Symmetry and Induction in Model Checking

Somesh Jha

October 1996

CMU-CS-96-202

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:
Edmund M. Clarke
Stephen Brookes
Daniel Jackson
Robert Kurshan, Bell Labs, Lucent

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597, and in part by This research was sponsored in part by the National Science Foundation under grant no. CCR-8722633, by the Semiconductor Research Corporation under contract 92-DJ-294..

1

Keywords: formal verification, temporal logic, CTL, model checking, μ calculus, symmetry, induction, partial order.

Abstract

With the increasing complexity of digital systems, testing of digital systems is becoming increasingly important. Perhaps, the most popular method for testing hardware is simulation. The incompleteness of simulation based testing methods has spurred the recent surge in the research on formal verification. In formal verification, one builds a precise model of the hardware under scrutiny and proves that the model satisfies a specification of interest. For example, suppose one wants to verify that a router chip does not deadlock. In this case the user will build a precise model of the router and the specification will express the property of deadlock freedom. The two approaches to formal verification are *model checking* and *theorem proving*. In this thesis we will only discuss model checking. Most model checking procedures suffer from the *state explosion problem*, i.e., the size of the state space of the system can be exponential in the number of state variables of the system. For certain systems, exploiting the inherent symmetry can alleviate the state-explosion problem. We discuss Model Checking procedures which exploit symmetry. Current model checkers can only verify a single state-transition system at a time. We also want to extend the model checking techniques to handle infinite families of finite-state systems.

In practice, finite state concurrent systems often exhibit considerable symmetry. We investigate techniques for reducing the complexity of temporal logic model checking in the presence of symmetry. In particular, we show that symmetry can frequently be used to reduce the size of the state space that must be explored during model checking. We also investigate complexity of various problems related to exploiting symmetry in model checking. Partial order based reduction techniques exploit independence of actions. We demonstrate that partial order and symmetry based reduction techniques can be applied simultaneously.

The ability to reason automatically about entire families of similar state-transition systems is an important research goal. Such families arise frequently in the design of reactive systems in both hardware and software. The infinite family of token rings is a simple example. More complicated examples are trees of processes consisting of one root, several internal and leaf nodes, and hierarchical buses with different numbers of processors and caches. A technique for verifying entire families of state-transition systems based on network grammars and process invariants is presented here.

Acknowledgements

My thesis advisor, Ed Clarke, deserves a significant portion of the credit for the work in this thesis. At times when I was at loss in regards to my next project, he was able to suggest several important problems. Most of the ideas in this thesis are a result of his suggestions. He was also very sympathetic to my involvement in problems not directly related to hardware verification.

Allen Emerson, Thomas Filkorn, Orna Grumberg, David Long, Will Marrero, Doron Peled and Prasad Sistla have all contributed to the ideas in this thesis. All of them have been coauthors on papers published on the topics related to this thesis. They also taught me a lot about the whole research process. I was a summer intern for Bellcore in 1991. During my stay at Bellcore my discussions with Linda Ness were very fruitful. I have worked on developing efficient algorithms for checking Software Specifications with Daniel Jackson and Craig Damon. I have benifited a great deal from this collaboration. Steve Brookes and Bob Kurshan have been very receptive to my ideas. I thank them for their encouragement. Steve Weeks and Greg Zelesnik read parts of my thesis and provided helpful comments.

For most of my stay at CMU, I shared my office with Sue Older and Xudong Zhao I had some very interesting, enlightening, and animated discussions with both of them. I have no doubt in mind that Sue and Xudong will be very successful in their respective carreers. And guys, thanks for putting up with my son when his school was off. Other members of the hardware verification group, Sergio Campos, Marius Minea, Sergey Berezin, and Vicki Hartonas have always been very helpful.

Thanks to my parents for their constant love and encouragement. Finally, to Sarita – I could never have done it with you. Akshaya, my son, has been a constant source of joy for me.

Contents

Chapter 1

Introduction

The use of digital systems, especially microprocessors, is becoming very wide spread. As the number of users of microprocessors increases, the impact of an error in a microprocessor can have a dramatic effect. Perhaps, the most glaring example of this is the Intel Pentium Bug. In light of this, ascertaining the correctness of a digital system before fabrication is becoming crucial. By far, the most popular method for checking the correctness of hardware is simulation. The major drawback of simulation is that it is not complete, i.e., using simulation one cannot be certain that a digital system has a required property. On the other hand formal verification techniques are able to verify that a digital system has the required property. The main disadvantage of formal verification techniques is that it requires more memory than simulation. However, due to some recent breakthroughs, formal verification is becoming a feasible alternative to simulation. Currently, there are two approaches in formal verification:

- **Theorem Proving:** In this case the system to be verified is described by a set of axioms. In order to prove that the system has the required property or specification, one proves using the axioms describing the system, that the required property is a theorem.

- **State Exploration Techniques:** In this case the system is given as a labeled directed graph. In order to ascertain that the system has a particular specification, one explores the labeled directed graph corresponding to the system. Model Checking belongs to this class.

In this thesis we will only consider Model Checking. Some advantages of Model Checking over theorem proving approaches are:

1. In model checking, the proof that the system has a particular property is done automatically. This has an advantage over most theorem provers which require manual assistance. Therefore, model checking tools are more suitable for industry.

2. If a system does not have a desired property, model checkers can provide a counter example. These counter examples are very useful to the designer. It is very hard to provide counter examples in theorem provers.

3. Model Checkers have no difficulty in handling partial specifications. On the other hand, theorem provers frequently require a complete set of axioms describing the system.

Next, we describe Model Checking and Temporal Logic in more detail. In the past, temporal logic was used by philosophers to reason about time [15]. The use of temporal logic to reason about properties of concurrent systems was first described in [62]. *Temporal Logic Model Checking* is a technique for determining whether a temporal logic formula is valid in a finite state system $M = (S, R, L)$, where S is the state space, R is the state transition relation, and L is a function that labels states with sets of atomic propositions. The Model Checking problem can be stated as

> `Given a Kripke structure` $M = (S, R, L)$`, a state` $s \in S$`, and a temporal formula, determine whether` $M, s \models f$`.`

An efficient procedure for model checking where the specification f is given in the computation tree logic *CTL* was given in [17]. Such a structure is usually called a *Kripke structure* and may have an enormous number of states because of the *state explosion* problem. Formally, the *state explosion problem* can be stated as follows:

> `A concurrent system which is a composition of` n `processes, with each process having` k `states, can have` k^n `total states.`

A schematic diagram for model checking is given in Figure 1.1. An efficient Model Checking procedure tries to reduce the number of states that are actually explored. Most research in temporal logic model checking is focussed on efficient data structures to represent state space. Perhaps the biggest pragmatic breakthrough in the area of model checking was the use of Ordered Binary Decision Diagrams (OBDDs) to represent the set of states during model checking [11, 13]. When OBDDs are used to represent state sets in model checking, it is called *symbolic model checking*. The distinction between model checking and symbolic model checking is pointed out in Figure 1.1. The use of model checking made it possible to find errors in nontrivial circuits which had been carefully designed [8, 27]. With the discovery of symbolic model checking, it is now possible to verify large systems [18, 56].

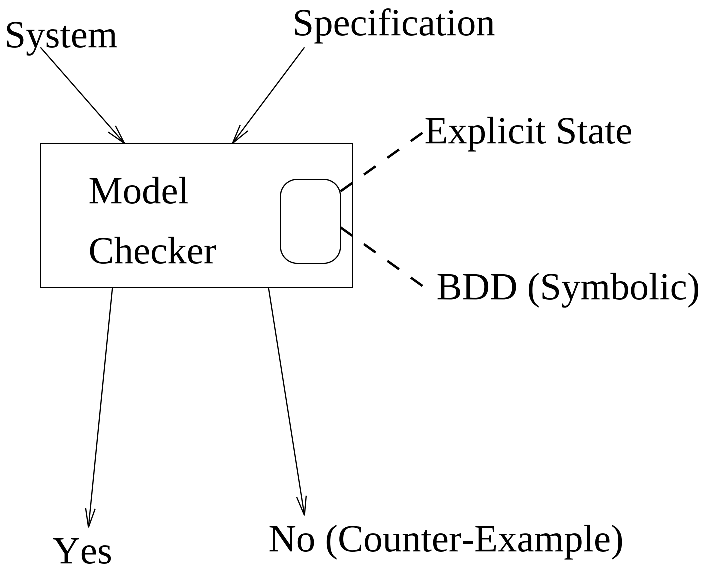

Figure 1.1: Schematic Diagram of Model Checking

1.1 Scope of the Thesis

The first part of this thesis describes techniques for exploiting symmetry to alleviate the state explosion problem. Finite state concurrent systems frequently exhibit considerable symmetry. It is possible to find symmetry in memories, caches, register files, bus protocols, network protocols – anything that has a lot of replicated structure. For example, a ring of processes exhibits rotational symmetry. This fact can be used to obtain an equivalent reduced model of the system. Given a Kripke Structure $M = (S, R, L)$, a symmetry group G is a group acting on the state set S that *preserves* the transition relation R. A symmetry group G acting on the state set S partitions the state set S into equivalence classes called *orbits*. A quotient model M_G is constructed that contains one representative from each orbit. The state space S_G of the quotient model will, in general, be much smaller than the the original state space S. This makes it possible to verify much larger structures.

We also investigate the complexity of exploiting symmetry in model checking algorithms. The first problem that we consider is computing the *orbit relation*, i.e., determining whether two states are in the same orbit or not. This is an important problem because the direct method of computing the quotient model M_G uses this relation. We also obtain lower bounds on the size of the OBDDs needed to encode the orbit relation.

Partial Order based reduction techniques exploit the independence of actions to tackle the state-explosion problem. The independence relation on actions induces an equivalence class on traces. Partial Order based reduction techniques explore only few traces from each equivalence class. This thesis presents a method to combine partial order and symmetry based reductions.

Most of the research done in the area of model checking focuses on verifying single finite-state systems. Typically, circuit and protocol designs are parametrized, i.e., define an infinite family of systems. For example, a circuit design to multiply two integers has the width of the integers n as a parameter. This thesis investigates methods to verify such parameterized designs. The problem of verifying parametrized designs can also be thought of as solving the *state explosion* problem because in this case the state set is unbounded. Formally, the problem of verifying parameterized designs can be stated as:

> `Given an infinite family` $F = \{P_i\}_{i=1}^{\infty}$ `of finite-state systems and a temporal specification` f`, determine whether` $P_i \models f$ `for all` i.

In general the problem is undecidable [4]. However, for *specific* families the problem can be solved. Most techniques are based on finding *network invariants* [47, 74]. Given an infinite family $F = \{P_i\}_{i=1}^{\infty}$ and a reflexive, transitive relation $\preceq$, an invariant I is a process such that $P_i \preceq I$ for all i. The relation $\preceq$ should preserve the property f we are interested in, i.e., if I satisfies f, then P_i should also satisfy f. Once the invariant I is found, traditional model checking techniques can be used to check that I satisfies f.

Main contributions of this thesis are:

1. A technique for exploiting symmetry during model checking. Additional complications arise in using symmetry in conjunction with OBDDs during model checking. This thesis provides a method of using OBDDs in combination with symmetry reductions.

2. We also investigate the complexity of various problems related to the use of symmetry with model checking. We also provide ways of deriving symmetries of finite-state systems.

3. Partial order methods exploit the independence of actions. We discuss techniques to combine partial order and symmetry reduction techniques.

4. We present a formalism based on network grammars to describe parameterized designs. We also present a logic based on regular expressions which can be interpreted over these parameterized designs. We provide a methodology for constructing invariants based on a technique called unfolding.

Chapter 2

Model Checking

The propositional μ-calculus is a powerful language for expressing properties of transition systems by using least and greatest fixpoint operators. Recently, the μ-calculus has generated much interest among researchers in computer-aided verification. This interest stems from the fact that many temporal and program logics can be encoded into the μ-calculus, and that finite-state verification procedures for the μ-calculus can be succinctly described. In addition, the wide-spread use of binary decision diagrams has made fixpoint based algorithms even more important, since methods that require the manipulation of individual states do not take advantage of this representation.

Several versions of the propositional μ-calculus have been described in the literature, and the ideas in this chapter will work with any of them. For the sake of concreteness, we will use the propositional μ-calculus of Kozen [44]. Closed formulas in this logic evaluate to sets of states. A considerable amount of research has focused on finding techniques for evaluating such formulas efficiently, and many algorithms have been proposed for this purpose. These algorithms generally fall into two categories, local and global.

Local procedures are designed for proving that a specific state of the transition system satisfies the given formula. Because of this, it is not always necessary to examine all the states in the transition system. However, the worst-case complexity of these approaches is generally larger than the complexity of the global methods. Tableau-based local approaches have been developed by Cleaveland [21], Stirling and Walker [67], and Winskel [73]. More recently, Andersen [2] and Larsen [48] have developed efficient local methods for a subset of the μ-calculus. Mader [52] has also proposed improvements to the tableau-based method of Stirling and Walker that seem to increase its efficiency.

In this chapter, we restrict ourselves to global model checking procedures. Global procedures generally work bottom-up through the formula, evaluating each subformula based on the values of its subformulas. Iteration is used to compute the fixpoints. Because of fixpoint nesting, a naive global algorithm may require about $O(n^k)$ iterations to evaluate a formula, where n is the number of states in the transition system and k is the depth of nesting of the fixpoints. Emerson and Lei [28] improved on this by observing that successively nested fixpoints of the same type do not increase the complexity of the computation. They formalize this observation using the notion of *alternation depth* and give an algorithm requiring only about $O(n^d)$ iterations, where d is the alternation depth. In an implementation, bookkeeping and set manipulations may add another factor of n or so to the time required. Subsequent work by Cleaveland, Klein, Steffen, and Andersen [2, 22, 23] has reduced this extra complexity, but the overall number of iterations has remained about $O(n^d)$. In [50] the authors have improved on this by giving an algorithm that uses only $O(n^{d/2})$ iterations to compute a formula with alternation depth d, thus requiring only about the square root of the time needed by earlier algorithms.

This chapter describes the propositional μ-calculus and general algorithms for evaluating μ-calculus formulas. Examples of verification problems that can be encoded within the language of the μ-calculus are also provided. The remainder of this chapter is organized as follows. A formal syntax and semantics for the propositional μ-calculus is given in Section 2.1. Section 2.2 discusses different algorithms for evaluation μ-calculus formulas and their complexities. A brief description of Ordered Binary Decision Diagrams (OBDDs) is given in Section 4. Section 5 presents the algorithm for encoding μ-calculus formulas with OBDDs. The syntax and semantics for CTL and for CTL with fairness constraints is given in Section 6, while a translation of these logics into the μ-calculus is given in Section 7. Definitions for different kinds of simulation preorders and bisimulation equivalences are given in Section 8 along with encodings for these relations in the μ-calculus.

2.1 The Propositional μ-Calculus

In the propositional μ-calculus, formulas are constructed as follows:

- atomic propositions $AP = \{p, p_1, p_2, \ldots\}$
- atomic propositional variables $VAR = \{R,\ R_1,\ R_2,\ \ldots\}$
- logical connectives $\neg\cdot$, $\cdot \wedge \cdot$ and $\cdot \vee \cdot$
- modal operators $\langle a \rangle\cdot$ and $[a]\cdot$, where a is an action in the set $Act = \{a, b, a_1, a_2, \ldots\}$
- fixpoint operators $\mu R_i.(\cdots)$ and $\nu R_i.(\cdots)$. Propositional variables bound by the fixpoint operators must be in the scope of the even number of negations.

There is a standard notion of free and bound variables (by fixpoint operators) in the formulas. Closed formulas are the formulas without free variables. Formulas in this calculus are interpreted relative to a transition system $M = (\top, T, L)$ that consists of:

- a nonempty set of states $\top$
- a mapping $L : AP \rightarrow 2^{\top}$ that takes each atomic proposition to some subset of $\top$ (the states where the proposition is true)
- a mapping $T : Act \rightarrow 2^{\top\times\top}$ that takes each action to a binary relation over $\top$ (the state changes that can result from making an action)

The intuitive meaning of the formula $\langle a \rangle\phi$ is "it is possible to make an a-action and transition to a state where ϕ holds". $[\cdot]$ is the dual of $\langle\cdot\rangle$; for $[a]\phi$, the intended meaning is that "ϕ holds in all states reachable (in one step) by making an a-action." The μ and ν operators are used to express least and greatest fixpoints, respectively. To emphasize the duality between least and greatest fixpoints, we write the empty set of states as $\perp$. Also, in the rest of this chapter, we will use the more intuitive notation $s \xrightarrow{a} s'$ to mean $(s, s') \in T(a)$.

Formally, a formula ϕ is interpreted as a set of states in which ϕ is true. We write such set of states as $[\![\phi]\!]_M \, e$, where M is a transition system and $e : VAR \rightarrow 2^{\top}$ is an environment. We denote by $e\,[R \leftarrow S]$ a new environment which is the same as e except that $e\,[R \leftarrow S]\,(R) = S$. The set $[\![\phi]\!]_M \, e$ is defined recursively as follows.

- $[\![p]\!]_M \, e = L(p)$
- $[\![R]\!]_M \, e = e(R)$
- $[\![\neg\phi]\!]_M \, e = \top - [\![\phi]\!]_M \, e$
- $[\![\phi \wedge \psi]\!]_M \, e = [\![\phi]\!]_M \, e \cap [\![\psi]\!]_M \, e$
- $[\![\phi \vee \psi]\!]_M \, e = [\![\phi]\!]_M \, e \cup [\![\psi]\!]_M \, e$
- $[\![\langle a \rangle \phi]\!]_M \, e = \{\, s \mid \exists t \, [s \xrightarrow{a} t \text{ and } t \in [\![\phi]\!]_M \, e] \,\}$
  $[\![[a]\,\phi]\!]_M \, e = \{\, s \mid \forall t \, [s \xrightarrow{a} t \text{ implies } t \in [\![\phi]\!]_M \, e] \,\}$
- $[\![\mu R.\phi]\!]_M \, e$ is the least fixpoint of the predicate transformer $\tau \colon 2^{\top} \rightarrow 2^{\top}$ defined by:
  $$\tau(S) = [\![\phi]\!]_M \, e\,[R \leftarrow S]$$
- The interpretation of $\nu R.\phi$ is similar, except that we take the greatest fixpoint.

Within formulas, the negation is restricted in use, and so the fixpoints are guaranteed to be well-defined. Formally, every logical connective except negation is monotonic ($\phi \rightarrow \phi'$ implies $\phi \wedge \psi \rightarrow \phi' \wedge \psi$, $\phi \vee \psi \rightarrow \phi' \vee \psi$, $\langle a \rangle \phi \rightarrow \langle a \rangle \phi'$, and $[a]\phi \rightarrow [a]\phi'$), and all the negations can be pushed down to the atomic propositions using De Morgan's laws and dualities ($\neg[a]\phi \equiv \langle a \rangle \neg\phi$, $\neg\langle a \rangle \phi \equiv [a]\neg\phi$, $\neg\mu R.\phi(R) \equiv \nu R.\neg\phi(\neg R)$, $\neg\nu R.\phi(R) \equiv \mu R.\neg\phi(\neg R)$). Since bound variables are under even number of negations, they will be negation free after this process. Thus, each possible formula in a fixpoint operator is monotonic and hence each possible τ is also monotonic ($S \subseteq S'$ implies $\tau(S) \subseteq \tau(S')$). This is enough to ensure the existence of the fixpoints [68]. Furthermore, since we will be evaluating formulas over finite transition systems, monotonicity of τ implies that τ is also $\cup$-continuous and $\cap$-continuous, and hence the least and greatest fixpoints can be computed by iterative evaluation:

$$[\![\mu R.\phi]\!]_M\, e = \bigcup_i \tau^i(\bot) \qquad [\![\nu R.\phi]\!]_M\, e = \bigcap_i \tau^i(\top).$$

($\tau^i(S)$ can be defined recursively as $\tau^0(S) = S$ and $\tau^{i+1}(S) = \tau(\tau^i(S))$.) Since the domain $\top$ is finite, the iteration must stop after a finite number of steps. More precisely, for some $i \leq |\top|$, the fixpoint is equal to $\tau^i(\bot)$ (for a least fixpoint) or $\tau^i(\top)$ (for a greatest fixpoint). To find the fixpoint, we repeatedly apply τ starting from $\bot$ or from $\top$ until the result does not change.

The *alternation depth* of a formula is intuitively equal to the number of alternations in the nesting of least and greatest fixpoints, when all negations are applied only to propositions. There are other more elaborate definitions of alternation depth [2, 3, 22], that take into account the possibility that nested fixpoints may still be independent. Such fixpoints do not depend on the value of approximations to outer fixpoints. Consequently, they only need to be evaluated once. This type of nesting does not increase the effective alternation depth. However, to simplify our presentation we will use the definition of alternation depth given by Emerson and Lei [28]. Formally, the alternation depth is defined as follows:

Definition 2.1.1

- The alternation depth of an atomic proposition or a propositional variable is 0;
- The alternation depth for formulas like $\phi \wedge \psi$, $\phi \vee \psi$, $\langle a \rangle \phi$, etc., is the maximum alternation depth of the subformulas ϕ and ψ.
- The alternation depth of $\mu R.\phi$ is the maximum of: one, the alternation depth of ϕ, and one plus the alternation depth of any top-level ν-subformulas of ϕ. A top-level ν-subformula of ϕ is a subformula $\nu R'.\psi$ of ϕ that is not contained within any other fixpoint subformula of ϕ. The *alternation depth* of $\nu R.\phi$ is similarly defined.

Example 2.1.1 Consider the following formula which will be discussed in Section 7.

$$\nu Y.(P \wedge \langle a \rangle [\mu X.(P \wedge \langle a \rangle X) \vee (h \wedge Y)])$$

This formula expresses the property "P holds continuously along some fair a-path" and has an alternation depth of two.

Because of the duality,

$$\nu R.\phi(\cdots, R, \cdots) \quad = \quad \neg \mu R. \neg \phi(\cdots, \neg R, \cdots)$$

we could have defined the propositional μ-calculus with just the least fixpoint operator and negation. In order to give a succinct description of certain constructions we sometimes use the dual formulation. However, the concept of alternation depth is easier to define using the formulation given earlier.

2.2 Evaluating Fixpoint Formulas

We define *model checking* as a technique of verifying a model relative to its specification in the μ-calculus. This is the same as evaluating a formula in a model, i.e., finding the set of states of the model where the formula is true. Figure 2.1 presents the naive, straightforward, recursive algorithm for evaluating μ-calculus formulas. The time complexity of the algorithm in Figure 2.1 is exponential in the length of the formula. To see this, we analyze the behavior of the algorithm when computing nested fixpoints. The algorithm computes fixpoints by iteratively computing approximations. These successive approximations form a chain ordered by inclusion. Since the number of strict inclusions in such a chain is limited by the number of possible states, we have that the loop will execute at most $n+1$ times, where $n = |\top|$. Each iteration of the loop involves a recursive call to evaluate the body of the fixpoint with a different value for the fixpoint variable. If in turn, the subformula being evaluated contains a fixpoint, the evaluation of its body will also involve a loop containing up to $n+1$ recursive calls. Thus, the total number of recursive calls will be $O(n^2)$. In general, the

function eval(ϕ, e)
if $\phi = p$ then return $L(p)$
if $\phi = R$ then return $e(R)$
if $\phi = \psi_1 \wedge \psi_2$ then
    return eval$(\psi_1, e) \cap$ eval(ψ_2, e)
if $\phi = \psi_1 \vee \psi_2$ then
    return eval$(\psi_1, e) \cup$ eval(ψ_2, e)
if $\phi = \langle a \rangle \psi$ then
    return $\{ s \mid \exists t\, [s \xrightarrow{a} t$ and $t \in$ eval$(\psi, e)] \}$
if $\phi = [a]\psi$ then
    return $\{ s \mid \forall t\, [s \xrightarrow{a} t$ implies $t \in$ eval$(\psi, e)] \}$

if $\phi = \mu R.\psi(R)$ then
    $R_{\text{val}} := \perp$
    repeat
        $R_{\text{old}} := R_{\text{val}}$
        $R_{\text{val}} :=$ eval$(\psi, e\,[R \leftarrow R_{\text{val}}])$
    until $R_{\text{val}} = R_{\text{old}}$
    return R_{val}

if $\phi = \nu R.\psi(R)$ then
    $R_{\text{val}} := \top$
    repeat
        $R_{\text{old}} := R_{\text{val}}$
        $R_{\text{val}} :=$ eval$(\psi, e\,[R \leftarrow R_{\text{val}}])$
    until $R_{\text{val}} = R_{\text{old}}$
    return R_{val}

Figure 2.1: Pseudocode for the naive algorithm

body of the innermost fixpoint will be evaluated $O(n^k)$ times where k is the maximum nesting depth of fixpoint operators in the formula.

Note that we have only considered the number of iterations required when evaluating fixpoints and not the number of steps required to evaluate a μ-calculus formula. While each fixpoint may only take $O(|\top|)$ iterations, each individual iteration can take up to $O(|M||\phi|)$ steps, where $M = (\top, T, L)$ is the model and $|M| = |\top| + \sum_{a \in Act} |T(a)|$. In general, then, this algorithm has time complexity $O[(|M||\phi|)n^k]$.

A result by Emerson and Lei demonstrates that the value of a fixpoint formula can be computed with $O((|\phi|n)^d)$ iterations, where d is the alternation depth of ϕ. Their algorithm is similar to the straightforward one described above, except when a fixpoint is nested directly within the scope of another fixpoint of the same type. In this case, the fixpoints are computed slightly differently.

A simple example will suffice to demonstrate the idea. When discussing the evaluation of fixpoint formulas, we will use $R_1, \ldots, R_k$ as the fixpoint variables, with R_1 being the outermost fixpoint variable and R_k being the innermost. We will use the notation $R_j^{i_1 \cdots i_j}$ to denote the value of the i_j-th approximation for R_j after having computed the i_l-th approximation for R_l for $1 \leq l < j$. We use $i_j = \omega$ to indicate that we are considering the final approximation (the actual fixpoint value) for R_j. For example, R_1^ω is the value of the fixpoint for R_1 and R_2^{30} is the initial approximation for R_2 after having computed the third approximation for R_1. Consider the formula

$$\mu R_1.\psi_1(R_1, \mu R_2.\psi_2(R_1, R_2)).$$

The subformula $\mu R_2.\psi_2(R_1, R_2)$ defines a monotonic predicate transformer τ taking one set (the value of R_1) to another (the value of the least fixpoint of R_2). When evaluating the outer fixpoint, we start with the initial approximation $R_1^0 = \bot$ and then compute $\tau(R_1^0)$. This is done by iteratively computing approximations for the inner fixpoint also starting from $R_2^{00} = \bot$ until we reach a fixpoint $R_2^{0\omega}$. Now R_1 is increased to R_1^1, the result of evaluating $\psi_1(R_1^0, R_2^{0\omega})$. We next compute the least fixpoint $\tau(R_1^1)$. Since $R_1^0 \subseteq R_1^1$, by monotonicity we know that $\tau(R_1^0) \subseteq \tau(R_1^1)$. Now note that the following lemma holds:

Lemma 2.2.1 If $S \subseteq \bigcup_i \tau^i(\bot)$ then $\bigcup_i \tau^i(S) = \bigcup_i \tau^i(\bot)$.

In other words, to compute a least fixpoint, it is enough to start iterating with any approximation known to be below the fixpoint. Thus , we can start iterating with $R_2^{10} = R_2^{0\omega} = \tau(R_1^0)$ instead of $R_2^{10} = \bot$. When we compute the fixpoint $R_2^{1\omega}$, we next compute the new approximation to R_1, which is R_1^2, the result of evaluating $\psi_1(R_1^1, R_2^{1\omega})$. Again, we know that $R_1^1 \subseteq R_1^2$ which implies that $\tau(R_1^1) \subseteq \tau(R_1^2)$. But $\tau(R_1^1) = R_2^{1\omega}$, the value of the last inner fixpoint computed, and $\tau(R_1^2) = R_2^{2\omega}$ the fixpoint to be computed next. Again, we can start iterating with any approximation below the fixpoint. So to compute $R_2^{2\omega}$ we begin with $R_2^{20} = R_2^{1\omega} = \tau(R_1^1)$. In general, when computing $R_2^{i\omega}$ we always begin with $R_2^{i0} = R_2^{(i-1)\omega}$. Since we never restart the inner fixpoint computation, we can have at most n increases in the value of the inner fixpoint variable. Overall, we only need $O(n)$ iterations to evaluate this expression, instead of $O(n^2)$. In general, this type of simplification leads to an algorithm that computes fixpoint formulas in time exponential in the alternation depth of the formula since we only reset an inner fixpoint computation when there is an alternation in fixpoints in the formula.

Thus, this algorithm for evaluating μ-calculus formulas is identical to the naive algorithm except in the case when the main connective is a fixpoint operator. The pseudocode for this part of the algorithm is given in Figure 2.2. Note that unlike the naive algorithm, the approximation values $A[i]$ are not reset when evaluating the subformula $\mu R_i.\psi(R_i)$ ($\nu R_i.\psi(R_i)$). Instead, we reset all top-level greatest (least) fixpoint variables contained in ψ. By the top-level fixpoints in a formula we mean all the fixpoints of the same type (μ or ν) that are not in the scope of the other type of fixpoints. This guarantees that when we evaluate a top-level fixpoint subformula of the same type, we do not start the computation from $\bot$ or $\top$, but from the previously computed value as in our example.

In [50] the authors observe that by storing even more intermediate values, the time complexity for evaluating fixpoint formulas can be reduced to $O(n^{\lfloor d/2 \rfloor + 1})$ where again d is the alternation depth of the formula. To simplify our discussion, we consider formulas with strict alternation of fixpoints. We present a small example to illustrate the idea behind this algorithm.

Consider the formula:

$$\mu R_1.\psi_1(R_1, \nu R_2.\psi_2(R_1, R_2, \mu R_3.\psi_3(R_1, R_2, R_3))).$$

To compute the outer fixpoint, we start with $R_1 = \bot$, $R_2 = \top$ and $R_3 = \bot$. As in the previous case, we denote these values by R_1^0, R_2^{00}, and R_3^{000} respectively. The superscript on R_k gives the iteration indices function eval(ϕ, e)

N := The number of fixpoint operators in ϕ
for $i := 1$ to N do $A[i]$:= if the i-th fixpoint of ϕ is μ then $\perp$ else $\top$
return evalrec(ϕ, e)

Where *evalrec* is defined recursively as

function evalrec(ϕ, e)

if $\phi = p$ then return $L(p)$
if $\phi = R$ then return $e(R)$
if $\phi = \psi_1 \wedge \psi_2$ then
    return evalrec(ψ_1, e)$\cap$ evalrec(ψ_2, e)
if $\phi = \psi_1 \vee \psi_2$ then
    return evalrec(ψ_1, e)$\cup$ evalrec(ψ_2, e)
if $\phi = \langle a \rangle \psi$ then
    return $\{ s \mid \exists t\, [s \xrightarrow{a} t$ and $t \in$ evalrec(ψ, e)$] \}$
if $\phi = [a]\psi$ then
    return $\{ s \mid \forall t\, [s \xrightarrow{a} t$ implies $t \in$ evalrec(ψ, e)$] \}$

if $\phi = \mu R_i.\psi(R_i)$ then
    For all top-level greatest fixpoint subformulas $\nu R_j.\psi'(R_j)$ of ψ
        do $A[j] := \top$
    repeat
        $R_{\text{old}} := A[i]$
        $A[i] :=$ evalrec($\psi, e\,[R_i \leftarrow A[i]]$)
    until $A[i] = R_{\text{old}}$
    return $A[i]$

if $\phi = \nu R_i.\psi(R_i)$ then
    For all top-level least fixpoint subformulas $\mu R_j.\psi'(R_j)$ of ψ
        do $A[j] := \perp$
    repeat
        $R_{\text{old}} := A[i]$
        $A[i] :=$ evalrec($\psi, e\,[R_i \leftarrow A[i]]$)
    until $A[i] = R_{\text{old}}$
    return $A[i]$

Figure 2.2: Pseudocode for the Emerson and Lei algorithm

for the fixpoints involving $R_1, \ldots R_k$. We then iterate to compute the inner fixpoint; call the value of this fixpoint $R_3^{00\omega}$. We now compute the next approximation R_2^{01} for R_2 by evaluating $\psi_2(R_1^0, R_2^{00}, R_3^{00\omega})$ and go back to the inner fixpoint. Eventually, we reach the fixpoint for R_2, having computed R_2^{00}, $R_3^{00\omega}$, R_2^{01}, $R_3^{01\omega}$, $\ldots$, $R_2^{0\omega}$, $R_3^{0\omega\omega}$. Now we proceed to $R_1^1 = \psi_1(R_1^0, R_2^{0\omega}, R_3^{0\omega\omega})$. We know that $R_1^0 \subseteq R_1^1$, and we are now going to compute $R_2^{1\omega}$. Note that the values $R_2^{0\omega}$ and $R_2^{1\omega}$ are given by

$$R_2^{0\omega} = \nu R_2.\psi_2(R_1^0, R_2, \mu R_3.\psi_3(R_1^0, R_2, R_3))$$

and

$$R_2^{1\omega} = \nu R_2.\psi_2(R_1^1, R_2, \mu R_3.\psi_3(R_1^1, R_2, R_3)).$$

By monotonicity, we know that $R_2^{1\omega}$ will be a superset of $R_2^{0\omega}$. However, since R_2 is computed by a greatest fixpoint, this information does not help; we still must start computing with $R_2^{10} = \top$. At this point, we begin to compute the inner fixpoint again. But now let us look at $R_3^{00\omega}$ and $R_3^{10\omega}$. We have

$$R_3^{00\omega} = \mu R_3.\psi_3(R_1^0, R_2^{00}, R_3)$$

and

$$R_3^{10\omega} = \mu R_3.\psi_3(R_1^1, R_2^{10}, R_3).$$

Since $R_1^0 \subseteq R_1^1$ and $R_2^{00} \subseteq R_2^{10}$, monotonicity implies that $R_3^{00\omega} \subseteq R_3^{10\omega}$. Now R_3 is a least fixpoint, so starting the computation of $R_3^{10\omega}$ anywhere below the fixpoint value is acceptable. Thus, we can start the computation for $R_3^{10\omega}$ with $R_3^{100} = R_3^{00\omega}$. Since $R_3^{00\omega}$ is in general larger than $\perp$, we obtain faster convergence. In addition, we have

$$R_2^{01} = \psi_2(R_1^0, R_2^{00}, R_3^{00\omega})$$

and

$$R_2^{11} = \psi_2(R_1^1, R_2^{10}, R_3^{10\omega})$$

Since $R_1^0 \subseteq R_1^1$, $R_2^{00} \subseteq R_2^{10}$, and $R_3^{00\omega} \subseteq R_3^{10\omega}$, we will have $R_2^{01} \subseteq R_2^{11}$. This means that we can use the same trick when computing $R_3^{11\omega}$: we start the computation from $R_3^{110} = R_3^{01\omega}$. And again, since $R_1^0 \subseteq R_1^1$, $R_2^{01} \subseteq R_2^{11}$, and $R_3^{01\omega} \subseteq R_3^{11\omega}$, we will have $R_2^{02} \subseteq R_2^{12}$. In general, we will have $R_2^{0j} \subseteq R_2^{1j}$ and $R_3^{0j\omega} \subseteq R_3^{1j\omega}$ so we can start computing

$R_3^{1j\omega}$ from $R_3^{1j0} = R_3^{0j\omega}$. Similarly, once we find R_1^2 (or in general, R_1^{k+1}), we can start computing the inner fixpoints from $R_3^{2m0} = R_3^{1m\omega}$ ($R_3^{(k+1)m0} = R_3^{km\omega}$).

The table in Figure 2.3 illustrates this by showing the relationship between all the different possible approximation values for R_3. Each row can have at most $n+1$ entries, one for each approximation to $\nu R_2.\psi_2$. At first glance, it seems possible that each column could have as many as n^2 entries. However, each chain represented by each column can have at most $n+1$ distinct values. Repeated values only appear when convergence is reached ($R_3^{ij\omega} = R_3^{ij(\omega-1)}$) and when we start a computation from a previously computed fixpoint ($R_3^{(i+1)j0} = R_3^{ij\omega}$). Convergence is reached every time the fixpoint is evaluated, and this fixpoint is evaluated once for every outer greatest fixpoint approximation of which there can be no more than $n+1$. Since there can be no more than $n+1$ evaluations, we can start from a previously computed fixpoint no more than n times. So the number of repeated values is bounded by $2n+1$. Thus, the total number of entries in any column is bound by $3n+2$ and the total number of assignments to R_3 during the entire computation is bound by $(3n+2)(n+1)$. This means that there are at most $O(n^2)$ iterations performed to compute the innermost fixpoint.

Again, this algorithm for evaluating a μ-calculus formula is identical to the naive algorithm except when the main connective is a fixpoint operator. To facilitate explanation, we consider only formulas with strict alternation of fixpoints, and in particular, with the form:

$$\begin{array}{rcl}
F_1 & \equiv & \mu R_1.\psi_1(R_1, \nu R_1'.\psi_1'(R_1, R_1', F_2)) \\
F_2 & \equiv & \mu R_2.\psi_2(R_1, R_1', R_2, \nu R_2'.\psi_2'(R_1, R_1', R_2, R_2', F_3)) \\
 & \vdots & \\
F_q & \equiv & \mu R_q.\psi_q(R_1, R_1', \ldots, R_q, \nu R_q'.\psi_q'(R_1, R_1', \ldots, R_q, R_q'))
\end{array}$$

The pseudocode for this part of the algorithm is given in Figure 2.4. For computing the outermost fixpoint (corresponding to R_1) we follow the naive algorithm, i.e., start with $\bot$ and iterate until convergence. The algorithm uses a table $\mathcal{T}_i$ to store the last computed fixpoint values for the μ-variables R_i (for $i \geq 2$). Initially, all entries in $\mathcal{T}_i$ are $\bot$. The table $\mathcal{T}_i$ is a multi-dimensional table. For the i-th least fixpoint

$$
\begin{array}{ccccc}
R_3^{\omega 0 \omega} & \supseteq & R_3^{\omega 1 \omega} & \supseteq \cdots \supseteq & R_3^{\omega\omega\omega} \\
\rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} \\
\vdots & & \vdots & & \vdots \\
\rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} \\
R_3^{\omega 01} & \supseteq & R_3^{\omega 11} & \supseteq \cdots \supseteq & R_3^{\omega\omega 1} \\
\rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} \\
R_3^{\omega 00} & \supseteq & R_3^{\omega 10} & \supseteq \cdots \supseteq & R_3^{\omega\omega 0} \\
\| & & \| & & \| \\
\vdots & & \vdots & & \vdots \\
\| & & \| & & \| \\
R_3^{10\omega} & \supseteq & R_3^{11\omega} & \supseteq \cdots \supseteq & R_3^{1\omega\omega} \\
\rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} \\
\vdots & & \vdots & & \vdots \\
\rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} \\
R_3^{101} & \supseteq & R_3^{111} & \supseteq \cdots \supseteq & R_3^{1\omega 1} \\
\rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} \\
R_3^{100} & \supseteq & R_3^{110} & \supseteq \cdots \supseteq & R_3^{1\omega 0} \\
\| & & \| & & \| \\
R_3^{00\omega} & \supseteq & R_3^{01\omega} & \supseteq \cdots \supseteq & R_3^{0\omega\omega} \\
\rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} \\
\vdots & & \vdots & & \vdots \\
\rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} \\
R_3^{001} & \supseteq & R_3^{011} & \supseteq \cdots \supseteq & R_3^{0\omega 1} \\
\rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} \\
R_3^{000} & \supseteq & R_3^{010} & \supseteq \cdots \supseteq & R_3^{0\omega 0}
\end{array}
$$

Figure 2.3: Monotonicity constraints on approximations to R_3

(corresponding to R_i) we index the table $\mathcal{T}_i$ by the iteration counters $k_1, \cdots, k_{i-1}$ of the greatest fixpoints in which the i-th least fixpoint is nested. When evaluating R_i, we start with the corresponding table value and iterate until convergence. At the end of the iteration, the table holds the fixpoint value. When evaluating R'_i, we always begin with $\top$ and iterate until convergence. Note that this algorithm implements

if $\phi = \mu R_i.\psi_i(R_i)$ and $i \geq 2$ then
$\quad R_{\text{val}} := \mathcal{T}_i[k_1]\cdots[k_{i-1}]$
$\quad$ repeat
$\qquad R_{\text{old}} := R_{\text{val}}$
$\qquad R_{\text{val}} := \text{evalrec}(\psi_i, e\,[R_i \leftarrow R_{\text{old}}])$
$\quad$ until $R_{\text{val}} = R_{\text{old}}$
$\quad \mathcal{T}_i[k_1]\cdots[k_{i-1}] := R_{\text{val}}$
$\quad$ return R_{val}

if $\phi = \nu R'_i.\psi'_i(R'_i)$ then
$\quad k_i := 0$
$\quad R_{\text{val}} := \top$
$\quad$ repeat
$\qquad R_{\text{val}} := \text{evalrec}(\psi'_i, e\,[R'_i \leftarrow R_{\text{val}}])$
$\qquad k_i := k_i + 1$
$\quad$ until $k_i = |\top|$
$\quad$ return R_{val}

Figure 2.4: Pseudocode for the efficient algorithm

the ideas in the previous example.

If we use these ideas, how many steps does the computation take? To try to answer this question, we look at the number of approximations computed for the R_is and R'_is in the algorithm. Let T_i denote the number of approximations for R_i, and let T'_i denote the number of approximations for R'_i. The fixpoint for R'_i is evaluated at most T_i times (the number of approximations to the enclosing R_i). Each evaluation can take at most $n+1$ iterations for a total of $(n+1)T_i$ approximations. Thus, $T'_i \leq (n+1)T_i$. The fixpoint for R_i has a table $\mathcal{T}_i$ with $(n+1)^{i-1}$ entries. Because of the monotonicity constraints, each entry can go through at most $n+1$ distinct values. Since there are $(n+1)^{i-1}$ entries, we have a total of $(n+1)^i$ iterations. These iterations correspond to the case when the loop test is false. In addition, each time we evaluate the fixpoint for R_i we will take one extra step to detect convergence which will not result in a new value for the corresponding table entry. We evaluate the fixpoint for R_i at most T'_{i-1} times. Thus we make at most T'_{i-1} iterations when the loop test is true. In total, we have $T_i \leq (n+1)^i + T'_{i-1}$. Solving this recurrence, we get:

$$\begin{aligned} T_i &\leq i(n+1)^i \\ T'_i &\leq i(n+1)^{i+1} \end{aligned}$$

Summing over all fixpoints and expressing the result in terms of the alternation depth $d = 2q$, we get that the algorithm takes $O\left(d(n+1)^{d/2+1}\right)$ iterations when computing the fixpoints in a formula. In comparison, previously known algorithms may require $O(n^d)$ iterations.

2.3 Ordered Binary Decision Diagrams (OBDDs)

In this section we give a brief description of an efficient data structure for representing boolean functions. Consider the space $\mathcal{BF}_n$ of boolean functions on n variables $x_0, x_1, \cdots, x_{n-1}$. We assume that there is a total ordering on the boolean variables. The ordering is given by the index, i.e., x_i is ordered before x_j iff $i < j$. The symbol OBDD(f) will denote the Ordered Binary Decision Diagram (OBDD) for the boolean function f [11]. OBDDs have the following *canonicity* property:

Theorem 2.3.1 (Canonicity Theorem): Given two boolean functions f and g in the space $\mathcal{BF}_n$, OBDD(f) = OBDD(g) iff $f = g$.

A detailed proof is given in [11].

We will give a succinct explanation of how OBDDs work through an example. For a more thorough treatment see [11, 14]. Consider the following boolean function f:

$$f = x_0 \oplus x_1 \oplus x_2$$

Figure 2.5 gives the binary tree T corresponding to the boolean function f. Notice that the binary subtree which we get by following the paths $(0, 1)$ and $(1, 0)$ from the root are the same. The same is true if we follow

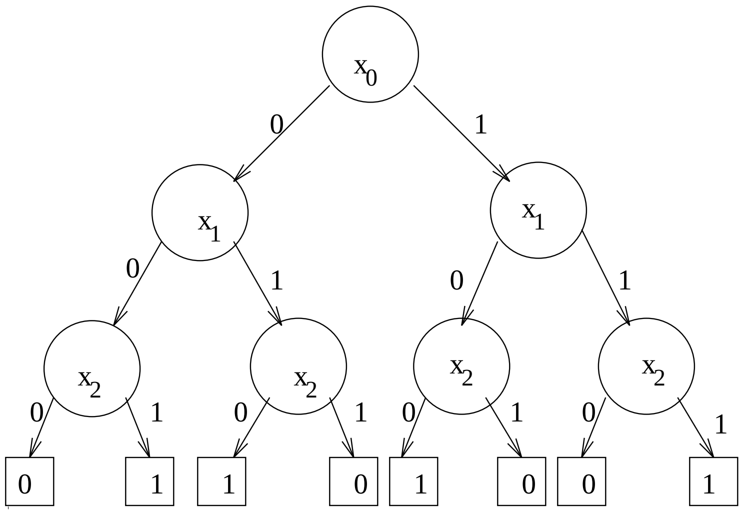

Figure 2.5: Tree for the 3 bit parity function

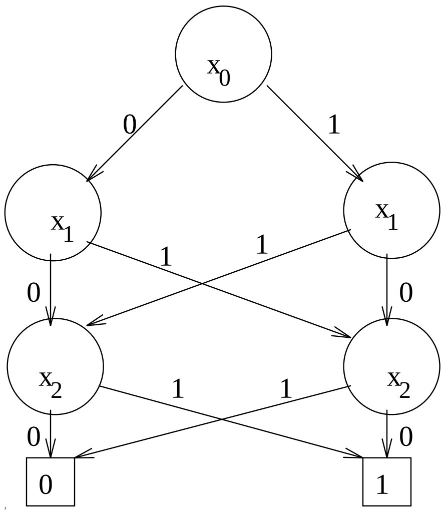

Figure 2.6: OBDD for the 3 bit parity function

the paths $(0,0)$ and $(1,1)$. Figure 2.6 reflects this sharing. Notice that the number of nodes is reduced from 15 to 7. In general, the binary tree corresponding to the parity of n bits has $2^{n+1}-1$ nodes. The OBDD for the same function has $2n+1$ nodes. Therefore, in some cases

OBDD can be exponentially more succinct than the straightforward representation. We will use $|\mathrm{OBDD}(f)|$ to denote the size of the OBDD for f, i.e., the number of nodes in $\mathrm{OBDD}(f)$. In addition to being a canonical representation, OBDDs support the usual operations on boolean functions efficiently. The complexity of some of the operations is shown below:

- Given the OBDDs for f and g, the OBDD for $f \vee g$ and $f \wedge g$ can be computed in time $O(|\mathrm{OBDD}(f)| \cdot |\mathrm{OBDD}(g)|)$.

- Given the OBDD for f, the OBDD for $\neg f$ can be computed in time $O(|\mathrm{OBDD}(f)|)$.

- Given the OBDD for f, the OBDDs for $\exists x_i f$ and $\forall x_i f$ can be computed in time $O(|\mathrm{OBDD}(f)|^2)$.

Variable ordering is extremely important to OBDDs. For example, consider the following boolean function :

$$f(x_1, \cdots, x_n, x'_1, \cdots, x'_n) \quad = \quad \bigwedge_{i=1}^{n} (x_i = x'_i)$$

The OBDD for f with the variable ordering

$$x_1 < x'_1 < x_2 < x'_2 < \cdots x_n < x'_n$$

has size $3n + 2$. As the following lemma shows, the OBDD for f can have size exponential in n under some variable orderings. Moreover, there are some functions whose OBDDs have exponential size under any variable ordering [11].

Lemma 2.3.1 Let $f(x_1, \cdots, x_n, x'_1, \cdots, x'_n)$ be the following boolean function:

$$\bigwedge_{i=1}^{n} (x_i = x'_i)$$

Let F be the OBDD for f such that all the unprimed variables are ordered before all the primed variables. In this case $|F| \geq 2^n$.

Proof: Consider two distinct assignments $(b_1, \cdots, b_n)$ and $(c_1, \cdots, c_n)$ to the boolean vector $(x_1, \cdots, x_n)$. These two assignments can be distinguished because of the following equation:

$$f(b_1, \cdots, b_n, b_1, \cdots, b_n) \neq f(c_1, \cdots, c_n, b_1, \cdots, b_n)$$

Let v_1 and v_2 be the nodes reached after following the path $(b_1, \cdots, b_n)$ and $(c_1, \cdots, c_n)$ from the top node. Since these two assignments can be distinguished, $v_1 \neq v_2$. There are 2^n different assignments to the boolean vector $(x_1, \cdots, x_n)$ and each of them corresponds to a different node (at level n) in the OBDD F. Therefore, the number of nodes at level n in the OBDD F is greater than or equal to 2^n. $\square$

2.4 Translating the μ-Calculus into OBDDs

In this section we describe how to use OBDDs in the model checking algorithms described earlier. First, we show how to encode a transition system $M = (\top, T, L)$ into OBDDs. The domain $\top$ is encoded by the set of values of the n boolean variables $x_1, \cdots, x_n$, i.e., $\top$ is now the space of 0-1 vectors of length n. Each variable x_i has a corresponding primed variable x_i'. Instead of writing $x_1, \cdots, x_n$, we sometimes use the vector notation $\vec{x}$. For example, we write $\mathrm{OBDD}_p(x_1, \cdots, x_n)$ as $\mathrm{OBDD}_p(\vec{x})$. Given an interpretation, we build the OBDDs corresponding to closed μ-calculus formulas in the following manner.

- Each atomic proposition p has an OBDD associated with it. We will denote this OBDD by $\mathrm{OBDD}_p(\vec{x})$. $\mathrm{OBDD}_p(\vec{x})$ has the property that $\vec{y} \in \{0,1\}^n$ satisfies OBDD_p iff $\vec{y} \in L(p)$.

- Each program letter a has an ordered binary decision diagram $\mathrm{OBDD}_a(\vec{x}, \vec{x}')$ associated with it. A 0-1 vector $(\vec{y}, \vec{z}) \in \{0,1\}^{2n}$ satisfies OBDD_a iff
$$(\vec{y}, \vec{z}) \in T(a)$$

Now we describe the encoding of the semantic sets of formulas into OBDDs. Assume that we are given a μ-calculus formula ϕ with free propositional variables $R_1, \cdots, R_k$. $\mathcal{A}[R_i]$ gives the OBDD corresponding to the propositional variable R_i. $\mathcal{A}\langle R \leftarrow B_R\rangle$ creates a new association by adding a propositional variable R and associating an OBDD B_R with R. In other words, $\mathcal{A}$ can be considered as an environment with OBDD representation. The procedure B given below takes a μ-calculus formula ϕ and an association list $\mathcal{A}$ ($\mathcal{A}$ assigns an OBDD to each free propositional variable occurring in ϕ) and returns an OBDD corresponding to the semantics of ϕ.

- $B(p, \mathcal{A}) = \text{OBDD}_p(\vec{x})$.
- $B(R_i, \mathcal{A}) = \mathcal{A}[R_i]$.
- $B(\neg\phi, \mathcal{A}) = \neg B(\phi, \mathcal{A})$
- $B(\phi \wedge \psi, \mathcal{A}) = B(\phi, \mathcal{A}) \wedge B(\psi, \mathcal{A})$.
- $B(\phi \vee \psi, \mathcal{A}) = B(\phi, \mathcal{A}) \vee B(\psi, \mathcal{A})$.
- $B(\langle a\rangle\phi, \mathcal{A}) = \exists\vec{x'}(\text{OBDD}_a(\vec{x}, \vec{x'}) \wedge B(\phi, \mathcal{A})(\vec{x'}))$
- $B([a]\phi, \mathcal{A}) = B(\neg\langle a\rangle\neg\phi, \mathcal{A})$.
  The second equation uses the dual formulation for $[a]$.
- $B(\mu R.\phi, \mathcal{A}) = \mathit{FIX}(\phi, \mathcal{A}, \text{FALSE-BDD})$.
- $B(\mu R.\phi, \mathcal{A}) = \mathit{FIX}(\phi, \mathcal{A}, \text{TRUE-BDD})$.

The OBDDs for the boolean functions **false** and **true** are denoted by FALSE-BDD and TRUE-BDD respectively. Notice that ϕ has an extra free propositional variable R. *FIX* is described in Figure 2.7.

Now we give a short example to illustrate our point.

Example 2.4.1 Assume that the state space $\top$ is encoded by n boolean variables $x_1, \cdots, x_n$. Consider the following formula:

$$\phi \quad = \quad \mu Z.(q \wedge Y \vee \langle a\rangle Z)$$

Notice that the variable Y is free in ϕ. Assume that the interpretation for q is an OBDD $\text{OBDD}_q(\vec{x})$. Similarly, the OBDD corresponding to function *FIX*($\phi, \mathcal{A}, B_R$)

result-bdd = B_R
do
    old-bdd = result-bdd
    result-bdd = $B(\phi, \mathcal{A}\langle R \leftarrow$ old-bdd$\rangle)$
while (not-equal(old-bdd, result-bdd))
return(result-bdd)

Figure 2.7: Pseudocode for the function *FIX*

the program letter a is $\text{OBDD}_a(\vec{x}, \vec{x'})$. Also assume that we are given an association list $\mathcal{A}$ which pairs the OBDD $B_Y(\vec{x})$ with Y. In the routine *FIX* the OBDD result-bdd is initially set to:

$$\begin{aligned} N^0(\vec{x}) &= \text{OBDD}_q(\vec{x}) \wedge B_Y(\vec{x}) \vee \exists \vec{x'}(\text{OBDD}_a(\vec{x}, \vec{x}') \wedge \text{FALSE-BDD}) \\ &= \text{OBDD}_q(\vec{x}) \wedge B_Y(\vec{x}) \end{aligned}$$

Let N^i be the value of result-bdd at the i-th iteration in the loop of the function *FIX*. At the end of the iteration the value of result-bdd is given by:

$$N^{i+1}(\vec{x}) = \text{OBDD}_q(\vec{x}) \wedge B_Y(\vec{x}) \vee \exists \vec{x}'(\text{OBDD}_a(\vec{x}, \vec{x}') \wedge N^i(\vec{x}'))$$

The iteration stops when $N^i(\vec{x}) = N^{i+1}(\vec{x})$.

2.5 Branching Time Temporal Logics

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

- S is a finite set of *states*,
- $T \subseteq S \times S$ is a *transition relation*, which must be total (i.e., for every state s_1 there exists a state s_2 such that $(s_1, s_2) \in T$).

- $L : S \rightarrow 2^{AP}$ is a *labeling function* which associates with each state a set of atomic propositions that are true in the state.

There are two types of formulas in the temporal logic $CTL^{\star}$: *state formulas* (which are true in a specific state) and *path formulas* (which are true along a specific path). The state operators in $CTL^{\star}$ are: **A** ("for all computation paths"), **E** ("for some computation paths"). The path operators in $CTL^{\star}$ are: **G** ("always"), **F** ("sometimes"), **U** ("until"), and **V** ("unless"). Let AP be a set of atomic propositions. A state formula is either:

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

Path formulas are defined as follows:

- every state formula is a path formula; and
- if f and g are path formulas, then $\neg f$, $f \vee g$, $\mathbf{X}\, f$, $f\, \mathbf{U}\, g$, and $f\, \mathbf{V}\, g$ are path formulas.

$CTL^{\star}$ is the set of state formulas generated by the above rules.

We define the semantics of $CTL^{\star}$ with respect to a Kripke structure $M = (S, T, L)$. A *path* in M is an infinite sequence of states $\pi = s_0, s_1, \ldots$ such that, for every $i \geq 0, (s_i, s_{i+1}) \in T$. π^i denotes the *suffix* of π starting at s_i. $\pi[i]$ denotes the i-th state on that path π. The starting state of path π is $\pi[0]$. We use the standard notation to indicate that a state formula f holds in a structure. $M, s \models f$ means that f holds at the state s in the structure M. Similarly, $M, \pi \models f$ means that the path formula f is true along the path π. Assume that f_1 and f_2 are state formulas and g_1 and g_2 are path formulas, then the relation $\models$ is defined inductively as follows:

1. $s \models p \Leftrightarrow p \in L(s)$
2. $s \models \neg f_1 \Leftrightarrow s \not\models f_1$

3. $s \models f_1 \vee f_2 \Leftrightarrow s \models f_1$ or $s \models f_2$

4. $s \models \mathbf{E}(g_1) \Leftrightarrow$ there exists a path π starting with s such that $\pi \models g_1$

5. $\pi \models f_1 \Leftrightarrow s\ \pi[0] \models f_1$.

6. $\pi \models \neg g_1 \Leftrightarrow \pi \not\models g_1$

7. $\pi \models g_1 \vee g_2 \Leftrightarrow \pi \models g_1$ or $\pi \models g_2$

8. $\pi \models \mathbf{X}\, g_1 \Leftrightarrow \pi^1 \models g_1$

9. $\pi \models g_1\ \mathbf{U}\ g_2 \Leftrightarrow$ there exists $k \geq 0$ such that $\pi^k \models g_2$ and for all $0 \leq j < k, \pi^j \models g_1$.

10. $\pi \models g_1\ \mathbf{V}\ g_2 \Leftrightarrow$ for every $k \geq 0$, if $\pi^j \not\models g_1$ for all $0 \leq j < k$, then $\pi^k \models g_2$.

$CTL^{\star}$-X is the subset of $CTL^{\star}$ without the $\mathbf{X}$ operator. LTL is a subset of $CTL^{\star}$ which only allows formula of the form $\mathbf{A}\,\phi$, where ϕ only has path operators ($\mathbf{V}$, $\mathbf{U}$ and $\mathbf{X}$). LTL-X is the subset of LTL without the next-time operator $\mathbf{X}$. Because of the following equalities we only consider the path operators $\mathbf{V}$ and $\mathbf{U}$.

$$\begin{aligned} \mathbf{F}\,\phi &= \mathbf{True}\ \mathbf{U}\,\phi \\ \mathbf{G}\,\phi &= \mathbf{False}\ \mathbf{V}\,\phi \end{aligned}$$

CTL is the subset of $CTL^{\star}$ in which the path formulas are restricted to be:

- if f and g are state formulas, then $\mathbf{X}\,f$, $f\,\mathbf{U}\,g$, and $f\,\mathbf{V}\,g$ are path formulas.

The basic modalities of CTL are $\mathbf{EX}\,f$, $\mathbf{EG}\,f$, and $\mathbf{E}(f\ \mathbf{U}\ g)$, where f and g are again CTL formulas. The operator $\mathbf{E}(f\,\mathbf{V}\,g)$ can be expressed as follows:

$$\begin{aligned} \mathbf{E}(f\ \mathbf{V}\ g) &= \mathbf{E}((\neg f \wedge g)\ \mathbf{U}\ f \wedge g) \vee \mathbf{EG}(\neg f \wedge g) \\ \mathbf{EF}\,f &= \mathbf{E}(true\ \mathbf{U}\ f) \end{aligned}$$

The operators $\mathbf{AG}\, f$, $\mathbf{AF}\, f$ and $\mathbf{A}(f\, \mathbf{U}\, g)$ can be expressed in terms of the basic modalities described above.

$$\begin{aligned} \mathbf{AG}\, f &= \neg\, \mathbf{EF}\, \neg f \\ \mathbf{AF}\, f &= \neg\, \mathbf{EG}\, \neg f \\ \mathbf{A}(f\, \mathbf{U}\, g) &= \neg \mathbf{E}(\neg f\, \mathbf{V}\, \neg g) \end{aligned}$$

Next, we discuss the issue of *fairness*. In many cases, we are only interested in the correctness along paths with certain conditions. For example, if we are verifying a protocol with a scheduler, we may wish to consider only executions where processes are not ignored by the scheduler, i.e., every process is given a chance to run infinitely often. This type of fairness constraint cannot be expressed in *CTL* [17]. In order to handle such properties we have to modify the semantics of *CTL*. A *fairness constraint* can be an arbitrary set of states, usually described by a *CTL* formula. Generally, there will be several fairness constraints. In this paper we will denote the set of all fairness constraints by $H = \{h_1, \cdots, h_n\}$. We have the following definition of a *fair* path.

Definition 2.5.1 Given a Kripke Structure $M = (S, T, L)$ and a set of fairness constraints $H = \{h_1, \cdots, h_n\}$, a path π in M is called *fair* iff each *CTL* formula h_i is satisfied infinitely often on the path π.

The semantics of *CTL* has to be modified to handle fairness constraints H. The basic idea is to restrict path quantifiers to fair paths. The formal definition is given below:

- $s \models \mathbf{EX}_H\, f$ iff there exists a fair path π starting from the state s such that $\pi[1] \models f$.

- $s \models \mathbf{E}(g_1\, \mathbf{U}_H\, g_2)$ iff there exists a fair path π starting from the state s and there exists $k \geq 0$ such that $\pi[k] \models g_2$ and for all $0 \leq j < k, \pi[j] \models g_1$.

- $s \models \mathbf{EG}_H\, f$ iff there exists a fair path π starting from the state s such that for all $i \geq 0$, $\pi[i] \models f$.

2.6 Translating *CTL* into the μ-Calculus

In this section we give a translation of CTL into the propositional μ-calculus. The algorithm Tr takes as its input a CTL formula and outputs an equivalent μ-calculus formula with only one action a.

- $Tr(p) = p$.
- $Tr(\neg f) = \neg Tr(f)$.
- $Tr(f \wedge g) = Tr(f) \wedge Tr(g)$.
- $Tr(\mathbf{EX}\, f) = \langle a \rangle Tr(f)$.
- $Tr(\mathbf{E}(f\, \mathbf{U}\, g)) = \mu Y.(Tr(g) \vee (Tr(f) \wedge \langle a \rangle Y))$.
- $Tr(\mathbf{EG}\, f) = \nu Y.(Tr(f) \wedge \langle a \rangle Y)$.

Note, that any resulting μ-calculus formula is closed. Therefore, we can omit the environment in the set $[\![\phi]\!]_M$.

Lemma 2.6.1 Let $M = (S, T, L)$ be a Kripke Structure, f be a CTL formula, and a be an action with interpretation T. Consider the predicate transformer τ.

$$\begin{aligned} \tau(Z) &= f \wedge \langle a \rangle Z \\ &= \{s \in S \mid s \models f \ \wedge \ \exists s' \in S((s, s') \in T \ \wedge \ s' \in Z)\} \end{aligned}$$

τ satisfies the following conditions:

- τ is monotonic.
- Let $\tau^{i_0}(\top)$ be the limit of the sequence $\top \subseteq \tau(\top) \subseteq \cdots$. For every $s \in S$, if $s \in \tau^{i_0}(\top)$ then $s \models f$, and there is a state s' such that $(s, s') \in T$ and $s' \in \tau^{i_0}(\top)$.

Proof: Let $P_1 \subseteq P_2$. In this case $\langle a \rangle P_1 \subseteq \langle a \rangle P_2$, i.e., the successor relation is monotonic. Therefore, we have that $\tau(P_1) \subseteq \tau(P_2)$. Since $\tau^{i_0}(\top)$ is the fixpoint of the predicate transformer τ, we have the following equation:

$$\tau(\tau^{i_0}(\top)) \;=\; \tau^{i_0}(\top)$$

Let $s \in \tau^{i_0}(\top)$. Using the equation given above we get that $s \in \tau(\tau^{i_0}(\top))$. By definition of τ we get that $s \models f$ and there exists a state s', such that $(s, s') \in T$ and $s' \in \tau^{i_0}(\top)$. $\square$

The theorem given below proves the correctness of the translation algorithm Tr.

Theorem 2.6.1 Let $M = (S, T, L)$ be the underlying Kripke Structure. Let ϕ be a CTL formula. Let the interpretation of the action a be T. An atomic proposition p in $Tr(\phi)$ has the interpretation $L(p)$. The set of states $\top$ is S. In this case, for all $s \in S$

$$s \models \phi \quad \Leftrightarrow \quad s \in [\![Tr(\phi)]\!]_M$$

Proof: The proof is by structural induction on ϕ.

- $\phi = p$: In this case the result is true by definition.
- $\phi = \neg f$: By definition $[\![Tr(\phi)]\!]_M = S - [\![Tr(f)]\!]_M$. The result follows by using the induction hypothesis on f.
- $\phi = f \wedge g$: By definition $[\![Tr(\phi)]\!]_M = [\![Tr(f)]\!]_M \cap [\![Tr(g)]\!]_M$. The result follows by using the induction hypothesis on f and g.
- $\phi = \mathbf{EX}\, f$: Let S_f be the set of states where f is true. By the induction hypothesis, $[\![Tr(f)]\!]_M = S_f$. The set of states satisfying ϕ is the set of states S_1 which have a successor in S_f. It is clear from the semantics of $\langle a \rangle$ that $[\![Tr(\phi)]\!]_M = S_1$.
- $\phi = \mathbf{EG}\, f$: Let Y_1 be the set of states s such that $s \models \mathbf{EG}\, f$. Let $\tau : 2^S \rightarrow 2^S$ be the following predicate transformer

  $$\tau(Z) \quad = \quad [\![Tr(f)]\!]_M \cap ([\![\langle a \rangle X]\!]_M\, e\, [X \leftarrow Z])$$

  By definition, the greatest fixpoint of τ is given by $\bigcap_i \tau^i(\top)$, where $\tau^0(\top) = \top$, and $\tau^{i+1}(\top) = \tau(\tau^i(\top))$. Using the semantics of $\mathbf{EG}$ we get that if $s \in Y_1$, then there exists a path π starting from s such that each state on the path satisfies f. Therefore, if $s \in Y_1$, then s has a successor s' such that $(s, s') \in T$, $s \models f$, and

$s' \models \mathbf{EG}\, f$. Hence Y_1 is a fixpoint for the predicate transformer τ, i.e.,

$$\tau(Y_1) \;=\; Y_1$$

Since $\bigcap_i \tau^i(\top)$ is the greatest fixpoint of τ, we have the following inclusion:

$$Y_1 \;\subseteq\; \bigcap_i \tau^i(\top)$$

Now assume that $s \in \bigcap_i \tau^i(\top)$. By Lemma 2.6.1, s is the start of an infinite path π such that each state s' on the path π satisfies f. Therefore, we have the following inclusion:

$$Y_1 \;\supseteq\; \bigcap_i \tau^i(\top)$$

Using the two equations we get that Y_1 is the greatest fixpoint of the predicate transformer τ.

- $\phi = \mathbf{E}(f\,\mathbf{U}\,g)$: Let S_1 be the set of states s such that $s \models \mathbf{E}(f\,\mathbf{U}\,g)$. Let $\tau : 2^S \rightarrow 2^S$ be the following predicate transformer:

$$\tau(Z) \;=\; [\![Tr(g)]\!]_M \cup ([\![Tr(f)]\!]_M \cap ([\![\langle a\rangle X]\!]_M\, e\,[X \leftarrow Z]))$$

First, we will show that S_1 is a fixpoint of τ, i.e.,

$$\tau(S_1) \;= S_1$$

By definition, $s \models \mathbf{E}(f\ \mathbf{U}\ g)$ iff there exists a path π starting from s such that there exists a $k \geq 0$ with the property that $\pi^k \models g$ and $\pi^i \models f$ (for $0 \leq i < k$). Equivalently, $s \models \mathbf{E}(f\ \mathbf{U}\ g)$ iff $s \models g$ or $s \models f$ and there exists a state s_1 such $(s, s_1) \in T$ and $s_1 \models \mathbf{E}(f\ \mathbf{U}\ g)$. From this condition it is clear that S_1 is a fixed point of the predicate transformer τ. By definition, the least fixpoint of τ is given by

$$\bigcup_i \tau^i(\perp)$$

Since S_1 is a fixpoint for τ, we have that

$$S_1 \supseteq \bigcup_i \tau^i(\perp)$$

Next we prove that

$$S_1 \subseteq \bigcup_i \tau^i(\perp)$$

which proves that S_1 is equal to the least fixpoint of the predicate transformer τ. By definition, if $s \in S_1$, then there exists a path π and a $k \geq 0$ such that $\pi^k \models g$ and $\pi^j \models f$ (for $j < k$). We will prove by induction on k that $s \in \tau^k(\perp)$. The basis case is trivial. If $k = 0$, then $s \models g$ and therefore $s \in \tau(\perp)$, which is equal to $[\![Tr(g)]\!]_M \cup ([\![Tr(f)]\!]_M \cap [\![\langle a \rangle \perp]\!]_M) = [\![Tr(g)]\!]_M$.

For the inductive step, assume that the above claim holds for every s and every $k \leq m$. Let s be the start of a path $\pi = s_0, s_1, \cdots$ such that $s_{m+1} \models g$ and for every $i < m+1$, $s_i \models f$. By induction hypothesis $s_1 \in \tau^m(\perp)$. Notice that $s_0 = s \in [\![Tr(f)]\!]_M$ and $s \in \langle a \rangle \tau^m(\perp)$. Therefore, by definition $s \in \tau^{m+1}(\perp)$. Hence, if $s \in S_1$, then $s \in \bigcup_i \tau^i(\perp)$.

Using Theorem 2.6.1 we have the following result:

Theorem 2.6.2 Given a Kripke Structure $M = (S, T, L)$, an initial state $s_0 \in S$, and a CTL formula f, one can decide in $O(|S||f|)$ iterations whether $M, s_0 \models f$. Where $|f|$ denotes the number of symbols in the formula f.

Proof: Consider the following formula:

$$\nu Y.(\mu Z.(q \vee (p \wedge \langle a \rangle Z)) \wedge \langle a \rangle Y)$$

Notice that the formula given above is $Tr(\mathbf{EG}(\mathbf{E}(p\,\mathbf{U}\,q)))$. Since the inner least fixpoint does not use the propositional variable Y (associated with the outer greatest fixpoint), we can compute it first and reuse that value in the outer fixpoint computation. Therefore, if we compute the inner fixpoint first, we can evaluate the formula given above in $O(2|S|)$ iterations. Notice that given a CTL formula f, $Tr(f)$ has the property that the inner fixpoints never use the variables associated with the outer fixpoint. By evaluating the fixpoints in the nesting order (evaluating the inner fixpoints first), we do not have to recompute the fixpoints. Therefore, the total complexity is the sum of the complexities for evaluating each fixpoint independently. This is bounded by $O(|S||f|)$.[1] □

Given fairness constraints $H = \{h_1, \cdots, h_n\}$, we extend the translation algorithm Tr in the following way:

- $Tr(\mathbf{EG}_H\, f) = \nu Y.\Big(Tr(f) \wedge \langle a \rangle \bigwedge_{i=1}^{n} \mu X.\, [(Tr(f) \wedge \langle a \rangle X) \vee (Tr(h_i) \wedge Y)]\Big)$

We introduce the following formula which is satisfied at a state s iff there is a fair path π starting from s.

- $\mathit{fair} \;=\; \mathbf{EG}_H\, \mathit{True}$
- $Tr(\mathbf{EX}_H\, f) = \langle a \rangle (Tr(f) \wedge Tr(\mathit{fair}))$.
- $Tr(\mathbf{E}(f\, \mathbf{U}_H\, g)) = \mu Y.(Tr(g) \wedge Tr(\mathit{fair}) \vee (Tr(f) \wedge \langle a \rangle Y))$.

We will give an informal proof of correctness for the $\mathbf{EG}_H$ case. Consider the following formula:

$$\nu Y.(P \wedge \langle a \rangle \mu X.\, [(P \wedge \langle a \rangle X) \vee (h \wedge Y)])$$

This corresponds to the formula $Tr(\mathbf{EG}_H\, f)$, where $H = \{h\}$ and $P = Tr(f)$. First, note that the condition "h holds infinitely often along a path" is equivalent to saying that from any point along that path in a finite number of steps we will reach a state where h holds. To understand the formula given above, notice that $\mu X.((P \wedge \langle a \rangle X) \vee (h \wedge Y))$ means that "P holds until $h \wedge Y$, and $h \wedge Y$ is reachable in a finite number of steps". Since the outer fixed point $\nu Y.(P \wedge \cdots)$ indicates that this property holds globally along the path, the formula exactly corresponds to the desired property.

---

[1] By definition of alternation depth given in [2], the formula $Tr(f)$ always has alternation depth one. Hence, the linear complexity of CTL model checking follows directly from the algorithm in [2].

2.7 Preorders and Equivalences.

2.7.1 Simulation and bisimulation.

In this section we will use essentially the same definition of a transition system that was introduced in Section 2.1, except for two special program letters τ and ε. The letter τ represents the *idle* action; its interpretation is always fixed: $T(\tau) = \{(s,s) \mid s \in S\}$. The program letter ε denotes the *invisible* action from CCS [58] and will be used in the definition of the weak simulation and bisimulation relations.

Definition 2.7.1 A relation $\mathcal{E} \subseteq S \times S$ is called a *simulation relation*, if for every $(s,s') \in \mathcal{E}$ the following condition holds:

$$\forall a \in Act. \forall q \in S. \text{ if } s \xrightarrow{a} q \text{ then } \exists q' \in S. s' \xrightarrow{a} q' \text{ and } (q,q') \in \mathcal{E}.$$

Definition 2.7.2 A relation $\mathcal{E} \subseteq S \times S$ is called a *bisimulation* if $\mathcal{E}$ and $\mathcal{E}^{-1}$ are both simulation relations. In other words, $\mathcal{E}$ satisfies the following conditions: $(s,s') \in \mathcal{E}$ iff

$$\begin{array}{ll} (i) & \forall a \in Act. \forall q \in S. \text{ if } s \xrightarrow{a} q \text{ then } \exists q' \in S. s' \xrightarrow{a} q' \text{ and } (q,q') \in \mathcal{E}; \\ (ii) & \forall a \in Act. \forall q' \in S. \text{ if } s' \xrightarrow{a} q' \text{ then } \exists q \in S. s \xrightarrow{a} q \text{ and } (q,q') \in \mathcal{E}. \end{array}$$

We define the *simulation preorder* as follows:

$$s \preceq s' \text{ iff there exists a simulation relation } \mathcal{E} \text{ such that } (s,s') \in \mathcal{E}.$$

We define *bisimulation equivalence* in a similar manner:

$$s \sim s' \text{ iff there exists a bisimulation relation } \mathcal{E} \text{ such that } (s,s') \in \mathcal{E}.$$

It is straightforward to check that $\preceq$ is a preorder. In fact, it is the maximal simulation relation under inclusion. It is also possible to show that bisimulation equivalence $\sim$ is an equivalence relation. Moreover, it is the maximal bisimulation relation under inclusion.

2.7.2 Encoding simulation and bisimulation

In order to check if the initial states of two transition systems are bisimilar using the propositional μ-calculus, we first need to construct a new transition system. Given two transition systems $M = (S, T, L)$ over Act and $M' = (S, T', L')$ over Act, we define the product $\tilde{M} = M \times M'$ over $\tilde{Act}$ as follows: $\tilde{M} = (\tilde{S}, \tilde{T}, \tilde{L})$, where

- $\tilde{Act} = Act \times Act = \{ab \mid a \in Act \text{ and } b \in Act\}$,
- $\tilde{S} = S \times S$,
- $(s, s') \xrightarrow{ab} (q, q')$ iff $s \xrightarrow{a} q$ and $s' \xrightarrow{b} q'$.
- $\tilde{L}$ may be arbitrary in this case.

We assume that M and M' have the same state and action sets. This is a technical issue because we can always define the transition systems on larger state and action sets.

Theorem 2.7.1 Let s and s' be the states of the two transition systems M and M'. Then $s \preceq s'$ iff the following formula holds in the state (s, s') of the transition system $\tilde{M}$:

$$\nu X. \Big(\bigwedge_{a \in Act} [a\tau] \langle \tau a \rangle X \Big)$$

Proof: Consider the definition of a simulation relation:

$(s, s') \in \mathcal{E}$ iff $\forall a \in Act. \forall q \in S.$ if $s \xrightarrow{a} q$ then $\exists q' \in S. s' \xrightarrow{a} q'$ and $(q, q') \in \mathcal{E}$.

This is the same as the equation

$$\mathcal{E} \equiv \bigwedge_{a \in Act} [a\tau] \langle \tau a \rangle \mathcal{E}$$

in the transition system $\tilde{M}$ (see the semantics of modalities in Section 2.1, and definition of $\tilde{M}$). Therefore, $\mathcal{E}$ is a simulation relation iff it is a fixpoint of the above equation. We show that $\preceq$ is the greatest fixpoint. Let $\mathcal{Y}$ denote the set $\nu X. \Big(\bigwedge_{a \in Act} [a\tau] \langle \tau a \rangle X \Big)$.

$\subseteq$: $s \preceq s'$ implies that $(s, s') \in \mathcal{E}$ for some simulation relation $\mathcal{E}$. Since $\mathcal{E}$ is a fixpoint of the equation, we have $\mathcal{E} \subseteq \mathcal{Y}$ by definition of the greatest fixpoint, therefore $(s, s') \in \mathcal{Y}$.

$\supseteq$: Let $(s, s') \in \mathcal{Y}$. Since $\mathcal{Y}$ satisfies the fixpoint equation, it is a simulation relation, hence $s \preceq s'$.

□

Theorem 2.7.2 Two states s and s' are bisimilar ($s \sim s'$) iff the following formula holds in the state (s, s') of the model $\tilde{M}$:

$$\nu X.\Big(\bigwedge_{a \in Act} [a\tau] \langle \tau a \rangle X \wedge [\tau a] \langle a\tau \rangle X \Big)$$

The proof of this theorem is almost identical to the proof of the previous theorem.

Obviously, the alternation depth of the formulas is one, therefore the complexity is $O(|\tilde{S}|$ iterations, where the size of $\tilde{S}$ is $|S|^2$. The time complexity is $O(|\tilde{S}||\tilde{Act}||\tilde{M}|)$ which is equal to $O(|S|^2|Act|^2|M||M'|)$. An algorithm for bisimulation equivalence with time complexity $O(|Act|(|T|+ |T'|)\log(|S|))$ is given in [59]. However, it is not clear if this algorithm can be modified to compute the simulation preorder or if it can be adapted to use OBDDs.

2.7.3 Weak simulation and bisimulation.

Weak simulation preorder and weak bisimulation equivalence require a more elaborate encoding. The definition of weak (bi)simulation is similar to (bi)simulation. The difference is that each of the transition systems is allowed to perform an unbounded but finite number of invisible actions ε. Formally, we first define a relation $\Rightarrow$ by

$$s \stackrel{a}{\Rightarrow} q \text{ iff } \exists s_1, s_2 . s \stackrel{\varepsilon^*}{\rightarrow} s_1 \stackrel{a}{\rightarrow} s_2 \stackrel{\varepsilon^*}{\rightarrow} q,$$

and $s \stackrel{\varepsilon^*}{\rightarrow} q$ means that q is reachable from s by 0 or more executions of ε.

Definition 2.7.3 A relation $\mathcal{E} \subseteq S \times S$ is called a *weak simulation* with invisible action ε, when $(s, s') \in \mathcal{E}$ iff

$$\forall a \in Act. \forall q \in S. \text{ if } s \stackrel{a}{\Rightarrow} q \text{ then } \exists q' \in S. s' \stackrel{a}{\Rightarrow} q' \text{ and } (q, q') \in \mathcal{E}.$$

Definition 2.7.4 A relation $\mathcal{E} \subseteq S \times S$ is called a *weak bisimulation* with invisible action ε, if $(s, s') \in \mathcal{E}$ iff

$$\begin{array}{ll} (i) & \forall a \in Act. \forall q \in S. \text{ if } s \stackrel{a}{\Rightarrow} q \text{ then } \exists q' \in S. s' \stackrel{a}{\Rightarrow} q' \text{ and } (q, q') \in \mathcal{E}; \\ (ii) & \forall a \in Act. \forall q' \in S. \text{ if } s' \stackrel{a}{\Rightarrow} q' \text{ then } \exists q \in S. s \stackrel{a}{\Rightarrow} q \text{ and } (q, q') \in \mathcal{E}. \end{array}$$

As before, we introduce a preorder called *weak simulation preorder*:

$s \preceq\!\!\!\prec s'$ iff there exists a weak simulation relation $\mathcal{E}$ such that $(s, s') \in \mathcal{E}$,

and an equivalence called *weak bisimulation equivalence*:

$s \approx s'$ iff there exists a weak bisimulation relation $\mathcal{E}$ such that $(s, s') \in \mathcal{E}$.

To encode the weak (bi)simulation in the propositional μ-calculus we again make use of the transition system $\tilde{M}$. Define the abbreviations:

$$\begin{array}{l} \langle \varepsilon^*; a; \varepsilon^* \rangle \phi \equiv_{df} \mu X.(\langle a \rangle (\mu Y.\phi \vee \langle \varepsilon \rangle Y) \vee \langle \varepsilon \rangle X) \\ [\varepsilon^*; a; \varepsilon^*]\, \phi \equiv_{df} \neg \langle \varepsilon^*; a; \varepsilon^* \rangle \neg \phi \end{array}$$

To understand the formulas better, notice that informally thay can be viewed as translations of $\mathbf{EF}(\langle a \rangle\, \mathbf{EF}\, \phi)$ and $\mathbf{AG}([a]\, \mathbf{AG}\, \phi)$, where CTL operators refer to ε-paths. Now, it is straightforward to show that the following theorems hold:

Theorem 2.7.3 Let s and s' be states of the two transition systems. Then $s \preceq\!\!\!\prec s'$ iff the following formula holds in the state (s, s') of the transition system $\tilde{M}$:

$$\nu X. \Big(\bigwedge_{a \in Act} [(\varepsilon\tau)^*; a\tau; (\varepsilon\tau)^*]\, \langle (\tau\varepsilon)^*; \tau a; (\tau\varepsilon)^* \rangle X \Big)$$

Theorem 2.7.4 Two states s and s' are weakly bisimilar ($s \approx s'$) iff the following formula holds in the state (s, s') of the model $\tilde{M}$:

$$\nu X. \Big(\bigwedge_{a \in Act} \begin{array}{l} [(\varepsilon\tau)^*; a\tau; (\varepsilon\tau)^*]\, \langle (\tau\varepsilon)^*; \tau a; (\tau\varepsilon)^* \rangle X \wedge \\ [(\tau\varepsilon)^*; \tau a; (\tau\varepsilon)^*]\, \langle (\varepsilon\tau)^*; a\tau; (\varepsilon\tau)^* \rangle X \end{array} \Big)$$

Although there are five levels of nesting in these formulas, the alternation depth is only two. Therefore, we can compute it by the algorithm given in [28] using $O((|\tilde{S}||\tilde{Act}|)^2)$ iterations or $O(|\tilde{Act}|^3|\tilde{M}||\tilde{(S)}|^2)$ time. Recall that each iteration can take upto $O(|\tilde{Act}||\tilde{M}|)$ time. However, there is another algorithm by H. Andersen [3] that can compute the fixpoints in $O(|Act|^2|S|^2|\tilde{M}|)$ time, which is generally better, since the number of states in a model is usually less than the number of transitions. The algorithm in [59] can also be adapted to compute weak bisimulation equivalence by precomputing the transitive closure of the ε relation. However, the expense of this step dominates the cost of the entire computation. Again, it is not clear that OBDDs can be used in the last two algorithms.

Chapter 3

Symmetry I

Finite state concurrent systems frequently exhibit considerable symmetry. It is possible to find symmetry in memories, caches, register files, bus protocols, network protocols – anything that has a lot of replicated structure. Generally, verification techniques do not take advantage of this fact. In this chapter we explain how to exploit symmetry to reduce the size of the state space that must be explored by temporal logic model checking algorithms.

A symmetry group G of a Kripke Structure $M = (S, R, L)$ induces an equivalence relation on the state space S. The quotient model M_G is obtained by picking one state out of each equivalence class. If the property being verified is also invariant under the group G, then f can be checked on the smaller quotient model M_G.

This chapter is organized as follows: Section 3.1 introduces some preliminary definitions and introduces the idea of symmetry group of a Kripke Structure. Section 3.3 presents a model-checking algorithm which exploits the inherent symmetries of the system. In Section 3.4 we explore the complexity of determining whether two states are equivalent under the action of a group. The complexity of this problem is also discussed in great detail in Chapter 4. Lower bounds on BDD for an equivalence relation induced by certain symmetry groups are provided in Section 3.5. Section 3.6 provides a method which works well in conjunction with BDDs. Empirical results are provided in Section 3.7. Comparison of coarsest bisimulation reduction and symmetry based reductions techniques are provided in Section 3.8. Section 3.9 discusses some related work.

3.1 Symmetry Groups

This section introduces the notion of symmetry of a Kripke Structure. We also introduce some preliminary group theoretic concepts which will be used throughout the chapter.

3.1.1 Basic Group Theory

A *group* G is a set S endowed with an associative binary operation "$\cdot$" with the following properties:

1. There exists an identity element $1 \in S$ such that for all $s \in S$, $s \cdot 1 = 1 \cdot = s$.

2. Each element $s \in S$ has an unique inverse s^{-1} such that $s \cdot s^{-1} = s^{-1} \cdot s = 1$.

The number of elements in S is called the *order* of the group. The symbol $[n]$ denotes the set $\{1, 2, \cdots, n\}$. S_n is the group of all permutations of the set $[n]$. A *cycle* of length k (denoted by $(i_0, \cdots, i_{k-1})$) is a permutation σ such that $\sigma(i_j) = i_{j+1 \pmod k}$. A *rotation group* (or *cyclic group* C_n) acting on the set $[n]$ is a permutation group generated by the cycle $(1, 2, \cdots, n)$. $Sym(X)$ denotes the group of all permutations on the set X. A group G is said to *act* on the set S if there exists a map $(g, x) \to gx$ from $G \times S$ into S satisfying

1. $1(x) = x$ for all $x \in S$.

2. $(g_1 g_2)(x) = g_1(g_2(x))$.

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

- S is a finite set of *states*,

- $R \subseteq S \times S$ is a *transition relation*, which must be total (i.e., for every state s_1 there exists a state s_2 such that $(s_1, s_2) \in R$).

- $L : S \to 2^{AP}$ is a *labeling function* that associates with each state a set of atomic propositions that are true in the state.

Let G be a permutation group, i.e., a set of bijective mappings acting on the state space S of the Kripke structure M. A permutation $\sigma \in G$ is said to be a *symmetry* of M if and only if it preserves the transition relation R. More formally, σ should satisfy the following condition:

$$(\forall s_1 \in S)(\forall s_2 \in S)\,((s_1, s_2) \in R \;\Rightarrow\; (\sigma s_1, \sigma s_2) \in R)$$

G is a *symmetry group* for the Kripke structure M if and only if every permutation $\sigma \in G$ is a *symmetry* of M. Notice that the definition of a symmetry group does not refer to the labeling function L. Furthermore, since every $\sigma \in G$ has an inverse, which is also a symmetry, it can be easily proved that a permutation $\sigma \in G$ is a symmetry for a Kripke structure if and only if σ satisfies the following condition:

$$(\forall s_1 \in S)(\forall s_2 \in S)\,((s_1, s_2) \in R \;\Leftrightarrow\; (\sigma s_1, \sigma s_2) \in R)$$

Example 3.1.1 The transposition $\sigma = (S_1, S_2)$ exchanges the states S_1 and S_2 in the Kripke Structure shown in figure 3.1. The states S_0 and S_3 are not affected by the permutation σ, so the successors of all the states remain the same when σ is applied. Hence, σ is a symmetry of the Kripke Structure.

Let $\langle g_1, \ldots, g_k \rangle$ be the smallest permutation group containing all the permutations $g_1, \ldots, g_k$. If $G = \langle g_1, \ldots, g_k \rangle$, then we say that the group G is *generated* by the set $\{g_1, \ldots, g_k\}$. It is easy to see that if every generator of the group G is a symmetry of M, then the group G is a symmetry group for M.

3.2 Quotient Models

Let G be a group acting on the set S and let s be an element of S. The *orbit* of s is the set $\theta(s) = \{t \mid (\exists \sigma \in G)(\sigma s = t)\}$; From each orbit $\theta(s)$ we pick a representative, which we call $rep(\theta(s))$ with the restriction that $rep(\theta(s)) \in \theta(s)$.

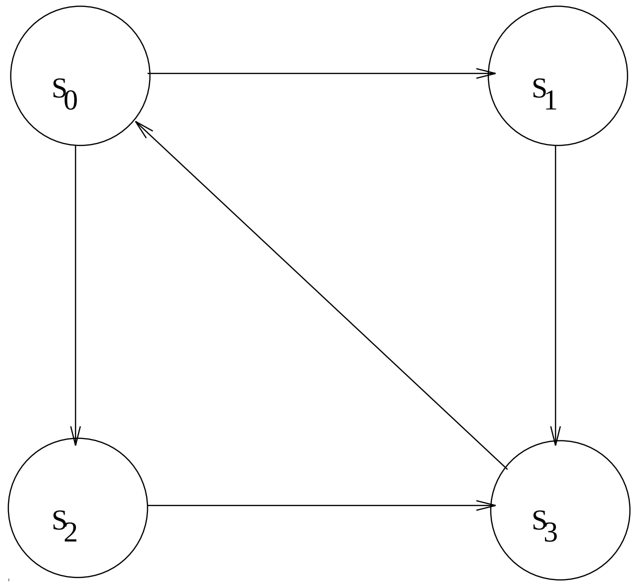

Figure 3.1: A Kripke Structure

Definition 3.2.1 Let $M = (S, R, L)$ be a Kripke Structure and let G be a symmetry group acting on M. We define the *quotient structure* $M_G = (S_G, R_G, L_G)$ in the following manner:

- The state set is $S_G = \{\theta(s)|s \in S\}$, the set of orbits of the states in S;
- The transition relation R_G is given by
$$R_G = \{(\theta(s_1), \theta(s_2))|(s_1, s_2) \in R\}; \tag{3.1}$$
- The labeling function L_G is given by $L_G(\theta(s)) = L(rep(\theta(s)))$.

Next, we define what it means for a symmetry group G of a Kripke Structure M to be an *invariance group* for an atomic proposition p. Intuitively, G is an invariance group for an atomic proposition p if and only if the set of states labeled by p is closed under the application of all the permutations of G. More formally, a symmetry group G of a Kripke Structure $M = (S, R, L)$ is an invariance group for an atomic proposition p if and only if the following condition holds:

$$(\forall \sigma \in G)(\forall s \in S)(p \in L(s) \Leftrightarrow p \in L(\sigma s))$$

Lemma 3.2.1 If G is an invariance group for an atomic proposition p and $p \in L(s)$, then $p \in L_G(\theta(s))$ in the quotient Kripke structure M_G.

Proof: Let $s_1 = rep(\theta(s))$. By the definition of orbit $s_1 = \sigma s$ for some $\sigma \in G$. If $p \in L(s)$, then $p \in L(\sigma s)$ because G is an invariance group for p. Therefore $p \in L(s_1)$, and since $L_G(\theta(s)) = L(s_1)$ we have that $p \in L_G(\theta(s))$. □

Definition 3.2.2 Given a Kripke structure $M = (S, R, L)$ and a symmetry group G, let $M_G = (S_G, R_G, L_G)$ be the quotient Kripke structure. Two paths $\pi = s_0, s_1, \ldots$ in M and $\pi_G = \theta(t_0), \theta(t_1), \ldots$ *correspond* if and only if $\forall i (s_i \in \theta(t_i))$.

Lemma 3.2.2 For every path starting from s_0 in M there exists a corresponding path starting from $\theta(s_0)$ in M_G, and for every path starting from $\theta(s_0)$ in M_G there exists a corresponding path starting from s_0 in M.

Proof:
($\Rightarrow$) Let $\pi = s_0, s_1, \ldots$ be a path in M. The corresponding path in M_G is the path produced by taking the orbits of the states, or $\pi_G = \theta(s_0), \theta(s_1), \ldots$. Notice that π_G is a valid path in the quotient structure M_G because $(s_i, s_{i+1}) \in R$ implies that $(\theta(s_i), \theta(s_{i+1})) \in R_G$.

($\Leftarrow$) Let $\pi_G = \theta(s_0), \theta(s_1), \ldots$ be a path in M_G. We show how to construct a path $\pi = t_0, t_1, \ldots$ in M such that $t_0 = s_0$ and $t_i \in \theta(s_i)$. We construct the path in an inductive manner. Initially, we let $t_0 = s_0$, and we maintain the invariant that $t_i \in \theta(s_i)$. Since $(\theta(s_0), \theta(s_1)) \in R_G$ there exists $u \in \theta(s_0)$ and $v \in \theta(s_1)$ such that $(u, v) \in R$. Since $u \in \theta(s_0)$ there exists a $\sigma \in G$ such that $s_0 = \sigma u$. By the definition of the symmetry group $(\sigma u, \sigma v) \in R$, or $(s_0, \sigma v) \in R$. Let $t_1 = \sigma v$. Notice that since $v \in \theta(s_1)$, we have that $t_1 \in \theta(s_1)$. Assume that we have constructed a path π up to t_k such that $t_k \in \theta(s_k)$. Using an argument similar to one given above we can find $t_{k+1} \in \theta(s_{k+1})$ such that $(t_k, t_{k+1}) \in R$. □

Intuitively, the theorem given below states that for a temporal formula f with invariant propositions it is safe to check the formula f on quotient model.

Theorem 3.2.1 Let $M = (S, R, L)$ be a Kripke Structure, G be a *symmetry group* of M, and h be a $CTL^{\star}$ formula. If G is an invariance group for all the atomic propositions p occurring in h, then

$$M, s \models h \Leftrightarrow M_G, \theta(s) \models h \tag{3.2}$$

where M_G is the quotient structure corresponding to M.

This theorem is a direct consequence of the following lemma.

Lemma 3.2.3 Let h be a either a state formula or a path formula such that G is an invariance group for all atomic propositions p occurring in h. Let $\pi = s, s_1, \ldots$ be a path in M and $\pi_G = \theta(s), \theta(t_1), \ldots$ be a corresponding path in M_G. Then

- $M, s \models h \Leftrightarrow M_G, \theta(s) \models h$ if h is a state formula, and
- $M, \pi \models h \Leftrightarrow M_G, \pi_G \models h$ if h is a path formula.

Proof: This proof is similar to the proof of Lemma 3.2 given in [9], and proceeds by structural induction.

Basis: $h \in AP$. Because G is an invariance group for h, it is easy to see by Lemma 3.2.1 and the definition of an invariance group that $M, s \models h \Leftrightarrow M_G, \theta(s) \models h$.

Induction: There are several cases.

- $h = \neg h_1$, a state formula.
  By the inductive hypothesis we have that $M, s \models h_1 \Leftrightarrow M_G, \theta(s) \models h_1$. Therefore $M, s \models h \Leftrightarrow M_G, \theta(s) \models h$. The same reasoning holds if h is a path formula.
- $h = h_1 \vee h_2$, a state formula

$$\begin{aligned} M, s \models h &\Leftrightarrow M, s \models h_1 \text{ or } M, s \models h_2 \\ &\Leftrightarrow M_G, \theta(s) \models h_1 \text{ or } M_G, \theta(s) \models h_2 \\ &\Leftrightarrow M_G, \theta(s) \models h \end{aligned}$$

  The second step uses the inductive hypothesis. We can also use this argument if h is a path formula.

- $h = E(h_1)$, a state formula
  Suppose $M, s \models h$. There is a path π starting with s such that $M, \pi \models h_1$. By Lemma 3.2.2 there is a corresponding path π_G in M_G starting with $\theta(s)$. By the inductive hypothesis $M, \pi \models h_1 \Leftrightarrow M_G, \pi_G \models h_1$. Therefore, $M, s \models E(h_1) \Rightarrow M_G, \theta(s) \models E(h_1)$. A similar argument holds in the other direction.

- $h = h_1$, where h is a path formula and h_1 is a state formula.
  Although the lengths of h and h_1 are the same, we can imagine that $h = path(h_1)$, where *path* is an operator which converts a state formula into a path formula. Now we can apply the inductive step.

- $h = Xh_1$, a path formula.
  By the definition of the next time operator $M, \pi^1 \models h_1$. Since π and π_G correspond, so do π^1 and π_G^1. Therefore, by the inductive hypothesis, $M_G, \pi_G^1 \models h_1$, so $M_G, \pi_G \models h$. A similar argument proves the implication in the other direction.

- $h = h_1 U h_2$, a path formula.
  Suppose that $M, \pi \models h_1 U h_2$. By the definition of the *until* operator, there is a k such that $M, \pi^k \models h_2$ and for all $0 \leq j < k$, $M, \pi^j \models h_1$. Since π and π_G correspond, so do π^j and π_G^j for any j. Therefore, by inductive hypothesis $M_G, \pi_G^k \models h_2$ and $M_G, \pi_G^j \models h_2$ for all $0 \leq j < k$. Therefore, we have $M_G, \pi_G \models h$. We can use the same argument in the other direction. $\Box$.

3.3 Model Checking with Symmetry

In this section we describe how to perform model checking in the presence of symmetry. We discuss how to find the set of states in a Kripke structure that are reachable from a given set of initial states using an explicit state representation. In the explicit state case, a breadth-first or depth-first search starting from the set of initial states is performed. Typically, two lists, a list of reached states and a list of unexplored states are maintained. At the beginning of the algorithm, the initial states are put on both the lists. In the exploration step, a state is removed from the list of unexplored states and all its successors are processed. An algorithm for exploring the state space of a Kripke structure in the presence of symmetry is discussed in [42]. The authors introduce a function $\xi(q)$, which maps a state q to the unique state representing the orbit of that state. While exploring the state space, only the unique representatives from the orbits are put on the list of reached and unexplored states. Figure 3.2 gives the pseudo-code for exploring the state space under the presence of symmetry. This simple reachability algorithm can be extended to a full CTL model checking algorithm by using the technique described in [17].

$Reached = \emptyset$;
$Unexplored = \emptyset$;
For each initial state s Do
    Append $\xi(s)$ to $Reached$;
    Append $\xi(s)$ to $Unexplored$;
Endforloop
While $Unexplored \neq \emptyset$ Do
    Remove a state s from $Unexplored$;
    For each succesor state q of s
        If $\xi(q)$ is not in $Reached$
            Append $\xi(q)$ to $Reached$;
            Append $\xi(q)$ to $Unexplored$;
        EndIf
    Endforloop
EndWhile

Figure 3.2: Algorithm for exploring state space in presence of symmetry

We can now focus on how to perform symbolic model checking in the presence of symmetries. The straightforward method of computing the quotient model uses the OBDD for the orbit relation $\Theta(x, y) \equiv (x \in \theta(y))$. Given a Kripke structure $M = (S, R, L)$ and a symmetry group G on M with r generators $g_1, g_2, \cdots, g_r$, the orbit relation Θ can be defined as the least fixpoint of the equation given below:

$$Y(x, y) \equiv (x = y) \vee (\exists z)(Y(x, z) \wedge (z = g_1 y \vee z = g_2 y \cdots \vee z = g_r y)) \tag{3.3}$$

The following lemma shows that this definition is correct.

Lemma 3.3.1 The least fixpoint of Equation 3.3 is the orbit relation Θ induced by the group G generated by $g_1, g_2, \cdots, g_r$.

Proof: First, we prove that Θ is a fixpoint of the equation given below:

$$Y(x, y) \;\; = \;\; (x = y \vee (\exists z)(Y(x, z) \wedge (z = g_1 y \vee z = g_2 y \cdots \vee z = g_r y)))$$

It is obvious by the transitivity and reflexivity of the orbit relation Θ that

$$\Theta(x, y) \;\; \supseteq \;\; (x = y \vee (\exists z)(\Theta(x, z) \wedge (z = g_1 y \vee z = g_2 y \cdots \vee z = g_r y))).$$

Suppose $\Theta(x, y)$ is true, then by the definition of the orbit relation there exists $\sigma \in G$ such that $x = \sigma y$. Without loss of generality assume that $x \neq y$. This means there exists a generator $g_k, k \leq r$ such that $x = \sigma_1 g_k y$. Setting $z = g_k y$, we see that $\Theta(x, z)$ and $z = g_k y$. Since x and y are arbitrary boolean vectors we get the following inclusion:

$$\Theta(x, y) \;\; \subseteq \;\; (x = y \vee (\exists z)(\Theta(x, z) \wedge (z = g_1 y \vee z = g_2 y \cdots \vee z = g_r y)))$$

Hence Θ is a fixpoint of Equation 3.3.

Next, we prove that if T is any fixpoint of equation 3.3, then $\Theta \subseteq T$ by showing that $\Theta(x, y) \Rightarrow T(x, y)$. The definition of the orbit relation $\Theta(x, y)$ implies that there exists a $\sigma = g_{i_m} \cdots g_{i_2} g_{i_1}, 1 \leq i_j \leq r$ such that $x = \sigma y$. Since T is a fixed point of Equation 3.3, it can be proved by induction that for all $1 \leq l \leq m$, $T(g_{i_l} \ldots g_{i_1} y, y)$ holds. Using this result for $l = m$, we see that $T(x, y)$ holds. Since $\Theta(x, y) \Rightarrow T(x, y)$, we obtain that $\Theta \subseteq T$. Hence, Θ is the least fixpoint. $\Box$

If a suitable state encoding is available, this fixpoint equation can be computed using OBDDs [13]. Once we have the orbit relation Θ, we need to compute a function $\xi : S \rightarrow S$, which maps each state s to the unique representative in its orbit. If we view states as vectors of values associated with the state variables, it is possible to choose the lexicographically smallest state to be the unique representative of the orbit. Since Θ is an equivalence relation, these unique representatives can be computed using OBDDs by the method of Lin [49].

Definition 3.3.1 Let $B = \{0, 1\}$ and $\mathcal{R} \subseteq B^r \times B^n$ be a total relation. A function $\mathcal{F} : B^r \to B^n$ is *compatible* with $\mathcal{R}$ if it has the following properties:

- For all $x \in B^r$, we have $(x, \mathcal{F}(x)) \in \mathcal{R}$;
- For every u and v in B^r, if the possible mappings of u and v are the same (i.e., $(\forall y \in B^n)((u, y) \in \mathcal{R} \Leftrightarrow (v, y) \in \mathcal{R})$) then $\mathcal{F}(u) = \mathcal{F}(v)$.

Lin [49] also defines a *compatible projection operator* which is a compatible function that maps $x \in B^r$ to the least $y \in B^n$ (with respect to some norm N) such that $(x, y) \in \mathcal{R}$. Formally, the compatible projection is defined as follows:

Definition 3.3.2 Let $\mathcal{R} \subseteq B^r \times B^n$ be a binary relation. The *projection* of $\mathcal{R}$, denoted by *projection*$(\mathcal{R})$, is the function $\mathcal{F}$ defined as follows:

$$\mathcal{F} = \{(x, y) | (x, y) \in \mathcal{R} \wedge (\forall z)((x, z) \in \mathcal{R} \Rightarrow N(z) \geq N(y))\}$$

where the norm $N(x)$ is defined as follows:

$$N(x) = \sum_{i=0}^{n} x_i 2^{n-i}$$

That is, $N(x)$ is the number whose binary representation is the vector x.

It can be shown that *projection*$(\mathcal{R})$ is a compatible function of $\mathcal{R}$. This function can be computed efficiently in a single bottom-up traversal of the OBDD representation of the relation $\mathcal{R}$. Given the orbit relation Θ, the function *projection*(Θ) maps each state to the unique representative of its orbit. Notice that *projection*(Θ) is exactly the function ξ introduced before.

Assuming that we have the OBDD representation of the mapping function ξ, the transition relation R_G of the quotient structure can be expressed as follows:

$$R_G(x,y) \quad = \quad (\xi(x) = x) \wedge (\exists y_1)(R(x,y_1) \wedge \xi(y_1) = y)$$

The formula $\xi(x) = x$ expresses the fact that x is the unique representative of its orbit. To construct the function $\xi(q)$ it is important to compute the orbit relation efficiently. In the next section we will discuss the computational complexity of finding the orbit relation.

3.4 Complexity of orbit calculations

The behavior of a sequential circuit or protocol is frequently determined by the values of a set of boolean state variables $x_1, x_2, \ldots, x_n$. For example, the behavior of a bus arbitration protocol may be determined by the boolean state variables that encode the command on the bus and the identity of the master. When we extract a Kripke structure from a circuit or protocol, we treat these state variables as atomic propositions. The set of atomic propositions is $AP = \{x_1, \cdots, x_n\}$. The resulting Kripke model $M = (S, R, L)$ will have the following components:

- $S \subseteq B^n$, where each state can be thought of as a truth assignment to the n state variables;
- $R \subseteq S \times S$, where R is determined by the behavior of the circuit or protocol;
- $L : S \to 2^{AP}$ where L is defined so that $x_i \in L(s)$ if and only if the i-th component of s is 1.

It is often the case that the symmetry group is also given in terms of the state variables. For example, in a two bit adder with inputs x_1, x_2 and x_3, x_4, the permutation (13)(24) is a symmetry because we can exchange the inputs without affecting the result. If we have a permutation σ, which acts on the set $\{1, 2, \ldots, n\}$, then σ acts on vectors in B^n in the following manner:

$$\sigma(x_1, x_2, \cdots, x_n) \quad = \quad (x_{\sigma(1)}, x_{\sigma(2)}, \cdots, x_{\sigma(n)}).$$

Given two vectors x and y in B^n and a permutation σ, it is easy to see that $x \neq y$ implies $\sigma x \neq \sigma y$. Therefore, a group G acting on the set $\{1, 2, \cdots, n\}$ induces a permutation group G_1 acting on the set B^n. In other words, a symmetry on the structure of a circuit induces a symmetry on the state space of the circuit.

Definition 3.4.1 Let G be a group acting on the set $\{1, 2, \cdots, n\}$. Assume that G is represented in terms of a finite set of generators. Given two vectors $x \in B^n$ and $y \in B^n$, the *orbit problem* asks whether there exists a permutation $\sigma \in G$ such that $y = \sigma x$.

Let G induce the permutation group G_1 acting on B^n. The orbit problem asks if x and y are in the same orbit under the action of the group G_1. As we show below, the *Orbit Problem* is as hard as the *Graph Isomorphism* problem.

Definition 3.4.2 Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ such that $|V_1| = |V_2|$, the *Graph Isomorphism problem* asks whether there exists a bijection $f : V_1 \rightarrow V_2$ such that the following condition holds

$$(i, j) \in E_1 \quad \Leftrightarrow \quad (f(i), f(j)) \in E_2$$

Theorem 3.4.1 The *orbit problem* is as hard as the *Graph Isomorphism* problem.

Proof:
Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ we construct a group G and two 0-1 vectors x and y such that x and y are in the same orbit under the action of the group G if and only if G_1 and G_2 are isomorphic. We assume that $|V_1| = |V_2| = n$ and are labeled by integers. Let $A = \{a_{ij}\}$ and $B = \{b_{ij}\}$ be the adjacency matrices of the graph G_1 and G_2 respectively. Let $x \in B^{n^2}$ be defined as follows:

$$x_{n(i-1)+j} \quad = \quad a_{ij} \, , 1 \leq i \leq n \, , 1 \leq j \leq n$$

The vector $x \in B^{n^2}$ is a list of the elements of the matrix A in row order. The vector $y \in B^{n^2}$ is defined in a similar fashion using the adjacency matrix B. Let (ij) be a transposition acting on the set

$\{1, 2, \cdots, n\}$. Intuitively, this transposition exchanges the vertices i and j in the graph G_1. This transposition corresponds to exchanging the rows i and j, and columns i and j, in the adjacency matrix and has exactly the same effect as applying the permutation σ given below to the vector x.

$$\begin{aligned}
\sigma_{row} &= (n(i-1)+1, n(j-1)+1)\ldots(n(i-1)+n, n(j-1)+n) \\
\sigma_{col} &= (i,j)\ldots((n-1)n+i,(n-1)n+j) \\
\sigma &= \sigma_{row}\sigma_{col}
\end{aligned}$$

Each permutation acting on the set of size n corresponds to a bijection $f : V_1 \to V_2$. We assume that the vertices are labeled by integers. If the bijection corresponding to the permutation (ij) is an isomorphism between G_1 and G_2, then exchanging rows i and j and columns i and j in the adjacency matrix A yields the matrix B. Thus $y = \sigma x$, because x and y are just encodings of the adjacency matrices A and B, respectively. Similarly, if $y = \sigma x$, then the bijection corresponding to the permutation (ij) is an isomorphism between the graph G_1 and G_2. Therefore, $y = \sigma x$ if and only if the bijection corresponding to the permutation (ij) is an isomorphism between G_1 and G_2. Every bijection $f : V_1 \to V_2$ corresponds to some permutation in the full symmetric group S_n. Since the group S_n acting on the set $\{1, 2, \cdots, n\}$ is generated by the transpositions (12),(13), ... $(1n)$ we have the result. We just have to code all these transpositions in the context of the 0-1 vectors x and y. $\Box$.

Example 3.4.1 Consider the two graphs G_1 and G_2 given in the figure 3.3. The vectors x and y given below encode the adjacency matrices of the graphs G_1 and G_2 respectively:

$$\begin{aligned}
x &= (011\,100\,100) \\
y &= (010\,101\,010)
\end{aligned}$$

The permutations are defined as follows:

$$\begin{aligned}
\sigma_{row} &= (1,4)(2,5)(3,6) \\
\sigma_{col} &= (1,2)(4,5)(7,8) \\
\sigma &= \sigma_{row}\sigma_{col}
\end{aligned}$$

Notice that $y = \sigma x$ and the bijection corresponding to the permutation $(1, 2)$ is an isomorphism between G_1 and G_2. The permutation σ_{row} corresponds to exchanging rows 1 and 2.

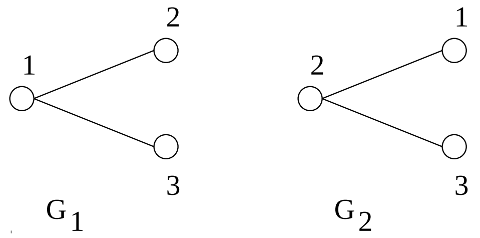

Figure 3.3: Two isomorphic graphs

A modified version of the *orbit problem* called the *bounded orbit problem* is defined as follows:

Definition 3.4.3 Given a group G generated by r permutations g_1,g_2, $\cdots$, g_r acting on the set $\{1, 2, \cdots, n\}$, two vectors $x, y \in B^n$, and an integer k, does there exist a permutation σ obtained by at most k applications of the generators such that $x = \sigma y$? That is, does σ have the following form $\sigma = g_{i_1} g_{i_2} \ldots g_{i_m}$, where $m \leq k$?.

Intuitively, in the *bounded orbit problem* we limit the number of times we can apply the generators. Although the graph isomorphism problem is not known to be *NP*-hard, the bounded orbit problem can be shown to be NP-complete. The reduction is from *EXACT COVER BY 3-SETS* [33].

Definition 3.4.4 The *EXACT COVER BY 3-SETS (X3C)* is defined as follows:
INSTANCE: Set X with $|X| = 3q$ and a collection C of 3-element subsets of X.
QUESTION: Does C contain an exact cover for X? That is, is there a subcollection $C' \subseteq C$ such that every element of X occurs in exactly one member of C'?

Theorem 3.4.2 The *bounded orbit problem* is NP-complete.

Proof: The reduction is from the $X3C$ problem. Consider an instance of $X3C$, where we are given a specific set X with $|X| = 3q$ and a collection C of 3-element subsets of X. We construct a group G acting on the set $\{1, 2, \cdots, 6q\}$. Let $n = 3q$. With an element $x_i \in X$ we associate the permutation $(i, n+i)$, $1 \leq i \leq n$. With each 3-element set in the collection C we associate a permutation which is the product of the permutation corresponding to its elements. For example, if we have the set $\{x_i, x_j, x_k\} \in C$, then the permutation associated with it is $(i, n+i)(j, n+j)(k, n+k)$. Let C_g be the set of permutations corresponding to the collection C. The group G is the group generated by the set of permutations C_g. Consider two 0-1 vectors x and y of length $6q$ defined as follows: x has 1s in first $3q$ positions and 0s in last $3q$ positions, and y has 0s in first $3q$ positions and 1s in last $3q$ positions. We will show that there exists a permutation σ, which is the product of q generators from the set C_g, such that $y = \sigma x$ if and only if the instance of $X3C$ has a cover C' for the set X.

Because we are dealing with 3-element subsets and X has $3q$ elements, we must use exactly q 3-element sets to cover X. Suppose there is a collection C' of q sets that covers X, and consider the permutation σ that is the product of all the permutations corresponding to the sets in the cover C'. Since C' is a cover for X, the transposition $(i, n+i)$ for each element x_i occurs in the permutation σ. Hence, it is obvious that $y = \sigma x$.

Suppose, on the other hand, there exists a permutation σ, which is the product of at most q generators from the set C_g, such that $y = \sigma x$. Let us assume that $\sigma = g_1 \ldots g_r$ where $r \leq q$ and $g_i \in C_g$ for $1 \leq i \leq r$. Since $x_i = 1$ and $y_{n+i} = 1$(for $1 \leq i \leq 3q$), the permutation σ has to include each transposition of the form $(i, n+i)$ for $1 \leq i \leq 3q$. Since we need at least $3q$ transpositions of the form $(i, n+i)$ to transform the vector x to y, we will have to use exactly q generators. Moreover, the q generators must be disjoint. It follows that the collection formed by the 3-element sets corresponding to the q generators of σ is a cover for X. So the instance of the bounded orbit problem with the set of generators C_g, 0-1 vectors x and y, and the integer bound q has a solution if and only if the instance of $X3C$ has solution.

The bounded orbit problem is obviously in NP because we can guess the string of k (the given integer) generators that generates the permutation σ such that $y = \sigma x$, where x and y are the specified 0-1 vectors $\Box$.

Example 3.4.2 Consider the following instance of $X3C$ with $q = 2$. The various sets in this instance of $X3C$ are defined as follows:

$$\begin{aligned} X &= \{x_1, x_2, x_3, x_4, x_5, x_6\} \\ C &= \{\{x_1, x_3, x_5\}, \{x_2, x_4, x_6\}, \{x_1, x_2, x_6\}\} \end{aligned}$$

Now we define the instance of the bounded orbit problem. With each element x_i, $1 \leq i \leq 6$ we associate the permutation $(i, 6+i)$. With the collection C we associate the following set C_g of permutations:

$$\begin{aligned} C_g = \; & \{(1, 6+1)(3, 6+3)(5, 6+5), (2, 6+2)(4, 6+4)(6, 6+6), \\ & (1, 6+1)(2, 6+2)(6, 6+6)\} \end{aligned}$$

For example, the permutation $(1, 6+1)(3, 6+3)(5, 6+5)$ corresponds to the set $\{x_1, x_3, x_5\}$. Consider the following vectors x and y in B^{12}:

$$\begin{aligned} x &= (111111000000) \\ y &= (000000111111) \end{aligned}$$

Notice that $\{\{x_1, x_3, x_5\}, \{x_2, x_4, x_6\}\}$ is a cover for x, and the permutation σ which is the product of the permutations corresponding to these sets satisfies the equation: $y = \sigma x$.

At first glance, the bounded orbit problem might seem much weaker than the general orbit problem. However, given a permutation group G with an arbitrary set of generators, we can always find a set of generators such that every permutation $\sigma \in G$ is the product of at most n of the new generators, where n is the size of the set on which G acts. This result follows immediately using an algorithm described in [32]. Given a group G acting on a set of size n and generated by $g_1, \cdots, g_k$, we can construct a table T with n rows (labeled 0 to $n-1$) and n columns (labeled 1 to n), of permutations with the following property: $\sigma \in G$ if and only if σ can be expressed as $a_0 a_1 \cdots a_r$, were a_i is a member of the i-th row. Using the permutations in the table T, the general orbit problem on G can be converted into an instance of the bounded orbit problem (with the bound n).

3.5 Complexity of the OBDD for the orbit relation

The use of OBDD has proved quite successful in model checking [14, 55]. In [20] it was proved that the OBDD for the orbit relation is exponential for transitive groups. A method using multiple representatives which avoids building the orbit OBDD was also provided in [20]. In this section, we expand the theorem given in [20] to the class of *separable* groups. Because of lemma 3.5.1 the result given in [20] is a special case of the theorem given here.

Definition 3.5.1 Let G be a group acting on a set S. Two subsets C and D of S are said to be *separable* iff there exists a permutation $\sigma \in G$ such that $\sigma(C)$ and D are disjoint. The group is said to be *k-separated* if and only if any sets C and D such that $|C| = k$ and $|D| < k$ are separable.

Definition 3.5.2 A group of G permutations acting on a set S is called *transitive*, if and only if for all $i, j \in S$ there exists $g \in G$ such that $g(i) = j$.

We have the following lemma.

Lemma 3.5.1 A transitive group $G \leq S_n$ is $\sqrt{n}$-separable. The full symmetric group S_n is $\frac{n}{2}$-separable.

Proof: The result for transitive groups is given in [6]. Given two subsets C and D of $[n]$ such that $|C| = \frac{n}{2}$ and $|D| \leq \frac{n}{2}$, one can always find a permutation $\sigma \in S_n$ such that $\sigma(C) = [n] - D$. The result follows from definition of separability.□

A special case of the above situation is the following: the set $S = B^{nm}$ (i.e., a system composed of n components each with m state variables) and the group G acting on S is induced by a group G' of permutations acting on the $[n]$ in the following way:

$$g(\langle x_{1,1}, \ldots, x_{1,m}, \ldots, x_{n,1}, \ldots, x_{n,m} \rangle) = \langle x_{g'(1),1}, \ldots, x_{g'(1),m}, \ldots, x_{g'(n),1}, \ldots, x_{g'(n),m} \rangle$$

with $g \in G$ and $g' \in G'$. Intuitively, the group G' tells us how to permute blocks where each block has m variables. Ring structures, where the components are ordered in a ring and can be rotated any number of steps, occur frequently in practice. The token ring protocol used in the solution to the distributed mutual exclusion problem uses this topology. In the terminology given above the symmetry group for the ring structure is induced by the rotation group. In star or bus topologies components are unordered and can be exchanged arbitrarily. Such situations occur, for example, in systems where components communicate via a common bus (e.g. multiprocessor systems), or in systems with broadcast and star-like communication structures. The symmetry group of these systems is induced by the full symmetric group, i.e., $G' = S_n$.

Definition 3.5.3 The orbit relation Θ of G is the set of pairs $\{\langle s,t\rangle \mid t \in \theta(s)\}$. The orbit relation *induced* by G' is the orbit relation of group G. If $S = B^{nm}$ the orbit relation can be represented by a characteristic boolean function $\Theta : B^{nm} \times B^{nm} \to B$.

Theorem 3.5.1 Let $S = B^{nm}$. For a d-separated group G acting on the set $\{1, 2, \cdots, n\}$ we can obtain the following lower bound for the OBDD representing the induced orbit relation Θ:

$$|\Theta| > 2^K \ with \ K = min(d, 2^{m-1} - 1)$$

Proof: We use unprimed variables $x_{i,j}$ for representing the first argument of the orbit relation and primed variables $x'_{i,j}$ for the second. For the proof we only consider the first variable of each component. First we determine a partition (L, R) of the variables — we go through all the variables in the given variable ordering, until we have K unprimed variables $x_{i,1}$ or K primed variables $x'_{i,1}$ in L and put the rest of the variables in R. Notice that all variables in L precede all the variables in R in the variable ordering. Without loss of generality, we assume that L contains K unprimed variables with indices $I = \{i_1, ..., i_K\}$ and less than K primed variables $x'_{j,1}$ with indices $j \in J$.

Since G is d-seperated, we can find a $g \in G$, such that all primed variables $x'_{g(i),1}$ for $i \in I$ are in R. In other words, $g(I)$ and J are disjoint. Notice that $K \leq d$. We construct an assignment in the following way: for $i_j \in I$, the variables $\langle x_{i_j,2}, ..., x_{i_j,m}\rangle$ and $\langle x'_{g(i_j),2}, ..., x'_{g(i_j),m}\rangle$ are instantiated with the binary representation of the number $\#j$ for

$1 \le j \le K$. Note that since K has to be representable in $m-1$ bits, we need that $K \le 2^{m-1} - 1$. The variables $x_{i,j}$ and $x'_{g(i),j}$ are instantiated with 0 for $i \notin I$. Let Θ' be the OBDD obtained from Θ by the above instantiation. Instantiating variables with constant values in a OBDD does not increase its size, so $|\Theta'| \le |\Theta|$. The OBDD Θ' only depends on the variables $x_{i,1}$ and $x'_{g(i),1}$ for $i \in I$ and by our construction the only valid assignments are those, where the values of $x_{i,1}$ and $x'_{g(i),1}$ agree. In other words, Θ' is the OBDD for the following proposition:

$$\bigwedge_{i \in I} x_{i,1} = x'_{g(i),1}$$

Since all variable $x_{i,1}$ precede all variables $x'_{g(i),1}$, the OBDD Θ' has atleast 2^K nodes (see Lemma 2.3.1.$\Box$

Notice that from lemma 3.5.1 and the previous theorem one gets exponential lower bounds for the orbit OBDD for transitive and full-symmetric groups.

3.6 Multiple Representatives

Since the OBDD for the orbit relation is large for some groups that occur frequently in practice, it is computationally expensive to use the single representative theory described earlier. For example, Lin's algorithm to extract an unique representative from each orbit requires the explicit construction of the OBDD for the orbit relation. In this section, we develop a scheme to use multiple representatives. In this scheme the size of OBDDs remain more manageable, because the orbit relation for the symmetry group is never explicitly constructed. Next, we explain the main idea behind the algorithm using multiple representatives. Consider n processes $P_1, \cdots, P_n$ contending for the bus. Assume that we can exchange processes arbitrarily. Consider a state s in which the control of the bus is granted to P_j $(j \ge 2)$. By permuting processes P_1 and P_j we can map state s into a state s' where P_1 has the control of the bus. This should allow us to always work in the restricted set of states where P_1 always has control of the bus.

First, we explain the concept of cosets which will be used in this section.

Definition 3.6.1 Let G be a group. Let $H \leq G$. Given two elements $a, b \in G$, we say that $a \equiv_r b$ if and only if $ab^{-1} \in H$. In a similar manner, $a \equiv_l b$ if and only if $a^{-1}b \in H$. It is easy to see that $\equiv_r$ and $\equiv_l$ are equivalence relations on the elements of G. The equivalence classes induced by $\equiv_r$ are called *right cosets* of G under H. Similarly, the equivalence classes induced by $\equiv_l$ are called *left cosets* of G under H.

Notice that Ha denotes the right coset containing the element a. aH denotes the left coset containing the element a. $[G : H]$ denotes the number of right cosets (which is equal to the number of left cosets [39]) of G under H. Given G and $H \leq G$, a *right traversal* of H in G is a set of $k = [G : H]$ elements $\{a_1, \cdots, a_k\}$ such that $G = \cup_{i=1}^{k} Ha_i$. Intuitively a right traversal has a representative from each right coset. *Left traversal* is defined in a similar manner using left cosets. A subgroup N of H is called a *normal subgroup* if and only if for all $a \in G$, $aNa^{-1} = N$. Given a normal subgroup $N \leq G$, the right and left cosets match, i.e., for all $a \in G$, $aN = Na$. Morover, the set of all right or left cosets, denoted by $G \backslash N$, is a group of order $[G : N]$ under the binary operation $(aN)(bN) = (abN)$(see [39]). A group G is called *simple* if and only if G has no proper normal subgroups. For an excellent treatment of group theoretic concepts see [39, 43].

Let Rep be the set of representatives. Let $\xi \subseteq S \times Rep$ be the representative relation. We are assuming the most general setting where there can be multiple representatives from one orbit and a state can be related by ξ to more than one representative from its orbit. All the basic modalities of CTL (EFp, EGp, and $E(qUp)$) can be expressed as fixpoints using the modality EX. The fixpoint equations are described below and the correctness of the equations was demonstrated in Chapter 2.

$$\begin{aligned} EFp &= \mu Y.(p \vee EXY) \\ EGp &= \nu Y.(p \wedge EXY) \\ E(qUp) &= \mu Y.(p \vee (q \wedge EXY)) \end{aligned}$$

Consequently, it is sufficient to give the semantics for EXf. Let Im_ξ be the forward image under the representative relation, and Im_R^{-1}

be the pre-image under the transition relation of the original structure $M = (S, R, L)$. The set of representatives satisfying EXf is $Im_\xi(Im_R^{-1}(K))$, where K is the set of representatives satisfying the state formula f. For example, consider the formula EFp such that p is an atomic proposition and G is an invariance group for p. Let $K_0 \subseteq Rep$ be the set of representatives labeled by p. Let K_i be the set of representatives at the i-th iteration. The equations given below describe the set of representatives that satisfy the formula EFp:

$$
\begin{aligned}
K_0 &= \{r \mid r \in Rep \text{ and } p \in L(r)\} \\
K_{i+1} &= Im_\xi(Im_R^{-1}(K_i)), \ \text{ for } i \geq 0
\end{aligned}
$$

If K is a set of representatives, then $Im_\xi(Im_R^{-1}(K))$ again is a set of representatives. Therefore, in this new model checking algorithm, we always maintain subsets of representatives, which can result in substantial savings. By placing some restrictions on the representative relation ξ, we can prove the correctness of this model. These restrictions on representative relations are quite general and we believe that they hold in most practical cases. There is an implicit assumption that for each orbit $\theta(s)$ of S under G there exists $r \in Rep$ such that $\theta(s) = \theta(r)$; that is, we assume that every orbit is represented.

Definition 3.6.2 Let G be a symmetry group for a Kripke structure $M = (S, R, L)$. Let $\xi \subseteq S \times Rep$ be a representative relation such that $(s, r) \in \xi$ implies that $\theta(s) = \theta(r)$. Let $C \subseteq G$ be a subset of permutations. The set C is called *complete* for ξ if and only if

- The condition $(s, r) \in \xi$ implies that $\exists(\sigma \in C)$ such that $\sigma s = r$.
- For all $r \in Rep$ and $\sigma \in C$ we have that $(\sigma r, r) \in \xi$.

The intuition for this definition will become clear when we prove Lemma 3.6.1. It allows us to translate a path in the quotient model M_G to a corresponding path in the model M_ξ which is defined below.

Definition 3.6.3 Given a Kripke structure $M = (S, R, L)$ and a representative relation $\xi \subseteq S \times Rep$, $M_\xi = (Rep, R_\xi, L_\xi)$ is defined as follows:

$$
\begin{aligned}
R_\xi &= \{(r_1, r_2) \mid \exists(s \in S)((s, r_1) \in \xi \wedge (s, r_2) \in R)\} \\
L_\xi(r) &= L(r)
\end{aligned}
$$

Notice that $Im_{R_\xi} = Im_R \circ Im_\xi^{-1}$ and $Im_{R_\xi}^{-1} = Im_\xi \circ Im_R^{-1}$. Therefore, the pre-image of a set of representatives K in the structure M_ξ is $Im_\xi(Im_R^{-1}(K))$. Hence, proving that the model checking procedure described at the beginning of the section is correct is equivalent to proving that M_ξ and M satisfy the same invariant $CTL^\star$ formulas. Since we have already proved the equivalence between M and the quotient model M_G, it suffices to prove the equivalence between M_ξ and M_G.

Lemma 3.6.1 If ξ has a complete set of permutations $C \subseteq G$, then the *corresponding path* theorem holds for M_ξ and M_G.

Proof: Let $\pi_\xi = (r_0, r_1, \cdots)$ be a path in M_ξ. From the definitions it is easy to see that $(r_1, r_2) \in R_\xi \Rightarrow (\theta(r_1), \theta(r_2)) \in R_G$, so $\pi_G = (\theta(r_0), \theta(r_1), \cdots)$ is a path in M_G.

For the other direction, let $\pi_G = (\theta(s_0), \theta(s_1), \cdots)$ be a path in M_G. Let r_0 be an arbitrary representative from the orbit $\theta(s_0)$. Since G is a symmetry group for M and $(\theta(s_0), \theta(s_1)) \in R_G$, there exists $t_1 \in \theta(s_1)$ such that $(r_0, t_1) \in R$. Let $r_1 \in Rep$ such that $(t_1, r_1) \in \xi$. Since C is a complete set for ξ, there exists $\sigma \in C$ such that $\sigma t_1 = r_1$. Since G is a symmetry group, $(\sigma r_0, \sigma t_1) \in R$. Since C is a complete set for ξ , $(\sigma r_0, r_0) \in \xi$. By definition of R_ξ, $(r_0, r_1) \in R_\xi$. Continuing the argument we can show that for every natural number i there exists $r_i \in Rep$ such that $\theta(r_i) = \theta(s_i)$ and $(r_i, r_{i+1}) \in R_\xi$. The path $\pi_\xi = (r_0, r_1, \cdots)$ is a corresponding path in M_ξ. $\Box$.

Theorem 3.6.1 Let $M = (S, R, L)$ be a Kripke Structure, G be a *symmetry group* of M, and h be a $CTL^\star$ formula. Let $\xi \subseteq S \times Rep$ be a representative relation and $C \subseteq G$ be the corresponding complete set. If G is an invariance group for all the atomic propositions p occurring in h, then for all $r \in Rep$

$$M_G, \theta(r) \models h \Leftrightarrow M_\xi, r \models h \tag{3.4}$$

where M_G is the quotient structure corresponding to M and M_ξ is defined above.

Proof: The proof of theorem is very similar to that of Lemma 3.2.3 using Lemma 3.6.1.□

Theorem 3.6.2 Let G be a permutation group acting on a finite set S, and let H be a sub-group of G. Let the set of representatives $Rep \subseteq S$ be the union of some orbits of S under H. Given the conditions above, there exists a representative relation ξ and a corresponding complete set C.

Proof: Let $G = H + H\psi_1 + \cdots + H\psi_r$ be the complete right traversal of H in G (each ψ_i is in a different right coset of $G \backslash H$). Define $C = \{e, \psi_1, \psi_1^{-1} \cdots, \psi_r, \psi_r^{-1}\}$ (e is the identity permutation). We define the representative relation as follows: $(s, r) \in \xi$ if and only if there exists $\sigma \in C$, $r \in Rep$, and $\sigma s = r$. The first condition for C to be complete for ξ follows from the definition. We now prove that the second condition holds. Consider σr for $\sigma \in C$. Since $\sigma^{-1} \in C$, we have that $(\sigma r, r) \in \xi$. Hence, C is a complete set for ξ, but we have to prove a *consistency* condition, i.e., for every $s \in S$ there exists $r \in Rep$ such that $(s, r) \in \xi$ and $\theta(s) = \theta(r)$. The condition states that every state is related to some representative in its orbit.

The way we have defined ξ means that proving *consistency* is equivalent to proving that for all $s \in S$ there exists a $\sigma \in C$ such that $\sigma s \in Rep$. Notice that since $C \subseteq G$, for all $\sigma \in C$ the states σs and s are in the same orbit of S under G. Let $\theta(s)$ be the orbit of $s \in S$ under G. Since H is a sub-group of G, the orbits of S under H are a refinement of orbits of S under G. Let $\theta_H \subseteq \theta(s)$ be an orbit of S under H such that $\theta_H \subseteq Rep$. Let $r \in \theta_H$ be an arbitrary representative. Since r and s belong to the same orbit of S under G, there exists $\sigma \in G$ such that $\sigma s = r$. Using the right traversal of H in G we can write $\sigma = \sigma_1 \psi$, were $\sigma_1 \in H$ and $\psi \in C$. Since $\sigma s = r$, $\psi s = \sigma_1^{-1} r$. Since $\sigma_1^{-1} \in H$, $\sigma_1^{-1} r \in \theta_H$. Therefore $\psi s \in Rep$. □.

The theorem given above assumes that Rep is the union of some orbits of S under H. Therefore, if the orbits of H are large, then the set of representatives could be large. In practice, Rep is provided and then the group H is determined as follows. Suppose the state set S is given by the assignment to n boolean state variables $x_1, \cdots, x_n$. Let Rep be those states such that $x_1 = 1$. If G is the symmetry group of the underlying structure, then G^1 (the subgroup of G which fixes the index 1) serves the purpose of H in the theorem given above. This fact will be shown later. Hence, in some sense the choice of H is fixed by the choice of *Rep*.

A permutation σ acting on the set S is said to *stabilize* a set $Y \subseteq S$ iff the following condition holds:

$$\forall (y \in Y)(\sigma(y) \in Y)$$

Notice that the condition given above is equivalent to the condition $\sigma(Y) = Y$. Let G be a permutation group acting on the set S. Given $Y \subseteq S$, the subgroup G_Y (the stabilizer of Y in G) of G is defined as follows:

$$G_Y \quad = \quad \{\sigma | (\sigma \in G) \wedge (\sigma(Y) = Y)\}$$

Now we prove a useful generalization of the theorem given above.

Theorem 3.6.3 Let $M = (S, R, L)$ be a Kripke structure and G be its symmetry group. Let $Rep \subseteq S$ be the set of representatives. Let H be a subgroup of G such that for all $\sigma \in H$, $\sigma(Rep) = Rep$ (notice that that this is another way of stating that *Rep* is the union of some orbits of S under H), i.e., H stabilizes *Rep*. Let $C \subseteq G$ be a set which satisfies the following conditions

1. For each coset of $G \backslash H$ we have a permutation $\psi \in C$ which belongs to that coset.

2. The set C is inverse closed, i.e $\psi \in C$ implies that $\psi^{-1} \in C$.

Let the representative relation ξ be defined as follows: $(s, r) \in \xi$ iff $s \in S$, $r \in Rep$ and there exists a $\psi \in C$ such that $\psi s = r$. Then ξ is valid representative relation and C is the corresponding complete set.

Proof: It is exactly same as the proof of Theorem 3.6.2. $\Box$.

In the theorem given below S is the set of states given by assignments to the boolean variables $x_1, x_2, \cdots, x_n$. Each state is a 0-1 vector of size n.

Theorem 3.6.4 Let the set of representatives *Rep* be given by the propositional formula $p(x_1, x_2, \cdots, x_n)$, i.e., a state $s = (y_1, \cdots, y_n)$ is a representative iff $p(y_1, \cdots, y_n) = 1$. If G is the symmetry group for the Kripke structure $M = (S, R, L)$, then there exists a representative relation $\xi \subseteq S \times Rep$ and a corresponding complete set C.

Proof: Let G_p be the invariance group of the propositional formula p. Let $H = G_p \cap G$. We prove that if $s \in S$ is a representative (i.e., $p(s) = 1$), then $\theta_H(s) \subseteq Rep$ ($\theta_H(s)$ is the orbit of s under H). Since every permutation $\sigma \in H$ is an invariant for p, $p(\sigma s) = 1$ for all $\sigma \in H$. Hence $\theta_H(S) \subseteq Rep$. Therefore, *Rep* is the union of orbits of H. Now we can use Theorem 3.6.2 to get a representative relation ξ and a complete set C. □

We give examples illustrating the utility of the result.

Example 3.6.1 Let $x_1, x_2, \cdots, x_n$ be the list of boolean state variables. Let G be a permutation group acting on the set $\{1, 2, \cdots, n\}$. G induces a permutation group $B(G)$ on the set $\{0, 1\}^n$, but we will work with G. Quite frequently representatives are given by an assignment to variables whose index is in a certain set $Y \subseteq \{1, 2, \cdots, n\}$. For example, $Y = \{1, 2\}$ and $x_1 = 0, x_2 = 1$ might describe the representatives (in this case the proposition $\bar{x_1} \wedge x_2$ describes the representatives). Let $\mathcal{A}$ be the assignment to the variables x_i such that $i \in Y$. The assignment $\mathcal{A}$ defines the set of representatives *Rep*. We use $\mathcal{A}_i$ to denote the value of the variable x_i ($i \in Y$) in the assignment. Let $Y_1 = \{i | \mathcal{A}_i = 1\}$ and $Y_0 = Y - Y_1$. In this case H (the invariance group of the proposition describing the representatives) is $(G_{Y_1})_{Y_0}$. Intuitively, all variables which are assigned the same value by the assignment $\mathcal{A}$ can be permuted freely. Now we can use Theorem 3.6.4 to get a representative relation ξ and a corresponding complete set C.

Example 3.6.2 We give an even more concrete example. Take the cache-coherence protocol with n processes. Each process has k local variables. The variables $x_{k(i-1)+1}, \cdots, x_{k(i-1)+k}$ are the local variables corresponding to process i. Let $x_{k(i-1)+1}$ correspond to the variable that indicates whether process i is the master: $x_{k(i-1)+1} = 1$ means process i is the master. Assume that we can switch the context of processes i and j, which corresponds to the permutation

$$\sigma_{ij} \quad = \quad (k(i-1)+1, k(j-1)+1)\cdots(k(i-1)+k, k(j-1)+k)$$

The symmetry group G of the Kripke structure M is generated by σ_{1i} $(2 \leq i \leq n)$. Suppose we choose the set of representatives as the states where process 1 is the master, i.e., $x_1 = 1$. The invariance group H of the proposition x_1 is G^1, where G^1 is the subgroup of G that fixes the index 1. Two permutations σ_1 and σ_2 are in the same coset of $G \backslash G^1$ iff $\sigma_1(1) = \sigma_2(1)$. Notice that for $k \neq j$, σ_{1j} and σ_{1k} lie in different coset of of $G \backslash G^1$ because they map 1 to different positions. Therefore, the complete right traversal of G^1 in G is given by the following equation.

$$G \quad = \quad G^1 + G^1\sigma_{12} + \cdots + G^1\sigma_{1n}$$

Using Theorem 3.6.3 we get a representative relation ξ and the complete set $\{e, \sigma_{12}, \cdots, \sigma_{1n}\}$

The next lemma will be used to prove the correctness of our experiments performed on a simple version of the Futurebus+ cache-coherence protocol. The experimental results are given in the next section. We extend the definition of the orbit of a state to a orbit of a set of states. Orbit of a set $Y \subseteq \{1, 2, \cdots, n\}$ is defined as follows:

$$\theta(Y) \quad = \quad \{Y' \mid \exists(\sigma \in G)(\sigma(Y) = Y')\}$$

The group G acts on the set $\{1, 2, \cdots, n\}$.

Lemma 3.6.2 Let G, Y, Y_0, Y_1 be as in Example 3.6.1. Let $C \subseteq G$ be a set which is inverse closed and such that for every pair of sets $Y_0' \in \theta(Y_0)$ and $Y_1' \in \theta(Y_1)$ there exists a $\sigma \in C$ such that $\sigma(Y_0) = Y_0'$ and $\sigma(Y_1) = Y_1'$. Let $\xi \subseteq S \times Rep$ be defined as follows: $(s, r) \in \xi$ iff there exist $\sigma \in C$ such that $\sigma(s) = r$. In this case ξ is a valid representative function and C is the corresponding complete set.

Proof: Let $H = (G_{Y_1})_{Y_0}$ be as defined in example 3.6.1. Two permutations σ_1 and σ_2 of G are in the same coset of H iff $\sigma_1(Y_0) = \sigma_2(Y_0)$ and $\sigma_1(Y_1) = \sigma_2(Y_1)$. Therefore, C has a permutation from every coset of $G \backslash H$ (the argument is very similar to the one given in the example before). Applying Theorem 3.6.2 we get the result.$\Box$

3.7 Empirical Results

This section provides empirical results to test the multiple representative theory given earlier.

3.7.1 Futurebus cache-coherence protocol

The first example is a simple cache coherence protocol for a single-bus multiprocessor system based on the Futurebus+ IEEE standard [41]. The verification of a more detailed version of the protocol with multiple buses is described in [18]. The system has a bus over which the processors and the global memory communicate. Each processor contains a local cache which consists of a fixed number of cache lines (see Figure 3.4).

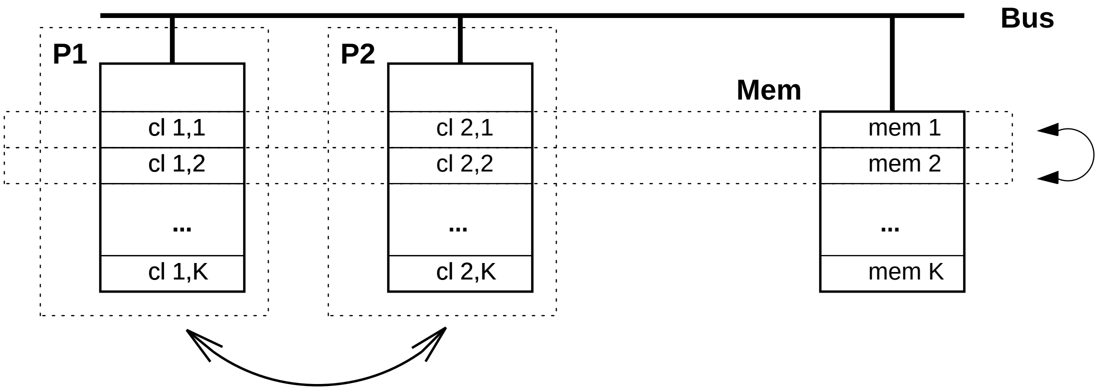

Figure 3.4: System structure

In each bus cycle the bus arbiter chooses one processor to be the master. The master processor selects a cache line address and a command it wants to put on the bus. The other processors and the memory respond to the bus command and change their local context. The reaction of the components is described in the protocol standard, which enforces the coherence of the cache lines among the different processors, i.e., only valid data values are read by the processors and no writes are lost. For the verification task the protocol is formalized, and cache coherence and other important system properties are expressed in temporal logic.

The behavior of the processors, the bus and the memory can be described by finite state machines. The state of the processor P_i is a combination of the states of each cache line in the processor cache and the state of the bus interface. The global bus is represented by the command on the bus, the active cache line address and other bus control signals (e.g., for bus snooping and arbitration.)

There are two obvious symmetries in the system. First, processors are symmetric: we can exchange the context of any two processors in the system. Second, cache lines are symmetric: any two cache lines can be exchanged simultaneously in all processors and the memory. To maintain consistency, along with applying the symmetries mentioned above, all the cache lines and processor addresses in the system must be renamed. Both symmetries are indicated in Figure 3.4 by arrows.

The complete system is the synchronous composition of all the components and can be described by a Kripke structure $M = (S, R, L)$. Since domains can be encoded in binary, a state is just a binary vector, and the transition relation R can be represented by a OBDD. The experiments were performed with two variable orderings which we call "concatenation" and "interleaving". The concatenation ordering is simply $P_1 \prec P_2 \prec \ldots \prec P_N$. The variables of processor i are ordered before the variables of processor $i+1$. In the interleaved ordering the processor variables are interleaved; that is $p_{1,1} < p_{2,1} < \cdots < p_{N,1} < p_{2,2}, \cdots$, where $p_{i,1}, \cdots, p_{i,K}$ are the state variables of processor P_i. The variables of the bus and the memory are ordered before all other variables in both orderings. In both orderings, each next state variable is placed immediately after the corresponding state variable.

Throughout this discussion we use N to denote the total number of processors and M to denote the total number of cache lines. Let π_{1j} be the permutation which exchanges the context of processor 1 with processor j. Let ψ_{1k} be the permutation that exchanges cache line 1 with cache line k. The symmetry group that uses processor (cache) symmetry alone is generated by the set of permutations $\Pi = \{\pi_{1j} | 1 \leq j \leq N\}(\Psi = \{\psi_{1k} | 1 \leq k \leq M\})$. The symmetry group that uses processor and cache symmetry is generated by the permutations $\Pi \cup \Psi$. When only the processor (cache) symmetry was used, the set of representatives were the states where processor 1 is the master (cache line 1 is active). When both the symmetries were used, the set of representatives were the set of states where processor 1 is the master and cache line 1 is active. The representative relation for all three cases is given below:

1. In the case of processor symmetry, $(s, r) \in \xi_p$ iff there exists π_{1j} $(1 \leq j \leq N)$ such that $\pi_{1j}(s) = r$.

2. In the case of cache symmetry, $(s, r) \in \xi_c$ iff there exists ψ_{1k} $(1 \leq k \leq M)$ such that $\psi_{1j}(s) = r$.

3. In the case when both symmetries are used, $(s, r) \in \xi_{pc}$ iff there exists $1 \leq j \leq N$ and $1 \leq k \leq M$ such that $\psi_{1k}(\pi_{1j}(s)) = r$.

We will prove that the set $C = \{\psi_{1k} \circ \pi_{1j} | k \leq M \ \wedge \ j \leq N\}$ is complete for the representative relation ξ_{pc}. The cases when we use only cache and processor symmetry are very similar to Example 3.6.2. Let $master_i$ be the index of the variable that tells that processor i is the master: $x_{master_i} = 1$ means processor i is that master. Let $active_j$ be the index that indicates that cache line j is active. Using the notation of Lemma 3.6.2 we have that $Y_1 = \{master_1, active_1\}$. The orbit of Y_1 is the set of pairs of indices of the form $\{master_j, active_k\}$. Consider a typical element $Y' = \{master_j, active_k\}$ in $\theta(Y_1)$. It is clear that $\psi_{1k}(\pi_{1j}(Y_1)) = Y'$. Therefore, by Lemma 3.6.2, C is a complete set for ξ_{pc}.

Consider the following properties which can be represented by a propositional formula:

1. Property p asserts that for all cache lines, if one processor is in EM(exclusive modified) state, then all other processors are in I(invalid) state.

2. Property q states that for all cache lines, if memory has valid data, then either all processors are in SU (shared unmodified) or I(invalid) state or one processor is in EU(exclusive unmodified) state.

3. Property m asserts that all cache lines in memory are valid.

4. Property c says that the command on the bus is either *read-modified* or *invalidate*

To verify that p and q remain true everywhere, we checked that the initial state does not belong to $EF\neg p$ and $EF\neg q$. These properties could be checked by doing reachability, but in order to test our theory we check them by computing $EF\neg p$ because this involves finding pre-images. We also checked whether the initial state satisfies the property $AG(m \rightarrow A(mUc))$ which asserts that if memory has valid data then it remains valid until an appropriate command is issued. This property turned out be false because there are reachable states where m is true and there exists a path where the command is never *read-modified* or *invalidate*. We also tested that from all the reachable states it is possible to get to a state where memory has valid data for all the cache lines, i.e., we checked that initial state satisfies the property $AG(EFc)$. The OBDD sizes for the property $EF\neg q$ were the largest. We ran the experiments with various system configurations. The results are summarized below. The OBDD sizes in the case of the interleaved ordering were much smaller than the OBDD sizes in the case of the concatenation ordering. Therefore, in the tables given below we have only included the data for the interleaved ordering.

Each row $jPkC$ in the table gives the results for a configuration of j processors and k cache lines. The first column gives the number of OBDD nodes for representing the transition relation. The remaining columns give the size of the largest OBDD and cpu time for model checking of the properties described above. First, no symmetry was used. Next, we give the results for symmetry between processors and symmetry between cache lines. In the last case, we used the combination of both symmetries. All experiments were run on a Sun Sparc10 workstation and the size of the largest OBDD gives a tight bound for the maximal memory usage. Table 3.1 gives the results without the use of symmetry. Table 3.2 summarizes the results when symmetry was exploited.

Exploiting the symmetry between processors or cache lines reduces the OBDD size by a factor that is linear in the number of processes or cache lines. The combination of these two symmetries reduces the size of the largest OBDD by the product of the number of processors and cache lines, because the two symmetries are independent. As a result, exploiting symmetry reduces the memory usage. The cpu times are not reduced by the same factor: exploiting symmetry requires additional

system config.	trans. relation	no symmetry	
	BDD nodes	BDD nodes	time sec.
2P2C	631	920	2
4P2C	8,534	6,048	11
2P4C	1,519	6,166	36
4P4C	22,154	42,595	231
2P8C	3,295	17,446	756
4P8C	49,394	121,475	5,911
2P12C	5,071	28,726	5,136
4P12C	76,686	-	-

Table 3.1: Empirical Results Without Symmetry

system config.	symmetry					
	processors		cache lines		combination	
	BDD nodes	time sec.	BDD nodes	time sec.	BDD nodes	time sec.
2P2C	668	1	518	1	368	1
4P2C	2,573	4	2,855	6	1,309	3
2P4C	3,917	18	1,458	9	1,178	6
4P4C	14,626	62	6,831	47	4,266	27
2P8C	10,837	407	2,618	152	2,338	98
4P8C	40,466	1,400	11,551	678	8,986	424
2P12C	17,757	2,884	3,778	841	3,498	577
4P12C	-	-	16,271	3,808	13,706	2,300

Table 3.2: Empirical Results With Symmetry

time to map states onto the representatives after each pre-image step in the model checking procedure.

3.7.2 Futurebus Arbiter

Our next example is the Futurebus Arbiter. Each module has a priority number. While competing for the bus, modules with higher priority number are given preference. Among competing modules with the same priority number, the winner is decided non-deterministically. Winner is the module which wins the competing phase. The arbitration cycle (which results in a winner) has six phases. We give a brief overview of these phases. The interested reader is referred to the IEEE Futurebus standard for a more detailed account [41].

- **Phase 0**
  In this phase modules decide to compete for the bus.

- **Phase 1**
  Noticing that a competition phase is about begin, other modules might decide to compete.

- **Phase 2**
  In this phase a winner (called the *master elect*) is selected from the set of competing modules. Winner is selected according to the rules described earlier.

- **Phase 3**
  Other modules check that the master elect had the highest priority number among the competing modules. If this is not the case, modules assert an error. If an error has not occurred, this phase continues until the master of the bus relinquishes its control. In this phase a module with higher priority than the master elect might start a new arbitration cycle. This is called *deposing* the master elect.

- **Phase 4**
  In this phase the current master of the bus might inhibit transfer of control of the bus to the master elect. If the master relinquishes its control over the bus, the arbitration cycle moves to the last phase.

- **Phase 5**
  In this phase the master elect gets control of the bus. This phase is called the *transfer of tenure* phase.

There are three boolean variables in each module which ensures the proper sequencing of the phases in an arbitration cycle. The internal state of the module depends on the outcome of the arbitration cycle and is shown below.

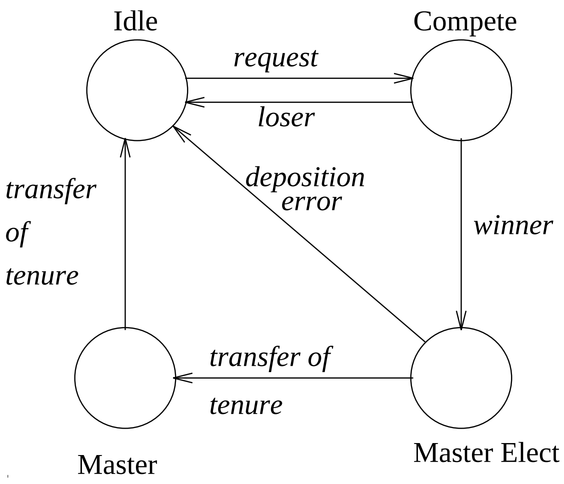

Figure 3.5: States of the Arbiter

Notice that two modules with the same priority number can be permuted without changing the behavior of the system. Formally, the the permutation corresponding to exchanging two modules with the same priority number is a symmetry of the system. However, two modules with different priority numbers cannot be exchanged. In the table given below the first column shows the configuration of the system. For example, 10m is the configuration with 10 modules having the same priority number. 10m10m denotes the configuration where the first 10 modules have a higher priority number than the last 10 modules (there are 10 modules in all). Assume that we have m sets of n modules (denoted by $\{M_{11}, \cdots, M_{1n}\}, \cdots, \{M_{m1}, \cdots, M_{mn}\}$. Moreover, the priority numbers of two modules M_{jl} and M_{rt} is the same iff $j = l$, or they belong to the same set. In this case the set of representatives is the set of states where the first module from a particular set is always chosen master elect. The reasoning that this is a valid set of representatives is very similar to the one given for the Futurebus cache-coherence protocol. The table of experimental results is shown below:

System Config	Time	BDD size	Time (Symm)	BDD size (Symm)
10m	163.41	369,705	37.04	27,671
12m	487.56	921,034	43.12	36,135
10m10m	1171.85	932,429	126.17	73,514
12m12m	-	-	198.195	93,094

3.8 Bisimulation Experiments

A relation $B \subseteq S \times S$ is called a *bisimulation relation* for a Kripke structure $M = (S, R, L)$ if and only if $(s, s_1) \in B$ implies that:

- for all s' such that $(s, s') \in R$ there exists a s'_1 such that $(s_1, s'_1) \in R$ and $(s', s'_1) \in B$

and an equivalent condition holds for s_1.

Given a symmetry group G for the Kripke structure $M = (S, R, L)$ the orbit relation induced by G is a bisimulation relation. A natural question is, why we do not use bisimulation minimization algorithms to obtain a quotient structure?

Bisimulation minimization algorithms require that a transition graph must be constructed (partially in "on-the-fly" algorithms), and then minimized using the fixpoint characterization of bisimulation. In contrast, symmetry can be already used during the construction of the transition graph (see Algorithm 3.2). Therefore, even infinite systems can be handled, if the quotient structure is finite.

The orbit relation corresponding to a symmetry group is not the largest bisimulation: two orbits might be bisimilar and could be merged by the coarsest bisimulation. As a result, the quotient structure obtained by symmetry may have more states than the one obtained using bisimulation minimization. However when using OBDDs the crucial parameter is not the number of states, but the "structure" of the state space. For example, using multiple representatives increased the number of representatives, but provided a significant performance improvement. Similarly, the experiments described below indicate that we get better reductions using symmetry (with multiple representatives) than using bisimulation minimization.

Let R be the transition relation of a Kripke structure and E an equivalence relation on the set of states. B_E is the largest bisimulation contained in E and can be computed by the following fixpoint computation:

$$
\begin{array}{l}
B_0(p,q) := E(p,q) \\
\textbf{repeat} \\
\qquad B_{i+1}(p,q) := \quad \begin{array}{l} \forall p' : (R(p,p') \rightarrow \exists q' : (R(q,q') \wedge B_i(p',q'))) \ \wedge \\ \forall q' : (R(q,q') \rightarrow \exists p' : (R(p,p') \wedge B_i(p',q'))) \end{array} \\
\textbf{until} \ \ B_i == B_{i+1} \\
B_E := B_i
\end{array}
$$

When implementing such an iterative scheme using OBDDs, the variable ordering is very important. There are two variable orderings to consider:

- The variable ordering of the p variables (similarly q variables). This ordering was the *interleaved ordering* defined in section 3.6.
- The variable ordering of q variables with respect to the p variables.

The experiments were performed with three different orderings between p and q: *concatenated*, (i.e., all p variables precede all q variables); *interleaved* (i.e., p_0,q_0,p_1,q_1,...); and *reordered*, where we used dynamic variable reordering [31, 63] during the iteration[1]. In Table 3.3 the column *bisim order* denotes the used order. Experiments with the interleaved ordering always behaved much worse with respect to time and space requirements than those with the concatenated ordering. Therefore, we have only included the data for the concatenated ordering and reordering in the table.

Results for various configurations of the cache-coherence protocol are listed in the Table 3.3. The computations were aborted if a spec-

[1]We used window and sifting techniques in the reorder algorithms

example	inv	reach (sec)	bisim order	time (sec)	nodes
2p2c	inv	no	c	5.4	935
2p2c	inv	0.5	c	1.2	506
2p2c	inv	0.6	r	3.5	405
2p2c	none	no	c	0.9	180
2p2c	none	0.6	c	1.1	506
2p2c	none	0.6	r	3.5	405
2p4c	inv	no	c	-	-
2p4c	inv	1.4	c	7.7	1986
2p4c	inv	1.5	r	21.8	1203
2p4c	none	no	c	4.0	344
2p4c	none	1.5	c	7.8	1986
2p4c	none	1.5	r	21.9	1203
4p2c	inv	no	c	-	-
4p2c	inv	2.2	c	-	-
4p2c	inv	2.0	r	-	-
4p2c	none	no	c	10.6	528
4p2c	none	2.1	c	10.9	1904
4p2c	none	2.1	r	40.0	1484

Table 3.3: Bisimulation results

ified time bound was exceeded. The column *inv* denotes the set used for E. The symbol *none* means that $E =$ true and *inv* means that two states are in E iff they agree on the property *conflict*, which is true if a cache line exists, for which the caches of two different processors are in conflict, e.g., both are in "exclusive-modified" states. Notice that the property *conflict* is basically the property $\neg p$ where p is defined in Section 3.6. We experimented with computing the reachable states in advance and performed bisimulation minimization only for the reachable states. For these experiments column *reach* gives the time for reachability computation. If reachability computation was not done, there is a *no* in that column. The column *time* gives the time for the bisimulation iteration (without reachability) and *nodes* the number of OBDD nodes for the OBDD for the relation B_E. For all larger configurations, such as 4p4c, not listed in the table the time bound was always exceeded.

In summary, these experiments show that for this example bisimulation minimization is more complex than performing model checking using multiple representatives. Using symmetry we showed that model checking can be made more efficient. So we conclude that in our class of applications exploiting symmetry is superior to performing bisimulation minimization.

3.9 Related Work

There has been relatively little research on exploiting symmetry in verifying finite state systems. Most of the work on this problem has been performed by researchers investigating the reachability problem for Petri nets [38, 66]. However, the work on Petri nets does not consider general temporal properties nor the complications that are caused by representing the state space using OBDDs. In related research, Ip and Dill [42] propose a data type *scalarset* which facilitates detection of symmetry in finite state systems. Their technique uses an explicit state representation rather than OBDDs and only considers reachability analysis. The research closest to the one presented in this chapter is of Emerson and Sistla [29], but their work does not investigate the complexity that arises while using OBDDs. In [45] an approach to cut down the cost of protocol analysis using quotient structures induced by automorphism is proposed, but this work considers only limited kind of symmetry.

Chapter 4

Symmetry II

This chapter investigates the complexity of problems associated with exploiting symmetry. For example, given a group G and two states s and s', the *orbit problem* asks whether s and s' in the same orbit. Determining whether two states are in the same orbit is at the core of any model checking procedure exploiting symmetry. We prove that the orbit problem is equivalent to an important problem in computational group theory. We also investigate the complexity of checking whether a permutation is a symmetry of a Kripke Structure. We explore ways of deriving symmetries of shared variable concurrent programs. Since the orbit problem in its full generality is quite hard, this chapter also shows that the orbit problem is easy for certain commonly occurring groups.

This chapter is organized as follows: In Section 4.1 we introduce a shared variable model. Section 4.2 gives a technique to derive symmetries of shared variable programs. Section 4.3 derives symmetry groups for certain commonly occuring architectures like hypercubes, toruses, and trees. In section 4.4 we discuss the complexity of checking symmetry. Section 4.5 investigates the complexity of the orbit problem. Section 4.6 investigates some special classes of the orbit problem.

4.1 A Shared Variable Model of Computation

A *shared variable program* is defined with the state sets and the transition relation as follows:

- $S = Loc^I \times D^V$ is the finite set of *states*, with Loc a finite set of individual process *locations*, I the set of process indices, and V is a finite set of shared *variables* over a finite *data domain* D.

- $R \subseteq S \times S$ which represents the transitions of the system.

For convenience, each state $s = (s', s'') \in S$ can be written in the form $(\ell_1, \ldots, \ell_n, v = d, \ldots, v' = d')$ indicating that processes $1, \ldots, n$ are in locations $\ell_1, \ldots, \ell_n$, respectively and the shared variables $v, \ldots, v'$ are assigned data values $d, \ldots, d'$, respectively.

Next, we define a labeling function for a shared variable program. The set of *terms* are expressions of the form l_i $(i \in I)$ and $v = d$ $(v \in V$ and $d \in D)$. The set of atomic propositions AP are constructed from the set of terms by the logical connectives $\wedge$ and $\neg$. Given an atomic proposition $p \in AP$ and a state $s \in S$, the satisfaction relation $s \models p$ is defined in the following way:

- $s \models l_i$ iff the i-th process in the state s is in location l_i.

- $s \models v = d$ iff the shared variable v has the value d in the state s.

- $s \models f \wedge g$ iff $s \models f$ and $s \models g$.

- $s \models \neg f$ iff $s \not\models f$.

Given a shared variable program, we can construct a corresponding Kripke Structure $M = (S, R, L)$ (S and R were defined before) by constructing the following labeling function $L : S \rightarrow 2^{AP}$.

- $p \in L(s) \Leftrightarrow s \models p$.

In practice, for ordinary model checking, M is the Kripke Structure corresponding to finite state *concurrent program* $\mathcal{P}$ of the form $\|_{i=1}^{n} K_i$ consisting of processes $K_1, \ldots, K_n$ running in parallel. Each K_i may be viewed as a finite state transition graph with node set Loc. An arc from node ℓ to node ℓ' may be labeled by a guarded command $B \rightarrow A$. The guard B is an atomic proposition that can inspect shared variables and local states of "accessible" processes. A is a set of *simultaneous assignments* to shared variables $v := d \parallel \cdots \parallel v' := d'$. When process K_i is in local state ℓ and the guard B evaluates to *true* in the current global state, the program $\mathcal{P}$ can nondeterministically choose to advance by firing this transition of K_i which changes the local state of K_i to be ℓ' and the shared variables in V according to A. Thus the arc from ℓ to ℓ' in K_i represents a *local transition* of K_i that we denote by $\ell : B \rightarrow A : \ell'$.

The Kripke structure $M = (S, R, L)$ corresponding to $\mathcal{P}$ is thus defined using the obvious formal operational semantics. First, the set of (all possible) states S is determined from $\mathcal{P}$ because it provides us with the set of local (i.e., individual process) locations *Loc*, process indices I, variables V, and data domain D. For states $s, t \in S$, we define $s \rightarrow t \in R$ iff
$\exists i \in I$ such that the process K_i can cause s to move to t, denoted $s \rightarrow_i t$ iff
$\exists i \in I$ $\exists$ local transition $\tau_i = \ell_i : B_i \rightarrow A_i : m_i$ of K_i which *drives* $s = (s', s'')$ to $t = (t', t'')$; i-th component of s' equals ℓ_i, the i-th component of t' equals m_i, all other components of s' equal the corresponding component of t', predicate $B_i(s) = true$, and $t'' = A_i(s'')$. $A_i(s'')$ is constructed from s'' by replacing the values of the shared variables according to the simulatneous assignment statement A_i. The labeling function L is defined as before.

4.2 Deriving Symmetry

This section deals with deriving symmetry for shared variable programs introduced in the previous section. Intuitively, if one has a graph G whose nodes corresponds to processes and the processes communicate over the edges of G, an automorphism of the graph G should manifest itself into a symmetry of the underlying structure. Succintly speaking, *structural symmetry introduces symmetry in the model.* This section proves that for certain cases one can derive the symmetry of the model from the topology of the system. Given a concurrent program $\mathcal{P} = \|_{i=1}^{n} K_i$, we build a hypergraph $HG(\mathcal{P})$. Under certain restrictions, we prove that the each automorphism of $HG(\mathcal{P})$ is also a symmetry of the underlying Kripke Structure M. A restricted version of the theorem appeared in [29]. For example, in [29] the authors assume that all processes are isomorphic and the variables are only shared between two processes. For instance, consider an arbiter which maintains a global *master* variable (indicating who has the resource). This arbiter could be handled by the framework presented in this section, but the theorem given in [29] does not apply in this context.

Let $\mathcal{P} = \|_{i=1}^{n} K_i$ be a concurrent program. In this section the index set is $I = [n]$. Each shared variable v is subscripted the by the set of indices of the processes which access that shared variable. For example, if x is accessed by processes 1, 4, and 5, we write x as $x_{\{1,4,5\}}$. Notice that each shared variable is uniquely determined by its name and subscript, but we allow shared variables to have the same name as long as their subscripts are different. For example, $x_{\{1,2\}}$ and $x_{\{3,4\}}$ are allowed. A permutation $\pi \in S_n$ acts on the variables in a natural manner, i.e., $\pi(x_w) = x_{\pi(w)}$. A permutation π acting on $[n]$ is called *consistent* if and only if for every shared variable x_w, $x_{\pi(w)}$ is a variable as well. This means that we only allow permutations which map shared variables to shared variables.

We define how a consistent permutation π acts on states, atomic propositions, and processes. Let π be a consistent permutation.

- Given a state $s = (l_1, \cdots, l_n, v_{w_1} = d_1, \cdots, v_{w_k} = d_k)$, the state $\pi(s)$ is defined as follows:
  - The i-th process is in location $l_{\pi(i)}$ in the state $\pi(s)$.
  - The shared variable $v_{\pi(w)}$ in the state $\pi(s)$ has the same value as the variable v_w in the state s.
- Let $p \in AP$ be an atomic proposition. $\pi(p)$ is recursively defined as follows:
  - $\pi(f \wedge g) \;=\; \pi(f) \wedge \pi(g)$.
  - $\pi(\neg f) \;=\; \neg\pi(f)$.
  - $\pi(l_i) \;=\; l_{\pi(i)}$.
  - $\pi(v_w = d) \;=\; (v_{\pi(w)} = d)$.
- Given a simultaneous assignment $A = (v_{w_1} = d_1 \| \cdots \| v_{w_k} = d_k)$, define $\pi(A)$ as the following simultaneous assignment.

$$v_{\pi(w_1)} = d_1 \| \cdots \| v_{\pi(w_k)} = d_k$$

- Given a process K_i, the process $\pi(K_i)$ is constructed in the following manner:
  - $l : B \to A : l'$ is a transition in K_i iff $l : \pi(B) \to \pi(A) : l'$ is a transition in $\pi(K_i)$.

The lemma given below says that applying consistent permutations preserves the satisfaction relation.

Lemma 4.2.1 Let π be a consistent permutation. Let s be a state and p be an atomic proposition. In this case, $s \models p$ iff $\pi(s) \models \pi(p)$.

Proof: The proof is by structural induction on p. $\Box$

The Lemma given below will be used in proving the main theorem.

Lemma 4.2.2 Let π be a consistent permutation such that $\pi(i) = j$ and $\pi(K_i) = K_j$. Assume that $s \to_i t$. Then $\pi(s) \to_j \pi(t)$.

Proof: Let $s = (l_1, \cdots, l_n, v_{w_1} = d_1, \cdots, v_{w_k} = d_k)$. Assume that $s \to_i t$. This means that a there is a transition $\tau : l_i : B \to A : l'_i$ in the process K_i which drives s to t. By definition, we have that $s \models B$. Using Lemma 4.2.1 we have that $\pi(s) \models \pi(B)$. Since $K_j = \pi(K_i)$, $l_i : \pi(B) \to \pi(A) : l'_i$ is an enabled transition in K_j in the state $\pi(s)$. Let $\pi(s) \to_j t'$. We have to prove that $t' = \pi(t)$. This follows straight from the definition.$\Box$

Definition 4.2.1 A *colored hypergraph* with n vertices and k colors is a 3-tuple $H = ([n], E, C)$ such that $E \subseteq 2^{[n]}$ is the *edge set*, and $C : [n] \to [k]$ is the *coloring function* which colors each node with one of the k colors. A permutation π acting on $[n]$ is called an *automorphism* of the hypergraph H iff the following two conditions hold:

- For all $1 \leq i \leq n$, $C(i) = C(\pi(i))$.
- $w \in E$ iff $\pi(w) \in E$.

The group of automorphisms of the hypergraph H is denoted by $Aut(H)$.

Definition 4.2.2 Given a concurrent program $\mathcal{P} = \|_{i=1}^{n} K_i$, define the *corresponding* colored hypergraph $HG(\mathcal{P}) = ([n], E, C)$ in the following manner:

- $w \in E$ iff there exists a shared variable with subscript w.
- Assume that we have a equivalence relation $\cong$ on the processes $K_1, \cdots, K_n$. Partition the processes $K_1, \cdots, K_n$ into equivalence classes induced by the relation $\cong$. Let $c_1, \cdots, c_k$ be the k equivalence classes. The coloring function C is defined as follows: $C(i) = r$ iff the process K_i is in the equivalence class c_r.

The definition of the equivalence relation $\cong$ will depend on the particular example, but here are two choices:

- $K_i \cong K_j$ if and only if there exists a consistent permutation π such that $\pi(K_i) = K_j$.
- K_i and K_j are the instances of the same `MODULE` definition [55].

Our goal is to relate the automorphism group of the Kripke Structure M (corresponding to the program $\mathcal{P} = \|_{i=1}^{n} K_i$) to the automorphism group of the hypergraph $HG(\mathcal{P})$. We want to make sure that every automorphism of $HG(\mathcal{P})$ which maps vertex i to j also maps process K_i to K_j, i.e., respects the structure of the program $\mathcal{P}$. To achieve this we introduce the following definition: Let $\Gamma(K_i)$ be the set of indices j such that there exists a shared variable x_w such that $\{i, j\} \subseteq w$. Intuitively, if $j \in \Gamma(K_i)$, then K_i and K_j share some variables. $\Gamma(K_i)$ is called the *neighborhood* of K_i. We require that $i \in \Gamma(K_i)$, i.e., a process is in its own neighborhood. Process K_i *respects* its neighborhood if and only if given any consistent bijection $f : \Gamma(K_i) \to \Gamma(K_j)$ such that $f(i) = j$ and for all $r \in \Gamma(K_i)$, $K_r \cong K_{f(r)}$, then $f(K_i) = K_j$. The bijection f acts on K_i exactly the same way as a permutation acts on K_i. The notion of consistency of bijections is similar to the consistency of permutations. Notice that the check that a process K_i respects its neighborhood is local , i.e., only involves processes with which its shares data. Moreover, sometimes processes are entirely symmetric, i.e., given a permutation π, $\pi(K_i) = K_{\pi(i)}$. In this case, K_i respects its neigborhood trivially.

Theorem 4.2.1 Let $HG(\mathcal{P})$ be the hypergraph corresponding to the program $\mathcal{P} = \|_{i=1}^{n} K_i$. Moreover, assume that for all $1 \leq i \leq n$, K_i respects its neighborhood. Let M be the Kripke Structure corresponding to $\mathcal{P}$. Given these conditions, $Aut(HG(\mathcal{P})) \leq Aut(M)$.

Proof: Let $\pi \in HG(\mathcal{P})$ be an aribitrary automorphism of the hypergraph corresponding to $\mathcal{P}$. Because of the second condition in the definition of the automorphism of a colored hypergraph, π is consistent. Let $s \rightarrow_i t$ be an arbitrary transition in M. Since π is an automorphism of $HG(\mathcal{P})$, $K_i \cong K_{\pi(i)}$ (this follows from the coloring function of $HG(\mathcal{P})$). Let $\Gamma(K_i)$ be the neigborhood of K_i. Notice that $\pi(\Gamma(K_i)) = \Gamma(K_{\pi(i)})$. Let $f_\pi : \Gamma(K_i) \rightarrow \Gamma(K_{\pi(i)})$ be the bijection corresponding to π, i.e., for all $i \in \Gamma(K_i)$, $f_\pi = \pi(i)$. Since K_i respects its neighborhood, $f_\pi(i)(K_i) = K_{\pi(i)}$. This also implies that $\pi(K_i) = K_{\pi(i)}$. Using Lemma 4.2.2 we get that $\pi(s) \rightarrow_{\pi(i)} \pi(t)$. Therefore, given a transition $s \rightarrow t$ in M, $\pi(s) \rightarrow \pi(t)$ is also a transition in M. Hence, $\pi \in Aut(M)$.$\Box$

4.3 Symmetries of Various Common Architectures

In section 4.2 we proved that in some cases structural symmetry induces symmetry in the underlying model. For example, if processes communicate over a hypercube, the symmetry of the hypercube will induce a symmetry on the underlying model. This section derives the automorphism group of three commonly occuring architectures: Hypercube, Torus, and Tree. First, we have to define certain operations on groups which will be used in this section. Two groups G and H acting on a set S are called *disjoint* iff for all $\sigma \in G$ and all $\psi \in H$, $\sigma(i) \neq i$ implies that $\psi(i) = i$, and $\psi(i) \neq i$ implies that $\sigma(i) = i$. Intuitively, G and H act on different parts of S. *Product* of G and H (denoted by $G \cdot H$) is the group generated the set

$$\{\psi.\sigma \mid \psi \in G \ \wedge \ \sigma \in H\}$$

Disjoint product is the product of two disjoint groups.

Definition 4.3.1 First, we give an informal definition of **wreath product** of two groups G and H denoted by $G \wr H$. Let $G \leq S_n$ and $H \leq S_m$. Take m disjoint copies of a set X of size n. The elements of $G \wr H$ work on $X_1 \cup \cdots \cup X_m$ in the following manner. First permute the elements of X_i according to a permutation in the group G and then permute the sets of X_i according to a permutation in H. Therefore, a permutation in $G \wr H$ can be written as $(g_1, \cdots, g_m, h)$ were $g_i \in G$ and

$h \in H$. Formally, consider the set $[nm]$. The i-th block of integers is $\{(i-1)n+1, \cdots, (i-1)n+n\}$. A permutation $(g_1, \cdots, g_m, h)$ acts on the set $[nm]$ in the following way:

- First, apply the permutations g_i in the following manner:

$$g_i((i-1)n+j) = (i-1)n+g_i(j)$$

- Then apply the permutation h in the following way:

$$h((i-1)n+j) = (h(i)-1)n+j$$

4.3.1 Hypercube

Definition 4.3.2 We will use B to denote the set $\{0,1\}$. A *Hypercube* of dimension n has 2^n vertices labeled by vectors from B^n. Two nodes u and v labeled by vectors $x, y \in B^n$ are connected iff x and y differ in just one bit, i.e., the hamming distance between them is 1. See Figure 4.1. Recall that the hamming distance between two 0-1 vectors is the number of positions in which they differ.

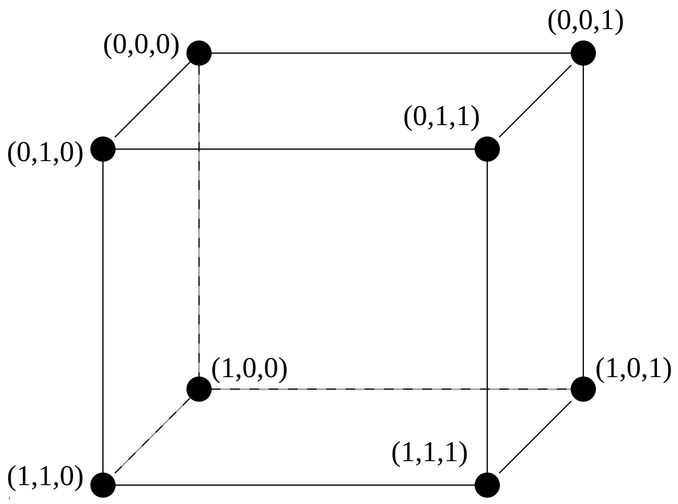

Figure 4.1: A 3-dimensional Hypercube

A permutation group S_n acts on the space B^n of n-dimenional 0-1 vectors in the natural way: a permutation $\sigma \in S_n$ maps a vector $(x_1, \cdots, , x_n)$ to $(x_{\sigma(1)}, \cdots, x_{\sigma(n)})$. We denote this group acting on B^n by $B(S_n)$.

Lemma 4.3.1 Let $B(S_n)$ be the permutation group acting on B^n induced by the full symmetric group S_n acting on $[n]$. A permutation $\sigma \in S_n$ maps $(x_1, \cdots, x_n)$ to $(x_{\sigma(1)}, \cdots, x_{\sigma(n)})$. An automorphism f of a n-dimensional hypercube which maps the zero vector $\vec{0}$ to itself belongs to $B(S_n)$.

Proof: Since an automorphism of the hypercube preserves hamming distances, f has to map a vector $x \in B^n$ to a vector which has the same number of 1s, i.e., $f(x)$ and x have same number of ones. This can be easily seen by comparing the hamming distance of $f(\vec{0}) = \vec{0}$ and $f(x)$. Let e_i (for $1 \leq i \leq n$) be the 0-1 vector which has a 1 in the i-th position and 0 everywhere else. Let $\sigma \in S_n$ be such that $\sigma(i) = j$ iff $f(e_i) = e_j$. Now it easy to see that f corresponds to $\sigma \in S_n$. $\Box$.

The i-th *complementation group* C_n^i acts on B^n in the following manner: It maps a vector $(x_1, \cdots, x_i, \cdots, x_n)$ to the vector $(x_1, \cdots, \overline{x_i}, \cdots, x_n)$, i.e., complements the i-th bit. The group C_n^i acts on B^n and is a cyclic group of order 2.

Theorem 4.3.1 The automorphism group of a n-dimensional hypercube is the group generated by $B(S_n)$ and C_n^i $(1 \leq i \leq n)$.

Proof: Let g be a automorphism of the n-dimesional hypercube. Let $g(\vec{0})$ have 1 in k positions $i_1, i_2, \cdots, i_k$. If $k = 0$, $g \in B(S_n)$ (by the previous lemma). Let h be the permutation which complements the bits in the positions $i_1, i_2, \cdots, i_k$. Notice that h is in the group generated by C_n^i $(1 \leq i \leq n)$. Using the property that g preserves hamming distances, we can see that $g \circ h^{-1}$ is an automorphism of the n-dimensional hypercube which maps $\vec{0}$ to $\vec{0}$. Now we use the previous lemma and get that $g \circ h^{-1} \in B(S_n)$. $\Box$.

Consider Figure 4.1. Now consider the permutation $(2, 3) \in S_3$ and the complementation group C_3^1. The cyclic permutation $(2, 3)$ has the effect of exchanging vertices $(0, 0, 1)$ with $(0, 1, 0)$ and $(1, 0, 1)$ with $(1, 1, 0)$ in the top and bottom squares of the hypercube. The complementation group C_3^1 exchanges the top and bottom squares. It is easy to see that this is an automorphism of the hypercube.

4.3.2 Torus

Definition 4.3.3 Consider a $n \times n$ grid whose nodes are labeled by tuples (i,j) $(1 \leq i \leq n$ and $1 \leq j \leq n)$. A **torus** is formed by adding edges between the nodes $(i,1)$ and (i,n) and $(1,i)$ and (n,i) to the grid. See Figure 4.2.

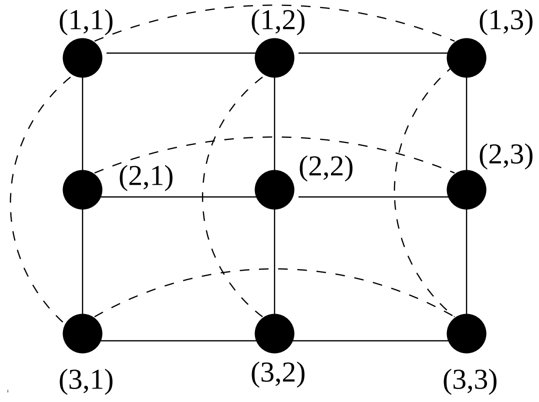

Figure 4.2: A 3×3 Torus

Definition 4.3.4 We define the following permutations:

1. The **flip** permutation F is defined as follows: $F((i,j)) = (j,i)$. The permutation F flips the torus and is an automorphism of a $n \times n$ torus. Notice the F is it owns inverese.

2. Let C_n be the cyclic group acting on $[n]$. Let $R = C_n \times C_n$. The permutations in R are of the form (α, β) such that $\alpha \in C_n$ and $\beta \in C_n$ and its acts on (i,j) in the following manner: $(\alpha,\beta)(i,j) = (\alpha(i), \beta(j))$.

Lemma 4.3.2 Let f be an automorphism of the $n \times n$ torus such that $f(1,1) = (i,1)$ and $f(2,1) = (j,1)$. In this case $f \in R$ and is of the form (α, e), were $\alpha \in C_n$ and e is the identity permutation.

Proof: Basically, the lemma states that if the node $(1,1)$ and $(2,1)$ are mapped to the same column by f, then f is equivalent to a permutation in C_n. The proof goes as follows: Just by looking at the torus it is easy to see that $f(1,k) = (i,k)$ and $f(2,k) = (j,k)$ (we are assuming that $f(1,1) = (i,1)$ and $f(2,1) = (j,1)$). Continuing this way we can see that all nodes in column k are mapped to nodes in column k. Let $\alpha(i) = j$ iff $f(i,1) = (j,1)$. Now it is clear that $f = (\alpha, e)$. $\Box$.

Theorem 4.3.2 The automorphism group of the $n \times n$ torus is generated by R and F.

Proof: Consider an automorphism f of a $n \times n$ torus. Let $f(1,1) = (i,j)$. Let $\beta \in C_n$ be such that $\beta(j) = 1$. Consider $\pi = (e, \beta)$. Let $g = \pi \circ f$. The permutation g has the property that $g(1,1) = (i,1)$. If $g(2,1) = (j,1)$, then by the previous lemma $g = \pi \circ f = (\alpha, e)$ or in other words $f = (\alpha, \beta^{-1})$. Suppose $g(2,1) = (i,k)$ (notice that $(2,1)$ has to remain adjacent to $(1,1)$ under automorphisms). In this case $g(1,2) = (j,1)$. It is easy to see that $g \circ F(2,1) = g(1,2) = (j,1)$. Therefore $g \circ F = (\alpha, e)$ (by the previous lemma) or $f = (e, \beta^{-1}) \circ (\alpha, e) \circ F^{-1}$ or in other words $f = (\alpha, \beta^{-1}) \circ F$. Since f was an arbitrary automorphism of the torus, the proof is complete.$\Box$.

Now we will illustrate how a generic automorphism of the torus is constructed through an example. Consider the 3×3 torus given in Figure 4.2. First, one can rotate the three columns simultaneously. After that, we can rotate all the rows simulataneously. After we are done rotating, we can flip the torus, i.e., node (i,j) becomes node (j,i). It is easy to see that all these three operations maintain the structure of the torus. The theorem given above says that all automorphisms of the torus can be generated in this manner.

4.3.3 Rooted Trees

Consider a tree $T = (V,E)$ with a distinguished vertex r as the root. We investigate the automorphism group of a rooted tree T. First color the nodes of the tree T with their isomorphism class, i.e., two nodes u and v are assigned the same color iff the trees rooted at u and v are isomorphic. We also assign level numbers to the nodes, i.e., leaves are assigned level 0 and the level of the root is the height of the tree. All these operations can be done in polynomial time (see [1]). We describe the automorphism group of the tree T in an inductive manner. First, assign a leaf u a trivial group acting on the set $\{u\}$. Let v be a node at level i. Let the sons of v be divided into m isomorphism classes. Let $C_k = \{s_1^k, \cdots, s_{n_k}^k\}$ be the sons of v which are in the k-th isomorphism class. The natural number k ranges from 1 to m. A typical permutation of the subtree rooted at v can permute the sons in an isomorphism class C_k. Let $G_{s_i^k}$ be the automorphism group of the tree rooted at s_i^k. Notice that a typical automorphism of T_v (the subtree rooted at v) which only permutes its sons in C_k can be represented by $(\sigma_1^k, \cdots, \sigma_{n_k}^k; \psi)$ where $\sigma_i \in G_{s_i^k}$ and $\psi \in S_k$. Basically, this means first we can permute the subtrees rooted at s_i^k and then permute the trees in C_k. Since by hypothesis $G_{s_i^k} \cong G_{s_j}^k$, the group G_k which only permutes the vertices in the k-th isomorphism class C_k is (by definition) isomorphic to $G_{s_1^k} \wr S_k$. The automorphism group of T_v is generated by $\bigcup_i^m G_i$. Carrying on in an inductive fashion, gives us the automorphism group of T which is just T_r. We will illustrate our ideas on the binary tree given in

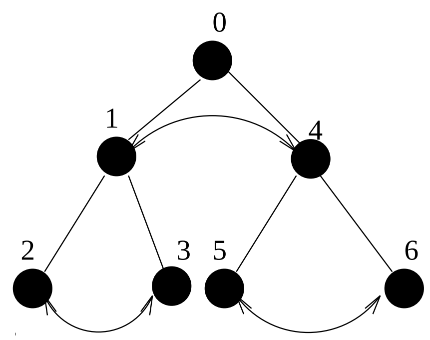

Figure 4.3: A Binary Tree

Figure 4.3. We can perform the following operations on the binary tree shown in Figure 4.3.

- Exchange the leaves 2 or 3 or leave them intact.

- Exchange the leaves 5 and 6 or leave them intact.

- Exchange the subtree rooted at 1 with the tree rooted at 4 or leave the trees unpermuted.

It is easy to see that these three operations leave the tree intact. Also if we regard the subtrees rooted at 1 and 4 as blocks of numbers ($\{1, 2, 3\}$ and $\{4, 5, 6\}$), the automorphism group generated by the three operations is isomorphic to $S_2 \wr S_2$.

4.4 Complexity of Checking Symmetry

In this section we assume that the state-space S of a Kripke Structure $M = (S, R, L)$ is given by assignments to n boolean state variables $x_1, \cdots, x_n$. Notice that in this case $S \cong B^n$ where $B = \{0, 1\}$. The transition relation R is given as a boolean function

$$R(x_1, \cdots, x_n, x'_1, \cdots, x'_n)$$

with the following semantics:

- Given a state $s = (y_1, \cdots, y_n)$ and $s' = (y'_1, \cdots, y'_n)$, there is a transition from s to s' if and only if

$$R(y_1, \cdots, y_n, y'_1, \cdots, y'_n) \;=\; 1$$

The set of atomic propositions AP are $\{x_1, \cdots, x_n\}$. The labelling function L is defined as follows:

- Given a state $s = (y_1, \cdots, y_n)$, $x_i \in L(s)$ if and only if $y_i = 1$.

Given a group $G = \langle g_1, \cdots, g_k \rangle$ acting on $[n]$, we want to check that G is indeed a symmetry group of M. Notice, that in order to check that G is a symmetry group for M we need to only verify that each of the generators g_i is a symmetry of G. First, we define how a permutation $\sigma \in S_n$ acts on the transition relation R. The permutation σ acts on R in the following manner:

$$\sigma(R(x_1, \cdots, x_n, x'_1, \cdots, x'_n)) \;=\; R(x_{\sigma(1)}, \cdots, x_{\sigma(n)}, x'_{\sigma(1)}, \cdots, x'_{\sigma(n)})$$

By definition, $\sigma \in S_n$ is a symmetry of M if and only if it preserves the transition relation R. Or in other words, $R = \sigma(R)$. Ordered Binary

Decision Diagram (OBDD) is an efficient data structure to represent boolean functions [11]. If we represent R as an OBDD, it is easy to obtain the OBDD for $\sigma(R)$ by variable substitution. Since OBDDs are a canonical representation for boolean functions, $R = \sigma(R)$ if and only if the OBDDs for R and $\sigma(R)$ are the same. Therefore, if R is given as a OBDD, we can check that $G = \langle g_1, \cdots, g_k \rangle$ is a symmetry group form M by verifying that $R = g_i(R)$ for all $1 \leq i \leq k$.

The basic step in checking symmetry is: given a permutation σ acting on $[n]$ and a boolean function $f(x_1, \cdots, x_n)$, check that $f = \sigma(f)$, i.e.,

$$f(x_1, \cdots, x_n) \;\; = \;\; f(x_{\sigma(1)}, \cdots, x_{\sigma(n)})$$

We call this problem the *symmetry checking* problem.

Theorem 4.4.1 Symmetry checking is *co-NP* complete

Proof: We will prove that the complement of the symmetry checking problem is *NP*-complete. The complement problem can be stated as follows:
Given a permutation σ acting on $[n]$ and a boolean function $f(x_1, \cdots, x_n)$ does there exist a $(y_1, \cdots, y_n) \in B^n$ such that

$$f(y_1, \cdots, y_n) \;\; \neq \;\; f(y_{\sigma(1)}, \cdots, y_{\sigma(n)})$$

The problem is obviously in NP because one can guess a vector $(y_1, \cdots, y_n) \in B^n$ and check that

$$f(y_1, \cdots, y_n) \;\; \neq \;\; f(y_{\sigma(1)}, \cdots, y_{\sigma(n)})$$

Next, we prove that the problem is NP-complete. The reduction is from SAT [33]. Assume that we are given a boolean function $f(x_1, \cdots, x_n)$ of n variables. Assume that $f(0, \cdots, 0) = 0$ (otherwise we have found a satisfying assignment to f). We construct a new function $g(x_1, \cdots, x_n, y_1, \cdots, y_n)$ of $2n$ variables in the following manner:

$$g(x_1, \cdots, x_n, x_{n+1}, \cdots, x_{2n}) \;\; = \;\; f(x_1, \cdots, x_n) \wedge \overline{x_{n+1}} \wedge \cdots \wedge \overline{x_{2n}}$$

Consider the permutation $\sigma = (1, n+1)(2, n+2) \cdots (n, 2n)$. We will prove that σ is not a symmetry of g if and only if f is satisfiable.

- Assume that f is satisfiable.
  Let $f(y_1, \cdots, y_n) = 1$. By assumption, $(y_1, \cdots, y_n) \neq (0, \cdots, 0)$. By the definition of g we have that

$$\begin{aligned} g(y_1, \cdots, y_n, 0, \cdots, 0) &= 1 \\ g(0, \cdots, 0, y_1, \cdots, y_n) &= 0 \end{aligned}$$

  Therefore, σ is not a symmetry of g.

- Assume that σ is not a symmetry of g.
  Hence, there exists $(y_1, \cdots, y_n, y_{n+1}, \cdots, y_{2n})$ such that

$$g(y_1, \cdots, y_n, y_{n+1}, \cdots, y_{2n}) \neq g(y_{n+1}, \cdots, y_{2n}, y_1, \cdots, y_n)$$

  The previous equations implies that one of the equations given below is true.

$$\begin{aligned} g(y_1, \cdots, y_n, y_{n+1}, \cdots, y_{2n}) &= 1 \\ g(y_{n+1}, \cdots, y_{2n}, y_1, \cdots, y_n) &= 1. \end{aligned}$$

  In either case using the definition of g we have that f is satisfiable.

It is obvious that our reduction can be done in polynomial time. The size of g and the size of f only differ by a polynomial. $\square$

4.5 Complexity of the Orbit Problem

In this section we assume that the state space of our system is given by assignments to n boolean state variables $x_1, \cdots, x_n$. Therefore, the state space is isomorphic to B^n (where $B = \{0, 1\}$). We assume that the symmetry group $G \leq S_n$ acts on B^n in the natural way: a permutation σ maps a vector $(z_1, \cdots, z_n)$ to $(z_{\sigma(1)}, \cdots, z_{\sigma(n)})$. The orbit problem is at the core of any method exploiting symmetry. The orbit problem asks whether two states s and s' (which in this case happen to be two 0-1 vectors of size n) are in the same orbit, i.e., there exists a permutation $\sigma \in G$ such that $s' = \sigma(s)$. In Chapter 3 it was proved that the graph isomorphism problem can be reduced to the orbit problem. Therefore, the orbit problem is harder than the graph isomorphism problem. In this section we show that the orbit problem is equivalent to the problem of finding a set stabilizer of a set Y in a coset (we call this problem SSC). Since the graph isomorphism problem can be reduced to SSC [37], this result subsumes the result which appeared in Chapter 3. Moreover, SSC (and hence the orbit problem) is equivalent to several important problems in computational group theory, which are harder than graph isomorphism, but not known to be NP-complete. Proofs of most the theorems are based on techniques introduced in [51].

The Orbit Problem (OP): Given two 0-1 vectors x and y of size n and a group $G \leq S_n$, does there exist a permutation $\sigma \in G$ which maps x to y, i.e., $y = \sigma(x)$.

Set Stabilizer in a coset (SSC): Given a set $Y \subseteq [n]$, let $G \leq S_n$ be a group and $\gamma \in S_n$ be a permutation. The problem is to find whether there exists $\sigma \in G\gamma$ which *stabilizes* the set Y, i.e., $\sigma(Y) = Y$.

Constructive Set Stabilizer in a coset ($CSSC$): Given a set $Y \subseteq [n]$, let $G \leq S_n$ be a group and $\gamma \in S_n$ be a permutation. The problem is to find whether there exists $\sigma \in G\gamma$ which stabilizes the set Y, i.e., $\sigma(Y) = Y$ and if so, to exhibit such a σ. $\sigma(Y) = Y$.

Lemma 4.5.1 The problems SSC and $CSSC$ are polynomially equivalent.

Proof: It is obvious that SSC is polynomially reducible to $CSSC$. To show the other direction, let $G, \gamma, Y \subseteq [n]$ be an instance of $CSSC$. Let G^i be the subgroup of G which fixes $\{1, 2, \cdots, i\}$. We have a chain of subgroups $I = G^{n-1} \leq \cdots \leq G^1 \leq G^0 = G$. This chain of subgroups and the right traversal of G^i/G^{i+1} can be found in polynomial time [32]. We perform the following steps:

1. First, determine using SSC whether there exists a $\sigma \in G\gamma$ such that $\sigma(Y) = Y$. If the answer is **no**, stop, otherwise perform the remianing steps.

2. Let $\{\sigma_1, \cdots, \sigma_k\}$ be the right traversal of G^0/G^1. By definition, here exists a $\sigma \in G\gamma$ such that $\sigma(Y) = Y$ iff there exists $j \leq k$ such that $\sigma \in G^1\sigma_j\gamma$ and $\sigma(Y) = Y$.

3. Make k calls to SSC (using the inputs $\{G^1, \sigma_1\gamma, Y\}, \cdots, \{G^1, \sigma_k\gamma, Y\}$) find a $j \leq k$ such that there exists a $\sigma \in G^1\sigma_j\gamma$ such that $\sigma(Y) = Y$.

4. Now use $G^1, \sigma_j\gamma, Y$ and repeat the above argument using G^1 instead of G and G^2 in place of G^1.

Iteratively, descending down the chain gives us the answer after $n-1$ steps. Since the number of elements in the right traversal of G^i/G^{i+1} is atmost n, we make less than n^2 calls to SSC $\Box$

We illustrate the proof of Theorem 4.5.1 by an example. Consider the vectors, $x = (0,1,1,0)$ and $y = (1,0,0,1)$, the group S_4, and the permutations $\sigma = (1,2)(3,4)$ and $\psi = (1,3)(2,4)$. Note that $y = \sigma(x)$ and $y = \psi(x)$. First, notice that $\pi = \sigma\psi^{-1} = (1,4)(2,3)$ is an automporhism of x, i.e., $\pi(x) = x$. In fact, given two arbitrary permutations σ_1 and σ_2 such that $\sigma_1(x) = \sigma_2(x) = y$, one can prove that $\sigma_1\sigma_2^{-1}$ is an automorphism of x. Moreover, any automorphism of x has to map a 1 to a 1 and a 0 to a 0. Therefore, an automorphism of x has to stabilize the set $\{2,3\}$. The proof is based on these observations.

Theorem 4.5.1 The problems OP and SSC are polynomially equivalent.

Proof:
($OP \Rightarrow SSC$:) Let x, y and G be given. Form a partition $\{C_0^x, C_1^x\}$ of the set $[n]$ in the following manner:

$$\begin{aligned} i \in C_0^x &\Leftrightarrow x_i = 0 \\ i \in C_1^x &\Leftrightarrow x_i = 1 \end{aligned}$$

Define C_0^y and C_1^y in a similar manner using the vector y. Find a permutation γ such that $\gamma(C_0^x) = C_0^y$ and $\gamma(C_1^x) = C_1^y$. In this case, $\gamma(x) = y$. Suppose there exists $\sigma \in G$ such that $\sigma(x) = y$. Since $\gamma(x) = y$, this means that $\sigma^{-1}\gamma(x) = x$. Or equivalently, $\sigma^{-1}\gamma(C_0^x) = C_0^x$ (recall that C_0^x is the set of positions of 0s in x). Therefore, there exists a $\psi \in (G\gamma)$ which stabilizes C_0^x ($\psi(C_0^x) = C_0^x$) iff $\psi^{-1}\gamma(x) = y$ or equivalently $\psi(x) = y$. Hence, OP can be reduced to SSC.

($SSC \Rightarrow OP$:) Let $Y \subseteq [n]$, G, and $\gamma \in S_n$ be an instance of the problem SSC. Define a 0-1 vector x in the following manner:

$$\begin{aligned} i \in Y &\Leftrightarrow x_i = 0 \\ i \in Y^c &\Leftrightarrow x_i = 1 \end{aligned}$$

Let $B_0 = \gamma(Y)$ and $B_1 = \gamma(Y^c)$. Define the 0-1 vector y such that $y_i = 0$ if $i \in B_0$ and $y_i = 1$ if $i \in B_1$. The group G in this instance of OP is just G. By the argument given in the previous part it is obvious that a permutation $\sigma \in G\gamma$ stabilizes the set Y iff $\sigma\gamma^{-1} \in G$ satisfies $y = \sigma\gamma^{-1}(x)$. $\Box$

In general, the SSC problem is hard because the *graph isomorphism* problem can be reduced to it [37]. We discuss conditions under which this problem can be solved in polynomial time.

Definition 4.5.1 Let G be a finite group. A subgroup tower

$$I = G^{(m)} \leq G^{(m-1)} \leq \cdots \leq G^{(1)} = G$$

of G is called a *subnormal series* of G. Furthurmore, if $G^{(i+1)}$ is a proper normal subgroup of $G^{(i)}$ and the factor groups $G^{(i)}/G^{(i+1)}$ are all simple, then the series is called a *composition series*.

Definition 4.5.2 For each natural number b define a class Γ_b consisting of those finite groups which have a composition series in which each factor group $G^{(i)}/G^{(i+1)}$ has order at most b.

Theorem 4.5.2 (Luks [51]) Let G be a permutation group acting on the set $[n]$ presented by a generating set consisting of at most $O(n^2)$ permutations. If G is in Γ_b, then there is a polynomial $p(x)$ whose degree depends only on b and an algorithm $\mathcal{A}$ such that algorithm $\mathcal{A}$ computes the setwise stablizer of Y in the coset $G\pi$ in time $p(n)$.

The theorem given above gives a polynomial time algorithm solving the orbit problem in a special case. The condition that G be given by at most $O(n^2)$ generators is not a restriction because it is always possible (see [32]). Given two groups A and $B \leq A$, the *centralizer* $C_A(B)$ of B in A is the following group:

$$\{a \in A \mid aBa^{-1} = B\}$$

Given a group G acting on $[n]$ and a set $Y \subseteq [n]$, the *stabilizer of* Y in G (denoted by G_Y) is the group given below.

$$\{\sigma \in G \mid \sigma(Y) = Y\}$$

In [37] it is proved that all these problems are polynomially equivalent.

Other problems

Problem 1 (Double Coset Membership) Given the groups $A, B \leq S_n$ by generating sets and the permutation $\pi, \psi \in S_n$, test whether $\psi \in A\pi B$.

Problem 2 (Group Factorization) Given the groups $A, B \leq S_n$ by generating sets and the permutation $\pi \in S_n$, test whether there are $\alpha \in A, \beta \in B$, such that $\pi = \alpha\beta$. Equivalently test whether $\pi \in AB$.

Problem 3 (Number of factorizations) Given the groups $A, B \leq S_n$ by generating sets and the permutation $\pi \in S_n$, determine the number $k \geq 0$ of distinct factorizations $\pi = \alpha\beta$ of π, where $\alpha \in A$, $\beta \in B$.

Problem 4 (Coset Intersection Emptiness) Given the groups $A, B \leq S_n$ by generating sets and given a permutation $\pi \in S_n$, test whether $A\pi \cap B$ is empty.

Problem 5 (Group Intersection) Given the groups $A, B \leq S_n$ by generating sets, determine a generating set for $C = A \cap B$.

Problem 6 (Setwise Stabilizer) Given the group $A \leq S_n$ by a generating set, and given a subset X of $[n]$, determine a generating set for the stabilizer A_X of X in A.

Problem 7 (Centralizer in Another Group) Given the groups $A, B \leq S_n$ by generating sets, determine a generating set for the centralizer $C_A(B)$ of B in A.

Problem 8 (Restricted Graph Automorphism) Given a graph $G = (V, E)$ and permutation group $A \leq Sym(V)$, determine generators for all automorphisms of G which are also in A, i.e., find generators for $A \cap Aut(G)$.

Theorem 4.5.3 The set stabilizer in a coset problem (SSC) is polynomially equivalent to Problems 1 through 8.

Proof:
We will prove that SSC is polynomially equivalent to Problem 4. Since Problem 4 is equivalent to Problems 1 through 8, the result follows. First we reduce to an instance of SSC to Problem 4. Let a set Y, a group G, and a permutation $\psi \in S_n$ be give. There exists a $\sigma \in G\psi$ which stabilizes the set Y iff $\sigma \in G\psi \cap Sym(Y)Sym([n]-Y)$. Notice that any permutation in the group $Sym(Y)Sym([n]-Y)$ stabilizes the set Y. Therefore, SSC is reducible to Problem 4.

Now we reduce Problem 4 to SSC. This part is bit more complicated. Given groups $A \leq S_n$, $B \leq S_n$, and a permutation $\pi \in S_n$, we want to test whether $A\pi \cap B$ is non empty. Consider the set $[n] \times [n]$ and the group $D = \{(\alpha, \beta) \mid \alpha \in A \text{ and } \beta \in B\}$. The permuations (α, β) act on (i,j) in the following manner $(\alpha,\beta)(i,j) = (\alpha(i), \beta(j))$. Let $Z = \{(i,i) | i \in [n]\}$ and $Z_\pi = \{(\pi^{-1}(i), i) | i \in [n]\}$. It is easy to see that there exists $\gamma \in D$ such that $\gamma(Z) = Z_\pi$ iff there exists $\gamma_1 \in D$ such that $\gamma_1(Z_\pi) = Z$ (γ^{-1} can serve as γ_1) iff there exists a $\sigma \in A\pi \cap B$. Now consider the group $H = D \wr C_2$ (C_2 is the cyclic group acting on the set $[2]$), the only permutation in C_2 is $(1,2)$). H acts on two copies of the set $[n] \times [n]$. Let Z_1 be the set corresponding to Z in the first copy and Z_2 be the set corresponding to Z_π in the second copy of $[n] \times [n]$. Let $\psi = (e, e, (1,2)$ be the permutation in G (e is the identity permutation). Let $G \leq H$ be all permuations of the form (α_1, α_2, e) such that $\alpha_1, \alpha_2 \in D$. Basically, in G we are not allowed to switch the two copies of the set $[n] \times [n]$. Now it is easy to see that $\sigma \in G\psi$ has the form $(\alpha_1, \alpha_2, (1,2))$. There exists a permutation $\sigma = (\alpha_1, \alpha_2, (1,2))$ which setwize stabilises $Z_1 \cup Z_2$ iff $\alpha_1(Z) = Z_\pi$ iff $A\pi \cap B$ is non-empty. $\Box$.

4.5.1 The Constructive Orbit Problem

Modeling states by boolean variables, in some cases, is too cumbersome and detailed. For example, consider the shared variable program introduced in Section 4.1. Let $\mathcal{P} = \|_{i=1}^{n} K_i$ be a concurrent program which does not have shared variables. Let the size of the set of locations *Loc* be k. In this case, a typical state in $\mathcal{P}$ is given by a vector of size n whose elements are integers between 1 and k, i.e., the space $[k]^n$. Permuting the processes K_i amounts to permuting the corresponding integers in that state. A symmetry group $G \leq S_n$ acts on the space $[k]^n$ in the following way: a permutation $\sigma \in G$ maps $(x_1, \cdots, x_n)$ to $(x_{\sigma(1)}, \cdots, x_{\sigma(n)})$.

Given a symmetry group G, one frequently needs a *representative function* $\xi : S \to S$ (S is the state space of the system) which has the following properties:

- s and $\xi(s)$ are in the same orbit.
- If s and s' are in the same orbit, then $\xi(s) = \xi(s')$.

Such a representative function is used during state exploration in [42]. The need to find such a representative function motivates the following problem.

Definition 4.5.3 The Constructive Orbit Problem (COP): Given a group acting on $[n]$ and vector $x = (x_1, \cdots, x_n)$ find the lexicographically least element (or lex-least element for short) in the orbit of x (the group G permutes the indices of x)

Notice that if one can solve COP in polynomial time, one can construct the representative function ξ. Given a state x, $\xi(x)$ is simply the lex-least element in its orbit. In [5] it is proved that the problem is NP-hard. The paper also shows that if the group G is in Γ_d, then COP can be solved in polynomial time. Actually, for our purposes it is enough to find a canonical element from each orbit.

4.6 Working Around the Orbit Problem

Results of section 4.5 prove that the Orbit problem is quite hard. In this section we discuss three possible techniques which will help circumvent the hardness of the orbit problem.

1. We prove that for a large class of groups, which occur commonly in practice, the orbit problem can be easily solved.

2. Given an arbitrary group G, we provide techniques to construct a subgroup $H \leq G$ such that the orbit problem for H can be solved in polynomial time. Notice that since we are working with a group which is smaller than G, we might not be getting full reduction.

3. An approach that uses multiple representatives from each orbit rather than just one was described in Chapter 3.

The two subsections outline the first two approaches.

4.6.1 Easy Groups

Notice that if a group $G \leq S_n$ has polynomial size, COP for G can be solved in polynomial time by exhaustive enumeration. For example, a rotation group acting on set of size $[n]$ has order n. Therefore, for the rotation group one can solve COP in linear time. The lemma given below states that if COP can be solved in polynomial time for two disjoint groups J and K, then COP can be solved in polynomial time for their direct product.

Lemma 4.6.1 Let $G = J \cdot K$ be a disjoint product of J and K. If *COP* for J and K can be solved in polynomial time, then *COP* for G can be solved in polynomial time.

Proof: Since J and K are disjoint, we have two disjoint sets of indices $I_J = \{i_1, \cdots, i_l\}$ and $I_K = \{j_1, \cdots, j_r\}$ such that J acts on I_J, K acts on I_K and $I_J \cup I_K = [n]$. Given a vector x, let x_J be the projection of the vector on the index set I_J. In a similar manner, let x_K be the projection on the index set I_K. We solve COP for x_J and x_K separately and put them together. □

The next lemma is similar to the previous one but considers wreath products.

Lemma 4.6.2 Let $G = J \wr K$. The group J, K, G act on the sets $[n], [m], [nm]$ respectively. If *COP* for J, K can be solved in polynomial time, then *COP* for G can be solved in polynomial time.

Proof: Let $x = (x_1, \cdots, x_{mn})$ be a vector of size mn. Recall that each permutation in $G = J \wr K$ can be thought of as a tuple of permutations $(\sigma_1, \cdots, \sigma_n, \gamma)$ such that $\sigma_i \in J$ and $\gamma \in K$. Let $B_i = (x_{ni}, \cdots, x_{ni+n-1})$ be the i-th block. We permute the blocks according to the permutation γ and then permute the elements in the block B_i according to the permutation σ_i. Regard B_i as an integer by concatenating all the integers in that block. Solve the COP for $(B_1, \cdots, B_m)$ and the group K and get a lex-least element $(B_{\gamma(1)}, \cdots, B_{\gamma(n)})$ where $\gamma \in K$. Now solve the COP for each $B_{\gamma(i)}$ and J. $\Box$

Lemma 4.6.3 Let S_n be the full symmetric group acting on the set $[n]$. The COP problem for S_n can be solved in polynomial time.

Proof: Given a vector $x = (x_1, \cdots, x_n)$, the lex-least element of x under the group S_n can be obtained by sorting the elements x_i. $\Box$.

Frequently in practice, symmetries are given as a set of transpositions. For example, a system which has the star topology, the two outer processes can be switched. The lemma given below states that if the group is only generated by transpositions, then COP for it can be solved in polynomial time.

Lemma 4.6.4 Let G be a permutation group acting on the set $[n]$. Assume that G is generated by a set of transpositions S. The COP problem for G can be solved in polynomial time.

Proof: Consider a graph $K = ([n], E)$ with n vertices. Let S be the set of transpositions generating the group G. The edge $\langle i, j \rangle \in E$ iff $(i, j) \in S$. Let $C_1, C_2, \cdots, C_m$ be the connected components of the graph K. We will prove that each connected component corresponds to a full-symmetric group acting on the vertices in that connected component. Without loss of generality let $\{1, \cdots, k\}$ be the vertices in the connected component C_1 (otherwise we can rename the vertices). We will prove that $(1, r) \in G$ for $2 \leq r \leq k$. Let $\langle 1, i_1 \rangle, \cdots, \langle 1, r \rangle$ be the path from vertex 1 to r. Composing all the transpositions along the path, we get that $(1, r) \in G$. Since $\{(12), (13), \cdots, (1k)\}$ generates the full symmetric group S_k, the connected component C_1 corresponds to S_k. Therefore, G is the disjoint direct product of m full symmetric groups (one group for each connected component). Now using lemma 4.6.3 and lemma 4.6.1 we get that COP for G can be solved in polynomial time. □

4.6.2 Finding Easy Subgroups

Let $G \leq S_n$ be a permutation group acting on the set $\{1, 2, \cdots, n\} = [n]$. We will find a subgroup G' of G such that the orbit problem for G' is solvable in polynomial time.

Definition 4.6.1 Given G acting on $[n]$ define $G^{[i,k]}$ as the subgroup of G which fixes the set $T = [n] - \{i, i+1, \cdots, i+k-1\}$, i.e., for every $j \in T$ and every $\sigma \in G^{[i,k]}$ we have that $\sigma(j) = j$.

Note that the generator for the subgroups $G^{[i,k]}$ can be found in polynomial time [32]. Let $m = \lceil \frac{n}{k} \rceil$. Given G acting on $[n]$, let G' be defined by the following equation:

$$G' = \left(\prod_{i=1}^{m} G^{[k(i-1)+1,k]} \right) \cdot G^{[km+1,n-m]}$$

In the formula given above, $\cdot$ denotes disjoint product of groups. Since each component in the formula given above is subgroup of G, it follows trivially that G' is a subgroup of G. Intuitively, G' is the subgroup of G in which we retrict ourselves to permute only k indices.

Lemma 4.6.5 The orbit problem for G' can be solved in polynomial (in n and k) time.

Proof: Notice that each component $G^{[i,k]}$ permutes a disjoint set of indices. Furthurmore, since $G^{[i,k]}$ only permutes k indices, the order of $G^{[i,k]}$ is $\leq k! \leq k^{\frac{k}{2}}$. Given two 0-1 vectors x and y, we can divide the orbit problem for G' into $m+1$ orbit problems. Each of these subproblems involves a subgroup of G which permutes only k indices and hence can be solved in time $k^{\frac{k}{2}}$. Therefore, the orbit problem for G' can be solved in time $O(nk^{\frac{k}{2}})$ □.

In practice the user will choose k and the group G' will be computed automatically. For example, if the user restricts himself to exchanging two adjacent processes, then k will correspond to the total number of local bits of two processes. Given a symmetry group G acting on $[n]$, let $H \subseteq G$ be the subgroup generated by the set of transpositions $S \subseteq G$ given below.

$$S = \{(i,j) \mid (i,j) \in G\}$$

Since one can test $(i,j) \in G$ in polynomial time [32], the set S can be found in polynomial time (there are only n^2 transpositions). By lemma 4.6.4 COP for H can be solved in polynomial time. Therefore, if we work with H instead of G, we are solving an easy instance of the orbit problem.

Chapter 5

Partial Order and Symmetry

Partial order based methods exploit the independence of actions [36, 60, 70, 71]. The basic idea is that given a set of interleaving sequence of actions, some of these sequences can be ignored because the actions occurring in them are independent and hence can be permuted. The method works by partitioning the sequences into equivalence classes, and then the algorithm explores only few sequences from each equivalence class. For example, assume that the actions α and β are independent. Independence means that it does not matter in what order the finite-state system executes the actions α and β. So a sequence $u\alpha\beta$ is equivalent to the sequence $u\beta\alpha$. Therefore, if we consider the sequence $u\alpha\beta$, we can ignore the sequence $u\beta\alpha$. Most methods work by choosing a small set of actions from a state. While doing depth-first search to explore the state-space, we only execute these small set of actions from the state. The enabled actions have the property that the method considers at least one sequence of actions from each equivalence class.

In chapter 3 symmetry based methods to avoid the state-explosion problem during model-checking were described. This chapter combines symmetry and partial order based reduction techniques. In a system comprised of states and actions, partial order based techniques exploit the independence of actions. On the other hand, symmetry based techniques exploit the symmetry on states. Since symmetry and partial order based methods work on different components of the system, it should be possible for both techniques to be applied simultaneously. In this chapter it is proved that this is indeed the case.

The chapter is organized as follows: Section 5.1 provides definitions used throughout the chapter. Section 5.2 gives an algorithm which preserves LTL formulas without the nexttime operator. Section 5.3 provides an algorithm which preserves $CTL^{\star}$ without the nexttime operator.

5.1 Definitions

In this section we introduce various definitions used throughout the chapter. Subsection 5.1.1 defines different pre-orders between LTSs. The notion of bisimulation preserving abstractions is defined in Subsection 5.1.2. The subsection after that defines what it means for two actions to be independent. The last subsection defines what it means for a LTS to be symmetric. Next, we introduce the concept of a labeled transition system (LTS).

Let AP be a set of atomic propositions. A *labeled transition system* (LTS) is 5-tuple $T = (S, R, L, Act, s_0)$, where

- S is a finite set of *states*,
- $R \subseteq S \times Act \times S$ is a *transition relation* ($(s, \alpha, s') \in R$ is also written as $s \stackrel{\alpha}{\rightarrow} s' \in R$).
- $L : S \rightarrow 2^{AP}$ is a *labeling function* which associates with each state a set of atomic propositions that are true in the state.
- Act is a finite set of actions.
- s_0 is the initial state.

The set $\alpha_T(s)$ is the set of all α-successors of s in T, i.e., $s' \in \alpha_T(s)$ iff $s \stackrel{\alpha}{\rightarrow} s' \in R$. An action α is said to be *enabled* from a state s in T if and only if there exists a s' such that $s \stackrel{\alpha}{\rightarrow} s' \in R$. The symbol $en_T(s)$ denotes the set of actions enabled from the state s in T. An action α is called *invisible* in T iff for all s and s' such that $s \stackrel{\alpha}{\rightarrow} s' \in R$ we have that $L(s) = L(s')$. Basically, an invisible action does not change the truth of atomic propositions. The set of invisible actions in T is denoted by $\mathtt{invis}_T$. The set of visible actions is denoted by $\mathtt{vis}_T$.

5.1.1 Various pre-orders between processes

Some pre-orders were defined in chapter 2, but we define them in this chapter in the context of LTSs. Given two LTSs $T_1 = (S_1, R_1, L_1, Act, s_{0,1})$ and $T_2 = (S_2, R_2, L_2, Act, s_{0,2})$, a relation $\mathcal{B} \subseteq S_1 \times S_2$ is called a *bisimulation* between T_1 and T_2 if and only if the following conditions hold:

- $s_{0,1}\ \mathcal{B}\ s_{0,2}$.
- Assume that $s\ \mathcal{B}\ s'$. Then the following conditions hold:
  - $L(s) = L(s')$
  - Given an arbitrary transition $s \stackrel{\alpha}{\rightarrow} s_1 \in R_1$, there exists $s_2 \in S_2$ such that $s' \stackrel{\alpha}{\rightarrow} s_2 \in R_2$ and $s_1\ \mathcal{B}\ s_2$.
  - A symmetric condition holds with roles of s' and s reversed.

T_1 and T_2 are said to be *bisimilar* (denoted by $T_1 \cong_B T_2$) if and only if there exists a bisimulation between T_1 and T_2.

Definition 5.1.1 Let $T_1 = (S_1, R_1, L_1, Act, s_{0,1})$ and $T_1 = (S_2, R_2, L_2, Act, s_{0,2})$ be two LTSs. Let $\mathcal{E} \subseteq S_1 \times S_2$ be a relation. Consider paths $\pi = s_0 \stackrel{\alpha_0}{\rightarrow} s_1 \stackrel{\alpha_1}{\rightarrow} \cdots$ in T_1 and $\pi' = t_0 \stackrel{\beta_0}{\rightarrow} t_1 \stackrel{\beta_1}{\rightarrow} \cdots$ in T_2. Paths π and π' are called *stuttering $\mathcal{E}$-equivalent* if and only if there exists sequences of natural numbers $i_0 = 0 < i_1 < i_2 < \cdots$ and $k_0 = 0 < k_1 < k_2 < \cdots$ such that for all $j \geq 0$ the following condition is true.

- For all $i_j \leq r < i_{j+1}$ and $k_j \leq m < k_{j+1}$, $s_r\ \mathcal{E}\ t_m$.

Paths π and π' are called *stuttering equivalent* if they are stuttering $\mathcal{L}$-equivalent where $s\ \mathcal{L}\ s'$ if and only if $L(s) = L(s')$. Sometimes, we will refer to the set of integers $\{i_j, i_j + 1, \cdots, i_{j+1} - 1\}$ and $\{k_j, k_j + 1, \cdots, k_{j+1} - 1\}$ as the *j-th blocks* B_j and B'_j.

Next, we define the notion of *stuttering bisimulation.* Stuttering bisimulation is similar to bisimulation, but each LTS is allowed to take several steps to simulate a path of the other LTS. Given two LTSs $T_1 = (S_1, R_1, L_1, Act, s_{0,1})$ and $T_2 = (S_2, R_2, L_2, Act, s_{0,2})$, a relation $\mathcal{E} \subseteq S_1 \times S_2$ is called a *stuttering bisimulation* between T_1 and T_2 if and only if the following conditions hold:

- $s_{0,1}\ \mathcal{E}\ s_{0,2}$.
- If $s\ \mathcal{E}\ s'$, then the following conditions hold:
  - $L(s) = L(s')$
  - For every path π starting from s in T_1 there exists a stuttering $\mathcal{E}$-equivalent path π' starting from s' in T_2. See definition 5.1.1 for the explanation of stuttering $\mathcal{E}$-equivalent.
  - The same condition as the previous one holds but with the roles of s and s' reversed.

T_1 and T_2 are said to be *stuttering bisimilar* if and only if there exists a stuttering bisimulation between them. We denote this by $T_1 \cong_{SB} T_2$.

$T_1 = (S_1, R_1, L_1, Act, s_{0,1})$ and $T_2 = (S_2, R_2, L_2, Act, s_{0,2})$ are said to be *stuttering path equivalent* (denoted by $T_1 \cong_{SPE} T_2$) if and only if

- For every path π starting from $s_{0,1}$ in T_1 there exists a stuttering equivalent path π' starting from $s_{0,2}$ in T_2.
- A symmetric condition holds with the roles of $s_{0,1}$ and $s_{0,2}$ reversed.

$T_1 = (S_1, R_1, L_1, Act, s_{0,1})$ and $T_2 = (S_2, R_2, L_2, Act, s_{0,2})$ are said to be *path equivalent* (denoted by $T_1 \cong_{PE} T_2$) if and only if

- For every path π starting from $s_{0,1}$ in T_1 there exists a path π' starting from $s_{0,2}$ in T_2 such that $L_1(\pi[0]) = L_2(\pi'[0])$.
- A symmetric condition holds with the roles of $s_{0,1}$ and $s_{0,2}$ reversed.

Next, we define the notion of *stammering bisimulation*. Stammering bisimulation is a stronger equivalence than stuttering bisimulation. Intuitively, each transition in one structure is simulated by a finite sequence of transitions in the other structure. The formal definition is given below:

Definition 5.1.2 Given two LTSs $T_1 = (S_1, R_1, L_1, Act, s_{0,1})$ and $T_2 = (S_2, R_2, L_2, Act, s_{0,2})$ a relation $\mathcal{SB} \subseteq S_1 \times S_2$ is called a *stammering simulation* if and only if $s_{0,1}\ \mathcal{SB}\ s_{0,2}$ and if $s\ \mathcal{SB}\ s'$, then the following conditions hold:

1. $L(s) = L(s')$.

2. If $s \xrightarrow{\beta} v$ is a transition in T_1, then

   - $\beta \in \texttt{invis}_{T_1}$ and $v\ \mathcal{SB}\ s'$, or
   - there exists a path $s' = t_0 \xrightarrow{\alpha_0} t_1 \xrightarrow{\alpha_1} \cdots t_n \xrightarrow{\alpha_n} v'$ in T_2 such that for $0 \leq i < n$, $s\ \mathcal{SB}\ t_i$ and $\alpha_i \in \texttt{invis}_{T_2}$. Moreover, $v\ \mathcal{SB}\ v'$.

3. If there is an infinite path $s = v_0 \xrightarrow{\beta_0} v_1 \xrightarrow{\beta_1} v_2 \cdots$ in T_1 such that for all $i \geq 0$, $\beta_i \in \texttt{invis}_{T_1}$ and $v_i\ \mathcal{SB}\ s'$, then there exists a path $s' = t_0 \xrightarrow{\alpha_0} t_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{j-1}} t_j \xrightarrow{\alpha_j} t_{j+1}$ in T_2 such that $s\ \mathcal{SB}\ t_i$ and $\alpha_i \in \texttt{invis}_{T_2}$ for $0 \leq i \leq j$, and $v_1\ \mathcal{SB}\ t_{j+1}$.

A relation $\mathcal{SB}$ is called a *stammering bisimulation* if and only if $\mathcal{SB}$ and $\mathcal{SB}^{-1}$ are both *stammering simulations*. T_1 and T_2 are called *stammering bisimilar* (denoted by $T_1 \equiv_{StB} T_2$) if and only if there exists a stammering bisimulation between them.

Notice that a stuttering bisimulation relation cannot distinguish between next states. Therefore, it is not surprising that stuttering bisimulation preserves the truth of CTL^* formula without the next time operator. The proof of this theorem first appeared in [9].

Theorem 5.1.1 Let f be a formula in $CTL^\star$-X. Let T_1 and T_2 be two stuttering bisimilar LTSs. Let $\mathcal{E}$ be a stuttering bisimulation relation between T_1 and T_2. If $s\ \mathcal{E}\ s'$, then $T_1, s \models f$ if and only if $T_2, s' \models f$.

The proof of this Theorem follows from the Lemma given below.

Lemma 5.1.1 Let T_1 and T_2 be two stuttering bisimilar LTSs. Let $\mathcal{E}$ be the stuttering bisimulation between T_1 and T_2. Assume that $s\ \mathcal{E}\ s'$. Let π and π' be two stuttering $\mathcal{E}$-equivalent paths in T_1 and T_2 respectively. Assume that f is state formula and g is a path formula in $CTL^\star$-X. In this case we have:

1. $s \models f$ iff $s' \models f$.

2. $\pi \models g$ iff $\pi' \models g$.

Proof:

Basis: $f \in AP$. By definition, $L(s) = L(s')$. Therefore, $s \models f$ if and only if $s' \models f$.

Induction: There are several cases.

- $f = \neg f_1$, a state formula.
  By the inductive hypothesis we have that $s \models f_1 \Leftrightarrow s' \models f_1$. Therefore $s \models f \Leftrightarrow s' \models f$. The same reasoning holds in case of a path formula.

- $f = f_1 \vee f_2$, a state formula

$$\begin{aligned} s \models f &\Leftrightarrow s \models f_1 \text{ or } s \models f_2 \\ &\Leftrightarrow s' \models f_1 \text{ or } s' \models f_2 \\ &\Leftrightarrow s' \models f \end{aligned}$$

  The second step uses the inductive hypothesis. We can also use this argument in case of the path formula.

- $f = E(f_1)$, a state formula
  Suppose $s \models f$. There is a path π starting with s such that $\pi \models f_1$. By definition, there exists a stuttering $\mathcal{E}$-equivalent path π' in T_2 starting with s'. By the inductive hypothesis $\pi \models f_1 \Leftrightarrow \pi' \models f_1$. Therefore, $s \models E(f_1) \Rightarrow s' \models E(f_1)$. A similar argument holds in the other direction.

- $g = f$, where g is a path formula and f is a state formula.
  Although the lengths of g and f are the same, we can imagine that $g = \mathit{path}(f)$, where *path* is an operator which converts a state formula into a path formula. Now we can apply the inductive step.

- $g = g_1 \mathbf{U} g_2$.
  Assume that π and π' are stuttering $\mathcal{E}$-equivalent. By definition, we have two strictly increasing sequences of natural numbers $\{i_j\}_{j=0}^{\infty}$ and $\{k_j\}_{j=0}^{\infty}$ such that $i_0 = k_0 = 0$ and for all $j > 0$ we have that:

- For all $i_j \leq m \leq i_{j+1}$ and $k_j \leq n < k_{j+1}$, $\pi[m]\ \mathcal{E}\ \pi'[n]$

We will prove that $\pi \models g$ implies that $\pi' \models g$. Assume that $\pi \models g$. By definition, there exists a $k \geq 0$ such that $\pi^k \models g_2$ and for all $0 \leq j < k$, $\pi^j \models g_1$. Let m be a natural number such that $i_m \leq k < i_{m+1}$. Notice that π'^{k_m} is stuttering $\mathcal{E}$-equivalent to π^k. Moreover, for every $0 \leq j < k$ we can find a natural number $0 \leq r < k_m$ such that π^j is stuttering $\mathcal{E}$-equivalent to π'^r. To see this, consider an arbitrary j. Let r be such that $i_r \leq j < i_{r+1}$. We have that π^j is stuttering $\mathcal{E}$-equivalent to π'^l for all l $(k_r \leq l < k_{r+1})$. By the induction hypothesis $\pi'^{k_m} \models \phi_2$ and $\pi'^r \models \phi_1$ for all $0 \leq r < k_m$. The proof that $\pi' \models \phi$ implies $\pi \models \phi$ is symmetric.

- $g = g_1\ \mathbf{V}\ g_2$
  Again, Assume that π and π' are stuttering $\mathcal{E}$-equivalent. Assume that we have two increasing sequences of natural numbers $\{i_j\}_{j=0}^{\infty}$ and $\{k_j\}_{j=0}^{\infty}$ as before. The proof is very similar to the one given above, so we will give a brief proof in this case. Assume that $\pi \models g$. We will prove that $\pi' \models g$. By definition, for all $k \geq 0$ if $\pi^j \not\models g_1$ for all $0 \leq j < k$, then $\pi^k \models g_2$. Let $k \geq 0$ be an arbitrary natural number such that $\pi'^j \not\models g_1$ for $0 \leq j < k$. Let $m \geq 0$ be such that $k_m \leq k < k_{m+1}$. It is easy to see that $\pi^r \not\models g_1$ for $0 \leq r < i_{m+1}$ (use the induction hypothesis and the fact that $\pi \models g$). Also, $\pi'^k \models g_2$ by the induction hypothesis and the fact that $\pi^r \models g_2$. Notice that π'^k is stuttering equivalent to π^r. Using the definition of the $\mathbf{V}$ operator it follows that $\pi' \models \phi$. The proof that $\pi' \models \phi$ implies $\pi \models \phi$ is symmetric.

Theorem 5.1.2 Let f be a formula in LTL-X. Let $T_1 = (S_1, R_1, L_1, Act, s_{0,1})$ and $T_2 = (S_2, R_2, L_2, Act, s_{0,2})$ be two stuttering path equivalent LTSs.

$$s_{0,1} \models f \quad \Leftrightarrow \quad s_{0,2} \models f$$

The proof of the theorem follows from the lemma which states that stuttering equivalent paths cannot distinguish between LTL-X formula.

Lemma 5.1.2 Let π and π' be two paths that are stuttering equivalent. Let ϕ be a formula with only the path operators **U** and **V**. We have that:

$$\pi \models \phi \quad \Leftrightarrow \quad \pi' \models \phi$$

Proof: Let $i_0 = 0 < i_1 < i_2 < \cdots$ and $k_0 = 0 < k_1 < k_2 < \cdots$ be the two sequences corresponding to π and π'. The sequences satisfy the conditions for π and π' to be stuttering equivalent. The proof is by structural induction on ϕ.

- **(Case** $\phi = p$**)**
  The formula ϕ is an atomic proposition. Since $L(\pi[0]) = L(\pi'[0])$, we have that $\pi \models p$ iff $\pi' \models p$.

- **(Case** $\phi = \neg\phi_1$**)**
  The result follows from structural induction.

- **(Case** $\phi = \phi_1 \, \mathbf{U} \, \phi_2$**)**
  We will prove that $\pi \models \phi$ implies that $\pi' \models \phi$. Assume that $\pi \models \phi$. By definition, there exists a $k \geq 0$ such that $\pi^k \models \phi_2$ and for all $0 \leq j < k$, $\pi^j \models \phi_1$. Let m be a natural number such that $i_m \leq k < i_{m+1}$. Notice that π'^{k_m} is stuttering equivalent to π^k. Moreover, for every $0 \leq j < k$ we can find a natural number $0 \leq r < k_m$ such that π^j is stuttering equivalent to π'^r. To see this, consider an arbitrary j. Let r be such that $i_r \leq j < i_{r+1}$. We have that π^j is stuttering equivalent to π'^l for all l $(k_r \leq l < k_{r+1})$. By the induction hypothesis $\pi'^{k_m} \models \phi_2$ and $\pi'^r \models \phi_1$ for all $0 \leq r < k_m$. The proof that $\pi' \models \phi$ implies $\pi \models \phi$ is symmetric.

- **(Case** $\phi = \phi_1 \, \mathbf{V} \, \phi_2$**)**
  The proof is very similar to the one given above, so we will give a brief proof in this case. Assume that $\pi \models \phi$. We will prove that $\pi' \models \phi$. By definition, for all $k \geq 0$ if $\pi^j \not\models \phi_1$ for all $0 \leq j < k$, then $\pi^k \models \phi_2$. Let $k \geq 0$ be an arbitrary natural number such that $\pi'^j \not\models \phi_1$ for $0 \leq j < k$. Let $m \geq 0$ be such that $k_m \leq k < k_{m+1}$. It is easy to see that $\pi^r \not\models \phi_1$ for $0 \leq r < i_{m+1}$ (use the induction hypothesis). Also, $\pi'^k \models \phi_2$ by the induction hypothesis and the

fact that $\pi^r \models \phi_2$. Using the definition of the **V** operator it follows that $\pi' \models \phi$. The proof that $\pi' \models \phi$ implies $\pi \models \phi$ is symmetric.

The basic idea in the proof is that within a block i_m and i_{m+1} (or k_m and k_{m+1}) the truth of a path formula does not change.□

Lemma 5.1.3 $T_1 \cong_B T_2$ implies that $T_1 \cong_{SB} T_2$. $T_1 \cong_{StB} T_2$ implies that $T_1 \cong_{SB} T_2$. Similarly, $T_1 \cong_{SB} T_2$ implies that $T_1 \cong_{SPE} T_2$.

Proof: Assume that T_1 and T_2 are bisimilar. Let $\mathcal{B}$ be a bisimulation relation between T_1 and T_2. Assume that $s \ \mathcal{B} \ s'$. Let π be a path starting from s in T_1. By definition, there exists a path π' starting from s' in T_2 such that $\pi[i] \ \mathcal{B} \ \pi'[i]$ (for all i). Paths π and π' are stuttering $\mathcal{B}$-equivalent with the $\{i_j = j\}_{j=0}^{\infty}$ and $\{k_j = j\}_{j=0}^{\infty}$. The other case is symmetric.

Now assume that T_1 and T_2 are stammering bisimilar. Let $\mathcal{E}$ be a stammering bisimulation between T_1 and T_2. Let $s \ \mathcal{E} \ s'$. By definition, $L(s) = L(s')$. Let π be a path starting from s in T_1. We will build a stuttering $\mathcal{E}$-equivalent path π' starting from s' in T_2. Assume that π is $s = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots$. We will construct a path π' (denoted by $s' = t_0 \xrightarrow{\beta_0} t_1 \xrightarrow{\beta_1} \cdots$) inductively. We maintain the following counters.

i: state s_i in path π is being processed.
j: length of the path π' so far.
k: current block number for π.
l: current block number for π'.

Initially, all the counters are 0. First, we will handle the second case in the definition of stammering bisimulation. Suppose we are processing the transition $s_i \xrightarrow{\alpha_i} s_{i+1}$ of the path π. We have the following two subcases.

- Suppose α_i is invisible and $s_{i+1} \ \mathcal{E} \ t_j$
  In this case we update $i = i + 1$.

- Suppose there exists a path $\pi_1 = t_j \stackrel{\beta_j}{\rightarrow} t_{j+1} \stackrel{\beta_{j+1}}{\rightarrow} \cdots \stackrel{\beta_{j+m-1}}{\rightarrow} t_{j+m}$ such that $0 \leq r < m$, $s_i \; \mathcal{E} \; t_{j+r}$ and $\beta_i \in \mathtt{invis}_{T_1}$, and $s_{i+1} \; \mathcal{E} \; t_{j+m}$. We append π_1 to π'. We also make the following updates: $i = i+1$, $j = j+m$, $k = k+1$, and $l = l+1$. Notice that we started a new block at state s_{i+1} and t_{j+m}.

The third case in the definition of stammering bisimulation is handled in a similar manner to the previous case. It is not hard to see that π' is stuttering $\mathcal{E}$-equivalent to π. The block numbers can be used to construct the stuttering equivalence between paths π and π'.

Let $\mathcal{E}$ be a stuttering bisimulation between T_1 and T_2. By definition, $s \; \mathcal{E} \; s'$ implies that $L(s) = L(s')$. From this it follows that if π and π' are stuttering $\mathcal{E}$-equivalent then they are stuttering equivalent. Since the initial state of T_1 and T_2 are related by $\mathcal{E}$, this shows that T_1 and T_2 are stuttering path equivalent. □

Lemma 5.1.4 The pre-orders $\cong_B$, $\cong_{SB}$, $\cong_{StB}$ and $\cong_{SPE}$ are transitive.

Proof: Proof follows from the definitions.□

The table given below summarizes the relationships between various pre-orders and temporal logics. The first column lists all the pre-orders. The second column lists various logics. If a pre-order and logic appear in the same row, then they are *equivalent*. For example, $T_1 \cong_{PE} T_2$ iff T_1 and T_2 satisfy the same LTL formulas.

Pre-order	Logic
$\cong_B$	$CTL^\star$
$\cong_{SB}$	$CTL^\star$-X
$\cong_{PE}$	LTL
$\cong_{SPE}$	LTL-X

5.1.2 Abstractions

Let $T = (S, R, L, Act, s_0)$ be an LTS. Let $h : S \rightarrow S$ be an *abstraction function*. We say that h is *bisimulation preserving* if and only if there exists a bisimulation relation $\mathcal{B} \subseteq S \times S$ between T and T such that;

- For all $s \in S$, $s \; \mathcal{B} \; h(s)$

- $s \mathcal{B} s'$ implies that $h(s) = h(s')$

In this case, we say that h *preserves* the bismulation relation $\mathcal{B}$. Intuitively, h picks a representative from each equivalence class of S induced by the bisimulation $\mathcal{B}$. Given a bisimulation preserving abstraction function h on an LTS $T = (S, R, L, Act, s_0)$, define the corresponding *abstract LTS* $T_h = (S_h, R_h, L_h, Act, h(s_0))$ in the following manner:

- $S_h = h(S)$.
- $r_1 \xrightarrow{\alpha} r_2 \in R_h$ if and only if there exists $s \in S$ such that $r_1 \xrightarrow{\alpha} s \in R$ and $h(s) = r_2$.
- For all $r \in S_h$, $L_h(r) = L(r)$.

Lemma 5.1.5 Given an LTS $T = (S, R, L, Act, s_0)$ and a bisimulation preserving abstract function h, T and T_h are bisimilar.

Proof: Let $\mathcal{B}$ be a bisimulation between T and T such that h preserves $\mathcal{B}$. Construct $\mathcal{B}_h \subseteq S \times S_h$ in the following way:

$$s \; B_h \; r \quad \Leftrightarrow \quad s \; \mathcal{B} \; r$$

We will prove that B_h is a bisimulation relation. Assume that $s \; \mathcal{B}_h \; r$. It is obvious that the labels of r and s match.

- Assume that $s \xrightarrow{\alpha} s' \in R$.
  Since $s \; \mathcal{B} \; r$, there exists a s_1 such that $r \xrightarrow{\alpha} s_1 \in R$ and $s' \; \mathcal{B} \; s_1$. By definition, $r \xrightarrow{\alpha} h(s_1) \in R_h$ and $s_1 \; \mathcal{B} \; h(s_1)$. By transitivity, $s' \; \mathcal{B} \; h(s_1)$, which implies that $s' \; B_h \; h(s_1)$.
- Assume that $r \xrightarrow{\alpha} r' \in R_h$.
  The definition of R_h implies that there exists s_1 such that $r \xrightarrow{\alpha} s_1 \in R$ and $h(s_1) = r'$. Since $s \; \mathcal{B} \; r$, there exists s' such that $s \xrightarrow{\alpha} s'$ and $s' \; \mathcal{B} \; s_1$. Using the fact that $s_1 \; \mathcal{B} \; r'$ (recall that $h(s_1) = r'$ and h is bismulation preserving) and transitivity of $\mathcal{B}$, $s' \; \mathcal{B} \; r'$. This implies that $s' \; B_h \; r'$.

By definition of B_h, $s_0 \; B_h \; h(s_0)$. The proof is complete.$\square$

5.1.3 Independent Actions

Now we define the concept of independent actions. The notion of independence is central to the partial-order reduction techniques.

Definition 5.1.3 Let $T = (S, R, L, Act, s_0)$ be an LTS. An *independence relation* on actions is an irreflexive and symmetric relation $I \subseteq Act \times Act$ such that for each pair of actions $(\alpha, \beta) \in I$ (called *independent actions*) it must hold that for each $s \in S$

- If $\{\alpha, \beta\} \subseteq en_T(s)$, then for each state $s' \in \alpha_T(s)$ we have that $\beta \in en_T(s')$.
- If $\{\alpha, \beta\} \subseteq en_T(s)$, then there exists a path from $s \xrightarrow{\alpha} s_1 \xrightarrow{\beta} s'$ in T iff there exists a path $s \xrightarrow{\beta} s'_1 \xrightarrow{\alpha} s'$ in T.

The first condition states that if α and β are independent, then executing α from a state s, does not disable the action β. The second condition states that independent actions are commutative (see Figure 5.1). Notice that I is an independence relation with respect to a particular LTS T.

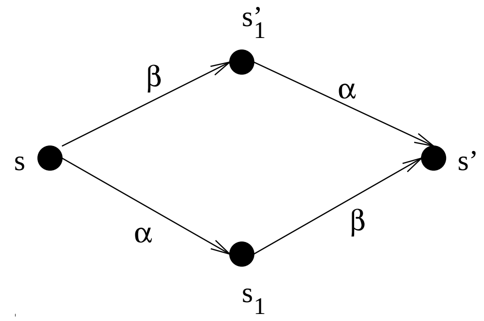

Figure 5.1: Commutativity of actions

The lemma given below states that if I is an independence relation for T, then I is also an independence relation for T_h. This means that given an independence relation for T, we can use the same independence relation while performing partial-order reduction on the abstract LTS T_h.

Lemma 5.1.6 Let I be an independence relation for a LTS $T = (S, R, L, Act, s_0)$. Let h be a bisimulation preserving abstraction function. Let $T_h = (S_h, R_h, L_h, Act, h(s_0))$ be the corresponding abstract LTS. Then, I is also an independence relation for T_h

Proof: Let I be an independence relation for T. We will prove that I is also an independence relation for I. Let $\mathcal{B}$ be a bisimulation relation between T and T which is preserved by h. Assume that $(\alpha, \beta) \in I$. Corresponding to the two conditions in the definition of the independence relation we have the following two cases.

- Assume that $\{\alpha, \beta\} \subseteq en_{T_h}(r)$. Let $r' \in \alpha_{T_h}(r)$. We have to prove that $\beta \in en_{T_h}(r')$. By definition, there exists $s \in S$ such that $r \stackrel{\alpha}{\rightarrow} s$ and $h(s) = r'$. Since I is an independence relation for T, $\beta \in en_T(s)$. Since h is bismulation preserving, we also have that $s \;\mathcal{B}\; r'$. Therefore, $\beta \in en_T(r')$, which in turn implies that $\beta \in en_{T_h}(r')$.

- Assume that $\{\alpha, \beta\} \subseteq en_{T_h}(r)$. Now suppose that there exists a path $r \stackrel{\alpha}{\rightarrow} r_1 \stackrel{\beta}{\rightarrow} r'$ in T_h. Let $s \in S$ such that $r \stackrel{\alpha}{\rightarrow} s \in R$ and $h(s) = r_1$. Also assume that $r_1 \stackrel{\beta}{\rightarrow} t_1 \in R$ and $h(t_1) = r'$. Since $\beta \in en_T(s)$, we can construct a path $r \stackrel{\alpha}{\rightarrow} s \stackrel{\beta}{\rightarrow} s'$ in T such that $s' \;\mathcal{B}\; t_1$. Since I is an independence relation for T, there exists a path $r \stackrel{\beta}{\rightarrow} s_1 \stackrel{\alpha}{\rightarrow} s'$ in T. A transition $h(s_1) \stackrel{\alpha}{\rightarrow} s''$ such that $s' \;\mathcal{B}\; s''$ exists because $s_1 \;\mathcal{B}\; h(s_1)$. By transitivity, $s'' \;\mathcal{B}\; t_1$. Therefore, $h(s'') = h(t_1) = r'$. Hence, $r \stackrel{\beta}{\rightarrow} h(s_1) \stackrel{\alpha}{\rightarrow} r'$ is a path in T_h.

Only the second part of the proof uses the fact that $s \;\mathcal{B}\; s'$ implies that $h(s) = h(s')$.□

Now we proceed to define an equivalence relation on paths which is induced by the independence relation. As usual $\Sigma^\star \cup \Sigma^\omega$ denotes the set of infinite and finite strings over the alphabet Σ. The symbol $\alpha[i]$ denotes the i-th letter in the string α. Given a relation E on the set Σ, a relation $\equiv_E$ on $\Sigma^\star \cup \Sigma^\omega$ is defined as follows. First, we define $\equiv_E$ on finite strings. Given two strings α and β, we say that $\alpha \equiv_E \beta$ iff there exists a sequence $\alpha_0, \cdots, \alpha_n$, where $\alpha_0 = \alpha$ and $\alpha_n = \beta$ and for each $0 \leq i \leq n$, $\alpha_i = \bar{\alpha} a a' \hat{\alpha}$ and $\alpha_{i+1} = \bar{\alpha} a' a \hat{\alpha}$ for some finite strings $\bar{\alpha}$ and

$\hat{\alpha}$, and aEa'. That is, we say two strings are related if the second string can be obtained from the first by permuting E-related letters. Given two finite strings $\alpha, \beta \in \Sigma^\star$, we say that $\alpha \preceq_E \beta$ iff there exists $\gamma \in \Sigma^\star$ such that $\beta \equiv_E \gamma$ and α is a prefix of γ. Next, we extend the definition to infinite strings. We say that $\alpha \preceq_E \beta$ iff every finite prefix u of α there exists a finite prefix w of β such that $u \preceq_E w$. Two infinite strings α and β are $\equiv_E$-related iff $\alpha \preceq_E \beta$ and $\beta \preceq_E \alpha$. If E is a irreflexive and symmetric relation on Σ, then one can show that $\equiv_E$ is an equivalence relation on strings [54]. Given an irreflexive and symmetric relation E, a *trace* is an equivalence class of finite or infinite strings induced by the equivalence relation $\equiv_E$. Given a string v, $[v]_E$ is the trace which contains v. Notice that if $v \preceq_E w$, then for all $z \in [v]_E$ and $y \in [w]_E$ it is true that $z \preceq_E y$. Therefore, we can extend the preorder $\preceq_E$ to traces in a natural way. Formally, $[v]_E \preceq_E [w]_E$ iff $v \preceq_E w$. Sometimes when the relation E is clear from the context, we write $[a]$ instead of $[a]_E$.

Let I be an independence relation on a LTS $T = (S, R, L, Act, s_0)$. Define $\equiv_I$ on the set $Act^\star \cup Act^\omega$ as described above. The relation $\equiv_I$ can be lifted to paths in the following way: $\pi \equiv_I \pi'$ iff $\pi \downarrow_2 \equiv_I \pi \downarrow_2$
The sequence of actions corresponding to a path π is denoted by $\pi \downarrow_2$.

5.1.4 Symmetry

Next, we define the concept of a symmetry group G of a LTS.

Definition 5.1.4 Given a LTS $T = (S, R, L, Act, s_0)$, a group G acting on S is called a *symmetry group* of T iff

- For all $\alpha \in Act$ and for all $\sigma \in G$, $s \xrightarrow{\alpha} s'$ iff $\sigma(s) \xrightarrow{\alpha} \sigma(s')$.
- For all $\sigma \in G$, $L(s) = L(\sigma(s))$.

Notice that if we are interested in checking a temporal formula f, the labeling function of the LTS can be restricted to the atomic propositions occurring in f. Therefore, all the restrictions on labelings given above only have to hold for the atomic propositions occurring in the temporal formula f of interest. We say that s and s' are in the same *orbit* iff there exists a $\sigma \in G$ such that $\sigma(s) = s'$. $\Theta \subseteq S \times S$ is the *orbit relation* induced by the symmetry group G. Given a LTS $T = (S, R, L, Act, s_0)$ and a symmetry group G acting on S, we define a *representative function* $\xi : S \rightarrow S$. The function ξ has two properties:

- s and $\xi(s)$ are in the same orbit.
- If s and s' are in the same orbit, then $\xi(s) = \xi(s')$.

The function ξ maps a state to an unique representative in its orbit. The lemma given below states that ξ is a bisimulation preserving abstraction function for T. This means that the entire framework automatically gives a method for combining partial-order and symmetry reductions.

Lemma 5.1.7 Assume that we are given a LTS $T = (S, R, L, Act, s_0)$ and a symmetry group G acting on S. Assume that ξ is a representative function corresponding to G. In this case, ξ is a bisimulation preserving abstraction.

Proof: Let $\Theta \subseteq S \times S$ be the following relation:

- $s \; \Theta \; s'$ iff s and s' are in the same orbit.

It is easy to prove that Θ is a bisimulation relation. By definition of ξ, ξ has the required properties with respect to the bisimulation relation Θ.$\Box$

In the definition of the symmetry group given at the beginning of this subsection we did not allow the actions to be permuted. This might seem overly restrictive. Now we will allow the symmetry group to permute states and actions simultaneously. Next, we will prove that this new seemingly more powerful notion of symmetry is equivalent to the definition of symmetry given before. Assume that we are given an LTS T and a symmetry group G according to the definition 5.1.5. We construct an LTS T_1 from T by relabeling actions such that G is a symmetry group for T_1 using definition 5.1.4. The group $\mathrm{Sym}(S) \times \mathrm{Sym}(Act)$ is the group of all permutations (π, σ) such that $\pi \in \mathrm{Sym}(S)$ and $\sigma \in \mathrm{Sym}(Act)$. Given a permutation $\psi = (\pi, \sigma) \in \mathrm{Sym}(S) \times \mathrm{Sym}(Act)$, for all $s \in S$ and $\alpha \in Act$ we define $\psi(s) = \pi(s)$ and $\psi(\alpha) = \sigma(\alpha)$.

Definition 5.1.5 Given a LTS $T = (S, R, L, Act, s_0)$, a group $G \leq \mathrm{Sym}(S) \times \mathrm{Sym}(Act)$ is called a *symmetry group* of T iff

- For $\pi \in G$, $s \xrightarrow{\alpha} s'$ iff $\pi(s) \xrightarrow{\pi(\alpha)} \pi(s')$.
- For all $\sigma \in G$, $L(s) = L(\sigma(s))$.

The *orbit* of an action $\alpha \in Act$ (denoted by $\theta_G(\alpha)$) is the following set:

$$\theta_G(\alpha) \quad = \quad \{\beta | \exists \pi \in G(\pi(\alpha) = \beta)\}$$

Let I be an independence relation on the LTS T. Let G be the symmetry group of T according to definition 5.1.5. Define $\Theta_G(I) \subseteq Act \times Act$ in the following manner:

- $(\alpha, \beta) \in \Theta_G(I)$ if and only if there exists α', β', and $\pi \in G$ such that $(\alpha', \beta') \in I$, $\alpha' = \pi(\alpha)$ and $\beta' = \pi(\beta)$

The lemma given below states that if I is an independence relation for T, then $\Theta_G(I)$ is also an independence relation for T.

Lemma 5.1.8 Let $T = (S, R, L, Act, s_0)$ be a LTS and $G \leq \mathrm{Sym}(S) \times \mathrm{Sym}(Act)$ be a symmetry group of T. If I is an independence relation on T, then $\Theta_G(I)$ is an independence relation on T.

Proof: Assume that $s \in S$, $\{\alpha, \beta\} \subseteq en_T(s)$ and $(\alpha, \beta) \in \Theta_G(I)$. By definition there exists a $\psi \in G$ such that $(\psi(\alpha), \psi(\beta)) \in I$. Using the fact that G is a symmetry group we have that $\{\psi(\alpha), \psi(\beta)\} \subseteq en_T(\psi(s))$. Moreover, we also have that there exists a path $\psi(s) \xrightarrow{\psi(\alpha)} s_1 \xrightarrow{\psi(\beta)} s'$ iff there exists a path $s \xrightarrow{\alpha} \psi^{-1}(s_1) \xrightarrow{\beta} \psi^{-1}(s')$. Similarly, there exists a path $\psi(s) \xrightarrow{\psi(\beta)} r_1 \xrightarrow{\psi(\alpha)} s'$ iff there exists a path $s \xrightarrow{\beta} \psi^{-1}(r_1) \xrightarrow{\alpha} \psi^{-1}(s')$. The result follows from these observations and definition of independence. $\Box$

Now we can assume that we are working with $\Theta_G(I)$ instead of I. Notice that in general, $\Theta_G(I)$ can be much larger than I. The lemma given below states that the property of an action being invisible is an invariant for an orbit.

Lemma 5.1.9 Let $\sigma \in G$ be an arbitrary permutation in G. An action $\alpha \in \mathtt{invis}_T$ iff $\sigma(\alpha) \in \mathtt{invis}_T$.

$\Theta(Act)$ denotes the set of orbits of the actions. Given an LTS $T = (S, R, L, Act, s_0)$, a symmetry group $G \leq \mathrm{Sym}(S) \times \mathrm{Sym}(Act)$ (according to the definition 5.1.5) and an independence relation $I \subseteq Act \times Act$, we construct an LTS $T_1 = (S_1, R_1, L_1, \Theta(Act), s_0)$, a symmetry group $G_1 \leq \mathrm{Sym}(S)$, and and independence relation $I_1 \subseteq \Theta(Act) \times \Theta(Act)$ in the following manner:

- $S_1 = S$.
- $L_1(s) = L(s)$.
- $s \stackrel{\theta_G(\alpha)}{\rightarrow} s' \in R_1$ iff $s \stackrel{\alpha}{\rightarrow} s' \in R$.
- $\pi \in G_1$ iff there exists σ such that $(\pi, \sigma) \in G$.
- $\theta_G(\alpha)\ I_1\ \theta_G(\beta)$ iff for all $\alpha' \in \theta_G(\alpha)$ and for all $\beta' \in \theta_G(\beta)$ we have that $\alpha'\ I\ \beta'$.

The lemma given below states that definition 5.1.4 can be used without loss of generality.

Lemma 5.1.10 Let G be a symmetry group of an LTS T using definition 5.1.5. Let T_1 and G_1 be constructed as before. In this case G_1 is the symmetry group of T_1 according to the definition 5.1.4. Moreover, I_1 is an independence relation for T_1.

Proof: Immediate from the construction of T_1. $\Box$

5.2 Algorithm for preserving LTL-X

Let $T = (S, R, L, Act, s_0)$ be an LTS and h be a bisimulation preserving abstraction function. In this section we will provide an algorithm which performs partial-order reduction and the reduction corresponding to the abstraction function h simultaneously. Basically, we give an algorithm which performs the partial-order reduction on the abstract LTS $T_h = (S_h, R_h, L_h, Act, h(s_0))$, but does not require the explicit construction of T_h. First we present an algorithm which uses the structure T_h. This algorithm is only given for the sake of the proof and was first presented in [60].

1 push($h(s_0)$)
2 expand-node($h(s_0)$)

3 **function** expand-node(s)
4 working-set(s) = `ample`(s)
5 **while** working-set(s) $\neq \phi$ **do**
6 α = some action in `ample`(s)
7 working-set = working-set(s)$\backslash\{\alpha\}$
8 **for all** $s' \in \alpha_{T_h}(s)$ **do**
9 **if** (new(s')) **then**
10 push(s')
11 expand-node(s')
12 create-edge(s,α,s')
13 **fi**
14 end **for all**
15 **end while**
16 mark s as explored.
17 end expand-node

Figure 5.2: State space expansion algorithm **(A1)**

The various routines used by the algorithm are described below:

- The routine new(s) checks that the state s has not been marked explored.
- The routine push(s) pushes the state s onto the search stack. We also assume that when a state s is marked explored (line 16), it is removed from the search stack.
- There is a LTS T' maintained by the algorithm. The routine

create(s,α,s') creates an transition between s and s' labelled by α.

$\mathcal{R}$ is called a *run* of the algorithm **A1** if and only if $\mathcal{R}$ is an execution of the algorithm **A1** where the sets `ample`(s) are chosen according to the rules **C1-h**, **C2-h** and **C3-h**. Notice that if we choose `ample`(s) = $en(s)$, the rules **C1-h**, **C2-h** and **C3-h** are satisfied, but this is same as reachability analysis with not partial-order reduction.

- **(C1-h)** For no action $\alpha \in Act \backslash$`ample`($s$) that is dependent on some action in `ample`(s) there exists a path π in T_h such that α appears in π before an action from `ample`(s) appears on π.

- **(C2-h)** If `ample`(s) is a proper subset of the actions enabled from s in T_h, then for no action $\alpha \in$ `ample`(s) it holds that a state in the set $\alpha_{T_h}(s)$ is on the search stack.

- **(C3-h)** If `ample`(s) is a *proper* subset of $en_{T_h}(s)$, then none of the actions in `ample`(s) are visible in T_h.

The following theorem states that any run of the algorithm **A1** produces a structure which is stuttering path equivalent to T_h.

Theorem 5.2.1 Let T' be the LTS produced by an arbitrary run of the algorithm **A1**. In this case $T_h \cong_{SPE} T'$.

Proof: See [60]. A proof of this theorem with some additional conditions appears at the end of this section. □

Now we modify algorithm **A1** to produce algorithm **A2**. Algorithm **A2** works on the LTS T, but because of some modifications it looks like that it is performing the partial order reduction on the LTS T_h. Algorithm **A2** is constructed from **A1** by changing lines 8, 9, 10, 11, and 12. We reproduce the whole algorithm for convenience, but mark the changed lines with a (**). $\mathcal{R}$ is called a *run* of the algorithm **A2** if and only if $\mathcal{R}$ is an execution of the algorithm **A2** where the sets `ample`(s) are chosen according to the rules **C1**, **C2**, and **C3**.

- **(C1)** For no action $\alpha \in Act \backslash$`ample`($s$) that is dependent on some action in `ample`(s) there exists a path π in T such that α appears in π before an action from `ample`(s) appears on π.

push($h(s_0)$)
expand-node($h(s_0)$)

function expand-node(s)
working-set(s) = `ample`(s)
while working-set(s) $\neq \phi$ **do**
α = some action in `ample`(s)
working-set = working-set(s)$\setminus\{\alpha\}$
for all $s' \in \alpha_T(s)$ **do** (**)
if (new($h(s')$) **then** (**)
push($h(s')$) (**)
expand-node($h(s')$) (**)
create-edge(s,α,$h(s')$) (**)
fi
end **for all**
end while
mark s as explored.
end expand-node

Figure 5.3: State space expansion algorithm **(A2)**

- **(C2)** If `ample`(s) is a proper subset of the actions enabled from s in T, then for no action $\alpha \in$ `ample`(s) it holds that a state in the set $h(\alpha_T(s))$ is on the search stack.

- **(C3)** If `ample`(s) is a *proper* subset of $en_T(s)$, then none of the actions in `ample`(s) are visible in T.

The lemma given below will be used in our main theorem.

Lemma 5.2.1 Let $T = (S, R, L, Act, s_0)$ be an LTS and h a bisimulation preserving abstraction function. Let $T_h = (S_h, R_h, L_h, Act, h(s_0))$ be the corresponding abstract LTS. Then, we have the following conditions:

- Let $s \in S_h$. There exists a path $s = s_0 \stackrel{\alpha_0}{\rightarrow} s_1 \stackrel{\alpha_1}{\rightarrow} \cdots$ in T if and only if there exists a path $s = t_0 \stackrel{\alpha_0}{\rightarrow} t_1 \stackrel{\alpha_1}{\rightarrow} \cdots$ in T_h. Notice that exactly the same actions appear in the two paths.

- An action α is visible in T if and only if it is visible in T_h.

- For all $s \in S_h$, $en_T(s) = en_{T_h}(s)$.

Proof: The results are a direct consequence of the fact that T and T_h are bisimilar (see lemma 5.1.5).□

Next, we prove that given a run of the algorithm **A2** there exists a run of the algorithm **A1** such that both runs produce the same LTS. We must emphasize again that algorithm **A1** only exists for the sake of the proof. In practice, **A2** will be implemented. The basic idea of the theorem is to run **A1** and **A2** in lockstep and show that the ample sets which satisfy conditions **C1**, **C2**, and **C3** for algorithm **A2** also satisfy conditions **C1-h**, **C2-h**, and **C3-h** for algorithm **A1** at each step.

Theorem 5.2.2 For every run $\mathcal{R}$ of the algorithm **A2** there exists a run $\mathcal{R}'$ of the algorithm **A1** such that the LTS produced by the two runs are the same.

Proof: We will construct a run $\mathcal{R}'$ of the algorithm **A1** as we trace the execution corresponding to the run $\mathcal{R}$ of the algorithm **A2**. At each point we will prove that the following invariants hold:

- If the run $\mathcal{R}$ chooses a set `ample`(s) in line 4 which satisfies conditions **C1**, **C2**, and **C3**, then `ample`(s) satisfies, **C1-h**, **C2-h** and **C3-h** for the run $\mathcal{R}'$.

- The state of the two runs are the same, i.e., the stacks have the same states and the same states are marked explored.

Initially, the invariants hold because both the runs will push $h(s_0)$ on the stack. Suppose that at some point in the execution the run $\mathcal{R}$ of the algorithm **A2** chooses a set `ample`(s) on line 4 which satisfies conditions **C1**, **C2**, and **C3**. Due to lemma 5.2.1 `ample`(s) also satisfies conditions **C1-h** and **C3-h**. Consider a state $r \in \alpha_{T_h}(s)$ where $\alpha \in$ `ample`(s). Notice that by definition there exists a state $s' \in \alpha_T(s)$ such that

$r = h(s')$. Now it is obvious from condition **C2** that r cannot be on the search stack because $r \in h(\alpha_T(s))$. So $\mathtt{ample}(s)$ considered by the run $\mathcal{R}$ satisfies condition **C2-h** for the run $\mathcal{R}'$. Also notice that lemma 5.2.1 implies that

$$(\mathtt{ample}(s) \neq en_T(s)) \quad \Leftrightarrow \quad (\mathtt{ample}(s) \neq en_{T_h}(s))$$

Now we advance the two runs, and assume that they consider the states in the same order in the **for all** loop starting at line 8. $\Box$

The theorem given below states that any run of the algorithm **A2** produces a LTS which stuttering path equivalent to T.

Theorem 5.2.3 Let $\mathcal{R}$ be an arbitrary run of the algorithm **A2** Let T' be the LTS produced by the run $\mathcal{R}$. Then we have that $T \cong_{SPE} T'$.

Proof: Let $\mathcal{R}'$ be the run of the algorithm **A1** which produces the same LTS as the run $\mathcal{R}$. Run $\mathcal{R}'$ exists because of theorem 5.2.2. By theorem 5.2.1 $T' \cong_{SPE} T_h$. By lemmas 5.1.3 and 5.1.4 $T \cong_{SPE} T'$.$\Box$.

Notice that dut to theorem 5.1.2, T and T' satisfy the same LTL-X formulas. Therefore, one can check a specification given in LTL-X on the smaller LTS.

Now we proceed to give the proof of Theorem 5.2.1. To make the proof simple we replace conditions **C3-h** by the following conditions:

- If $(\alpha, \beta) \in I$, then either $\alpha \in \mathtt{invis}_{T_h}$ or $\beta \in \mathtt{invis}_{T_h}$.
- **(F)** If an action α is enabled from some state of a path in T_h, then some action that is dependent on α (possibly α itself) must appear later (or immediately) in this path.

The results given here are true without these modifications. The proofs in this section are based on [60]. The proofs of the results without the additional constraints are also given in [60]. First, we have the following lemma.

Lemma 5.2.2 Let π and π' be two paths in T_h such that $\pi \equiv_I \pi'$. In this case π and π' are stuttering equivalent.

Proof: Throughout the proof we assume that $\mathtt{vis} = \mathtt{vis}_{T_h}$. Given a path π, let $\pi \downarrow_{\mathtt{vis}}$ be the projection of the path down to the visible actions. First, notice that if we have two paths π_1 and π_2 such that $\pi_1 \equiv_I \pi_2$, then $\pi_1 \downarrow_{\mathtt{vis}}$ and $\pi_2 \downarrow_{\mathtt{vis}}$ are the same. This is because when we permute two independent actions α and β, one of them has to be invisible. Therefore, permutation of independent actions does not change the sequence of visible actions.

We will prove that π and π' are stuttering equivalent. By the observation given at the beginning of the proof, we have that $\pi \downarrow_{\mathtt{vis}}$ and $\pi' \downarrow_{\mathtt{vis}}$ are equal. There are two cases.

- $\pi \downarrow_{\mathtt{vis}}$ is infinite.
  For concreteness, let $\pi \downarrow_{\mathtt{vis}} = \gamma_1, \gamma_2, \cdots$. Let i_j be such that $\pi[i_j] \stackrel{\gamma_j}{\to} \pi[i_j+1]$. Similarly, let k_j be such that $\pi'[k_j] \stackrel{\gamma_j}{\to} \pi'[k_j+1]$. Using the sequences $i_0 = 0 < i_1 < i_2 < \cdots$ and $k_0 = 0 < k_1 < k_2 < \cdots$ and the observation that invisible actions do not change labels of states, we have that π and π' are stuttering equivalent.

- $\pi \downarrow_{\mathtt{vis}}$ is finite.
  In this case we use the same construction as in the infinite case, but to make the sequence of integers $\{i_j\}$ and $\{k_j\}$ infinite we use the integers corresponding to the invisible actions after the last visible action on the paths π and π'. Formally, let $\pi \downarrow_{\mathtt{vis}} = \gamma_1, \gamma_2, \cdots, \gamma_l$. Construct two finite sequences $i_0 = 0 < i_1 < \cdots < i_l$ and $k_0 = 0 < k_1 < \cdots < k_l$ as in the previous case. The two sequences can be made infinite by setting $i_j = i_l + (j - l)$ and $k_j = k_l + (j - l)$ for $j > l$. $\square$

Throughout the proof we will be using the fact that we are working with the LTS T_h. Given a finite path $\pi = s_0 \stackrel{\alpha_0}{\to} s_1 \stackrel{\alpha_1}{\to} \cdots \stackrel{\alpha_{n-1}}{\to} s_n$ and path $\pi' = t_0 \stackrel{\beta_0}{\to} \cdots$, $\pi \stackrel{\beta}{\to} \pi'$ denotes the following path:

$$s_0 \stackrel{\alpha_0}{\to} s_1 \stackrel{\alpha_1}{\to} \cdots \stackrel{\alpha_{n-1}}{\to} s_n \stackrel{\beta}{\to} t_0 \stackrel{\beta_0}{\to} \cdots$$

Lemma 5.2.3 Let $\mathtt{ample}(s)$ be a set satisfying conditions **C1-h**. Let π be **F**-fair path. Let $\pi = \pi_1 \stackrel{\beta}{\to} \pi_2$ be such that the last state on the finite path π_1 is s. There exists an action $\alpha \in \mathtt{ample}(s)$ such that $[\alpha] \preceq_I [\pi_2 \downarrow_2]$.

Proof: To clean up the notation let $w = \pi_2 \downarrow_2$. Consider the string of actions βw. Let v be the maximal prefix of βw such that $v \cap \mathtt{ample}(s) = \phi$. Using the fairness constraint **F** and the condition **C1-h**, we have that v is finite. Let $v\alpha w' = \beta w$. By definition, $\alpha \in \mathtt{ample}(s)$ and the actions in v are independent of α (otherwise we have a violation of condition **C1-h**). Therefore, $\alpha v w' \equiv_I \beta w$. Hence, $[\alpha] \preceq_I [\beta w]$. □

The lemma given below states that if a node s is marked explored, then every transition from it can be eventually taken. In other words, no transition reachable from the initial state $h(s_0)$ is delayed forever, or eliminated from the reduced structure produced by the algorithm **A1**.

Lemma 5.2.4 Let s be an explored state. Let π be **F**-fair path in an LTS T_h. Let $\pi = \pi_1 \xrightarrow{\alpha} \pi_2$ such that the last state on the finite path π_1 is a state s in the reduced LTS T' produced by the algorithm **A1**. Then there exists a path $s = s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t$ in T_h such that $\beta_1, \cdots, \beta_n$ are independent of α and $[\beta_1 \cdots \beta_n \alpha] = [\alpha \beta_1 \cdots \beta_n] \preceq_I [\alpha(\pi_2 \downarrow_2)]$

Proof: The proof is by induction on the time when the state is marked explored. There are two cases to be considered.

- **(Case** $\alpha \in \mathtt{ample}(s)$**)**
  In this case the statement trivially holds with $n = 0$.

- **(Case** $\alpha \notin \mathtt{ample}(s)$**)**
  Using Lemma 5.2.3 there exists a $\gamma \in \mathtt{ample}(s)$ such that γ appears on the path π_2. Let s' be a state such that $s \xrightarrow{\gamma} s'$. Using condition **C2-h** we have that s' is an explored state. Since the algorithm **A1** explores the nodes in the depth first search order, s' has to be marked explored before s. By permuting γ to the front of α, (by the proof of Lemma 5.2.3 γ and α are independent) we get a new path $\pi' = \pi \xrightarrow{\gamma} s' \xrightarrow{\alpha} \pi_2'$. Since s' is marked explored before the state s, we can apply the induction hypothesis to s'. There exists a $\beta_1', \cdots, \beta_m'$ such that $s' = s_0 \xrightarrow{\beta_1'} s_1 \xrightarrow{\beta_2'} \cdots \xrightarrow{\beta_m'} s_m \xrightarrow{\alpha} t$. Concatenating γ at the beginning of $\beta_1', \cdots, \beta_m'$ we get the required result.

The lemma is at the core of the partial-order reduction method. It says that no action is delayed infinitely, and can be eventually taken. $\square$

Notice that T' constructed by the algorithm **A1** is a sub-structure of the abstract LTS T_h. Therefore, every path of T' is also a path of T_h. Hence, the theorem given below proves that $T_h \cong_{SPE} T'$ (which is Theorem 5.2.1).

Theorem 5.2.4 Let π be a path in the abstract LTS T_h. Then there exists a path π' in the LTS T' constructed by the algorithm **A1** such that $\pi \equiv_I \pi'$.

Proof: Let π be a path in the abstract LTS T_h. We will construct a path π' in T' starting from $\pi[0]$. The path is constructed in an inductive manner. We will maintain the following data structures:

r: The sequence of actions in π which have been processed.
t: The partial path (a part of π') constructed in T' so far.
l: The sequence of actions from t which have not been processed.
s: The current state in π'.

Initially, let $r = l = \epsilon$ and t be the empty path. Also at the initial stage $s = \pi[0]$. Whenever a transition $s_i \xrightarrow{\alpha} s_{i+1}$ is processed from the path π, following updates are made:

1. $r = r\alpha$.

2. If $l = u\alpha w$ for some $u, w \in Act^\star$ and all actions in u are independent of α, then let $l = uw$.

3. If the condition for executing step 2 does not hold, choose a sequence of actions starting from the state s and ending in the state s' such $[t \downarrow_2 \beta_1\beta_2\cdots\beta_n\alpha] = [t \downarrow_2 \alpha\beta_1\cdots\beta_2] \preceq_I [\pi \downarrow_2]$ and make the following updates:

   - $s = s'$.
   - $l = l\beta_1\cdots\beta_n$.
   - $t = t \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} t_2 \cdots \xrightarrow{\beta_n} t_n \xrightarrow{\alpha} s'$.

The following invariants are maintained by the procedure given above.

1. $[r][l] = [t \downarrow_2]$.

2. $[t \downarrow_2] \preceq_I [\pi \downarrow_2]$.

3. If step 2 cannot be executed, then all actions in l are independent of α.

4. The choice of the sequence of actions $\beta_1\beta_2 \cdots \beta_n\alpha$ can always be made in step 3.

We will now show that the invariants hold throughout the procedure. It is trivial to check that the invariants hold initially. By the induction hypothesis we have the following equation

$$[r][l] \ = \ [t] \ \preceq_I [\pi \downarrow_2] \ \ = \ [r][\alpha w]$$

Therefore, l can be written as $u\alpha z$, where actions in u are independent of α. If step 2 cannot be executed, then l does not contain α, so actions in l are independent of α. Invariant 4 is direct consequence of Lemma 5.2.4. Let π' be the infinite path collected in the variable t. Notice that π' can be regarded as an infinite run on the input π of the ω-automata described by the three update rules. By the first invariant every prefix r of $\pi \downarrow_2$ is $\preceq_I \pi' \downarrow_2$. Hence, $\pi \downarrow_2 \preceq_I \pi' \downarrow_2$. By the second invariant every prefix t of $\pi' \downarrow_2$ is $\preceq_I \pi \downarrow_2$. Therefore, $\pi' \downarrow_2 \preceq_I \pi \downarrow_2$. Hence by definition $\pi \equiv_I \pi'$. $\Box$

5.3 Algorithm Preserving $CTL^\star$-X

The algorithm given in the previous section only preserved the existence of equivalent paths. The semantics of branching time logics (like $CTL^\star$) are based on computation trees. Therefore, these logics can distinguish the branching structure of a state. Hence, to preserve branching time logics one has to put more stringent restrictions on the set $\texttt{ample}(s)$ considered by the algorithms. We call $\mathcal{R}$ a *run* of the of the algorithm **A1** if the $\texttt{ample}(s)$ satisfies the following condition in addition to conditions **C1-h**, **C2-h**, and **C3-h** given earlier.

- **(C4-h)** The set $\texttt{ample}(s)$ is a singleton set or $\texttt{ample}(s) = en_{T_h}(s)$.

In a similar manner, We call $\mathcal{R}$ a *run* of the of the algorithm **A2** if the `ample`(s) satisfies the following condition in addition to conditions **C1**, **C2**, and **C3**.

- **(C4)** The set `ample`(s) is a singleton set or `ample`$(s) = en_T(s)$.

The treatment is exactly the same as in section 5.2. Therefore, we will skip all the proofs. The proofs will use lemma 5.2.1 to establish that condition **C4** implies condition **C4-h**. In order for the proofs to work, we will have strengthen the notion of independence. The new definition of independence is given below:

Definition 5.3.1 Let $T = (S, R, L, Act, s_0)$ be an LTS. An *independence relation* on actions is an irreflexive and symmetric relation $I \subseteq Act \times Act$ such that for each pair of actions $(\alpha, \beta) \in I$ (called *independent actions*) it must hold that for each $s \in S$

- If $\{\alpha, \beta\} \subseteq en_T(s)$, then for each state $s' \in \alpha_T(s)$ we have that $\beta \in en_T(s')$.
- If $\{\alpha, \beta\} \subseteq en_T(s)$, the for all $s' \in \alpha(s)$ and for all $s'' \in \beta(s)$ we should have that $s' \xrightarrow{\beta} s_1$ if and only if $s'' \xrightarrow{\alpha} s_1$.

An LTS $T = (S, R, L, Act, s_0)$ is called *deterministic* if and only if for all $s \in S$ and $\alpha \in Act$ we have that $|\alpha(s)| \leq 1$. It is straight forward to prove that if T is deterministic then the new definition of independence and the old one are equivalent.

Theorem 5.3.1 Let T' be the LTS produced by an arbitrary run of the algorithm **A1**. In this case $T_h \cong_{SB} T'$.

Proof: The proof of the theorem appears in [35] and [61]. For sake of completeness, we give the proof at the end of the section. By Lemma 5.3.4 we have that $T_h \cong_{StB} T'$. Now by lemma 5.1.3 we have that $T_h \cong_{SB} T'$. $\Box$.

The proof of theorem given below is exactly the same as the proof of the theorem 5.2.2.

Theorem 5.3.2 For every run $\mathcal{R}$ of the algorithm **A2** there exists a run $\mathcal{R}'$ of the algorithm **A1** such that the LTS produced by the two runs are the same.

The theorem given below states that any run of the algorithm **A2** produces a LTS which is stuttering bismular to T.

Theorem 5.3.3 Let $\mathcal{R}$ be an arbitrary run of the algorithm **A2**. Let T' be the LTS produced by the run $\mathcal{R}$. Then we have that $T \cong_{SB} T'$.

Proof: Let $\mathcal{R}'$ be the run of the algorithm **A1** which produces the same LTS as the run $\mathcal{R}$. Run $\mathcal{R}'$ exists because of theorem 5.3.2. By theorem 5.3.1 $T' \cong_{SB} T_h$. By lemmas 5.1.3 and 5.1.4 $T \cong_{SB} T'$.□

Notice that because of theorem 5.1.1 T and T' satisfy the same CTL-X formulas. Therefore, one can check a specification given in CTL-X on the smaller LTS T' instead of T.

Now we proceed to show that there exists a stammerring bisimulation between T_h and T'. The proof here is based on the proofs which appear in [35, 61]. Let $T_h = (S_h, R_h, L_h, Act, h(s_0))$ be the abstract LTS. Let $T' = (S', R', L', Act, s_0')$ be the reduced LTS produced by algorithm **A1**. Notice that the the `ample` sets should satisfy condition **C1-h**, **C2**, **C3-h**, and **C4-h**. First, we define a relation $\sim \subseteq S_h \times S_h$. An action α is called a *singleton* action from a state s if and only if $\alpha \in \mathtt{invis}_{T_h}$ and α satisfies the condition **C1-h** from the state s. A path $t_0 \xrightarrow{\beta_0} t_1 \xrightarrow{\beta_1} \cdots t_n$ is called a *singleton* path if and only if β_i is a singleton action from the state t_i. We say that $s \sim s'$ if and only if there is singleton path from s to s'. The length of the shortest path between s and s' is called the *distance* between s and s'. The lemma given below states that if α is singleton action from a state s, then α is a singleton action from all successors of s' such that $s \xrightarrow{\beta} s'$ and $\alpha \neq \beta$. We assume throughout the proof that we are working with the LTS T_h.

Lemma 5.3.1 Let $s \xrightarrow{\alpha} t$ be such that α is a singleton action from s. Let $s \xrightarrow{\beta} r$ such that $\alpha \neq \beta$. In this case α is a singleton action from r.

Proof: Condition **C1-h** implies that β is independent of α. This means that α is enabled from r. By definition α is invisible. Suppose α is not a singleton action from r. Then there exists a path π starting from r such that an action dependent on α appears in π but α does not appear in π. In this case, path $s \xrightarrow{\beta} \pi$ violates the condition **C1-h** from s. A contradiction.□

The lemma given below is a characterization of singleton paths which will be used in the main lemma. The second part of the lemma is particularly useful in performing induction over the length.

Lemma 5.3.2 Let $s = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} s_n = s'$ be a singleton path. If $s \xrightarrow{\beta} t$, then there are two possibilities.

1. If β is independent of α_i (for $0 \leq i \leq n-1$), then $t = t_0 \xrightarrow{\alpha_0} t_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} t_n$ is a singleton path such that $s_i \xrightarrow{\beta} t_i$ for all i.

2. Supoose β is independent of α_i $(0 \leq i < j \leq n-1)$ and β is dependent on α_j. In this case, there exists a singleton path of length $n-1$ from t to s'.

Proof: The proof of the lemma follows from the following observations (see Figure 5.4 and 5.5).

- If α_i and β are independent, then α_i is enabled from t_i (recall that $s_i \xrightarrow{\beta} t_i$). Moreover, because of lemma 5.3.1 α_i is a singleton action from t_i.

- Now we consider the second case. Notice that β and α_j are dependent and both actions are enabled from s_j. Since α_j is a singleton action from s_j, $\beta = \alpha_j$. Otherwise, we have a violation of condition **C1-h**. We have that $t_{j-1} \in \beta(s_{j-1})$ and $s_j \in \alpha_{j-1}(s_{j-1})$. Using the fact that β and α_{j-1} are independent under the new definition, we can choose $t_j = s_{j+1}$.

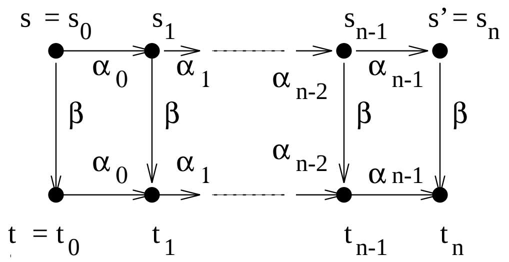

Figure 5.4: β is indpendent of α_i

The corollary given below will be crucial in establishing a stammering bisimiulation between T_h and T'.

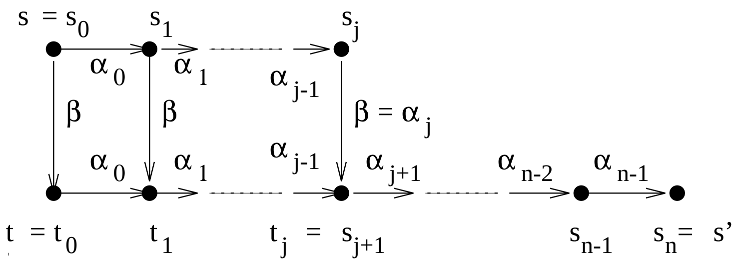

Figure 5.5: β is dependent on α_j

Corollary 5.3.1 Let $s \sim s'$ and $s \xrightarrow{\beta} t$. In each of the following cases, there exists and edge $s' \xrightarrow{\beta} t'$ such that $t \sim t'$.

1. β does not appear on some singleton path from s to s' (in particular $\beta \in \mathtt{vis}_{T_h}$), or
2. $t \not\sim s'$.

Proof: Let $s = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} s_n = s'$ be a singleton path from s to s' such that β does not appear on this path. If β is dependent on α_j, then by the proof of lemma 5.3.2 β appears on the singleton path from s to s' (recall that $\alpha_j = \beta$). Therefore, β is independent of α_i for all i. Now by the first part of lemma 5.3.2 there exists a singleton path from t to t' such that $s' \xrightarrow{\beta} t'$. By definition, $t \sim t'$.

Now we move on to the second case. Assume that $t \not\sim s'$. Let $s = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} s_n = s'$ be a singleton path from s to s'. In this case β is also independent of α_i (for all i). Notice that if β was dependent on α_j, then because of lemma 5.3.2 there would exist a singleton path between t and s'. But, that would imply $t \sim s'$, a contradiction. Now by lemma 5.3.2 there exists a singleton path between t and t' such that $s' \xrightarrow{\beta} t'$.□

In the reduced LTS T' produced by algorithm **A1** a state s whose $\mathtt{ample}(s)$ is a singleton set is called *partial*. Otherwise, the state s is called *full*. The lemma given below states that in a reduced structure there always exists a singleton path from a partial state s to a full state s', or given a partial state s one can always find a full state s' such that $s \sim s'$.

Lemma 5.3.3 Let T' be the reduced LTS produced by the algorithm **A1**. Given a partial state s, there exists a singleton path from s to a full state s', i.e., $s \sim s'$.

Proof: The proof is by induction on the order in which the states are marked explored by algorithm **A1**. Let s be a partial node. Let $s \xrightarrow{\alpha} s_1$ be a transition in T'. If s_1 is a full state, we are done. Assume that s_1 is a partial state. Since algorithm **A1** explores the states in the depth first order, s_1 has to be marked explored before s (otherwise we have a violation of condition **C2-h**). By the induction hypothesis there exists a singleton path π from s_1 to a full node s'. By definition, $s \xrightarrow{\beta} \pi$ is a singleton path from s to s'.$\square$

Let $\approx \subseteq S_G \times S'$ be the relation $\sim$ with the right hand side is restricted to S', i.e., $\approx = \sim \cap (S_G \times S')$.

Lemma 5.3.4 The relation $\approx$ is a stammering bisimulation between the abstract LTS $T_h = (S_h, R_h, L_h, Act, h(s_0))$ and the LTS $T' = (S', R', L', Act, s'_0)$ produced by the algorithm **A1**.

Proof: Observe that since $\sim$ is reflexive, $h(s_0) \approx h(s_0)$. A singleton path can only have invisible actions, so $s \approx s'$ implies that $L(s) = L(s')$. This proves the first condition of the stammering bisimulation (see definition 5.1.2).

Now we move on to proving the second condition of the stammering bisimulation. Assume that $s \xrightarrow{\beta} t \in R_h$. There are two case.

- **Case 1:** $t \sim s'$ and β is invisible.
  In this case the condition is true by definition.

- **Case 2:** $t \not\sim s'$ or β is visible.
  By corollary 5.3.1 there exists t' such that $s' \xrightarrow{\beta} t'$ and $t \sim t'$. Since we cannot guarantee that $t' \in S'$, we cannot assert that $t \approx t'$. By lemma 5.3.3 there exists a full node s'' such that $s' \approx s''$. By transitivity, $s \approx s''$. There are two subcases here.

  - **Case 2.1:** $t' \sim s''$ and β is invisible.
    In this case, $t \sim t' \sim s''$. Therefore, by transitivity $t \approx s''$.

- **Case 2.2:** $t \not\sim s''$ or β is visible.

  By corollary 5.3.1 there exists a t'' such that $s'' \xrightarrow{\beta} t''$ with $t' \sim t''$. Since s'' is a full node, $t'' \in S'$. Therefore, $t \approx t''$.

Now we handle the other direction. Suppose, $s' \xrightarrow{\beta} t'$ is a transition in T'. Since $s \sim s'$, there exists a singleton path π, $s = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} s_n = s'$. In this case the path $\pi \xrightarrow{\beta} t'$ satisfies condition 2 of stammering bisimulation (see definition 5.1.2).

Finally, we prove that $\approx$ satisfies the third condition corresponding to stammering bisimulation. Let $s = s_0 \xrightarrow{\beta_0} t_1 \xrightarrow{\beta_1} \cdots$ be an infinite path in T_h such that for all i, $\beta_i \in \mathtt{invis}_{T_h}$ and $t_i \approx s'$. By Lemma 5.3.3 there exists a full state s'' in T' such that $s' \approx s''$. By transitivity, $t_i \approx s''$ for all i.

First, we will show that there exists a β_j such that β_j does not appear on a singleton path from t_j to s''. We will prove the result by contradiction. Let π_0 be a finite singleton path between $s = t_0$ and s''. By assumption β_0 appears in π_0. Path π_0 exists because $s \approx s''$. By Lemma 5.3.2 there exist a singleton path π_1 from t_1 to s'' whose length is 1 less than that of π_0. Continuing this way, we can construct an infinite sequence of finite paths $\pi_0, \pi_1, \cdots$ such that the length of π_{i+1} is one less than that of π_i. This is not possible because π_0 is of finite length. Therefore, assume that β_j does not appear on a singleton path from t_j to s''. By corollary 5.3.1 there exists t'' such that $s'' \xrightarrow{\beta} t''$ and $t_{j+1} \sim t''$. Since s'' is a full state, $t'' \in S'$. Therefore, $t_{j+1} \approx t''$.

Now assume that $s' = r_0 \xrightarrow{\alpha_0} r_1 \xrightarrow{\alpha_1} \cdots$ is an infinite path such that for all i, $\alpha_i \in \mathtt{invis}_{T_h}$ and $s \approx r_i$. By definition, there exists a singleton path π from s to s'. Path $\pi \xrightarrow{\alpha_0} r_1$ satisfies the required condition.$\Box$

5.4 Example

In this section we given an example to illustrate our ideas. Figure 5.6 shows a solution to the two process mutual exclusion problem. N_i denotes that process i is the *nuetral* section. T_i is the *trying* region for the process i. C_i signals that process i is in the *critical* section. Since we are only dealing with two processes, $i = 1$ or $i = 2$. Whenever, process 2 makes a transition from N_2 to T_2 it sets an auxiliary variable $t = 1$. This signals the fact that process 1 can move into its critical section. A symmetric transition appears in process 1. It is obvious that exchanging indices 1 and 2 is a symmetry for this system. Let G be the corresponding symmetry group. There are 8 possible actions corresponding to the transitions shown below:

χ_1	$N_1 \rightarrow T_1$	χ_2	$N_2 \rightarrow T_2$
α_1	$T_1 \rightarrow C_1$	α_2	$T_2 \rightarrow C_2$
β_1	$N_1 \rightarrow T_1$	β_2	$N_2 \rightarrow T_2$
δ_1	$C_1 \rightarrow N_1$	δ_2	$C_2 \rightarrow N_2$

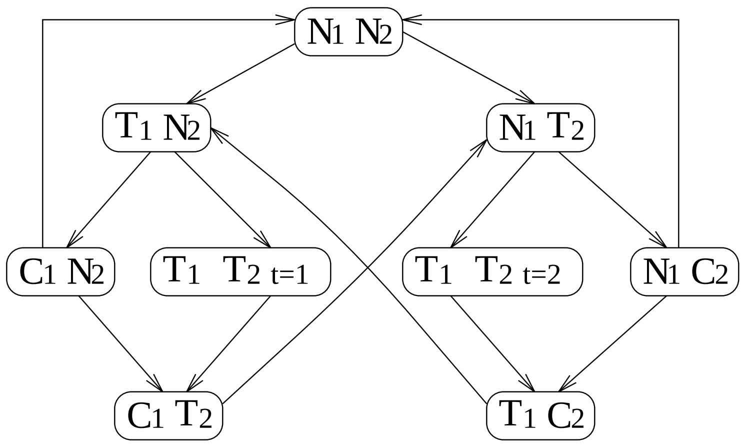

Figure 5.6: Token Ring

Following the discussion in subsection 5.1.4, actions with the same name but different indices are in the same orbit under the action of the group G. For example, α_1 and α_2 are in the same orbit. Renaming the actions and performing the symmetry reductions we get the abstract structure given in the Figure 5.7. From the figure it should be clear what the representative function is. Also, notice that action α and β are independent. Now performing partial-order reduction on the abstract structure we get the structure given in the Figure 5.8.

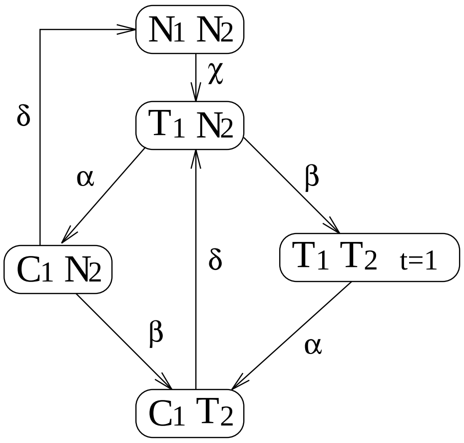

Figure 5.7: Quotient Structure

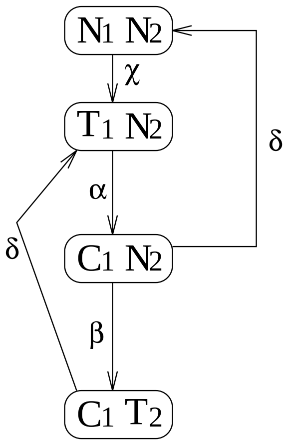

Figure 5.8: Quotient Structure with PO reduction

Chapter 6

Verifying Parameterized Networks

This chapter considers the problem of verification of family of state-transition systems. The verification problem for a family of similar state-transition systems can be formulated as follows:

> Given a family $F = \{P_i\}_{i=1}^{\infty}$ of systems P_i and a temporal formula f, verify that each state-transition system in the family F satisfies f.

In general the problem is undecidable [4]. However, for *specific* families the problem may be solvable.

The technique presented in this chapter is based on finding *network invariants* [47, 74]. Given an infinite family $F = \{P_i\}_{i=1}^{\infty}$, this technique involves constructing an invariant I such that $P_i \preceq I$ for all i. The preorder $\preceq$ preserves the property f we are interested in, i.e., if I satisfies f, then P_i satisfies f. Once the invariant I is found, traditional model checking techniques can be used to check that I satisfies f. Following [53] and [64] we restrict our attention to families of systems derived by *network grammars*. The advantage of such a grammar is that it is a finite (and usually small) representation of an infinite family of finite-state systems (referred to as the language of the grammar). While [53, 64] use the grammar in order to find a representative that is equivalent to any system derived by the grammar, the technique presented here finds a representative that is greater in the simulation preorder than all of the systems that can be derived using the grammar.

This chapter is organized as follows. Section 6.1 defines the basic notions, including network grammars and regular languages used as state properties. In Section 6.2 abstract systems are defined. Section 6.3 presents a verification method. Section 6.4 describes a synchronous model of computation and show that it is suitable for the verification technique presented here. Section 6.5 describes an asynchronous model of computation. In Section 6.6 the verification method is applied to two non-trivial examples.

6.1 Definitions and Framework

Definition 6.1.1 (LTS) A *Labeled Transition System* or an *LTS* is a structure $M = (S, R, ACT, S_0)$ where S is the set of states, $S_0 \subseteq S$ is the set of initial states, ACT is the set of actions, and $R \subseteq S \times ACT \times S$ is the *total* transition relation, such that for every $s \in S$ there is some action a and some state s' for which $(s, a, s') \in R$. We use $s \xrightarrow{a} s'$ to denote that $(s, a, s') \in R$.

A path π from a state s in an LTS M is a sequence of transitions $s = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \cdots$. The suffix of π starting from the i-th state is denoted by π^i. The i-th state on the path π is denoted by $\pi[i]$. Let L_{ACT} be the class of LTSs whose set of actions is a subset of ACT. Let $L_{(S,ACT)}$ be the class of LTSs whose state set is a subset of S and the action set is the subset of ACT. The definition of LTS given here is different from the one given in Chapter 5 in that the labeling function is ignored in the current definition. Moreover, the current definition of LTS allows more than one initial state.

Definition 6.1.2 (Composition) A function $\| : L_{ACT} \times L_{ACT} \mapsto L_{ACT}$ is called a *composition function* iff given two LTSs $M_1 = (S_1, R_1, ACT, S_0^1)$ and $M_2 = (S_2, R_2, ACT, S_0^2)$ in the class L_{ACT}, $M_1 \| M_2$ has the form $(S_1 \times S_2, R', ACT, S_0^1 \times S_0^2)$. Notice that we write the composition function in infix notation.

Our verification method handles a set of LTSs referred to as a *network*. Intuitively, a network consists of a set of LTSs obtained by composing any number of LTSs from $L_{(S,ACT)}$. Thus, each LTS in a network is defined over the set of actions ACT, and over a set of states in S^i, for some i.

Definition 6.1.3 (Network) Given a state set S and a set of actions ACT, any subset of $\bigcup_{i=1}^{\infty} L_{(S^i,ACT)}$ is called a *network* on the tuple (S, ACT).

6.1.1 Network grammars

Following [53] and [64] we use context-free network grammars as a formalism to describe networks. The set of all LTSs derived by a network grammar (as "words" in its language) constitutes a network. Let S be a state set and ACT be a set of actions. Then, $G = \langle T, N, \mathcal{P}, \mathcal{S} \rangle$ is a grammar where:

- T is a set of terminals, each of which is an LTS in $L_{(S,ACT)}$. These LTSs are sometimes referred to as *basic processes*.
- N is a set of non-terminals. Each non-terminal defines a network.
- $\mathcal{P}$ is a set of production rules of the form

$$A \rightarrow B \|_i C$$

  where $A \in N$, and $B, C \in T \cup N$, and $\|_i$ is a composition function. Notice that each rule may have a different composition function.
- $\mathcal{S} \in N$ is the start symbol that represents the network generated by the grammar.

Example 6.1.1 We clarify the definitions on a network consisting of LTSs that perform a simple mutual exclusion using a token ring algorithm. The production rules of a grammar that produces rings with one process Q and at least two processes P are given below. P and Q are terminals, and A and $\mathcal{S}$ are nonterminals where $\mathcal{S}$ is the start symbol.

$$\begin{aligned} \mathcal{S} &\rightarrow Q \| A \\ A &\rightarrow P \| A \end{aligned}$$

$$A \rightarrow P \| P$$

P and Q are LTSs defined over the set of states $\{nc, cs\}$ and the set of actions $ACT = \{\tau,$ `get-token`, `send-token`$\}$. They are identical, except for their initial state, which is cs for Q and nc for P. Their transition relation is shown in Figure 6.1.

For this example we assume a synchronous model of computation in which each process takes a step at any time. We will not give a formal definition of the model here. In Sections 6.4 and 6.5 we suggest suitable definitions for synchronous and asynchronous models. Informally, a process can always perform a τ action. However, it can perform a `get-token` action if and only if the process to its left is ready to perform a `send-token` action. In the composed LTS, the two actions `get-token` and `send-token` are replaced by τ.

We can apply the following derivation

$$\mathcal{S} \Rightarrow Q \| A \Rightarrow Q \| P \| P$$

to obtain the LTS $Q \| P \| P$. The reachable states with their transitions are shown in Figure 6.2.

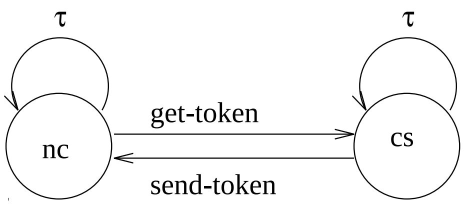

Figure 6.1: Process Q, if $S_0 = \{cs\}$; process P if $S_0 = \{nc\}$

6.1.2 Specification language

Let S be a set of states. From now on we assume that we have a network defined by a grammar G on the tuple (S, ACT). The automaton defined below has S as its alphabet. Thus, it accepts words which are sequences of state names.

Definition 6.1.4 (Specification) $\mathcal{D} = (Q, q_0, \delta, F)$ is a *deterministic automaton* over S, where

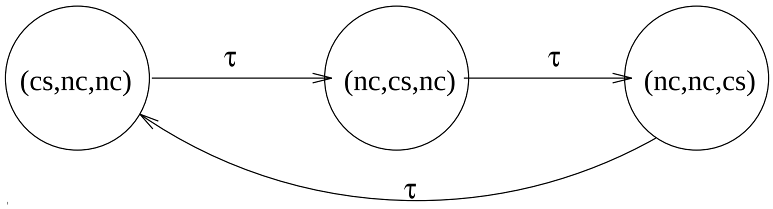

Figure 6.2: Reachable states in LTS $Q\|P\|P$

1. Q is the set of automaton states.
2. $q_0 \in Q$ is the initial state.
3. $\delta \subseteq Q \times S \times Q$ is the transition relation.
4. $F \subseteq Q$ is the set of accepting states.

$\mathcal{L}(\mathcal{D}) \subseteq S^\star$ is the set of words accepted by $\mathcal{D}$.

Our goal is to specify a network of LTSs composed of any number of components (i.e., of basic processes). We will use finite automata over S in order to specify atomic state properties. Since a state of an LTS is a tuple from S^i, for some i, we can view such a state as a word in $S^\star$. Let $\mathcal{D}$ be an automaton over S. We say that s satisfies $\mathcal{D}$, denoted $s \models \mathcal{D}$, iff $s \in \mathcal{L}(\mathcal{D})$. Our specification language is a *universal branching temporal logic* (e.g., $\forall CTL$, $\forall CTL^*$ [19]) with finite automata over S as the atomic formula. In this chapter, we only define $\forall CTL$, but the theorems hold for $\forall CTL^\star$ as well.

Definition 6.1.5 The logic $\forall CTL$ is defined inductively as follows:

1. The constant *true* is an atomic formula.
2. The specification automaton $\mathcal{D}$ is an atomic formula.
3. If ϕ is an atomic formula, then $\neg\phi$ is a formula.
4. If ϕ and ψ are formulas, then $\phi \wedge \psi$ and $\phi \vee \psi$ are formulas.
5. If ϕ and ψ are formulas, then $\mathbf{AX}\,\phi$, $\mathbf{A}(\phi\,\mathbf{V}\,\psi)$, and $\mathbf{A}(\phi\,\mathbf{U}\,\psi)$ are formulas.

Consider a LTS $M = (S^i, R, ACT, S_0)$. The satisfaction relation ($\models$) is inductively defined as follows. Given $s \in S^i$, we say:

1. $s \models \mathcal{D} \Leftrightarrow s \in \mathcal{L}(\mathcal{D})$

2. $s \models \neg f_1 \Leftrightarrow s \not\models f_1$

3. $s \models f_1 \vee f_2 \Leftrightarrow s \models f_1 \, \mathrm{or} \, s \models f_2$

4. $s \models f_1 \wedge f_2 \Leftrightarrow s \models f_1 \, \mathrm{and} \, s \models f_2$

5. $s \models \mathbf{AX}\, g_1$ iff for all states s' and for all actions α if $s \xrightarrow{\alpha} s'$, then $s' \models g_1$.

6. $s \models \mathbf{A}(g_1 \mathbf{\ U\ } g_2)$ iff for all paths π starting from s there exists $k \geq 0$ such that $\pi[k] \models g_2$ and for all $0 \leq j < k, \pi[j] \models g_1$.

7. $s \models \mathbf{A}(g_1 \mathbf{\ V\ } g_2)$ iff for all paths π starting from s and for every $k \geq 0$, if $\pi[j] \not\models g_1$ for all $0 \leq j < k$, then $\pi[k] \models g_2$.

Example 6.1.2 Consider again the network of Example 6.1.1. Let $\mathcal{D}$ be the automaton of Figure 6.3, defined over $S = \{cs, nc\}$, with $\mathcal{L}(\mathcal{D}) = \{nc\}^* cs \{nc\}^*$. The formula $\mathbf{AG}\, \mathcal{D}$ specifies mutual exclusion, i.e., at any moment there is exactly one process in the critical section. Let $\mathcal{D}'$ be an automaton that accepts the language $cs\{nc\}^*$, then the formula $\mathbf{AG}\, \mathbf{AF}\, \mathcal{D}'$ expresses non-starvation for process Q. Note that, for our simple example non-starvation is guaranteed only if some kind of fairness is assumed.

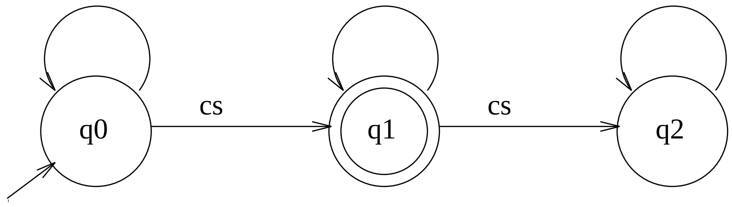

Figure 6.3: Automaton $\mathcal{D}$ with $\mathcal{L}(\mathcal{D}) = \{nc\}^* cs \{nc\}^*$

6.2 Abstract LTSs

In the following sections we define abstract LTSs and abstract composition in order to reduce the state space required for the verification of networks. The abstraction should preserve the logic under consideration. In particular, since we use $\forall CTL$, there must be a *simulation preorder* $\preceq$ such that the given LTS is smaller by $\preceq$ than the abstract LTS. We also require that composing two abstract states will result in an abstraction of their composition. This will allow us to replace the abstraction of a composed LTS by the composition of the abstractions of its components. For the sake of simplicity, we first assume that the specification language contains a single atomic formula $\mathcal{D}$. We later show how to extend the framework to a set of atomic formulas.

6.2.1 State equivalence

We start by defining an equivalence relation over the state set of an LTS. The equivalence classes will then be used as the abstract states of the abstract LTS. Given an LTS M, we define an equivalence relation on the states of M, such that if two states are equivalent then they both either satisfy or falsify the atomic formula. This means that the two states are either both accepted or both rejected by the automaton $\mathcal{D}$. We also require that our equivalence relation is preserved under composition. This means that if s_1 is equivalent to s_1' and s_2 is equivalent to s_2' then (s_1, s_2) is equivalent to (s_1', s_2').

We will use $h(M)$ to denote the abstract LTS of M. The straightforward definition that defines s and s' to be equivalent iff they both are in or not in the language $\mathcal{L}(\mathcal{D})$ has the first property, but does not have the second one. To see this, consider the following example.

Example 6.2.1 Consider LTSs defined by the grammar of Example 6.1.1. Let $\mathcal{D}$ be the automaton in Figure 6.3, i.e., $\mathcal{L}(\mathcal{D})$ is the set of states that have exactly one component in the critical section. Let s_1, s_1', s_2, s_2' be states such that $s_1, s_1' \in \mathcal{L}(\mathcal{D})$, and $s_2, s_2' \notin \mathcal{L}(\mathcal{D})$. Furthermore assume that, the number of components in the critical section are 0 in s_2 and 2 in s_2'. Clearly, $(s_1, s_2) \in \mathcal{L}(\mathcal{D})$ but $(s_1', s_2') \notin \mathcal{L}(\mathcal{D})$. Thus, the equivalence is not preserved under composition.

We therefore need a more refined equivalence relation. Our notion of equivalence is based on the idea that a word $w \in S^*$ can be viewed as a function on the set of states of an automaton. We define two states to be equivalent if and only if they induce the same function on the automaton $\mathcal{D}$. Formally, given an automaton $\mathcal{D} = (Q, q_0, \delta, F)$ and a word $w \in S^\star$, $f_w : Q \mapsto Q$, the *function induced* by w on Q is defined by

$$f_w(q) = q' \text{ iff } q \xrightarrow{w} q'.$$

Note that $w \in \mathcal{L}(\mathcal{D})$ if and only if $f_w(q_0) \in F$, i.e., w takes the initial state to a final state. The set of functions associated with an automaton $\mathcal{D}$ is a monoid and has been studied extensively.

Let $\mathcal{D} = (Q, q_0, \delta, F)$ be a deterministic automaton. Let f_w be the function induced by a word w on Q. Then, two states s and s' are *equivalent*, denoted by $s \equiv s'$, iff $f_s = f_{s'}$. It is easy to see that $\equiv$ is an equivalence relation. The function f_s corresponding to the state s is called the *abstraction* of s and is denoted by $h(s)$. Let $h(s) = f_1$ and $h(s') = f_1'$. The abstract state of (s, s') is $h((s, s')) = f_1 \circ f_1'$ where $f_1 \circ f_1'$ denotes composition of functions. We follow the convention that when we write $f_1 \circ f_1'$ we apply f_1 before f_1'. Note that $s \equiv s'$ implies that $s \in \mathcal{L}(\mathcal{D}) \Leftrightarrow s' \in \mathcal{L}(\mathcal{D})$. Thus, we have $s \models \mathcal{D}$ iff $s' \models \mathcal{D}$. We also have,

Lemma 6.2.1 If $h(s_1) = h(s_2)$ and $h(s_1') = h(s_2')$ then $h((s_1, s_1')) = h((s_2, s_2'))$.

In order to interpret specifications on the abstract LTSs, we extend $\models$ to abstract states so that $h(s) \models \mathcal{D}$ iff $f_s(q_0) \in F$. This guarantees that $s \models \mathcal{D}$ iff $h(s) \models \mathcal{D}$.

Our framework can easily be extended to any set of atomic formulas. The restriction to one atomic formula was done in order to simplify the presentation. However, in practice we may want to have several such formulas, related by boolean and temporal operators. The notion of equivalence is extended to any set of atomic formulas in the following way. Let $\mathcal{AF} = \{\mathcal{D}_1, \ldots, \mathcal{D}_k\}$ be a set of atomic formulas, where $\mathcal{D}_i = (Q_i, q_0^i, \delta_i, F_i)$. Let f_s^i be the function induced by s on Q_i. Then, two states s and s' are *equivalent* if and only if for all i, $f_s^i = f_{s'}^i$. The abstraction of s is now $h(s) =< f_s^1, \ldots, f_s^k >$, and we have that, if

$s \equiv s'$ then for every i, $s \in \mathcal{L}(\mathcal{D}_i) \Leftrightarrow s' \in \mathcal{L}(\mathcal{D}_i)$. The relation $\models$ is extended for abstract states by defining $h(s) \models \mathcal{D}_i$ iff $f^i_s(q^i_0) \in F_i$. Thus, we again have that for every $\mathcal{D}_i \in \mathcal{AF}$, $s \models \mathcal{D}_i$ iff $h(s) \models \mathcal{D}_i$.

Example 6.2.2 Consider again the automaton $\mathcal{D}$ of Figure 6.3 over $S = \{cs, nc\}$. $\mathcal{D}$ induces functions $f_s : Q \mapsto Q$, for every $s \in S^*$. Actually, there are only three different functions, each identifying an equivalence class over S^*. $f_1 = \{(q_0, q_0), (q_1, q_1), (q_2, q_2)\}$ represents all $s \in nc^*$ (i.e., $f_s = f_1$ for all $s \in nc^*$). $f_2 = \{(q_0, q_1), (q_1, q_2), (q_2, q_2)\}$ represents all $s \in nc^*\ cs\ nc^*$, and $f_3 = \{(q_0, q_2), (q_1, q_2), (q_2, q_2)\}$ represents all $s \in nc^*\ cs\ nc^*\ cs\ \{cs, nc\}^*$.

6.2.2 Abstract process and Abstract Composition

Let $\mathcal{F}_\mathcal{D}$ be the set of functions corresponding to the deterministic automaton $\mathcal{D}$. Let Q be the set of states $\mathcal{D}$. In the worst case $|\mathcal{F}_\mathcal{D}| = |Q|^{|Q|}$, but in practice the size is much smaller. Note that $\mathcal{F}_\mathcal{D}$ is also the set of abstract states for $s \in \Sigma^*$ with respect to $\mathcal{D}$. Subsequently, we will apply abstraction both to states $s \in \Sigma^*$ and to abstract states f_s for $s \in \Sigma^*$. To unify notation we first extend the abstraction function h to $\mathcal{F}_\mathcal{D}$ by setting $h(f) = f$ for $f \in \mathcal{F}_\mathcal{D}$. We now further extend the abstraction function h to $(S \cup \mathcal{F}_\mathcal{D})^*$ in the natural way, i.e., $h((a_1, a_2, \cdots, a_n)) = h(a_1) \circ \cdots \circ h(a_n)$. From now on we will consider LTSs in the network $\mathcal{N}$ on the tuple $(S \cup \mathcal{F}_\mathcal{D}, ACT)$.

We now define abstract LTSs. The abstract transition relation is defined as usual for an abstraction that should be greater by the simulation preorder than the original structure [25]. If there is a transition between one state and another in the original structure, then there is a transition between the abstract state of the one to the abstract state of the other in the abstract structure. Formally,

Definition 6.2.1 Given an LTS $M = (S^i, R, ACT, S_0)$ in the network $\mathcal{N}$, the corresponding *abstract LTS* is defined by $h(M) = (S^h, R^h, ACT, S^h_0)$, where

- $S^h = \{h(s) \mid s \in S^i\}$ is the set of abstract states.
- $S^h_0 = \{h(s) \mid s \in S_0\}$.

- The relation R^h is defined as follows. For any $h_1, h_2 \in S^h$, and $a \in ACT$

$$(h_1, a, h_2) \in R^h \quad \Leftrightarrow \quad \exists s_1, s_2 [h_1 = h(s_1) \text{ and } h_2 = h(s_2) \text{ and } (s_1, a, s_2) \in R].$$

We say that $M' = (S', R', Act, S'_0)$ *simulates* $M = (S, R, Act, S_0)$ [57] (denoted $M \preceq M'$) if and only if there is a *simulation preorder* $\mathcal{E} \subseteq S \times S'$ that satisfies the following conditions: for every $s_0 \in S_0$ there is $s'_0 \in S'_0$ such that $(s_0, s'_0) \in \mathcal{E}$. Moreover, for every s, s', if $(s, s') \in \mathcal{E}$ then

1. $h(s) = h(s')$.
2. For every s_1 such that $s \xrightarrow{a} s_1$ there is s'_1 such that $s' \xrightarrow{a} s'_1$ and $(s_1, s'_1) \in \mathcal{E}$.

Notice that the definition of the simulates relation uses the fact that M and M' are defined on the same action sets. If $(s, s') \in \mathcal{E}$, we say that $s \preceq s'$. Given a path π in M and a path π' in M', we say that $\pi \preceq \pi'$ iff $\pi[i] \preceq \pi'[i]$ for all i.

Lemma 6.2.2 Let M and M' be two LTSs such that $M \preceq M'$. Let π be a path starting from s in M. If $s \preceq s'$, then there exists a path π' starting from s' in M' such that $\pi \preceq \pi'$.

Proof: Proof follows from the definition of the simulation relation $\mathcal{E}$.

Lemma 6.2.3 We have the following results.

1. $M \preceq h(M)$, i.e., $h(M)$ simulates M.
2. If $M \preceq M'$, then $h(M) \preceq h(M')$.

Proof: For the proof of the first part, consider the following simulation relation:

$$(s, a) \in \mathcal{E} \quad \Leftrightarrow \quad (h(s) = a)$$

It is easy to prove that $\mathcal{E}$ has all the desired properties. For the second part, let $\mathcal{E}$ be the simulation relation between M and M'. Define the relation $\mathcal{E}_h$ in the following manner:

$$(a, b) \in \mathcal{E}_h \quad \Leftrightarrow \quad \exists s_1, s_2 (h(s_1) = a \ \wedge h(s_2) = b \ \wedge \ (s_1, s_2) \in \mathcal{E})$$

It is easy to prove that $\mathcal{E}_h$ is the required relation between $h(M)$ and $h(M')$. $\Box$

Recall that the abstraction h guarantees that a state and its abstraction agree on the atomic property corresponding to the automaton $\mathcal{D}$. Based on that and on the previous lemma, the following theorem is obtained. A proof of a similar result appears in [19].

Theorem 6.2.1 Let ϕ be a formula in $\forall CTL$ over atomic formula $\mathcal{D}$. Let M and M' be two LTSs such that $M \preceq M'$. Let $s \preceq s'$. Then $s' \models \phi$ implies $s \models \phi$.

Proof: The proof is by structural induction on ϕ. Notice that $(s' \models \phi \Rightarrow s \models \phi)$ is equivalent to $(s \not\models \phi \Rightarrow s' \not\models \phi)$. Sometimes we will use the second formulation.

- **Case** $\phi = \mathcal{D}$: $h(s) = h(s')$ implies that $s \models \mathcal{D}$ if and only $s' \models \mathcal{D}$.

- **Case** $\phi = f_1 \vee f_2$: Assume that $s' \models \phi$. By definition $s' \models f_1$ or $s' \models f_2$. By the inductive hypothesis $s \models f_1$ or $s \models f_2$. Therefore, $s \models \phi$.

- **Case** $\phi = f_1 \wedge f_2$: This is very similar to the case given above.

- **Case** $\phi = \mathbf{AX}\, g_1$: Assume that $s \not\models \phi$. Then there exists s_1 such that $s \xrightarrow{\alpha} s_1$, and $s_1 \not\models g_1$. By definition there exists s_1' such that $s' \xrightarrow{\alpha} s_1'$ and $s_1 \preceq s_1'$. By the inductive hypothesis $s_1' \not\models g_1$. Therefore, $s' \not\models \phi$.

- **Case** $\phi = \mathbf{A}(g_1 \,\mathbf{U}\, g_2)$: Assume that $s \not\models \phi$. This means that there exists a path π starting from s such that for all k, either $\pi[k] \not\models g_2$ or there exists $0 \leq j < k$ such that $\pi[j] \not\models g_1$. Using Lemma 6.2.2 there exists a path π' starting from s' in M' such that $\pi \preceq \pi'$, i.e., for every $i \geq 0$, $\pi[i] \preceq \pi'[i]$. By the induction hypothesis we have that for all k, either $\pi'[k] \not\models g_2$ or there exists $0 \leq j < k$ such that $\pi'[j] \not\models g_1$. Thus, $\pi \not\models g_1 \,\mathbf{U}\, g_2$, which implies that $s' \not\models \phi$.

- **Case $\phi = \mathbf{A}(g_1 \,\mathbf{V}\, g_2)$:** Assume that $s' \models \phi$. We will prove that $s \models \phi$. Let π be an arbitrary path starting from s in M. Let π' be a path starting from s' in M' such that $\pi \preceq \pi'$. Since $s' \models \phi$, for all k, either there exists a j such that $0 \leq j < k$, $\pi'[j] \models g_1$ or $\pi'[k] \models g_2$. By the induction hypothesis, we can conclude that for k, $\pi[k] \models g_2$ or there exists j such that $0 \leq j < k$ and $\pi[j] \models g_1$. Therefore, $\pi \models g_1 \,\mathbf{V}\, g_2$. Since π is an arbitrary path starting from s, we have that $s \models \phi$. $\square$

Definition 6.2.2 A composition $\|$ is called *monotonic* with respect to a simulation preorder $\preceq$ iff given LTSs such that $M_1 \preceq M_2$ and $M_1' \preceq M_2'$, we have that $M_1 \| M_1' \preceq M_2 \| M_2'$. A network grammar G is called *monotonic* if and only if all rules in the grammar use only monotonic composition operators.

6.3 The Verification method

Given a network grammar G, we associate with each symbol A of the grammar a *representative process* $rep(A)$. A set of representative processes is said to satisfy the *monotonicity property* iff

- For every terminal A, $h(rep(A)) \succeq h(A)$.
- For every rule $A \rightarrow B \| C$ we have the following condition:

$$h(rep(A)) \;\; \succeq \;\; h(h(rep(B)) \| h(rep(C)))$$

We have the following theorem:

Theorem 6.3.1 Let G be a monotonic grammar and suppose we can find representatives for the symbols of G which satisfy the monotonicity property. Let A be a symbol of the grammar G and let a be an LTS derived from A using the rules of the grammar G. Then, $h(rep(A)) \succeq a$.

Proof: We will prove that $h(rep(A)) \succeq h(a)$. Since $h(a) \succeq a$, the result follows by transitivity. Let $A \Rightarrow^k a$, i.e., A derives a in k steps. Our result is by induction on k.

- ($k = 0$): In this case A is a terminal and $a = A$. The result follows from the monotonicity property.

- ($k > 0$): In the derivation of a from A, let the first rule be $A \rightarrow B \| C$. Assume $B \Rightarrow^i b$ and $C \Rightarrow^j c$ such that $i < k$, $j < k$, and $a = b \| c$. By the induction hypothesis $h(rep(B)) \succeq h(b)$ and $h(rep(C)) \succeq h(c)$. We have the following equations:

$$
\begin{array}{rcl}
h(rep(B)) \| h(rep(C)) & \succeq & h(b) \| h(c) \;\; \textit{(monotonicity of } \| \textit{)} \\
 & \succeq & b \| c \;\textit{(Lemma 6.2.3 and monotonicity of } \| \textit{)} \\
h(h(rep(B)) \| h(rep(C))) & \succeq & h(b \| c) \;\textit{(Lemma 6.2.3)} \\
 & \succeq & h(a)
\end{array}
$$

  By the monotonicity property we have:

$$h(rep(A)) \;\; \succeq \;\; h(h(rep(B)) \| h(rep(C)))$$

The result follows. $\square$

6.3.1 Verification Method

Here we describe our verification method. Assume that we are given a monotonic grammar G and a $\forall CTL$ formula ϕ with atomic formulas $\mathcal{D}_1, \cdots, \mathcal{D}_k$. To check that every LTS derived by the grammar G satisfies ϕ we perform the following steps:

1. For every symbol A in G choose a representative process $rep(A)$ and construct the abstract LTS $h(rep(A))$ with respect to the atomic formulas $\mathcal{D}_1, \cdots, \mathcal{D}_k$.

2. Check that the set of representatives satisfy the monotonicity property. By Theorem 6.3.1 we have that for every a derived by the grammar G, $h(rep(\mathcal{S})) \succeq a$.

3. Model Check ϕ on $h(rep(S))$. If $h(rep(S)) \models \phi$, then by Theorem 6.2.1 we can conclude that for all LTSs M derived by the grammar G, $M \models \phi$.

Next, we describe an unfolding heuristic which might help us find the set of representatives which satisfy the monotonicity property.

6.3.2 The Unfolding Heuristic

We discuss a heuristic that might be helpful in finding representatives automatically. Initially, the representative of a symbol A in G is the smallest LTS that can be derived from A using the rules of G. A further search for monotonic representatives is based on the observation that often certain behaviors only occur when a process is composed with other processes (these other processes provide the environment). This leads to the idea that by *unfolding* the derivation rules of G we may find a larger set of potential representatives which might satisfy the monotonicity property. Given two sets of LTSs X and Y we define $X \| Y$ in the following way:

$$X \| Y \quad = \quad \{p \| q \mid p \in X \text{ and } q \in Y\}$$

Now we describe a heuristic to find representatives with the monotonicity property. An *association* for a grammar G assigns a set of processes $set(A)$ to each symbol A of the grammar. Given an association for a grammar G, we define an *unfolding* of the association which is again an association. The unfolding associates a new set of processes $set'(A)$ with each symbol A of grammar G in the following manner:

- If A is a terminal, $set'(A) = set(A)$.
- Assume that A is a non-terminal. A process $p \in set'(A)$ iff there exists a rule $A \to B \| C$ such that $p \in set(B) \| set(C)$ or $p \in set(A)$. Note that B or C may be A itself.

Now we define an iterative process to find representatives. First, we describe how to find the initial association. For a terminal A, $set_0(A) = \{A\}$. We repeat the following step until we have a non-empty association for each symbol A.

- For each rule $A \rightarrow B \| C$ if $set_0(A)$ is empty and $set_0(B)$ and $set_0(C)$ are defined then:

$$set_0(A) \;=\; set_0(B) \| set_0(C)$$

Each iteration consists of the following two steps.

1. If there exists representatives for each nonterminal A such that $rep(A) \in set_i(A)$ and the representatives satisfy the monotonicity property, we are done. Otherwise go to the next step.

2. Create a new association $set_{i+1}(A)$ by unfolding the i-th association. Go back to the previous step.

It is not hard to see that each iteration increases the set of processes associated with a nonterminal. Therefore, as we unfold, we have more choices to find representatives with the required property. In the examples we show how unfolding is used to find representatives. Notice that if we find representatives with the monotonicity property such that $h(rep(\mathcal{S})) \not\models \phi$, we cannot conclude anything about the correctness of the network derived by the grammar G. In this case the counter-example might aid the user in finding a more refined invariant or we may want to apply the unfolding technique again.

6.4 Synchronous model of computation

In this section we develop a synchronous framework that will have the properties required by our verification method. We define a synchronous model of computation and a family of composition operators. We show that the composition operators are monotonic with respect to $\preceq$.

Our models are a form of LTSs, $M = (S, R, I, O, S_0)$, that represent *Moore machines*. Traditionally, in Moore machines the outputs are associated with the states. We assume that the outputs are moved from the states to the transitions emanating from it. Our models have an explicit notion of inputs I and outputs O that must be disjoint. In addition, they have a special internal action denoted by τ (called silent action in the terminology of CCS [58]). The set of actions is

$ACT = \{\tau\} \cup 2^{I \cup O}$, where each non-internal action is a set of inputs and outputs.

The composition of two LTSs M and M' is defined to reflect the synchronous behavior of our model. It corresponds to standard composition of Moore machines. To understand how this composition works, we can think of the inputs and outputs as "wires". If M has an output and M' has an input both named a, then in the composition the output wire a will be connected to the input a. Since an input can accept signal only from one output, $M \| M'$ will not have a as input. On the other hand, an output can be sent to several inputs, thus $M \| M'$ still has a as output. Consequently, the set of outputs of $M \| M'$ is $O \cup O'$ while the set of inputs is $(I \cup I') \setminus (O \cup O')$.

A transition $s \xrightarrow{a} t$ from s in a machine M with $a = i \cup o$ such that $i \subseteq I$ and $o \subseteq O$ occurs only if the environment supplies inputs i and the machine M produces the outputs o. Assume transitions $s \xrightarrow{a} t$ in M and $s' \xrightarrow{a'} t'$ in M'. There is a transition from (s, s') to (t, t') iff the outputs provided by M agree with the inputs expected by M' and the outputs provided by M' agree with the inputs expected by M.

Formally, let $O \cap O' = \emptyset$. The *synchronous composition* of M and M', $M'' = M \parallel M'$ is defined by:

1. $S'' = S \times S'$.
2. $S_0'' = S_0 \times S_0'$.
3. $I'' = (I \cup I') \setminus (O \cup O')$.
4. $O'' = O \cup O'$.[1]
5. $(s, s') \xrightarrow{a''} (s_1, s_1')$ is a transition in R'' iff the following holds: $s \xrightarrow{a} s_1$ is a transition in R and $s' \xrightarrow{a'} s_1'$ is a transition in R' for some a, a' such that $a \cap (I' \cup O') = a' \cap (I \cup O)$ and $a'' = a \cup a'$.

Lemma 6.4.1 The composition $\parallel$ is monotonic with respect to $\preceq$.

[1]Note that, $ACT'' = 2^{I'' \cup O''} \cup \{\tau\}$ is not identical to ACT and ACT'. This is a technical issue that can be resolved by defining some superset of actions from which each LTS takes its actions.

Proof: Let $M = (S, R, I, O, S_0)$, $M' = (S', R', I', O', S'_0)$, $M_1 = (S_1, R_1, I_1, O_1, S_{1,0})$, and $M'_1 = (S'_1, R'_1, I'_1, O'_1, S'_{1,0})$ be four Moore machines. Assume that $M \preceq M'$ and $M_1 \preceq M'_1$. Let $\mathcal{E} \subseteq S \times S'$ and $\mathcal{E}_1 \subseteq S_1 \times S'_1$ be the corresponding simulation relations. We say that $((s, s_1), (s', s'_1)) \in \mathcal{E} \times \mathcal{E}_1$ iff $(s, s') \in \mathcal{E}$ and $(s_1, s'_1) \in \mathcal{E}_1$. We will show that $\mathcal{E} \times \mathcal{E}_1$ has the required properties. It is clear from the definition that given states $s_0 \in S_0$ and $s_{0,1} \in S_{1,0}$, there exists a $s'_0 \in S'_0$ and $s'_{1,0} \in S'_{1,0}$ such that

$$((s_0, s_{1,0}), (s'_0, s'_{1,0})) \quad \in \quad \mathcal{E} \times \mathcal{E}_1$$

Assume that $((s, s_1), (s', s'_1)) \in \mathcal{E} \times \mathcal{E}_1$.

1. By assumption, we have that $h(s) = h(s')$ and $h(s_1) = h(s'_1)$. Therefore, $h((s, s_1)) = h(s) \circ h(s_1)$ is equal to $h((s', s'_1)) = h(s') \circ h(s'_1)$.

2. Let $(s, s_1) \xrightarrow{a''} (t, t_1)$ be a transition in $M \| M_1$. This means that there exists a transition $s \xrightarrow{a} t$ in M and a transition $s_1 \xrightarrow{a_1} t_1$ in M_1 such that $a \cap (I_1 \cup O_1) = a_1 \cap (I \cup O)$ and $a'' = a \cup a_1$ By definition there exists t' and t'_1 such that $s' \xrightarrow{a} t'$ and $s'_1 \xrightarrow{a_1} t'_1$, where $(t, t') \in \mathcal{E}$ and $(t_1, t'_1) \in \mathcal{E}_1$. Therefore, $(s', s'_1) \xrightarrow{a''} (t', t'_1)$ and $((t, t_1), (t', t'_1)) \in \mathcal{E} \times \mathcal{E}_1$. $\Box$

6.4.1 Network grammars for synchronous models

Only a few additional definitions are required in order to adapt our general definition of network grammars to networks of synchronous models. Like before a network grammar is a tuple $G = \langle T, N, \mathcal{P}, \mathcal{S} \rangle$, but now, every terminal and nonterminal A in $T \cup N$ is associated with a set of inputs I_A and a set of outputs O_A. In G we allow different composition operators $\|_i$ for the different production rules. In order to define the family of operators to be used in this framework we need the following definitions.

A *renaming function* $\mathcal{R}$ is an injection which acts on the input and outputs of the Moore machines. When applied to A, it maps inputs to inputs and outputs to outputs such that $\mathcal{R}(I_A) \cap \mathcal{R}(O_A) = \emptyset$. Applying

$\mathcal{R}$ to a LTS M results in an LTS $M' = \mathcal{R}(M)$ with $S = S'$, $S_0 = S'_0$, $I' = \mathcal{R}(I)\backslash\{\tau\}$, $O' = \mathcal{R}(O)\backslash\{\tau\}$, and $(s, a, s') \in R$ iff $(s, \mathcal{R}(a), s') \in R'$.

A *hiding function* $\mathcal{R}_{act}$ for $act \subseteq I \cup O$, is a renaming function that maps each element in act to τ.

A typical composition operator in this family is associated with two renaming functions, $\mathcal{R}_{\texttt{left}}$, $\mathcal{R}_{\texttt{right}}$ and a hiding function $\mathcal{R}_{act}$, in the following way.

$$M\|_i M' = \mathcal{R}_{act}(\mathcal{R}_{\texttt{left}}(M)\|\mathcal{R}_{\texttt{right}}(M')),$$

where $\|$ is the synchronous composition defined before. The renaming functions are applied to LTSs and not to nonterminals. Therefore, the derivations are composed bottom up. For example, consider the rule of the form shown below:

$$A \rightarrow \mathcal{R}_{act}(\mathcal{R}_{\texttt{left}}(B)\|\mathcal{R}_{\texttt{right}}(C))$$

In this case first we apply the derivations for B and C to derive two LTS b and c. Then the renaming and hiding functions are applied to complete the derivation.

To use our framework, we need to show that every such operator is monotonic, i.e., if $M_1 \preceq M_2$ and $M'_1 \preceq M'_2$ then $M_1\|_i M'_1 \preceq M_2\|_i M'_2$. The latter means that

$$\mathcal{R}_{act}(\mathcal{R}_{\texttt{left}}(M_1)\|\mathcal{R}_{\texttt{right}}(M'_1)) \quad \preceq \quad \mathcal{R}_{act}(\mathcal{R}_{\texttt{left}}(M_2)\|\mathcal{R}_{\texttt{right}}(M'_2))$$

The following lemma, together with monotonicity of the synchronous composition $\|$ imply the required result.

Lemma 6.4.2 Let M, M' be Moore Machines and let $\mathcal{R}$ be a renaming function. If $M \preceq M'$ then $\mathcal{R}(M) \preceq \mathcal{R}(M')$.

Proof: Let $\mathcal{E}$ be the simulation preorder between M and M'. It is easy to show that $\mathcal{E}$ is also a simulation preorder between $\mathcal{R}(M)$ and $\mathcal{R}(M')$. $\square$

Corollary 6.4.1 The composition operators $\|_i$, defined as above are monotonic.

Example 6.4.1 We return to Example 6.1.1 and reformulate it within the synchronous framework. Doing so we can describe more precisely the processes and the network grammar that constructs rings with any number of processes. The processes P and Q will be identical to those described in Figure 6.1 except that now we also specify for both processes $I = \{$ `get-token`$\}$ and $O = \{$`send-token`$\}$.

The derivation rules in the grammar apply two different composition operators:

$$\begin{aligned} \mathcal{S} &\rightarrow Q\|_1 A \\ A &\rightarrow P\|_2 A \\ A &\rightarrow P\|_2 P \end{aligned}$$

$\|_1$ is defined as follows.

- $\mathcal{R}^1_{\texttt{left}}$ maps `send-token` to some new action `cr` (stands for `connect right`) and `get-token` to `cl` (stands for `connect left`).
- $\mathcal{R}^1_{\texttt{right}}$ maps `send-token` to `cl` and `get-token` to `cr`.
- The hiding function hides both `cr` and `cl` by mapping them to τ.

Thus, the application of this rule results in a network with one terminal Q and one nonterminal A, connected as a *ring*. $\|_2$ is defined by (see Figure 6.4):

- $\mathcal{R}^2_{\texttt{left}}$ maps `send-token` to `cr` and leaves `get-token` unchanged.
- $\mathcal{R}^2_{\texttt{right}}$ maps `get-token` to `cr` and leaves `send-token` unchanged.
- The hiding function hides `cr`

The application of the third rule, for instance, results in a network in which the nonterminal A is replaced by a LTS consisting of two processes P, such that the `send-token` of the left one is connected to the `get-token` of the right one. The `get-token` of the left process and `send-token` of the right one will be connected according to the connections of A (see Figure 6.4 and Figure 6.5).

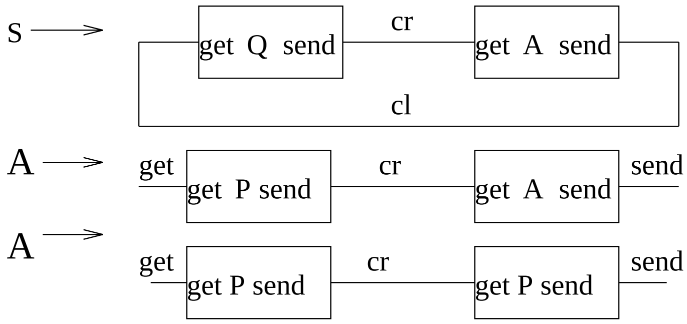

Figure 6.4: Derivation rules with renaming

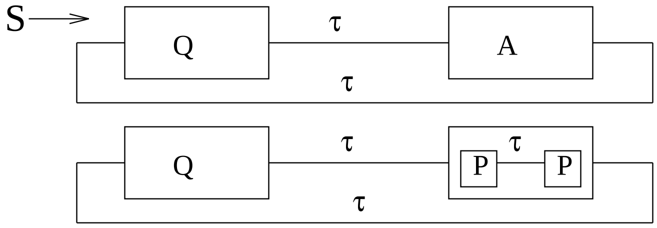

Figure 6.5: Derivation of a ring of size 3

6.5 Asynchronous Model of Computation

In this section we describe a model for asynchronous communication. The model is similar to CCS [58]. Each process is associated with a set of actions. The symbol τ denotes the internal (non communication) actions of a process.

Definition 6.5.1 A **process** M is a 4-tuple,

$$M = (ACT, S, R, S_0)$$

where

- ACT is a finite set of actions (ports or channels) not containing τ.
- S is a finite set of states.
- R is a labeled transition relation, $R \subseteq S \times (ACT \cup \{\tau\}) \times S$.

- S_0 is the set of initial states.

When two processes M and M' are combined into a new process $M\|M'$, zero or more channels are formed. Each channel pairs a port of M and a port of M' that have identical names. The channel is used merely for synchronization (i.e., no data is transferred). However, data can be encoded by sequences of synchronizations. The unpaired ports of M and M' become the ports of the combined process $M\|M'$.

Definition 6.5.2 The **composition** of $M = (S, R, ACT, S_0)$ and $M' = (S', R', ACT', S'_0)$ is a new process

$$M'' = M\|M' = (S'', R'', ACT'', S''_0)$$

where

- $ACT'' = (ACT \cup ACT')\backslash(ACT \cap ACT')$
- $S'' = (S \times S')$
- $S''_0 = S_0 \times S'_0$
- $R'' =$

$$\begin{array}{l}\{((s,s'),\tau,(s_1,s'_1)) \ \mid \ \exists\alpha \ : \ s \stackrel{\alpha}{\to} s_1 \ \wedge \ s' \stackrel{\alpha}{\to} s'_1 \ \wedge \ \alpha \in ACT \cap ACT'\Big\} \cup \\ \{((s,s'),\alpha,(s_1,s')) \ \mid \ s \stackrel{\alpha}{\to} s_1 \ \wedge \ \alpha \notin ACT'\Big\} \cup \\ \{((s,s'),\alpha,(s,s'_1)) \ \mid \ s' \stackrel{\alpha}{\to} s'_1 \ \wedge \ \alpha \notin ACT\Big\}\end{array}$$

When combining processes, we sometimes need to change their action names in order to form a network of a desirable structure. The definition of renaming and hiding functions are similar to the one introduced for the synchronous model. Let $\mathcal{R}$ be an 1-1 *renaming function* of the ports of the process M. $\mathcal{R}(M)$ denotes a new process M' identical to M except that:

- $s \stackrel{\mathcal{R}(\alpha)}{\to} t \in R'$ iff $s \stackrel{\alpha}{\to} t \in R$.

A *hiding function* $\mathcal{R}_{act}$ (where $act \subseteq ACT$) disallows synchronization on the actions in the set act by renaming them to the silent action τ. Formally $\mathcal{R}_{act}(M)$ is a new process M', which is identical to M except that its transition relation R' is:

- $s \xrightarrow{\alpha} t \in R'$ iff $s \xrightarrow{\alpha} t \in R$ and $\alpha \notin act$.
- $s \xrightarrow{\tau} t \in R'$ iff $s \xrightarrow{\alpha} t \in R$ and $\alpha \in act$.

The network grammar $G = \langle T, N, \mathcal{P}, \mathcal{S} \rangle$ has the production rules of the following form.

- $\mathcal{P}$ is the set of production rules of the form

$$A \rightarrow \mathcal{R}_{act}(\mathcal{R}_{\mathtt{left}}(B) \| \mathcal{R}_{\mathtt{right}}(C))$$

where $A \in N$, $B, C \in T \cup N$, $\mathcal{R}_{\mathtt{left}}$ and $\mathcal{R}_{\mathtt{right}}$ are renaming functions, and $\mathcal{R}_{act}$ is the hiding function for that rule.

We have the following theorems.

Theorem 6.5.1 (Monotonicity theorem) Let $M = (S, R, ACT, S_0)$, $M_1 = (S_1, R_1, ACT_1, S_{0,1})$, $M' = (S', R', ACT, S'_0)$, $M'_1 = (S'_1, R'_1, ACT_1, S'_{0,1})$ be processes such that $M \preceq M'$ and $M_1 \preceq M'_1$. In this case $M \| M_1 \preceq M' \| M'_1$.

Proof: Let $\mathcal{E} \subseteq S \times S'$ and $\mathcal{E}_1 \subseteq S_1 \times S'_1$ be the simulation relations. Consider the following relation $\mathcal{E} \times \mathcal{E}_1 \subseteq (S \times S_1) \times (S' \times S'_1)$

- $((s, s_1), (s', s'_1)) \in \mathcal{E} \times \mathcal{E}_1$ iff $(s, s') \in \mathcal{E}$ and $(s_1, s'_1) \in \mathcal{E}_1$

We will prove that $\mathcal{E} \times \mathcal{E}_1$ is the simulation relation between $M \| M_1$ and $M' \| M'_1$, and hence, $M \| M_1 \preceq M' \| M'_1$. Note that set of actions of $M \| M_1$ and $M' \| M'_1$ are both identical to $(ACT \cup ACT_1) \setminus (ACT \cap ACT_1)$.

- For all $(s_0, s_{0,1}) \in S_0 \times S_{0,1}$ there exists $(s'_0, s'_{0,1}) \in S'_0 \times S'_{0,1}$ such that

$$((s_0, s_{0,1}), (s'_0, s'_{0,1})) \in \mathcal{E} \times \mathcal{E}_1$$

This follows because we can find $s'_0 \in S'_0$ and $s'_{0,1} \in S'_{0,1}$ such that $(s_0, s'_0) \in \mathcal{E}$ and $(s_{0,1}, s'_{0,1}) \in \mathcal{E}_1$.

- Let $((s, s_1), (t, t_1)) \in \mathcal{E} \times \mathcal{E}_1$. By assumption, we have that $h(s) = h(t)$ and $h(s_1) = h(t_1)$. Therefore, $h((s, s_1)) = h(s) \circ h(s_1)$ is equal to $h((t, t_1)) = h(t) \circ h(t_1)$. Assume that $(s, s_1) \xrightarrow{\alpha} (s', s'_1)$. There are three cases.

- $(s, s_1) \xrightarrow{\tau} (s', s_1')$ and $s \xrightarrow{\alpha} s'$, $s_1 \xrightarrow{\alpha} s_1'$ for $\alpha \in ACT \cap ACT_1$. In this case, there exists t' and t_1' such that $t \xrightarrow{\alpha} t'$ and $t_1 \xrightarrow{\alpha} t_1'$ such that $(s', t') \in \mathcal{E}$ and $(s_1', t_1') \in \mathcal{E}_1$. It is clear that $(t, t_1) \xrightarrow{\tau} (t', t_1')$ and $((s', s_1'), (t', t_1'))$ is in $\mathcal{E} \times \mathcal{E}_1$.
- $(s, s_1) \xrightarrow{\alpha} (s', s_1')$ and $s \xrightarrow{\alpha} s'$, $s_1 = s_1'$ and $\alpha \notin ACT_1$. In this case, there exists t' such $t \xrightarrow{\alpha} t'$ and $(s', t') \in \mathcal{E}$. It is clear that $(t, t_1) \xrightarrow{\alpha} (t', t_1)$ and $((s', s_1), (t', t_1))$ is in $\mathcal{E} \times \mathcal{E}_1$. Notice that by assumption $(s_1, t_1) \in \mathcal{E}_1$.
- $(s, s_1) \xrightarrow{\alpha} (s', s_1')$ and $s = s'$, $s_1 \xrightarrow{\alpha} s_1'$, and $\alpha \notin ACT$. This case is similar to the one given above. $\Box$

The theorem given below states that renaming and hiding preserve monotonicity.

Theorem 6.5.2 Let M and M' be two processes such that $M \preceq M'$. Let $\mathcal{R}$ and $\mathcal{R}_{act}$ be renaming and hiding functions respectively. In this case $\mathcal{R}(M) \preceq \mathcal{R}(M')$ and $\mathcal{R}_{act}(M) \preceq \mathcal{R}_{act}(M')$.

Proof: The proof of this theorem is straightforward.$\Box$

Using the two theorems given above we can prove that the composition operators used in the grammar are monotonic.

6.6 Examples

We implemented the algorithm for network verification and applied it to two examples of substantial complexity. The first example uses the asynchronous composition. The second example uses the synchronous model of composition.

6.6.1 Dijkstra's Token Ring

The first is the famous Dijkstra's token ring algorithm [26]. This algorithm is significantly more complicated than the one used as a running example along the chapter. The example uses the asynchronous model of computation. There is a token t which passes in the clockwise direction. To avoid the token from passing unnecessarily, there is a signal r (r stands for request) which passes in the counter-clockwise direction.

Whenever a process wishes to have the token, it sends the signal r to its left neighbor, i.e., the process on the counter-clockwise side. The states of the processes have the following four components:

- It is either b (black–an interest in the token exists to the right), w (white–no one is interested in the token)
- It is either n (in the neutral state), d (the process is delayed waiting for the token) or c (the process is in the critical section)
- It is either t (with the token), or e (empty–without the token).
- A set of outputs o enabled from the state. The possible values of o are $\emptyset$, $\{r\}$, $\{t\}$, and $\{r, t\}$.

Each process has $r?$ and $t?$ as input actions and $r!$ and $t!$ as output actions. This notation is borrowed from CSP. While synchronizing, $r!$ is matched with $r?$. Similarly, $t!$ is matched with $t?$. Each state has an output associated with it (kept in the component o). The name of the state is a combination of its properties. Thus $\langle wne, \{r\}\rangle$ is a neutral state with no request on the right, no token and has output r. We will describe a state with a 2-tuple, i.e., $\langle x, o\rangle$ where x describes the first three components and o is the set of outputs enabled from that state. If the output from a state s' is non-empty ($o \neq \emptyset$), then there is a transition to s' with every action in o labelling the transition. Given a state $\langle x, o\rangle$, the following transitions are present in the system:

$$\forall(\alpha \in o)\left[\langle x, o\rangle \stackrel{\alpha!}{\to} \langle x, o\backslash\{\alpha\}\rangle\right]$$

For conciseness, the list of transitions for a process that performs the token ring protocol is given in the table below.

$wne \stackrel{r/r}{\to} bne$	$wne \stackrel{/r}{\to} wde$	$bne \to bde$
$bne \stackrel{t/t}{\to} wne$	$wde \stackrel{r/}{\to} bde$	$wde \stackrel{t/}{\to} wct$
$bde \stackrel{t/}{\to} bct$	$wnt \stackrel{r/t}{\to} wde$	$wnt \to wct$
$wct \stackrel{r/}{\to} bct$	$wct \to wnt$	$bct \stackrel{/t}{\to} wne$

We explain how the transitions given above can be translated into our notation. The first component on the transition label is the input and the second is the output. We consider the four kind of transitions:

- $x \rightarrow y$ corresponds to the transitions $\langle x, o\rangle \xrightarrow{\tau} \langle y, o\rangle$.
- $x \xrightarrow{/v} y$ corresponds to transitions $\langle x, o\rangle \xrightarrow{\tau} \langle y, o \cup \{v\}\rangle$.
- $x \xrightarrow{u/} y$ corresponds to transitions $\langle x, o\rangle \xrightarrow{u?} \langle y, o\rangle$.
- $x \xrightarrow{u/v} y$ corresponds to transitions $\langle x, o\rangle \xrightarrow{u?} \langle y, o \cup \{v\}\rangle$.

Notice that each entry in the table actually corresponds to a group of transitions which only differ from each other by the value of the second component.

Let Q be the process with $\langle wnt, \emptyset\rangle$ as the initial state and the transitions shown above. Let P be the process with $\langle wne, \emptyset\rangle$ as the initial state and the same transitions as Q. The network grammar generating a token-ring of arbitrary size is similar to that of Example 6.4.1.

Let S be the set of states in a *basic* process of the token ring. Let `t` be the subset of states which have the token. Let `not-t` be the set $S \setminus$`t`. The automaton $\mathcal{D}$ is the same as the automaton in Figure 6.3 with `t` substituted for *cs* and `not-t` substituted for *nc*. The automaton accepts strings $S^{\star}$ such that the number of processes with the tokens is exactly one. Let h be the abstraction function induced by the automaton. The initial association is:

$$\begin{aligned} set(Q) &= \{Q\} \\ set(P) &= \{P\} \\ set(A) &= \{P \| P\} \\ set(\mathcal{S}) &= \{Q \| P \| P\} \end{aligned}$$

Unfortunately, the corresponding representatives did not satisfy the monotonicity condition. Therefore, we unfolded the initial association to get the following association:

$$\begin{aligned} set(A) &= \{P \| P \| P, P \| P\} \\ set(\mathcal{S}) &= \{Q \| P \| P \| P, Q \| P \| P\} \end{aligned}$$

Notice that unfolding does not change the associations for the terminals. If we choose the representatives as

$$\begin{aligned} rep(P) &= P \\ rep(Q) &= Q \\ rep(A) &= P\|P\|P \\ rep(\mathcal{S}) &= Q\|P\|P\|P \end{aligned}$$

then the monotonicty condition holds, i.e.,

$$\begin{aligned} h(rep(A)) &\succeq h(h(rep(P)\|h(rep(P))) \\ h(rep(A)) &\succeq h(h(rep(A))\|h(rep(P))) \\ h(rep(\mathcal{S})) &\succeq h(h(rep(Q))\|h(rep(A))) \end{aligned}$$

By Theorem 6.3.1 we conclude that $rep(\mathcal{S})$ simulates all the LTSs generated by the grammar G. Notice that if $rep(\mathcal{S})$ satisfies the property $\mathbf{AG}\,\mathcal{D}$, then Theorem 6.2.1 implies that every LTS generated by the grammar G satisfies $\mathbf{AG}\,\mathcal{D}$. Using our system we established that $rep(\mathcal{S})$ is a model for $\mathbf{AG}\,\mathcal{D}$.

6.6.2 Parity tree

We consider a network of binary trees, in which each leaf has a bit value. We describe an algorithm that computes the parity of the values at the leaves. The algorithm is taken from [69]. A context-free grammar G generating a binary tree is given below, where `root, inter and leaf` are terminals (basic processes) and $\mathcal{S}$ and SUB are nonterminals. The precise definition of the composition operator is given later.

$$\boxed{\begin{array}{l} \mathcal{S} \rightarrow \texttt{root}\|\mathrm{SUB}\|\mathrm{SUB} \\ \mathrm{SUB} \rightarrow \texttt{inter}\|\mathrm{SUB}\|\mathrm{SUB} \\ \mathrm{SUB} \rightarrow \texttt{inter}\|\texttt{leaf}\|\texttt{leaf} \end{array}}$$

An intuitive description of the algorithm follows. The `root` process initiates a wave by sending the *readydown* signal to its children. Every internal node that gets the signal sends it further to its children. When the signal *readydown* reaches a leaf process, the leaf sends the *readyup* signal and its *value* to its parent. An internal node that receives the *readyup* and *value* from both its children, sends the *readyup* signal and the `xor`($\oplus$) of the values received from the children to its parent. When the *readyup* signal reaches the root, one wave of the computation is terminated and the root can initiate another wave. The semantics of the composition used in the grammar G should be clear from Figure 6.6. For example, the inputs *readyp_l* and *value_l* of an internal node are identified with the outputs *readyup* and *value* of its left child.

Next, we describe the various signals in detail. First we describe the

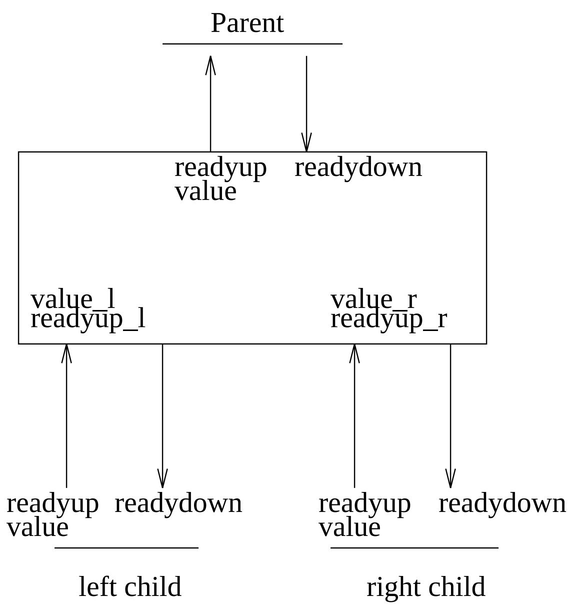

Figure 6.6: Internal node of the tree

process `inter`. The process `inter` is the process in the internal node of the tree. The various variables for the process are shown in the table:

state vars	output vars	input vars
$root\_or\_leaf$	$readydown$	$readydown$
$readydown$	$readyup$	$readyup\_l$
$readyup\_l$	$value$	$readyup\_r$
$readyup\_r$		$value\_l$
$value$		$value\_r$
$readyup$		

The following equations are invariants for the state variables:

$$\begin{aligned} root\_or\_leaf &= 0 \\ readyup &= readyup\_l \wedge readyup\_r \end{aligned}$$

The output variables have the same value in each state as the corresponding state variable, e.g., the output variable $readydown$ has the same value as the state variable $readydown$. The equations given below show how the input variables affect the state variables. The primed variables on the left hand side refer to the next state variables and the right hand side refers to the input variables.

$$\begin{aligned} readydown' &= readydown \\ readyup\_l' &= readyup\_l \\ readyup\_r' &= readyup\_r \\ value' &= (readyup\_l \wedge value\_l) \oplus (readyup\_r \wedge value\_r) \end{aligned}$$

Since the `root` process does not have a parent, it does not have the input variable $readydown$. The invariant $root\_or\_leaf = 1$ is maintained for the root and the leaf process. Since the `leaf` process does not have a child, the output variable $readydown$ is absent. The `leaf` process has only one input variable $readydown$ and the following equation between the next state variables and input variables is maintained:

$$readyup' = readydown$$

For each `leaf` process the assignment for the state variable $value$ is decided non-deterministically in the initial state and then kept the same throughout the computation.

A state in the basic processes `(root,leaf,inter)` is a specific assignment to the state variables. We call this state set S. Notice that the state set $S \cong \{0,1\}^6$ because there are 6 state variables.

The automata we describe accepts strings from $S^\star$. Let $value_1, \cdots, value_n$ be the values in the n leaves. Let $value$ be the value calculated at the `root`. Since at the end of the computation the `root` process should have the parity of the bits $value_i$ ($1 \leq i \leq n$), the following equation should hold at the end of the computation:

$$value \oplus \bigoplus_{i=1}^{n} value_i \quad = \quad 0.$$

Let `p` be defined by the following equation:

$$\mathtt{p} \quad = \quad \{s \in S \mid s \text{ satisfies } root\_or\_leaf \wedge value\}.$$

Let $\mathtt{not(p)} = S - \mathtt{p}$. The automaton given in Figure 6.7 accepts the strings in $S^\star$ which satisfy the equation given above. Since $root\_or\_leaf = 0$ for internal nodes, the automaton essentially ignores the values at the internal nodes. We call this automaton **parity**. We also want to assert

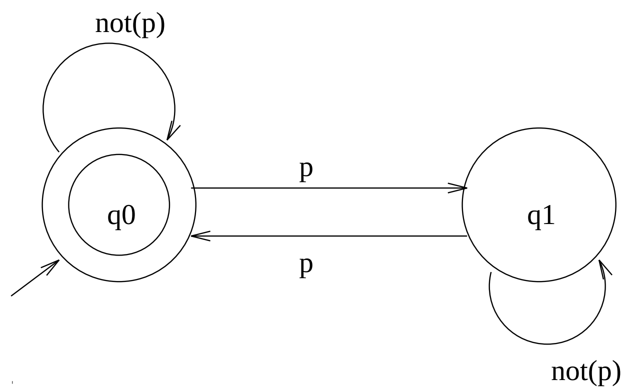

Figure 6.7: Automaton for parity

that everybody is finished with their computation. This is signaled by the fact that $readyup = 1$ for each process. The automaton given in Figure 6.8 accepts strings in $S^\star$ iff $readyup = 1$ in each state. i.e. all processes have finished their computation. We call this automaton **finished**. The automata **parity** and **finished** are our atomic formulas.

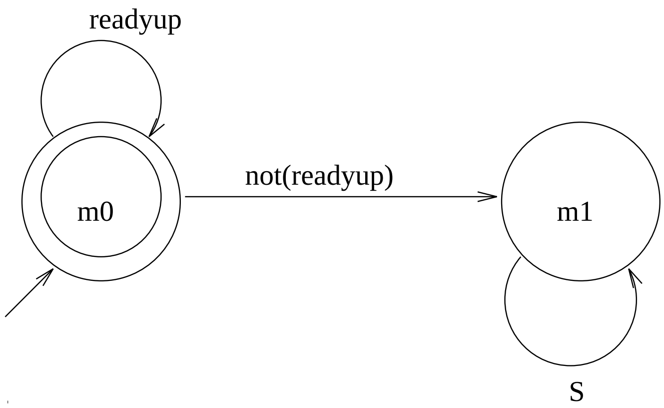

Figure 6.8: Automaton for ready

We want to verify that if the computation is finished then the parity at the root is correct. Hence, we want to check that the initial state satisfies the property $\mathbf{AG}(\mathbf{finished} \rightarrow \mathbf{parity})$. Let h be the abstraction function induced by the atomic formulas (see Section 6.2 for the definition of h). The initial association is:

$$\begin{aligned} set_0(\mathrm{SUB}) &= \{\mathtt{inter} \| \mathtt{leaf} \| \mathtt{leaf}\} \\ set_0(\mathcal{S}) &= root \| set_0(\mathrm{SUB}) \| set_0(\mathrm{SUB}) \end{aligned}$$

The association corresponding to the terminals is simply the process associated with the terminal. The association for $\mathcal{S}$ can be derived from $set(\mathrm{SUB})$. The representatives corresponding to the initial association did not satisfy the monotonicty condition. Unfolding the initial association we get:

$$\begin{aligned} I &= \mathtt{inter} \| \mathtt{leaf} \| \mathtt{leaf} \\ I_1 &= \mathtt{inter} \| I \| I \\ set_1(\mathrm{SUB}) &= \{I_1\} \cup set_0(\mathrm{SUB}) \\ set_1(\mathcal{S}) &= \{\mathtt{root} \| I_1 \| I_1\} \cup set_0(\mathcal{S}) \end{aligned}$$

Now we could find representatives that did satisfy the monotonicity condition. Using Theorem 6.3.1 we can conclude that $H = h(rep(\mathcal{S}))$ simulates all the networks generated by the context free grammar G.

We checked that $H, s_0 \models \mathbf{AG}(\mathbf{finished} \rightarrow \mathbf{parity})$ (s_0 is the initial state of the process H). Theorem 6.2.1 implies that that every network derived by G has the desired property, i.e., when the computation is finished the root process has the correct property.

6.7 Related Work

The first paper to investigate infinite family of finite-state systems was [10]. They considered the problem of verifying a family of token rings. In order to verify the entire family, the authors established a *bisimulation* relation between a 2-process token ring and an n-process token ring for any $n \geq 2$. It follows that the 2-process token ring and the n-process token ring satisfy exactly the same temporal formulas. The drawback of their technique is that the bisimulation relation had to be constructed manually.

Induction at the process level has also been used to solve problems of this nature by two research groups [30, 34]. They prove that for rings composed of certain kinds of processes there exists a k such that the correctness of the ring with k processes implies the correctness of rings of arbitrary size. In [72] an alternative method is proposed for checking properties of parametrized systems. In this framework there are two types of processes: G_s (slave processes) and G_c (control processes). There can be many slave processes with type G_s, but only one control process with type G_c. The slave processes G_s can only communicate with the control process G_c.

In [53, 64] context-free network grammars are used to generate infinite families of processes with multiple repetitive components. Using the structure of the grammar they generate an invariant I and then check that I is *equivalent* to every process in the language of the grammar. If the method succeeds, then the property can be checked on the invariant I. The requirement for equivalence between all systems in F is too strong in practice and severely limits the usefulness of the method. In this chapter, *equivalence* was replaced with a suitable *preorder* while still using network grammars.

Chapter 7

Conclusion

Chapter 2 showed the importance of the propositional μ-calculus by giving translations of various graph-based verification algorithms into the μ-calculus. An OBDD based algorithm for μ-calculus model checking which has proved to be extremely efficient in practice was also presented. Finally, the best known algorithm for evaluating μ-calculus formulas was described. However, there is still much work to be done in each of these areas.

Although OBDDs do not reduce the worst-case complexity of the model checking problem for the μ-calculus, their use in model checking has had an enormous effect on formal verification. Before the use of OBDDs, it was only possible to verify models with at most 10^6 states [17]. By using the OBDD techniques described in this paper, in practice, it is now possible to verify examples with up to 10^{120} states and several hundred state variables [12]. However, there is no theoretical framework which explains when OBDDs will work well in practice. The algorithm given in Chapater 2 does not depend on the data structure used to represent boolean functions, so it should be possible to use any better data structures that may be discovered.

In addition to the verification problems we have considered, there are other graph theoretic problems that can be encoded in the μ-calculus. An important question is how useful these OBDD and fixpoint techniques are for problems like finding minimum spanning trees, determining graph isomorphism, etc. For example, let $E(u, v)$ be the edge relation for a directed graph and let each vertex v be a state encoded by an assignment $\vec{v}$ to the boolean variables $\vec{x} = x_1, \ldots, x_k$. The formula

$$\phi(\vec{x}) \equiv \mu R.\vec{x} \vee \langle a \rangle R$$

computes the set of states reachable from the state encoded by the assignment to $\vec{x}$, where the interpretation for the program letter a is the edge relation E. Then the graph satisfies the formula

$$[\vec{u} \rightarrow \phi(\vec{v})] \wedge [\vec{v} \rightarrow \phi(\vec{u})]$$

if and only if the two vertices u and v are in the same strongly connected component. In general, the graph is strongly connected if and only if it satisfies the formula

$$\forall \vec{x}.\phi(\vec{x}).$$

Although strictly speaking this is not a μ-calculus formula according to the syntax presented earlier, recall that we allow quantification over boolean variables in our translation of the μ-calculus into OBDDs.

Efficient evaluation algorithms, which exploit monotonicity properties when evaluating fixpoints were also described in Chapter 2. However, these algorithms remain exponential in the alternation depth. I conjecture that there is no polynomial-time algorithm for determining if a state satisfies a given formula. Consider an algorithm that computes least fixpoints by iterating, and that guesses greatest fixpoints. The guess for a greatest fixpoint can be easily checked to see that it really is a fixpoint. Furthermore, while we cannot verify that it is the greatest fixpoint, we know that the greatest fixpoint must contain any verified guess. Then by monotonicity, the final value computed by this nondeterministic algorithm will be a subset of the real interpretation of the formula. The state in question satisfies the formula if and only if it is in the set computed by some run of the algorithm. Also note that one can negate formulas, so the complexity of determining if a state satisfies a formula is the same as the complexity of determining if a state does not satisfy the formula. Thus, the problem is in the intersection of NP and co-NP. This suggests that the conjecture will be very difficult to prove.

In Chapter 3 various techniques for exploiting symmetry during model checking were discussed. There are a number of directions for future research. It would be insightful to have tight lower bounds on the size of the OBDDs needed for the orbit relation for symmetry groups occurring in practice. This type of information would be useful in determining if it is feasible to construct the quotient model directly using the orbit relation or whether it is necessary to develop special techniques for mapping states onto representatives. An automatic procedure for identifying symmetries in circuits would definitely be useful. Techniques based on the Walsh transform have been tried for this purpose in the past [40]. However, I suspect that this will always be a hard problem and that some information from the designer of the circuit will usually be required. It will also be very insightful to try other hardware examples with more complicated topologies in addition to cache coherency protocols.

Chapter 4 investigated the complexity of various problems associated with exploiting symmetry in model checking. We also provided ways of deriving symmetry for shared variable concurrent programs. An important research problem will be to take some existing hardware description languages and derive symmetry information statically from the system descriptions written in that language. Perhaps, some of the ideas presented in section 4.2 could be used. This chapter also made the connection between exploiting symmetry in model checking and computational group theory. An important research direction will be to use some of the powerful techniques available in the computational group theory literature in the model checking domain.

Chapter 5 described techniques to combine partial-order and symmetry reduction methods. In the future, I would like to implement these methods on some existing verification tools and try some examples. Another interesting problem is to derive symmetry and independence information from the description of the LTS being verified. Presently, most verification systems rely on the user to provide this information. I would also like to investigate whether some other reduction techniques could be combined using similar ideas.

Chapter 6 described a new technique for reasoning about families of finite-state systems. This work combines network grammars and abstraction with a new way of specifying state properties using regular languages. I have implemented this verification method and used it to check two non-trivial examples. In the future, I intend to apply the method to even more complex families of state-transition systems. There are several directions for future research. The context-free network grammars can be replaced by context-sensitive grammars. Context-sensitive grammars can generate networks like square grids and complete binary tree which cannot be generated by the context-free grammars. The specification language can be strengthened by replacing regular languages by more expressive formalisms (like ω-regular languages). An interesting extension would be to add fairness to the models under consideration.

Bibliography

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algprithms*. Addison-Wesley, 1974.

[2] H. R. Andersen. Model checking and boolean graphs. In B. Krieg-Bruckner, editor, *Proceedings of the Fourth European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*. Springer-Verlag, February 1992.

[3] H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126:3–30, 1994.

[4] K. Apt and D. Kozen. Limits for automatic verification of finite-state systems. *IPL*, 15:307–309, 1986.

[5] L. Babai and E.M. Luks. Canonical labeling of graphs. In *Proceedings of the 15th ACM STOC*, 1983.

[6] B.J. Birch, R.G Burns, S.O. Macdonald, and P.M. Neumann. On the orbit sizes of permutation groups containing elements separating finite subsets. *Bull. Australian Mathematical Society*, pages 7–10, 1976.

[7] G. V. Bochmann and D. K. Probst, editors. *Proceedings of the Fourth Workshop on Computer-Aided Verification*, volume 663 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1992.

[8] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, December 1986.

[9] M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.

[10] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite-state processes. *Information and Computation*, 81(1):13–31, April 1989.

[11] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[12] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, April 1994.

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1990.

[14] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[15] J.P. Burgess. Basic tense logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosphical Logic*, pages 89–134. D.Reidel, 1984.

[16] L. Claesen, editor. *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, April 1993.

[17] 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, April 1986.

[18] 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. In Claesen [16].

[19] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1992.

[20] E.M. Clarke, T.Filkorn, and S.Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis [24].

[21] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–747, September 1990.

[22] R. Cleaveland, M. Klein, and B. Steffen. Faster model checking for the modal mu-calculus. In Bochmann and Probst [7].

[23] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(2):121–147, April 1993.

[24] C. Courcoubetis, editor. *Proceedings of the Fifth Workshop on Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1993.

[25] D.R. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving ACTL*,ECTL*, and CTL*. In *IFIP working conference and Programming Concepts, Methods and Calculi (PROCOMET'94)*, San Miniato, Italy, June 1994.

[26] E.W. Dijkstra. Invariance and non-determinacy. In C.A.R. Hoare and J.C. Sheperdson, editors, *Mathematical Logic and Programming Languages*. Prentice Hall, 1985.

[27] D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings*, Part E 133(5), 1986.

[28] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1986.

[29] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. In Courcoubetis [24].

[30] E.A. Emerson and Kedar S. Namjoshi. Reasoning about rings. In *Proceedings of the Twenty Second Annual ACM Symposium on Principles of Programming Languages*, January 1995.

[31] E. Felt, G. York, R. Brayton, and A. Sangiovanni Vincentelli. Dynamic variable reordering for bdd minimization. In *Proceedings of the EuroDAC*, pages 130–135, September 1993.

[32] M.L. Furst, J.E. Hopcroft, and E. Luks. Polynomial-time algorithms for permutations groups. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, 1980.

[33] M.R. Garey and D.S Johnson. *Computers and Intractibility*. W.H. Freeman and Company, 1979.

[34] S.M. German and A.P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39:675–735, 1992.

[35] Rob Gerth, Ruurd Kuiper, Doron Peled, and Wojciech Penczek. A partial order approach to branching time logic model checking. In *Third Israel Symposium on Theory on Computing and Systems*, pages 130–139, Tel Aviv, Israel, 1995. IEEE.

[36] P. Godefroid. Using partial orders to improve automatic verification methods. In Kurshan and Clarke [46].

[37] C.M. Hoffman. *Group Theoretic Algorithms and Graph Isomorphism*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[38] P. Huber, A.M. Jensen, L.O. Jepsen, and K. Jensen. Towards reachability trees for high-level petri nets. *In G. Rozenberg, editor, Advances on Petri Nets*, pages 215–233, 1984.

[39] T.W. Hungerford. *Algebra*. Springer-Verlag, 1980.

[40] S. L. Hurst, D. M. Miller, and J. C. Muzio. *Spectral Techniques in Digital Logic*. Academic Press, Inc., 1985.

[41] IEEE Computer Society. *IEEE Standard for Futurebus+—Logical Protocol Specification*, March 1992. IEEE Standard 896.1–1991.

[42] C.W. Ip and D. Dill. Better verification through symmetry. In Claesen [16].

[43] N. Jacobson. *Basic Algebra*. W.H. Freeman and Company, 1985.

[44] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, December 1983.

[45] R. P. Kurshan. Testing containment of ω-regular languages. Technical Report 1121-861010-33-TM, Bell Laboratories, 1986.

[46] R. P. Kurshan and E. M. Clarke, editors. *Proceedings of the 1990 Workshop on Computer-Aided Verification*. Springer-Verlag, June 1990.

[47] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, August 1989.

[48] K. G. Larsen. Efficient local correctness checking. In Bochmann and Probst [7].

[49] B. Lin and A. R. Newton. Efficient symbolic manipulation of equvialence relations and classes. In *Proceedings of the 1991 International Workshop on Formal Methods in VLSI Design*, January 1991.

[50] D. Long, A. Browne, E. Clarke, S. Jha, and W. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In D. Dill, editor, *Proceedings of the 1995 Workshop on Computer-Aided Verification*, pages 338–350. Springer-Verlag, June 1994.

[51] E.M. Luks. Permutation groups and polynomial-time computation. In *Workshop on Groups and Computation*, volume 11 of *Dimacs*. American Mathematical Society, October 1991.

[52] A. Mader. Tableau recycling. In Bochmann and Probst [7].

[53] R. Marelly and O. Grumberg. GORMEL—Grammar ORiented ModEL checker. Technical Report 697, The Technion, October 1991.

[54] A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets*, number 225 in LNCS, pages 279–324, Bad Honnef, Germany, 1986. Springer-Verlag.

[55] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.

[56] K. L. McMillan and J. Schwalbe. Formal verification of the Gigamax cache consistency protocol. In N. Suzuki, editor, *Shared Memory Multiprocessing.* MIT Press, 1992.

[57] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the Second International Joint Conference on Artificial Intelligence*, September 1971.

[58] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[59] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 6 (16):973–989, 1987.

[60] Doron Peled. All from one, one from all: on model checking using representatives. In *5th International Conference on Computer Aided Verification, Greece*, number 697 in LNCS, pages 409–423, Elounda Crete, Greece, 1993. Springer-Verlag.

[61] Doron Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In *Partial Orders Methods*

in Verification, DIMACS, Princeton, NJ, USA, 1996. American Mathematical Society.

[62] A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science*, 1977.

[63] R.L. Rudell. Dynamic variable reordering for ordered binary decision diagrams. In *Proceedings of the IEEE ICCAD*, pages 42–47, November 1993.

[64] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In Sifakis [65].

[65] J. Sifakis, editor. *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[66] P.H. Starke. Reachability analysis of petri nets using symmetries. *Syst. Anal. Model. Simul.*, 8(4/5):293–303, 1991.

[67] C. Stirling and D. J. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, October 1991.

[68] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[69] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.

[70] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the Tenth International Conference on Application and Theory of Petri Nets*, 1989.

[71] A. Valmari. A stubborn attack on the state explosion problem. In Kurshan and Clarke [46].

[72] I. Vernier. Parameterized evaluation of CTL-X formulae. In *Workshop accompanying the International Conference on Temporal Logic (ICTL'94)*, 1994.

[73] G. Winskel. Model checking in the modal ν-calculus. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, 1989.

[74] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In Sifakis [65].