

Copyright © 1997, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

**STRUCTURAL SYMMETRIES AND MODEL
CHECKING**

by

Gurmeet Singh Manku

Memorandum No. UCB/ERL M97/92

22 December 1997

**STRUCTURAL SYMMETRIES AND MODEL
CHECKING**

Copyright © 1997

by

Gurmeet Singh Manku

Memorandum No. UCB/ERL M97/92

22 December 1997

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Structural Symmetries and Model Checking

Gurmeet Singh Manku

Abstract

We present a fully automatic framework for identifying symmetries in structural descriptions of digital circuits and CTL* formulas and using them in a model checker. We show how the set of sub-formulas of a formula can be partitioned into equivalence classes so that truth values for only one sub-formula in any class need be evaluated for model checking. We unify and extend the theories developed by Clarke *et al* [CEFJ96] and Emerson and Sistla [ES96] for symmetries in Kripke structures. We formalize the notion of structural symmetries in net-list descriptions of digital circuits and CTL* formulas. We show how they relate to symmetries in the corresponding Kripke structures. We also show how such symmetries can automatically be extracted by constructing a suitable directed labeled graph and computing its automorphism group. We present a novel fast algorithm for solving the graph automorphism problem for directed labeled graphs.

1940

iv
v
vi
vii
viii
ix
x
xi
xii
xiii
xiv
xv
xvi
xvii
xviii
xix
xx
xxi
xxii
xxiii
xxiv
xxv
xxvi
xxvii
xxviii
xxix
xxx

To my parents

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

Contents

List of Figures	vii
List of Tables	viii
1 Introduction	1
1.1 Symmetries	1
1.1.1 CTL* Model Checking	2
1.1.2 Scalarsets	2
1.2 New Ideas	3
1.3 Organization of Report	4
2 Preliminaries	5
2.1 Kripke Structures	5
2.2 Temporal Logic CTL*	6
2.2.1 Syntax	6
2.2.2 Semantics	6
2.2.3 Logically Equivalent Formulas	7
2.2.4 Structurally Equivalent Formulas	7
2.3 Model Checking Problem	7
2.4 Permutation Groups	8
2.4.1 Definitions	8
2.4.2 Definition of $G \bowtie H$	9
2.5 Graph Isomorphism Problem	10
2.6 Graph Automorphism Problem	11
3 Symmetric Sub-formulas	13
3.1 Definitions	13
3.1.1 Definition of $Aut_M L$	13
3.1.2 Definition of $Aut_M L \cdot AP$	13
3.1.3 Definition of $Aut_M L \cdot X$	14
3.2 Theory for Symmetric Sub-formulas	14
3.3 Equivalence Classes of Sub-formulas	16
3.4 Model Checking with Equivalence Classes	16
3.5 Computing Equivalence Classes of Sub-formulas	17

3.5.1	Definition of $Aut_f AP \cdot SF$	17
3.5.2	Definition of \approx_s^{GMH}	18
3.5.3	Putting It All Together	18
3.5.4	Critique	18
3.6	Generating New Formulas	18
3.7	Summary	19
4	Quotient Structures	21
4.1	Definitions	21
4.1.1	Definition of $[s]_G$ and ξ_G	21
4.1.2	Definition of Quotient Structure M_G	22
4.2	Theory for Quotient Structures	22
4.2.1	The Path Correspondence Lemma	22
4.2.2	Label Preserving Quotients	22
4.2.3	Definition of $Aut_f AP \cdot MPS$	23
4.2.4	Larger Label Preserving Quotients	23
4.2.5	An Extension of Emerson-Sistla Theorem	24
4.2.6	Summary	24
4.3	Model Checking with Quotient Structures	25
4.3.1	Explicit State Enumeration Using ξ_G	25
4.3.2	Symbolic Model Checking Using ξ_G	25
4.4	The Canonical State Problem	27
4.4.1	Equivalence Relation Approach	27
4.4.2	Direct Computation of ξ_G	27
4.4.3	Other Ideas	29
4.5	Research Problems	29
5	Structural Symmetries	31
5.1	Motivation	31
5.2	BLIF-MV	32
5.2.1	Characterizing a BLIF-MV Circuit	33
5.2.2	Definition of Structural Symmetries	34
5.2.3	Relationship with $Aut_M L \cdot AP$	34
5.2.4	Other Structural Languages	35
5.3	Graphs for BLIF-MV Circuits	35
5.3.1	Tables	35
5.3.2	Latches	36
5.3.3	Inputs and Outputs	37
5.3.4	Interconnection Signals	37
5.3.5	Relationship with $Aut_M L \cdot AP$	37
5.3.6	The “=” Construct	39
5.3.7	Is Every Group Possible?	40
5.3.8	Identifying Scalarsets	40
5.4	CTL* Formulas	40
5.4.1	Graphs for Computing $H \leq Aut_f AP \cdot SF$	40

5.4.2	Graphs for Computing $H \leq \text{Aut}_f AP \cdot MPS$	41
5.5	Computing $G \bowtie H$	42
5.6	The Big Picture	42
6	Computing Graph Automorphisms	45
6.1	Definitions	45
6.1.1	Properties of Bipartitions	47
6.1.2	Problem Definition	47
6.2	Branch and Bound Algorithm	49
6.2.1	The Bounding Step	49
6.2.2	The Branching Step	50
6.2.3	Proof of Correctness	50
6.2.4	Time Complexity	51
6.3	Refinement	52
6.3.1	Computing U using Vertex Invariants	52
6.3.2	Other Vertex Invariants	52
6.3.3	Repeated Refinement	53
6.4	Computing the Automorphism Group of a Graph	53
6.4.1	Base and Strong Generating Set	53
6.4.2	Jerrum's Representation	54
6.4.3	Graph Automorphism Algorithms	55
6.4.4	Base Change Algorithms	56
6.4.5	Extending our Algorithm	56
6.4.6	Is Every Group Possible?	57
6.5	Why a New Graph Automorphism Program?	57
6.5.1	Domain Specific Optimizations	58
6.5.2	Further Optimizations	59
7	Results and Future Work	61
7.1	Results	61
7.2	Conclusions	64
7.3	Future work	65
	Bibliography	69

List of Figures

4.1	Modified explicit state enumeration algorithm using ξ_G	26
4.2	Algorithm for computing ξ from a set of generators for a group acting on L	28
4.3	Flow diagram for model checking using quotient structures.	30
5.1	A BLIF-MV table for the carryover bit in a 3-bit adder.	32
5.2	Graph for a BLIF-MV table.	37
6.1	Algorithm for finding an automorphism, given graph A and bipartition P	48
7.1	Labeled directed graph obtained from ping-pong.v.	62

List of Tables

7.1	Some statistics for graphs produced for different examples.	63
7.2	Performance of branch and bound algorithm for different examples.	63

Acknowledgements

I thank my research advisor, Professor Robert Brayton for his encouragement and support. He gave me ample time and freedom to choose a research problem. I thank Ramin Hojati for introducing me to symmetries. His insightful questions helped me clarify my thoughts on several occasions.

I also thank Ramin Hojati and Shaz Qadeer for carefully reading a preliminary draft of this report and Amit Mehrotra for helping me with his knowledge of L^AT_EX.

Chapter 1

Introduction

As design cycles become shorter, circuit size and complexity grow larger and costs of design errors increase in magnitude [Hof95], traditional approaches like simulation are being augmented by more complete techniques like formal verification [CW⁺96]. One popular verification technique is model checking [Eme90], in which temporal logic formulas are verified on a non-deterministic finite state machine that represents the system under scrutiny. Model checking algorithms [CES86, CMB90, BCMD90, TSL⁺90, BCL⁺94] typically explore the states of such a finite state machine. A major problem faced by state space exploration techniques is due to the fact that the size of the machine could be exponential in the size of the system description. This phenomenon is commonly referred to as *State Space Explosion*.

Several techniques are being developed for countering the state space explosion problem. Partial order methods [Pel93, God96], abstraction [Lon93], compositional approaches [CLM89] and symmetries [Ip96, CEFJ96, ES96] appear to be most promising. To date, none of these techniques has been fully automated.

1.1 Symmetries

Hardware circuits and distributed algorithms abound with symmetries. Hardware systems with symmetries include memories, caches, arithmetic circuits and distributed memory architectures. Several distributed algorithms exhibit symmetry. The same holds for concurrent programs. Typically all designs that have replicated subcomponents exhibit symmetry.

Huber, Jepsen and Jensen [HJJ84] and Starke [Sta91] have investigated the use of symmetries for expediting reachability analysis for Petri nets. An algebraic approach for reducing the cost of protocol analysis has been proposed by Kurshan [Kur87] and Attie and Emerson [AE89]. The approach uses quotient structures induced by automorphisms of the system. For example, symmetry between 0 and 1 in the alternating bit protocol is factored out to reduce the size of the state space by one half. Verification of cache consistency protocols has been shown to benefit from symmetries. See the survey by Pong and Dubois [PD97] for details. Symmetries have proven useful even for transistor-level verification of digital circuits [PB97].

1.1.1 CTL* Model Checking

Emerson and Sistla [ES96] and Clarke *et al* [CEFJ96] have developed a theory of symmetries for CTL* model checking. Both papers show how symmetries in Kripke structures and CTL* formulas allow the construction of a smaller sized *quotient* structure such that the formula need be verified only for the quotient. Clarke *et al* focus on symbolic techniques and study the complexity of related BDDs. Using carefully constructed formulas, they are able to verify the correctness of IEEE Futurebus+ standard [CGH⁺95]. Emerson and Sistla focus on systems composed of isomorphic subprocesses. In both works, symmetries are specified manually. Theory for using symmetries in the presence of fairness constraints has been developed by Emerson and Sistla [ES95]. More recently, Gyuris and Sistla [GS97] have developed an on-the-fly model checker that utilizes symmetries under fairness. Theory for combining partial orders and symmetries has been developed by Emerson, Jha and Peled [EJP97].

1.1.2 Scalarsets

Ip and Dill [ID93, ID96] use symmetries for speeding up verification of safety properties using explicit state exploration techniques for designs specified in a guarded command language. They propose augmentation of the language itself by introducing a new data type called *scalarset*. A set of state variables is said to constitute a scalarset if they are fully symmetric with respect to each other. The augmented language imposes syntactic constraints on their usage. In [Ip96], it is shown how reflexive ring symmetries like those found in the *Dining Philosophers Problem* can similarly be handled.

Scalarsets have two advantages. First, violation of syntactic constraints can be detected during compile time. Second, since the types of symmetries are limited in number and well understood, algorithms for using symmetry information in a model checker get simplified.

The idea of introducing new data types for different kinds of symmetries has some drawbacks. There would always be a limit on the kind of symmetries that a designer is allowed to declare. A more serious flaw is that specification languages in the real world are not amenable to modification. For example, Verilog [Ver97] and VHDL [VHD93] are two IEEE Hardware Description Languages that cannot be modified easily.

1.2 New Ideas

We have developed a practical framework for automatically identifying symmetries in CTL* formulas and structural descriptions of digital circuits and using them in a model checker to speed up verification. The following are our key accomplishments:

1. We show that given a Kripke structure and a CTL* formula f , the set of sub-formulas of f can be partitioned into equivalence classes such that it suffices to compute the truth values for any one sub-formula in a class. This idea is distinct from building a quotient structure. The two can be used in conjunction.
2. We unify the theories for symmetries in Kripke structures and CTL* formulas, developed by Emerson and Sistla [ES96] and Clarke *et al* [CEFJ96]. We extend their fundamental theorems pertaining to quotient structures.
3. We formalize the notion of structural symmetries in net-list descriptions of digital circuits and CTL* formulas. We show how these symmetries relate to those in the corresponding Kripke structures. We propose algorithms for identifying these symmetries automatically. The algorithm requires no assistance from the designer.
4. A major problem to be solved before existing symbolic or explicit model checkers can successfully exploit quotient structures is the Canonical State Problem. We outline a new algorithm for solving the problem using symbolic techniques.
5. We propose a novel fast algorithm for solving the graph automorphism problem for directed labeled graphs. This problem arises during automatic extraction of structural

symmetries.

We note that our framework can be extended to formulas in μ -calculus, as done by Emerson and Sistla [ES96].

1.3 Organization of Report

In Chapter 2, we review background material. In Chapter 3, we show how the set of sub-formulas of a CTL* formula can be partitioned into equivalence classes. In Chapter 4, we develop theory for quotient structures. In Chapter 5, we formalize the notion of structural symmetries and show how such symmetries can automatically be identified. The identification procedure requires computation of all automorphisms of a directed labeled graph. In Chapter 6, we describe algorithms from computational group theory that solve the graph automorphism problem. In Chapter 7, we tabulate results, draw conclusions and list open research problems.

Chapter 2

Preliminaries

2.1 Kripke Structures

Let AP be a set of atomic propositions. A *Kripke structure* over AP is a triple $M = (S, R, K)$, where

- S is a finite set of *states*.
- $R \subseteq S \times S$ is a *transition relation* that is *total*, i.e. $(\forall s \in S)(\exists t \in S)((s, t) \in R)$.
- $K : S \rightarrow 2^{AP}$ is a *labeling function*.

The labeling function K associates with each state a set of atomic propositions that are true in that state. Note that K could be *many-to-one*. Let states in S be encoded such that there is a 1-1 mapping from S into 2^L for some L . Since each state corresponds to a unique element in 2^L , the labeling function K can be looked upon as a multi-output boolean function $K : 2^L \rightarrow 2^{AP}$.

In the context of hardware verification, L corresponds to the set of latches, AP corresponds to the set of outputs of the circuit and K represents boolean predicates on latches that generate the outputs.

A Kripke structure models the state transition graph of a Moore machine [HU79], where the outputs are functions of the current state variables. In general, it cannot model a Mealy machine, where the outputs could depend on the inputs as well. There are effective procedures for converting a Mealy machine into a Moore machine. In general, such a conversion changes the set of state variables.

2.2 Temporal Logic CTL*

CTL* was first proposed by Emerson and Halpern [EH86]. It has two kinds of boolean formulas: *state formulas* that evaluate to true in a specific state, and *path formulas* that evaluate to true along a specific path.

2.2.1 Syntax

Let AP be a finite set of atomic propositions.

A state formula is either

- p , where $p \in AP$,
- $\neg f$ or $f \vee g$, where f and g are state formulas, or
- $E(f)$, where f is a path formula.

A path formula is either

- a state formula, or
- $\neg f$, $f \vee g$, Xf or fUg , where f and g are path formulas.

CTL was proposed by Clarke and Emerson [CE81] much earlier than CTL*. It is that subset of state formulas of CTL* in which the path formulas are restricted to be Xf and fUg , where f and g are CTL formulas.

2.2.2 Semantics

Let $M = (S, R, K)$ be a Kripke structure. An infinite sequence of states $\psi = s_0, s_1, \dots$, is said to be a path in M if $(\forall i. i \geq 0)((s_i, s_{i+1}) \in R)$. Let ψ^i denote the suffix of ψ starting at s_i . Let $(M, s \models f)$ denote that the state formula f is true for state s in Kripke structure M . Similarly, let $(M, \psi \models g)$ denote that path formula g is true for path ψ in Kripke structure M . Let f_1 and f_2 be state formulas. Let g_1 and g_2 be path formulas. Then the relation \models is defined inductively as follows:

$$\begin{aligned}
 M, s \models p & \Leftrightarrow p \in K(s). \\
 M, s \models \neg f_1 & \Leftrightarrow M, s \not\models f_1. \\
 M, s \models f_1 \vee f_2 & \Leftrightarrow M, s \models f_1 \text{ or } M, s \models f_2. \\
 M, s \models E(g_1) & \Leftrightarrow \text{there exists a path } \psi \text{ starting at } s \text{ such that } M, \psi \models g_1.
 \end{aligned}$$

$$\begin{aligned}
M, \psi \models f_1 &\Leftrightarrow s \text{ is the first state of } \psi \text{ and } M, s \models f_1. \\
M, \psi \models \neg g_1 &\Leftrightarrow M, \psi \not\models g_1. \\
M, \psi \models g_1 \vee g_2 &\Leftrightarrow M, \psi \models g_1 \text{ or } M, \psi \models g_2. \\
M, \psi \models Xg_1 &\Leftrightarrow M, \psi^1 \models g_1. \\
M, \psi \models g_1 U g_2 &\Leftrightarrow (\exists n \geq 0)((M, \psi^n \models g_2) \wedge (\forall j. 0 \leq j < n)(M, \psi^j \models g_1)).
\end{aligned}$$

2.2.3 Logically Equivalent Formulas

We say that two CTL* formulas f and g are *logically equivalent* if their truth value is the same for every state in any Kripke structure they are interpreted on.

2.2.4 Structurally Equivalent Formulas

We say that two CTL* formulas f and g are *structurally equivalent* if they not only are logically equivalent but also have isomorphic parse trees. The latter condition can be rephrased as follows: when written in postfix notation, the sequences of operators in both formulas should be identical and there should exist a permutation defined on the set AP such that replacing every occurrence of $p \in AP$ in formula f yields the sequence of operands in formula g .

Intuitively, when f and g are structurally equivalent, g is the same formula as f written in a structurally different way. The difference arises solely due to the commutativity of some operators.

2.3 Model Checking Problem

The model checking problem is: *Given a set of atomic propositions AP , a Kripke structure $M = (S, R, K)$, a CTL* formula f defined on AP and a set of initial states $I \subseteq S$, does every state in I satisfy f ?*

Typically, M is not specified explicitly. It is derived from the system to be verified. Common systems are hardware circuits, distributed protocols and concurrent programs. The set S and the transition relation R are derived from the system specification. The atomic propositions in AP correspond to outputs in a hardware circuit or state variables in a protocol. What makes model checking challenging is the fact that the size of S can be

exponential in the size of the specification.

Efficient procedures have been developed for solving the model checking problem. Clarke, Emerson and Sistla [CES86] presented the first algorithm for CTL model checking based on explicit state space exploration. Their algorithm is linear in the size of the formula and the number of states.

In 1986, Bryant [Bry86] described an efficient implementation of Binary Decision Diagrams (BDDs), a data structure for representing boolean functions first introduced by Akers [Ake78]. Soon after Bryant announced the success of his BDD package, several groups started adapting explicit state space exploration techniques for BDDs. Some of the earliest symbolic techniques using BDDs were proposed by Coudert *et al* [CMB90], Burch *et al* [BCMD90] and Touati *et al* [TSL⁺90]. A symbolic model checker that can handle more than 10^{120} states on some pipelined circuits has been described by Burch *et al* [BCL⁺94].

Both symbolic and explicit CTL model checking algorithms first compute the set of states reachable from I . They also construct a parse-tree of f . Leaves of the parse tree correspond to atomic propositions in AP . Non-leaf nodes contain operators and correspond to different sub-formulas of f . The parse tree is processed in post-order. When a state sub-formula is encountered, its truth value for the set of reachable states is evaluated and remembered. After the root is processed, the algorithms check whether f is true for all the states in I or not.

CTL* model checking proceeds in a similar fashion. The parse tree is traversed in post order. Evaluation of truth values for a path formula can be done by using a model checker for LTL, another temporal logic. Model checking of LTL formulas can be done using language containment and tableau construction [VW86].

2.4 Permutation Groups

A classic reference for group theory is a textbook by Herstein [Her75]. Permutation groups are studied in a book by Wielandt [Wie64]. Practical algorithms for manipulating permutation groups are presented by Butler [But91].

2.4.1 Definitions

A permutation π is a bijective mapping $\pi : S \rightarrow S$ defined over a finite non-empty set S . Permutations are usually written in cycle form. Singleton cycles are ignored. For

example, $(1, 4, 2)(3, 7)$ denotes a permutation on the set $\{1, 2, 3, 4, 5, 6, 7\}$ where the images of elements 1, 2, 3, 4, 5, 6, 7 are 4, 1, 7, 2, 5, 6, 3.

We denote the action of π on an element $s \in S$ by πs . Let a binary operator \circ be defined on a pair of permutations as follows: $\pi_1 \circ \pi_2 = \pi_3$, where $(\forall s \in S) (\pi_3 s = \pi_2 \pi_1 s)$. It is easily verified that π_3 is a permutation.

A non-empty set of permutations G together with the operator \circ constitutes a group if it satisfies the following properties:

1. CLOSURE $(\forall \pi_1, \pi_2 \in G)(\pi_1 \circ \pi_2 \in G)$
2. ASSOCIATIVITY $(\forall \pi_1, \pi_2, \pi_3 \in G)(\pi_1 \circ (\pi_2 \circ \pi_3) = (\pi_1 \circ \pi_2) \circ \pi_3)$
3. IDENTITY $(\exists e \in G)(\forall \pi \in G)(e \circ \pi = \pi \circ e = \pi)$
4. INVERSE $(\forall \pi \in G)(\exists \pi^{-1} \in G)(\pi \circ \pi^{-1} = \pi^{-1} \circ \pi = e)$

Strictly speaking, the set G along with the operator \circ together define a group. However, in this report, we will use G as a convenient shorthand for denoting the group as we deal only with permutation groups.

Let H be a subset of permutations in the group G . The set H *generates* G if every element π of G can be written as a composition $\pi_{i_1} \circ \pi_{i_2} \circ \dots \circ \pi_{i_k}$ of elements of H for some k dependent on π . Elements of H are also called *generators* of G .

A group H is a subgroup of G if H is a group and every member of H belongs to G . We use $H \leq G$ to denote this relationship.

We use $G_1 \cap G_2$ to denote the intersection of the groups G_1 and G_2 , which itself is a group.

For a set $T \subseteq S$, we define $\pi T = \{s \mid s = \pi t \text{ where } t \in T\}$. This overloads the operator π but buys us notational convenience.

For a set $X \subseteq S$, such that $\pi X = X$, we use $\pi_{\langle X \rangle} : X \rightarrow X$ to denote the restriction of π to X .

2.4.2 Definition of $G \bowtie H$

We introduce a binary operator \bowtie to represent a function that we will encounter several times in this report.

Let G denote a permutation group over $S_1 \cup S_2$ such that $(\forall \pi \in G)((\pi S_1 = S_1) \wedge (\pi S_2 = S_2))$. Let H denote a permutation group over $S_2 \cup S_3$ similarly. Then $G \bowtie H$

is a permutation group over $S_1 \cup S_3$. A permutation π belongs to $G \bowtie H$ if and only if there exists a pair of permutations $g \in G$ and $h \in H$ such that $(\forall s \in S_1)(gs = \pi s)$, $(\forall s \in S_3)(hs = \pi s)$ and $(\forall s \in S_2)(gs = hs)$.

Effectively, $G \bowtie H$ joins permutations from G and H if and only if their action on S_2 is the same. The symbol \bowtie has been borrowed from relational database literature, where it denotes the *join* of two tables i.e. the operation of taking the cross product of two sets of tuples (tables) and choosing those that satisfy some predicate. The predicate is commonly the equality of some members (table columns) of the original tuples coming from the two tables.

2.5 Graph Isomorphism Problem

The graph isomorphism problem is: *Given a pair of directed graphs $A_1 = (V_1, E_1)$ and $A_2 = (V_2, E_2)$, is there a bijective mapping $\pi : V_1 \rightarrow V_2$ such that*

$$(\forall v_1, v_2 \in V_1)((v_1, v_2) \in E_1 \Leftrightarrow (\pi v_1, \pi v_2) \in E_2)?$$

The book by Hoffman [Hof82] and Section 2.6 of the survey by van Leeuwen [vL90] present a comprehensive summary of algorithmic results pertaining to the graph isomorphism problem. The problem has withstood all attempts at a solution to date. It is neither known to be NP-complete nor known to be polynomially solvable. There is some theoretical evidence that it is not NP-complete because if it were so, the Meyer-Stockmeyer polynomial hierarchy would collapse. The problem remains hard for directed acyclic graphs, bipartite graphs and regular graphs.

Polynomial algorithms are known for a few special cases. For rooted trees with n vertices, the problem can be solved in $O(n)$ time (Theorem 3.3 in the book by Hopcroft and Ullman [HU79]). For planar graphs with n vertices, Hopcroft and Wong [HW74] present an $O(n)$ time algorithm. A linear time algorithm for interval graphs is due to Lueker and Booth [LB79]. For random graphs, an $O(n \log n)$ algorithm is due to Deo *et al* [DDL77]. A polynomial time algorithm for graphs with bounded degree has been discovered by Luks [Luk82]. For general graphs, the best known algorithm is due to Babai and Luks [BL83]. It has a worst-case time complexity of $O(c^{n^{1/2+o(1)}})$.

2.6 Graph Automorphism Problem

Given a directed graph $A = (V, E)$, a permutation $\pi : V \rightarrow V$ is said to be an automorphism of A if it satisfies $(\forall u, v \in V)((u, v) \in E \Leftrightarrow (\pi u, \pi v) \in E)$. The set of all such permutations forms a group as it satisfies the four properties listed in Section 2.4.1. The graph automorphism problem is to compute this group from a description of A .

The problem of computing whether a graph has a non-trivial automorphism is as hard as graph isomorphism. See the survey by van Leeuwen [vL90] for further results. Practical algorithms for computing the automorphism group of a directed graph have received much attention in the last two decades. We discuss them in Chapter 6.

Chapter 3

Symmetric Sub-formulas

In this chapter, we develop theory for partitioning the set of sub-formulas of a CTL* formula into equivalence classes such that it suffices to evaluate the truth values for only one sub-formula in any class for model checking. We also outline a technique for identifying these classes automatically.

3.1 Definitions

Let $M = (S, R, K)$ be a Kripke structure with 2^L states.

3.1.1 Definition of $Aut_M L$

Let $\pi : L \rightarrow L$ be a permutation. It induces a permutation $\Pi : 2^L \rightarrow 2^L$ naturally. Let π be such that Π is an automorphism of the directed unlabeled graph (S, R) . The set of all such π forms a group, which we denote by $Aut_M L$.

3.1.2 Definition of $Aut_M L \cdot AP$

Consider a permutation $\pi : L \cup AP \rightarrow L \cup AP$ such that $(\pi L = L)$ and $(\pi_{\langle L \rangle} \in Aut_M L)$ and $(\forall x \in 2^L)(\forall y \in 2^{AP})((K(x) = y) \Leftrightarrow (K(\pi x) = \pi y))$. Recall that $\pi_{\langle L \rangle}$ denotes the restriction of π to L . The set of all such permutations π forms a group which we denote by $Aut_M L \cdot AP$.

3.1.3 Definition of $Aut_M L \cdot X$

Sometimes in later chapters, it will be convenient to consider a Kripke structure M having additional labels drawn from a set X . In such a case, the new labels can be looked upon as a mapping $K' : 2^L \rightarrow 2^X$.

Consider a permutation $\pi : L \cup X \rightarrow L \cup X$ such that $(\pi L = L)$ and $(\pi_{\langle L \rangle} \in Aut_M L)$ and $(\forall x \in 2^L)(\forall y \in 2^{AP})((K'(x) = y) \Leftrightarrow (K'(\pi x) = \pi y))$. Recall that $\pi_{\langle L \rangle}$ denotes the restriction of π to L . The set of all such permutations π forms a group which we denote by $Aut_M L \cdot X$.

3.2 Theory for Symmetric Sub-formulas

For $s \in S$ and $\pi \in Aut_M L \cdot AP$, let πs denote the state obtained by applying π to the encoding of s . For any path ψ in M , let $\pi \psi$ denote the path obtained by applying π to every state in ψ . For a CTL* formula f defined on AP , let πf denote the formula obtained by replacing every occurrence of $p \in AP$ by πp .

Theorem 3.1 *For a Kripke structure $M = (S, R, K)$ and a permutation $\pi \in Aut_M L \cdot AP$,*

$$(M, s \models f) \Leftrightarrow (M, \pi s \models \pi f)$$

$$(M, \psi \models g) \Leftrightarrow (M, \pi \psi \models \pi g)$$

for any state $s \in S$, any path ψ in M , any CTL state formula f and any CTL* path formula g .*

Proof:

The syntax of CTL*, as described in Section 2.2, allows us to write path formulas that are the same as state formulas. To differentiate between the two, we will assume the existence of an operator *Path* that explicitly converts state formulas into path formulas, as allowed by the syntax. Then the shortest formulas are of the kind $f = p$, where f is a state formula and $p \in AP$.

We prove by induction on the length of the formula. We will use the easily proven identities: $(\pi(\neg f) = \neg(\pi f))$, $(\pi(f \vee g) = \pi f \vee \pi g)$, $(\pi(Xf) = X(\pi f))$, $(\pi(Eg) = E(\pi g))$ and $(\pi(g_1 U g_2) = \pi g_1 U \pi g_2)$.

Base case:

Let $f = p$ where $p \in AP$. Let $(M, s \models f)$. Then $p \in K(s)$. From the definition of $Aut_M L \cdot AP$, we obtain $\pi p \in K(\pi s)$. Thus $(M, \pi s \models \pi f)$. We can prove the result in the other direction similarly to obtain $(M, s \models f) \Leftrightarrow (M, \pi s \models \pi f)$.

Induction step:

We show that if the theorem holds for all formulas of length k , then it holds for all formulas of length $k + 1$.

State formulas:

- $\neg f$: Formula f has length k . By assumption, $(M, s \models f) \Leftrightarrow (M, \pi s \models \pi f)$. It readily follows that $(M, s \models \neg f) \Leftrightarrow (M, \pi s \models \neg \pi f) \Leftrightarrow (M, \pi s \models \pi(\neg f))$.
- $f_1 \vee f_2$: We have $(M, s \models f_1 \vee f_2) \Leftrightarrow ((M, s \models f_1) \text{ or } (M, s \models f_2)) \Leftrightarrow ((M, \pi s \models \pi f_1) \text{ or } (M, \pi s \models \pi f_2)) \Leftrightarrow (M, \pi s \models (\pi f_1 \vee \pi f_2)) \Leftrightarrow (M, \pi s \models \pi(f_1 \vee f_2))$.
- $E(g)$: By definition, $(M, s \models E(g)) \Leftrightarrow$ there exists a path ψ starting at s such that $(M, \psi \models g)$. Since g is of length k , $(M, \psi \models g) \Leftrightarrow (M, \pi\psi \models \pi g)$. It follows that $(M, s \models E(g)) \Leftrightarrow (M, \pi s \models E(\pi g)) \Leftrightarrow (M, \pi s \models \pi E(g))$.

Path formulas:

- Formulas of the kind $\neg g$ and $g_1 \vee g_2$ can be handled in the same way as state formulas.
- Xg : By definition, $\pi\psi^1 = (\pi\psi)^1$. Since g is of length k , $(M, \psi^1 \models g) \Leftrightarrow (M, \pi\psi^1 \models \pi g) \Leftrightarrow (M, (\pi\psi)^1 \models \pi g) \Leftrightarrow (M, \pi\psi \models X(\pi g)) \Leftrightarrow (M, \pi\psi \models \pi(Xg))$.
- $g_1 U g_2$: Let $(M, \psi \models g_1 U g_2)$. By definition, $(\exists n \geq 0)((M, \psi^n \models g_2) \text{ and } (\forall j. 0 \leq j < n)(M, \psi^j \models g_1))$. Both g_1 and g_2 have length at most k . We have $(\forall i. i \geq 0)(\pi\psi^i = (\pi\psi)^i)$. Thus, $(M, \psi^n \models g_2) \Leftrightarrow (M, \pi\psi^n \models \pi g_2) \Leftrightarrow (M, (\pi\psi)^n \models \pi g_2)$. Similarly, $(\forall j. 0 \leq j < n)((M, \psi^j \models g_1) \Leftrightarrow (M, (\pi\psi)^j \models \pi g_1))$. Combining them, we obtain $(M, \psi \models g_1 U g_2) \Leftrightarrow (M, \pi\psi \models \pi g_1 U \pi g_2) \Leftrightarrow (M, \pi\psi \models \pi(g_1 U g_2))$.
- *Path f*: By definition, $(M, \psi \models \text{Path } f) \Leftrightarrow (M, s \models f)$ where s is the first state of ψ . Since f is of length k , $(M, s \models f) \Leftrightarrow (M, \pi s \models \pi f)$. It follows that $(M, \psi \models \text{Path } f) \Leftrightarrow (M, \pi\psi \models \text{Path } (\pi f)) \Leftrightarrow (M, \pi\psi \models \pi(\text{Path } f))$.

□

3.3 Equivalence Classes of Sub-formulas

For a Kripke structure M and CTL* formula f defined on AP , let SF denote the set of all sub-formulas of f , including any atomic propositions in AP that occur in f . Recall the definitions of logical and structural equivalence from Sections 2.2.3 and 2.2.4. For a subgroup $G \leq \text{Aut}_M L \cdot AP$, we define a relation $\approx^G \subseteq SF \times SF$ as $(\forall f_1, f_2 \in SF)((f_1 \approx^G f_2) \Leftrightarrow (\exists \pi \in G)(\pi f_1 \text{ and } f_2 \text{ are logically equivalent}))$. We also define a relation \approx_s^G the same way as \approx^G but replacing logical equivalence by structural.

Theorem 3.2 *For $G \leq \text{Aut}_M L \cdot AP$, the relations \approx^G and \approx_s^G are equivalence relations, with \approx^G inducing a partition coarser than that induced by \approx_s^G .*

Proof:

Reflexivity: The identity permutation belongs to every permutation group G and leaves any formula unchanged.

Symmetry: From the definition of permutation groups, $\pi \in G \Rightarrow \pi^{-1} \in G$. If π is a witness for $(f_1 \approx^G f_2)$, then π^{-1} is a witness for $(f_2 \approx^G f_1)$. The same holds for \approx_s^G also.

Transitivity: If $\pi_1 \in G$ is a witness for $(f_1 \approx^G f_2)$ and $\pi_2 \in G$ is a witness for $(f_2 \approx^G f_3)$, then $\pi_1 \circ \pi_2$ is a witness for $(f_1 \approx^G f_3)$, where $\pi_1 \circ \pi_2 \in G$, from the definition of permutation groups. A similar argument holds for \approx_s^G as well. \square

3.4 Model Checking with Equivalence Classes

How do Theorem 3.1 and Theorem 3.2 expedite model checking? Let us assume that we have identified the equivalence classes induced by \approx^G or \approx_s^G . Consider two sub-formulas g and h in the same class. Let $\pi \in \text{Aut}_M L \cdot AP$ be a witness that transforms h into g . If the truth value of h has been evaluated for all states in S , the truth value for g is immediately available. In a symbolic technique, the BDD for g can be computed from that for h by variable substitution corresponding to π . Alternatively, we can obtain the BDD for g in a different variable ordering by simply renaming variables in h .

The truth values for only one formula in any equivalence class defined by \approx^G or \approx_s^G need be evaluated; all others follow immediately. This could contribute to significant savings if the formulas in the orbit are big or they are path formulas with a temporal

operator at the top. For the latter case, some fixed point computations are obviated in a symbolic technique.

3.5 Computing Equivalence Classes of Sub-formulas

Problem: Given $G \leq \text{Aut}_M L \cdot AP$ and a CTL* formula f , how do we find two sub-formulas g and h such that $g \approx^G h$? This is a computationally hard problem even if f is a simple boolean formula without path operators or temporal quantifiers [AT96].

Does the problem become easier if we replace \approx^G by \approx_s^G ? If we allow operators with arbitrary arity (which is the interesting case, in practice), we can show that the problem is as hard as graph isomorphism. The reduction is simple. Given two undirected graphs, let AP be a set of atomic propositions with cardinality equal to the number of vertices in either graph. Label the vertices of the first graph with AP such that each vertex has a unique label. Now, label each edge (x, y) with the sub-formula $p_x \wedge p_y$, where p_x and p_y are the labels of x and y respectively. Finally, construct the disjunction of all these sub-formulas. Construct a similar formula for the other graph. The two graphs are isomorphic if and only if the two formulas are structurally equivalent. We can also show that the problem remains hard even if we restrict the arity of operators to $O(|AP|)$.

We now outline an algorithm that is limited in the types of symmetric sub-formulas it can identify but can be very effective in practice.

3.5.1 Definition of $\text{Aut}_f AP \cdot SF$

For a CTL* formula f , let SF denote the set of sub-formulas of f , including all atomic propositions that occur in f . Consider the group consisting of permutations $\pi : AP \rightarrow AP$ such that f and πf are structurally equivalent. Every permutation in this group implicitly defines a permutation on the set $AP \cup SF$. For example, let $AP = \{p_1, p_2, p_3\}$ and $f = \vee(f_1, f_2, f_3)$ where $f_1 = p_1Up_2$, $f_2 = p_2Up_3$ and $f_3 = p_3Up_1$. Then $SF = \{f, f_1, f_2, f_3, p_1, p_2, p_3\}$. The permutation $\pi = (p_1, p_2, p_3)$ belongs to $\text{Aut}_f AP$ and implicitly defines the permutation $(p_1, p_2, p_3)(f_1, f_2, f_3)(f)$. We denote this group on $AP \cup SF$ by $\text{Aut}_f AP \cdot SF$.

3.5.2 Definition of \approx_s^{GMH}

Let $G \leq Aut_M L \cdot AP$. Let $H \leq Aut_f AP \cdot SF$. Recall the definition of \bowtie from Section 2.4.2. We see that the group $G \bowtie H$ is well defined. We define a relation $\approx_s^{GMH} \subseteq SF \times SF$ as $(\forall f_1, f_2 \in SF)((f_1 \approx_s^{GMH} f_2) \Leftrightarrow (\exists \pi \in G \bowtie H)(\pi f_1 \text{ and } f_2 \text{ are structurally equivalent}))$. This is an equivalence relation. In general, the partition induced by \approx_s^{GMH} is finer than that induced by \approx_s^G for $G = Aut_M L \cdot AP$.

3.5.3 Putting It All Together

The overall idea is to compute a group $G \leq Aut_M L \cdot AP$, a group $H \leq Aut_f AP \cdot SF$ and then $G \bowtie H$. In Chapter 5, we will show how all the three steps can be carried out automatically by constructing a suitable graph and solving the graph automorphism problem for it. The representation for $G \bowtie H$ would allow us to easily identify the partitions induced by \approx_s^{GMH} and produce witnesses that transform one sub-formula into another in the same partition.

3.5.4 Critique

We concede that the technique outlined above would identify equivalent sub-formulas only if the symmetry in the specification is reflected in the formula as well. It appears that a coarser partition can be obtained by devising better heuristics, as the size of formulas is small in practice. This could be the theme of a short research project.

3.6 Generating New Formulas

Having computed $G \leq Aut_M L \cdot AP$ and proved the correctness of a CTL* formula f , one can use Theorem 3.1 to generate new formulas whose truth value is already known. These can be obtained by producing a non-trivial $\pi \in G$ and constructing πf . A model checker can present new formulas to a designer in a controlled fashion using an interactive user interface.

3.7 Summary

Theorem 3.1 and Theorem 3.2 show how sub-formulas of a CTL* formula can be partitioned into equivalence classes such that the truth value for only one formula in any class need be evaluated. In Section 3.5, we outlined a procedure for automatically identifying these classes. Briefly, we need to compute $G \leq Aut_M L \cdot AP$, $H \leq Aut_f AP \cdot SF$ and $G \bowtie H$.

The idea of symmetric sub-formulas does not attack the state space explosion problem. It expedites model checking on the original structure. In the next section, we develop the notion of quotient structures, which are aimed directly at reducing the state space. Once they are described, it will become clear that identification of symmetric sub-formulas contributes to savings on top of quotient structures.

Chapter 4

Quotient Structures

In this chapter, we develop a theory of symmetries for Kripke structures, unifying those developed by Clarke *et al* [CEFJ96] and Emerson and Sistla [ES96]. We first show how a smaller sized *quotient* structure can be constructed from a Kripke structure and a CTL* formula so that it suffices to check the formula on the quotient. We then show how a model checker need be modified to use quotient structures. A major problem to be solved before model checkers can successfully use quotients is the Canonical State Problem. We propose a novel symbolic algorithm for this problem. Finally, we list areas requiring further research.

4.1 Definitions

Let $M = (S, R, K)$ be a Kripke structure with 2^L states. Recall the definition of $Aut_M L \cdot X$ from Section 3.1.3.

4.1.1 Definition of $\llbracket s \rrbracket_G$ and ξ_G

Let $G \leq Aut_M L \cdot X$ for some set of labels X . Let two states s and t in S be related if there exists $\pi \in G$ such that $\pi s = t$. This defines an equivalence relation, partitioning S into equivalent sets called *orbits*. We denote the orbit of a state $s \in S$ by $\llbracket s \rrbracket_G$. We pick a state from each orbit to obtain a set of representatives. We then define a function $\xi_G : S \rightarrow S$ such that each state is mapped to the representative of the orbit it belongs to. Note that ξ_G is not unique. The results in this chapter hold for any ξ_G .

4.1.2 Definition of Quotient Structure M_G

For a Kripke structure $M = (S, R, K)$ and $G \leq \text{Aut}_M L \cdot X$ for some set of labels X , the *quotient structure* is defined as $M_G = (S_G, R_G, K_G)$, where

- $S_G = \{[s]_G \mid s \in S\}$
- $R_G = \{([s]_G, [t]_G) \mid (s, t) \in R\}$
- $K_G([s]_G) = K(\xi_G(s))$

4.2 Theory for Quotient Structures

An important relationship between a Kripke structure and its quotient is captured by what is called the Path Correspondence Lemma.

4.2.1 The Path Correspondence Lemma

Lemma 4.1 [CEFJ96, ES96]

For a Kripke structure M and any group $G \leq \text{Aut}_M L \cdot X$, there is a correspondence between paths of M and its quotient structure M_G , characterized by

- *If s_0, s_1, \dots , is a path in M , then $[s_0]_G, [s_1]_G, \dots$, is a path in M_G .*
- *If $[s_0]_G, [s_1]_G, \dots$, is a path in M_G , then for every state $t_0 \in [s_0]_G$, there exists a path t_0, t_1, \dots , such that $t_i \in [s_i]_G$ for $i \geq 0$.*

□

4.2.2 Label Preserving Quotients

The fundamental result in [CEFJ96] is captured by the following theorem:

Theorem 4.1 [CEFJ96]

For a Kripke structure $M = (S, R, K)$ and a group $G \leq \text{Aut}_M L \cdot AP$, if

$$(\forall \pi \in G)(\forall p \in AP)(\pi p = p)$$

then

$$(\forall s \in S)((M, s \models f) \Leftrightarrow (M_G, [s]_G \models f))$$

for any CTL formula f .*

□

Application of Theorem 4.1 requires that the truth value of every atomic proposition be invariant under every permutation in G . In group-theoretic terminology, we require that each $p \in AP$ be centralized.

In the extreme case, we could have $AP = L$, giving each state a unique label. As a consequence, G is trivial and the quotient structure is not smaller than the original. Therefore, to use Theorem [CEFJ96], we should be able to generate *good* outputs as atomic propositions. This is possible in practice. See [CEFJ96] for examples.

A motivating example where Theorem 4.1 does not allow reduction but symmetries clearly exist, is a system consisting of n black-boxes interconnected in a ring topology. The *Dining Philosophers Problem* is a good example of such a system. Let each black box have a local output. Let f be a fully symmetric boolean formula on these n outputs. The quotient structure implied by Theorem 4.1 would be trivial.

If we replace the old set of labels with a single label corresponding to the truth value of the overall formula, we can leverage Theorem 4.1. This gives us a generalization that we now develop.

4.2.3 Definition of $Aut_f AP \cdot MPS$

A maximal propositional sub-formula is a sub-formula containing only the boolean operators \neg and \vee such that it is not an operand of another formula with topmost operator \neg or \vee .

For a CTL* formula f , let MPS be the set of its maximal propositional sub-formulas. Let f_{MPS} be the corresponding multi-output boolean function $2^{AP} \rightarrow 2^{MPS}$.

We define $Aut_f AP \cdot MPS = \{\pi : AP \cup MPS \rightarrow AP \cup MPS \mid \pi \text{ is a permutation, } \pi AP = AP, \pi MPS = MPS, (\forall y \in MPS)(\pi y = y) \text{ and } (\forall x \in 2^{AP})(f_{MPS}(x) = f_{MPS}(\pi x))\}$. This set forms a group.

4.2.4 Larger Label Preserving Quotients

Recall the definition of the operator \bowtie from Section 2.4.2. We see that for $G \leq Aut_M L \cdot AP$ and $H \leq Aut_f AP \cdot MPS$, the group $G \bowtie H$ is well defined.

If we replace the labels of M with a set of labels corresponding to truth values of sub-formulas in MPS , we can define $Aut_M L \cdot X$ (see Section 3.1.2 for its definition) with $X = MPS$. From the definitions in Section 4.1.1, we see that $[s]_{G \bowtie H}$ and $\xi_{G \bowtie H}$ are well

defined since it is true that $G \bowtie H \leq \text{Aut}_M L \cdot X$, where $X = MPS$.

Theorem 4.2 For $G \leq \text{Aut}_M L \cdot AP$ and $H \leq \text{Aut}_f AP \cdot MPS$,

$$(\forall s \in S)((M, s \models f) \Leftrightarrow (M_{G \bowtie H}, [s]_{G \bowtie H} \models f))$$

Proof: The crux of the proof lies in showing that $G \bowtie H \leq \text{Aut}_M L \cdot X$, where $X = MPS$. Then replacing labels of M by labels corresponding to evaluations of sub-formulas in MPS allows a straightforward application of Theorem 4.1 to get the desired result. □

It is clear that Theorem 4.1 is a special case of Theorem 4.2, which captures the symmetries of the black-box system we outlined at the end of Section 4.2.2.

We note that the definition of H is along the same lines as that of $\text{Auto}'f$ in [ES96]. However, their theory is built for a system composed of isomorphic processes such that all states are uniquely labeled i.e. $AP = L$.

In Chapter 5, we will show how G , H and $G \bowtie H$ can be computed automatically from a structural specification and CTL* formula.

4.2.5 An Extension of Emerson-Sistla Theorem

Emerson and Sistla [ES96] build their theory of Kripke structures for systems of communicating isomorphic processes, with the set of atomic propositions being the set of shared variables. In our terminology, it amounts to assuming $AP = L$ and restricting the number of initial states to one.

We have extended their fundamental theorem to the case where this assumption is not true. Briefly, it amounts to introducing a new set of labels corresponding to all maximal propositional sub-formulas along with all sub-formulas that have E , X or U as the topmost operator. The proof proceeds along the same lines as Theorem 4.2. Due to lack of time, we omit the extended theorem and its proof from this report. We however note that computation of the corresponding H and $G \bowtie H$ can indeed be automated.

4.2.6 Summary

Theorem 4.2 stipulates a set of sufficient conditions that a Kripke structure together with a CTL* formula should satisfy in order to facilitate the construction of a smaller sized quotient structure such that it suffices to check the formula on the quotient.

To construct the quotient, we need to identify the groups $G \leq Aut_M L \cdot AP$ and $H \leq Aut_f AP \cdot MPS$ and construct $G \bowtie H$. In Chapter 5, we will show how the three steps can be carried out automatically.

Once we have constructed $G \bowtie H$, how do we use it to expedite model checking? We explore this in the next section.

4.3 Model Checking with Quotient Structures

From $G \leq Aut_M L \cdot X$, let us assume that we magically obtain the function $\xi_G : S \rightarrow S$, as defined in Section 4.1.1. The function maps each state in S to a representative state in its orbit induced by G . How do we use ξ_G in a model checker?

4.3.1 Explicit State Enumeration Using ξ_G

In Figure 4.1 on the following page, we outline how an explicit search can be modified using ξ_G . If the set *Unexplored* is maintained as a stack, the search becomes depth first. We have borrowed the algorithm from [CEFJ96].

4.3.2 Symbolic Model Checking Using ξ_G

In a symbolic technique, BDDs are used to store the transition relation $R : 2^L \times 2^L \rightarrow 2^1$ and the set of initial states $I : 2^L \rightarrow 2^1$. Assuming the existence of a BDD for $\xi_G : 2^L \rightarrow 2^L$, the search step can be modified in any of several different ways.

First, we could compute a new transition relation $R_{\xi_G} : 2^L \times 2^L \rightarrow 2^1$ as:

$$R_{\xi_G}(x, y) \equiv (\exists x', y' \in 2^L)(R(x', y') \wedge (\xi_G(x') = x) \wedge (\xi_G(y') = y))$$

The set of initial states can be modified to $I_{\xi} : 2^L \rightarrow 2^1$ as:

$$I_{\xi_G}(x) \equiv (\exists x' \in 2^L)(I(x') \wedge (\xi_G(x') = x))$$

BDDs for R_{ξ_G} and I_{ξ_G} can be computed from R and I simply by substituting variables by functions. This operation is one of the BDD primitives. From the theory developed in Section 4.2, it follows that model checking can be done using I_{ξ_G} , R_{ξ_G} and the original CTL* formula. The space complexity of BDDs for R_{ξ} for interesting groups has not yet been studied by researchers.

```

Reached =  $\phi$ ;
Unexplored =  $\phi$ ;
For each initial state s Do
    Append  $\xi_G(s)$  to Reached;
    Append  $\xi_G(s)$  to Unexplored;
Endforloop
While (Unexplored  $\neq \phi$ ) Do
    Remove a state s from Unexplored;
    For each successor state q of s Do
        If ( $\xi_G(q) \notin \text{Reached} \cup \text{Unexplored}$ )
            Append  $\xi_G(q)$  to Reached;
            Append  $\xi_G(q)$  to Unexplored;
        EndIf
    Endforloop
EndWhile

```

Figure 4.1: Modified explicit state enumeration algorithm using ξ_G .

Alternatively, given a BDD for a set of representative states which are *Unexplored*, we could use ξ_G to obtain a BDD for the next set of representative states as:

$$\text{Next}(z) \equiv (\exists x', y' \in 2^L)(\text{Unexplored}(x') \wedge R(x', y') \wedge (\xi_G(y') = z))$$

This equation could be used at each step of the breadth first search. It does not require the construction of R_{ξ_G} .

In [HAM97], it is argued that explicit methods should be preferred to symbolic ones for symmetric systems. They show that the BDD for the set of reachable states for some real systems encountered in practice is exponential in size. However, implicit techniques might benefit from R_{ξ_G} by storing only the representatives of states reached. There are no known results pertaining to the space complexity of R_{ξ_G} .

4.4 The Canonical State Problem

We now explore how ξ_G can be computed. For $s \in 2^L$, $\xi_G(s)$ is said to be the *canonical state* of s . The *Canonical State Problem* is to compute ξ_G from a description of the system to be verified. Once we have computed $G \leq \text{Aut}_M L \cdot X$ in the form of a set of generators, the crucial question is: *How difficult is it to obtain ξ_G from a set of generators for a group acting on L ?*

In [CEFJ96], it is shown that the following problem is as hard as graph isomorphism: *Given two states in 2^L and a set of generators for a group G acting on the set L , do the two states have the same canonical state?* Therefore, the chances of devising a general purpose fast algorithm using standard representations for generators are remote. However, in an explicit model checker, we need to pose the query $\xi_G(s)$ repeatedly for a set of states which is of size $O(2^L)$. Can we devise an algorithm that pre-processes the set of generators such that the cost of pre-processing gets amortized over a multitude of subsequent queries? This is an open problem.

4.4.1 Equivalence Relation Approach

An idea suggested by [CEFJ96] is to construct a BDD for the equivalence relation $\Theta_G \subseteq S \times S$ defined as $(\forall s, t \in S)(\Theta_G(s, t) \Leftrightarrow ([s]_G = [t]_G))$. The motivation is to use the method of Lin [LN91] to construct ξ_G from Θ_G . However, it may be noted that once we have a BDD for Θ_G , we can compute $\xi_G(s_0)$ for a state s_0 efficiently without constructing ξ_G as follows: simply plug in $s = s_0$ and obtain a BDD for $\Theta_G(s_0, t)$; then choose any element from $\Theta_G(s_0, t)$ consistently. One choice could be the lexicographically largest/smallest value for t . Thus evaluating $\xi_G(s_0)$ requires just one path traversal through the BDD for Θ_G . Clarke *et al* [CEFJ96] have identified some interesting groups which are encountered in practice and for which the space complexity of any BDD for Θ_G has an exponential lower bound.

4.4.2 Direct Computation of ξ_G

We can attempt to construct ξ_G directly without constructing Θ_G . Note that Θ_G is a relation whereas ξ_G is a function. The complexity of BDDs for ξ_G is not known. Intuitively, they should require less space because ξ_G is a function. In Figure 4.2, we

INPUT: $\langle \pi_1, \pi_2, \dots, \pi_p \rangle$, a set of 1-1 and onto functions $2^L \rightarrow 2^L$.
OUTPUT: Canonical state function $\xi : 2^L \rightarrow 2^L$.

$\xi(x) \triangleq x$
for $i = 1$ to p **do**
 $\tau(x) \triangleq \pi_i(x)$
 $\eta(x) \triangleq \xi(x)$
do
 $\tilde{\tau}(x) \triangleq \tau(x)$
 $\tilde{\eta}(x) \triangleq \eta(x)$
 $\eta(x) \triangleq \begin{cases} \tau(\tilde{\eta}(x)), & \text{if } \tilde{\eta}(x) \leq \tau(\tilde{\eta}(x)) \\ \tilde{\eta}(x) & \text{otherwise.} \end{cases}$
 $\tau(x) \triangleq \tau(\tau(x))$
while $(\eta \neq \tilde{\eta})$
 $\tilde{\xi}(x) \triangleq \xi(x)$
 $\xi(x) \triangleq \begin{cases} \max \{ \eta(y) \mid \tilde{\xi}(y) = x \}, & \text{if } \tilde{\xi}(x) = x \\ \tilde{\xi}(x) & \text{otherwise.} \end{cases}$
do
 $\tilde{\xi}(x) \triangleq \xi(x)$
 $\xi(x) \triangleq \tilde{\xi}(\tilde{\xi}(x))$
while $(\xi \neq \tilde{\xi})$

Figure 4.2: Algorithm for computing ξ from a set of generators for a group acting on L .

outline a technique that constructs ξ_G from a set of generators without ever constructing an intermediate BDD that resembles Θ_G .

In Figure 4.2 on the preceding page, \triangleq should be looked upon as a BDD operation. The function $\max()$ takes a set $T \in 2^{2^L}$ as its argument and returns the *largest* element in T according to some total order on elements of 2^L .

The algorithm processes elements from the set of generators one by one. The first **do-while** loop computes η , the canonical state function for the cyclic group generated by π_i . This function is used to compute the new ξ , representing the canonical state function for the group generated by the set of permutations $\langle \pi_1, \pi_2, \dots, \pi_{i-1} \rangle$. It is easy to formally prove the correctness of the algorithm and to show that each **do-while** loop takes $O(|L|)$ iterations.

The exact complexity for this algorithm is yet to be determined. Does this one or a variant of it work well in practice?

4.4.3 Other Ideas

In [HAM97], a heuristic procedure is used to compute $\xi(s)$ for individual states, one at a time, for an explicit model checker. Another idea proposed by [CEFJ96] is the use of multiple representatives for each orbit.

We can solve the problem for special classes of groups for which ξ is readily computed. This is the approach used by several authors [CEFJ96, ES96, ID96, HAM97]. For example, if the system is composed of a hierarchy of fully symmetric subsystems, then ξ can be computed for a given state by simply computing the canonical state for each subsystem recursively and then sorting them.

4.5 Research Problems

Figure 4.3 on the following page is a flow diagram of a fully automatic setup that extracts symmetries and exploits quotient structures.

The three black boxes correspond to the following three sub-problems:

1. How do we compute $G \bowtie H$ from a system specification and logic formula?
2. How do we obtain ξ from a set of generators of a group $G \bowtie H$?

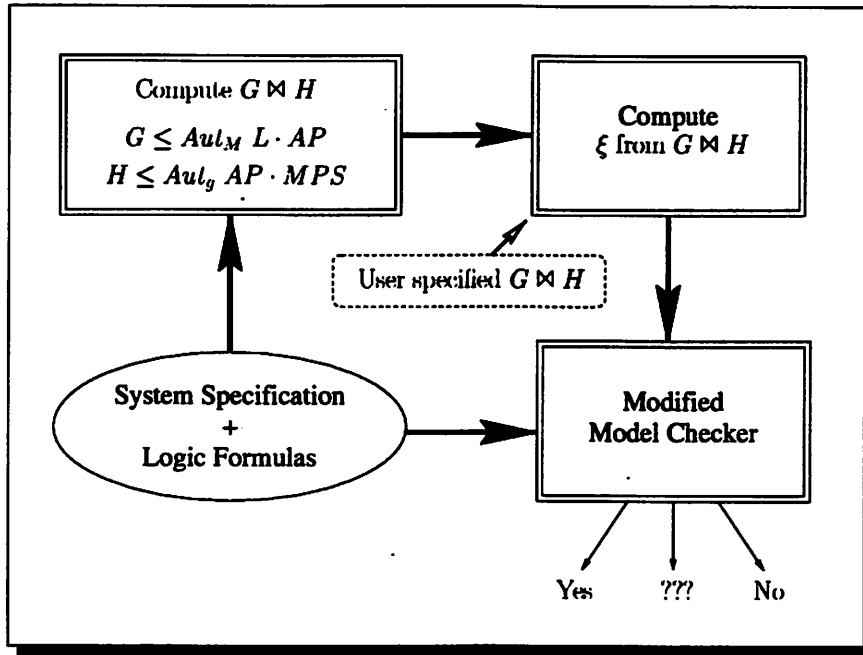


Figure 4.3: Flow diagram for model checking using quotient structures.

3. How do we modify a model checker so that it takes advantage of ξ ?

In Chapter 5, we outline how the first problem can be handled effectively for structural specifications. It would be interesting to extend these ideas to higher level specifications.

We need theoretical research for the second problem. It is very well defined. We note that it seems unlikely that ξ can be computed directly without computing $G \bowtie H$.

The third problem is of an experimental nature. We believe that the techniques outlined in Section 4.3 would serve as a good starting point.

Chapter 5

Structural Symmetries

In this chapter, we develop theory for structural symmetries in hardware circuits and CTL* formulas. We show the relationship between structural symmetries and symmetries in the corresponding Kripke structure. We also show how such symmetries can automatically be extracted from designs specified in a structural hardware description language like BLIF-MV [BCH⁺91].

5.1 Motivation

In Section 3.5, we saw how knowledge of groups $G \leq \text{Aut}_M L \cdot AP$ and $H \leq \text{Aut}_f AP \cdot SF$ would help us partition sub-formulas of a CTL* formula f into equivalence classes. In Section 4.2.4, we saw how knowledge of the same group G but a different $H \leq \text{Aut}_f AP \cdot MPS$ would allow us to construct quotient structures. In both the cases, we need to compute $G \bowtie H$.

In [CEFJ96, ES96, HAM97], a group similar to $G \bowtie H$ is specified by the designer manually. In [HAM97], it is mentioned that a permutation specified by the designer can be verified simply by checking that the transition relation BDD obtained by the corresponding variable substitution is the same as the original one. In [ID96], the symmetries are extracted from high-level descriptions having a new data type with restricted usage. *A priori* knowledge of symmetries obviates the need for computing $G \bowtie H$.

In this section, we describe how G , H and $G \bowtie H$ can be computed automatically, with no assistance from the designer. We start with a brief description of BLIF-MV [BCH⁺91, Ber93]. We then formally state what we mean by *structural symmetries* of a

BLIF-MV circuit and how they relate to the symmetries of the corresponding Kripke structure. We then show how a graph can be constructed from a BLIF-MV description so that there is a 1-1 correspondence between structural symmetries of the circuit and automorphisms of the graph. We do the same for CTL* formulas. Finally, we show how the two can easily be combined.

5.2 BLIF-MV

BLIF-MV [BCH⁺91, Ber93] is a structural hardware description language. It allows specification of a circuit in terms of its primary inputs, primary outputs, latches, tables and interconnection signals.

A latch has one output and two input ports. One of the input port specifies the initial values for the latch.

Tables represent combinational functions. They could have multiple input and output ports. The function implemented by a table is typically given in tabular form. See Figure 5.1 for an example. There is one column corresponding to every input and output port. A table description consists of entries. An entry is simply a subset of the values taken by the domain of its column. In each row, the input entries define a cube in the input domain space and the output entries define the output for that cube. Non-determinism can be achieved by specifying more than one output for the same input. Default values for each output can also be specified. The table in Figure 5.1 has three input ports, one output port and twelve entries.

```

.table a b c -> carry
.default 0
1 1 - 1
1 - 1 1
- 1 1 1

```

Figure 5.1: A BLIF-MV table for the carryover bit in a 3-bit adder.

Interconnections signals among primary inputs, primary outputs and ports of tables and latches respect the constraint that at most one signal feeds any input port. We assume that the circuit has no combinational cycles.

5.2.1 Characterizing a BLIF-MV Circuit

We model a BLIF-MV circuit as a five tuple $\mathcal{C} = \langle \mathcal{I}, \mathcal{O}, \mathcal{L}, \mathcal{T}, \mathcal{S} \rangle$, consisting of a set of primary input ports \mathcal{I} , a set of primary output ports \mathcal{O} , a set of latches \mathcal{L} , a set of tables \mathcal{T} and a set of interconnection signals \mathcal{S} . Intuitively, \mathcal{C} is a big black-box with I/O ports (primary inputs and outputs) consisting of smaller black boxes (tables and latches) whose I/O ports are interconnected with signals.

A table $T \in \mathcal{T}$ has input ports i_1^T, i_2^T, \dots and output ports o_1^T, o_2^T, \dots . With each output o_i^T , we associate a function f_i^T that takes the ordered tuple $\langle i_1^T, i_2^T, \dots \rangle$ as its argument. A latch $L \in \mathcal{L}$ has two input ports i_1^L, i_2^L and one output port o_1^L . The second input port specifies the initial value for the latch.

Source and Sink Ports

Let $P_{sink} = P_{in}^T \cup P_{in}^L \cup \mathcal{O}$, where $P_{in}^T = \bigcup_{T \in \mathcal{T}} \{i_1^T, i_2^T, \dots\}$ and $P_{in}^L = \bigcup_{L \in \mathcal{L}} \{i_1^L\}$. Let $P_{source} = P_{out}^T \cup P_{out}^L \cup \mathcal{I}$, where $P_{out}^T = \bigcup_{T \in \mathcal{T}} \{o_1^T, o_2^T, \dots\}$ and $P_{out}^L = \bigcup_{L \in \mathcal{L}} \{o_1^L\}$. Thus P_{sink} is the set of primary outputs and input ports of tables and latches, except those for initial values for latches. And P_{source} is the set of all primary inputs and all output ports of tables and latches.

We were motivated to use the sub-scripts *sink* and *source* because interconnection signals emanate from some port in P_{source} and feed a set of ports in P_{sink} .

Domains and Functions

Each port is associated with a domain. For $p \in P_{sink} \cup P_{source}$, let the function $dom(p)$ compute its domain. For $o_j^T \in P_{out}^T$, let the function $f_j^T(i_1^T, i_2^T, \dots)$ be the boolean function specified in its table that corresponds to the output produced at o_j^T . This function that takes an ordered list of input ports as its arguments. It could be non-deterministic.

Interconnection Signals

The interconnection signals \mathcal{S} simply define a relation $S_{ext} \subseteq P_{source} \times P_{sink}$. We also define a relation $S_{int} = \bigcup_{T \in \mathcal{T}} (\{i_1^T, i_2^T, \dots\} \times \{o_1^T, o_2^T, \dots\}) \bigcup_{L \in \mathcal{L}} (i_1^L, o_1^L)$. Thus S_{int} captures the internal dependencies of input and output ports within a latch or a table. And S_{ext} captures the *external* dependencies between primary inputs, primary outputs and I/O ports of tables and latches.

5.2.2 Definition of Structural Symmetries

A *structural symmetry* of \mathcal{C} is an automorphism $\pi : P_{sink} \cup P_{source} \rightarrow P_{sink} \cup P_{source}$ of the directed unlabeled graph $(P_{sink} \cup P_{source}, S_{int} \cup S_{ext})$ that satisfies the following constraints:

1. $\pi(P_{in}^T) = P_{in}^T$, $\pi(P_{out}^T) = P_{out}^T$, $\pi(P_{in}^L) = P_{in}^L$, $\pi(P_{out}^L) = P_{out}^L$, $\pi(\mathcal{I}) = \mathcal{I}$ and $\pi(\mathcal{O}) = \mathcal{O}$.
2. $(\forall p \in P_{sink} \cup P_{source})(dom(p) = dom(\pi p))$.
3. $(\forall o_j^T \in P_{out}^T)(f_j^T(i_1^T, i_2^T, \dots) = f_j^{T'}(i_1^{T'}, i_2^{T'}, \dots))$ where $(o_j^{T'} = \pi o_j^T)$, $(i_1^{T'} = \pi i_1^T)$, $(i_2^{T'} = \pi i_2^T)$, \dots

Condition 3 requires some explanation. From the first two conditions it can be deduced that vertices corresponding to a table will be mapped onto vertices corresponding to another table with the same number of inputs and outputs in such a way that their domains match. Condition 3 stipulates that even the functionality of the two tables should match. Note that no total ordering has been assumed on the table inputs or outputs.

It may be verified that the set of structural symmetries forms a group as it satisfies all the four conditions specified in Section 2.4.1.

5.2.3 Relationship with $Aut_M L \cdot AP$

How are structural symmetries in \mathcal{C} related to symmetries in some Kripke structure M ? For a circuit \mathcal{C} , there exists a Kripke structure $M = (S, R, K)$ with 2^L states and the set of atomic propositions AP . The set L corresponds to the latches. The set AP corresponds to outputs in \mathcal{O} . The function K represents the boolean predicate on latches that generate outputs. We assume that the outputs in \mathcal{C} do not depend on the inputs i.e. \mathcal{C} defined a

Moore machine. However, we note that the basic ideas of structural symmetries developed in this report are applicable to all circuits, in general.

Definition of $Aut_C L \cdot AP$

For a structural symmetry π , let $\pi_C : L \cup AP \rightarrow L \cup AP$ denote the permutation naturally induced by π . Let $Aut_C L \cdot AP$ denote the set of all such permutations. It is easily verified that $Aut_C L \cdot AP$ forms a group.

Theorem 5.1 $Aut_C L \cdot AP \leq Aut_M L \cdot AP$. □

5.2.4 Other Structural Languages

We have modeled a structural description as a black-box with I/O ports composed of smaller black-boxes (latches or tables) with I/O ports connected via signals. Different languages will allow the table functions to be expressed in different ways. This is where Condition 3 in Section 5.2.2 comes into play.

Although we used BLIF-MV terminology to formalize the notion of structural symmetries, we believe that our definition is general enough to be applicable to gate level descriptions like those expressible in EDIF [EDI97], Verilog [Ver97] and VHDL [VHD93]. We need be a little careful with a general language that involves `if`, `for` and `while` constructs.

5.3 Graphs for BLIF-MV Circuits

One problem with the definition of structural symmetries in Section 5.2.2 is that Condition 3 cannot be expressed in purely graph theoretic terms. In this section, we describe how BLIF-MV circuits can be processed to generate a labeled directed graph such that there is 1-1 correspondence between structural symmetries in the BLIF-MV circuit and automorphisms of the graph. This allows us to leverage results from computational group theory developed for identifying graph automorphisms.

5.3.1 Tables

For every table $T \in \mathcal{T}$, we have four types of vertices, namely C , S , A and O , which are acronyms for *Copy*, *Subset*, *And* and *Output* respectively. Their intuitive meaning will

soon become clear. The type of a vertex should not be confused with its label. The two are distinct. Types have been introduced solely for ease of presentation.

We have one vertex of type C for every input port, labeled with the corresponding input domain. We have one vertex of type S for each table entry in an input column. It is labeled with the subset of domain values in that entry. We have one vertex of type A for each entry in the output column. It is labeled with the subset of domain values in that entry. Finally, we have one vertex of type O for each output port, which is labeled with both the corresponding domain and the default value specified for that port.

We draw an edge from a vertex of type C to every vertex of type S that corresponds to entries in that input column. We draw an edge into a vertex of type A from every vertex that corresponds to the input entries in that row. Finally, we draw edges into a vertex of type O from all vertices corresponding to entries in that output column. A special case arises if the table has no entries at all. Then there are no nodes of type C or type A and none of the edges described above are drawn. Another special case is when there are no table inputs. Then there are no vertices of type C or type S . At the same time, if there are no rows, then there are no vertices of type A as well.

See Figure 5.2 on the next page for the graph corresponding to the BLIF-MV table in Figure 5.1 on page 32. The intuitive meaning associated with the vertex types should now be clear. The vertices capture the functional elements of a table and the edges capture how signals arrive at the inputs, propagate through these functional elements and define the output signals.

For simplicity, we have ignored how entries corresponding to the “=” construct [BCH⁺91, Ber93] need be taken care of. These nodes are easy to handle. We will outline the necessary changes in Section 5.3.6.

5.3.2 Latches

For every latch, we have a vertex of type C for the input and a vertex of type O . Both are labeled with the domain of the latch. We draw an edge from the input to the output vertex.

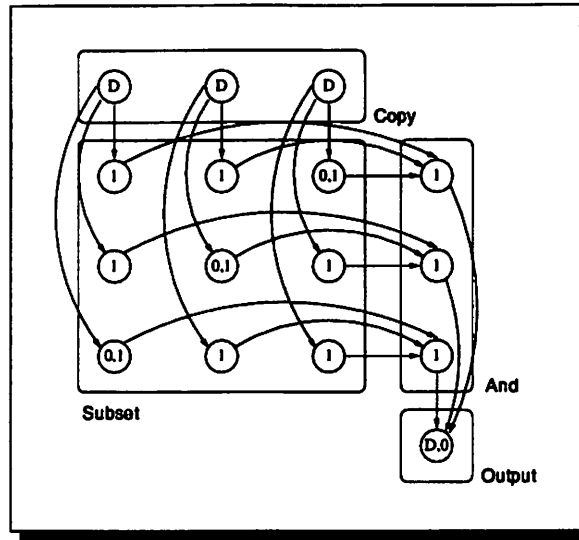


Figure 5.2: Graph for a BLIF-MV table.

5.3.3 Inputs and Outputs

We have one vertex of type O for every primary input, labeled with its domain. We have one vertex of type C for each primary output. All of these are labeled identically.

There are no vertices corresponding to initial values of latches. This essentially leaves the circuit that defines initial values, *floating* i.e. disconnected from the rest of the circuit.

5.3.4 Interconnection Signals

Interconnection signals emanate from a vertex of type O and feed vertices of type C . We draw an edge between the two, coming out of the vertex of type O . At most one edge is incident upon any vertex of type C . We assumed that the circuit is devoid of combinational cycles. As a consequence, removal of all edges between the input and output vertices of all the latches leaves the graph acyclic.

5.3.5 Relationship with $Aut_M L \cdot AP$

For a circuit C , let A_C denote the directed labeled graph whose construction we just described. As before, let $M = (S, R, K)$ be the Kripke structure with 2^L states and a set of

atomic propositions AP . The set L corresponds to the latches. The set AP corresponds to the outputs in \mathcal{O} . The function K represents the boolean predicate on latches that generate outputs.

Definition of $Aut_{A_C} L \cdot AP$

For an automorphism π of A_C , let $\pi_A : L \cup AP \rightarrow L \cup AP$ denote the 1-1 mapping naturally defined by π . Let $Aut_{A_C} L \cdot AP$ denote the set of all such permutations. It is easily verified that $Aut_{A_C} L \cdot AP$ forms a group.

We will leave it to the reader to verify that there is a 1-1 correspondence between structural symmetries of \mathcal{C} and automorphisms of the graph A_C . We will directly prove that the set of automorphisms $Aut_{A_C} L \cdot AP$ is a subset of $Aut_M L \cdot AP$.

Theorem 5.2 $Aut_{A_C} L \cdot AP \leq Aut_M L \cdot AP$.

Proof:

Recall the definition of $Aut_M L \cdot AP$ from Section 3.1.2. We need to show that $(\forall \pi \in Aut_{A_C} L \cdot AP)((\pi L = L) \wedge (\pi_{\langle L \rangle} \in Aut_M L) \wedge (\forall x \in 2^L)(\forall y \in 2^{AP})((K(x) = y) \Leftrightarrow (K(\pi x) = \pi y)))$.

The condition $\pi L = L$ is easily seen to hold from the definition of $Aut_{A_C} L \cdot AP$.

We now prove the second condition $\pi_{\langle L \rangle} \in Aut_M L$.

Let $\pi \in Aut_{A_C} L \cdot AP$ and $s \in S$. We will show that for every $(s, t) \in R$, if $s' = \pi s$ then $(s', t') \in R$ where $t' = \pi t$. This suffices to prove that $\pi_{\langle L \rangle} \in Aut_M L$.

Remove all edges between input and output vertices of latches. For a circuit without combinational cycles, the residual graph is acyclic. Since $(s, t) \in R$, there must exist an assignment p to primary inputs in \mathcal{C} such that when s appears at the latch outputs, the latch inputs for the next clock cycle evaluate to t . In other words, the combinational circuit corresponding to the residual graph evaluates to $\langle t \rangle$ when the input is $\langle s, p \rangle$.

For the purposes of this proof, we associate functionality with each vertex in A that depends on $\langle s, p \rangle$. Vertices of type C are Copiers. Their output is the same as the input value. Note that all Copiers have in-degree at most one, which makes their behavior well defined. Only Copiers corresponding to primary inputs, latch outputs and tables without input columns (e.g. constants) have in-degree zero. Their output is derived appropriately from $\langle s, p \rangle$.

Vertices of type S always have in-degree one, with a vertex of type C incident upon them. They have a boolean output which is TRUE if and only if the incoming value lies in the set which labels that vertex.

Vertices of type A output a 2-tuple. The first is simply the label of that vertex. The second is boolean. It is TRUE if and only if all inputs to the vertex are TRUE. All inputs to such a vertex are of type S . Some vertices of type A have no input vertices. These correspond to tables without input columns (pseudo inputs or constants). Their output is simply the appropriate value from $\langle p \rangle$. Their second output is always TRUE.

Vertices of type O have only vertices of type A incident upon them. If none of the inputs is TRUE, the default value is output. If any of the inputs is TRUE, a domain value from among those that the label of A represents, is output.

We claim that the functionality we just ascribed to vertices correctly captures the semantics of tables and interconnections (something that we never formally stated).

Since $\pi \in \text{Aut}_{A_C} L \cdot AP$, by definition, π is derived from some automorphism Π of the graph A_C . Let p' be the assignment to primary and pseudo inputs obtained by applying Π to p . Similarly let t' be the assignment to latch inputs obtained by applying Π to t .

Consider the circuit A_C with functionality associated with $\langle s, p \rangle$. At the same time, consider the same circuit with the functionality associated with $\langle s', p' \rangle$. By definition, Π preserves the labels of vertices of A_C i.e. if $v' = \Pi v$, then v and v' have the same label and hence the same functionality.

It follows that if the output (at the latch inputs) corresponding to $\langle s, p \rangle$ is t , then the output (at the latch inputs) corresponding to $\langle s', p' \rangle$ is t' . And since there is an edge between each individual latch input and latch output, we get $t' = \pi t$.

Thus we have proved that every $\pi \in \text{Aut}_{A_C} L \cdot AP$ satisfies $\pi \in \text{Aut}_M L$.

The third condition that $(\forall x \in 2^L)(\forall y \in 2^{AP})(K(x) = y) \Leftrightarrow K(\pi x) = \pi y)$ remains to be shown to hold. The proof for this proceeds along the same lines as above.

□

5.3.6 The “=” Construct

For table entries of the form “= x ”, we simply add an edge from the vertex of type C corresponding to the variable x , to the vertex corresponding to the entry “= x ”. We leave it to the reader to prove that Theorem 5.2 still holds.

5.3.7 Is Every Group Possible?

Let $G \subseteq \text{Aut}_M L \cdot AP$. Recall from Section 4.4 that the canonical state problem takes $G_{\langle L \rangle}$, the restriction of G to L , into account. An interesting theoretical question that comes to mind is: Is there any group $G_{\langle L \rangle}$ that does not correspond to any BLIF-MV circuit? If so, then we can focus on the remaining groups to solve the canonical state problem, described in Section 4.4. However, the answer is negative. We defer the proof till Section 6.4.6 because we require the notions of base and strong generating set, which we will define in Chapter 6. In practice, the story might be different; we might come across only a few types of groups.

5.3.8 Identifying Scalarsets

How difficult is it to identify scalarsets automatically?

A scalarset is an automorphism of the graph A_C such that the automorphism can be written as a product of disjoint transpositions. Note that A_C is not an arbitrary directed graph. It has been derived from a valid BLIF-MV circuit. We now show that the problem of identifying scalarsets is as hard as graph isomorphism.

We are given a pair of connected directed graphs, each with n vertices. We say that a vertex u belongs to the set of immediate predecessors of another vertex v if there is an edge from u to v . We construct a BLIF-MV circuit consisting of $2n$ latches, one for each vertex in the two graphs. The input of a latch is produced by a circuit that chooses randomly from among the outputs of other latches that correspond to its set of immediate predecessors. The combinational logic for this circuit can be specified in such a way that the resulting BLIF-MV table is fully symmetric with respect to all its inputs. It is easily shown that the given graphs are isomorphic to each other if and only if the resulting BLIF-MV circuit contains a scalarset that has a transposition of two latches, the two corresponding to vertices in different graphs.

5.4 CTL* Formulas

5.4.1 Graphs for Computing $H \leq \text{Aut}_f AP \cdot SF$

Recall the definition of $\text{Aut}_f AP \cdot SF$ from Section 3.5.1. To compute $H \leq \text{Aut}_f AP \cdot SF$, draw the parse tree for the formula f . Label each internal node with the

operator it corresponds to. The leaf nodes correspond to propositions in AP .

For each internal node that corresponds to the *Until* operator, introduce two new nodes labeled *Left* and *Right*. Replace the edge between *Until* and its left operand by two edges: one from *Until* to *Left* and one from *Left* to the left operand. Replace the edge between *Until* and its right operand similarly.

Collapse all leaf nodes which correspond to the same $p \in AP$ into a single node. These nodes are labeled identically with a color, say *White*.

Introduce a new node for every proposition $p \in AP$. These are labeled identically with a new color, say *Black*. Now, draw an edge from a *White* node to a *Black* node if they correspond to the same atomic proposition p . Some *Black* edges may not have any edge incident upon them if f does not depend on that proposition.

Definition of $Aut_{A_f} AP \cdot SF$

Let A_f denote the graph we constructed. It is clear that the nodes of A_f , except those labeled *Left* or *Right*, are in 1-1 correspondence with elements of $AP \cup SF$. For every automorphism of the graph A , let π denote its restriction to nodes corresponding to $AP \cup SF$. The set of all such π forms a group, which we denote by $Aut_{A_f} AP \cdot SF$.

Theorem 5.3 $Aut_{A_f} AP \cdot SF \leq Aut_f AP \cdot SF$. □

5.4.2 Graphs for Computing $H \leq Aut_f AP \cdot MPS$

Recall the definition of $Aut_f AP \cdot MPS$ from Section 4.2.4. To compute $H \leq Aut_f AP \cdot MPS$ from a CTL* formula f , we first identify MPS , the set of all maximal propositional sub-formulas in f .

For each sub-formula $g \in MPS$, construct the parse tree for g . Label each internal node with the operator it corresponds to. The leaf nodes correspond to propositions in AP . Collapse all leaf nodes which correspond to the same $p \in AP$ into a single node. Label them identically with a new color, say *White*.

Introduce a new node for every proposition $p \in AP$, all labeled identically with a new color, say *Black*. Now, draw an edge from every *White* node to a *Black* node if they correspond to the same atomic proposition p . Some *Black* edges may not have any edge incident upon them if no $w \in W$ depends on that proposition.

Finally, re-label each root node corresponding to a sub-formula $g \in MPS$, with a distinct new color.

Definition of $Aut_{A_f} AP \cdot MPS$

Let \tilde{A}_f denote the graph we constructed. Let $\pi : AP \cup MPS \rightarrow AP \cup MPS$ denote the permutation corresponding to the restriction of some automorphism of \tilde{A}_f to the vertices corresponding to AP and MPS . The labels of \tilde{A}_f ensure that $\pi AP = AP$ and $(\forall g \in MPS)(\pi g = g)$. The set of all such π forms a group, which we denote by $Aut_{\tilde{A}_f} AP \cdot MPS$.

Theorem 5.4 $Aut_{\tilde{A}_f} AP \cdot MPS \leq Aut_f AP \cdot MPS$. □

5.5 Computing $G \bowtie H$

One approach is to compute the groups G and H separately and then compute $G \bowtie H$ using group-theoretic algorithms for group intersection. An advantage of this idea is that we would need to compute G only once. We can compute different groups H for different formulas f . However, computing group intersections is as hard as graph isomorphism [Hof80]. Polynomial time algorithms do exist for special cases [Hof80].

Another approach is to simply join the two graphs corresponding to G and H together at the vertices corresponding to AP . This can be done by drawing an edge between every pair of vertices that correspond to the same $p \in AP$ in both the graphs. The key to correctness lies in the fact that the sets of labels in the two graphs, except for the vertices corresponding to AP , are mutually exclusive.

5.6 The Big Picture

Given a BLIF-MV circuit \mathcal{C} , a CTL* formula f and a set of initial states I , we first compute sets of symmetric sub-formulas of f , as defined in Section 3.3. This can be accomplished by constructing the graphs $A_{\mathcal{C}}$ and A_f , described in Section 5.3 and Section 5.4.1 respectively, joining them as described in Section 5.5 and solving the graph automorphism problem for the resulting graph. The data structure for representing graph automorphisms will be described in Chapter 6. It will allow identification of partitions of sub-formulas of f easily.

We then compute $H \leq \text{Aut}_f AP \cdot MPS$ by constructing the graph \tilde{A}_f described in Section 5.4.2, join it with A_C as described in Section 5.5 and solve the graph automorphism problem for the resulting graph. This would give us generators for the group $G \rtimes H$. We are now ready to take advantage of Theorems 3.1 and 3.2. From the generators for $G \rtimes H$, we compute the function $\xi_{G \rtimes H}$.

Finally, we feed the sets of symmetric sub-formulas and the function $\xi_{G \rtimes H}$ to a modified model checker that understands symmetries. Model checking can be done on the quotient using the techniques outlined in Section 4.3. It can also take advantage of Theorem 3.2 to avoid computing truth values for all sub-formulas.

After having evaluated the truth value of f for all initial states, the modified model checker can start offering new formulas to the designer, whose truth value can be deduced from the symmetry information already computed, as described in Section 3.6. One can envisage an interactive user interface that presents new formulas to the designer in a controlled fashion.

Chapter 6

Computing Graph Automorphisms

We start with definitions in Section 6.1. We then present a branch and bound algorithm for computing a single automorphism of a graph in Section 6.2. We discuss the notion of *refinement* in Section 6.3. It plays an important role in speeding up the branch and bound algorithm. Then we show the necessary changes required in the algorithm for computing all the automorphisms in Section 6.4. Finally, we provide justification for writing our own graph automorphism program in Section 6.5.

We use A to denote a directed labeled graph (V, E) . The number of vertices and edges of a graph will be denoted by n and m respectively.

6.1 Definitions

Bipartition

Let V be a set of vertices. A bipartition P is a set of ordered pairs $\cup_{1 \leq i \leq k} \{(V_i, W_i)\}$, where

1. $\cup_{1 \leq i \leq k} V_i = \cup_{1 \leq i \leq k} W_i = V$
2. $(\forall i. 1 \leq i \leq k)(|V_i| = |W_i| \neq 0)$
3. $(\forall j. 1 \leq i < j \leq k)(V_i \cap V_j = W_i \cap W_j = \phi)$

The set of edges of a graph or its labeling function play no role in the definition.

Unipartition

A bipartition P is a unipartition if $(\forall i.1 \leq i \leq k)(V_i = W_i)$. It is simply a partition of the set of vertices V into disjoint non-empty sets.

Refinement

A bipartition $Q = \cup_{1 \leq i \leq q} \{V_i^Q, W_i^Q\}$ is a refinement of another bipartition $P = \cup_{1 \leq i \leq p} \{V_i^P, W_i^P\}$ if they are defined over the same set of vertices V and $(\forall i.1 \leq i \leq q)(\forall j.1 \leq j \leq p)((V_i^Q \cap V_j^P = \phi) \vee (V_i^Q \subseteq V_j^P \wedge W_i^Q \subseteq W_j^P))$. We denote this relationship by $Q \preceq P$. We also say that P is coarser than Q and that Q is finer than P . The relation \preceq is reflexive and transitive.

Compatibility

Two bipartitions $P = \cup_{1 \leq i \leq p} \{(V_i^P, W_i^P)\}$ and $Q = \cup_{1 \leq i \leq q} \{(V_i^Q, W_i^Q)\}$ are compatible if $(\forall i.1 \leq i \leq p)(\forall j.1 \leq j \leq q)(|V_i^P \cap V_j^Q| = |W_i^P \cap W_j^Q|)$.

Intersection

The intersection of two compatible bipartitions P and Q is defined as $P \wedge Q = \cup_{1 \leq i < p, 1 \leq j \leq q} \{(V_i^Q \cap V_j^P, W_i^Q \cap W_j^P)\} - \{(\phi, \phi)\}$, which itself is a bipartition.

Note that among the definitions introduced so far, none involves the edges or labels of a graph. The next one does.

Automorphism

Let $A = (V, E)$ be a directed labeled graph with labeling function c . A bipartition P is an automorphism of A if

1. $(\forall i.1 \leq i \leq k)(|V_i| = |W_i| = 1)$
2. $(\forall v, w \in V)(\forall i.1 \leq i \leq k)((v \in V_i \wedge w \in W_i) \Rightarrow (c(v) = c(w)))$
3. $(\forall v, v', w, w' \in V)(\forall i.1 \leq i \leq k)(\forall j.1 \leq j \leq k)((v, v') \in E \Leftrightarrow (w, w') \in E)$

This definition is no different from that given earlier in Section 2.6. We have presented it in a slightly different notation to allow convenient presentation of algorithms.

The set of all automorphisms of A forms a group. We denote it by $Aut A$.

Consistency

A bipartition P is consistent with an automorphism if there exists an automorphism P' of A such that $P' \preceq P$.

For notational convenience, we will denote both a vertex $v \in V$ and a singleton set $\{v\}$ by simply v . This would allow us to write a set like $\{\{u\}, \{v\}\}$ as (u, v) .

6.1.1 Properties of Bipartitions

The following lemmas are immediate:

Lemma 6.1 *If P, Q and R are bipartitions of A such that $P \preceq Q$ and $P \preceq R$, then Q and R are compatible and $P \preceq Q \wedge R$.* \square

Lemma 6.2 *For a directed labeled graph A , let U_{max} be a unipartition such that two vertices of A lie in the same set if and only if they have the same label. Then, U_{max} is consistent with every automorphism of A .* \square

Lemma 6.3 *For a directed labeled graph A , let U_{min} be a unipartition such that two vertices u and v lie in the same set if and only if $(\exists \pi \in \text{Aut } A)(\pi u = v)$. Then, U_{min} is consistent with every automorphism of A and is the finest such unipartition.* \square

Lemma 6.4 *If P is a bipartition such that $P \in \text{Aut } A$ and $(u, v) \in P$, then $P \preceq \{(succ(u), succ(v)), (V - succ(u), V - succ(v))\}$, where $succ(x) = \{y \mid (x, y) \in E\}$.* \square

Lemma 6.5 *If P is a bipartition such that $P \in \text{Aut } A$ and $(u, v) \in P$, then $P \preceq \{(pred(u), pred(v)), (V - pred(u), V - pred(v))\}$, where $pred(x) = \{y \mid (y, x) \in E\}$.* \square

6.1.2 Problem Definition

Given a bipartition P for a directed labeled graph $A = (V, E)$, produce an automorphism of A consistent with P , if one exists.

We now present a branch and bound solution for this problem.

```

SEARCH_AUTOMORPHISM(Graph A, Bipartition P)
  Compute  $U_{max}$ ;
   $U = \text{REFINE}(U_{max})$ ;
  if (COMPATIBLE (P, U))
     $P = P \wedge U$ ;
  else return 0;
  return BRANCH_AND_BOUND(A, P,  $\phi$ );

```

```

BRANCH_AND_BOUND (Graph A, Bipartition P, PairSet S)
  while (( $\exists u, v \in V$ )( $(u, v) \in P \wedge (u, v) \notin S$ ))
     $S = S \cup (u, v)$ ;
     $Q = (\text{succ}(u), \text{succ}(v)) \cup (V - \text{succ}(u), V - \text{succ}(v))$ ;

    if (COMPATIBLE (Q, P))
       $P = P \wedge Q$ ;
    else return 0;

     $Q = (\text{pred}(u), \text{pred}(v)) \cup (V - \text{pred}(u), V - \text{pred}(v))$ ;

    if (COMPATIBLE (Q, P))
       $P = P \wedge Q$ ;
    else return 0;
  if (SET_COMPLETE(A, S))
    return 1;

  ( $v, W$ ) = CHOOSE_VICTIM (P);
  foreach ( $w \in W$ )
     $P' = P - (V, W) \cup (v, w) \cup (V - v, W - w)$ 
     $S' = S$ ;
    if BRANCH_AND_BOUND (A, P', S')
      return 1;
  return 0;

```

Figure 6.1: Algorithm for finding an automorphism, given graph A and bipartition P .

6.2 Branch and Bound Algorithm

Pseudo-code for the algorithm is given in Figure 6.1 on the preceding page.

We start by computing U_{max} , as defined in Lemma 6.2, since U_{max} is consistent with every automorphism of A . Ideally, we would like to start with U_{min} , as defined in Lemma 6.3, as it is the finest such partition. However, computing U_{min} itself is as hard as graph isomorphism [vL90]. Therefore, we compute an approximation to U_{min} , namely a unipartition U such that $U_{min} \preceq U \preceq U_{max}$ using the function `REFINE`. We will describe this function in more detail in Section 6.3. Note that U is consistent with every automorphism of A .

After computing U , we check whether P and U are compatible. If not, then from Lemma 6.1, we deduce that P is not consistent with any automorphism of A ; the algorithm terminates. If P and U are compatible, we compute their intersection $P \wedge U$. From Lemma 6.1, any automorphism consistent with P and U has to be consistent with $P \wedge U$. Finally, we invoke `BRANCH_AND_BOUND`.

6.2.1 The Bounding Step

The `while` loop in `BRANCH_AND_BOUND` is the bounding step. From Lemma 6.4, we conclude that if $(u, v) \in P$ and if there exists an automorphism consistent with P , then it has to be consistent with $Q = \{(succ(u), succ(v)), (V - succ(u), V - succ(v))\}$. From Lemma 6.1, we conclude that P and Q must be compatible and that the automorphism must be consistent with $P \wedge Q$ as well. A similar argument holds for $Q = \{(pred(u), pred(v)), (V - pred(u), V - pred(v))\}$ also. Thus, the non-compatibility of P and Q is evidence that there is no automorphism consistent with P and `BRANCH_AND_BOUND` terminates.

The bounding step also helps to refine P by computing $P \wedge Q$, which in turn might generate new singleton pairs $(u, v) \in P$. Intuitively, the *implications* of mapping u to v are getting *propagated*. The set S remembers such pairs (u, v) , thereby avoiding duplicate work. The `while` loop terminates when no such pairs remain. At this point, all pairs in P which have size one, lie in S . The function `SET_COMPLETE` checks whether all vertices in V have found their way into S . If so, we have discovered an automorphism¹. Otherwise, it is time to branch.

¹There is no need at this point to verify the three conditions laid down earlier for a bipartition to be an automorphism. See the proof of correctness.

6.2.2 The Branching Step

The branching step is straightforward. The routine `CHOOSE_VICTIM` first selects a pair (V, W) in P such that $|V| \neq 1$ using some heuristic. It then selects a vertex $v \in V$ using another heuristic and returns (v, W) . The choice of v and W is important for at least two reasons. First, small sized W implies fewer branches to follow. Second, branches that lead to dead ends need be avoided. Our implementation is not fancy: we simply choose the smallest sized W available, breaking ties arbitrarily; our choice of $v \in V$ is also arbitrary.

Having chosen v and W , we try to discover $w \in W$ such that v maps to w in some automorphism of A . To this end, we compute $P' = P - (V, W) \cup (v, w) \cup (V - v, W - v)$ and invoke `BRANCH_AND_BOUND`. Clearly, if all choices of w fail, there is no automorphism consistent with P and the function terminates unsuccessfully.

6.2.3 Proof of Correctness

The correctness of `BRANCH_AND_BOUND` is obvious from the following lemma, whose proof is simple.

Lemma 6.6 *For a directed graph $A = (V, E)$, a bipartition $P = \cup_{1 \leq i \leq n} \{(V_i, W_i)\}$ is an automorphism of A if and only if*

1. $(\forall i. 1 \leq i \leq n)(|V_i| = |W_i| = 1)$
2. $P \preceq U_{max}$
3. $(\forall i. 1 \leq i \leq n)(\forall v, w \in V)((v \in V_i \wedge w \in W_i) \Rightarrow (P \preceq \{(succ(v), succ(w)), (V - succ(v), V - succ(w))\}))$

□

Condition 1 in Lemma 6.6 is true at the end of the algorithm, as we invoke `SET_COMPLETE` to verify it. Condition 2 follows from the fact that we compute $P \wedge U$ in `SEARCH_AUTOMORPHISM` where $U \preceq U_{max}$. Condition 3 is checked for each vertex pair in the `while` loop. The entire branch and bound algorithm simply verifies Condition 3 for each vertex pair generated by the branching step.

6.2.4 Time Complexity

Compatibility of two bipartitions P and Q can be checked in $O(n)$ time, where $n = |V|$. The intersection $P \wedge Q$ can also be computed in $O(n)$ time. Computing U_{max} is trivial.

From Figure 6.1 on page 48, it might appear that each level of recursion requires a different bipartition P and set S . However, this is not necessary. The trick lies in remembering set boundaries at each recursion level and quickly merging subsets when backtracking. We have implemented the algorithm using only nine arrays of size n , apart from the usual adjacency lists for graph edges.

Assuming that we never backtrack and that `CHOOSE_VICTIM` returns sets of bounded size (in our experiments, we rarely encountered sets of size three or four), our implementation runs in $O(m+n)$ time. Thus, we take optimal time provided `CHOOSE_VICTIM` makes good choices. In the case of a bad choice, the bounding step is expected to quickly propagate its implications and produce evidence that no automorphism is consistent with this choice.

Summary

The algorithm in Figure 6.1 on page 48 solves the problem of producing an automorphism, if one exists, given a graph A and a bipartition P . It is possible to produce *all* such automorphisms by continuing the search even after discovering one. However, this is not desirable as the total number of such automorphisms could be exponential in n . We need a succinct representation of $Aut A$ and an algorithm to compute it. We describe both in the Section 6.4, where the algorithm in Figure 6.1 serves as a backbone.

Our initial motivation for implementing Algorithm 6.1 was to convince ourselves that our modeling of the circuit is sufficient to allow discovering structural symmetries. As it stands, it is useful in the following scenario: A circuit verifier might suspect that certain symmetries exist in the circuit at hand. She can provide a bipartition using her intuition and ratify it by running this algorithm. Then she can use the theory for symmetric sub-formulas developed in Chapter 3.

6.3 Refinement

Given a graph $A = (V, E)$, the function REFINE in Figure 6.1 on page 48 computes a unipartition U such that $U_{min} \preceq U \preceq U_{max}$. Why is refinement useful? First, it might generate singleton pairs (u, v) whose implications can be propagated immediately in the bounding step. Second, by shrinking the sizes of individual pairs of sets, fewer branches may have to be taken later on. Ideally, for a graph that has no non-trivial automorphisms, the refinement step should produce a bipartition all of whose sets are singletons.

6.3.1 Computing U using Vertex Invariants

A unipartition U can also be looked upon as a function that computes the same value for two vertices if they lie in the same set. Some such functions are easy to compute from a description of $A = (V, E)$ with the guarantee that $U_{min} \preceq U$. Having computed two such functions U_1 and U_2 , we can compute the intersection $U_1 \wedge U_2$, which is also guaranteed to be at least as coarse as U_{min} . A good exposition of these ideas can be found in [FHG⁺83], where such functions are called *vertex invariants*. Mittal [Mit88] also uses vertex invariants to compute automorphisms.

Some vertex invariant that can be computed in $O(m + n)$ time are the in-degree and out-degree of vertices, the set of degrees of vertices incident at a vertex and the set of degrees of vertices which a vertex is incident upon. If the graph is acyclic, the *levels* of vertices constitute a vertex invariant.

6.3.2 Other Vertex Invariants

We can compute some more vertex invariants at the cost of more computation by constructing the distance matrix for A [SD76]. Entries in the distance matrix are the lengths of the shortest path between pairs of vertices. Let d_{ij} be the number of vertices that are a distance j from i . Then, the function that computes the sequence $\langle d_{i1}, d_{i2}, \dots, d_{in} \rangle$ for vertex i is a vertex invariant. It remains so even under some other interpretations of d_{ij} . One such interpretation is the number of edges that are at a distance j from vertex i . Another is the number of vertices whose shortest path to i is of length j .

The distance matrix can be computed using an algorithm by Seidel [Sei95] in $O(n^2 \log n)$ expected time. In practice, graphs are sparse and the out-degree of any vertex

can be bounded by a small constant. In such a case, we are better off using n different breadth first searches to compute the distance matrix for a total time of $O(n^2)$.

Several other vertex invariants, which are useful for special types of graphs can be found in [McK90]. In practice, we should use only those that can be computed in linear time.

6.3.3 Repeated Refinement

An important trick is to use a unipartition U to refine itself. Let U be looked upon as a labeling function. Define a function U' as follows: for a vertex v , it computes the set of labels of vertices incident upon v . Then U' is a unipartition and a vertex invariant such that $U_{min} \preceq U'$ [FHG⁺83]. We can now refine U by computing the intersection $U \wedge U'$. This can be done repeatedly until $U = U'$.

Computing U' and $U \wedge U'$ requires $O(m + n)$ time. We stop after at most n iterations. Thus repeated refinement requires polynomial time. In practice, we stop after a few iterations. See Table 7.2 on page 63 for the number of iterations required for graphs that we experimented with.

6.4 Computing the Automorphism Group of a Graph

What is a good representation for $Aut A$ and how do we compute it?

6.4.1 Base and Strong Generating Set

Charles Sims [Sim70] introduced the idea of using a base and strong generating set for representing a group. This is a key concept underlying essentially all polynomial-time algorithms in computational group theory. A good exposition can be found in the book by Butler [But91].

Let G be a permutation group acting on a set Ω with cardinality n . Let $x \in \Omega$ be any point in the set. Let πx denote the image of x under the action of $\pi \in G$. A permutation π is said to *stabilize* an element x if $\pi x = x$. If $H \leq G$ and $\pi \in G$, the set of elements $\{\pi x \mid x \in H\}$ is said to be a right coset of H . Two right cosets of H are either the same or disjoint. All right cosets have the same cardinality.

Define $G_x = \{\pi \mid \pi x = x, \pi \in G\}$ i.e. the set of permutations that stabilize x .

This set is a subgroup. Let $x^G = \{y \mid \pi x = y, \pi \in G\}$ i.e. the orbit of x under G . It turns out that for each $y \in x^G$, the set $\{\pi \mid \pi x = y, \pi \in G\}$ is a right coset of G_x . All these right cosets have the same cardinality. Their union is the set G .

Multiplying the subgroup with a member of a particular coset generates that coset. Given a subgroup, a set of permutations consisting of exactly one member from each coset, is said to be a set of coset representatives. For a group G and an element x , let R_x be the set of coset representatives for G_x . Then, every permutation in G can uniquely be written as a product of two permutations, one from G_x and the other from R_x . Effectively, we have factored G . Continuing in this fashion, we can factor G_x with respect to another point x' . A given sequence $\langle x = x_1, x_2, \dots, x_k \rangle$ of points is said to be a *base* if the sequence of sets $R_x = R_{x_1}, R_{x_2}, \dots, R_{x_n}$ is such that R_{x_k} consists only of the identity permutation. The union of all these sets is said to be a *strong generating set*. Note that any ordering of Ω can act as a base. In general, some R_{x_i} might consist only of the identity permutation. The corresponding x_i can be removed from the base to obtain a *reduced base*.

Sims [Sim70, Sim71] presented an algorithm for constructing a base and strong generating set from a set of generators of size s . The time complexity of the algorithm was shown to be $O(n^6 + sn^2)$ by Furst, Hopcroft and Luks [FHL80] for a version of Sim's algorithm. In a note in 1981 (a preliminary version of [Knu91]), Knuth gave a variant with running time $O(n^5 + sn^2)$. The same bound was also achieved by Jerrum [Jer86]. Subsequently, an $O(n^4 \log^c n + sn^2)$ algorithm was discovered by Babai, Luks and Seress [BLS97].

6.4.2 Jerrum's Representation

Jerrum [Jer86] devised an elegant data structure called *labeled branchings* to store a strong generating set in $O(n^2)$ space. Essentially, a given strong generating set is reduced to at most $n-1$ permutations and stored in a certain way. Assuming that each permutation π is represented by storing the images of elements of Ω in an array, the bound of $O(n^2)$ is optimal as there exist groups which require at least $\Omega(n)$ generators [Jer86].

The data structure supports fast membership testing in $O(n^2)$ time. Given a base $\langle x_1, x_2, \dots, x_k \rangle$, computation of the orbits of the subgroup G_{x_1, x_2, \dots, x_i} for any i is also readily accomplished.

6.4.3 Graph Automorphism Algorithms

For several classes of combinatorial objects such as graphs, Latin squares, block designs, Hadamard matrices and error-correcting codes, there is a notion of isomorphism mapping one object into another. The problem of finding whether two objects are isomorphic and the related problem of computing the automorphism group of a single object, have received considerable attention [Leo84, BL85]. The paper by Leon [Leo91] presents powerful techniques for general combinatorial objects.

One of the first practical graph isomorphism algorithms was proposed by McKay [Bre76, McK77]. It employed branch and bound and was soon improved [McK81]. Subsequently, Butler and Lam [BL85] presented a modification that prunes false branches by dynamically changing the base and strong generating set. Leon [Leo84] also discusses similar techniques. All the algorithms employ backtrack search.

The overall idea is similar to the algorithm in Figure 6.1 on page 48, i.e. branch by trying to map a point v to a set of candidates W and bound if we have evidence that it is impossible to extend the current partial permutation to an automorphism. A new automorphism is used to augment the subgroup discovered so far. This subgroup also serves to prune search paths that lead only to those permutations that already lie in it. Such a pruning mechanism is based on a lemma by Butler and Lam [BL85].

The lemma is useful when we are trying to extend a partial permutation by mapping v to some vertex in W and a subgroup G of $\text{Aut } A$ is at hand. In such a case, it is not necessary to try to map v to all the vertices in W . It suffices to try to map v to exactly one member from each orbit of the subgroup G_{w_1, w_2, \dots, w_j} , where each w_i is such that some vertex v_i has been mapped to it. Formally,

Lemma 6.7 [BL85]

Let a bipartition $\cup_{1 \leq i \leq k} \{(V_i, W_i)\}$ be such that every vertex in $\{w_1, w_2, \dots, w_j\}$ belongs to some singleton set. Let $v \in V$. Let G be a known subgroup of $\text{Aut } A$. Let G_{w_1, w_2, \dots, w_j} be the subgroup of G that centralizes each of w_1, w_2, \dots, w_j . Let x, y lie in some orbit of G_{w_1, w_2, \dots, w_j} . Then x, y have to belong to some set \tilde{W} such that every automorphism consistent with $P - (V, \tilde{W}) \cup (v, x) \cup (V - v, \tilde{W} - x)$ can be obtained by composing some automorphism in G_{w_1, w_2, \dots, w_j} with an automorphism consistent with $P - (V, \tilde{W}) \cup (v, y) \cup (V - v, \tilde{W} - y)$. \square

What operations need be supported by a data structure that represents groups

using a base and strong generating set? First, the application of Lemma 6.7 requires availability of $G_{w_1, w_2, \dots, w}$, and its orbits at each search step. This is straightforward provided we have the ability to efficiently change the base and the corresponding strong generating set. Second, we need the ability to augment the group by adding a newly discovered permutation.

6.4.4 Base Change Algorithms

Base change algorithms require elementary group theory but are somewhat involved. Sims proposed the first base change algorithm [Sim71]. Butler and Lam [BL85] showed that Sim's base change algorithm has worst case time complexity $O(n^5)$. The algorithm can transpose two adjacent points in the base in $O(n^3)$ time. Since $O(n^2)$ such transpositions suffice to transform a base into any other, we get the time bound of $O(n^5)$. Brown, Finkelstein and Purdom [BFJ89] generalized this idea to computing a cyclic right shift of any contiguous subset of the base in $O(n^2)$ time, using Jerrum's labeled branchings. Since $O(n)$ such cyclic shifts suffice to transform a base into another, the worst case time complexity is $O(n^3)$. Recently, Las Vegas algorithms have also been proposed [CF94, BCF⁺95, BLS] with expected time an order of magnitude less.

6.4.5 Extending our Algorithm

How do we extend our algorithm to compute the automorphism group of a directed labeled graph? The best algorithm appears to be the following: We run the algorithm in Figure 6.1 on page 48 as before but do not stop upon discovering the first automorphism. We also maintain the group formed by automorphisms discovered so far using Jerrum's labeled branchings. As new automorphisms are discovered, we augment this group using the technique described in [Jer86]. Alternatively, the randomized algorithm in [BLS] can be employed. As we backtrack, we dynamically change base using the algorithms in [BFJ89] or use the randomized algorithm described in [CF94]. As we branch, we use Lemma 6.7 to avoid re-discovering any automorphism already in the group [BL85]. The structure of the group can also prune out some branches that are guaranteed to not lead to any automorphism. However, discovering them requires much computation [Leo84].

In practice, it might be useful to tailor our algorithm to perform really well on the kind of symmetries that occur in practice. It might also be advantageous to compute only

a sub-group of the automorphism group but save time.

6.4.6 Is Every Group Possible?

We take a short digression to answer a question that we posed in Section 5.3.7. Given a group G on the set L , we show how a BLIF-MV circuit can be constructed so that the group $G' = \text{Aut}_M AP \cdot L$ defined for the corresponding Kripke structure is such that the group $G'_{\langle L \rangle}$ is exactly G .

We let $L = AP$. Let G be represented in the form of a base and strong generating set. Let the sequence of coset representatives, as defined in Section 6.4.1, be R_1, R_2, \dots, R_n , where $n = |AP|$. We define n BLIF-MV sub-circuits C_1, C_2, \dots, C_n , each with $|L|$ inputs and $|L|$ outputs. The sub-circuit C_i simply permutes the inputs according to some permutation chosen randomly from among those in R_i . For $1 \leq i < n$, the outputs of C_i are the inputs of C_{i+1} . The outputs of C_n are the inputs of the $|L|$ latches. The outputs of the latches are the inputs of C_1 .

Let M be the Kripke structure corresponding to the BLIF-MV circuit we just constructed. It can be shown that the $G' = \text{Aut}_M L \cdot AP$ is such that $G'_{\langle L \rangle}$ is exactly the group G .

6.5 Why a New Graph Automorphism Program?

There exist some software packages for manipulating graphs and groups. One of the earliest graph automorphism programs was written by McKay [McK81]. It is still available as *nauti* [McK90] and can be used with a package called GRAPE [Soi90], which provides routines for graphs and groups. GRAPE itself is one of the several packages provided by GAP [Gap], which is a general purpose software package for combinatorial objects. Another general purpose package is MAGMA [W. 96], which has been derived from an earlier package called CAYLEY [Can84].

What is the motivation for writing a new graph automorphism program? First, existing packages are general purpose and carry around a lot of baggage. In our experience, they are slow. We can specialize our algorithm for labeled graphs. The labels play a crucial role in our branch and bound algorithm. Also, we do not expect the out-degree of vertices to be very large. We can also specialize for the kinds of groups we expect to encounter in

practice. The symmetric group, cyclic groups and their combinations are most common in hardware circuits.

We now list some more compelling reasons for writing our own graph automorphism program.

6.5.1 Domain Specific Optimizations

First, we expect our graphs to have tens of thousands of vertices. However, we are interested in the group formed by only those vertices that correspond to latches and primary outputs. See Chapter 5 for details. Therefore, it appears wasteful to first compute the group over all the vertices and then restrict it to the vertices of interest. This is what existing packages offer. We can modify our algorithm as follows: We maintain a group only on the vertices of interest. The branching step always chooses v from among the vertices of interest, if one is available. At some point, all such vertices would have been exhausted. Then, we are simply interested in discovering a single extension to the current partial permutation. This discovery can be made by using the algorithm in Figure 6.1 without modification.

Second, some BLIF-MV tables could be very large. In Chapter 5, we described our construction of a graph from a table, where we have vertices for all inputs, outputs and entries of a table. Note that vertices of a table can be mapped only onto vertices of another table with the same structure. The key idea is that the images of individual entries can be inferred from the images of inputs, outputs and *rows* of a table. We need not store the images of entries at all. This can be used to save space.

Third, structural languages like BLIF-MV allow hierarchical specification of the circuit, with subcomponents instantiated multiple times. Our graph is essentially a representation of the flattened circuit. Thus several of its subgraphs are isomorphic to each other. These can be identified *a priori* from the hierarchical structure of the specification. We can develop techniques to store isomorphic subgraphs compactly. Intuitively, it is very likely that these subgraphs will map only among themselves.

Fourth, we would like to extend our framework to hierarchical verification. A large circuit is usually composed of several subcomponents. Having identified the symmetries of a subcomponent and used them to verify its properties, we would like to re-use our knowledge of symmetries for verifying the larger circuit. Using the techniques described so far in this report, we would have to construct a flattened circuit, build a gigantic graph

and solve the graph automorphism problem restricted to latches and primary outputs. An alternative is to build machinery to handle the problem of computing the automorphisms of a graph constructed with black boxes and interconnections, with known permutation groups attached to each black box. Such a setup can also be useful when we use standard libraries as building blocks. It is easy to identify the groups of fairly large sized blocks such as multiplexers, register files, adders etc. in the library once and for all. We have some preliminary ideas for solving this black-box graph problem and intend to explore them further.

For completeness, we list some other optimizations that can be carried out, though these are not justifications for writing a new automorphism program.

6.5.2 Further Optimizations

On several occasions, by analyzing a table, we can infer that the only permutation among its inputs is identity. If we can also reliably compute the set of tables that have the same functionality as this one, we can build a graph much smaller in size than what we described in Section 5.3. We still have vertices for input and output ports. However, there are no vertices for entries. The labels of all vertices are distinct. Any other table with the same structure has the same set of labels.

There is a tradeoff between the cycle form and the image form for storing permutations. The former requires us to store merely those cycles which have length at least 2. Thus, finding the image of an element cannot be done in $O(1)$ time in general. Storing the images of all elements requires $O(n)$ space but buys us $O(1)$ time image retrieval. The cycle form can be useful if only a few elements are affected by the action of a permutation.

As a closing remark, we wish to mention that the algorithms listed in Section 6.4 are far from being linear in the size of the set on which the permutations act. Thus, we need some insight into their workings and employ implementation tricks to make them fast. In practice, we need linear time algorithms as the graphs we deal with would have tens of thousands of vertices.

Chapter 7

Results and Future Work

We have implemented an algorithm for constructing a graph from a BLIF-MV [BCH⁺91, Ber93] description, as described in Chapter 5. We have also implemented the search algorithm described in Chapter 6 to search for an automorphism. Our package is integrated with VIS [BHSV⁺96], a verification system developed jointly by researchers at University of California at Berkeley and University of Colorado. The source code for our work is available at <http://www-cad.eecs.berkeley.edu/~manku/symmetries/>.

7.1 Results

Our experiments deal with both sequential and combinational circuits. We start with a Verilog description. We then obtain a BLIF-MV description using a compiler called `v12mv` written by Cheng [CYB93]. The BLIF-MV description is *flattened* using a standard VIS command. The flattened description and a set of partial mappings is then fed to our program. Our program first generates a suitable labeled directed graph, then refines the labels using the techniques mentioned in Section 6.3 and finally runs the branch and bound graph automorphism algorithm.

The graph obtained from a Verilog description `ping-pong.v` is drawn in Figure 7.1 on the following page.

Table 7.1 on page 63 shows n , m and c , the number of vertices, edges and colors respectively, in the initial graph. Since refinement impacts the running time of the algorithm greatly, we also tabulate the number of colors after successive refinement steps. The value c_1 denotes the number of colors after the in-degree and out-degrees have been used as vertex

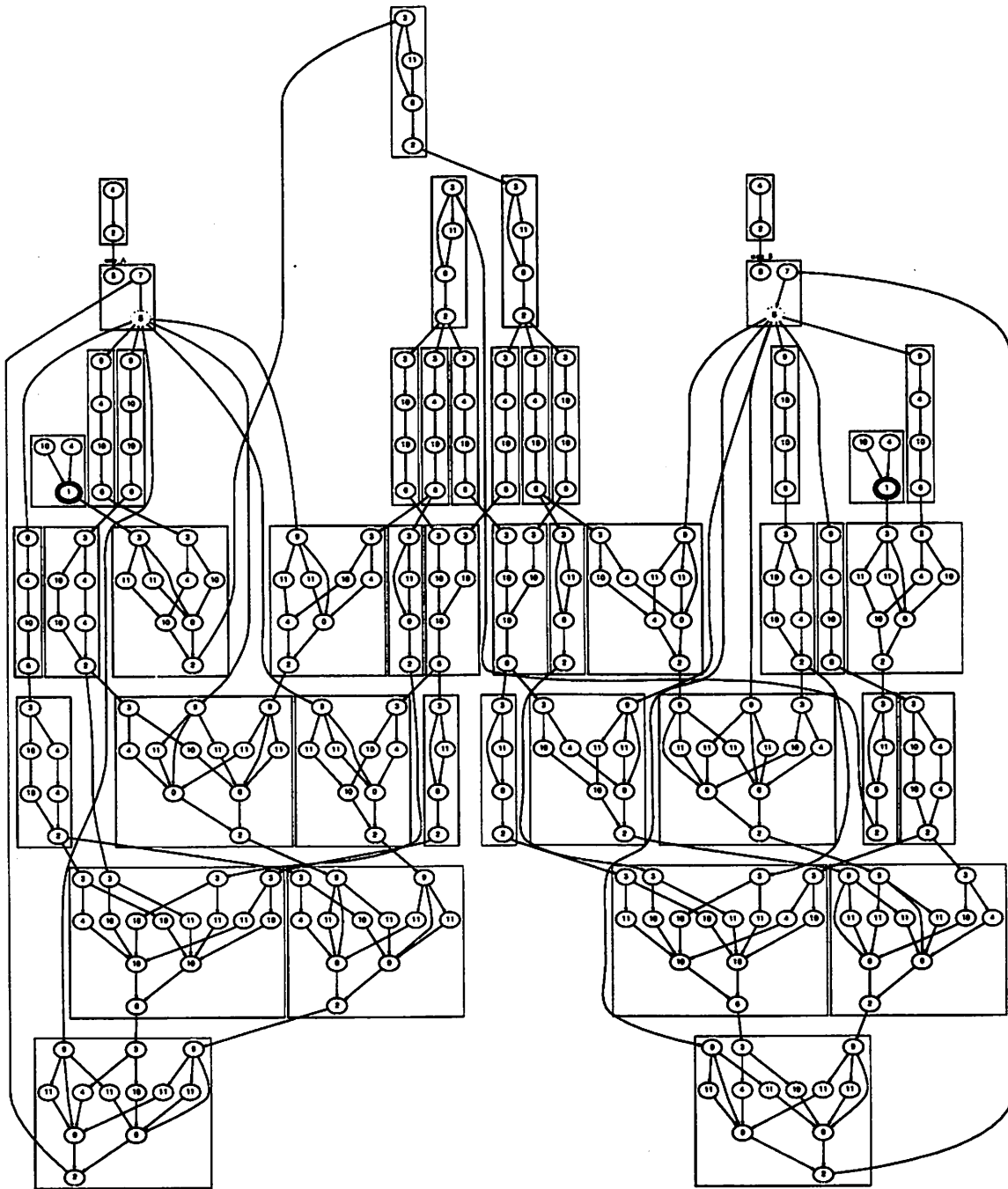


Figure 7.1: Labeled directed graph obtained from ping-pong.v. Rectangles correspond to tables. The nodes for primary inputs and latches are drawn with differently. There are no outputs. The numbers in vertices denote their labels.

<i>Example</i>	<i>n</i>	<i>m</i>	<i>c</i>	<i>c₁</i>	<i>c₂</i>	<i>c_f</i>	<i>iter</i>
ctlp20	4920	6740	15	34	51	246	7
ping-pong	288	378	11	25	39	144	7
z4ml	527	929	5	14	19	108	4
4-arbit	3158	4000	19	52	105	3110	60

Table 7.1: *Some statistics for graphs produced for different examples.*

<i>Example</i>	<i>back-track</i>	<i>maxset</i>	<i>numchoices</i>
ctlp20	0	20	19
ping-pong	0	2	1
z4ml	8	4	21

Table 7.2: *Performance of branch and bound algorithm for different examples.*

invariants. The value c_2 denotes the number of colors after the set of in-degrees of fan out vertices and the set of out-degrees of fan out vertices have been used as vertex invariants. The value c_f denotes the final number of colors after iterative refinement has been carried out. The column *iter* is the number of iterations required.

Table 7.2 lists the number of times our branch and bound algorithm had to backtrack, the number of times the routine CHOOSE_VICTIM in Figure 6.1 on page 48 was invoked and the maximum size of the set returned by this routine. Recall that our algorithm runs in linear time if we never backtrack and if the size of sets returned by CHOOSE_VICTIM is bounded. Table 7.2 clearly shows that the two conditions are almost satisfied.

We were able to identify symmetries in all the examples listed in Table 7.1. The example ping-pong implements a game of ping-pong between two players. The players form a set of fully symmetric system of size 2. The example ctlp20 is a solution for the dining philosophers problem for 20 philosophers. The set of philosophers constitute a cyclic group.

The example z4ml is an interesting combinational circuit with seven inputs, four outputs and four tables, one for each output. A close examination of the tables shows that the inputs can be partitioned into three sets of fully symmetric variables with different sizes.

The running time of our algorithm is small.

7.2 Conclusions

We have developed a fully automatic framework for identifying and exploiting symmetries in structural description of circuits and CTL* formulas. We have shown how the set of sub-formulas of a formula can be partitioned into equivalence classes such that it suffices to evaluate the truth value of only one member in each class for CTL* model checking. This idea can be used by both implicit and explicit methods. We have also shown how we can generate new formulas whose truth value is available at little cost. One can envisage an interactive user interface which provides users with such formulas in a controlled manner. We can also use this theory for pruning sub-formulas from a big formula so that it suffices to prove the resulting smaller sized formula.

We have cast the theory of symmetries developed for Kripke structures by Clarke et al [CEFJ96] and Emerson and Sistla [ES96] into a common framework and generalized their results. We have identified and describe some issues related to modifying a model checker that exploits symmetries. To this end, we have outlined a new algorithm for solving the canonical state problem using symbolic techniques.

We have formalized the notion of structural symmetries in a structural specification language and a CTL* formula. We have shown how they relate to symmetries in Kripke structures. We have also shown how they can be computed automatically, by constructing suitable directed labeled graphs and solving the graph automorphism problem.

We have also developed and implemented an algorithm for finding whether a graph has a non-trivial automorphism. It can be used for finding whether there exists an extension of a partial permutation specified by a circuit designer. We have also described how our algorithm can be extended to compute the entire set of automorphisms of the graph. This brings considerable computational group-theoretic machinery into play.

A limitation of our framework is that it exploits only the structure of the specification and the formula. Thus we can identify fewer symmetries than we would were we to take their complete functionality into account. However, it is not clear how that can be accomplished. We believe that structural symmetries are sufficient for several circuits that occur in practice.

7.3 Future work

Several hurdles need be crossed before quotient structures can routinely be used in model checking algorithms.

Fundamental Problems

Recall the three subproblems described in Section 4.5. First, we need a mechanism to identify the group that captures the symmetries in the specification. We have done this for structural languages. It would be interesting to extend these ideas to higher level specification languages. Second, we need to solve the canonical state problem. As a last resort, we can solve it for special cases occurring in practice. Third, we need to integrate knowledge of symmetries into existing model checkers.

Hierarchical Verification

We can extend these ideas to hierarchical verification. Having computed the symmetries of subcomponents (and proved some properties thereof), we should be able to leverage this knowledge when computing the symmetries of the entire circuit. This would require major changes to our algorithm outlined in Figure 6.1 on page 48.

We might benefit from identifying symmetries in low level components completely, taking not only their structure but also their functionality into account. One idea is to build BDDs for small tables.

Programs for Graph Automorphism

The number of vertices in our graphs are of the order of tens of thousands. Hierarchical specification allows *a priori* identification of isomorphic subgraphs. It would be useful to build machinery to exploit this knowledge.

We also need to get a good implementation of the algorithm for computing the automorphism group of a graph. Some optimizations have been outlined in Chapter 6.

Debugging

An interesting situation arises when the designer believes that a certain symmetry should hold in her circuit but our tool thinks otherwise. What do we tell the designer?

A binary yes/no answer is clearly unsatisfactory. We need to present the reason for the absence of some symmetry in a succinct way without assuming the designer to be proficient in group theory or labeled graphs.

Speeding up Combinational Verification

Finally, we have an idea for speeding up functional verification of combinational circuits by using symmetries. A straightforward algorithm would first build the BDDs for all outputs of the two circuits and then compare the corresponding pointers pairwise. Let C be a combinational circuit. Using the terminology of Chapter 5, assume that we can efficiently compute the automorphism group $Aut A_C$ for the graph A_C . If the group is non-trivial, we need not build BDDs for all the outputs! Consider the orbits of outputs defined by $Aut A_C$. For each orbit, we need construct only one BDD for any one output in that set and simply remember permutations that take one output to another. Intuitively, this would allow for a better variable ordering for that single BDD. We need to explore how to use this idea in conjunction with existing functional verification algorithms.

A New Class of BDDs?

An extension to the above idea is to maintain some functions as traditional BDDs and some others as ‘BDD + permutation’, a compressed representation. The latter one need not be materialized unless required. We can thus define a new class of BDDs. We need to develop theory for defining operations that take such BDDs as inputs and produce them as outputs. We would also need theory for keeping the sets of permutations around. Finally, we need to identify scenarios where this is useful.

Fairness and Partial Orders

We need to incorporate fairness constraints into our framework, as done by Emerson and Sistla [ES95]. Recently, Gyuris and Sistla [GS97] have developed an on-the-fly model checker that utilizes symmetries under fairness. It might also be interesting to study how our framework can be extended to incorporate both partial orders and symmetries, as accomplished by Emerson, Jha and Peled [EJP97].

Symmetries Everywhere

Symmetric systems show up almost everywhere. From high level design to layout, several CAD algorithms might benefit from *a priori* knowledge of symmetries in the circuit under consideration. Pandey and Bryant [PB97] have demonstrated how transistor-level verification of digital circuits can benefit from symmetries. It would be interesting to extend the applicability of symmetries to other domains.

Bibliography

- [AE89] P. C. ATTIE AND E. A. EMERSON. Synthesis of Concurrent Systems with many Similar Sequential Processes. *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 191–201, 1989.
- [Ake78] S. B. AKERS. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-37:509–516, June 1978.
- [AT96] M. AGRAWAL AND T. THIERAUF. The Boolean Isomorphism Problem. In *Proc. Symp. on Foundations of Computer Science*, pages 422–430, Burlington, Vermont, October 1996.
- [BCF⁺95] L. BABAI, G. COOPERMAN, L. FINKELSTEIN, E. LUKS, AND A. SERESS. Fast Monte Carlo Algorithms for Permutation Groups. *J. Comp. Sys. Sci.*, 50(2):296–308, April 1995.
- [BCH⁺91] R. K. BRAYTON, M. CHIDO, R. HOJATI, T. KAM, K. KODANDAPANI, R. P. KURSHAN, S. MALIK, A. L. SANGIOVANNI-VINCENTELLI, E. M. SENTOVICH, T. SHIPLE, K. J. SINGH, AND H.-Y. WANG. BLIF-MV: An Interchange Format for Design Verification and Synthesis. Technical Report UCB/ERL M91/97, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, November 1991.
- [BCL⁺94] J. R. BURCH, E. M. CLARKE, D. E. LONG, K. L. McMILLAN, AND D. L. DILL. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

- [BCMD90] J. R. BURCH, E. M. CLARKE, K. L. McMILLAN, AND D. L. DILL. Sequential Circuit Verification Using Symbolic Model Checking. In *Proc. Design Automation Conf.*, pages 46–51, June 1990.
- [Ber93] VLSI-CAD GROUP BERKELEY. Revisiting BLIF-MV, An Intermediate Format for Verification and Synthesis of Hierarchical Networks of FSMs, 1993.
- [BFJ89] C. A. BROWN, L. FINKELSTEIN, AND P. W. PURDOM JR. A New Base Change Algorithm for Permutation Groups. *SIAM J. Computing*, 18(5):1037–1047, October 1989.
- [BHSV⁺96] R.K. BRAYTON, G.D. HACHTEL, A. SANGIOVANNI-VINCENTELLI, F. SOMENZI, A. AZIZ, S.-T. CHENG, S. EDWARDS, S. KHATRI, Y. KUKIMOTO, A. PARDO, S. QADEER, R.K. RANJAN, S. SARWARY, T.R. SHIPLE, G. SWAMY, AND T. VILLA. VIS: A System for Verification and Synthesis. In *Proc. 8th Intl. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer-Verlag, 1996.
- [BL83] L. BABAI AND E. M. LUKS. Canonical Labeling of Graphs. *Proc. ACM Symp. on the Theory of Computing*, pages 171–183, 1983.
- [BL85] G. BUTLER AND C. W. H. LAM. A General Backtrack Algorithm for the Isomorphism Problem of Combinatorial Objects. *J. Symbolic Computation*, 1:363–381, 1985.
- [BLS] L. BABAI, E. M. LUKS, AND A. SERESS. Fast Management of Permutation Groups II. In Preparation.
- [BLS97] L. BABAI, E. M. LUKS, AND A. SERESS. Fast Management of Permutation Groups I. *SIAM J. Computing*, 26(5):1310–1342, October 1997.
- [Bre76] BRENDON MCKAY. Backtrack Programming and the Graph Isomorphism Problem. Master's thesis, University of Melbourne, 1976.
- [Bry86] R. BRYANT. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.
- [But91] G. BUTLER. *Fundamental Algorithms for Permutation Groups*, volume 559 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

- [Can84] J. J. CANNON. An Introduction to the Group Theory Language Cayley. In M. D. Atkinson, editor, *Computational Group Theory*, pages 145–183. London: Academic Press, 1984.
- [CE81] E. M. CLARKE AND E. A. EMERSON. Design and Synthesis of Synchronization Skeletons Using Branching Time Logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.
- [CEFJ96] E. M. CLARKE, R. ENDERS, T. FILKORN, AND S. JHA. Exploiting Symmetry in Temporal Logic Model Checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.
- [CES86] E. M. CLARKE, E. A. EMERSON, AND A. P. SISTLA. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
- [CF94] G. COOPERMAN AND L. FINKELSTEIN. A Random Base Change Algorithm for Permutation Groups. *J. Symbolic Computation*, 17(6):513–528, June 1994.
- [CGH⁺95] E. M. CLARKE, O. GRUMBERG, H. HIRAISHI, S. JHA, D. E. LONG, K. L. McMILLAN, AND L. A. NESS. Verification of the Futurebus+ Cache Coherence Protocol. *Formal Methods in System Design*, 6(2):217–232, March 1995.
- [CLM89] E. M. CLARKE, D. LONG, AND K. McMILLAN. Compositional Model Checking. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 353–362, 1989.
- [CMB90] O. COUDERT, J. C. MADRE, AND C. BERTHET. Verifying Temporal Properties of Sequential Machines Without Building Their State Diagrams. In E. M. Clarke and R. P. Kurshan, editors, *Proc. Workshop on Computer-Aided Verification*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 23–32. American Mathematical Society, June 1990.
- [CW⁺96] E. M. CLARKE, J. M. WING, ET AL. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–43, December 1996.
- [CYB93] S.-T. CHENG, G. YORK, AND R. K. BRAYTON. VL2MV: A Compiler from Verilog to BLIF-MV, October 1993.

- [DDL77] N. DEO, J. M. DAVIS, AND R. E. LORD. A New Algorithm for Graph Isomorphism. *BIT*, 17:16–30, 1977.
- [EDI97] EDIF Version 3.0.0 IEC Standard 61690-1, 1997. Version 4.0.0 is an EIA/ANSI Standard and being currently reviewed within IEC as IEC 61690-2.
- [EH86] E. A. EMERSON AND J. Y. HALPERN. “Sometimes” and “Not Never” Revisited: on Branching versus Linear Time Temporal Logic. *J. of the ACM*, 33(1):151–178, 1986.
- [EJP97] E. A. EMERSON, S. JHA, AND D. PELED. Combining Partial Order and Symmetry Reductions. In E. Brinksma, editor, *Proc. Third Intl. Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 19–34. Springer-Verlag, April 1997.
- [Eme90] E. A. EMERSON. Temporal and Modal Logic. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier Science, 1990.
- [ES95] E. A. EMERSON AND A. P. SISTLA. Utilizing Symmetry when Model Checking under Fairness Assumptions: An Automata-theoretic Approach. In P. Wolper, editor, *Proc. 7th Intl. Conf. on Computer Aided Verification*, pages 309–324. Springer-Verlag, July 1995.
- [ES96] E. A. EMERSON AND A. P. SISTLA. Symmetry and Model Checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.
- [FHG⁺83] G. FOWLER, R. HARALICK, F. G. GRAY, C. FEUSTEL, AND C. GRINSTEAD. Efficient Graph Automorphism by Vertex Partitioning. *Artificial Intelligence*, 21:245–269, 1983.
- [FHL80] M. L. FURST, J. HOPCROFT, AND E. M. LUKS. Polynomial Time Algorithms for Permutation Groups. *Proc. 21st IEEE Foundations of Computer Science*, pages 36–41, 1980.
- [Gap] GAP: Groups, Algorithms and Programs, Version 3, Release 4. Available via ftp from <ftp.math.rwth-aachen.de>, Lehrstuhl D fuer Mathematik, RWTH Aachen, Germany; directory `/pub/gap`.

- [God96] P. GODEFROID. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
- [GS97] V. GYURIS AND A. P. SISTLA. On-the-Fly Model Checking under Fairness that Exploits Symmetry. In *Proc. 9th Intl. Conf. on Computer Aided Verification, Haifa, Israel, June 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 232–243. Springer-Verlag, 1997.
- [HAM97] W. N. N. HUNG, A. AZIZ, AND K. MCMILLAN. Heuristic Symmetry Reduction for Invariant Checking. In *Workshop Notes of Intl. Workshop on Logic Synthesis*, 1997.
- [Her75] I. N. HERSTEIN. *Topics in Algebra*. Xerox College Pub., Lexington, Massachusetts, 1975.
- [HJJ84] P. HUBER, L. JEPSEN, AND K. JENSEN. Towards Reachability Trees for High-level Petri Nets. In G. Rozenberg, editor, *Advances on Petri Nets*, pages 215–233, 1984.
- [Hof80] C. M. HOFFMAN. On the Complexity of Intersecting Permutation Groups and its Relationship with Graph Isomorphism. Technical Report 4/80, Institut for Informatik und Praktische Mathematik, Christian-Albrechts-Universitat Kiel, 1980.
- [Hof82] C. M. HOFFMAN. *Group-Theoretic Algorithms and Graph Isomorphism*, volume 136. Springer Verlag, 1982.
- [Hof95] R. D. HOF. Intel takes a bullet – and barely breaks stride. *Business Week*, pages 38–39, January 1995.
- [HU79] J. E. HOPCROFT AND J. D. ULLMAN. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- [HW74] J. E. HOPCROFT AND J. K. WONG. Linear Time Algorithm for Isomorphism of Planar Graphs. In *Proc. ACM Symp. on the Theory of Computing*, pages 172–184, 1974.

- [ID93] C. N. IP AND D. L. DILL. Better Verification through Symmetry. In *11th IFIP WG10.2 Intl. Conf. on Computer Hardware Description Languages and their Applications*, pages 97–111, April 1993.
- [ID96] C. N. IP AND D. L. DILL. Better Verification Through Symmetry. *Formal Methods in System Design*, 9(1/2):41–76, 1996.
- [Ip96] NORRIS IP. *State Reduction Methods for Automatic Formal Verification*. PhD thesis, Stanford University, December 1996.
- [Jer86] M. JERRUM. A Compact Representation for Permutation Groups. *J. Algorithms*, 27:60–78, 1986.
- [Knu91] D. E. KNUTH. Efficient Representation of Permutation Groups. In *Combinatorica*, pages 33–44, 1991.
- [Kur87] R. P. KURSHAN. Testing Containment of Omega-regular Languages. In *Reducibility in Analysis of Coordination*, volume 103 of *Lecture Notes in Computer Science*, pages 19–39. Springer-Verlag, 1987.
- [LB79] G. S. LUEKER AND K. S. BOOTH. A Linear Time Algorithm for Deciding Interval Graph Isomorphism. *J. of the ACM*, 26:183–195, 1979.
- [Leo84] J. S. LEON. Computing Automorphism Groups of Combinatorial Objects. *Computational Group Theory*, pages 321–335, 1984.
- [Leo91] J. S. LEON. Permutation Group Algorithms based on Partitions. I. Theory and Algorithms. *J. Symbolic Computation*, 12(4-5):533–583, 1991.
- [LN91] B. LIN AND A. R. NEWTON. Efficient Symbolic Manipulation of Equivalence Relations and Classes. In *Proc. 1991 Intl. Workshop of Formal Methods in VLSI Design*, January 1991.
- [Lon93] D. LONG. *Model Checking, Abstraction and Compositional Verification*. PhD thesis, Carnegie Mellon University, 1993.
- [Luk82] E. M. LUKS. Isomorphism of Graphs of Bounded Valence can be Tested in Polynomial Time. *J. Comp. Sys. Sci.*, 25(1):42–65, August 1982.

- [McK77] B. D. MCKAY. Computing Automorphisms and Canonical Labelings of Graphs. In *Intl. Conf. on Combinatorial Mathematics, Canberra*, volume 686 of *Lecture Notes in Computer Science*, pages 223–232. Springer Verlag, 1977.
- [McK81] B. D. MCKAY. Practical Graph Isomorphism. In *Proc. Tenth Manitoba Conf. on Numerical Math. and Computing, Winnipeg, 1980, vol 1*, volume 30 of *Congr. Numer.*, pages 45–87, 1981.
- [McK90] B. D. MCKAY. Nauty Users Guide (Version 1.5). Technical Report TR-CS-90-02, Computer Science Department, Australian National University, Australia, 1990.
- [Mit88] H. B. MITTAL. A Fast Backtrack Algorithm for Graph Isomorphism. *Inf. Proc. Let.*, 29:105–110, 1988.
- [PB97] M. PANDEY AND E. BRYANT. Exploiting Symmetry when Verifying Transistor-Level Circuits by Symbolic Trajectory Evaluation. In *Proc. 9th Intl. Conf. on Computer Aided Verification, Haifa, Israel, June 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 244–255. Springer-Verlag, 1997.
- [PD97] F. PONG AND M. DUBOIS. Verification Techniques for Cache Coherence Protocols. *ACM Computing Surveys*, 29(1):82–126, March 1997.
- [Pel93] D.A. PELED. All from One, One for All: On Model Checking Using Representatives. In *Proc. 5th Intl. Conf. on Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1993.
- [SD76] D. C. SCHMIDT AND L. E. DRUFFEL. A Fast Backtrack Algorithm to Test Directed Graphs for Isomorphism using Distance Matrices. *J. of the ACM*, 23:433–445, 1976.
- [Sei95] R. SEIDEL. On the All-Pairs-Shortest-Path Problem in Unweighted Undirected Graphs. *J. Comp. Sys. Sci.*, 51(3):400–403, May 1995.
- [Sim70] C. C. SIMS. Computational Methods in the Study of Permutation Groups. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 169–183. Pergamon, Elmsford, N.Y., 1970.

- [Sim71] C. C. SIMS. Computation with Permutation Groups. In S. R. Petrick, editor, *Proc. Second Symp. on Symbolic and Algebraic Manipulations*, pages 23–28. Association for Computing Machinery, New York, 1971.
- [Soi90] L. H. SOICHER. GRAPE: A System for Computing with Graphs and Groups. In L. Finkelstein and W. M. Kantor, editors, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 11, pages 287–291, 1990.
- [Sta91] P. STARKE. Reachability Analysis of Petri Nets using Symmetries. *Syst. Anal. Model. Simul.*, 8(4/5):293–303, 1991.
- [TSL+90] H. TOUATI, H. SAVOJ, B. LIN, R. K. BRAYTON, AND A. L. SANGIOVANNI-VINCENTELLI. Implicit State Enumeration of Finite State Machines using BDD's. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 130–133, Santa Clara, CA, November 1990.
- [Ver97] Verilog: IEEE 1364 Standard, 1997.
- [VHD93] VHDL: IEEE 1076 Standard, 1993.
- [vL90] J. VAN LEEUWEN. Graph Algorithms. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, pages 525–631. Elsevier Science, 1990.
- [VW86] M. Y. VARDI AND P. L. WOLPER. An Automata-Theoretic Approach to Program Verification. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 332–334, 1986.
- [W. 96] W. BOSMA AND J. J. CANNON AND C. PLAYOUST. The Magma Algebra System I: The User Language, 1996. Submitted to *J. Symbolic Computation*.
- [Wie64] H. WIELANDT. *Finite Permutation Groups*. Academic Press, New York, 1964.