure. From Theorem 2, it follows that s satisfies ϕ in *QS_Stru*$(G, \mathcal{G}, \mathcal{Q})$ iff node u satisfies ϕ in *GQS_Stru*$(H, \mathcal{H}, \mathcal{Q})$ iff u satisfies ϕ in *Reduced_Stru*$(H, \mathcal{H}, \mathcal{Q})$. □

The above theorem provides a procedure for model checking a CTL^* formula ϕ by using $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$ or $Reduced\_Stru(H, \mathcal{H}, \mathcal{Q})$. If ϕ is a CTL^* formula, we can check whether ϕ is satisfied at a state in $Reduced\_Stru(H, \mathcal{H}, \mathcal{Q})$ by using the CTL^* model-checking algorithm given in Clarke et al. [1986], and Emerson and Lei [1987]. If ϕ is a CTL formula, any standard CTL model-checking algorithm can also be used. Section 4 presents further improvements to this approach.

3.3 An Alternate Approach for Handling Guards

In this section, we briefly discuss an alternate approach for handling guards in the GQS. We can delete guards from the GQS and incorporate them into the formula that we want to check instead, as explained below. Let $G = (S, E)$ and $H = (S, F)$ be state graphs as previously defined. Note that every infinite path in G is also a path in H, while the converse may not be true. We define a CTL^* path formula g satisfying the following condition: an infinite path in $K\_Stru(H, \mathcal{Q})$ satisfies g iff it is a path in $K\_Stru(G, \mathcal{Q})$. Such a formula g can be obtained easily from the edge predicates $\Theta_1, \ldots, \Theta_r$, and can be written as $\mathsf{G}(g')$ where g' uses only the temporal operator X.

Let p be a state CTL^* formula which we want to check. We transform p as follows. We replace every subformula of the form $\mathsf{E}(q)$ by the subformula $\mathsf{E}(g \wedge q)$ starting with the smallest such subformula (we assume that universal path quantifiers are defined in terms of E). Let p' be the resulting formula. Clearly, a state s in the Kripke structure $K\_Stru(G, \mathcal{Q})$ satisfies p iff the state s satisfies p' in the Kripke structure $K\_Stru(H, \mathcal{Q})$. Now, to check the latter condition, we simply construct the AQS of H with respect to the set of symmetries $\mathcal{H}$ and use the approach given in Emerson and Sistla [1996, 1997]. We call this approach the AQS-based approach. The AQS of H with respect to $\mathcal{H}$ is same as $GQS(H, \mathcal{H}, \mathcal{Q})$ without the guards. That is, if $GQS(H, \mathcal{H}, \mathcal{Q}) = (\bar{V}, \bar{F}, C)$ then AQS of H is simply the pair $(\bar{V}, \bar{F})$.

The AQS-based approach has the following problems. First, if p is a CTL formula then p' is a CTL^* formula but not a CTL formula, and a CTL^* model checker is thus necessary to check p'. Second, the AQS-based approach given in Emerson and Sistla [1996, 1997] constructs an automaton corresponding to the linear temporal formulas in p' and then computes its product with the AQS. This automaton can be large because it also includes atomic propositions appearing in the edge predicates. In contrast, tracking these as edge conditions as in the GQS-based method can be much more efficient, since any symmetry in the edge conditions can be exploited and may result in much fewer nodes.

4. FORMULA DECOMPOSITION AND SUBFORMULA TRACKING

In this section, we discuss two complementary techniques that can improve the direct approach of the previous section.

4.1 Formula Decomposition

Any CTL^* state formula ϕ can be rewritten as a boolean combination of atomic propositions and existential subformulas of the form $\mathsf{E}\phi'$. Let $Eform(\phi)$ denote

the set of existential subformulas of ϕ that are not subformulas of any other existential subformula of ϕ (i.e., they are the top-level existential subformulas of ϕ). Checking whether a state s satisfies a state formula ϕ can be done by checking whether s satisfies each subformula in $Eform(\phi)$ separately, and then combining the results.

For each $\phi' \in Eform(\phi)$, we can determine whether s satisfies ϕ' in the structure $K\_Stru(G, \mathcal{Q})$ by unwinding $GQS(H, \mathcal{H})$, with respect to the predicates in $pred(\phi')$ only, to obtain the Kripke structure $GQS\_Stru(H, \mathcal{H}, pred(\phi'))$ and by checking if the corresponding node satisfies ϕ' in this structure. Formulas in $Eform(\phi)$ that have the same set of atomic propositions can be grouped and their satisfaction can be checked at the same time using the same unwinding. Obviously, unwinding with respect to smaller sets of predicates can yield dramatic performance improvements.

Correlations between predicates can also be used to limit the number of unwindings necessary for model checking. Two predicates Q_i and Q_j in $\mathcal{Q}$ are *correlated* if, for all $f \in \mathcal{H}$, $f(Q_i) = Q_i$ iff $f(Q_j) = Q_j$. It is easy to see that the relation "correlated" is an equivalence relation. We say that two atomic propositions are correlated if their corresponding predicates are correlated. Correlations between predicates can sometimes be detected very easily. For instance, with the framework of Emerson and Sistla [1996, 1997] where automorphisms induced by process permutations are considered, two predicates referring to variables of a same process are correlated: the predicates $x[1] = 5$ and $y[1] = 10$ are correlated if $x[1]$ and $y[1]$ refer to the local variables x and y of process 1, respectively.

If two predicates Q_i and Q_j are correlated, the following property can be proved: if C is a subset of $\mathcal{Q}$ containing Q_i and $C' = C \cup \{Q_j\}$, then the Kripke structures obtained by unwinding with respect to either C or C' will be isomorphic. This fact allows us to combine unwindings corresponding to different formulas in $Eform(\phi)$ whose atomic propositions are correlated. First, we define an equivalence relation among formulas in $Eform(\phi)$: two formulas x and y in $Eform(\phi)$ are equivalent if every atomic proposition in x is correlated to some atomic proposition in y, and vice versa. This equivalence relation partitions $Eform(\phi)$ into disjoint sets $G_1, \ldots, G_w$. Let $pred(G_i) = \{\cup pred(\phi') : \phi' \in G_i\}$. Now for each set G_i, we can unwind $GQS(H, \mathcal{H})$ with respect to $pred(G_i)$ and check whether each formula in G_i is satisfied at $rep(s, \mathcal{H})$.

The number of unwindings can be further reduced by ordering the sets $G_1, \ldots, G_w$ as follows. We say that G_i *is above* G_j if every predicate in $pred(G_j)$ is correlated to some predicate in $pred(G_i)$. The relation "above" is a partial order. We call G_i a *top-set* if there is no set above it. Observe that, if G_i is above G_j, we can combine their unwindings. Hence, if $H_1, .., H_v$ denote the top-sets defined by the sets $G_1, \ldots, G_w$ $(v \leq w)$, we can unwind $GQS(H, \mathcal{H})$ with respect to the predicates in $pred(H_i)$ for each set H_i separately, and check the satisfaction in state s of each formula in H_i and in all the sets G_i "below" it using this unwinding.

Note that using the formula decomposition technique can sometimes be less efficient than the direct approach of the previous section. This can be the case

when there is a lot of overlap between the sets $pred(H_i)$ of predicates corresponding to the sets H_i obtained after partitioning $Eform(\phi)$.

4.2 Subformula Tracking

A CTL^* formula sometimes exhibits itself some internal symmetry. Exploiting formula symmetry was already proposed in Emerson and Sistla [1996]. Here, we generalize these ideas by presenting a unified unwinding process where decomposition and symmetry in a formula can be both exploited simultaneously. First, we need the following definition.

Let ϕ be a CTL^* formula. Consider two state subformulas ϕ' and ϕ'' of ϕ. We say that ϕ' *dominates* ϕ'' in ϕ if ϕ'' is a subformula of ϕ' and every occurrence of ϕ'' in ϕ is inside an occurrence of ϕ'. We say that ϕ' and ϕ'' are *independent* in ϕ if neither of them dominates the other in ϕ. Thus, formulas that are not subformulas of each other are independent. Note that even if a formula is a subformula of another formula, it is possible for them to be independent: for instance, in the formula q given by $\mathsf{E}(\mathsf{EG}q_1 \,\mathsf{U}\, \mathsf{E}(q_1 \mathsf{U} q_2))$, the state subformulas q_1 and $\mathsf{E}(q_1 \mathsf{U} q_2)$ are independent since there is an occurrence of q_1 which does not appear in the context of $\mathsf{E}(q_1 \mathsf{U} q_2)$. Let $Sform(\phi)$ be the set of all subformulas of ϕ that are state formulas. Let $\mathcal{R}$ be a subset of $Sform(\phi)$. We say that $\mathcal{R}$ is a *maximal independent set* if it is a maximal subset of $Sform(\phi)$ such that the state formulas in $\mathcal{R}$ are all pairwise independent. There can be many such maximal independent subsets of $Sform(\phi)$. For instance, the set of all atomic propositions appearing in ϕ is obviously a maximal independent set. For the formula q given above, the set consisting of $\mathsf{EG}q_1$ and $\mathsf{E}(q_1 \mathsf{U} q_2)$ is a maximal independent set.

In what follows, we are interested in exploiting "good" maximal independent sets, that is, sets $\mathcal{R}$ whose elements are symmetric or partially symmetric. A formula q is *symmetric* if, for every automorphism f in $\mathcal{G}$, $f(q) = q$; it is *partially symmetric* when this property holds for almost all f in $\mathcal{G}$. In general, detecting whether a subformula is symmetric is computationally hard. However, when syntactically symmetric constructs (similar to those in $ICTL^*$ [Emerson and Sistla 1997]) are used, it is then easy to determine whether a sub-formula is symmetric. For instance, when only process permutations are used as automorphisms (as in Emerson and Sistla [1996, 1997]), the subformula $\wedge_{i \in I} h(i)$ is symmetric when I is the set of all process indices and $h(i)$ is a formula that only refers to the local variables of process i; the same sub-formula is partially symmetric when I contains most process indices.

Let $\mathcal{R} = \{r_1, \ldots, r_m\}$ be a (preferably good) maximal independent set of subformulas of ϕ. For each $i = 1, \ldots, m$, let R_i denote the set of states in $K\_Stru(G, \mathcal{Q})$ that satisfy the formula r_i. Let $\mathcal{R}' = \{R_1, \ldots, R_m\}$. We identify the sub-formulas in $\mathcal{R}$ with the sets in $\mathcal{R}'$. With this understanding, we consider the Kripke structure $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$ obtained by unwinding $GQS(H, \mathcal{H})$ with respect to $\mathcal{R}$. In a similar way, we can define $Reduced\_Stru(H, \mathcal{H}, \mathcal{R})$ following the procedure of Section 3.

Let ψ denote the formula obtained from ϕ by replacing every occurrence of the sub-formula r_i by a fresh atomic proposition r_i' (i.e., an atomic proposition

not appearing in ϕ), for all $i = 1, \ldots, m$. The following theorem relates the satisfaction of ϕ and ψ.

THEOREM 4. *Let s be a state in S and f be an automorphism in $\mathcal{H}$ such that $s = f(rep(s, \mathcal{H}))$. Then, the formula ϕ is satisfied at state s in the structure $K\_Stru(G, \mathcal{Q})$ iff ψ is satisfied at the node $u = (rep(s, \mathcal{H}), f^{-1}(R_1), \ldots, f^{-1}(R_m), f^{-1}(\Theta_1), \ldots, f^{-1}(\Theta_r))$ in the structure $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$ iff ψ is satisfied at node u in the structure $Reduced\_Stru(H, \mathcal{H}, \mathcal{R})$.*

PROOF. Let $K\_Stru(G, \mathcal{R})$ be the Kripke structure $(S, E, \mathcal{X}', L')$ where $\mathcal{X}' = \{r_i' : 1 \leq i \leq m\}$ and for every $t \in S$, $L'(t) = \{r_i' : t \in R_i\}$. From the semantics of CTL^*, it is easy to see that the formula ϕ is satisfied at state s in $K\_Stru(G, \mathcal{Q})$ iff the formula ψ is satisfied at s in $K\_Stru(G, \mathcal{R})$. From Theorem 3, we see that the formula ψ is satisfied at s in $K\_Stru(G, \mathcal{R})$ iff ψ is satisfied at the node $u = (rep(s, \mathcal{H}), f^{-1}(R_1), \ldots, f^{-1}(R_m), f^{-1}(\Theta_1), \ldots, f^{-1}(\Theta_r))$ in the structure $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$ iff ψ is satisfied at node u in the structure $Reduced\_Stru(H, \mathcal{H}, \mathcal{R})$. The theorem follows from the above observations. □

Thus, the previous theorem makes it possible to check a formula ϕ "hierarchically", by recursively checking subformulas r_i and then combining the results via the unwinding of $GQS(H, \mathcal{H})$ with respect to $\mathcal{R}'$ only.

We now begin to discuss the construction of the structures $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$ and $Reduced\_Stru(H, \mathcal{H}, \mathcal{R})$. A state u of both of these structures is of the form $(\bar{s}, X_1, \ldots, X_m, \Phi_1, \ldots, \Phi_r)$. The sets X_i $(1 \leq i \leq m)$ and Φ_j $(1 \leq j \leq r)$ are obtained by using an automorphism in $\mathcal{H}$ and by applying it to R_i $(1 \leq i \leq m)$ and Θ_j $(1 \leq j \leq m)$ respectively; note that this automorphism is the composition of the automorphisms labeling the edges along a path from the initial state to $\bar{s}$ in $GQS(H, \mathcal{H})$. Let f_u denote the automorphism used to obtain the state u. One way of representing the state u is to *explicitly* represent the sets X_i and Φ_j; in this case, the above structures are constructed by recursively model checking for the subformulas r_i and computing the sets R_i $(1 \leq i \leq m)$ and then applying the unwinding process. Alternately, we can represent a state u *implicitly* by the pair $(\bar{s}, f_u)$; in this case, each set X_i can be computed using f_u and R_i for $1 \leq i \leq m$. This second representation can use less space than the previous one but typically requires more time to be computed. Indeed, whenever we need to check if an already generated state u is the same as a newly generated state, we need to recompute the sets X_i of the state u. A third alternative is to represent each set X_i $(1 \leq i \leq m)$ in the state u implicitly by a CTL^* formula. (We have already stated that each Φ_i is represented implicitly.). We describe this approach below.

We assume that the original set $\mathcal{Q}$ of predicates $\{Q_1, \ldots, Q_l\}$ is closed under the automorphisms in $\mathcal{H}$, that is, for each $f \in \mathcal{H}$ and for each Q_i, $f(Q_i)$ is also in $\mathcal{Q}$ (if this condition is not satisfied, we can expand the set $\mathcal{Q}$ to satisfy it). Recall that the atomic propositions $q_1, \ldots, q_l$ are implicit representations of the predicates $Q_1, \ldots, Q_l$ respectively. We extend the application of the automorphisms in $\mathcal{H}$ to the atomic propositions q_i $(1 \leq i \leq l)$ as follows: For any automorphism f, if $f(Q_i) = Q_j$, then we define $f(q_i)$ to be q_j. Now, for any CTL^* formula ψ over the atomic propositions $q_1, \ldots, q_l$ and $f \in \mathcal{H}$, let $f(\psi)$ denote

the formula obtained by replacing every occurrence of every atomic proposition q_i (for $1 \leq i \leq l$) in ψ by $f(q_i)$.

Consider a node $u = (\bar{s}, X_1, \ldots, X_m, \Phi_1, \ldots, \Phi_r)$ in $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$ as defined above. Let f_u be the automorphism in $\mathcal{H}$ that is associated with u as defined earlier. (Recall that $f_u(R_i) = X_i$ for $1 \leq i \leq m$). We represent the sets X_i (for $1 \leq i \leq m$) by the CTL^* formula $f_u(r_i)$. The following lemma gives a sufficient condition for checking if two states u, v are equal.

LEMMA 1. *Let* $u = (\bar{s}, X_1, \ldots, X_m, \Phi_1, \ldots, \Phi_r)$ *and* $v = (\bar{t}, Y_1, \ldots, Y_m, \Delta_1, \ldots, \Delta_r)$ *be nodes in* $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$. *If* $\bar{s} = \bar{t}$, *the* CTL^* *formula* $f_u(r_i)$ *is equivalent to* $f_v(r_i)$ *for each* i *such that* $1 \leq i \leq m$, *and* $\Phi_j = \Delta_j$ *for each* j *such that* $1 \leq j \leq r$, *then* u, v *are the same nodes.*

PROOF. Consider the graph $f_u(G) = (f_u(S), f_u(E))$. Since $f_u \in \mathcal{H}$, it is the case that $f_u(S) = S$. Recall that $E = \bigcup_{e \in \bar{F}} Class(e, \mathcal{H}) \cap C(e)$ where $\bar{F}$ is the set of labeled edges in $GQS(H, \mathcal{H})$. From this, it is not difficult to see that $f_u(E) = \bigcup_{e \in \bar{F}} Class(e, \mathcal{H}) \cap f_u(C(e))$. (This is because $Class(e, \mathcal{H})$ is closed under f_u.) Since $C(e) = \Theta_j$ (for some $1 \leq j \leq r$), it follows that $f_u(C(e)) = \Phi_j$. Consider any i such that $1 \leq i \leq m$. Any state s in $K\_Stru(G, \mathcal{Q})$ satisfies r_i iff the state $f_u(s)$ in $K\_Stru(f_u(G), \mathcal{Q})$ satisfies $f_u(r_i)$ (this can be proved by a simple induction on the structure of r_i). Hence, we see that $X_i = f_u(R_i)$ is exactly the set of states in $K\_Stru(f_u(G), \mathcal{Q})$ that satisfy $f_u(r_i)$. By a similar argument, we see that Y_i is the set of states in $K\_Stru(f_u(G), \mathcal{Q})$ that satisfy $f_v(r_i)$. From the hypothesis of the lemma, we have $f_u(r_i)$ is equivalent $f_v(r_i)$. Hence, $X_i = Y_i$. This holds for each i such that $1 \leq i \leq m$. The lemma easily follows from this. □

Note that the above lemma gives only a sufficient condition. So it is possible for two states u and v to be the same without satisfying the condition of the lemma. In this case, our procedure will treat these states as distinct. This may increase the number of states in the constructed structures and hence increase the time for model checking. However, this does not effect the soundness of the model checking procedure.

Another important aspect in the construction of $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$ is the generation of $N(\bar{s})$ for each state $\bar{s}$. For the node $u = (\bar{s}, X_1, \ldots, X_m, \Phi_1, \ldots, \Phi_r)$, $r_i' \in N(u)$ iff $\bar{s} \in X_i$. From the observations made in the proof of the above lemma, we see that $\bar{s} \in X_i$ iff the $\bar{s}$ satisfies the formula $f(r_i)$ in the structure $K\_Stru(f(G), \mathcal{Q})$. The latter condition can be checked by recursively invoking the procedure for checking $f(r_i)$ using the edge conditions $f(\Theta_1), \ldots, f(\Theta_r)$, that is, the edge conditions $\Phi_1, \ldots, \Phi_r$.

In summary, we represent each of the sets X_i in the node u by a CTL^* formula x_i. In the initial node of $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$, the value of x_i will be simply r_i, for $1 \leq i \leq m$. During the unwinding process, whenever we unwind along an edge labeled with the automorphism h, the value of x_i in the new node are obtained by applying h^{-1} to the value of x_i in the previous node. To check if two nodes $u = (\bar{s}, x_1, \ldots, x_m, \Phi_1, \ldots, \Phi_r)$ and $v = (\bar{t}, y_1, \ldots, y_m, \Delta_1, \ldots, \Delta_r)$ are the same, we check that $\bar{s} = \bar{t}$, the CTL^* formulas x_i, y_i are equivalent for $1 \leq i \leq m$ and $\Phi_j = \Delta_j$ for $1 \leq j \leq r$. Checking the equivalence of CTL^*

formulas can be computationally hard. Note that, if the CTL^* formula ϕ uses syntactically symmetric constructs such as those in $ICTL^*$ [Emerson and Sistla 1996], then this check can always be done efficiently. To check if $r_i' \in N(u)$, we recursively invoke the procedure on the CTL^* formula x_i using the initial conditions $\Phi_1, \ldots, \Phi_r$.

We thus obtain a complete recursive procedure which constructs different structures corresponding to the different sub-formulas R_i of ϕ. Note that the formula decomposition technique of Section 4.1 can be used to decompose sub-formulas R_i. Thus, formula decomposition and sub-formula tracking are complementary and can be both applied recursively. It is to be noted that if no good maximal independent set $\mathcal{R}$ can be found then the procedure of Section 4.1 should be applied directly.

The following procedure summarizes the different steps for checking if the formula ϕ is satisfied at state s in $K\_Stru(G, \mathcal{Q})$. The input to the procedure is ϕ and the description of the concurrent program. We assume that the group $\mathcal{H}$ is known or can be deduced from the structure of the concurrent program as is done in SMC or Murphi.

(1) If the nesting depth of ϕ is one or less, then use the procedure given in section 4.1. Otherwise, go to Step (2).
(2) Analyze the formula ϕ and obtain a good maximal independent set $\mathcal{R}$. $\mathcal{R}$ is considered good if many of its members are symmetric or partially symmetric. If no such set $\mathcal{R}$ can be found then use the decomposition approach given in section 4.1 and exit. Otherwise, go to Step (3).
(3) Let $\mathcal{R} = \{r_1, \ldots, r_m\}$. Construct the formula ψ from ϕ by replacing each occurrence of r_i in ϕ by a new atomic proposition r_i', for $i = 1, \ldots, m$. Construct the Kripke structure $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$ or $Reduced\_Stru(H, \mathcal{H}, \mathcal{R})$. Check if the formula ψ is satisfied in the constructed structure at the node u where u is the node defined in Theorem 4 (i.e., $u = (rep(s, \mathcal{H}), f^{-1}(R_1), \ldots, f^{-1}(R_m), f^{-1}(\Theta_1), \ldots, f^{-1}(\Theta_r))$. This checking can be done directly or by recursive invocation of this algorithm. Note that constructing either $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$ or $Reduced\_Stru(H, \mathcal{H}, \mathcal{R})$ requires determining the satisfaction of sub-formulas in $\mathcal{R}$ at the nodes in these structures; this can be done by recursive invocation of this procedure.

Example. We illustrate the procedure above with a simple example. Assume that we are using automorphisms induced by process permutations, as in Emerson and Sistla [1996, 1997]. Consider a concurrent system of n processes which are all similar except for one of them. Also consider a formula $\phi = \mathsf{E}(q_1 \mathsf{U} \wedge_{i \in I} \mathsf{E}h(i))$ where $h(i)$ is a path formula with no further path quantifiers which only refers to the local propositions of process i, I is the set of all process indices except process 1, and q_1 is a local proposition of process 1. Let ϕ' denote the subformula $\wedge_{i \in I} \mathsf{E}h(i)$. It is a partially symmetric subformula. We take $\mathcal{R}$ to be the set $\{q_1, \phi'\}$, since it is a "good" maximal independent set. Note that the formula ψ of Theorem 4 is obtained by replacing q_1, ϕ' in ϕ by two new atomic propositions r_1' and r_2' respectively. In this case, we get ψ to be $\mathsf{E}(r_1' \mathsf{U} r_2')$.

We construct $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$ where $\mathcal{H}$ is the set of automorphisms induced by the group consisting of all permutations over the n processes. Suppose that there is one edge predicate that references a single process. Let M be the total number of nodes in $GQS(H, \mathcal{H})$. M can be exponentially smaller than the number of nodes in the full reachability graph, that is, the number of nodes in $K\_Stru(G, \mathcal{Q})$. Each node in $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$ is of the form $(\bar{s}, q_j, \wedge_{i \in J} \mathsf{E}(h(i)), \Phi)$ where j is a process index, J is the set of process indices other than j and Φ is a tracked edge predicate. Note that the second, third and fourth component of the node is obtained by applying a permutation to subformulas q_1, ϕ' and to the edge predicate respectively. It is easy to see that there are at most n different values of j. Further more, there will be at most n different tracked edge predicates appearing in these nodes. From this we see that the number of nodes in $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$ is at most n^2M. During the construction of $GQS\_Stru(H, \mathcal{H}, \mathcal{R})$, we need to determine which of its nodes satisfy subformulas of the form $\wedge_{i \in J} \mathsf{E}h(i))$ where J is a set containing $n - 1$ process indices (these subformulas are obtained by applying a permutation to ϕ'). We do this as follows. We use formula decomposition of Section 4.1. For each tracked edge predicate and for each i, we recursively determine nodes that satisfy $\mathsf{E}(h(i))$. This is done by unwinding $GQS(H, \mathcal{H})$. The resulting unwound structure is of size $O(nM)$. Thus, for each tracked-edge predicate and for each i, determining nodes that satisfy $\mathsf{E}(h(i))$ is of complexity $O(nM)$. Since this has to be done for each edge predicate in the unwound structure (and there are at most n such edge predicates) and for each i, the overall complexity is $O(n^3M)$.

However, if we use the direct approach and unwind $GQS(H, \mathcal{H})$ (or if we use $QS\_Stru(G, \mathcal{G}, pred(\phi))$), we then obtain the full reachability graph. Since M is exponentially smaller than the number of nodes in the full reachability graph, we see that the above example is a case where formula decomposition together with subformula tracking yields an exponentially better complexity than the direct approach alone. Of course, one can give examples of formulas with no symmetric (or partially symmetric) subformulas for which the method of this section does not help.

Note that the formula ϕ defined above is not an $ICTL^*$ formula and hence the method of Emerson and Sistla [1997] for $ICTL^*$ formulas cannot be applied. Furthermore, the approach of Emerson and Sistla [1996] that explicitly computes the formula symmetry will not be of any use in reducing the state space because this method would consider ϕ not to contain any symmetry at all, that is, it would compute the set of symmetries for ϕ to be the identity set.

5. IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this section, we present experimental results evaluating the techniques proposed in this article. More specifically, we compare three different methods: (1) the QS-based method using the quotient structure QS defined in Section 3; (2) the GQS-based method using the GQS defined in Section 3; (3) the AQS-based method, which is the alternate method presented in Section 3.3.

The techniques based on formula decomposition can be used in conjunction with all three methods. For example, with the QS-based method, to check the

formula $g \wedge h$ where g and h do not share any correlated predicates, we can check g and h separately and then combine the results. (Constructing two separate quotient structures can be faster than constructing one quotient structure with fewer symmetries.)

In Section 3, we have already defined the structures $GQS\_Stru(H, \mathcal{H}, \mathcal{Q})$ and $Reduced\_Stru(H, \mathcal{H}, \mathcal{Q})$. As indicated earlier, we will use the $Reduced\_stru(H, \mathcal{H}, \mathcal{Q})$ in our *GQS*-based method. We briefly describe the procedure for incrementally constructing the reachable part of $Reduced\_Stru(H, \mathcal{H}, \mathcal{Q})$ from $GQS(G, \mathcal{H})$. The procedure maintains a set *To_explore* of nodes that have been visited but are yet to be explored. Exploration of a node consists of generating all its successor nodes. Initially, *To_explore* contains nodes of the form $(s_0, Q_1, \ldots, Q_l, \Theta_1, \ldots, \Theta_r)$ where s_0 is the representative of an equivalence class containing an initial state. We iterate the following procedure until *To_explore* is empty. We remove a node $z = (t, Z_1, \ldots, Z_l, \Psi_1, \ldots, \Psi_r)$ from *To_explore*. For each labeled edge $e = (t, t', f)$ in $GQS(G, \mathcal{H})$, we check if the edge $(t, f(t'))$ satisfies the edge predicate Ψ_j, where j is the unique integer such that Θ_j is the edge predicate $C(e)$. If this condition is satisfied we proceed as follows. We construct the node $u' = (t', Y_1, \ldots, Y_l, \Delta_1, \ldots, \Delta_r)$ where $Y_i = f^{-1}(Z_i)$ for $1 \leq i \leq l$ and $\Delta_j = f^{-1}(\Psi_j)$ for $1 \leq j \leq r$. Then, we check if there exists a node $u = (t', X_1, \ldots, X_l, \Phi_1, \ldots, \Phi_r)$ in the partially constructed $Reduced\_Stru(H, \mathcal{H}, \mathcal{Q})$ and a $h \in \mathcal{H}$ such that $t' = h(t')$ and $X_i = h(Y_i)$ for all $i = 1, \ldots, l$, and $\Phi_j = h(\Delta_j)$ for all $j = 1, \ldots, r$ (i.e., the condition of Part 4 of Theorem 2 is checked). If this condition is satisfied, we add an edge from z to u; otherwise, we add u' as a new node, include it in *To_explore* and add an edge from z to u'.

5.1 Implementation of Priorities

5.1.1 *Priorities in the GQS-Based Method.* We implemented the *GQS*-based method for systems with priorities by extending the SMC tool [Sistla et al. 2000]. We call the extended system the Prioritized SMC (or simply PSMC). The SMC tool takes as input a concurrent program and a property automaton, and checks if there exists a fair computation of the concurrent program that is accepted by the automaton. The automaton is an incorrectness automaton, that is, it accepts exactly the incorrect computations. Thus, SMC checks CTL^* properties of the form $E(p)$ where the path formula p is a linear time property specified by an automaton. The input concurrent program is divided into modules where each module consists of a set of processes that are identical up to renaming. SMC considers automorphisms induced by process permutations. Given the above syntax of the concurrent program, it is easy to see that any permutation mapping processes in a module to processes with in the same module is an automorphism of the reachability graph of the concurrent program. These automorphisms are used to construct the *AQS* by SMC.

Processes in the concurrent program communicate through shared variables. A shared variable is specified by a name together with a list of process indices. If a process in a module `C` has a shared variable with another process in module `D`, then every process in `C` has such a shared variable with every process in `D`. A

different type of variables, called index variables, are used in defining the processes in a module. An index variable ranges over the process IDs of a module. A module specification starts with the declaration of a single index variable, called the *primary index variable*, and is followed by a set of transition schemas. The primary index variable identifies the process to which the instance of the transition schema belongs to. Each transition schema is given by a condition part and an action part. The condition part is a boolean expression over atomic conditions and the action part is a set of concurrent assignment statements. The following SMC input describes a resource controller example.

```
// declaration of modules
Module server          = 2;
Module client          = 80;

// declaration of program variables
busy[server]           = 0;
request[server, client]  = 0;
reply[server, client]  = 0;
lc[client]             = 0;

//index variable declaration
s of server;
c of client;

// server specification starts
s: busy[s] == 0 && request[s,c] == 0 -> reply[s,c] =1,
                                        busy[s] = 1;

// client specification starts
c: {
    lc[c] == 0                    -> lc[c] = 1, request[s,c] = 1;
    lc[c] == 1 && reply[s,c] == 1 -> lc[c] = 2, request[s,c] =0;
    lc[c] == 2 && reply[s,c] == 1 -> lc[c] = 0, busy[s] =0,
                                     reply[s,c]=0;
   }
```

The above example declares two modules called `server` and `client` having two and eighty processes, respectively. It also declares two sets of variables `request[s,c]`, `reply[s,c]`, for each $0 \leq s < 2$ and $0 \leq c < 80$. For each server s, it declares the variable `busy[`s`]`. For each client c, it declares the location counter variable `lc[`c`]`. All these variables are initialized to zero. It also declares two index variables s, c ranging over server and client processes respectively.

The server module has a single transition schema (note that s is used as the primary index variable). For each value of $s = 0, 1$, the server process s has eighty transitions obtained by substituting the values 0, .., 79 for the client index variable c. The client module has three transition schemas (note c is the primary index variable).

In the PSMC system, priorities can be specified with transition schemas having two index variables, where priorities are defined with respect to the nonprimary index variable. We call this nonprimary index variable the *secondary index variable*. For instance, the single transition schema in the server module of the above example has the two index variables s, c appearing in it, where c is the secondary index variable. Priorities for this transition schema can be defined by having the following command immediately after it:

```
Priority (X1;X2;...;Xk),
```

where $X1, X2, \ldots, Xk$ are disjoint sets of process ids belonging to the client module. In this command, each Xi is specified as a list of process IDs (or ranges of process IDs) separated by commas. Such a specification states that, for this transition schema, all client processes belonging to Xi have the same priority and, for $i < j$, processes belonging to Xi have higher priority than processes belonging to Xj. The formal semantics of this priority scheme for the transition schema in the server module is defined as follows: Fix the the value of the index variable s. Let t_i, for $0 \leq i < 80$, denote the transitions in the server process s when the constant i is substituted for the index variable c in the above transition schema. Let δ denote a global state of the above system. Transition t_i can be executed in δ if t_i is enabled in δ (i.e., its condition part is satisfied in δ) and there is no j $(0 \leq j < 80)$ such that t_j is enabled in δ and j has higher priority than i. Thus, priorities in the above transition schema states that the server process s must grant the waiting request to one of the clients with the highest priority.

5.1.2 *Implementation Details of the GQS-Based Method.* The SMC system as well as the PSMC system are based on using automorphisms induced by process permutations. The SMC system is modified in the following way to get the PSMC system. First, observe that if G is the reachability graph of the input concurrent program with the priority specifications then H is simply the reachability graph of the same concurrent program without the priority specifications. The *AQS* structure is constructed ignoring the priority specifications associated with the transition schemas. Already in the SMC system, the label of each transition of the *AQS* includes a permutation identifier and the index of the process to which this transition belongs; this index value is same as the instantiated value of the primary index variable. In addition to this information, the PSMC system is designed in such a way that, if a transition schema has a priority specification associated with it, then all the *AQS* transitions corresponding to this transition schema also contain the instantiated value of the secondary index variable and a pointer to the priority specification. The resulting structure is the *GQS*. Note that PSMC does not use any edge predicate, but instead stores the above additional information with each *GQS* transition.

We briefly describe how the additional information associated with the *GQS* transitions are used during the unwinding process with an example. Consider a transition schema T that specifies highest priority to process 0 of module C, but equal priorities to all other processes belonging to module C. During the unwinding process we track process 0 of module C. (Indeed, edge predicates in the *GQS* only refer to process 0 of module C; tracking the values of edge

predicates thus reduces to tracking process 0 in this example.) Suppose during the unwinding process, i.e., during the construction of *GQS_Stru*, we are at node u whose *GQS* state component is $\bar{s}$ and whose tracked process value is c. Assume that there is a *GQS* transition t from state $\bar{s}$ which is obtained from T by instantiating its primary and secondary index variables with values d and e respectively (note that all this information is contained within t). Consider the instantiation of the transition schema T by substituting d and c for the primary and secondary index variables respectively. (Note that c is the tracked value of the index of the highest priority process.) If this instantiated schema is enabled in the *GQS* state $\bar{s}$ and $e \neq c$, then we cannot use transition t; otherwise, we can use t and continue unwinding along this transition. The correctness of this implementation is easily seen from our earlier discussion.

The above implementation is generalized naturally to the case when the priority specification is given by multiple priority classes where each class has more than one process. This generalization requires tracking sets of processes, not individual processes; this is done by simply applying the permutations to the sets of process indices during the unwinding process.

Another important aspect of PSMC is that it directly constructs the reduced structure, that is, *Reduced_Stru*, from the *GQS*. Recall that during the construction of the reduced structure, described at the beginning of this section, we check if the condition in part (4) of Theorem 2 is satisfied; in this condition $u' = (t', Y_1, \ldots, Y_l, \Delta_1, \ldots, \Delta_r)$ is a newly generated node and $u = (t', X_1, \ldots, X_l, \Phi_1, \ldots, \Phi_r)$ is a previously generated node. This condition requires the existence of an automorphism $h \in \mathcal{H}$ such that $h(t') = t'$ and $X_i = h(Y_i)$ for $i = 1, \ldots, l$ and $\Phi_j = h(\Delta_j)$ for $j = 1, \ldots, r$. Any automorphism h such that $h(t') = t'$ is called a state symmetry of t' [Emerson and Sistla 1996; Gyuris and Sistla 1999; Sistla et al. 2000]. For programs specified in SMC and in PSMC a large subgroup of the group of state symmetries of a state can be computed and represented efficiently at the time of construction of $GQS(G, \mathcal{H})$ [Gyuris and Sistla 1999; Sistla et al. 2000]. We use this group of state symmetries. The construction of the *Reduced_Stru*$(H, \mathcal{H}, \mathcal{Q})$ involves marginal additional cost compared to the construction of *GQS_Stru*$(H, \mathcal{H}, \mathcal{Q})$.)

5.1.3 *Priorities in the QS-Based and the AQS-Based Methods.* The *AQS*-based method incorporates priorities in the property automaton. In this method, the *AQS* is constructed assuming that all processes have equal priorities. The property automaton only considers computations that satisfy the priority requirements. In our experiments, we have manually incorporated the priorities into the automaton.

The *QS*-based method is evaluated for systems with priorities using the SMC system as follows. Processes in an original module are divided into new modules based on their priorities, so that all processes having identical priorities are placed in the same new module, and processes in different new modules have different priorities. The transitions of the processes are modified appropriately to encode the priorities. These new modules form the input modules to the SMC system. This encoding of priorities was done manually for our experiments.

5.2 Case Study

We have tested the *QS*-based, *GQS*-based and *AQS*-based methods on several examples of systems. Experimental results are presented below for two examples: a model of the IEEE Fire-wire protocol [IEEE 1995] and a simpler resource-controller example. All experiments were conducted on a Sun Ultra2 workstation running Sun-OS 5.5.1.

A simplified description of the asynchronous data transfer mode of the protocol is the following. The protocol is composed of three main layers: the transaction layer, the link layer and the physical layer. Loosely speaking, the transaction layer communicates with the applications, the physical layer handles the physical transfer of the data, and the link layer serves as an intermediary between the transaction layer and the physical layer. The different layers communicate through *actions* or *services*. When the transaction layer of a client on one station (site) wants to send a message p to some other client, it indicates this to the link layer by executing the appropriate action. The link layer then requests the physical layer for arbitration of the common bus. This request is either granted or denied by the physical layer. If the link layer wins the arbitration, it then forwards the packet p to the physical layer. The physical layer notifies the recipient of the incoming packet at the destination station. At this station, the message is forwarded by the link layer to the transaction layer. The transaction layer at the destination site acknowledges the message and this acknowledgment is forwarded back to the sender via the bus. Sending of the acknowledgement also requires arbitration for the bus. A more detailed (yet simpler than the IEEE standard given in [IEEE 1995]) description of the protocol can be found in Sighireanu and Mateescu [1999], and Sistla et al. [2000].

We evaluated the three methods developed in this paper by checking various properties of the fire-wire protocol extended with priorities. We concentrated on the link layer part of the protocol. A deadlock in this protocol was previously detected using SMC and reported in Sistla et al. [2000] (the same deadlock was also detected by Sighireanu and Mateescu [1999]). This deadlock was due to a missing transition in the specification. After adding this transition, no further deadlock was found. In the current experiments, we introduced priorities in to the deadlock-free protocol, and analyzed several properties using the three different methods. We modeled the link layer part of the protocol in detail, while the physical layer was represented by a single process modeling the interface of the physical layer with the link layer (including arbitration, etc.) and the actual transmission of the data on the bus. Some of the transitions of the physical-layer process model the arbitration scheme between the link layer parts of different stations. We have introduced priorities for these transitions so that station 1 is given the highest priority.

We considered one safety property and two liveness properties. The safety property states that whenever a station sends a non-broadcast message to another station, it will not send any further messages as long as an acknowledgment for the current message is not received. We actually checked whether there exists a computation that violates this property. As described earlier, this is done by specifying an automaton which accepts all sequences that violate the

Table I. Table for the Safety Property (Fire-wire protocol) : Time and Memory

# of stations	QS-based		AQS-based		*GQS*-based	
	Time	Memory	Time	Memory	Time	Memory
2	2	671	1	401	1	435
3	95	23081	39	10000	36	10808

Table II. Table for the Safety Property (Fire-wire protocol): R-nodes and P-nodes

# of stations	QS-based		AQS-based		GQS-based	
	R-nodes	P-nodes	R-nodes	P-nodes	R-nodes	P-nodes
2	2745	3156	1375	3414	1375	3156
3	71921	80530	24361	88100	24361	80530

above property and then checking if any computation of the concurrent program is accepted by this automaton. Note that the existence of an incorrect computation can be specified by a CTL^* formula of the form $\mathsf{E}(p)$ where p does not contain any path quantifiers, that is, p is a linear temporal logic formula. PSMC and SMC check whether a property of the form $\mathsf{E}(p)$ holds at the initial state where p is a linear time property given by an automaton. (Thus, these model checkers currently do not handle full CTL^*.) We use the automaton specifying the incorrectness as the input automaton for the *QS*-based and *GQS*-based methods. In the case of the *AQS*-based method, we use an automaton which is a product of the incorrectness automaton and the priority automaton. All the three methods proved that the above property is indeed satisfied.

Results of experiments with this property are given in Tables I and II. In both tables, the first column gives the number of stations used in the experiment. Table I gives the overall model checking time (in seconds) and the maximum memory used (in Kilobytes) for each of the three methods. Table II has two columns for each of the methods: the R-nodes and P-nodes columns. An entry in a R-nodes column gives the number of representative states; that is, it gives the number of nodes in the *QS*, the *AQS* and the *GQS*, for the *QS*-based, the *AQS*-based and the *GQS*-based methods, respectively. An entry in a P-nodes column gives the number of nodes in the product structure constructed by the corresponding method.

—In the case of the *QS*-based method, the product structure is the product of the Kripke structure *QS_Stru* (defined in Section 3) and of the incorrectness automaton.
—In the case of the *GQS*-based method, the product structure is the the product of the structure *Reduced_Stru* (given in Section 3) and of the incorrectness automaton.
—In the case of the *AQS*-based method, the product structure is the product of the *AQS*, of the incorrectness automaton and of the priority automaton.

It is to be noted that the *QS*-based method uses the maximum set of possible automorphisms. That is, in the case of three stations where station 1 is given

highest priority, the set of automorphisms used is the set of all permutations among the stations 2 and 3.

As is seen from the tables, the number of representative states is much less for the *GQS*-based method compared to the *QS*-based method, while the number of product nodes is the same for both methods. However, the overall time and memory requirements for the *GQS*-based method is much less than those for the *QS*-based method. This can be explained as follows: Note that both these methods involve two steps. In the first step, the *QS* or the *GQS* is constructed. In the *QS*-based method the second step involves construction of the product structure and its simultaneous exploration on the fly. In case of the *GQS*-based method, the second step also involves unwinding of the *GQS* to obtain the *Reduced_Stru*; this additional function is carried out simultaneously with product construction and exploration. For the *GQS*-based method, the time for the first step takes a lot less time since the size of the *GQS* is much smaller than that of the *QS*. Since the number of nodes in the product structure is the same, the component of the time for construction and exploration of the product structure during the second step is almost the same for both of these methods. However, the second step for the *GQS*-based method involves an additional function of unwinding the *GQS*. It turns out that this does not make much of a difference in the time taken for the second step. For example, in the case of three stations, the first step of the *GQS*-based method took 30 seconds while the second step took 6 seconds. In comparison, the first step of the *QS*-based method took 89 seconds while its second step took 6 seconds (these detailed statistics are not given in the tables). Construction of the *QS* and *GQS* involves identifying all the enabled transitions, executing them and checking if each newly generated state is equivalent to an already generated state. For this reason the first step of the *QS*-based method takes more time than the first step of the *GQS*-based method. On the other hand, the additional function of unwinding carried out in the second step of the *GQS*-based method does not involve identifying and executing enabled transitions of the program and it does not involve sophisticated equivalence checking; it involves simple equivalence checking under the state symmetry (actually the information about state symmetry is already calculated during the first step). For this reason, this function does not cause any noticeable increase in runtime for the second step. Similarly, the memory requirement is dominated by the size of the *QS* and *GQS* respectively. As a consequence, the overall memory requirement for the *GQS*-based method is much less than that of the *QS*-based method.

The runtime for the *GQS*-based method is consistently smaller than that for the *AQS*-based method. This is because the number of product nodes for the *AQS*-based method is higher than that for the *GQS*-based method. Indeed, the input automaton for the *AQS*-based method has more states since it also captures the priority specification. The overall memory needed for the *GQS*-based method (respectively, *AQS*-based method) is the sum of the memory needed for building the *GQS* (respectively, *AQS*) plus the memory needed to explore the product structure. The memory needed for storing the *GQS* is larger than for the *AQS* because we store additional information with each *GQS* transition. As a consequence, we see that the overall memory requirements for the

Table III. Table for the first Liveness Property (Fire-wire protocol): Time and Memory

# of stations	QS-based		AQS-based		GQS-based	
	Time	Memory	Time	Memory	Time	Memory
2	2	690	1	459	1	449
3	92	23422	39	11345	36	11052

Table IV. Table for the first Liveness Property (Fire-wire protocol): R-nodes and P-nodes

# of stations	QS-based		AQS-based		GQS-based	
	R-nodes	P-nodes	R-nodes	P-nodes	R-nodes	P-nodes
2	2745	3684	1375	3970	1375	3684
3	71921	90015	24361	98159	24361	90015

GQS-based method can be slightly higher than that for the *AQS*-based method. On the other hand, the memory needed for exploration of the product structure is higher in the case of the *AQS*-based method. Moreover, when the number of priority levels increases, the memory requirements for the product structure in the *AQS*-based method grow rapidly and can become much higher than that for the *GQS*-based method. This can be seen from the results obtained for the resource controller example given in the next subsection.

The first liveness property we consider states that, whenever station 1 wants to arbitrate and send a point-to-point message to another station, it will successfully send the message and eventually receive an acknowledgment. All three methods correctly prove that this liveness property is not satisfied by the protocol under weak fairness, even though station 1 has the highest priority in the arbitration. Indeed, after station 1 successfully gets the bus, some other station may continuously request access to the bus and be denied: this continuous unsuccessful arbitration activity by another station keeps the physical layer process occupied and prevents it from transmitting the message sent by station 1. To avoid this behavior, we check if the same liveness property is satisfied under the condition that another station can unsuccessfully request the bus only a finite number of times, that is, under the assumption of a finite number of arbitration requests by other stations. Under this additional condition, all three methods prove that station 1 can successfully transmit its message and eventually receive an acknowledgment. The corresponding experimental results are presented in Tables III and IV.

The second liveness property we consider is similar to the first liveness property, but is stated for station 2. In this case, even under the assumption that other stations arbitrate only a finite number of times, the property is not satisfied. This is because station 2 has lower priority than station 1 and will lose the arbitration if station 1 sends a request at the same time. We then modified the property by requiring that this liveness property hold under the assumption that there is no arbitration from any other station at the same time. All three methods were then able to prove that the modified property is satisfied. Results of these experiments are given in Tables V and VI.

Table V. Table for the second Liveness Property (Fire-wire protocol): Time and Memory

# of stations	QS-based		AQS-based		GQS-based	
	Time	Memory	Time	Memory	Time	Memory
2	2	676	1	395	1	435
3	99	25732	49	12593	43	13074

Table VI. Table for the second Liveness Property (Fire-wire protocol): R-nodes and P-nodes

# of stations	QS-based		AQS-based		GQS-based	
	R-nodes	P-nodes	R-nodes	P-nodes	R-nodes	P-nodes
2	2745	3000	1375	3236	1375	3000
3	71921	146177	24361	160113	24361	146177

From the above results, we see that the *GQS*-based method and the *AQS*-based method perform better than the *QS*-based method in terms of time as well as memory usage. The *GQS*-based method performs slightly better than the *AQS*-based method in runtime although the difference is not very significant. This may be explained by the fact that there are only few stations and only two priority levels. In the resource controller example given in the next section, where many processes are used with multiple priority levels, the *GQS*-based method performs substantially better than the *AQS*-based method in both the overall time and memory requirements.

It is to be noted that the symmetry based method (i.e., SMC) outperforms the method that does not employ symmetry for the Fire-Wire protocol. The interested reader can refer to Sistla et al. [2000], and Gyuris and Sistla [1999] for this experimental comparison.

5.3 Another Example

In this section, we study the performance of the three methods developed in this paper for the verification of certain properties of a client/server resource controller system with many client processes and multiple priority levels. We considered a system with one server and 80 client processes. We carried out the following sets of experiments. In all these experiments, the *QS*-based method used the maximal possible set of automorphisms, that is, $\mathcal{G}$ is the set of automorphisms induced by permutations that permute processes within the same priority class arbitrarily. In the case of the *GQS*-based method, the set of automorphisms $\mathcal{H}$ consists of those that are induced by permuting the client processes arbitrarily.

We considered systems with k priority levels for the client processes, for different values of k and with different number of processes in each priority level. That is, the client processes are divided into k classes. For each $i = 1, \ldots, k-1$, the process in the ith class have higher priority than the processes in class j, for each $j = i+1, \ldots, k$.

We checked the property stating that the resource cannot be free and held by a process at the same time. This is an indirect specification of the mutual

Table VII. Indirect Mutual Exclusion Property (resource controller example)

value of k	QS-based		AQS-based		GQS-based	
	Time	Memory	Time	Memory	Time	Memory
2	12	3017	12	2282	14	2529
3	21	5402	21	4088	18	3077
4	42	10468	52	9160	27	4136
5	97	21141	158	23538	51	6627

exclusion property. (We are only using this indirect mutual exclusion property for evaluation of the different methods. It is not difficult to see that mutual exclusion is a symmetric property that can be checked more efficiently directly on the *QS* or the *GQS* without any unwinding. If this method is used, then *GQS*-based method outperforms the *QS*-based method substantially since the size of the *GQS* is much smaller than that of the *QS*.) The indirect mutual exclusion property is expressed as a disjunction of k clauses: the ith clause (for $i = 1, \ldots, k$) tests whether a process in the ith class holds the resource, while at the same time the resource is indicated to be free.

In the first set of experiments, we considered systems in which the first $k-1$ classes each have only one client process and the kth class has the remaining $(80 - k + 1)$ client processes. The results of these experiments are given in Table VII. Each row in the table corresponds to a different value of k and each column gives the results obtained when the method indicated at the top of the column is employed. It is easy to see that the *GQS*-based method outperforms the other methods for $k = 3$ and above. For $k = 2$, its performance is slightly worse. The difference in the performance of the *GQS*-based method and the other two methods increases with the value of k. Compared to the results obtained with the Fire-wire protocol, we see that the *GQS*-based method can substantially outperform the *AQS*-based method for high values of k. This is partly due to the fact that the size of the automaton used in the *AQS*-based method increases when there are more priority levels.

In the second set of experiments, we considered a system with two priority classes (i.e., with $k = 2$) each having 40 processes. In this case, we compared the *QS*-based and the *GQS*-based method using the indirect mutual exclusion property as specified above. (We could not test the *AQS*-based method for this case since including the priorities in the automata will break the symmetry existing in each of the priority classes.) We could test both these methods using formula decomposition. (To check for a formula of the form $g \vee h$ using formula decomposition with the *QS*-based method, we simply check g and h separately and combine their results.) The *QS*-based method took 291 seconds and used 36,596 K of memory, while the *GQS*-based method took only 138 seconds and used 7,088 K of memory. We were able to check the property without formula decomposition for the *GQS*-based method, but not for the *QS*-based method. For the *GQS*-based without formula decomposition, it took 178 seconds and 12,410 K of memory. This shows that decomposition can substantially improve the performance, especially when there are many processes in each priority class.

Table VIII. Unsolicited Resource Allocation

value of k	QS-based		GQS-based	
	Time	Memory	Time	Memory
2	12	2851	13	2409
3	20	4945	15	2748
4	40	9578	20	3589
5	91	19510	33	5593

In the third set of experiments, we considered systems where each of the first $k-1$ classes have exactly two processes. For $k=2$, the *QS*-based method took 17 seconds and used 4,577 K of memory, while the *GQS*-based method took 18 seconds and used 3,056 K of memory. For $k=3$, the *QS*-based method took 53 seconds and used 1,5276 K of memory, while *GQS*-based method took 47 seconds and used 6,734 K of memory. For $k=4$, we could not test the *QS*-based method at all even using formula decomposition, while we were able to test the *GQS*-based method using formula decomposition and it took 184 seconds and used 9,693 K of memory.

In the fourth set of experiments, we checked a different property by considering systems where the first $k-1$ classes have exactly one process. The property checks whether the highest priority process can ever be granted a resource without requesting the resource. We call this property the "unsolicited resource allocation". We compared the *QS*-based and *GQS*-based methods. Both methods correctly reported that this is not possible. Results are given in Table VIII.

From the above experiments, we see that the *GQS*-based method outperforms the other two methods both in time and memory in most of the cases. We believe that the few cases (when $k=2$) where the *QS*-based method perfomed better than the *GQS*-based method can be explained by the sub-optimal implementation of the *GQS*-based method in PSMC. Specifically, our current implementation of PSMC is not very efficient as we implemented it by extending SMC, and hence were forced to use some of the same data structures. In particular, the data structures used during the unwinding of the *GQS* for tracking the priority classes are not optimal.

As the number of priority classes increases, we see that the difference in performance of the *GQS*-based method comapred to the *QS*-based method increases. Also, as the number of processes in each priority class increases, we can check the above indirect mutual exclusion property only when using formula decomposition. When we apply formula decomposition, the *GQS*-based method performs better than the *QS*-based method. This is due to the following reason. To check a formula of the form $g \vee h$ using formula decomposition, in the *GQS*-based method we construct the *GQS* and then unwind it twice for checking g and h separately; thus, we need to construct the *GQS* only once. On the other hand, when we use decomposition in the *QS*-based method we check for g and h independently and we need to construct two quotient structures from scratch. Also, in the *QS*-based method, formula decomposition can only be applied at the top level, while in the *GQS*-based method, formula decomposition can sometimes be applied to subformulas inside temporal operators

as well, since we can use subformula tracking and decomposition mutually recursively.

6. CONCLUSIONS AND RELATED WORK

We have presented new algorithmic techniques for exploiting symmetry in model checking. We have generalized symmetry reduction to a larger class of automorphisms, so that systems with little or no symmetry can be verified more efficiently using symmetry reduction. We also presented novel techniques based on formula decomposition and subformula tracking. Most of the proposed algorithms have been implemented in the SMC verification system and evaluated on several protocol examples. Our experimental results indicate that the new techniques can significantly improve traditional symmetry-reduction techniques. An earlier version of this paper, that does not include implementation details, experimental results and proofs of theorems, was published in Sistla and Godefroid [2001].

As mentioned earlier, symmetry reduction in model checking has been extensively studied in Ip and Dill [1993], Clarke et al. [1993], Emerson and Sistla [1996, 1997], Emerson and Sistla [1997], Gyuris and Sistla [1999], Clarke and Jha [1995], Kurshan [1994], Emerson and Trefler [1999], and Emerson et al. [2000]. The problem of verifying properties of systems with little or no symmetry was first considered in Emerson and Trefler [1999], and Emerson et al. [2000]. The work presented in Emerson et al. [2000] also considered general automorphisms. The method used in this earlier work constructs the quotient structure of an expanded graph H and checks the correctness property on this quotient structure. This method works only for the verification of symmetric properties. In contrast, our method constructs a *GQS* which is an extension of the quotient structure where each edge has additional information in the form of an edge condition and an automorphism, and can be used to verify any property specified in CTL^*, even if the property is not symmetric. In the case of symmetric properties, we can proceed as indicated in Emerson and Trefler [1999], and Emerson et al. [2000], that is, perform standard model checking on the *GQS* without unwinding it (the edge conditions are then simply ignored during the model-checking process.)

Formula symmetry was explicitly considered in Emerson and Sistla [1996] where quotient structures are constructed with respect to automorphisms representing symmetries of the program as well as of the formula. Formula decomposition was also suggested in Emerson and Sistla [1996] for model checking. In that approach, a conjunctive formula ϕ is decomposed into its conjuncts, model checking is done with respect to each of the conjuncts separately and the results are combined to get the result for ϕ. Since each conjunct of ϕ has more symmetry than ϕ itself, the quotient structures with respect to each conjunct will be smaller than the quotient structure with respect to ϕ. Thus, formula decomposition suggested in Emerson and Sistla [1996] will involve construction of multiple smaller quotient structures (one for each of the conjuncts) rather than constructing one big quotient structure with respect to ϕ. In contrast, we use here formula decomposition during the unwinding process. While

unwinding with respect to ϕ may produce a large structure, unwinding with respect to its components will result in several smaller structures, which may be more efficient than dealing with the large structure. Also, formula symmetry is used during the construction of the quotient structure Emerson and Sistla [1996]. Here, we do not consider formula symmetry in the construction of the *GQS*, but symmetries of subformulas are used in subformula tracking dynamically as the *GQS* is unwound. Emerson and Sistla [1997] presents a verification method for $ICTL^*$ formulas. Our subformula tracking technique can also be used to efficiently verify properties specified in $ICTL^*$, in addition to being applicable to any CTL^* formula.

ACKNOWLEDGMENTS

We are grateful to the anonymous referees for their valuable comments. We are also thankful to Xiaodong Wang for his help in conducting some of the experiments.

REFERENCES

CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8*, 2, 244–263.

CLARKE, E. M., FILKORN, T., AND JHA, S. 1993. Exploiting symmetry in temporal logic model checking. In *Proceedings of the 5th International Conference on Computer Aided Verification*. Springer-Verlag, New York, 450–462.

CLARKE, E. M. AND JHA, S. 1995. Symmetry and induction in model checking. Lecture Notes in Computer Science, vol. 1000, Springer-Verlag, New York.

EMERSON, E. A., 1990. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B. Elsevier Science Publishers, Amsterdam, The Netherlands, 996–1072.

EMERSON, E., HAVLICEK, J., AND TREFLER, R. 2000. Virtual symmetry reduction. In *Proceeding of the 15th Symposium on Logic in Computer Science (LICS' 00)*. IEEE, Computer Society Press, Los Alamitos, Calif., 121–131.

EMERSON, E. A. AND LEI, C.-L. 1987. Modalities for model checking: Branching time strikes back. In *Conference Record of the 14th ACM Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 84–96.

EMERSON, F. A. AND SISTLA, A. P. 1996. Symmetry and model checking. *Form. Meth. Sys. Desi.: An Int. J. 9*, 1/2 (Aug.), 105–131.

EMERSON, E. A. AND SISTLA, A. P. 1997. Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach. *ACM Trans. Prog. Lang. Sys. 19*, 4 (July), 617–638.

EMERSON, E. A. AND TREFLER, R. J. 1999. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Proceedings of the Conference on Correct Hardware Design and Verification Methods*. 142–156.

GODEFROID, P. 1999. Exploiting symmetry when model-checking software. In *FORTE*, J. Wu, S. T. Chanson, and Q. Gao, Eds. IFIP Conference Proceedings, vol. 156. Kluwer.

GYURIS, V. AND SISTLA, A. P. 1999. On-the-fly model checking under fairness that exploits symmetry. *Form. Meth. Sys. Desi.: An Int. J. 15*, 3 (Nov.), 217–238.

IEEE. 1995. *IEEE Standard for High Performance Serial Bus. IEEE Standard 1394*. Institution of Eletrical and Eletronic Engineering.

IP, C. N. AND DILL, D. L. 1993. Better verification through symmetry. In *Computer Hardware Description Languages and their Applications*, D. Agnew, L. Claesen, and R. Camposano, Eds. Elsevier Science Publishers B.V., Amsterdam, (Ottawa, Ont., Canada). Netherland. 87–100.

KURSHAN, R. P. 1994. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, Princeton, N. J.

SIGHIREANU, M. AND MATEESCU, R. 1999. Validation of the link layer protocol of the IEEE-1394 serial bus ("firewire"): an experiment with E-LOTOS.

SISTLA, A. P. AND GODEFROID, P. 2001. Symmetry and reduced symmetry in model checking. In Lecture Notes in Computer Science, Springer-Verlag, New York, 2102

SISTLA, A. P., GYURIS, V., AND EMERSON, E. A. 2000. SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Trans. Soft. Eng. Meth. 9*, 2 (Apr.), 133–166.

Received May 2002; revised June 2003; accepted February 2004