CHECKING EXTENDED CTL PROPERTIES USING GUARDED QUOTIENT STRUCTURES

BY

XIAODONG WANG
B.S. (Wuhan University) 1997
M.S. (Institute of Software, Chinese Academy of Science) 2000

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2004

Chicago, Illinois

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to my wife, Min, who has supported me throughout the my research period. Her encouragement has helped me to get through all the difficulties.

I would like to sincerely thank my adviser, Professor Prasad Sistla for his extensive help and support throughout my graduate studies. He has been the driving force behind this research. His help and assistance made this project possible. I would like to thanks the members of the thesis committee, Professor Buy and Professor Murata.

I feel a deep sense of gratitude toward my parent who always have strong confidence on me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

State explosion problem is the major hurdle toward model checking concurrent systems consisting of a large number of processes. Symmetry based methods have been proved to be very useful to contain the state explosion problem when model checking such systems. In (1) Guarded Quotient Structure (GQS) was introduced as the extension to Quotient Structure (QS) and Annotated Quotient Structure (AQS). GQS is succinct representation of the reachability graph of a partially symmetric or even asymmetric system. Model checking employing GQS can be much more efficient than with the original system. This thesis employs GQS for model checking purpose. We extend CTL logic to a logic called COUNT CTL (CCTL) which can be used to specify properties of concurrent systems with large number of processes. Given correctness specification in CCTL, we present a model checking algorithm for symmetric or partially symmetric systems employing GQS. The thesis is organized into 6 chapters. Chapter 1 introduces the early symmetry based methods. Chapter 2 defines the syntax and semantics of CCTL formulas. Chapter 3 describes the input language in which the concurrent program and correctness specification are specified. Chapter 4 presents the model checking algorithm for CCTL formulas. Chapter 5 describes the implementation and reports experimental results. Chapter 6 concludes the thesis with discussion of related work.

CHAPTER 1

SYMMETRY BASED METHODS

There has been much interest in symmetry based methods for containing the state explosion problem in model checking. The early symmetry based methods, introduced in (2; 3; 4), exploit the symmetries in the system to identify states that are equivalent under symmetry and construct a quotient structure. The model checking is carried out on the quotient structure. This method can primarily be used for verifying symmetric properties specified in temporal logic, i.e. properties in which the atomic propositions have same truth values on equivalent states. A later approach introduced in (5) constructs an Annotated Quotient Structure (AQS) and unwinds it partially to verify a temporal property. This method can be used for checking both symmetric and asymmetric properties under various notions of fairness. In (6) the AQS method is further extended to check correctness under fairness on-the-fly. These methods have been implemented in the SMC model checker (7).

In (8; 9), the method based on quotient structures is extended to verify symmetric properties of partially symmetric and asymmetric systems. In (1), the AQS based method is extended to verify symmetric and asymmetric properties of partially symmetric and asymmetric systems as well. This method is based on constructing a Guarded Quotient Structure (GQS). It can be used to verify asymmetric properties of such systems as well.

This chapter introduces these symmetry based methods. It is organized into 3 sections. Section 1.1 introduces the QS method. Section 1.2 briefly describes the AQS method. Section 1.3 describes the GQS method.

1.1 QS **Method**

We consider a concurrent program $\mathcal{K}$ consisting of n processes. We denote the process ids by the integers $0, ..., n-1$ and let I denote the set $\{0, 1...., n-1\}$. The processes in the concurrent program communicate through variables. We call these as program variables. The name of a program variable is given by an identifier subscripted with the names of processes that share the variable. For example, $u_{1,2}$, $u_{2,3}$ are variables shared by processes $1, 2$ and by processes $2, 3$ respectively. Identifiers subscripted with only one process are local variables in that process. For example, u_1 is a local variable in process 1. A state s of the program is a mapping associating values to program variables.

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

A permutation π is called an automorphism of the graph $G = (S, E)$ if $\pi(G) = G$, i.e., $\pi(S) = S$ and $\pi(E) = E$. The set of all automorphisms of G form a group and let $\mathcal{G}$ denote this group. As has been shown in earlier works (2; 4; 3), $\mathcal{G}$ induces an equivalence relation $\equiv_{\mathcal{G}}$ on the set of states S given by $s \equiv_{\mathcal{G}} t$ if there exists a $\pi \in \mathcal{G}$, such that $\pi(s) = t$. QS method constructs a quotient structure

$QS(G, \mathcal{G})$induced by $\equiv_{\mathcal{G}}$. We simply write it as QS when G is understood in later description. This quotient structure QS can be employed to model check symmetric properties of G.

The QS method can be illustrated with the mutual exclusion protocol of two concurrent processes. In this mutual exclusion protocol, concurrent processes iterate sequentially among non-critical section (denoted by N), trying section (denoted by T) and critical section (denoted by C). These processes are coordinated such that there is at most one process in the critical section at any time. The two processes in this protocol are symmetric to each other. They go from non-critical section to trying section to ask for permission to enter the critical section. If there is only one process in trying section, it will be given the permission. If there are more than one process in the trying section, only one of them will be non-deterministically picked and given the permission. All the other processes in trying section will remain in trying section until a process leaves the critical section. At that time, another non-deterministic selection will be made to give permission to another process in trying section. The process in the critical section goes back to non-critical section and starts another iteration. The reachability graph G of this protocol is showed in Figure 1. The nodes of reachability graph G are elements belonging to the set $\{N_1, T_1, C_1\} \times \{N_2, T_2, C_2\}$. Each node s of G is a two element set. For any such node s, if $N_i \in s$ or $T_i \in s$ or $C_i \in s$ (for $i = 1, 2$), this intuitively denotes that process i is in the non-critical section or in the trying section or in the critical section, respectively. The group of automorphisms of G ($\mathcal{G}$) consists of two permutations: $Flip$ and id. Here $Flip$ is the permutation which interchanges processes 1 and 2; it defines an automorphism on the nodes of G 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 $\{D_{Flip(i)}, E_{Flip(j)}\}$. id is the identity permutation defining the identity automorphism. According to QS method, $\mathcal{G}$ induces the quotient structure in Figure 2.

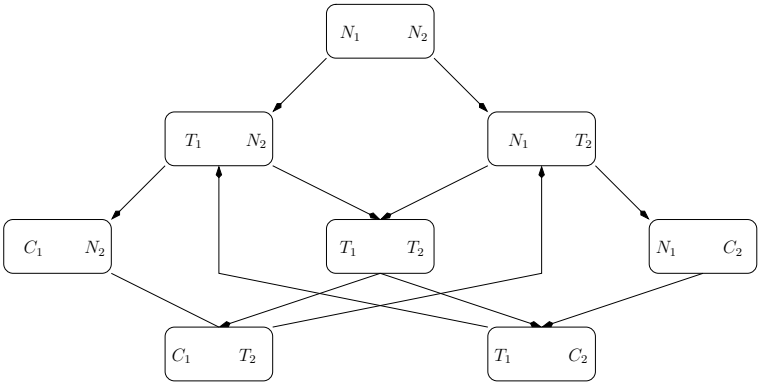

Figure 1. Reachability Graph of Mutual Exclusion Protocol of 2 Processes

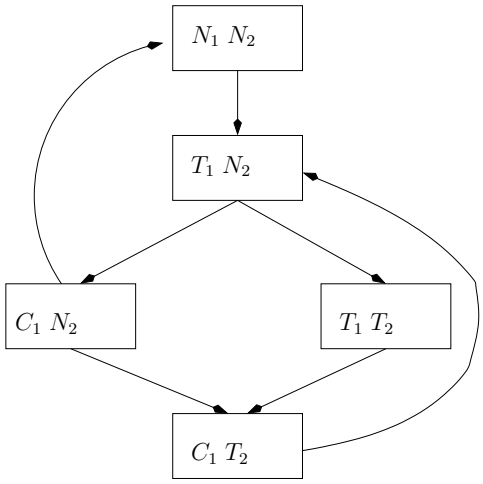

Figure 2. Quotient Structure Obtained with QS Method

It has been proved in (4) that it is equivalent to model check symmetry properties employing QS which can be much smaller than G.

1.2 AQS Method

In (5; 6), the QS method is extended to AQS method where an annotated quotient structure $AQS(G, \mathcal{G})$ is constructed. The annotated quotient structure is obtained by annotating the edges in $QS(G, \mathcal{G})$ with permutations. These permutations indicate how the process ids shift as G is compressed to obtain $AQS(G, \mathcal{G})$. We will write $AQS(G, \mathcal{G})$ as AQS when G is understood. Figure 3 gives the annotated quotient structure AQS of the above mutual exclusion protocol.

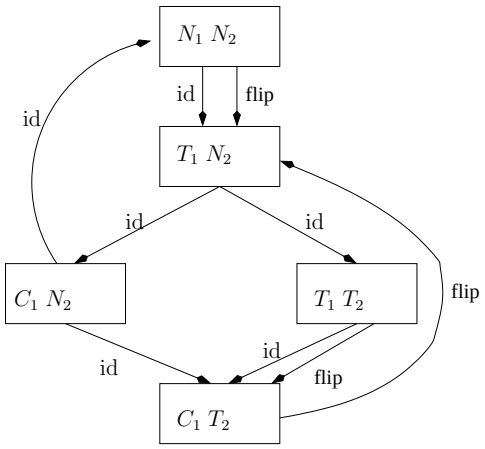

Figure 3. Quotient Structure Obtained with AQS Method

With the permutations on the edges of AQS, AQS can be employed to check both symmetric and asymmetric properties specified in CTL^* (the atomic propositions need not have the same truth values in equivalent states). (5; 6) describes how model checking can be carried out employing AQS.

When employed to model check asymmetric properties, AQS is partially unwound. During unwinding, process ids in the CTL^* properties are permuted along the unwound path. By doing this, asymmetric properties can be checked.

1.3 GQS Method

In (8; 9), the method based on the quotient structure (QS) is further extended to check symmetric properties of systems with less symmetry or no symmetry. In (1) the method based on the AQS is extended to handle systems with less or no symmetry. The later work is based on constructing a Guarded Quotient Structure (GQS). This method works as follow. In order to model check a property for a program $\mathcal{K}$, another program $\mathcal{K}'$ is considered so that the later program has lot more symmetries. Usually, $\mathcal{K}'$ is obtained from $\mathcal{K}$ by simple transformations (such as ignoring process priorities, etc.). Formally, if G and H are the global state graphs of $\mathcal{K}, K'$ respectively, then they have the same set of states, but H has more edges. Further, the set of symmetries of H is a super set of the set of symmetries of G. The GQS is constructed by first constructing the AQS of H and by adding edge conditions to the AQS. By unwinding GQS with respect to its edge conditions, we can get G from GQS.

The above method can be described in a more formal way. To make the concurrent program more symmetric, we considering another graph $H = (S, F)$ which has the same set of states as G and satisfies the following conditions: (i) $F \supseteq E$, i.e., it has all the edges of G and possibly more; (ii) its set of automorphisms is a superset of $\mathcal{G}$. We let $\mathcal{H}$ denote the set of automorphisms of H. Usually we choose H so that $\mathcal{H}$ is much larger than $\mathcal{G}$. The equivalence relation $\equiv_{\mathcal{H}}$ is extended to the edges in F in the obvious way, i.e. for $e, e' \in F$, $e \equiv_{\mathcal{H}} e'$ if there exists a permutation $\pi \in \mathcal{H}$ such that $e' = \pi(e)$. Let $\mathtt{class}(e, \mathcal{H})$ be the set of edges in the equivalence class of e. The GQS of H with respect to $\mathcal{H}$

is denoted by $GQS(H, \mathcal{H}, G)$ and is a triple $(\overline{V}, \overline{F}, C)$ defined as follows: $\overline{V} \subseteq S$ is a set of states that contains one representative for each equivalence class of $\equiv_{\mathcal{H}}$; $\overline{F} \subseteq \overline{V} \times \overline{V} \times \mathcal{H}$ is a set of labeled edges such that, for every $\overline{s} \in \overline{V}$ and $t \in S$ such that $(\overline{s}, t) \in F$, there exists an element $(\overline{s}, \overline{t}, \pi) \in \overline{F}$ such that $\pi(\overline{t}) = t$; C is a function that associates a condition $C(e)$ with each labeled edge e in $\overline{F}$; $C(e)$ denotes an edge condition such that the set of edges in $class(e, \mathcal{H})$ that satisfy $C(e)$ is exactly the set of edges $class(e, \mathcal{H}) \cap E$. The edge conditions $C(e)$ are specified by a propositional condition on program variables. In context where G and H is understood, we write $GQS(H, \mathcal{H}, G)$ as GQS.

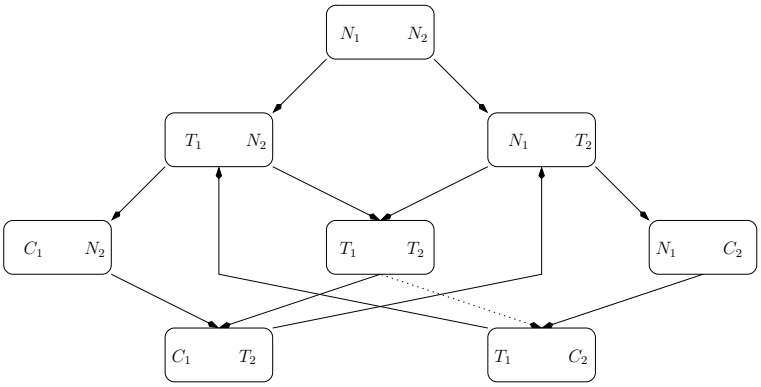

Figure 4. Reachability Graph of Mutual Exclusion Protocol with Process Priorities

The GQS method can be illustrated with the mutual exclusion protocol with process priorities. The only difference between this protocol and the protocol in in section 1.1 is that when both processes are in trying section, process 1 now has higher priority to be given the permission. Figure 4 shows

the reachability graph G of the protocol. Note that the group of automorphism of G only consists of permutation id. Permutation Flip is no longer a automorphism of G because process 1 and process 2 are not symmetric (process 1 has higher priority when both of them are trying to enter critical section). According to the QS and AQS method, the $QS(G, \mathcal{G})$ and $AQS(G, \mathcal{G})$ constructed for G are of the same size of G. That is, no reduction can be achieved with QS and AQS method. GQS method can produce a much smaller quotient structure for this protocol. To construct GQS, we first ignore the priorities by adding an edge from the node (T_1, T_2) to (T_1, C_2) (denoted by the dotted arrow in Figure 4) to G and obtain H. H is more symmetric than G and thus the corresponding quotient structure can be more succinct. The GQS corresponding to H is shown in Figure 5.

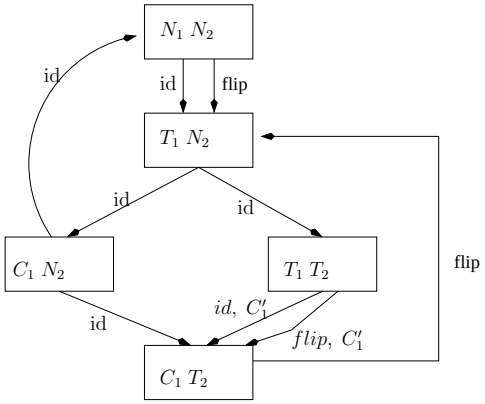

Figure 5. Quotient Structure Obtained with GQS Method

For any F in $\{N_1, N_2, T_1, T_2, C_1, C_2\}$, we let F-nodes denote the set of nodes in H that contain the element F. In the GQS in Figure 5 two edges have non-trivial edge conditions expressed as edge predicates. 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). Only edges from the node (T_1, T_2) to the node (C_1, T_2) are labeled by this predicate.

In many situations, the $GQS(H, \mathcal{H}, G)$ can be constructed directly from the concurrent program description. The GQS constructed from some asymmetric systems can be much smaller than AQS.

Model checking symmetric and asymmetric properties against symmetric and asymmetric systems can be carried out efficiently employing GQS. In (1), the authors developed a method for checking a property, specified in CTL^*, of the program by unwinding $GQS(H, \mathcal{H}, G)$ appropriately. They also presented optimizing techniques involving formula decomposition and sub formula tracking. The method given there was partially implemented in the system $PSMC$.

CHAPTER 2

CCTL LOGIC

In this chapter, we extend the branching time temporal logic CTL to a new logic called COUNT CTL (CCTL) for specifying properties of concurrent programs. CCTL has all the path quantifiers and temporal operators of CTL and the fair path quantifiers of Fair_CTL (10). In addition, it also allows another construct, called COUNT, which is a function that returns the number of processes that satisfy a given property in a given state. For example, if $C(i)$ is an atomic proposition denoting that process i is in critical section and M is a set of processes, then $COUNT(i, M, C(i))$ gives the number of processes in M that are in the critical section in the state. COUNT can be nested with temporal operators.

COUNT is useful for specifying correctness specification of system with large number of processes. With COUNT, CCTL can specify that a property should hold for some processes (or for all processes) belonging to a process class; that is, it can express process quantifiers. CCTL is more expressive than the logic ICTL considered in earlier paper (11). For example, it can express uniformly the property that the number of processes i satisfying property $P(i)$ equals the number of processes j satisfying property $Q(j)$. This property cannot be expressed in ICTL uniformly.

In this chapter, section 2.1 defines the syntax of CCTL formulas and section 2.2 defines the semantics of CCTL formulas.

2.1 Syntax of CCTL

CCTL formulas use process variables that range over process ids of the system. They only use program variables in which all the subscripts are process variables, i.e. no process id is used. CCTL formulas also use constants belonging to the domains of program variables, a function symbol COUNT, sets of process ids and comparison operators in $\{=, <, >, >=, <=\}$. In addition to path quantifiers over all paths, CCTL formula employs path quantifier $\mathsf{E}_{\mathtt{fair}}$ which quantifies over fair paths. We use standard weak process fairness in (10), i.e. a path is strongly fair if every process is either executed or disabled infinitely often.

Formally, CCTL formulas are defined as follows:

TABLE I

SYNTAX OF CCTL FORMULA

⟨ **formula** ⟩ :: ⟨ atomic formula ⟩ |

⟨ count-term ⟩ ⟨ comp-operator ⟩ ⟨ count-term ⟩ |

⟨ formula ⟩ ∧ ⟨ formula ⟩ | ¬ ⟨ formula ⟩ |

EX (⟨ formula ⟩) | E_{fair}X (⟨ formula ⟩) |

EG (⟨ formula ⟩) | E_{fair}G (⟨ formula ⟩) |

E (⟨ formula ⟩ U ⟨ formula ⟩) |

E_{fair} (⟨ formula ⟩ U ⟨ formula ⟩)

⟨ **count-term** ⟩ :: COUNT(i, M, ⟨formula⟩) |

⟨ constant ⟩

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

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

A count-term is a term of the form $COUNT(i, M, \phi(i))$ where $\phi(i)$ is another CCTL formula, M is a set of process ids and i is a process variable whose occurrences in ϕ are not bounded. A count-term can also be an integer constant. Unless otherwise stated, throughout the paper, a count term will refer to a non-constant count-term. The set M of process ids may either be explicitly given or be specified by a symbolic name which is bound to a set of process ids. We say that the occurrences of i in $COUNT(i, M, \phi(i))$ are bound.

2.2 Semantics of CCTL

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

The semantics of a formula ϕ are defined in a global state graph $G = (S, E)$ with respect to an evaluation for ϕ in an inductive manner. We denote the satisfaction of ϕ in a state s in G with respect to an evaluation f by $G, s, f \models \phi$. Since G is understood here, we simply write $s, f \models \phi$. The satisfaction

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

For any count-term u, state s, evaluation f for u, we define the value $val(s, u, f)$ as follows. If u is a constant then $val(s, u, f)$ is the constant itself. Let u be the count-term $COUNT(i, M, g(i))$. Let $g'(i)$ be the formula obtained from $g(i)$ by instantiating process variables in g according to evaluation f. Note that process variable i will not be instantiated because evaluation f does not associate any process id with bounded process variables in u such as i. For any process id c in M, let f_c denote the evaluation for $g'(i)$ such that $f_c(i) = c$. We define $val(s, u, f)$ to be the number of distinct values of c in M such that $s, f_c \models g(i)$.

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

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

CHAPTER 3

INPUT LANGUAGE

The concurrent program and its correctness specification are given in an input language extended from the input language in (12). An input consists of three sections: the program section specifying the concurrent program from which GQS is constructed; the formula section specifying the correctness specification of the program as a CCTL formula; the evaluation section giving the evaluation for the CCTL formula. Recall that CCTL formula does not use process ids. To specify the correctness specification concerning specific process ids, we replace these process ids with a new set of process variables in the formula. These process variables are then instantiated to the corresponding process ids in the evaluation section.

The concurrent program given in the program section is divided into modules where each module consists of a set of processes that are identical up to renaming. Automorphisms induced by process permutations are considered to construct GQS. 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 without process priorities. These automorphisms form a group which can be used to construct quotient structures such as GQS.

Processes in the concurrent program communicate through shared variables. A shared variable is specified by a name together with a list of process Indies. 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 transitions instantiated from the transition schemas are enabled when its condition part is evaluated to true. When an enabled transition is fired, the program variables are updated according to transition's action part.

Processes within a module may have different priorities for a transition schema. Priorities can be specified with transition schema having two index variables, where priorities are defined with respect to the non-primary index variable. We call this non-primary index variable the *secondary index variable*. For instance, the single transition schema in the controller module of the resource controller protocol given in Table II has the two index variables $\mathtt{cl}, \mathtt{k}$ appearing in it, where $\mathtt{k}$ 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 $\mathtt{X1}, \mathtt{X2}, ..., \mathtt{Xk}$ are disjoint sets of process ids belonging to the client module. In this command, each $\mathtt{Xi}$ is specified either as a symbolic representation of a set of process ids or 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 $\mathtt{Xi}$ have the same priority and, for $\mathtt{i} < \mathtt{j}$, processes belonging to $\mathtt{Xi}$ have higher priority than processes belonging to $\mathtt{Xj}$. 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 controller 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 state that the server process s must grant the waiting request to one of the clients with the highest priority.

In (12), the input language only allows one priority specification be specified for a process module. The input language is extended here by allowing multiple priority specifications to be specified with transition schemas in the same process module. It also allows user to define and use priority class which is the symbolic representation of a set of process ids that have the same priority.

The resource controller protocol given in Table II is used to illustrate the input language. In the resource controller protocol, the concurrent program consists of a controller module and a client module. The controller module in this example has only one controller process which controls the resource allocation such that only one client process can hold the resource at a time. The client module consists of several client processes. The clients request for the resource through requesting channel (implemented with shared variables $request[controller, client]$). The controller acknowledges the request from the client with highest priority through replying channel (implemented with shared variables $reply[controller, client]$) if the resource is available. The client to which the resource is granted hold the resource by changing its status to $lk[k] = 2$. It then releases the resource ($buzy[cl] = 0$) and

changes back its status ($lk[k] = 0$). The initial state of the processes are given by initial values of the program variables given in the beginning of the program.

The CCTL formula following the input program specifies the correctness specification. It asserts that no two client processes can hold the resource at the same time. Note that the universal quantifier used in the CCTL formula is a short-cut defined in chapter 2.

Since the CCTL formula does not contain any free process variable, an empty evaluation is given for the CCTL formula.

TABLE II

RESOURCE CONTROLLER PROTOCOL

```
Program

Module controller = 1;
Module client = 13;

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

i of controller;

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

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

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

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

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

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

Formula
```

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

```
Evaluation
```

CHAPTER 4

MODEL CHECKING ALGORITHM

PSMC system which implements the model checking algorithm in (1) have various limitations. The system only checked properties of the form $E(p)$ where p is a linear temporal formula. It did not implement the full CTL*. It did not implement formula decomposition and sub-formula tracking. The experiments given in (12) employed a high level formula decomposition that was manually carried out. The model checking algorithm given in this chapter checks correctness specifications given in CCTL which is an extension of CTL. It uses formula decomposition and sub-formula tracking naturally and implicitly.

We consider the model checking problems for partially symmetric and asymmetric systems when the correctness specifications are given in CCTL. We employ GQS for the model checking purpose. GQS is constructed first by ignoring the priority specifications on transitions and constructing the AQS; it then adds edge conditions to the AQS to reflect the priorities and obtains the GQS. We assume that GQS has already been constructed from the concurrent program in the input before we apply the model checking algorithm with it. The CCTL formula given in the input is then checked inductively in the initial state employing the GQS.

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

Our model-checking algorithm is top-down in its approach. The algorithm works inductively over the structure of the CCTL formula to be checked and thus employs formula decomposition in a seamless manner. With formula decomposition, the algorithm can minimize the unwinding of GQS. GQS is unwound with respect to the process ids in sub-formulas instead of all those process ids in the main formula; This increases the efficiency of the algorithm. Formula decomposition was introduced earlier in (1) to check CTL* properties. Unlike in that paper, formula decomposition is naturally incorporated into our algorithm.

This chapter is organized into 5 sections. Section 4.1 describes how GQS is employed during model checking. Section 4.2 describe how to efficiently evaluate COUNT term. Section 4.3 describes the data structures used in the model checking algorithm. Section 4.4 gives the model checking procedures. Section 4.5 analyzes the complexity of the algorithm.

4.1 Model Checking Employing GQS

We assume that $GQS(H, \mathcal{H}, G)$ has already been constructed as given section 1.3 before a CCTL formula is checked. Recall that the edges in $GQS(H, \mathcal{H}, G)$ are annotated with permutations and are also associated with edge conditions which act as guards. To model check a CCTL formula employing GQS, we change the evaluation of the process variables in the formula and track the process ids that appear in edge condition in GQS instead of unwinding GQS directly.

Consider a path $s_0, s_1, ..., s_l$ in the guarded quotient structure. Let $\pi_1, ..., \pi_l$ be the permutations labeling the corresponding edges, and $e_1, ..., e_l$ be the edge conditions of the edges respectively. Let π'_i for $i = 1, ..., l$ be the composition of the permutations $\pi_1, ..., \pi_i$ from left to right in that order. Also let $t_0, ..., t_l$ be a sequence of states such that $t_0 = s_0$ and for $i = 1, ..., l$ $t_i = \pi'_i(s_i)$. From the

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

We check the edge conditions as we traverse along a path. We can traverse an edge only if the corresponding edge condition is satisfied. This is straightforward if we use the graph H; simply evaluate the edge condition on the edge and traverse it only if it is satisfied. With $GQS(H, \mathcal{H}, G)$, we accomplish it by tracking the process ids that appear in all the edge conditions by changing it along the path. Suppose in a given GQS all the edge conditions refer to only process 0. As we traverse along a path, we change this process according to the permutations along the path. Let k_i denote this process when we reach node s_i. Initially, k_0 is set to 0. When we reach node s_i, we set k_i to be $(\pi'_i)^{-1}(0)$. To determine, if edge (s_i, s_{i+1}) can be traversed, we replace process 0 with process k_i in the edge condition e_{i+1} and evaluate this new edge condition. Again note that $k_i = (\pi_i)^{-1}(k_{i-1})$. Thus, successive values of k can be obtained by applying the inverse of the permutation labeling the edge. We use $\vec{k}$ to denote the vector

of the process ids that appearing in all edge conditions. Each process id in $\vec{k}$ will change along the path in the same way as explained.

4.2 Evaluate COUNT Term

To evaluate COUNT term $\mathrm{COUNT}(i, M, \phi(i))$ in a GQS state s, the naive method instantiates $\phi(i)$ by replacing every occurrence of process variable i in $\phi(i)$ with each process id in M. For each instantiation, $\phi(i)$ is evaluated. The value of COUNT term is the number of instantiations of $\phi(i)$ that are evaluated to true in s. This naive method can be very inefficient when M contains a large number of process ids. In the naive method, $\phi(i)$ need to be instantiated and evaluated as many as $|M|$ times where $|M|$ denotes the cardinality of M.

Our model checking algorithm exploits state symmetries of the state to evaluate $\mathrm{COUNT}(i, M, \phi(i))$. It first partitions the set of process ids M over which i ranges into equivalence classes. Instead of checking $\phi(i)$ with every process id in M, we check $\phi(i)$ for those process ids that are representatives of equivalence classes. The value of $\mathrm{COUNT}(i, M, \phi(i))$ is obtained by summing up the cardinalities of the equivalence classes if $\phi(i)$ holds with i instantiated with its representative.

In the following text, we define the equivalence relation among process ids in M. We prove that with the equivalence relation, the above method will give the same result as the naive method.

First, we need the following definition. Recall that an evaluation, for a CCTL formula or for a count-term, is a partial function whose domain is the set of free variables in the formula or the count-term, respectively. Recall that H is the reachability graph of the concurrent program ignoring the priority specification and $\mathcal{H}$ is set of automorphisms of H. We make the assumption that the set of process ids in every count term, appearing in the formula we want to check, is invariant under the permutations in

$\mathcal{G}$. That is, for every count term of the form $\mathrm{COUNT}(i, M, \phi(i))$ and for every $\pi \in \mathcal{G}$, $\pi(M) = M$. Here let $\pi(M)$ be the set $\{\pi(c) : c \in M\}$.

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

First, we consider the problem of evaluating the value of a count term $\mathrm{COUNT}(i, M, \phi(i))$ in a particular state s in which ϕ uses only process variables (no process id). As indicated before, we exploit the state symmetries (see (4; 6)) in G to evaluate the count term efficiently. For a state s in H, let $Aut(s)$ denote the set of all $\pi \in \mathcal{G}$ such that $\pi(s) = s$. We call the permutations in $Aut(s)$ as symmetries of the state s.

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

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

$$\pi(f(v)) = f(v),\ \pi(c_1) = c_2$$

.

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

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

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

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

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

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

This method can be illustrated using a slightly different resource controller protocol as the one given in Table II where no priority is specified in controller module of this protocol. To check the properties given in Table II, `check` procedure is invoked in the initial state s_0 of GQS constructed on CCTL

formula $\forall i \in client \forall j \in client(i \neq j \rightarrow AG(lk[i] \neq 2 \vee lk[j] \neq 2))$. Since no client requests for the resource at s_0, $Aut(s_0)$ consists of all the permutations over process ids of the clients. According to the definition 2, all client process ids form an equivalence class. Instead of checking $\forall j \in client(i \neq j \rightarrow AG(lk[i] \neq 2 \vee lk[j] \neq 2))$ with i instantiated to every client process id, we choose an arbitrary client process id k as representative for all clients and check $\forall j \in client(i \neq j \rightarrow AG(lk[i] \neq 2 \vee lk[j] \neq 2))$ with an evaluation which instantiates i with k.

4.3 Data Structures

The model checking algorithm associates two data structures, $L(s)$ and $marked(s)$, with each state s in GQS. $L(s)$ is a set of labels organized as a hash table. Each label in $L(s)$ is a triple of the form $(\phi, f, \vec{k})$ such that $s, f \models \phi$ using the process ids in $\vec{k}$ in the edge conditions. A checksum is computed for each label in $L(s)$ and used as the hash key for the hash table. $marked(s)$ contains the set of triples of the form $(\phi, f, \vec{k})$ which are generated when $EUCheck$, $EGCheck$, $E_{fair}GCheck$ and $efpCheck$ procedure are invoked for the first time with $\phi, f, \vec{k}, s$ as parameters. $marked(s)$ is also organized as a hash table to speed up the searching for an existing mark. Similarly, another checksum is computed for each mark and used as the hash key for this hash table.

4.4 Model Checking Procedures

In order to check with fairness, CCTL formulas need to be transformed before they are checked. We introduce a new atomic formula $Exists\_fair\_path$ for this purpose. For a state s and an empty evaluation f, $G, s, f \models Exists\_fair\_path$ iff there exists a fair path in G starting from the state s. Note that a fair path is defined as in chapter 2. Now it is easy to see that $E_{fair}X(\phi_1)$ is equivalent to $EX(\phi_1 \wedge Exists\_fair\_path)$ and $E_{fair}(\phi_1 U \phi_2)$ is equivalent to $E(\phi_1 U(\phi_2 \wedge Exists\_fair\_path))$.

We replace the sub-formulas of the above forms by the corresponding equivalent formulas and model check for them.

This section presents all the model checking procedures. Subsection 4.4.1 gives the main procedure which invokes other procedures for some types of sub-formulas. Subsection 4.4.2 gives the procedure to check sub-formulas of the form $E(\phi_1 U \phi_2)$. Subsection 4.4.3 gives the procedure to check sub-formulas of the form $EG(\phi)$. Subsection 4.4.4 gives the procedure to check $Exists\_fair\_path$ atomic formula. Subsection 4.4.5 gives the procedure to check sub-formulas of the form $E_{fair}G(\phi)$.

4.4.1 check Procedure

As indicated earlier, our model-checking algorithm uses lazy evaluation and works in top-down fashion. Initially check procedure is invoked on formula ϕ in the initial state with an evaluation of ϕ. The initial values of the parameter $\vec{k}$ are the process ids that appear in the edge conditions.

The check procedure is given in Table III. Given a GQS and a CTL formula ϕ, the procedure checks if $s \models \phi$. The check procedure first verifies if the state s has already been labeled with $(\phi, f, \vec{k})$ or $(\neg\phi, f, \vec{k})$ where ϕ, f, $\vec{k}$ are the parameters for this invocation of check procedure. If either of them is labeled in s, this procedure will return with appropriate true value. If none of them is labeled, check works inductively on the structure of ϕ. If ϕ is an atomic formula other than $Exist\_fair\_path$, it is evaluated directly in s. If $\phi = Exist\_fair\_path$, ϕ is evaluated by procedure efpCheck. If ϕ is $\phi_1 \wedge \phi_2$ then check is invoked on ϕ_1 first and, if ϕ_1 holds, on ϕ_2 using the evaluations f', f'' respectively; here f', f'' are restrictions of f to the free variables of ϕ_1 and ϕ_2 respectively. The cases when $\phi = EG(\phi_1)$, $\phi = E(\phi_1 U \phi_2)$ and $\phi = E_{fair}G(\phi_1)$ are evaluated by procedure EGCheck, EUCheck and $E_{fair}G$Check respectively. The case when $\phi = (COUNT(i, M, \phi_1) = c)$ is handled

TABLE III

CHECK PROCEDURE

Procedure $check\ (\phi, f, \vec{k}, s)$

1. If $(\phi, f, \vec{k}) \in L(s)$, then return true
2. If $(\neg\phi, f, \vec{k}) \in L(s)$, then return false
3. Switch(ϕ)

case ϕ is an atomic formula:

if s satisfies $\phi[f]$ then return true, else return false;

case $\phi = \neg\phi_1$:

$flag \leftarrow \neg check(\phi_1, f, \vec{k}, s)$;

case $\phi = (\phi_1 \wedge \phi_2)$:

$flag \leftarrow check(\phi_1, f', \vec{k}, s) \wedge check(\phi_2, f'', \vec{k}, s)$;

case $\phi = EX\phi_1$:

if there is at least one edge from s ($s \xrightarrow{\pi, e(\vec{c})} t$) such that $(s, \pi(t)) \models e(\vec{k}/\vec{c})$ and

$check(\pi^{-1}(\phi_1), \pi^{-1}(f), \pi^{-1}(\vec{k}), t)$

then $flag \leftarrow true$

else $flag \leftarrow false$

case $\phi = EG(\phi_1)$:

$flag \leftarrow EGCheck(s, EG(\phi_1), f, \vec{k})$;

case $\phi = E_{fair}G(\phi_1)$:

$flag \leftarrow E_{fair}GCheck(s, EG(\phi_1), f, \vec{k})$;

case $\phi = E(\phi_1 U \phi_2)$:

$flag \leftarrow EUCheck(s, E(\phi_1 U \phi_2), f, \vec{k})$;

case ϕ is $(COUNT(i, \phi_1) = c)$:

$sum \leftarrow 0$;

for every equivalence class x of $\approx_{s,f}$

if $\exists j \in x$ such that $(\neg\phi_1, f_j, \vec{k}) \in L(s)$ then continue;

if $\exists j \in x$ such that $(\phi_1, f_j, \vec{k}) \in L(s)$,

then $sum \leftarrow sum + |x|$,

continue;

choose some $j \in x$; if $Check(\phi_1, f_j, \vec{k}, s)$ then $sum \leftarrow sum + |x|$;

if $sum = c$ then $flag \leftarrow True$;

else $flag \leftarrow False$;

4. If $flag$, then add $(\phi, f, \vec{k})$ to $L(s)$, return true
5. If $\neg flag$, then add $(\neg\phi, f, \vec{k})$ to $L(s)$, return false

as explained in section 4.2. Formulas, such as $(\mathrm{COUNT}(i, M, \phi_1) = \mathrm{COUNT}(i, M, \phi_2))$, can be handled similarly with evaluation f restricted to f' and f'' accordingly as in the case of $\phi_1 \wedge \phi_2$. At the end of this procedure, a new label is created and stored in $L(s)$ according to the checking result indicated by $flag$.

4.4.2 EUCheck Procedure

EUCheck procedure is invoked from check procedure if the sub-formula to be checked is of the form $E(\phi_1 U \phi_2)$. In the EUCheck procedure given in Table IV, the $GQS(H, \mathcal{H}, G)$ is traversed appropriately. In the beginning of EUCheck procedure, a new mark is created and stored with s to indicate that EUCheck is being invoked on s with the invocation parameters for the first time. EUCheck will not be invoked on a state s which has already been marked with the invocation parameters. In the for loop of EUCheck procedure, if $(\pi^{-1}(\phi), \pi^{-1}(f), \pi^{-1}(\vec{k})) \in marked(t)$ but $(\pi^{-1}(\phi), \pi^{-1}(f), \pi^{-1}(\vec{k})) \notin L(t)$, it implies that either there is a cycle and the eventuality is not fulfilled, or its sub-formula has been checked and has been found to be not satisfied.

TABLE IV

EUCHECK PROCEDURE

Procedure $\mathtt{EUCheck}(s, \mathtt{E}(\phi_1 \mathtt{U} \phi_2), f, \vec{k})$

$\phi \leftarrow \mathtt{E}(\phi_1 \mathtt{U} \phi_2)$;

add $(\phi, f, \vec{k})$ to $\mathtt{marked}(s)$;

if $\mathtt{check}(\phi_2, f, \vec{k}, s)$, then return true;

if $\neg\mathtt{check}(\phi_1, f, \vec{k}, s)$, then return false;

for each edge from s $(s \xrightarrow{\pi, e(\vec{c})} t)$ where $(s, \pi(t)) \models e(\vec{k}/\vec{c})$

if $(\phi, \pi^{-1}(f), \pi^{-1}(\vec{k})) \in \mathtt{marked}(t)$ and $(\phi, \pi^{-1}(f), \pi^{-1}(\vec{k})) \in L(t)$ then

return true;

if $(\phi, \pi^{-1}(f), \pi^{-1}(\vec{k})) \notin \mathtt{marked}(t)$ then

flag $\leftarrow \mathtt{check}(\phi, \pi^{-1}(f), \pi^{-1}(\vec{k}), t)$;

if flag then

add $(\mathtt{E}(\phi_1 \mathtt{U} \phi_2), f, \vec{k})$ to $L(s)$ and return true

add $(\neg\phi, f, \vec{k})$ to $L(s)$ and return false

4.4.3 EGCheck Procedure

EGCheck procedure is invoked from check procedure if the sub-formula to be checked is of the form $EG(\phi_1)$. The EGCheck procedure given Table V is similar as EUCheck procedure and thus can be easily understood.

TABLE V

EGCHECK PROCEDURE

Procedure $EGCheck(s, EG(\phi_1), f, \vec{k})$

$\phi \leftarrow EG(\phi_1)$;

add $(\phi, f, \vec{k})$ to $marked(s)$;

if $\neg check(\phi_1, f, \vec{k}, s)$, then add $(EG(\phi_1), f, \vec{k})$ to $L(s)$, return false;

for each edge from s $(s \xrightarrow{\pi, e(\vec{c})} t)$ where $(s, \pi(t)) \models e(\vec{k}/\vec{c})$

if $(\phi, \pi^{-1}(f), \pi^{-1}(\vec{k})) \in marked(t)$ and $(\phi, \pi^{-1}(f), \pi^{-1}(\vec{k})) \in L(t)$ then

return true;

if $(\phi, \pi^{-1}(f), \pi^{-1}(\vec{k})) \notin marked(t)$ then

flag $\leftarrow check(\phi, \pi^{-1}(f), \pi^{-1}(\vec{k}), t)$;

if flag then

add $(EG(\phi_1), f, \vec{k})$ to $L(s)$ and return true

add $(\neg EG(\phi_1), f, \vec{k})$ to $L(s)$ and return false

4.4.4 efpCheck **Procedure**

efpCheck procedure is invoked by check procedure when atomic formula Exist_fair_path is to be checked in s. efpCheck procedure is given in Table VI. It is based on the standard algorithm which finds out the maximum strongly connected components in a directed graph. Starting from state s where Exist_fair_path is to be checked, efpCheck explores the GQS using depth first search. The depth first search uses two stacks. One stack is used to store the part of the path that has been explored. This stack is implemented implicitly as the calling stack in efpCheck procedure which recursively calls itself. The other stack, the maximum strongly connected component stack, is explicit in the efpCheck procedure. It is used to store the explored part of the maximum strongly connected component containing s. When the depth first search starting from s is finished, all states of the maximum strongly connected component containing s are on the stack. It is assumed by this procedure that the maximum strongly connected component stack is created by check procedure before efpCheck is invoked.

Like EUCheck procedure, a new mark is created and stored with s when efpCheck is invoked on s for the first time with the parameters. For each state s on which efpCheck is invoked, it is associated with a partition array. This array is initialized when the mark is created such that s.partition[i].enabled equals to True only if process i has a transition enabled in s. For all processes i, s.partition[i].executed is initialized to False. They are updated to true later if an enabled transition in process i from s is part of the maximum strongly connected component containing s. State s, along with the associated partition array, is then pushed onto the maximum strongly connected component stack.

TABLE VI

EFPCHECK PROCEDURE

Procedure $efpCheck(s, Exist\_Fair\_Path, f, \vec{k})$

add $(Exist\_Fair\_Path, f, \vec{k})$ to $marked(s)$;

Initialize $s.partition$ and Push s onto maximum strongly connected component stack

for each edge from s ($s \xrightarrow{\pi, e(\vec{c})} t$) where $(s, \pi(t)) \models e(\vec{k}/\vec{c})$

if $(Exist\_Fair\_Path, \pi^{-1}(f), \pi^{-1}(\vec{k})) \in marked(t)$ and

$(Exist\_Fair\_Path, \pi^{-1}(f), \pi^{-1}(\vec{k})) \in L(t)$

then add $(Exist\_Fair\_Path, f, \vec{k})$ to $L(s)$ and return true;

if $(Exist\_Fair\_Path, \pi^{-1}(f), \pi^{-1}(\vec{k})) \notin marked(t)$ and $s \xrightarrow{\pi, e(\vec{c})} t$ is non-tree-edge

Construct $partition_e$ for this edge and Combine $s.partition$ with $partition_e$;

Update $s.partition$'s execution bits;

if $(Exist\_Fair\_Path, \pi^{-1}(f), \pi^{-1}(\vec{k})) \notin marked(t)$ and $s \xrightarrow{\pi, e(\vec{c})} t$ is tree-edge

if $(efpCheck(Exist\_Fair\_Path, \pi^{-1}(f), \pi^{-1}(\vec{k}), t))$

add $(Exist\_Fair\_Path, f, \vec{k})$ to $L(s)$ and return true;

else

Combine $t.partition$ with $s.partition$;

Update $s.partition$'s execution bits;

if $s.partition$ indicates there is a fair path from s

add $(Exist\_Fair\_Path, f, \vec{k})$ to $L(s)$ and return true;

else

Pop out those states from s in the maximum strongly component stack

add $(\neg Exist\_Fair\_Path, f, \vec{k})$ to $L(s)$ and return false;

For each edge $s \xrightarrow{\pi, e(\vec{c})} t$ from s, it is classified either as a non-tree-edge or a tree-edge according to (6). A tree-edge goes from a reachable state from s to another state which has not been explored before. A non-tree-edge forms a loop by connecting a reachable state from s to a upper-level reachable state which is on the maximum strongly connected component stack. If $s \xrightarrow{\pi, e(\vec{c})} t$ is a tree-edge, $efpCheck$ is invoked on t and $s.partition$ is updated accordingly after the invocation. If $s \xrightarrow{\pi, e(\vec{c})} t$ is a non-tree-edge, $efpCheck$ will not be invoked on t. $s.partition$ is updated according to the loop formed when this non-tree-edge is taken. After all edges $s \xrightarrow{\pi, e(\vec{c})} t$ from s have been checked and if there is no fair path found from t, $s.partition$ is examined to determine if there is a fair path from s. If there is, the procedure returns true; otherwise, all those states explored from s (including s) that are on the maximum strongly connected component stack are popped out from the stack and the procedure returns false.

4.4.5 $E_{fair}GCheck$ Procedure

$E_{fair}GCheck$ procedure is invoked by $check$ procedure when the sub-formula to be checked is of the form $E_{fair}G$. Procedure $E_{fair}GCheck(s, E_{fair}G(\phi_1), f, \vec{k})$ is adopted from $efpCheck$ with some minor changes. It can be looked as applying $efpCheck$ over a reduced graph of G where each state satisfies ϕ_1. This reduced graph need not to be constructed explicitly. Instead, it can be constructed implicitly by avoiding those states that do not satisfy ϕ_1. This procedure is given in Table VII. It is assumed by this procedure that $s, f \models \phi_1$ using the process ids in $\vec{k}$ in the edge conditions when $E_{fair}GCheck$ is invoked from $check$ procedure; Otherwise, $s, f \not\models E_{fair}G(\phi_1)$ according to the semantics of $E_{fair}G(\phi_1)$.

TABLE VII

E_{FAIR}GCHECK PROCEDURE

Procedure $E_{fair}GCheck(s, E_{fair}G(\phi_1), f, \vec{k})$

add $(E_{fair}G(\phi_1), f, \vec{k})$ to $marked(s)$ true;

Initialize $s.partition$ and Push s onto maximum strongly connected component stack

for each edge from s ($s \xrightarrow{\pi, e(\vec{c})} t$) where $(s, \pi(t)) \models e(\vec{k}/\vec{c})$ and $check(\phi_1, \pi^{-1}(f), \pi^{-1}(\vec{k}), t)$

if $(E_{fair}G(\phi_1), \pi^{-1}(f), \pi^{-1}(\vec{k})) \in marked(t)$ and

$(E_{fair}G(\phi_1)), \pi^{-1}(f), \pi^{-1}(\vec{k})) \in L(t)$

then add $(E_{fair}G(\phi_1), f, \vec{k})$ to $L(s)$ and return true;

if $(E_{fair}G(\phi_1), \pi^{-1}(f), \pi^{-1}(\vec{k})) \notin marked(t)$ and $s \xrightarrow{\pi, e(\vec{c})} t$ is non-tree-edge

Construct $partition_e$ for this edge and Combine $s.partition$ with $partition\_e$;

Update $s.partition$'s execution bits;

if $(E_{fair}G(\phi_1), \pi^{-1}(f), \pi^{-1}(\vec{k})) \notin marked(t)$ and $s \xrightarrow{\pi, e(\vec{c})} t$ is tree-edge

if $(E_{fair}GCheck(E_{fair}G(\phi_1)), \pi^{-1}(f), \pi^{-1}(\vec{k}), t))$

add $(E_{fair}G(\phi_1), f, \vec{k})$ to $L(s)$ and return true;

else

Combine $t.partition$ with $s.partition$;

Update $s.partition$'s execution bits;

if $s.partition$ indicates there is a fair path from s

add $(E_{fair}G(\phi_1), f, \vec{k})$ to $L(s)$ and return true;

else

Pop out those states from s in the maximum strongly component stack

add $(\neg E_{fair}G(\phi_1), f, \vec{k})$ to $L(s)$ and return false;

4.5 Complexity Analysis

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

CHAPTER 5

IMPLEMENTATION AND EXPERIMENTAL RESULTS

5.1 Implementation

We have implemented the model-checking algorithm as an extension of the SMC model-checker (7). The tool first constructs GQS from the input concurrent program. Then the input CCTL formula is checked inductively by invoking the $check$ procedure on the formula in the initial state of GQS with the input evaluation.

Even though formula decomposition and sub-formula tracking are used in our model checking algorithm to minimize the unwinding of GQS (indirectly), the states explored during unwinding can still be a very large number for a real industry level protocol. For example, to check the mutual exclusion property of the cache coherence protocol of 4 clients, the algorithm will explore more than 100,000 states. Recall that in chapter 4, when EUCheck procedure, EGCheck procedure, E_{fair}GCheck procedure and $efpCheck$ procedure are invoked for the first time with the invocation parameters on a state, a mark (of the form $(\phi, f, \vec{k})$) will be generated and stored with the state. After a formula has been evaluated in a state, a new label (of the form $(\phi, f, \vec{k})$) will be generated and stored to record the evaluation result. The labels and marks may use up a lot of memory if they are stored with a large number of states. To reduce the memory used by the labels and marks, we replace the f component and $\vec{k}$ component in the mark or label with a permutation π such that $\pi(f_0) = f$ and $\pi(\vec{k}_0) = \vec{k}$. Here f_0 denotes the evaluation given in the input and $\vec{k}_0$ denotes the vector of the process ids that appearing in all edge conditions of

the GQS. This permutation can be computed easily by composing the inverse of those permutations along the path from initial state to the state against which the formula is check.

As shown in the procedures in chapter 4, we often need to determine if a state in GQS has already been associated with a given label or mark. Given above representation of the label and mark, we need to reconstruct the set of labels or marks associated with the state before we can search among them. The reconstruction can be time-consuming, especially when the same label or mark needs to be reconstructed again and again. To speed up this searching process, we compute and store a checksum for each label and mark when they are generated. The checksum is computed from the f component and $\vec{k}$ component of a label or mark. The set of labels or marks associated with a state are organized as hash table using the checksum as the hash key. Before the label or mark is reconstructed, the checksum is compared. Only if the checksum matches, the labels or marks will be reconstructed and compared. This substantially speeds up the searching process while minimizing the memory consumption.

5.2 Experimental Results

This tool has been applied to the resource controller protocol as well as industry level protocol such as cache coherency protocol. We observed significant performance improvement when checking some useful properties over these protocols.

We checked the mutual exclusion property $\forall i \in client\, \forall j \in client(i \neq j \rightarrow AG(lk[i] \neq 2 \vee lk[j] \neq 2))$ with resource controller protocol. Here $lk[j] \neq 2$ denotes that client j is not in critical section. For cache coherence protocol, we checked the property $\forall i \in client\, \forall j \in client(i \neq j \rightarrow AG(cache[i] \neq exclusive \vee cache[j] \neq exclusive))$ asserting that no two clients can hold the cache

line in exclusive mode simultaneously. Here $cache[i] \neq exclusive$ denotes that client i does not hold the cache line exclusively. The experimental results are presented in Table VIII.

TABLE VIII

EXPERIMENTAL RESULTS

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

The column $quant\_elim$ indicates if our approach of quantifier elimination through state symmetry is employed. Without using quantifier elimination, the protocols are checked in the naive approach. $mark\#$ gives the number of marks generated in the experiments. Recall that a mark is generated when

$\mathtt{EUCheck}$, $\mathtt{EGCheck}$, $\mathtt{E_{fair}GCheck}$ or $\mathtt{efpCheck}$ procedure is invoked for the first time on a state. The running time in column $\mathtt{time(s)}$ is given by running the experiments on a Intel Pentium M 1.3G PC. In some experiments where some path is too long to be held in the calling stack, we encountered stack overflow. This is indicated with * in the table. Our tool showed performance improvement for both protocols. While with cache coherence protocol of 4 clients, our tool ran 20-30 percentage faster by utilizing quantifier elimination, we got much more performance improvement with resource controller protocol of more than 10 clients. This confirms that this tool is especially useful when verifying properties with quantifiers over a large set of processes.

CHAPTER 6

CONCLUSION

This thesis introduce CCTL which is an extension of CTL. It also presents a model check algorithm which can efficiently check CCTL formula employing GQS without unwinding it completely. The algorithm exploits state symmetries. Here, for the first time, we use them to model-check for complex properties, using the COUNT functions and process quantifiers, efficiently. The algorithm uses formula decomposition and sub-formula tracking naturally and implicitly. The formula decomposition is used in the sense that when we invoke the check procedure on a sub-formula ϕ we only track the process ids required for it. Similarly sub-formula tracking is used implicitly.

CITED LITERATURE

1. Sistla, A. P. and Godefroid, P.: Symmetry and reduced symmetry in model checking. In CAV, pages 91–103, 2001.

2. Clarke, E. M., Jha, S., Enders, R., and Filkorn, T.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design, 9(1/2):77–104, 1996.

3. C.N. Ip and D.L. Dill: Better verification through symmetry. In Computer Hardware Description Languages and their Applications, eds. D. Agnew, L. Claesen, and R. Camposano, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

4. Emerson, E. A. and Sistla, A. P.: Symmetry and model checking. Formal Methods in System Design, 9(1/2):105–131, 1996.

5. Emerson, E. A. and Sistla, A. P.: Utilizing symmetry when model checking under fairness assumptions: An automata-theoretic approach. In CAV, pages 309–324, 1995.

6. Gyuris, V. and Sistla, A. P.: On-the-fly model checking under fairness that exploits symmetry. Formal Methods in System Design: An International Journal, 15(3):217–238, November 1999.

7. Sistla, A. P., Gyuris, V., and Emerson, E.: SMC: a symmetry-based model checker for verification of safety and liveness properties. ACM Transactions on Software Engineering and Methodology, 9(2):133–166, April 2000.

8. Emerson, E. A. and Trefler, R. J.: From asymmetry to full symmetry: New techniques for symmetry reduction in model checking, August 08 1999.

9. Emerson, E., Havlicek, J., and Trefler, R.: Virtual symmetry reduction. In 15th Symposium on Logic in Computer Science (LICS' 00), pages 121–131, Washington - Brussels - Tokyo, June 2000. IEEE.

10. Emerson, E. A. and Lei, C.-L.: Temporal reasoning under generalized fairness constraints. In STACS, pages 21–36, 1986.

11. Clarke, E. M., Grumberg, O., and Browne, M. C.: Reasoning about networks with many identical finite-state processes. In PODC '86: Proceedings of the fifth annual ACM symposium on Principles of distributed computing, pages 240–248. ACM Press, 1986.

12. Sistla, A. P. and Godefroid, P.: Symmetry and reduced symmetry in model checking. ACM Transactions on Programming Languages and Systems, 26(4):702–734, July 2004.

VITA

NAME: Xiaodong Wang

EDUCATION: Bachelor of Science in Computer Science,

Wuhan University, China, 1997.

Master of Engineering in Computer Science,

Institute of Software, Chinese Academy of Science, China, 2000

Master of Science in Computer Science,

University of Illinois at Chicago (UIC), Chicago, Illinois, 2004.

EXPERIENCE: UIC Research Assistant, Computer Science Department, 01/2003 to 12/2004.

Research focus on model checking concurrent system

Software Engineer, Wireless Department, Bell-labs China, 07/2000 to 11/2002.

Software Development in GSM and CDMA Base Station Controller project.

HONORS: Member of Bell-labs Golden Award Winner Team, 2001.

Lucent Technology Excellence Award winner, 2002