FORMAL ANALYSIS OF SYNCHRONOUS CIRCUITS

by

Thomas Robert Shiple

Memorandum No. UCB/ERL M96/76

4 December 1996

FORMAL ANALYSIS OF SYNCHRONOUS CIRCUITS

by

Thomas Robert Shiple

Memorandum No. UCB/ERL M96/76

4 December 1996

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

Formal Analysis of Synchronous Circuits

by

Thomas Robert Shiple

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

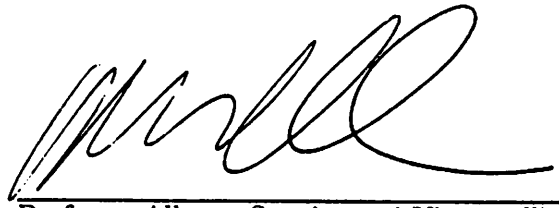
University of California at Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

This dissertation addresses three separate, but related problems concerning the formal analysis of synchronous circuits and their associated finite state machines. The first problem is the logical analysis of synchronous circuits containing combinational cycles. The presence of such cycles can cause unstable behavior at the outputs of a circuit, but this is not necessarily always the case. This work determines when cycles are harmless, and when they are not. In particular, three classes of circuits are defined that tradeoff time to decide the class, with the permissiveness of the class. For each class, the complexity of the corresponding decision problem is proven and a procedure to decide the class is given. In addition, if a circuit is determined to be within a given class, then a new circuit can be generated with the same input/output behavior, but without combinational cycles. This is an important utility, as many CAD tools do not accept circuits with combinational cycles.

The second problem that is addressed is the CTL model checking of interacting FSMs. A state equivalence is presented that is defined with respect to a given CTL formula. Since it does not attempt to preserve all CTL formulas, like bisimulation does, we can expect to compute coarser equivalences. This equivalence is used to manage the size of the transition relations encountered when model checking a system of interacting FSMs. Specifically, the equivalence is used to reduce the size of each component FSM, so that their product will be smaller. We show how to apply the method, whether an explicit representation is used for the FSMs, or BDDs are used. Also, we show that in some cases this approach can detect if a formula passes or fails, without composing all the component machines. The method is exact and completely automatic, and handles full CTL.

These two problems are PSPACE-hard (in the number of flip-flops) to decide; approximate methods may be useful to find a solution in affordable CPU time. To demonstrate the use of approximate methods in logical analysis, we address the state reachability problem in FSMs, which is the problem of determining if one set of states can reach another. State reachability has broad applications in formal verification, synthesis, and testing of synchronous circuits. This work attacks this problem by making a series of under- and over-approximations to the state transition graph, using the over-approximations to guide the search in the under-approximations for a potential path from one state set to the other. Central to this method is an algorithm to approximate a Boolean function by another function having a smaller BDD.



Professor Alberto Sangiovanni-Vincentelli
Dissertation Committee Chair

...to Suzanne

Contents

List of Figures

List of Tables

Acknowledgements

I have been privileged to work with Professors Alberto Sangiovanni-Vincentelli and Bob Brayton. Their integrity, love of learning, quality of work, and breadth of knowledge have inspired me, and will have a lasting influence on me.

I would like to express gratitude to my research collaborators. Without their willingness to listen to and improve upon my ideas, I do not think I would have achieved as much. The work on combinational cycles began at Digital's Paris Research Laboratory and INRIA with Hervé Touati and Gérard Berry, and later included Vigyan Singhal at Berkeley. The research on formula-dependent equivalences for CTL, which stemmed from earlier work with Macs Chiodo, was done jointly with Adnan Aziz and Vigyan Singhal. The work on reachability for large FSMs was undertaken with Rajeev Ranjan. I would like to give special thanks to Adnan, Vigyan and Rajeev, for I have learned a lot from them. Finally, it has been a special pleasure being part of the hardworking and synergistic VIS team: Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev Ranjan, Shaker Sarwary, Gitanjali Swamy, and Tiziano Villa.

I want to thank the senior students who helped me to get started at Berkeley and who exhibited qualities to emulate: Wendell Baker, Timothy Kam, Luciano Lavagno, Sharad Malik, Rajeev Murgai, Alex Saldanha, Hamid Savoj, Ellen Sentovich, Narendra Shenoy, K. J. Singh, Paul Stephan, Hervé Touati, Tiziano Villa, and Huey-Yih Wang.

One of my most intense work periods at Berkeley was preparing for the Preliminary Examination. I would not have succeeded without the support of my fellow study group members: Ramin Hojati, Sriram Krishnan and Henry Sheng. In particular, Ramin impressed upon me the meaning and importance of rigorous mathematic proof.

The staff supporting the CAD group has been uniformly helpful and amiable. Thanks go to Kia Cooper, Ruth Gjerde, Brad Krebs, Elise Mills and Flora Oviedo.

I would like to acknowledge the financial support of the Semiconductor Research Corporation. Not only was most of this work conducted under the auspices of SRC research funding (DC-324), but I also had the honor of being an SRC Graduate Fellow for the last four years. Special thanks to Ginny Poe at SRC headquarters for her friendly attitude and for answers to all my questions.

Throughout my graduate career, I have enjoyed the love and support from not only my own family, but also from Suzanne's family. I always felt that they were there

with me, and that they share in my academic degrees. Thanks also to Caleb, my son, for giving his birth date as a fixed deadline by which to finish this dissertation. Finally, I owe everything to my wife, Suzanne. Her advice, understanding and love have been the pillars upon which I have drawn my strength. Thanks!

Chapter 1

Introduction

This dissertation addresses three separate, but related problems concerning the formal analysis of synchronous circuits and their associated finite state machines (FSMs). Synchronous circuits play a central role in the design of digital systems, because their uniform clocking methodology allows circuits to be composed with predictable results. Here we provide algorithms for solving three decision problems relating to synchronous circuits and FSMs.

Chapter 2 deals with the formal analysis of combinational cycles in synchronous circuits. Combinational cycles can lead to unstable and unpredictable behavior at the outputs of a circuit, but this need not always be the case. In fact, some circuits are purposely designed with combinational cycles to affect a more efficient implementation, to create state-holding devices, or to provide a more lucid, symmetric description of a function. In all these cases, the intention is that such cycles do not lead to unstable or unpredictable behavior.

Our contribution is to provide algorithms to determine if the combinational cycles of a circuit are harmful. Others have touched on this problem, but have not addressed it with sufficient rigor. In particular, we define three formal classes of well-behaved circuits, that tradeoff time to decide the class, with the permissiveness of the class. The definition of these classes is grounded in the up-bounded inertial delay model, and the GMW analysis of synchronous circuits of Brzozowski and Seger [3].

For each class, we prove the complexity of deciding the class, and give an algorithm that provably decides the class exactly. If a circuit does not fall within a particular class, then we give a method to generate an error trace demonstrating a sequence of inputs that

leads to unstable behavior at an output. This is useful to debug the operation of a circuit. On the other hand, if a circuit falls within a class, then we can produce a new circuit having the same input/output behavior, without combinational cycles. Given this, an FSM (at the abstract level) can be easily derived. This is an important utility because many CAD algorithms, including our own algorithms of Chapters 3 and 4, only accept inputs without combinational cycles. Hence, the algorithms of Chapter 2 can serve as preprocessors.

Chapter 3 presents a method for CTL model checking systems of interacting FSMs. CTL (for computation tree logic) is a language for specifying correctness properties of FSMs, and model checking is the problem of determining if a given FSM satisfies a particular CTL formula. CTL model checking has emerged as one of the main automatic methods for formal verification, or property checking.

CTL model checking a system of interacting FSMs is PSPACE-complete in the number of FSMs. Thus, heuristics are needed that always give the correct answer, but which are as fast as possible. Our approach is to compute an equivalence relation with respect to a given CTL formula, on the states of each component FSM. Because this relation is computed with respect to a single formula, it is more coarse (meaning the equivalence classes are larger), and hence more effective, than equivalences that preserve all of CTL, such as bisimulation.

The formula-dependent equivalence is used to make each component machine smaller before composing it with other components. If an explicit data representation is used for FSMs, then the quotient machine, with respect to the equivalence, is used to yield a smaller machine. If an implicit data representation is used, such as BDDs, then this equivalence is used to define a range of permissible substitutes for the component, among which we want to use the one with the smallest BDD.

After each component has been reduced, their product is formed to yield a single, more tractable, FSM. Standard CTL model checking is applied to this machine to yield the final answer. In some cases, our algorithm can determine the truth or falsity of a formula without building the reduced product FSM. Our method can be applied to any formula of CTL, and it is completely automatic.

The problems of state reachability and CTL model checking are PSPACE-hard (in the number of flip-flops) to decide, and thus approximate methods to solve them would be useful. To demonstrate the general idea of approximate methods, Chapter 4 addresses the similar problem of *state reachability*, which is the problem of deciding whether a set of initial

states I can reach a set of final states F in an FSM M. This problem has applications in the formal verification, synthesis, and testing of synchronous circuits, so an efficient solution would benefit these areas of computer-aided design (CAD) of digital systems. One approach to this problem is to calculate the set R of *all* states reachable from I, and then test if F and R intersect. This may involve more work than is necessary to solve the more specific question of state reachability. Our goal is to increase the size of FSMs that can be analyzed.

Our approach is to construct a series of under- and over-approximations of the state transition graph G of M. If a path from I to F exists in an under-approximation, then the path also exists in G. On the other hand, every path from I to F in G must exist in an over-approximation. Our strategy is to use the set of paths from I to F in an over-approximation to guide the search for such a path in an under-approximation.

Following the lead of other researchers over the last decade, we use binary decision diagrams (BDDs) to represent FSMs. However, we have defined a new optimization problem, the *BDD approximation problem*, whose efficient solution is crucial to our algorithm for state reachability. Given a Boolean function f, the goal is to find another function g, such that $g \supseteq f$ (i.e., the onset of g contains the onset of f) and g has a small BDD. The closer g is to f, and the smaller the BDD for g, the better is the approximation. This problem was independently formulated by Ravi and Somenzi [4]. We develop some theory related to the problem, and present a heuristic solution.

Each of the main chapters of this dissertation can be read independently of the others after first reviewing some common terminology in Section 1.1. Each chapter has its own extensive introduction, discussion of related work, and summary. Where appropriate, lengthy proofs have been relegated to a separate section within each chapter.

The appendix describes the architecture of the software system VIS (Verification Interacting with Synthesis). This system, developed jointly at the University of California, Berkeley, and the University of Colorado, Boulder, provides a framework for implementing algorithms related to the verification and synthesis of synchronous circuits. In particular, the algorithms described in this dissertation could be implemented within VIS.

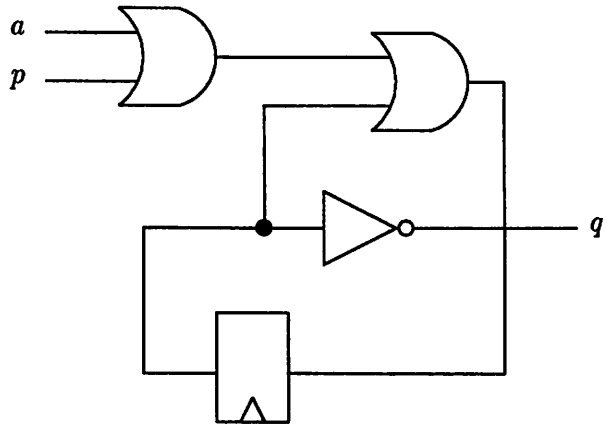


Figure 1.1: A circuit with two inputs and one output.

1.1 Terminology

1.1.1 Synchronous circuits

We are interested in the study of circuits at the "gate" level. A *logic gate* is a device with k binary-valued inputs and one binary-valued output, that computes a Boolean function $f : \mathbb{B}^k \to \mathbb{B}$ ($\mathbb{B}$ denotes the set $\{0, 1\}$). A logic gate has an associated delay; this will be discussed further in Chapter 2. A *flip-flop* is a device with a binary data input, a binary clock input, and a data output. On the rising edge of the clock, the data input value is copied to the data output.

A *circuit* is an arbitrary interconnection of logic gates and flip-flops. A *synchronous circuit* is a circuit whose flip-flops are enabled by the same clock, and whose inputs change, and outputs are sampled, on the rising edge of the clock. The part of the circuit consisting of just the logic gates is called the *combinational part.* A *combinational cycle* in a circuit is a directed cycle of logic gates, when the circuit is viewed as a directed graph with vertices corresponding to the logic gates and flip-flops.

Figure 1.1 shows a circuit with two inputs, a and p, one output q, and one flip-flop.

1.1.2 Finite state machines

A finite state machine (FSM) is an abstraction of a synchronous circuit. In Chapter 2, we use FSMs as a tool in deciding output-stability. In Chapters 3 and 4, we pose the problems of CTL model checking and state reachability directly on FSMs. There are

various types of FSMs, and various notations to describe them. We cover the major aspects here, and then customize the definitions as needed later.

An FSM consists of the following components.

- A finite set of *states*, S.
- A set of *initial* states, $I \subseteq S$.
- A finite *input alphabet*, Σ_I.
- A finite *output alphabet*, Σ_O.
- An *output relation*, O. For *Moore* machines, O only depends on the state, $O \subseteq S \times \Sigma_O$. For *Mealy* machines, O also depends on the input, $O \subseteq S \times \Sigma_I \times \Sigma_O$. If O can be expressed as a function (i.e., for Moore, $O : S \to \Sigma_O$, and for Mealy, $O : S \times \Sigma \to \Sigma_O$), then the output is *deterministic*, otherwise it is *nondeterministic.*
- A *transition relation*, $T \subseteq S \times \Sigma_I \times S$. If $(x, a, y) \in T$, this means that from state x on input a, the machine can move to y. The notation $x \xrightarrow{a} y$ is shorthand for $(x, a, y) \in T$. If T can be written as a function, then the next state is deterministic, otherwise it is nondeterministic. We require that T is *complete*, meaning that for each $a \in \Sigma_I$ and $x \in S$, there exists at least one $y \in S$ such that $(x, a, y) \in T$. This guarantees that progress is always possible, regardless of the input.

For a synchronous circuit C, an FSM is derived as follows.

- If C has l flip-flops, $S = \mathbb{B}^l$.[1] Flip-flop j has a *present state variable* x_j and a *next state variable* y_j.
- If each flip-flop j has initial value(s) $I_j \subseteq \mathbb{B}$, then I is the Cartesian product, $I = \times_{j=1}^{l} I_j$.
- If C has a set $\mathcal{X}$ of n inputs, then $\Sigma_I = \mathbb{B}^n$.
- If C has a set $\mathcal{O}$ of p outputs, then $\Sigma_O = \mathbb{B}^p$.
- If output j is driven by $\zeta_j : S \times \Sigma_I \to \mathbb{B}$, then $O = \prod_{j=1}^{p} (z_j \equiv \zeta_j)$, where z_j is identified with output j.

[1] The case where state is stored by the logic gates is handled specially in Chapter 2.

- If the data input of flip-flop j is driven by $\delta_j : S \times \Sigma_I \to \mathbb{B}$, then $T = \prod_{j=1}^{l}(y_j \equiv \delta_j)$. T is complete, since a circuit must have a reaction for each input.

Figure 1.2 shows the FSM M_2 corresponding to the circuit in Figure 1.1. State $1'$ corresponds to flip-flop value 0, and state $2'$ to value 1. Throughout the diagram, Boolean equations are used to refer to subsets of the Boolean space. Hence, the label $\overline{q}$ refers to the output $\{0\}$, and the edge labels $\overline{a}\,\overline{p}$, $a + p$, and T refer to $\{00\}, \{01, 10, 11\}$, and $\{00, 01, 10, 11\}$, respectively.

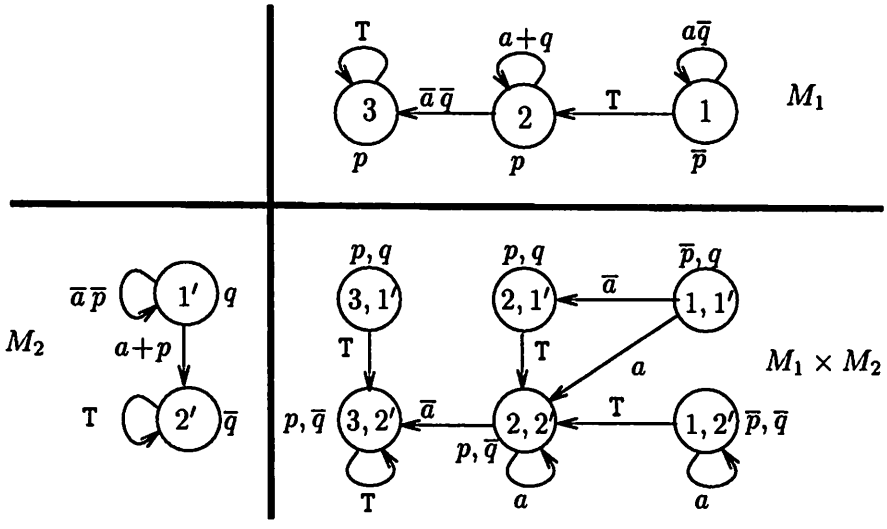


Figure 1.2: Example of FSM composition: p is the output of M_1, q is the output of M_2, and a is an external input. $a\overline{q}$ is shorthand for the input assignment 10. The union of $a\overline{q}$, $\overline{a}q$ and aq is denoted by $a+q$. The tautology is denoted by T.

The *state transition graph*, G, of an FSM is a directed graph with vertex set S and edges $\{(x, y) | \exists a \in \Sigma_I$ such that $(x, a, y) \in T\}$. A state x is *reachable* if there exists a path in G from an initial state to x. The set of *reachable states* is all states that are reachable.

Suppose we wish to *compose* two Moore FSMs, M_1 and M_2, whose outputs are disjoint, and where some outputs of M_1 are connected to inputs of M_2, and vice versa.[2] This creates a new Moore FSM M where

- $S = S_1 \times S_2$
- $I = I_1 \times I_2$

[2] If M_1 and M_2 are both Mealy machines, this can lead to combinational cycles, addressed in Chapter 2.

- $\mathcal{X} = (\mathcal{X}_1 \cup \mathcal{X}_2) \setminus (\mathcal{X}_1 \cap \mathcal{O}_2) \cup (\mathcal{X}_2 \cap \mathcal{O}_1)$ (the original inputs, minus those those driven by one of the components)
- $\Sigma_I = \mathbb{B}^{|\mathcal{X}|}$
- $\mathcal{O} = \mathcal{O}_1 \cup \mathcal{O}_2$
- $\Sigma_O = \mathbb{B}^{|\mathcal{O}|}$
- $O = O_1 \times O_2$
- $T \subseteq S \times \Sigma_I \times S$ where $((x,s),(a,c),(y,t)) \in T$ iff $\exists b \in \mathcal{O}_2$ s.t. $(x, a{\cdot}b, y) \in T_1$ and $(s,b) \in O_2$, and $\exists d \in \mathcal{O}_1$ s.t. $(s, c{\cdot}d, t) \in T_2$ and $(x,d) \in O_1$.

Figure 1.2 shows the composition of two Moore FSMs, M_1 and M_2. Note that M_1 has a nondeterministic next state on input $a\overline{q}$ from state 1. The sets of inputs and outputs for M_1 are $\{a,q\}$ and $\{p\}$ respectively; and for M_2 are $\{a,p\}$ and $\{q\}$ respectively. For the composition $M_1 \times M_2$, the sets of inputs and outputs are $\{a\}$ and $\{p,q\}$ respectively.

1.1.3 Binary decision diagrams

Suppose we want to represent Boolean functions spanned by the binary variables $x_1, \ldots, x_n$. A binary decision diagram (BDD) is a data structure used to store and manipulate such functions [5]. A BDD is a rooted, directed, acyclic graph. There are two types of vertices. Terminal vertices have no outgoing edges and are labeled by the constants ZERO or ONE. Non-terminal vertices are labeled by a variable x_i, and have a *lo* child and a *hi* child.

The variables of the function to be represented must be ordered. Here, we assume the ordering $x_1 \prec \ldots \prec x_n$. The variable labels of the vertices of any root-to-terminal path must respect this ordering. Two rules are applied to reduce a BDD.

1. If a node exists where the *lo* and *hi* children are the same, then that node is eliminated, and all incoming edges are redirected to the child node.
2. If two nodes exist that have the same variable label, the same *lo* child, and the same *hi* child, then these two nodes are merged into a single node.

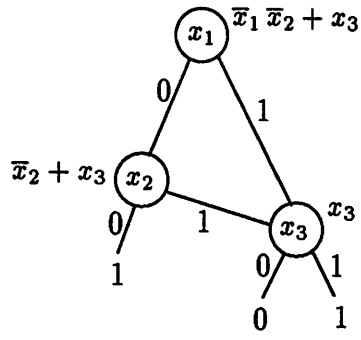


Figure 1.3: A BDD: the function represented by each node is shown. The *lo* child is indicated by a 0 edge, and the *hi* child by a 1 edge.

The function $f(G)$ represented by a BDD node G, with label x_i, is defined inductively as follows:

$$\begin{aligned} f(\text{ONE}) &= 1 \\ f(\text{ZERO}) &= 0 \\ f(G) &= x_i \cdot f(G.hi) + \overline{x}_i \cdot f(G.lo) \end{aligned}$$

For example, the BDD of Figure 1.3 represents the function

$$x_1(x_3) + \overline{x_1}(\overline{x_2} + x_3) = \overline{x_1}\,\overline{x_2} + x_3.$$

For a fixed ordering, BDDs are canonical, in that each Boolean function has a unique BDD representation.

A BDD can be used to represent an arbitrary set by considering the characteristic function of the set. In particular, consider a set S taken from some universe U. Then the *characteristic function*, χ_S, is a Boolean function $\chi_S : U \rightarrow \mathbb{B}$, where

$$\chi_S(s) = \begin{cases} 1 & \text{if } s \in S \\ 0 & \text{otherwise.} \end{cases}$$

This gives us the power to use BDDs to represent a set of states of an FSM, or to represent the transition relation, which is nothing more than a set of 3-tuples (x, a, y).

Even though BDDs provide a compact representation for many functions, sometimes their size is too large to manipulate effectively. To combat this problem, in some applications the function being represented can be modified in an attempt to reduce its BDD size. In particular, suppose that a function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ has an associated *care set*

$c : \mathbb{B}^n \rightarrow \mathbb{B}$, such that for any $x \in \mathbb{B}^n$ where $c(x) = 0$, we are free to modify the value of f. This freedom can be used to find a new function g such that $f \cdot c \subseteq g \subseteq f + \overline{c}$. The operators *restrict* and *constrain* can be used for this purpose [6, 7, 8].

Chapter 2

Logical Analysis of Combinational Cycles In Synchronous Circuits

2.1 Introduction

We analyze the logical behavior of synchronous circuits described at the gate and flip-flop level. A *combinational cycle* in such a circuit is a structural cycle containing only logic gates. The analysis of circuits without combinational cycles is straightforward. Consider for a moment a circuit with no flip-flops. A Boolean function for each node, in terms of the circuit inputs, can be derived by applying functional composition in a topological order. These Boolean functions exactly correspond to the steady-state electrical behavior of the circuit. If flip-flops are present, then these Boolean functions give the behavior at each clock cycle.

On the other hand, circuits with combinational cycles are usually avoided because the presence of cycles can lead to oscillating or unpredictable behavior. However, not all combinational cycles lead to such undesirable behavior. Informally, we say that a circuit is *well-behaved* if for every input, the output stabilizes to a unique value within a bounded amount of time. All acyclic circuits are well-behaved in this sense. Also, some cyclic circuits are well-behaved. For example, for the circuit in Figure 2.1, the output is $z = x$, even though there can be an oscillation at node y when $x = 0$. Other cyclic circuits may not be well-behaved. In Figure 2.2, on input $x = 0$, there exists an assignment of delay values to the circuit such that the output z will oscillate (even though the output of the

AND would seem to be forced to 0).

Combinational cycles arise in practical situations, and therefore techniques to analyze their synchronous behavior are useful. Consider the following situations in which combinational cycles arise:

1. State-holding elements specified at the gate-level. An example of this is an RS-latch designed as a pair of cross-coupled NOR gates.

2. Transistor-level circuits consisting of bidirectional transistors.

3. High-level synthesis, where cycles are created to share circuit resources. An example is Figure 2.3, which computes $z =$ if (c) then $F(G(x))$ else $G(F(x))$. Because c and $\overline{c}$ are mutually exclusive, the cycle is false (i.e., never closed).

4. The composition of Mealy machines. When a single FSM is synthesized within the context of a set of interacting FSMs, the resulting composition may create a combinational cycle [9].

5. The specification of reactive programs in synchronous programming languages. A language like Esterel allows the specification of "zero-delay cycles," and it is the task of the compiler to determine if such cycles are false.

In some cases (1 and 2), combinational cycles are created intentionally to hold state. In other cases (3 and 5), combinational cycles are also created intentionally, but with the knowledge that the cycles are false (i.e., for every input provided by the operating environment, no event can be propagated around the cycle). In still other cases (2, 4 and 5), the cycles may have been created inadvertently, and the circuit may or may not be well-behaved. Regardless of how or why a combinational cycle is created, the only issue is whether the resulting circuit is well-behaved from a black-box point of view.

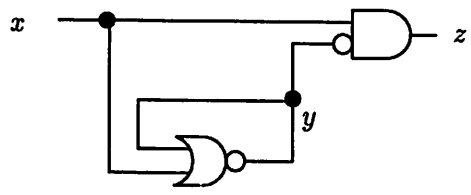


Figure 2.1: Well-behaved, even though y oscillates when $x = 0$ (Figure 4b from [1]).

We analyze the behavior of circuits composed of an arbitrary interconnection of logic gates and flip-flops (operated on a global clock). Even though we are interested in the *logical* analysis of combinational cycles, the issue of circuit delays cannot be avoided because they can affect the steady-state behavior of a circuit. Thus, before any class of well-behaved circuits can be defined, we must state precisely what the underlying delay model is, and what the temporal interaction is between the circuit and its environment. For delays, we use the up-bounded inertial delay model of Brzozowski and Seger [3]. For the interaction with the environment, we assume that the circuit is clocked with a sufficient period so that any output that will eventually stabilize has time enough to do so. Also, the environment provides inputs, and samples outputs, at the clock ticks.

When defining a class of well-behaved circuits, there are two factors to consider. The first is whether the definition is made with respect to single (but arbitrary) input vectors, "combinational," or sequences of input vectors,"sequential." The second factor is the assumed mode of operation of a circuit, "constructive" or "extended fundamental." In the constructive mode, it is assumed that at each clock tick, all the combinational nodes (those driven by logic gates) "forget" their values from the previous clock cycle, and thus are incapable of storing state. On the other hand, in the extended fundamental mode, combinational nodes "remember" their values across clock ticks, and hence state-holding elements can be embedded in the combinational part of a circuit.

Circuits that are well-behaved over sequences of inputs, operating in the extended fundamental mode, are called *sequentially output-stable.* Circuits that are well-behaved over sequences in the constructive mode are called *constructive.*

When considering well-behavedness over single input vectors, the sequential operating mode is irrelevant. Hence, circuits that are well-behaved over single input vectors

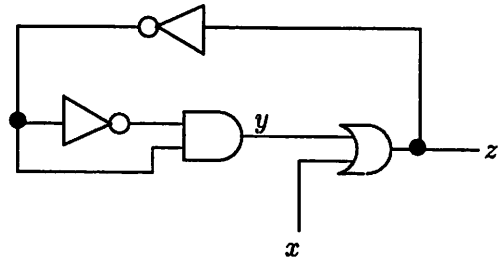


Figure 2.2: Not well-behaved when $x = 0$ (Figure 6b from [1]).

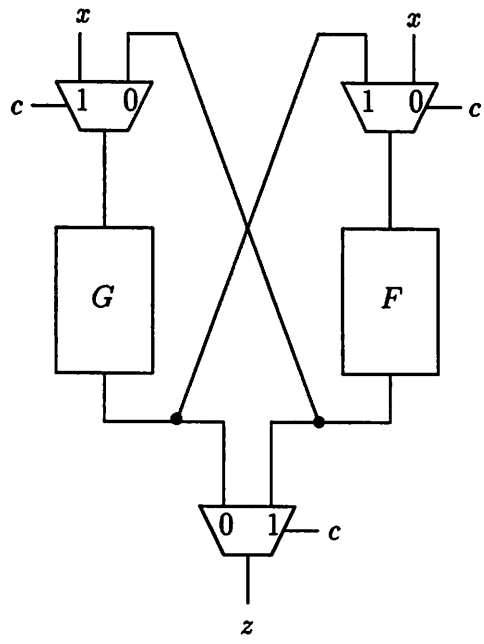


Figure 2.3: Sharing of resources leads to a false combinational cycle (Figure 2 from [1]).

should logically be called "combinational," as Malik calls them. However, since the term "combinational" is often used to describe any network of logic gates, we instead use the term *combinationally output-stable.*

Thus, we have established three classes of well-behaved circuits: combinationally output-stable, constructive, and sequentially output-stable. The relationship between these three classes is shown in Figure 2.4. We will see that the more permissive the class, the more expensive is the classification test. Often, a conservative approximation will suffice for a particular application. We now briefly discuss each class.

Combinational output-stability treats the inputs and outputs of flip-flops as circuit outputs and inputs, respectively. A circuit is combinationally output-stable if for every input vector, the outputs stabilize to a unique value in bounded time, regardless of the initial node values and circuit delay values. The complexity of deciding if a circuit is combinationally output-stable is co-NP-complete in the size of the circuit description.

A circuit is constructive if for every input sequence, there exists a unique output and next state sequence. This is more permissive than combinational output-stability because it considers state reachability. In particular, consider an output that is not well-behaved for a given valuation of the flip-flops (i.e., a state): even if this state is not reachable,

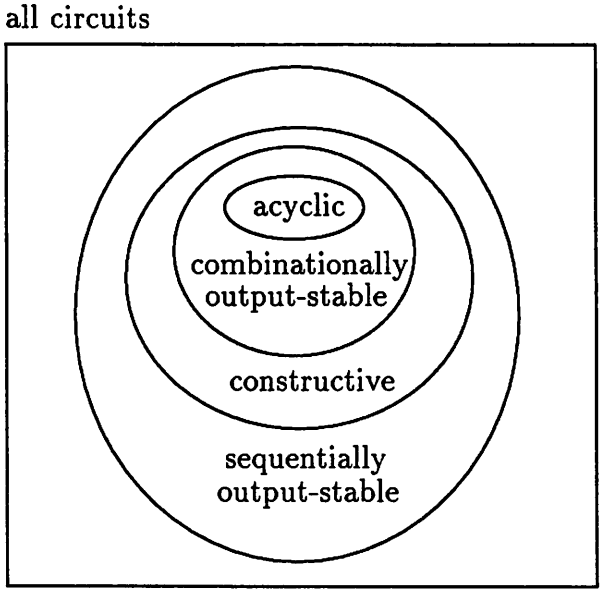


Figure 2.4: Three classes of well-behaved circuits.

the circuit is not combinationally output-stable. The class of constructive circuits is important for two reasons: 1) it coincides exactly with the class of well-behaved Esterel programs, and 2) it is insensitive to glitching on circuit inputs. The complexity of deciding this class is PSPACE-hard in the size of the circuit description.

Whereas the combinationally output-stable and constructive classes are knowingly conservative with respect to hardware, the class of sequentially output-stable circuits is the most permissive class of well-behaved circuits of which we could conceive. First, it is more permissive than the constructive class because it allows the combinational nodes to hold their values across clock ticks. Second, sequential output-stability does not require uniqueness on the flip-flop next state. Thus, the circuit is simply viewed as a black box. However, the definition of this class assumes a stricter condition on the environment, namely that the circuit inputs do not glitch. The complexity of deciding this class is PSPACE-hard in the size of the circuit description.

We provide decision procedures for all three classes of circuits. If a circuit is not well-behaved, then each algorithm returns a sequence of inputs (or a single vector in the case of combinational output-stability) that demonstrates a condition when the circuit is not well-behaved. If a circuit is well-behaved, then a by-product of each algorithm is a functional description of the circuit. This description can be easily translated to a

circuit *without* combinational cycles, having the same I/O behavior as the original circuit. This capability is significant because many CAD tools, such as cycle-based simulators and formal verification tools, require as input, circuits without combinational cycles. Hence, our algorithms can be used as preprocessors for these tools.

A circuit may be well-behaved only if certain assumptions are made about the inputs that the environment provides. Each of our algorithms allows the specification of possible input sequences, and well-behavedness is then checked with respect to this set of sequences.

To illustrate the above discussion, consider the RS-latch in Figure 2.5. This circuit is not combinationally output-stable because the output q is not uniquely determined when the input $rs = 11$. For the same reason, the circuit is not constructive. The circuit is also not sequentially output-stable, but the analysis becomes more interesting. The circuit is not sequentially output-stable when the first input is 00, or when the input 00 follows 11. Assume we know that the environment of the RS-latch never produces such sequences; then we can describe the possible input sequences by the input-less Moore machine in Figure 2.6. With respect to this environment, the RS-latch is sequentially output-stable, and we can derive a Mealy machine with the same I/O behavior (under the synchronous hypothesis); see Figure 2.7. As a last step, a circuit can be synthesized from this machine (Figure 2.8). In summary, we started with a state-holding combinational circuit, and produced, under certain environmental assumptions, a circuit with flip-flops and no combinational cycles.

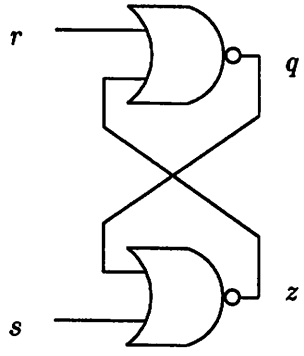


Figure 2.5: RS-latch.

The outline of this chapter is as follows. Section 2.2 discusses related work. Section 2.3 reviews the relevant work of Brzozowski and Seger, which provides the foundation for our analysis. Sections 2.4, 2.5 and 2.6 treat the classes of combinationally output-

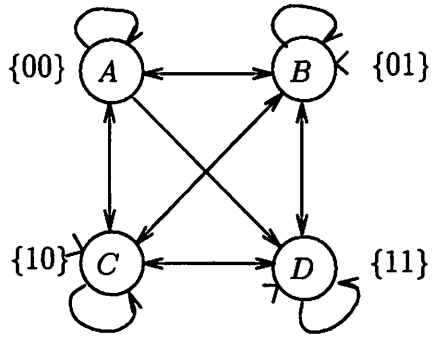


Figure 2.6: An environment for the RS-latch. States B, C and D are initial, as indicated by the incident wedges.

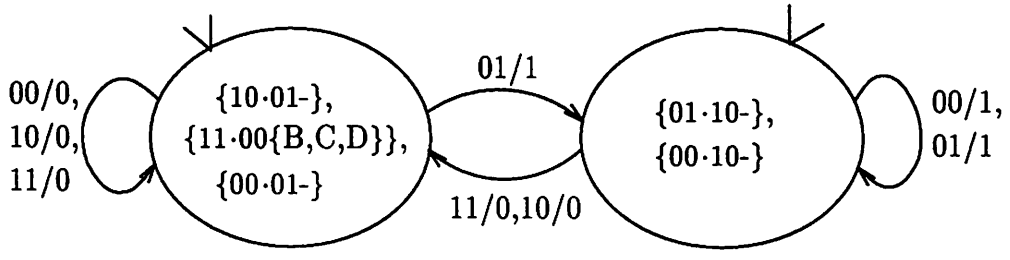


Figure 2.7: Minimized Mealy FSM for the RS-latch composed with an environment.

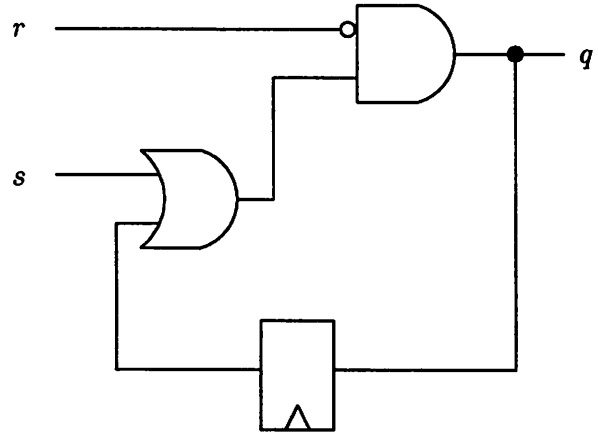


Figure 2.8: Acyclic circuit with the functionality of an RS-latch.

stable, sequentially output-stable, and constructive circuits, respectively.[1] Some proofs are relegated to Section 2.7. Finally, Section 2.8 gives the summary and discusses future work.

2.2 Related work

We divide related work into four categories.

1. Motivation: This work highlights the need for rigorous analysis of combinational cycles.
2. Circuit analysis: These works provide techniques to analyze the behavior of circuits, without attempting to classify well-behaved circuits.
3. Circuit classification: These works classify circuits based on their well-behavedness.
4. FSM extraction: These works provide algorithms to extract finite state machines from transistor-level netlists. Their classification of well-behaved circuits is implicit in the result of their algorithms.

2.2.1 Motivation

Stok [2] explains how false combinational cycles arise naturally when datapath resources (e.g., adders, shifters) are allocated during high-level synthesis. He gives as an example the following scheduled code fragment:

$$\begin{array}{lll} S_1 & c = a + b; & 1 \\ & d = c + e; & 2 \\ \hline S_2 & f = g + e; & 3 \\ & i = f + b; & 4 \end{array}$$

Additions 1 and 2 are scheduled during a different cycle than additions 3 and 4. Hence, one possibility is to share additions 1 and 4, and additions 2 and 3. This results in the datapath shown in Figure 2.9. The combinational cycle shown in bold arises because addition 2 must follow 1, and 4 must follow 3, but 1 and 4 are being shared, and 2 and 3 are being shared. Nonetheless, this cycle is false because the multiplexors m_1 and m_2 are controlled in such a way that the loop is never closed.

[1] A summary of the material in Section 2.5 is presented in [10], and the material in Section 2.6 is presented in [11].

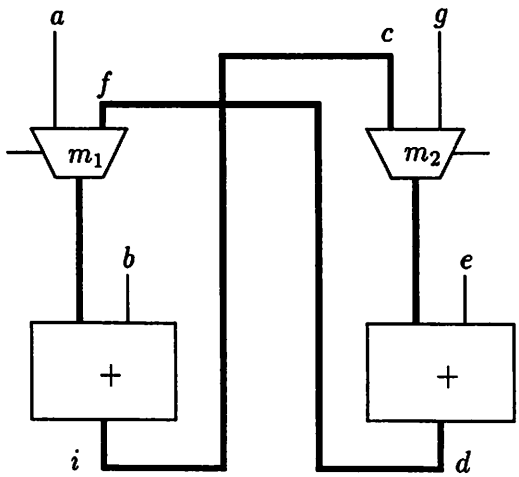


Figure 2.9: Datapath with combinational cycle shown in bold (Figure 2 from [2]).

Thus, even though the natural tendency of high-level synthesis tools is to create (false) combinational cycles in order to share resources, Stok notes that such cycles are undesirable because downstream tools (e.g., logic synthesis, timing analysis) cannot handle cyclic circuits. His approach to solving this problem is to modify the resource sharing phase of high-level synthesis algorithms to prevent cycles from being created in the first place.

The procedure that he presents is quite successful in that for the benchmarks he tested, he was able to generate acyclic circuits without increasing the number of functional units (although more control circuitry was needed in some cases). Nonetheless, we know examples exist (see Figure 2.3) where extra functional units must be added to eliminate the combinational cycles. Given this, our philosophy is to provide rigorous analysis so that circuits with cycles can be handled directly.

2.2.2 Circuit analysis

Brzozowski and Seger [3] study asynchronous circuits under various delay models. In particular, for the up-bounded inertial delay model, they present two methods to analyze the behavior of a circuit. The first, GMW analysis, correctly gives all the possible state sequences that a circuit can follow, although it abstracts away the time at which states are visited. The second, ternary simulation, abstracts away state sequences themselves, and just "summarizes" the set of states a circuit can be in after it has had time to settle.

This work does not address the classification of circuits according to well-behaved-

ness. However, it is pivotal to our research because it provides the theoretical foundation upon which we define and analyze well-behaved circuits. Section 2.3 is devoted to reviewing Brzozowski and Seger's work.

Burch *et al.* [12] model logic gates by ternary-valued relations, where the third value, $\perp$, represents an oscillating or intermediate voltage. By using $\perp$, oscillating behaviors caused by combinational cycles are preserved when gates are composed (by taking the intersection of their corresponding ternary-valued relations). This is in contrast to the use of Boolean relations to model gates, where oscillating behaviors "disappear" when gates are composed. Burch uses the ternary model to solve various substitution and rectification problems for gate-level circuits. However, they do not address the problem of well-behaved circuits.

Maler and Pnueli [13] provide an elegant method to translate asynchronous circuits, described at the gate-level, into timed automata. They use a delay model that is equivalent to the bi-bounded inertial delay model of Brzozowski and Seger; this is more general that the up-bounded inertial delay model that we use because it allows the specification of a lower bound on the delay.

For each gate in the circuit, they introduce a delay element with an associated timer (or clock). They prove that the resulting timed automaton has the same I/O behavior over time as the original circuit, for the given delay model. Using the timed automaton, they are able to perform state reachability and solve several synthesis problems. However, they do not address the well-behavedness problem, nor is it immediate how this problem can be solved within their framework. Also, their analysis, as presented, does not allow flip-flops in the circuit.

The use of timers complicates the analysis considerably. As we will show for the up-bounded inertial delay model, the well-behavedness property is independent of the delay bounds in the circuit, and hence translating to timed automata is excessive. However, for the bi-bounded inertial delay model, this complication may be necessary, and hence this approach may be useful.

2.2.3 Circuit classification

Malik [14] provided the original inspiration for our research. He noted that combinational cycles do arise in practice, but that no method for rigorously analyzing such circuits had ever

been proposed. To remedy this situation, he introduced the class of "combinational" circuits to capture well-behavedness, and proposed using ternary simulation to decide whether or not a circuit is combinational. However, he did not make precise the underlying delay model, which is crucial for formally defining any notion of well-behaved circuits. Consequently, although his intuition was correct, it is not possible to give a formal proof of the correctness of his decision procedure.

The first goal of our research was to formalize Malik's work. We use the up-bounded inertial delay model, and based on this, we are able to formally define the class of combinationally output-stable circuits, which captures the intuition of "combinational" circuits. Also, we are able to formally prove that Malik's algorithm does in fact decide this class correctly. With this goal completed, we then extend the spirit of Malik's work to define well-behavedness over sequences of inputs, rather than just over single input vectors.

Halbwachs and Maraninchi [15] define a class of well-behaved circuits called *consistent* circuits. Basically, they view a circuit as a system of Boolean equations (one equation for each gate), and consider the solutions of this system. For a given input valuation, if the system has at least one solution, and for each output, this output has the same value for all solutions, then the circuit is deemed *weakly consistent.* As a special case, if there is exactly one solution, then the circuit is *strongly consistent.*

This class is not comparable to our class of combinationally output-stable circuits. The circuit in Figure 2.1 is combinationally output-stable, but it is not weakly consistent because there is no consistent assignment to variable y when $x = 0$. On the other hand, the circuit in Figure 2.2 is strongly consistent, but not combinationally output-stable. It is strongly consistent because 0 is the only consistent value for y in the system of equations. It is not combinationally output-stable because when $x = 0$, y can in fact oscillate.

Akin to our study of well-behaved circuits, Halbwachs and Maraninchi perform consistency analysis for circuits with flip-flops, taking into account care inputs and reachable states. Likewise, they can generate a loop-free circuit if a circuit is found to be consistent.

2.2.4 FSM extraction

The works in this group are difficult to compare to our research because of their lack of formality. First, they do not address the underlying delay model. They simply accept as input to their own tools, the output of a circuit extraction tool, like TRANALYZE [16]

or ANAMOS [17], without formal regard to how the tool does the extraction. Second, they do not formally classify those circuits that can be represented by an FSM (i.e., are well-behaved), and those that cannot. Instead, they implicitly define well-behavedness by the result of their extraction algorithms: if the algorithm is successful in extracting an FSM, then the circuit can be considered well-behaved, otherwise not. Third, there is no proof that, when the algorithm is able to extract an FSM, that this FSM has the same behavior as the original circuit; that such proofs are not given comes as no surprise, since a formal framework is never established.

Despite this lack of formality, we describe their algorithms, and discuss their results on some specific circuits. Also, these tools have some interesting capabilities that provide directions for extending our own research.

Singh and Subrahmanyam [18] propose a method to extract FSMs, at the Boolean function level, from transistor netlists. They employ TRANALYZE as a preprocessor, which generates a network of zero-delay logic blocks (defined over the four values $0, 1, X, Z$) and unit-delay elements, from a transistor netlist. The unit-delay elements are introduced by TRANALYZE to break feedback loops and to model charge storage nodes. An assignment of values to the unit-delay elements is called a *configuration*; this is the state of the network. From this, Singh creates the *unit-step relation* by setting each unit-delay variable equal to the zero-delay logic function driving the delay element, and then taking the product of all of these terms.

Singh first tests this relation for stable behavior; that is, for every input combination, there exists some stable binary configuration. The circuit in Figure 2.1 fails this test because when $x = 0$, there is no consistent assignment to the unit-delay element used to break the feedback. If a circuit fails this test, then no further analysis is done. Otherwise, the unit-step relation is massaged to produce the *stable unit-step relation.* This is done by replacing a sequence of transitions passing through transient states by a single transition to the final steady state. The *combinational* test is then performed on this relation: for output z, and for each input combination a, z must evaluate to either 0 or 1, but not both. If a circuit passes this test, then the output functions can be used to derive an equivalent acyclic circuit. Otherwise, if the designer distinguishes certain inputs as clocks, then the *synchronous* test is applied.

For the synchronous test, each delay-element signal is classified into one of three

groups.

1. Combinational: the signal value is independent of the clock phase.
2. Level-sensitive: the signal changes during exactly one clock phase, and is inactive during the rest of the clock cycle.
3. Undefined: the signal is neither combinational nor level-sensitive.

A circuit is declared synchronous if no output depends on an undefined signal, and all the signals are uniquely defined for each stable binary configuration.

From our point of view, the primary shortcoming of this work is that it does not address the issue of delay models directly. Without this, it is not possible to give precise definitions of combinational or synchronous circuits, and hence it is not possible to prove the correctness of the classification algorithms. Instead, this work just accepts whatever the preprocessor (in this case, TRANALYZE) gives to it. TRANALYZE uses a 4-valued algebra in deriving logic gates from transistors. Ironically though, it uses a 2-valued algebra to simplify the logic gates that compose a zero-delay logic block. Hence, the expression $y \cdot \overline{y}$ is simplified to 0. For this reason, the circuit in Figure 2.2 is simplified by TRANALYZE to $z = x$, and thus Singh classifies it as combinational. This is counter to our classification.

This work also suffers from two other weaknesses. The first is embodied in the critical assumption that if there exists a stable binary configuration corresponding to a given input, then the circuit will settle in that configuration when the input is applied. This assumption ignores the possibility that the circuit may settle into an indefinite, race-free oscillation instead. The second weakness is that for a given input and current configuration, only one next configuration is possible. This is inherent in the fact that TRANALYZE uses functions, and not relations, to model the outputs of zero-delay logic blocks. Thus, different next configurations arising from critical races cannot be modeled. Taken together, these two points demonstrate that this technique is incompatible with an inertial delay model.

Pandey *et al.* [19] give a procedure for extracting a cycle-based FSM from a transistor netlist. The first step is to execute TRANALYZE. The second step is to perform symbolic, 3-valued simulation, over one complete clock cycle. This step takes user input specifying the relationship between the clocks, specifying on which clock phases each input is stable, and specifying on which phases each output is sampled. Symbolic simulation is initialized

by setting the unit-delay elements to Boolean symbolic values. A primary input is set to a Boolean value if it is stable at the given clock phase, otherwise it is set to X. At each clock phase, simulation is iterated until the circuit is stable. Since some initial combinations of unit-delay element values may cause instability, "oscillation suppression" is employed to guarantee convergence. Simulation for the next clock phase begins with the unit-delay element values from the end of the previous phase. The symbolic function for a given output is saved from the last clock phase in which the output is sampled; the next state functions for the unit-delay elements are taken from the final result of the simulator. The procedure halts with an error if any next-state function, in the transitive fanin of an output, evaluates to X for some input/present state combination.

This procedure is inherently conservative for several reasons. First, X's at state nodes may only occur for unreachable states, thus having no ill-effect on the circuit. Second, even if there are X's at state nodes for reachable states, these X's may not have an adverse effect on the observable outputs. Third, the environment driving the inputs is not considered; it may be that certain input sequences, which can lead to unstable behavior in a circuit, are never produced by the environment. On the other hand, this procedure may be too permissive for the same reason as in Singh's work, because TRANALYZE performs Boolean simplification.

Kam and Subrahmanyam [20] proposed an algorithm to extract FSMs from transistor netlists. The first step in their algorithm is to invoke ANAMOS, a tool that extracts the ternary valued "excitation function" of each potential storage node in the netlist. Then, holding the inputs and clock fixed, they compute the steady-state response (i.e., the fixed point) of the system of excitation functions. The main weakness of their approach is that this fixed point computation is not guaranteed to converge for some cyclic circuit structures, even though these structures do not lead to unstable behavior. Also, like the work of Pandy, Kam's approach does not take into account sequential input don't cares or state reachability.

2.3 Background

In this section we formally introduce the circuit model and two methods of circuit analysis, the general multiple winner method and ternary simulation. The notation, definitions, and results introduced in this section are largely those of Brzozowski and Seger [3].

2.3.1 Circuits and networks

A circuit is an arbitrary interconnection of logic gates and flip-flops. For the time being, we neglect flip-flops; instead, we treat their I/Os as circuit I/Os. A network is a circuit with delay elements placed on various wires.

2.3.1.1 Circuits

The objects we analyze are circuits composed of an arbitrary interconnection of gates. The topology of a gate circuit is given by a *circuit graph.*

Definition 2.1 [Brzozowski and Seger] A *circuit graph* is a 5-tuple $G = \langle \mathcal{X}, \mathcal{I}, \mathcal{G}, \mathcal{W}, \mathcal{E} \rangle$ where

- $\mathcal{X}$ is a set of *input vertices*, labeled $X_1, X_2, \ldots, X_n$;
- $\mathcal{I}$ is a set of *input-delay vertices*, labeled $x_1, x_2, \ldots, x_n$;
- $\mathcal{G}$ is a set of *gate vertices*, labeled $y_1, y_2, \ldots, y_r$;
- $\mathcal{W}$ is a set of *wire vertices*, labeled $z_1, z_2, \ldots, z_j$; and
- $\mathcal{E} \subseteq (\mathcal{X} \times \mathcal{I}) \cup ((\mathcal{I} \cup \mathcal{G}) \times \mathcal{W}) \cup (\mathcal{W} \times \mathcal{G})$ is a set of directed edges.

■

Notice that a circuit graph is bipartite, with vertex classes $\mathcal{I} \cup \mathcal{G}$ and $\mathcal{X} \cup \mathcal{W}$; for example, two wire vertices cannot be directly connected, nor can a gate vertex be connected to itself. An arbitrary subset $\mathcal{O}$ of circuit vertices (i.e., $\mathcal{X} \cup \mathcal{I} \cup \mathcal{G} \cup \mathcal{W}$) may be designated as *outputs.*

Example 2.2 [Gate circuit and circuit graph] Figure 2.10 shows a gate circuit, and Figure 2.11 gives its corresponding circuit graph. X_1 is an input vertex, x_1 is an input-delay vertex, $y_1, \ldots, y_4$ are gate vertices, and $z_1, \ldots, z_7$ are wire vertices. ■

For each gate, wire, and input-delay vertex, there is an associated *vertex function.*

Definition 2.3 [Brzozowski and Seger] The *vertex function* of a vertex is defined as follows:[2]

[2]For convenience, all gate vertex functions are defined over $\mathbb{B}^{|\mathcal{W}|}$, even though any given gate may only depend on a strict subset of $\mathcal{W}$. A similar comment applies to wire vertex functions. In addition, in the sequel we usually consider a vertex function V as being defined over all input-delay, wire, and gate vertices: $V : \mathbb{B}^{|\mathcal{I}|} \times \mathbb{B}^{|\mathcal{W}|+|\mathcal{G}|} \to \mathbb{B}$.

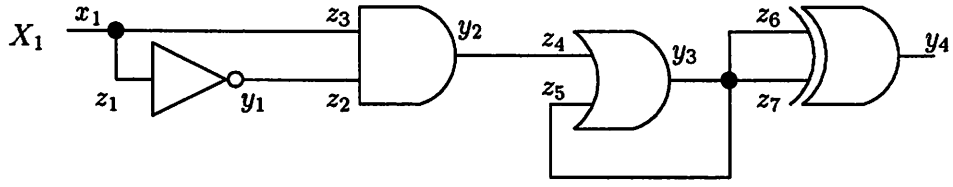


Figure 2.10: Gate circuit. (Figure 4.5 from [3].)

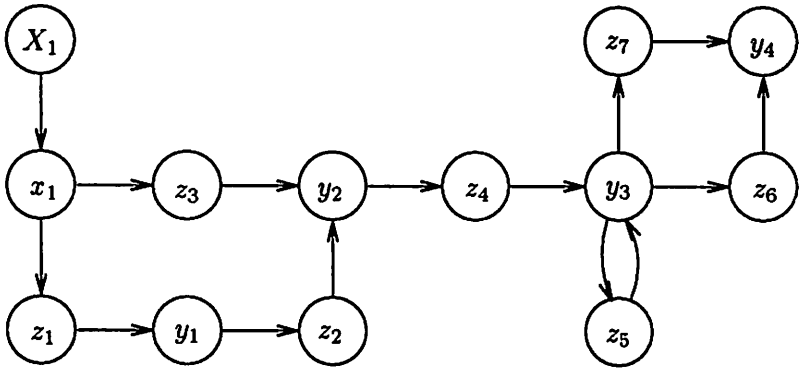


Figure 2.11: Circuit graph corresponding to the gate circuit in Figure 2.10 (Figure 4.6 from [3].)

- gate vertex y_i: $Y_i : \mathbb{B}^{|\mathcal{W}|} \to \mathbb{B}$ (maps a wire-vertex state to $\mathbb{B}$; this is just the Boolean function of the gate corresponding to the gate vertex);
- wire vertex z_i: $Z_i : \mathbb{B}^{|\mathcal{I}|+|\mathcal{G}|} \to \mathbb{B}$ (provides the value of the input-delay or gate vertex driving the wire vertex);
- input-delay vertex x_i: the vertex function is X_i, where X_i maps a state of the environment to $\mathbb{B}$ (i.e., X_i is the input value provided by the environment).

■

Example 2.4 [Vertex functions] In Figure 2.10, the vertex function for y_2 is $Y_2 = z_2 z_3$, for z_1 is $Z_1 = x_1$, and the input-delay vertex function is X_1. ■

2.3.1.2 Up-bounded inertial delays

The value of a vertex and the value of the corresponding vertex function may be different. This permits the physical notion of delay. A delay element has an input $X(t)$, an output $x(t)$, and a delay $\delta(t)$, as depicted in Figure 2.12. The signals $X(t)$ and $x(t)$ vary

with time. They are assumed to be binary and capable of instantaneous changes from 0 to 1 and from 1 to 0. x is *unstable* at time t if $x(t) \neq X(t)$.

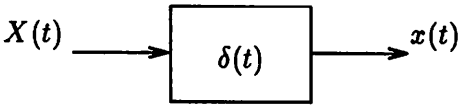


Figure 2.12: Delay element. (Figure 3.3 from [3].)

We employ the *up-bounded inertial* (UIN) delay model.

Definition 2.5 [Brzozowski and Seger] In the *up-bounded inertial* delay model,

$$0 < \delta(t) < D,$$

and the following two properties must be satisfied:

1. *If x changes, then it must have been unstable.*
   Formally, if $x(t)$ changes from α to $\overline{\alpha}$ at time τ, then there exists $\delta > 0$ such that $X(t) = \overline{\alpha}$ for $\tau - \delta \leq t < \tau$.

2. *x cannot be unstable for D units of time without changing.*
   Formally, if $X(t) = \alpha$ for $\tau \leq t < \tau + D$, then there exists a time $\tilde{\tau}, \tau \leq \tilde{\tau} < \tau + D$, such that $x(t) = \alpha$ for $\tilde{\tau} \leq t < \tau + D$. (Note that this property implies that the δ in Property 1 must be less than D.)

■

Intuitively, if an input pulse is at least D units of time, then the output must respond within D. If an input pulse is less than D, then the output may or may not respond. Figure 2.13 shows two possible responses to an input waveform, where $D = 2$.

2.3.1.3 Networks

A *network* N is derived from a circuit graph by associating delay elements with some subset of vertices, called the *state vertices*. The minimum requirement is that each cycle in the circuit graph must contain at least one state vertex. A network where each vertex is a state vertex is called a *complete network*. Each state vertex has a *state variable* s_i, a *delay bound* D_i, and an *excitation function* S_i, defined as follows.

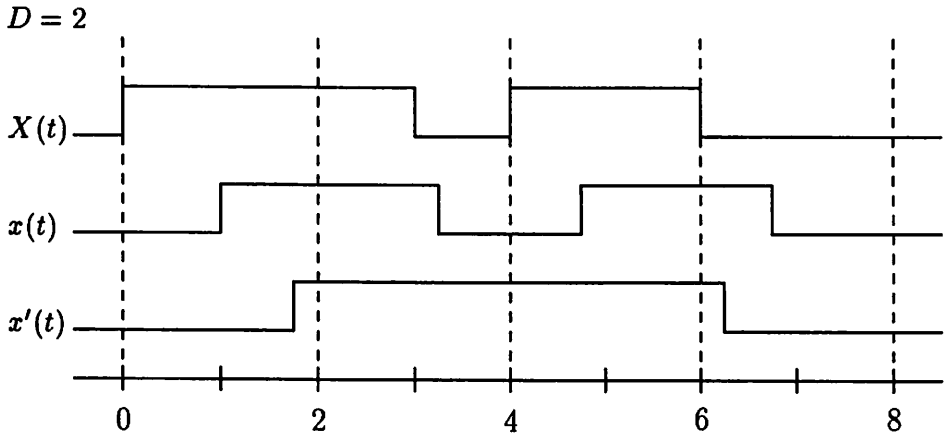


Figure 2.13: Up-bounded inertial delay waveforms.

Definition 2.6 [Brzozowski and Seger] Start with the vertex function. Then repeatedly remove all dependencies on vertices that have not been chosen as state vertices, by using functional composition of the vertex functions. The result of this process is the *excitation function* S_i. ∎

A *total state* $c = a{\cdot}b$ of a network is an $(n+m)$-tuple of binary values, the n-tuple a being the value of the input excitations, and the m-tuple b being the values of the state variables $s_1, s_2, \ldots, s_m$. The total state $a{\cdot}b$ is *stable* if $b_i = S_i(a{\cdot}b)$ for each state variable s_i, and is *unstable* otherwise.

In Section 2.4, we need to also reason about the vertices not selected as state vertices. The depth of a vertex in a network is the longest path from a state vertex to that vertex.

Definition 2.7 Consider a vertex v in a circuit graph G, and a network N derived from G. The *depth* of v in N is:

$$depth(v) = \begin{cases} 0 & \text{if } v \text{ is a state vertex in } N, \\ 1 + \max\{depth(u) | (u,v) \in \mathcal{E}\} & \text{otherwise.} \end{cases}$$

Since the state vertices are required to form a feedback-vertex set, the depth of v is uniquely defined. ∎

A value for each of the state vertices uniquely determines the value for each of the remaining vertices of the circuit graph. These values are computed by the set of circuit equations of the graph.

Definition 2.8 Let $a \cdot b$ be a total state of network N. Let v be a vertex with vertex function V. The *circuit equation* F of v is defined inductively on the depth of v.

$$F(a \cdot b) = \begin{cases} b_i & \text{if } v \text{ corresponds to state variable } s_i, \\ V(a \cdot F_1(a \cdot b) \cdot F_2(a \cdot b) \cdot \ldots \cdot F_{|\mathcal{W}|+|\mathcal{G}|}(a \cdot b)) & \text{otherwise.} \end{cases}$$

Remember that state variables have depth 0 by definition. The value of F on $a \cdot b$ is uniquely defined because it only depends on the values of vertices with lower depth. ∎

Note that for any vertex in a complete network, the vertex function, excitation function, and circuit equation are the same.

2.3.1.4 Behavior

Thus far, we have described the topology of a circuit and the concepts of vertex functions and delay elements. The concept of a *UIN_a-history* captures the behavior of a network as it evolves over time in response to an input a. For state variable s_i, $s_i(t)$ is the value of s_i at time t, and $S_i(X(t) \cdot s(t))$ is the value of the corresponding excitation at time t, where $X(t) \cdot s(t)$ gives the value at time t of the total state.

Definition 2.9 [Brzozowski and Seger] A UIN_a-history of a network N for some $a \in \mathbb{B}^n$ is an ordered triple $\mu = \langle \Theta, X(t), s(t) \rangle$, where

- Θ is a strictly increasing sequence $\Theta = (t_0, t_1, \ldots)$ of real numbers giving the instants at which the state vector s changes.
- $X(t) = a$ for all $t \geq t_0$.
- $s(t)$ maps the real numbers to $\mathbb{B}^m$, and satisfies the properties that:
  1. $s(t)$ is constant during any interval $[t_i, t_{i+1})$,
  2. $s(t)$ changes at each t_i,
  3. if the sequence $(t_0, t_1, \ldots, t_r)$ is finite, then the last state reached, $s(t_r)$, must be stable, and
  4. if the sequence $(t_0, t_1, \ldots)$ is infinite, then only a finite number of state changes occur in any finite time interval (i.e., non-Zeno).
- the input/output waveform of each delay element satisfies the properties of UIN delays.

■

Note that the definition of UIN_a-history places no restriction on the initial state $s(t_0)$. A UIN_a-history can be seen as a timed sequence of states. Also, one can associate a corresponding untimed history giving the sequence of states through which the network passes.

Example 2.10 [UIN history] A UIN history is shown for the network in Figure 2.14, where $D_1 = 1$, $D_2 = 3$ and $D_3 = 2$. The UIN_1-history is given by $\Theta = (0, 0.5, 3.0, 4.3)$, $X(t) = 1$ for all $t \geq 0$, and $s(t)$ is given by the waveforms s_1, s_2, s_3 in Figure 2.14. One can verify that these waveforms are consistent with the properties of UIN delays, for the specified delay bounds. The corresponding untimed history is $(001, 101, 111, 110)$. ■

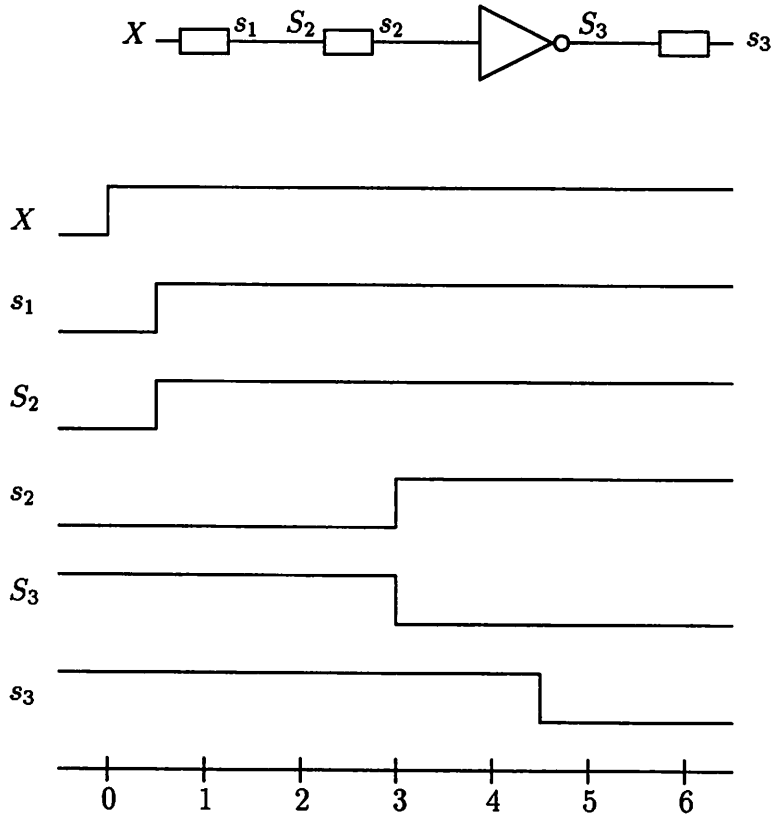


Figure 2.14: A UIN_1-history; $D_1 = 1$, $D_2 = 3$ and $D_3 = 2$.

Because each inertial delay is up-bounded, the network can remain in the "transient" phase after an input change for only a bounded time, before passing into the "non-transient" phase. The following definition and theorem make this notion precise.

Definition 2.11 [Brzozowski and Seger] Let N be a network with maximum delay bound D. Let N be started in state b with the input held constant at a. A state b' is said to be $D(a,b)$-*nontransient* with limit τ for $a{\cdot}b$ if it is reachable from b and there exists a UIN_a-history μ and a time $t \geq \tau$ such that $s(t) = b'$. ∎

That is, a state is $D(a,b)$-nontransient with limit τ if the network can be in that state some time after τ. Note that nontransient does not mean non-changing; it simply means that the network can be in any of a certain set of states after the bound τ.

Theorem 2.12 (Brzozowski and Seger) *Let N be a network with m up-bounded delays with upper bound D. Suppose N is in state b at time 0 and the input is held constant at a from time 0 until time $t \geq (2^m - 2)D$. Then the state of the network at time t is a $D(a,b)$-nontransient state, with limit $\tau = (2^m - 2)D$.*

In other words, by waiting at most $(2^m - 2)D$ time, the network has enough time to pass through any transients. In the sequel, $D(a,b)$-nontransient states will always be with limit $(2^m - 2)D$. We refer to a state as simply *nontransient* if it is $D(a,b)$-nontransient for a given a and b.

2.3.2 GMW analysis

General multiple winner (GMW) analysis is a technique to determine the response of a network to a given input. The technique is called "general" because the relative values of the delays are not specified. The only assumption is that the delays are bounded from above.

For a total state $c = a{\cdot}b$, the set of *unstable state variables* is defined as

$$\mathcal{U}(a{\cdot}b) = \{s_i \mid b_i \neq S_i(a{\cdot}b)\}.$$

That is, a state variable s_i is unstable with respect to state $a{\cdot}b$ if applying the corresponding excitation function S_i to $a{\cdot}b$ yields a value different from the current value b_i. State c is stable if $\mathcal{U}(c) = \emptyset$. The GMW relation $R_a \subseteq \mathbb{B}^m \times \mathbb{B}^m$ describes how the internal state of network N evolves with the input held constant at a.

Definition 2.13 [Brzozowski and Seger] For any $b \in \mathbb{B}^m$,

- bR_ab, if $\mathcal{U}(a{\cdot}b) = \emptyset$, i.e., the total state $a{\cdot}b$ is stable

- $bR_ab^{\mathcal{K}}$, if $\mathcal{U}(a{\cdot}b) \neq \emptyset$, and $\mathcal{K}$ is any nonempty subset of $\mathcal{U}(a{\cdot}b)$, where $b^{\mathcal{K}}$ means b with all the variables in $\mathcal{K}$ complemented.

■

The model is called "multiple winner," because in a race condition, any nonempty subset of unstable state variables can change at the same time. Note that R_a makes no reference to an initial state.

The relation R_a can be depicted as a directed graph, where an edge from b to b' indicates that bR_ab'. A state b may have more than one immediate successor, indicating a race condition. A state b with a self-loop indicates that $a{\cdot}b$ is stable. $R_a(b)$ denotes the graph R_a restricted to those states reachable from b.

Example 2.14 [RS-latch] Consider the RS-latch in Figure 2.5, with inputs r and s and state variables q and z. The graph of R_{01} is shown in Figure 2.15. Each pair of binary values is a state qz. An underlined value indicates that the corresponding variable is unstable in that state. The states 01 and 00 each have one unstable variable, so each has a unique successor state. The state 11 has two unstable variables, so it has three successor states, one for each nonempty subset of unstable variables. State 10 is stable, so it has a self-loop. The graph corresponds to $R_{01}(11)$, the subgraph induced by those states reachable from 11, holding the input constant at 01. ■

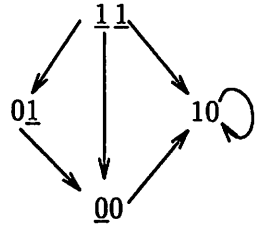


Figure 2.15: Possible state sequences over qz, for the RS-latch with input $rs = 01$.

2.3.2.1 Nontransient behavior

For GMW analysis, there is a concept corresponding to the nontransient states of the UIN delay model. Consider the GMW relation of Figure 2.16, for some input a. The cycle $(0\underline{0}\,\underline{0}, 0\underline{1}\,\underline{0})$ is called *transient* because there exists a variable (the third one)

that is unstable and has the same value (0) in every state of the cycle. Since all delays are up-bounded, the network cannot remain in this cycle indefinitely. Contrast this to the *nontransient* cycle ($\underline{0}\underline{0}1, \underline{1}\underline{1}0$). The network can remain in this cycle indefinitely because each unstable variable changes value during the cycle. As an aside, the network is not constrained to remain in this cycle, since it can transition to the stable state 101 whenever it is in state 001.

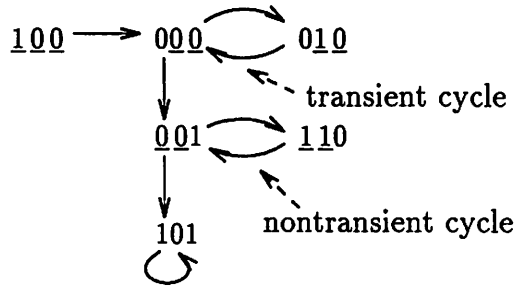


Figure 2.16: A general multiple winner relation.

Those states that are in a nontransient cycle, or follow a nontransient cycle, are called *outcome* states. Formally, Brzozowski and Seger define $out(R_a(b))$, the *outcome states of* b, as that set of states in the graph $R_a(b)$ that are reachable from b via a nontransient cycle. For the graph in Figure 2.16, $out(R_a(100)) = \{001, 110, 101\}$ and $out(R_a(101)) = \{101\}$. Note that states of a transient cycle can be outcome states of b, as long as they are reachable from b via a nontransient cycle. The next result establishes the correspondence between the nontransient states of the UIN delay model and the outcome states of GMW analysis.

Theorem 2.15 (Brzozowski and Seger) *Let N be a network with maximum delay bound D. Let N be started in state b with input held constant at a. Then $D(a,b)$-nontransient = $out(R_a(b))$.*

In our analysis of combinational cycles, we are interested only in the states that a network can be in after the network has had time to "settle". The sequence of state transitions made to reach these states is of no concern to us. Theorem 2.15 enables us to capture this set of states using GMW analysis.

2.3.3 Ternary simulation

In theory, GMW analysis could be used to compute the outcome states of an input change. However, in practice, constructing the graph of R_a and traversing it is computationally intractable. Ternary simulation is an efficient means to "summarize" the set of outcome states. Furthermore, as we will show in Section 2.4, the result of ternary simulation is sufficient to determine if a network is combinationally output-stable.

2.3.3.1 Definitions

Ternary simulation uses a third value, Φ, to denote an uncertain or changing value on a wire. The set $\{0, 1, \Phi\}$ is a partially ordered set on the "uncertainty" relation $\sqsubseteq$ where,

$$0 \sqsubseteq 0, 1 \sqsubseteq 1, \Phi \sqsubseteq \Phi, 0 \sqsubseteq \Phi, \text{ and } 1 \sqsubseteq \Phi.$$

When $s \sqsubseteq t$, we say that t *covers* s. Likewise, the vector $\langle t_1, t_2, \ldots, t_n \rangle$ covers $\langle s_1, s_2, \ldots, s_n \rangle$ if $s_i \sqsubseteq t_i$, for all i. Any nonempty subset of $\{0, 1, \Phi\}$ has a *least upper bound*, or *lub*. In particular, $lub\{0\} = 0$, $lub\{1\} = 1$, and the *lub* of every other nonempty subset is equal to Φ.[3]

A *ternary function*[4] $\mathbf{f}$ is a mapping from $\{0, 1, \Phi\}^n$ to $\{0, 1, \Phi\}$. For any Boolean function f there exists a natural *ternary extension*, defined as follows:

$$\mathbf{f}(\mathbf{a}) = lub\{f(t) \mid t \in \mathbb{B}^n \text{ and } t \sqsubseteq \mathbf{a}\}.$$

Figure 2.17 shows the ternary extension for several Boolean functions. They follow the basic rule that a 0 or 1 output value can be deduced whenever there is sufficient information available at the inputs. For example, a 0 at any input of an AND gate forces the output to 0. An important property of the ternary extension $\mathbf{f}$ of any Boolean function f is *monotonicity*:

$$\mathbf{a} \sqsubseteq \mathbf{b} \text{ implies } \mathbf{f}(\mathbf{a}) \sqsubseteq \mathbf{f}(\mathbf{b}).$$

That is, if $\mathbf{b}$ is at least as uncertain as $\mathbf{a}$, then the output $\mathbf{f}(\mathbf{b})$ is at least as uncertain as $\mathbf{f}(\mathbf{a})$.

[3] Other authors refer to the third value as $\perp$ or X. And rather than having an ordering on uncertainty, they may use an ordering on certainty, or information content, where $\perp$ or X is the least element of the partial order.

[4] Following Brzozowski and Seger's convention, boldface is used to refer to ternary valued functions, relations, and variables.

a	NOT
0	1
1	0
Φ	Φ

a	b	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0
Φ	0	0	Φ	Φ
Φ	1	Φ	1	Φ
0	Φ	0	Φ	Φ
1	Φ	Φ	1	Φ
Φ	Φ	Φ	Φ	Φ

Figure 2.17: Ternary extension for the NOT, AND, OR, and XOR functions.

Given a binary network N with n inputs and m state variables, its ternary extension $\mathbf{N}$ is just N with each excitation function $S_i : \{0,1\}^{n+m} \rightarrow \{0,1\}$ replaced by its ternary extension $\mathbf{S}_i : \{0,1,\Phi\}^{n+m} \rightarrow \{0,1,\Phi\}$. The vector of ternary excitation functions is denoted by $\mathbf{S}$. This corresponds to the interpretation of the network in Scott's ordered Boolean domain $B_\perp = \{\perp, 0, 1\}$, familiar in other communities [21, 22].

2.3.3.2 Algorithm A

Ternary simulation is applied to a ternary network $\mathbf{N}$, starting from a binary valued initial state b with the input held constant at the binary value a. As presented by Brzozowski and Seger, it is implemented by applying two algorithms, A and B, in sequence. Algorithm A takes as input the total state $a \cdot b$, and propagates maximum uncertainty into the network, while leaving the input fixed at a.

Algorithm A
$h := 0$;
$\mathbf{s}^0 := b$;
repeat
  $h := h + 1$;
  $\mathbf{s}^h := lub\{\mathbf{s}^{h-1}, \mathbf{S}(a \cdot \mathbf{s}^{h-1})\}$;
until $\mathbf{s}^h = \mathbf{s}^{h-1}$;

$\mathbf{s}^h$ denotes the ternary vector of state values at each iteration. Due to the monotonicity of the ternary extensions of the excitation functions, it can be shown that Algorithm A

converges in at most m steps. The final value of $\mathbf{s}^h$ is denoted by $\mathbf{s}^A_{a,b}$, or by AlgA(a, b).

Example 2.16 [RS-latch] Algorithm A is illustrated in Figure 2.18, where the input is $rs = 01$ and the initial state is $qz = 01$, and $rs{\cdot}qz$ is shown at each step. ■

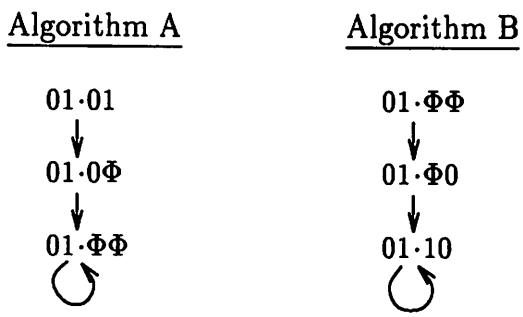


Figure 2.18: Algorithms A and B in operation on the RS-latch, over states $rs{\cdot}qz$.

Since m passes are necessary for convergence in the worst case, and each pass requires $O(m)$ time, the complexity of Algorithm A is $O(m^2)$ (this assumes that the excitation functions are evaluated in topological order, and each function can be evaluated in constant time). In fact, Berry gives a linear time algorithm for ternary simulation on concrete inputs [23]. Ultimately, we are interested in simulating a network for all inputs, and hence in Section 2.4 we turn to symbolic methods to gain efficiency.

The result of Algorithm A is exactly equal to the *lub* of the reachable states found by GMW analysis starting in the total state $a{\cdot}b$.

Theorem 2.17 (Brzozowski and Seger) *Let N be a complete binary network and* N *be its ternary extension. Then*

$$\mathbf{s}^{\mathrm{A}}_{a,b} = \mathit{lub}\ \mathit{reach}(R_a(b)).$$

2.3.3.3 Algorithm B

The second phase of ternary simulation is to apply Algorithm B to N starting from the initial state $\mathbf{s}^A_{a,b}$ while holding the input constant at a. Algorithm B removes as much as possible the uncertainty introduced by Algorithm A.

Algorithm B

$h := 0$;
$\mathbf{t}^0 := \mathbf{s}^A_{a,b}$;
repeat
  $h := h + 1$;
  $\mathbf{t}^h := \mathbf{S}(a \cdot \mathbf{t}^{h-1})$;
until $\mathbf{t}^h = \mathbf{t}^{h-1}$;

Again, it can be shown that Algorithm B converges in at most m steps. The final value of $\mathbf{t}^h$ is denoted by $\mathbf{t}^B_{a,b}$, or by AlgB(a, AlgA(a, b)).

Example 2.18 [RS-latch] Algorithm B is illustrated in the right hand side of Figure 2.18, where the input is still $rs = 01$ and the initial state is the final value from Algorithm A, $qz = \Phi\Phi$, ∎

The complexity of Algorithm B is $O(m^2)$. Algorithm B is computing the greatest fixed point of the excitation functions, over the domain $\{0, 1, \Phi\}^m$. As such, there are other ways of computing this fixed point [21], but we are interested not in the method, but only in the result, which we use to characterize combinationally output-stable networks in Section 2.4.

The key theorem is that the result of Algorithm B is exactly equal to the *lub* of the set of outcome states found by GMW analysis starting in the total state $a \cdot b$.

Theorem 2.19 (Brzozowski and Seger) *Let N be a complete binary network and* N *be its ternary extension. Then*

$$\mathbf{t}^{\mathrm{B}}_{a,b} = \mathit{lub}\ \mathit{out}(R_a(b)).$$

In particular, note that if $\mathit{out}(R_a(b))$ consists of a single state, then $\mathbf{t}^B_{a,b}$ will in fact be a binary vector identifying this state.

2.4 Combinational output-stability

In this section, we give an operational definition of combinationally output-stable networks, and then show three equivalent notions of combinational output-stability. One of these definitions leads directly to an implicit algorithm proposed by Malik for deciding combinational output-stability. We present this algorithm, along with a refinement to the algorithm that heuristically minimizes the computation involved.

2.4.1 Definition and properties of combinational output-stability

Intuitively, a network is combinationally output-stable if for every input value, there exists a unique output value to which the network stabilizes in bounded time, regardless of the initial state of the network. This section deals with the static analysis of networks; that is, analysis for a single input vector. In the next section, we add explicitly clocked elements to networks, and then analyze the behavior of networks over multiple clock cycles.

2.4.1.1 Combinational output-stability

We begin with an operational definition of what it means for a network with up-bounded inertial delays to be combinationally output-stable. Let the set $\mathcal{O}$ of output vertices be $\{j_1, j_2, \ldots, j_p\}$. For example, $F_{j_i}(a{\cdot}b)$ refers to the circuit equation of output j_i evaluated on the total state $a{\cdot}b$.

Definition 2.20 Network N is *combinationally output-stable* if for every input a, there exists a unique $d \in \mathbb{B}^p$ such that $\forall b \in \mathbb{B}^m, \forall b' \in D(a,b)$-nontransient, $F_{j_i}(a{\cdot}b') = d_i$, for $1 \leq i \leq p$. ∎

This says that a network is combinationally output-stable if for every input a, there is a unique output value d to which the network stabilizes in bounded time; this condition must hold regardless of the starting state b and for all possible delay values respecting the delay bounds. (As stated, this condition is vacuously true if N has no input vertices. In order to avoid the vacuous case, a dummy input connected to nothing could be added to the network. This possibility will not be discussed in the sequel.)

Combinational output-stability is defined for networks, and not circuits, because combinational output-stability is dependent on which circuit vertices are chosen as state vertices (or in other words, where delay elements are placed). Note that an acyclic network is combinationally output-stable regardless of the placement of delays, because the outputs are functionally determined by the inputs.

Example 2.21 [Placement of delays] Consider the circuit in Figure 2.19 (Figure 6a from Malik [1]). Let N_1 be the network with just one state variable, y_1. The excitation function for y_1 is $S_1 = x + (\overline{y_1}{\cdot}y_1) = x$. Since y_1 is uniquely determined for all values of the input

x, the output is also uniquely determined, and hence N_1 is combinationally output-stable. Now let N_2 be the network with state variables y_1 and y_2, and corresponding excitation functions $S_1 = x + (\overline{y_1} \cdot y_2)$ and $S_2 = y_1$. The GMW relations over $y_1 y_2$ for $x = 0$ and $x = 1$ are shown in Figure 2.20. When $x = 1$, $y_1 y_2$ stabilizes to 11; this is fine. However, when $x = 0$, both 01 and 10 are $D(0, 01)$-nontransient states. Since the circuit equation for the output z is S_1, and $S_1(0 \cdot 01) = 1$ and $S_1(0 \cdot 10) = 0$, the output is not uniquely determined, and hence N_2 is not combinationally output-stable. ∎

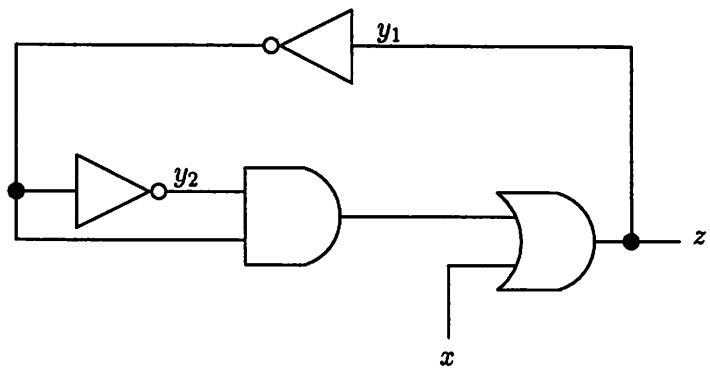


Figure 2.19: Different placement of delay elements affects combinational output-stability.

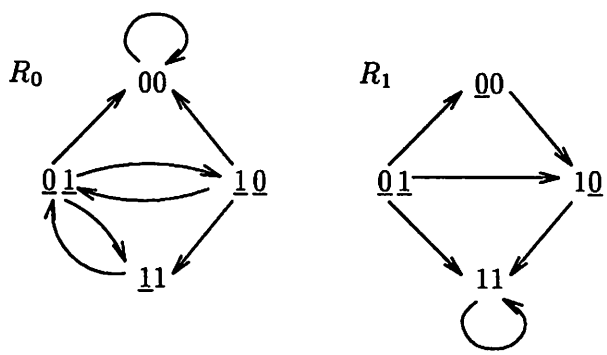


Figure 2.20: GMW relation over $y_1 y_2$ for network N_2.

As a side note, the work of Kautz is often cited as proving the existence of logic functions whose minimal circuit implementation using 2-input NOR gates must have combinational cycles [24]. Interestingly, the example circuit he gives actually fails the test for combinational output-stability, under the UIN delay model. In particular, he suggests a

class of m-input, m-output logic functions, which for $m = 3$, have the form:

$$\begin{aligned} z_1 &= \overline{x_1}x_2 + \overline{x_1}\,\overline{x_3} \\ z_2 &= \overline{x_1}\,\overline{x_2} + \overline{x_2}x_3 \\ z_3 &= x_1\overline{x_3} + \overline{x_2}\,\overline{x_3} \end{aligned}$$

He claims that the minimum 2-input NOR gate implementation is the circuit in Figure 2.21, which has a combinational cycle. However, under the assumption that $\hat{y}_3$ and z_3 are among the chosen state variables, then for the input $x_1 = 1$, $x_2 = 1$ and $x_3 = 0$, z_3 is not uniquely determined. The key point is that the signal y_3 has two paths to the gate at $\hat{y}_3$, and these two paths may have different delays; this can cause an oscillation at $\hat{y}_3$, and hence at z_3. The analysis is similar to that of network N_2 in Example 2.21.

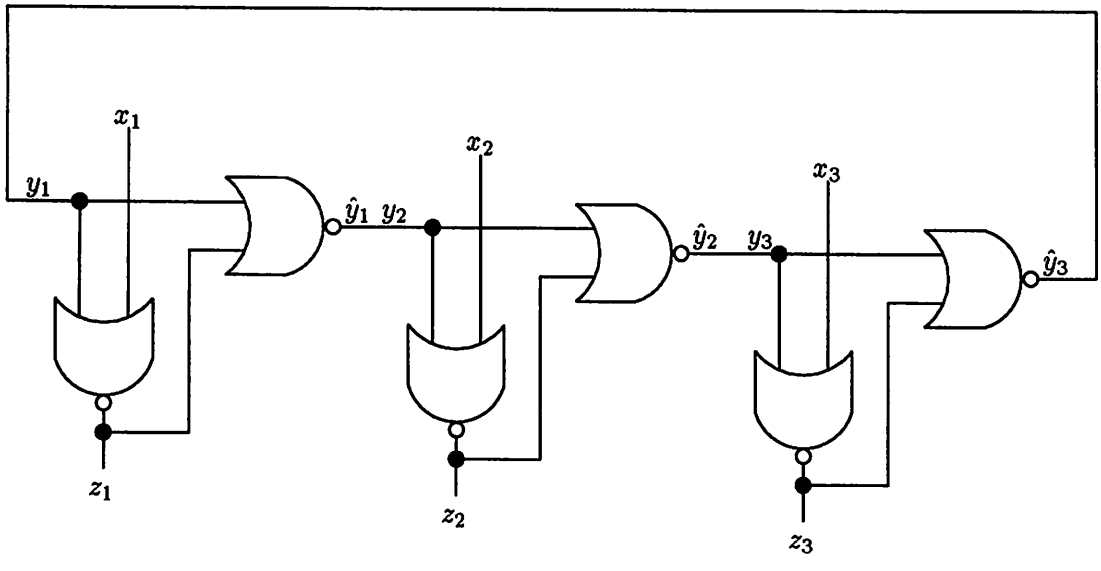


Figure 2.21: Kautz's circuit with a combinational cycle.

2.4.1.2 Properties and equivalent characterizations of combinational output-stability

The definition of combinational output-stability is based on the nontransient states of a network, where the network has a specific maximum delay bound, say D. One might wonder if the set of nontransient states changes as the delay bounds change, and hence if the property of being combinationally output-stable depends on the delay bounds. The answer is no, as demonstrated by the following theorem.

Theorem 2.22 *Consider two networks N and N' that are exactly the same, except that N has maximum UIN delay bound D and N' has maximum UIN delay bound D'. Let a be an input and b be an internal state. Then $D(a,b)$-nontransient $= D'(a,b)$-nontransient.*

Proof The definition of the GMW relation R_a is independent of the delay bounds (it only assumes they are finite), and hence the definition of $out(R_a(b))$ is independent of the delay bounds. By Theorem 2.15

$$\begin{aligned} out(R_a(b)) &= D(a,b)\text{-nontransient, and} \\ out(R_a(b)) &= D'(a,b)\text{-nontransient,} \end{aligned}$$

and the result follows trivially. ∎

In summary, once the state variables of a circuit have been chosen, then the combinational output-stability property is a function only of the circuit equations, and not of the specific delay bounds.

The next proposition reduces the test for combinational output-stability to the GMW analysis of the outcome states.

Proposition 2.23 *Network N is combinationally output-stable if and only if for every input a, there exists a unique $d \in \mathbb{B}^p$ such that $\forall b \in \mathbb{B}^m, \forall b' \in out(R_a(b))$, $F_{j_i}(a \cdot b') = d_i$, for $1 \leq i \leq p$.*

Proof Trivial, since by Theorem 2.15, $D(a,b)$-nontransient $= out(R_a(b))$. ∎

We have shown that the placement of delay elements affects the combinational output-stability property, but we have not addressed where delay elements should be placed. The most conservative approach is to assume that each gate and each wire can have a delay independent of the others. This corresponds to a complete network. This assumption is warranted because it corresponds to the reality of circuits. Also, one would not want to rely on matched delays to guarantee the correct logical operation of a circuit. It turns out that adopting this conservative approach allows up to leverage the power of ternary simulation, and hence simplify the analysis. The following theorem gives the correspondence between combinational output-stability of complete networks and ternary simulation. Remember that a complete network contains all the output vertices as state variables; we denote by $b{\downarrow_O}$ an internal state b restricted to the output state variables.

Theorem 2.24 *Let N be a complete network. The following are equivalent statements, where a is a network input value, d is a binary output value and b and b' are binary state values.*

1. *N is combinationally output-stable.*
2. *$\forall a$, there exists a unique d such that $\forall b \in \mathbb{B}^m, \mathrm{t}^{\mathrm{B}}_{a,b}{\downarrow_O} = d$.*
3. *$\forall a$, there exists a unique d such that $AlgB(a, \Phi^m){\downarrow_O} = d$.*[5]

The proof of this theorem is presented in Section 2.7.1. These statements have the following interpretations. For every input a, there exists a unique binary output d such that

1. regardless of the starting state b, every state that the network can be in after the transient phase, when restricted to the output variables, has value d.
2. regardless of the starting state b, the result of Algorithm A followed by Algorithm B, when restricted to the output variables, has value d.
3. the result of Algorithm B applied to the starting state of all Φ values, when restricted to the output variables, has value d.

Using a complete network would seem to complicate the analysis of combinational output-stability, because there would inevitably be more state combinations to consider. Fortunately, as will be shown in the next subsection, to determine if a complete network is combinationally output-stable, it suffices to apply Algorithm B to a network containing only feedback variables.

Deciding if a network is combinationally output-stable is intrinsically hard. Malik stated the following result, and gave a proof within his context. The proof is reproduced here for our circuit model, using our terminology.

Theorem 2.25 (Malik) *Deciding if a complete network is combinationally output-stable is co-NP-complete.*

[5]This is just another way of saying that the greatest fixed point of the ternary excitation output functions does not have any Φ components.

Proof We show that deciding if a network is *not* combinationally output-stable is NP-complete.

Membership in NP: To show that a network is not combinationally output-stable, one needs to produce an input a on which an output is unstable. A guess can be verified in time polynomial in the network size by examining the result of $\mathrm{AlgB}(a, \Phi^m)$, in accordance with Theorem 2.24.

NP-hardness: The reduction is from Boolean satisfiability. Let f be a Boolean function that we wish to check for satisfiability. Consider the complete network in Figure 2.22. Clearly, z is not combinationally output-stable if and only if f is satisfiable. ∎

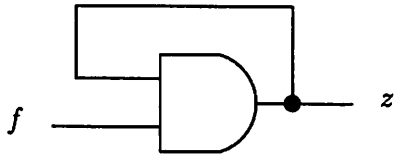


Figure 2.22: z is not combinationally output-stable if and only if f is satisfiable.

2.4.2 Malik's algorithm for deciding combinational output-stability

Independently of the work of Brzozowski and Seger, Malik devised a BDD-based algorithm for determining whether or not a network is combinationally output-stable (Malik uses the term "combinational" instead) [1]. It turns out that Malik's algorithm is closely related to the test given in statement 3 of Theorem 2.24. In this subsection, we present the details of Malik's algorithm, and in the next subsection we propose a refinement to Malik's algorithm.

2.4.2.1 Malik's algorithm on concrete values

The algorithm proposed by Malik works with symbolic input values; however, we begin by presenting his algorithm for concrete input values. Before the algorithm is invoked, a vector $\mathbf{y}$ of k feedback gate vertices is selected to serve as the state variables of the circuit. The algorithm starts with the feedback variables initialized to Φ (line 2). In each round, the input a and the current values of the feedback variables are propagated through the network to compute the new value at each gate vertex (lines 5-6). At the end of each round, the values of the feedback variables are updated with the new values of the feedback gate

vertices (lines 7-8). The algorithm terminates when one complete round fails to change the value of any feedback variable.

Malik's Algorithm

1 $h := 0$;
2 $\mathbf{y}^0 := \Phi^k$;
3 **repeat**
4 $\quad h := h + 1$;
5 $\quad$ for each gate vertex in topological order
6 $\quad\quad \mathbf{F}_j(a{\cdot}\mathbf{y}^{h-1}) := \mathbf{V}_j(a{\cdot}\mathbf{F}_1(a{\cdot}\mathbf{y}^{h-1}){\cdot}\mathbf{F}_2(a{\cdot}\mathbf{y}^{h-1}){\cdot}\ldots{\cdot}\mathbf{F}_{|\mathcal{G}|}(a{\cdot}\mathbf{y}^{h-1}))$;
7 $\quad$ for each feedback vertex
8 $\quad\quad \mathbf{y}_i^h := \mathbf{F}_j(a{\cdot}\mathbf{y}^{h-1})$; /* where $\mathbf{F}_j$ is the circuit equation of $\mathbf{y}_i$ */
9 **until** $\mathbf{y}^h = \mathbf{y}^{h-1}$;

Note that $\mathbf{F}_j$ only depends on the value of vertices of lower depth, whose values for round h have already been computed when $\mathbf{F}_j$ is evaluated in round h. Malik's test for combinational output-stability is that for every output vertex j_i, $\mathbf{F}_{j_i}(a{\cdot}\mathbf{y}^M) \neq \Phi$, where M is the final value of h in the algorithm.

This algorithm looks almost identical to Algorithm B. However, the test for combinational output-stability given in Theorem 2.24 assumes that the network is *complete*, whereas Malik's algorithm only assumes a feedback-vertex network. Hence, we need to prove that using just a feedback-vertex set as state variables suffices for checking combinational output-stability. To show this, we need to first introduce Brzozowski and Seger's concept of a reduced network.

2.4.2.2 Ternary simulation on reduced networks

Consider a ternary network $\mathbf{N}$ with state variables $\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_m$. $\mathbf{s}_i$ is a *legal reduction variable* if the corresponding excitation function $\mathbf{S}_i$ does not depend on any input excitation function, nor on the value of $\mathbf{s}_i$ itself. Note that this specifically excludes input-delay variables as legal reduction variables. A reduced network $\dot{\mathbf{N}}$ of $\mathbf{N}$ is created by removing a legal reduction variable, and re-expressing the remaining functions in terms of the remaining variables. Without loss of generality, assume that the variable to be removed is $\mathbf{s}_m$.

Definition 2.26 [Brzozowski and Seger] Let $\mathbf{N}$ be a ternary network and $\mathbf{s}_m$ a legal reduction variable. Then the *reduced network* $\dot{\mathbf{N}}$ has the state variables $\dot{\mathbf{s}}_1, \dot{\mathbf{s}}_2, \ldots, \dot{\mathbf{s}}_{m-1}$, excitation functions

$$\dot{\mathbf{S}}_i(a{\cdot}\dot{\mathbf{s}}) = \mathbf{S}_i(a{\cdot}\dot{\mathbf{s}}{\cdot}\mathbf{S}_m(a{\cdot}\dot{\mathbf{s}}{\cdot}\Phi)) \text{ for } 1 \leq i < m,$$

and circuit equations

$$\dot{\mathbf{F}}_i(a{\cdot}\dot{\mathbf{s}}) = \begin{cases} \mathbf{S}_m(a{\cdot}\dot{\mathbf{s}}{\cdot}\Phi)) & \text{if } i = m, \\ \mathbf{F}_i(a{\cdot}\dot{\mathbf{s}}{\cdot}\mathbf{S}_m(a{\cdot}\dot{\mathbf{s}}{\cdot}\Phi)) & \text{otherwise.} \end{cases}$$

Note that when evaluating $\mathbf{S}_m$, the value of $\mathbf{s}_m$ is immaterial, since $\mathbf{S}_m$ is assumed to be independent of $\mathbf{s}_m$. ∎

Brzozowski and Seger state the following theorem, which says that Algorithm B gives the same result on $\mathbf{N}$ and $\dot{\mathbf{N}}$, with respect to the variables present in $\dot{\mathbf{N}}$.

Proposition 2.27 (Brzozowski and Seger) *Suppose vertex m is a legal reduction variable. Assume $a{\cdot}\mathbf{s}$ is a total state of $\mathbf{N}$ such that vertex m is ternary stable, i.e., $\mathbf{s}_m = \mathbf{S}_m(a{\cdot}\mathbf{s})$. Let $\mathbf{t}^B$ be the result of Algorithm B for $\mathbf{N}$, when $\mathbf{N}$ is started in state $a{\cdot}\mathbf{s}$. Similarly, let $\dot{\mathbf{t}}^B$ be the result of Algorithm B for $\dot{\mathbf{N}}$, when $\dot{\mathbf{N}}$ is started in state $a{\cdot}\dot{\mathbf{s}}$, where $\dot{\mathbf{s}}_j = \mathbf{s}_j$, for $1 \leq j \leq m-1$. Then for $1 \leq j \leq m-1$,*

$$\mathbf{t}_j^B = \dot{\mathbf{t}}_j^B.$$

Brzozowski and Seger state and prove a similar result for Algorithm A. Although they do not explicitly give the proof for Proposition 2.27, they state that it is the dual of the proof they give for the result on Algorithm A. For completeness, we present this dual proof in Section 2.7.2.

The next proposition extends Proposition 2.27 to an arbitrary set of legal reduction variables, which has the property that any variable in the set remains a legal reduction variable even after any subset of other variables in the set has been removed.

Proposition 2.28 (Brzozowski and Seger) *Let $a{\cdot}\mathbf{s}$ be a total state of $\mathbf{N}$. Let $R \subseteq \{1, 2, \ldots, m\}$, where $\forall i \in R$, $\mathbf{s}_i$ is a legal reduction variable and $\mathbf{s}_i$ is stable on $a{\cdot}\mathbf{s}$. Let $\dot{\mathbf{N}}$ be the reduced version of $\mathbf{N}$ with respect to the variables R. Then $\forall i \in \{1, 2, \ldots, m\} \backslash R$ (i.e., the remaining variables),*

$$\mathbf{t}_i^B = \dot{\mathbf{t}}_i^B.$$

Proof Proposition 2.27 says the result is true for $|R| = 1$. The result follows by induction on the size of R. ∎

As a corollary to Proposition 2.28, Algorithm B gives the same result when applied to $\mathbf{N}$ and any feedback-vertex network $\dot{\mathbf{N}}$, with respect to the feedback vertices.

The last step in proving the correctness of Malik's algorithm is to extend Proposition 2.28 to the values computed by the circuit equations. The proof of the following result appears in Section 2.7.3.

Proposition 2.29 *Let G be a circuit graph, and let $\mathbf{N}$ be the corresponding complete, ternary network with m state variables. Let $a{\cdot}\mathbf{s}$ be a total state of $\mathbf{N}$. Let $\dot{\mathbf{N}}$ be a reduced version of $\mathbf{N}$ where the eliminated variables are legal reduction variables and are stable on $a{\cdot}\mathbf{s}$. Then for $1 \leq i \leq m$*

$$\mathbf{t}_i^B = \dot{\mathbf{F}}_i(a{\cdot}\dot{\mathbf{t}}^B).$$

That is, the value of state variable $\mathbf{s}_i$ found by Algorithm B on complete network $\mathbf{N}$ is the same as that computed by the corresponding circuit equation evaluated on the result of Algorithm B when applied to the reduced network $\dot{\mathbf{N}}$.

2.4.2.3 Combinational output-stability on feedback-vertex networks

Now we are ready to state the combinational output-stability condition in terms of the result of applying Algorithm B to a feedback-vertex network. Recall that the set $\mathcal{O}$ of output vertices is $\{j_1, j_2, \ldots, j_p\}$.

Theorem 2.30 *Let N be a complete network and $\dot{\mathbf{N}}$ be a ternary, feedback-vertex network of N. Let $\dot{\mathbf{t}}^B$ be the result of applying Algorithm B to $\dot{\mathbf{N}}$ starting from the total state $a{\cdot}\Phi^{k+n}$, where k is the number of feedback vertices and n is the number of inputs. Then N is combinationally output-stable if and only if*

$$\forall a, \dot{\mathbf{F}}_{j_i}(a{\cdot}\dot{\mathbf{t}}^B) \neq \Phi, \text{ for } 1 \leq i \leq p.$$

That is, for every input, the circuit equation for each output has a unique binary value.

Proof By Theorem 2.24, N is combinationally output-stable if and only if

$$\forall a, \text{there exists a unique } d \text{ such that } \mathrm{AlgB}(a, \Phi^m)\downarrow_{\mathcal{O}} = d.$$

Since the result of applying Algorithm B to a given total state is unique, this is true if and only if

$$\forall a, \mathrm{AlgB}(a, \Phi^m)_{j_i} \neq \Phi, \text{ for } 1 \leq i \leq p.$$

Since each state variable is stable on total state $a \cdot \Phi^m$, we can apply Proposition 2.29, where $\mathbf{t}^B = \mathrm{AlgB}(a, \Phi^m)$, to yield

$$\mathrm{AlgB}(a, \Phi^m)_{j_i} = \dot{\mathbf{F}}_{j_i}(a \cdot \dot{\mathbf{t}}^B), \text{ for } 1 \leq i \leq p.$$

This proves the theorem. ∎

As stated, this result requires the presence of the n input-delay state variables. In fact, for the special case where all state variables are initially Φ, the input-delay variables are not needed since in the first round of Algorithm B, the only variables to change value are the input-delay variables: they change from Φ to their corresponding input value. Thus, running Algorithm B on a network with the input-delay variables absent is equivalent to starting Algorithm B from the second round with these variables present.

Theorem 2.30 is stated for Algorithm B, and not for Malik's algorithm. However, Malik's algorithm is exactly Algorithm B on a feedback-vertex network, except that the statement $\mathbf{y}^h := \mathrm{S}(a \cdot \mathbf{y}^{h-1})$ in Algorithm B is replaced with lines 5-8 in Malik's algorithm. To explain this difference, first note that since $\mathbf{y}_i$ is a state variable of the reduced network, the corresponding circuit equation $\dot{\mathbf{F}}_i$ is the same as the excitation function S_i. Thus, the difference reduces to the question of how $\dot{\mathbf{F}}_i$ is evaluated on $a \cdot \mathbf{y}^{h-1}$. Algorithm B does not specify a procedure for this. On the other hand, Malik's algorithm explicitly evaluates $\dot{\mathbf{F}}_i$ on $a \cdot \mathbf{y}^{h-1}$ by simulating the network (i.e., propagating $a \cdot \mathbf{y}^{h-1}$ through the network in topological order). Hence, Malik's algorithm is in fact a specialization of Algorithm B. Thus, Theorem 2.30 proves the correctness of Malik's algorithm for concrete input values.

2.4.2.4 Symbolic version of Malik's algorithm

As presented, Malik's algorithm would have to be executed 2^n times, once for each input combination, to determine if a network is combinationally output-stable. In fact, the algorithm proposed by Malik works on symbolic input values, using BDDs. In effect, all 2^n cases are handled in parallel, with possible sharing of work among the cases.

The conversion from the explicit algorithm to the symbolic algorithm is straightforward. The circuit equations are defined over the circuit inputs. That is, for each valuation

of the inputs, a given circuit equation gives the ternary value for the corresponding vertex. Since the inputs are assumed to be binary valued, the functions to be represented are of the form $f : \{0,1\}^n \rightarrow \{0,1,\Phi\}$. Such functions are in turn represented by a pair of boolean functions (f^1, f^0), where f^1 (resp. f^0) is the characteristic function of the set of inputs for which f evaluates to 1 (resp. 0). The set of inputs for which f is evaluates to Φ is computed as $f^\Phi = \overline{f^1 + f^0}$. The functions f^1 and f^0 are represented by BDDs.

To start the algorithm, each input is initialized to a Boolean symbolic variable, and each circuit equation corresponding to a feedback vertex is initialized to the function Φ. As before, within each round, the gates are visited in topological order. For each gate, the new circuit equation is computed by combining the circuit equations of lower depth according to the Boolean operation implied by the vertex function V. For example, if V is the Boolean conjunction of two vertices represented by the equations g and h, then the new circuit equation for f is given by $f^1 = g^1 \cdot h^1$ and $f^0 = g^0 + h^0$. The algorithm repeats until none of the circuit equations at the feedback vertices change from one round to the next. Convergence is guaranteed within k rounds, where k is the number of feedback vertices. Correctness of the symbolic algorithm follows from the fact that it is just a symbolic implementation of the concrete algorithm.

When the algorithm terminates, the circuit equations $F_{j_1}, F_{j_2}, \ldots, F_{j_p}$ of the outputs are examined. If $F_{j_i}^\Phi \neq 0$ for some $1 \leq i \leq p$, then any satisfying assignment of $F_{j_i}^\Phi$ gives an input valuation for which output i is not combinationally output-stable. If $F_{j_i}^\Phi = 0$ for all $1 \leq i \leq p$, then the network is combinationally output-stable, and $F_{j_i}^1$ gives the Boolean function representing output i. Since $F_{j_i}^1$ is represented as a BDD, which has a trivial transformation to an acyclic, multi-level circuit, then a by-product of the algorithm is an equivalent acyclic implementation of the circuit.

Malik mentions that the test for combinational output-stability can be done with respect to a care set of inputs. Such a set expresses a constraint on the combinations of input values that can occur. If all of the satisfying assignments for $F_{j_i}^\Phi$, for $1 \leq i \leq p$, fall outside of the set of care inputs, then the network is combinationally output-stable.

2.4.3 Proposed refinement to Malik's algorithm

Here we propose a different method from Malik's to compute the ternary-valued circuit equations. The goal is to minimize the number of gate evaluations performed during

ternary simulation. As far as correctness is concerned, the gates of the network can be evaluated in any order. The only requirement is that the process of evaluating gates continues until convergence is reached. On the other hand, the number of evaluations is sensitive to the order.

Example 2.31 [Order of evaluation] Consider the circuit in Figure 2.23 on input 1,1,1. All vertices are initialized to Φ. Say we break the feedback at the outputs of gates 3 and 4, and then evaluate the gates in the order 1, 2, 3, 4. This would require 3 passes to reach convergence. However, if we break the circuit at the output of 2 and use the order 3, 1, 4, 2, we reach convergence in a single pass. ∎

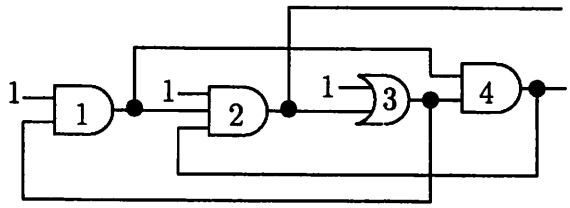


Figure 2.23: Number of gate evaluations depends on evaluation order.

We apply an evaluation ordering scheme by Bourdoncle [25] to heuristically minimize the number of gate evaluations. Bourdoncle's algorithm takes as input a directed graph and produces a *weak topological ordering* (WTO). A WTO can be thought of as a decomposition of a graph into recursive, strongly connected components (SCCs). Consider the graph in Figure 2.24. The set $\{3,4,5,6,7\}$ forms an SCC. By removing vertex 3 from the subgraph induced by this set, we see that $\{5,6\}$ forms an SCC. In this way, SCCs can be nested. The WTO for the example graph is

$$1\ 2\ (\underline{3}\ 4\ (\underline{5}\ 6)\ 7)\ 8.$$

The elements within a matching pair of parentheses constitute a *component*, and the first element of a component is the *head* (heads are underlined above). The *depth*[6] of an element is the number of nested components containing the element (e.g., element 3 has depth 1; 6 has depth 2). The important properties of a WTO are that 1) each component is strongly connected, 2) the set of heads constitutes a feedback-vertex set (that is, all backward edges are incident upon heads), and 3) it gives a total ordering on all the vertices.

[6]This *depth* is different from the one used earlier; the meaning will be clear from the context.

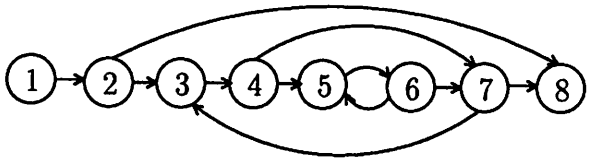


Figure 2.24: A directed graph.

Bourdoncle proposed a gate-evaluation order using a recursive strategy whereby an inner component is stabilized each time one pass is made of its containing component. So in the above example, we first evaluate 1, 2, 3, 4, 5, 6. But then, instead of going to 7, we return to 5, and continue looping between 5 and 6 until there is no change. Then 7 is evaluated, and then we return to 3. The process repeats until the component (3 4 (5 6) 7) is stabilized, and lastly 8 is evaluated. Bourdoncle showed that the total number of evaluations is bounded by $\sum depth(v)$, where the sum is taken over all vertices. This contrasts to the method that Malik uses, which is bounded by $N(k+1)$, where N is the number of gates and k is the number of feedback arcs. It can be shown that $\sum depth(v) < N(k+1)$. However, for a given network, both methods may converge faster than these bounds, and it is possible that Malik's method may converge sooner.

Returning to the problem of combinational output-stability, we compute the circuit equation for each gate by evaluating the gates using Bourdoncle's recursive strategy. In addition to using Bourdoncle's method, we employ event-driven ternary simulation to further reduce the number of evaluations. With this technique, a gate is scheduled for evaluation only if the circuit equation of one of its fanins has changed. Once convergence is reached, we examine the circuit equations as explained above to determine whether the network is combinationally output-stable.

2.5 Sequential output-stability

In this section, we extend the analysis of combinational output-stability to sequences of input vectors. Also, we now allow circuits with explicitly clocked storage devices, called flip-flops. Considering sequences of input vectors is more complicated than the case of single input vectors considered in Section 2.4, but the intuition remains the same: roughly, a network is sequentially output-stable if for every input sequence, there is a unique output sequence. The theory for sequential output-stability does not build upon the

theory of combinational output-stability presented in Section 2.4, but instead starts with the background established in Sections 2.3.1 and 2.3.2.

We start by extending the circuit model to include flip-flops, and discuss the operation of such circuits over multiple clock cycles. Next, we define the transition graph of a network, which captures the cycle-based behavior of a network. We then give a formal definition of sequential output-stability, and present an algorithm to decide this class. If a network is sequentially output-stable, we show how an equivalent Mealy machine can be derived. If a network is not in this class, we show how an error trace can be generated demonstrating an unstable output. Finally, we discuss how information about the environment can be taken into account.

2.5.1 Circuit model and mode of operation

2.5.1.1 Circuit model

The definition of a circuit graph given in Definition 2.1 is extended to include a set of *flip-flop* vertices, and a corresponding set of *flip-flop-delay* vertices. With respect to the combinational part of a circuit, these vertices play the same role as input and input-delay vertices, respectively. Like gate vertices, flip-flop vertices can be driven only by wire vertices.

We change the notation a bit from Section 2.3 to emphasize the sequential operation of a circuit. u is a binary n-tuple giving the values of the input vertices. x is a binary l-tuple giving the values of the flip-flop vertices; a valuation of the flip-flops is called an *external state*. The concatenation of u and x is a *combinational input*, and is referred to by a. Some subset of the input-delay, flip-flop-delay, gate, and wire vertices is chosen as the set of state variables; this subset must obey the minimum requirement that each cycle has a delay. Without loss of generality, and for ease of exposition, we assume that each output and flip-flop input is chosen as a state variable (otherwise, we would have to refer to the circuit equations, rather than directly to the state variables). A value for all the state variables is an *internal state*, and is referred to by b. This binary m-tuple is partitioned into the outputs z, the wires y driving the flip-flop vertices (the next external state), and the remainder of the internal state w. The concatenation of y and z is a *combinational output*. The concatenation of a and b is referred to as a *total state* q. In summary $q = a \cdot b$, where

$a = u \cdot x$ and $b = w \cdot y \cdot z$.[7] Figure 2.25 illustrates the composition of a circuit with flip-flops.

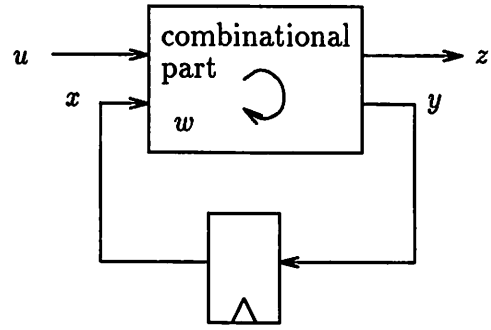


Figure 2.25: The variables of a circuit.

2.5.1.2 Extended fundamental mode of operation

Now we discuss the operation of a circuit over multiple clock cycles. We assume the existence of a global clock that drives all the flip-flops and controls the interaction of the environment with the circuit. At a given clock tick, the environment samples the output value z computed from the previous cycle, provides a new input value u, and causes the flip-flops input value y to be copied to the flip-flops output value x. All of this occurs simultaneously and instantaneously. The values of internal state variables are carried over across clock ticks. In accordance with Theorem 2.12, we assume that the time between clock ticks is at least $(2^m - 2)D$, which allows the combinational part enough time to pass through the transient phase (recognize though that the internal state can be unstable even after the transient phase has passed). Between clock ticks, it is assumed that the input a is held constant. We call this the *extended fundamental mode* of operation. Note that the usual fundamental mode for asynchronous circuits requires the internal state to be stable before the inputs are allowed to change.

A few detailed points about the extended fundamental mode are in order.

1. For a given clock cycle, all changes on the input vector a are assumed to be simultaneous. Nonetheless, since a delay element can be placed on each input wire, a change on an input may be "seen" by the circuit any time within D_i units of the clock tick.

[7] Sometimes we omit the concatenation symbol "$\cdot$". Also, if a reference to a total state includes subscripts or superscripts, then these annotations carry over to the components a, b, u, x, w, y, and z.

Thus, the model effectively accounts for all possible orderings of arrivals on the input vector, at each clock cycle.

2. Each input vertex is assumed to change at most once per clock cycle; that is, glitches on inputs are not allowed. For circuits whose correct operation relies on state-holding elements in the combinational part, this is a reasonable assumption. For example, an RS-latch whose inputs are glitchy will not function properly. If for a particular application, this assumption is not valid, then the constructivity test of Section 2.6 should be used; although it is conservative, it is insensitive to glitches.

   This assumption also implies that the outputs of the flip-flops present in a circuit do not glitch. Again, the constructivity test should be used if this assumption is invalid.

3. It is assumed that unstable internal variables do not change at the same instant that the clock ticks. This assumption is made to ease the exposition; it will be shown that this assumption does not alter the class of sequentially output-stable networks, nor does it affect the equivalent acyclic circuits generated.

2.5.1.3 Operation at power-up

Lastly, we need to discuss the operation of a circuit at power-up. For every flip-flop, it is assumed that a nonempty subset $I \subseteq \mathbb{B}$ is given that specifies the allowable initial values of the flip-flop. At the first clock tick, each flip-flop must take one of its initial values. We assume that at least $(2^m - 2)D$ time passes between power-up and the first clock tick. This gives the circuit enough time to pass through the transient phase. The only assumptions made on the value of $q = uxwyz$ at the moment before the first clock tick is that q is a possible nontransient state and y is consistent with an initial value of the external state. Such total states are called initial states, and are defined as follows (recall, by convention, $q = a \cdot b = ux \cdot wyz$).

Definition 2.32 If N has flip-flops, then q is an *initial state* of network N if y is an external initial state of N and there exists $\tilde{b}$ such that $b \in D(a, \tilde{b})$-nontransient (i.e., the nontransient state b can be reached from some power-up state $\tilde{b}$ when the input is held constant at a). If there are no flip-flops in N, then the set of initial states is exactly the set of nontransient states. The set of initial states of N is denoted *init*. ∎

Example 2.33 [Flip-flop with XOR gate] Consider the network in Figure 2.26 with initial external state $I = \{0\}$ (this network is uninteresting because it lacks combinational cycles, but it suffices to illustrate the definitions). At power-up, all eight binary combinations of values for $ux \cdot y$ are possible. However, only four of these (one corresponding to each input combination ux) are possible nontransient states. And of these four, only $00 \cdot 0$ and $11 \cdot 0$ are consistent with the external initial state $y = 0$. Hence, the set of initial states of the network over $ux \cdot y$ is $\{00 \cdot 0, 11 \cdot 0\}$. ∎

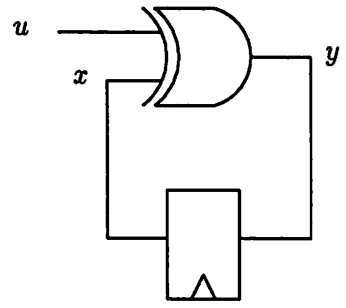


Figure 2.26: A simple network containing a flip-flop and an XOR gate.

The set *init* is partitioned into the set of blocks Δ_I where

- if there are flip-flops in N, then there are as many blocks as there are external initial states, and $q, q' \in$ *init* are in the same block if they have the same y component, and
- if there are no flip-flops in N, then there are $|$*init*$|$ blocks, each containing exactly one initial state.

Each block in Δ_I is called an *initial block*. For Example 2.33, the only initial block is $\{00 \cdot 0, 11 \cdot 0\}$. The intuition is that for a network to be sequentially output-stable, we want all the states within a given initial block to have the same, deterministic behavior.

2.5.1.4 Runs on a network

The sequential operation of a network can be formalized by the notion of a run.

Definition 2.34 A *run* on network N on input sequence $\alpha = u^1, u^2, \ldots$ starting from state q^0 is a sequence $\gamma = q^0, q^1, \ldots$, where

1. $b^{i+1} \in D(a^{i+1}, b^i)$-nontransient for all $i \geq 0$ (i.e., the nontransient internal state b^{i+1} can be reached from b^i when the input is held constant at a^{i+1}), and

2. $x^{i+1} = y^i$ for all $i \geq 0$ (i.e., flip-flop outputs after clock tick $i+1$ are equal to the flip-flop inputs before clock tick $i+1$).

A run is *initialized* if q^0 is an initial state. ∎

Example 2.35 [Flip-flop with XOR gate] Consider the initialized runs for the network in Figure 2.26 on input sequence $\alpha = 1, 1, 0, \ldots$, starting from the external initial state $y^0 = 0$. As mentioned above, the possible values for uxy before the first clock tick are 000 and 110. Upon the first clock tick, x becomes 0 (the external initial state), u becomes 1 (the first input from the environment), and y settles to 1 (the exclusive OR of 0 and 1). After the second clock tick, $uxy = 110$, and after the third, $uxy = 000$. Hence, the initialized runs are $\gamma = 000, 101, 110, 000, \ldots$ and $\gamma = 110, 101, 110, 000, \ldots$. Note that γ has a zeroth component, whereas α does not. ∎

2.5.2 Transition graph of a network

In the next subsection, we define the notion of sequential output-stability directly on a network. However, it helps to first have in mind the concept of a transition graph, which is an abstraction of a network that captures the cycle-based behavior.

Definition 2.36 A *transition graph* $G = (Q, J, T)$ is a directed graph where Q is a finite set of states, $J \subseteq Q$ is a set of initial states, and $T \subseteq Q \times Q$ is a set of directed edges. ∎

The transition graph $G_N = (Q_N, J_N, T_N)$ corresponding to a network N is defined using GMW analysis as follows:

$$\begin{aligned} Q_N &= \{q \mid q = uxwyz \text{ is a valuation for the variables of } N\} \\ J_N &= \{q \mid y \text{ is an external initial state of } N \text{ and } \exists \tilde{b} \text{ s.t. } b \in out(R_a(\tilde{b}))\} \\ T_N &= \{(q, q') \mid x' = y \text{ and } b' \in out(R_{a'}(b))\} \end{aligned}$$

If there are no flip-flops in N, then J_N is exactly the set of outcome states. The concept of initial blocks carries over from networks. Note that the definition of $T_N(q, q')$ is independent of the component $a = ux$ of q. Also note that since every $q \in J_N$ is an outcome state, and for every $(q, q') \in T_N$, q' is an outcome state, then every state reachable from J_N is an outcome state. In the examples of T_N that follow, we typically show just the reachable states.

We can define runs on G_N in the same way we defined runs on N.

Definition 2.37 For a network N, a *run* on transition graph $G_N = (Q_N, J_N, T_N)$ on input sequence $\alpha = u^1, u^2, \ldots$ starting from state q^0 is a sequence $\gamma = q^0, q^1, \ldots$ where $(q^i, q^{i+1}) \in T_N$, for all $i \geq 0$. The run is *initialized* if $q^0 \in J_N$. ■

Proposition 2.38 *A network N and its corresponding transition graph G_N have the same set of runs and initialized runs.*

Proof This is immediate from Theorem 2.15, which states that $out(R_a(b)) = D(a,b)$-nontransient. ■

This result allows us to abstract the details of the UIN delay model and the transient states of a network, in favor of the simpler, cycle-based transition graph.

2.5.2.1 Examples

Before moving on to the definition of sequential output-stability, we illustrate in detail the construction of the transition graph for two networks.

Example 2.39 [Flip-flop with XOR gate] For the network in Figure 2.26, the state space Q_N is all eight binary combinations for values of u, x, and y. As mentioned above, the set of initial states is $J_N = \{00{\cdot}0, 11{\cdot}0\}$. To construct T_N, we must examine the GMW relation, shown in Figure 2.27 (here we show the input value at each state). An edge from $ux{\cdot}y$ to $ux{\cdot}y'$ indicates that starting from internal state y and holding the combinational input constant at ux, the internal state can change in one step to y'. From the GMW relation, it is easy to see that there are four outcome states, 00·0, 01·1, 10·1, and 11·0 (all of which are stable). The outcome states are indicated by heavy ellipses.

Figure 2.28 shows the transition relation T_N over the outcome states. Consider starting from the initial state 00·0. If on the next clock tick the input u remains at 0, then the network remains at state 00·0; hence the self-loop. If the input changes to 1, then the network initially moves to the non-outcome state 10·0 before settling in state 10·1. Hence T_N has an edge from 00·0 to 10·1. Now consider an input change from 1 to 0 at state 10·1. The new value of x is 1 (the old value of y), so the new combinational input is 01, causing an initial change to non-outcome state 01·1. Since this state is stable, the network remains in 01·1; hence the edge from 10·1 to 01·1. Note that in moving from state $ux{\cdot}y$ to $u'x'{\cdot}y'$ in T_N, the new combinational inputs u' and x' are used to compute the new internal state y'. ■

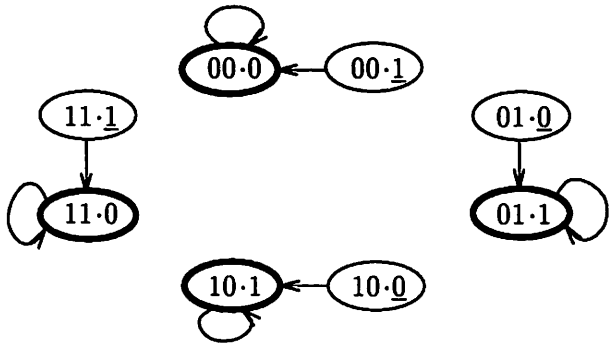


Figure 2.27: The GMW relation over states $ux \cdot y$ for the network of Figure 2.26.

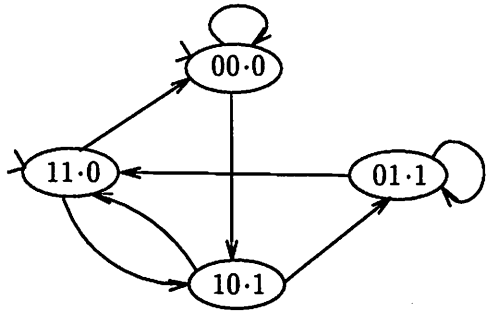


Figure 2.28: The transition graph over states $ux \cdot y$ for the flip-flop with XOR gate.

Example 2.40 [RS-latch] The RS-latch of Figure 2.5 does not have any flip-flops, but we can still analyze the sequential behavior of the network. The transition graph has $2^4 = 16$ states. The GMW relation is shown in Figure 2.29. There are seven outcome states. For example, 00·10 is an outcome state because from 00·11, holding the input constant at 00, the network can be in 00·10 after an unbounded amount of time. Since there are no flip-flops constraining the possible initial states, all seven outcome states are initial states.

Figure 2.30 shows the transition relation T_N. Since there are seven outcome states and four possible values for rs at each clock tick, there are a total of 28 transitions. For example, from 01·10 on input 10, the network initially moves to 10·10, and then passes through 10·00 before reaching the stable state 10·01; hence the edge from 01·10 to 10·01. As another example, from 11·00 on input 00, the network initially moves to 00·00. Holding the input constant at 00, after an unbounded amount of time, the network may be in any of the states 00·00, 00·01, 00·10, or 00·11; hence there are edges from 11·00 to each of these states. On input sequence $10, 11, 00, 11, \ldots$, the possible initialized runs on T_N are

$$q^0, 10{\cdot}01, 11{\cdot}00, q^3, 11{\cdot}00, \ldots$$

where q^0 can be any one of the seven outcome states, and q^3 can be any one of 00·00, 00·01, 00·10, and 00·11. ∎

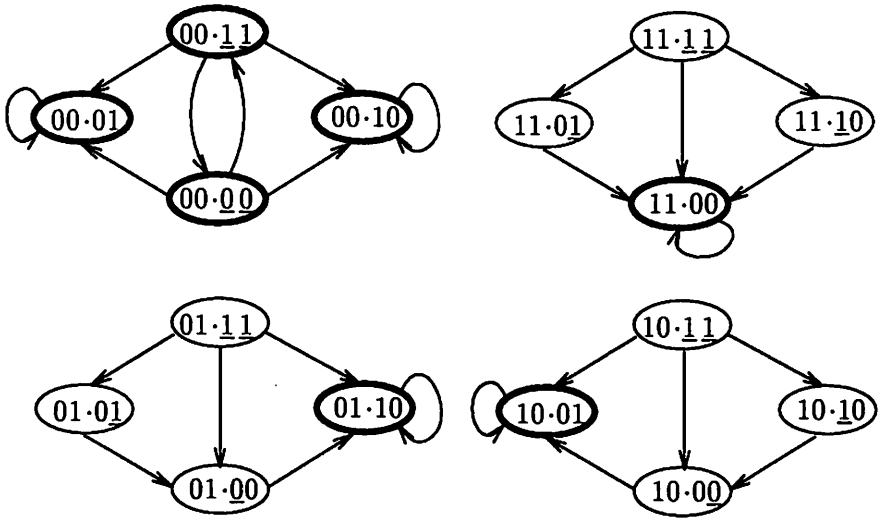


Figure 2.29: The GMW relation over states $rs{\cdot}qz$ for the RS-latch.

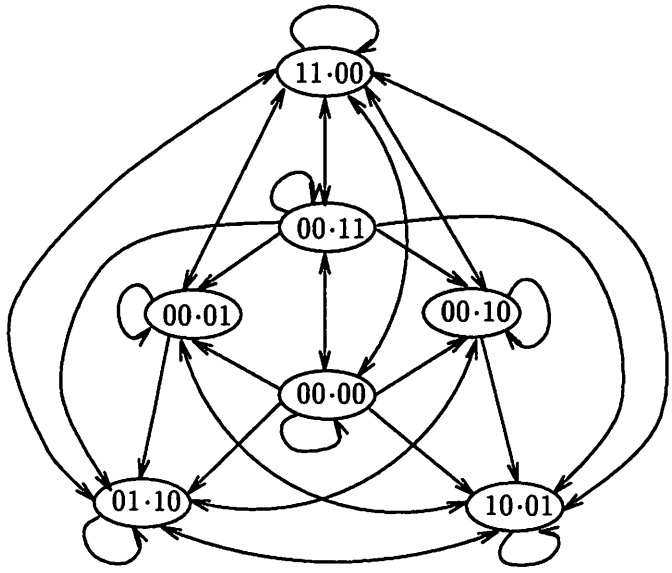


Figure 2.30: The transition graph over states $rs{\cdot}qz$ for the RS-latch.

2.5.3 Definition and properties of sequential output-stability

2.5.3.1 Sequential output-stability

Now that we have described the operation of a network over a sequence of inputs, we are in position to define sequential output-stability. Let $\gamma = q^0, q^1, \ldots$ be a run. Let $\gamma{\downarrow}^1_{\mathcal{O}} = z^1, z^2, \ldots$ be the projection of γ onto the outputs *ignoring* the output z^0 of q^0. This distinction is important because for initialized runs, we do not care about the value of the output before the first clock tick.

Definition 2.41 A network N is *sequentially output-stable* if for every input sequence $\alpha = u^1, u^2, \ldots$ and for every initial block B, there exists an output sequence $\sigma = z^1, z^2, \ldots$ such that for every initialized run γ on α starting from a state in B, $\gamma{\downarrow}^1_{\mathcal{O}} = \sigma$. ∎

In other words, a network is sequentially output-stable, with respect to the UIN delay model and extended fundamental mode of operation, if for every input sequence and initial block, the network produces a unique output sequence starting from any state in that initial block. This condition must be satisfied for all possible delays respecting the bounds of each delay element. There are several points to make regarding this definition.

1. The definition does not constrain the placement of delay elements. However, as with combinational output-stability, the placement of delays can affect the property of se-

quential output-stability. As before, using a complete network is the most conservative approach.

2. There is no stipulation that the projection onto the flip-flop inputs is unique.

3. The definition makes no assumptions about input sequences generated by the environment; we will see later how knowledge of the environment can be taken into account to weaken the definition.

4. By Theorem 2.22, the sequential output-stability property is independent of the delay bounds used.

5. We can make the analogous definition for sequential output-stability of a transition graph. Since the runs of a network N and its transition graph G_N are the same, N is sequentially output-stable if and only if G_N is sequentially output-stable.

Example 2.42 [Flip-flop with XOR gate] The network in Figure 2.26, where we take $z = y$ to be the output, is sequentially output-stable because for every input sequence $\alpha = u^1, u^2, \ldots$ and for the initial block $\{00{\cdot}0, 11{\cdot}0\}$, the unique output sequence is $\sigma = z^1, z^2, \ldots$, where $z^i = u^i \oplus y^{i-1}, \forall i \geq 1$. ∎

Example 2.43 [RS-latch] The RS-latch is not sequentially output-stable on the output q. Consider the input sequence $\alpha = 10, 11, 00, 11, \ldots$. As shown in Example 2.40, both

$$\begin{aligned} \gamma_1 &= 00{\cdot}00, 10{\cdot}01, 11{\cdot}00, 00{\cdot}00, 11{\cdot}00, \ldots, \text{ and} \\ \gamma_2 &= 00{\cdot}00, 10{\cdot}01, 11{\cdot}00, 00{\cdot}10, 11{\cdot}00, \ldots \end{aligned}$$

are possible initialized runs on α. But $\gamma_1 \downarrow_O^1 = 0, 0, 0, 0, \ldots$ and $\gamma_2 \downarrow_O^1 = 0, 0, 1, 0, \ldots$, which are different. This difference occurs because the input 00 follows 11. We will see later that under certain environmental assumptions, the RS-latch is sequentially output-stable. ∎

The property of sequential output-stability is not compositional, in the following sense. Consider a network N composed of two networks N_1 and N_2, where N_1 drives N_2. Even if each of N_1 and N_2 is sequentially output-stable, this does not imply that N itself is sequentially output-stable. The reason is that the outputs of N_1, which are the inputs of N_2, may be glitchy, which violates the assumption that the inputs are glitch-free. However, if the outputs of N_1 pass through glitch-free flip-flops, then we can compose N_1 and N_2. Alternatively, if N_2 passes the constructivity test, then it is safe to drive N_2 by N_1.

2.5.3.2 Properties of sequential output-stability

Checking sequential output-stability is even harder than checking combinational output-stability.

Theorem 2.44 *Deciding if a network is sequentially output-stable is PSPACE-hard.*

Proof The reduction is from single state reachability [26].
Instance: Given an acyclic network N with n inputs, m gates, l flip-flops, an external initial state $x' \in \mathbb{B}^l$ and a state $x'' \in \mathbb{B}^l$.
Question: Is x'' reachable from x'?

Construct the network N' shown in Figure 2.31, which is the network N with an additional output s_5. The logic block $x \equiv x''$ produces a 1 if the current state x is equal to x'', and 0 otherwise. Its size is $O(l)$.

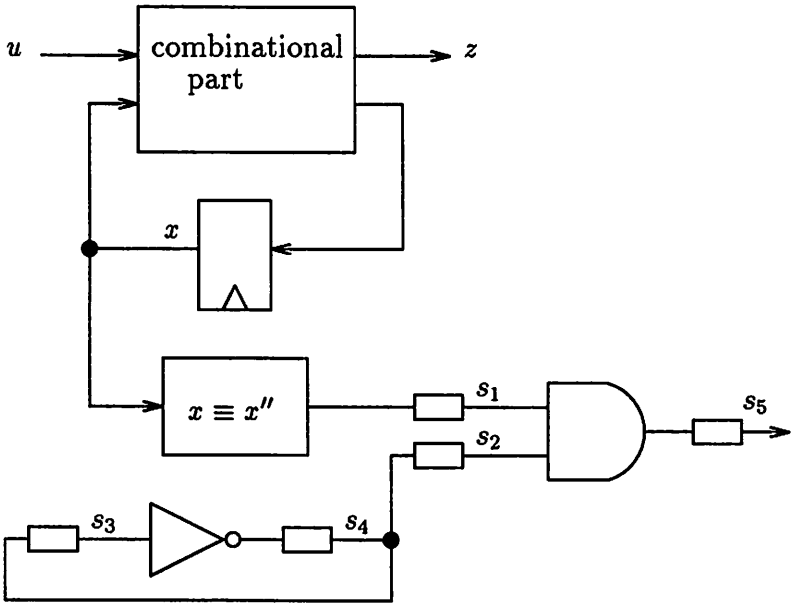


Figure 2.31: Network N', used in the PSPACE-hard proof.

Claim: x'' is reachable from x' if and only if N' is not sequentially output-stable.

First note that since s_1 is driven by acyclic logic, s_1 must stabilize to 0 or 1 during each clock cycle. Now focus attention on the state variables s_2, s_3, and s_4. These three variables participate in a nontransient cycle as illustrated in Figure 2.32. Since these three variables are not influenced by the values of the remaining variables of N', the nontransient

cycle exists independently of the state of the rest of the network. In this cycle, s_2 oscillates. This oscillation is seen at the output of the AND gate s_5 if and only if s_1 is stable at 1.

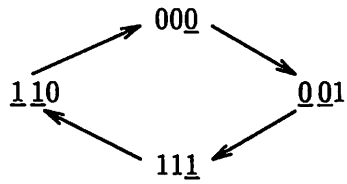


Figure 2.32: Nontransient cycle on variables $s_2 s_3 s_4$.

($\Rightarrow$) Suppose x'' is reachable from x' in the kth step on input α. This implies that in the kth clock cycle, s_1 stabilizes to 1. As argued above, when s_1 is stable at 1, the output s_5 can oscillate. Thus, in the kth clock cycle on input α, s_5 can be either 0 or 1. Hence, N' is not sequentially output-stable.

($\Leftarrow$) Suppose N' is not sequentially output-stable. Since N is acyclic in the combinational part, the outputs of N must be stable in every reachable state of N'. This implies that there exists some input α such that when the network is started in state x', at some clock cycle k, the remaining output s_5 can be either 0 or 1. As argued above, this implies that s_1 is stable at 1 in cycle k, which then implies that x'' is reachable from x'. ∎

As mentioned in Section 2.5.1, we assume that unstable internal variables do not change simultaneously with clock ticks. Call this assumption ASMP. The following theorem demonstrates that this assumption does not alter the definition of sequentially output-stable networks. The intuition is that although removing the assumption adds runs, the additional runs are already "accounted for" by the runs that meet the assumption.

Theorem 2.45 *Network N is not sequentially output-stable with ASMP if and only if N is not sequentially output-stable without ASMP.*

The proof of this theorem appears in Section 2.7.4. We will see later that when a network is sequentially output-stable, we can derive an equivalent acyclic, Mealy machine. In this case, the presence of the extra transitions without ASMP does not affect the derived Mealy machine, because these transitions do not introduce any new I/O behaviors.

2.5.3.3 Combinational output-stability versus sequential output-stability

It is interesting to compare the class of combinationally output-stable networks to the class of sequentially output-stable networks. Combinational output-stability is more

conservative than sequential output-stability, in the following sense.

Proposition 2.46 *Let N be a network with flip-flops. Let N' be the same network, except where the flip-flops are removed and the flip-flop outputs become network inputs, and the flip-flop inputs become network outputs. If N' is combinationally output-stable, then N is sequentially output-stable.*

Proof Let $\alpha = u^1, u^2, \ldots$ be an input sequence for N, and let q and q' be states of some initial block B, where the flip-flop initial state is y^0. Inductively define $y^i z^i$, for $i \geq 1$, as the unique value produced by N' on input $u^i y^{i-1}$. Let γ and γ' be runs on α starting from q and q' respectively. Since q and q' have the same initial flip-flop value and N is combinationally output-stable, then by induction γ and γ' must agree on the y component, and hence must agree on the z component. Thus, $\gamma{\downarrow}^1_O = \gamma'{\downarrow}^1_O$. ∎

The following example shows that combinational output-stability is strictly more conservative than sequential output-stability.

Example 2.47 [Combinational output-stability versus sequential output-stability] Consider the network in Figure 2.33 with state variables w and z, where the flip-flop value is initially 1. This network is not combinationally output-stable because when $x = 0$, z can be either 0 or 1. However, the network is sequentially output-stable, because upon the first clock tick x is stable at 1, which then forces z to be stable at 1. In succeeding clock cycles, regardless of the input value u, a stable 1 is "locked in" at z. Hence, z is in fact logically equivalent to the constant 1. This is clearly seen by examining the transition graph over states $ux{\cdot}wz$ of the network (see Figure 2.34). The initial block is $\{10{\cdot}00, 10{\cdot}01, 10{\cdot}10, 10{\cdot}11, 11{\cdot}11\}$. After the first clock tick, the output z is always 1 in every state of every run. ∎

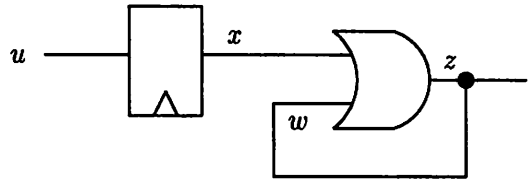


Figure 2.33: A network of a flip-flop with an OR gate. This network is sequentially output-stable, but not combinationally output-stable.

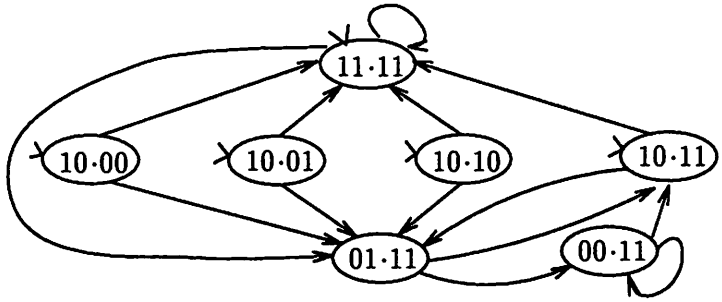


Figure 2.34: Transition graph over states $ux \cdot wz$ of the network in Figure 2.33.

There are three reasons why combinational output-stability is more conservative than sequential output-stability.

1. Combinational output-stability assumes that the internal state variables "forget" their values at each clock tick. This is implicit in the fact that combinational output-stability analysis assumes that the initial internal state is arbitrary. Hence, combinational output-stability ignores the fact that the combinational part can hold state from one clock cycle to the next (in the above example, z remembers if a 1 has been seen at x).

2. Combinational output-stability requires the flip-flop inputs to be stable. This is conservative because nondeterminism in the external state does not necessarily lead to unstable outputs.

3. Combinational output-stability implicitly assumes that all external states are reachable. It is possible that a network is not combinationally output-stable for some external state that, in fact, can never be reached.

In Section 2.6, we extend combinational output-stability to resolve the third limitation. The resulting notion is still conservative with respect to sequential output-stability, but it has significant application in the software domain, and it is insensitive to input glitching.

2.5.4 Bisimulation and the quotient Mealy machine

In the next subsection, we present an algorithm for checking sequential output-stability that takes as input the quotient Mealy machine with respect to bisimulation. In this subsection, we define the bisimulation relation and give the quotient construction.

2.5.4.1 Bisimulation

Bisimulation is an equivalence relation on states [27, p. 88]. This relation is finer than I/O trace equivalence, but coarser than graph isomorphism. We define bisimulation on finite transition graphs as a least fixed point operation.

Definition 2.48 Let N be a network, $G_N = (Q_N, J_N, T_N)$ be its corresponding transition graph, and $p, q \in Q_N$. Then, p and q are bisimilar, denoted $p \simeq^B q$, if $p \simeq_i^B q$ for all $i \geq 0$, where

- $p \simeq_0^B q$ iff $p{\downarrow_O} = q{\downarrow_O}$, and
- $p \simeq_{i+1}^B q$ iff $p \simeq_i^B q$ and for all inputs u,
  - if $(p, p') \in T_N$ where $p'{\downarrow_X} = u$, then there exists q' such that $(q, q') \in T_N$, $q'{\downarrow_X} = u$, and $p' \simeq_i^B q'$, and
  - if $(q, q') \in T_N$ where $q'{\downarrow_X} = u$, then there exists p' such that $(p, p') \in T_N$, $p'{\downarrow_X} = u$, and $p' \simeq_i^B q'$.

■

Of course, since we only deal with finite-state systems, there is guaranteed to be some $k \geq 0$ such that $\simeq_k^B = \simeq_j^B$, for all $j \geq k$. We denote by B an equivalence class of the bisimulation relation.

For a network N, the bisimulation relation can be computed in time $O(|T_N| \cdot \log |Q_N|)$ — linear in the number of edges and logarithmic in the number of states of the corresponding transition graph — using the Paige-Tarjan algorithm for stable partition refinement [28]. Of course, this bound is exponential in the size of N (as would be expected since the problem of testing for sequential output-stability is PSPACE-hard in the size of N).

2.5.4.2 Quotient Mealy machine

Mealy FSMs are defined in Section 1.1.2. For the purpose of this discussion, a Mealy FSM is a 6-tuple $\langle S, I, \Sigma_I, \Sigma_O, O, T \rangle$.

Given a network N, the quotient Mealy FSM induced by bisimulation on N is denoted by $M^B = \langle S^B, I^B, \Sigma_I^B, \Sigma_O^B, O^B, T^B \rangle$ where

$$
\begin{aligned}
S^B &= \{B \mid B \text{ is a bisimulation equivalence class}\},\\
I^B &= \{B \mid \exists q \in B \text{ s.t. } q \in \mathit{init}\},\\
\Sigma_I^B &= \mathbb{B}^n (n \text{ is the number of binary inputs of } N),\\
\Sigma_O^B &= \mathbb{B}^p (p \text{ is the number of binary outputs of } N),\\
O^B &= \{(B, u', z') \mid \exists (q, q') \in T^B \text{ s.t. } q \in B, q'{\downarrow_{\mathcal{X}}} = u', \text{ and } q'{\downarrow_{\mathcal{O}}} = z'\}, \text{ and}\\
T^B &= \{(B, u', B') \mid \exists (q, q') \in T^B \text{ s.t. } q \in B, q' \in B' \text{ and } q'{\downarrow_{\mathcal{X}}} = u'\}.
\end{aligned}
$$

Example 2.49 [Flip-flop with XOR gate] Taking y as the output, the bisimulation equivalence relation is just $\simeq_0^B$. Hence, the equivalence classes are $\{00{\cdot}0, 11{\cdot}0\}$ and $\{01{\cdot}1, 10{\cdot}1\}$. The quotient Mealy FSM is shown in Figure 2.35. For example, the edge $01{\cdot}1 \rightarrow 11{\cdot}0$ in T_N (see Figure 2.28) results in the transition $\{01{\cdot}1, 10{\cdot}1\} \stackrel{1/0}{\rightarrow} \{00{\cdot}0, 11{\cdot}0\}$ in the Mealy machine. ∎

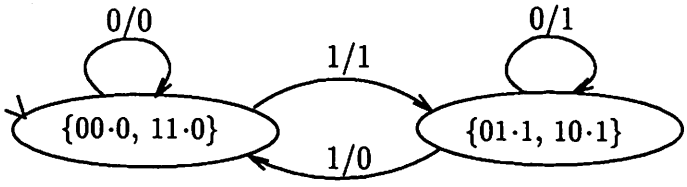


Figure 2.35: Quotient Mealy FSM for the network of the flip-flop and XOR gate.

Example 2.50 [Flip-flop with OR gate] Consider the transition graph of Figure 2.34. The equivalence classes of $\simeq_0^B$ are $\{00{\cdot}11, 01{\cdot}11, 10{\cdot}01, 10{\cdot}11, 11{\cdot}11\}$ and $\{10{\cdot}00, 10{\cdot}10\}$. This is in fact the fixed point because regardless of the input value, each state transitions to a state in the first equivalence class. The quotient machine is shown in Figure 2.36. Note that even though the two states of the quotient generate the same I/O language, they do not form a single equivalence class, because they generate different outputs (i.e., their output components z are different). ∎

Example 2.51 [RS-latch] Considering the transition graph of Figure 2.30, where the output is q, and the initial partition $\simeq_0^B$ is $\{11{\cdot}00, 00{\cdot}01, 00{\cdot}00, 10{\cdot}01\}$ and $\{00{\cdot}11, 00{\cdot}10, 01{\cdot}10\}$.

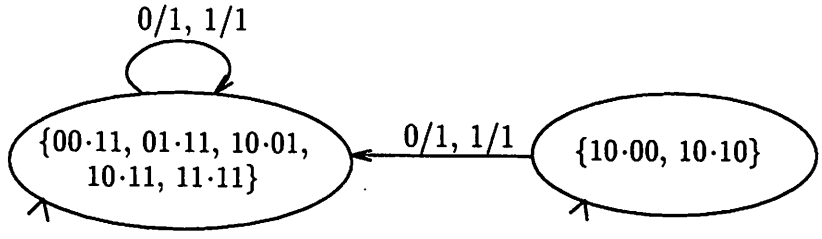


Figure 2.36: Quotient Mealy FSM for the network of the flip-flop and OR gate.

The first equivalence class is subdivided into $\{11{\cdot}00, 00{\cdot}00\}$ and $\{00{\cdot}01, 10{\cdot}01\}$ because on input 00, the states in the former class can move to the state $00{\cdot}10$ where the output is 1, whereas the latter states cannot. Likewise, $\{00{\cdot}11, 00{\cdot}10, 01{\cdot}10\}$ is split into $\{00{\cdot}11\}$ and $\{00{\cdot}10, 01{\cdot}10\}$. The resulting quotient machine is shown in Figure 2.37; all states are initial. ■

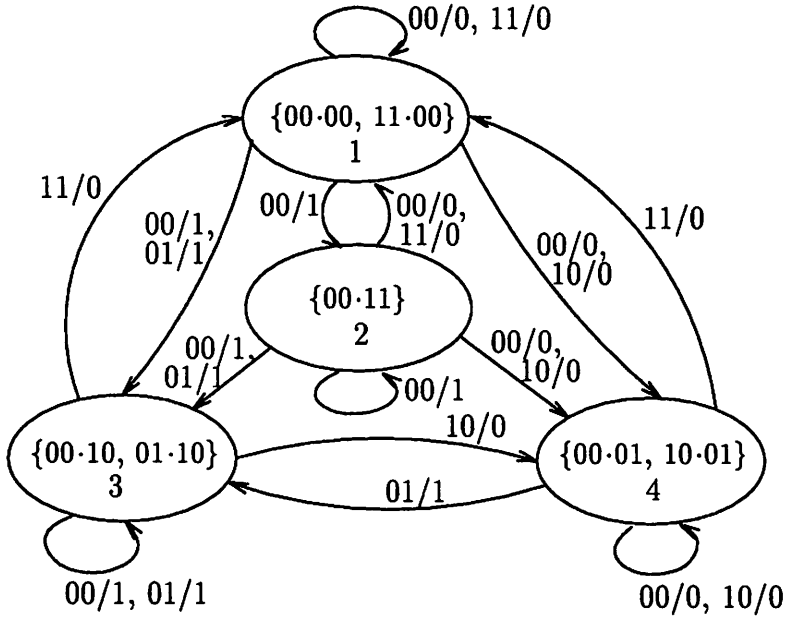


Figure 2.37: Quotient Mealy FSM for the RS-latch.

2.5.4.3 I/O equivalence of transition graph and quotient machine

As we will see, we can test for sequential output-stability directly on the quotient machine, because the transition graph T_N and quotient machine M^B have precisely the same I/O behavior.

Lemma 2.52 *Let N be a network. Let $p, q \in B$, where B is a bisimulation equivalence class. If $(p, p') \in T_N$, where $p' \in B'$, $p'{\downarrow_{\mathcal{X}}} = u'$ and $p'{\downarrow_{\mathcal{O}}} = z'$, then there exists $q' \in B'$ such that $(q, q') \in T_N$, $q'{\downarrow_{\mathcal{X}}} = u'$, and $q'{\downarrow_{\mathcal{O}}} = z'$.*

Proof The statement of the lemma is illustrated in Figure 2.38. By definition of bisimulation, if $p \simeq^B q$ and $(p, p') \in T_N$ where $p'{\downarrow_{\mathcal{X}}} = u'$, then there exists q' such that $(q, q') \in T_N$, $q'{\downarrow_{\mathcal{X}}} = u'$, and $p' \simeq^B q'$. Since $p'{\downarrow_{\mathcal{O}}} = z'$, then $q'{\downarrow_{\mathcal{O}}} = z'$. ∎

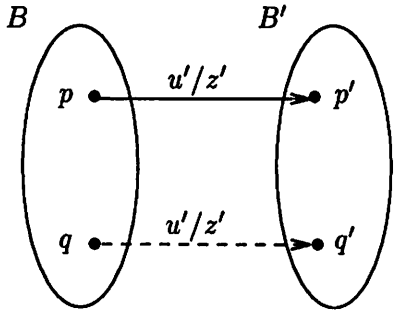


Figure 2.38: The existence of transition (p, p') implies the existence of transition (q, q').

The following theorem states the equivalence of runs on M^B and T_N.

Theorem 2.53 *Let N be a network. Let q^0 be a state and B^0 a block such that $q^0 \in B^0$. Then there exists a run*

$$B^0 \xrightarrow{u^1/z^1} B^1 \xrightarrow{u^2/z^2} B^2 \xrightarrow{u^3/z^3} \ldots$$

in M^B if and only if there exists a run

$$q^0 \rightarrow q^1 \rightarrow q^2 \rightarrow \ldots$$

in T_N, where $q^i \in B^i$, $q^i{\downarrow_{\mathcal{X}}} = u^i$, and $q^i{\downarrow_{\mathcal{O}}} = z^i$, for all $i \geq 1$.

Proof ($\Rightarrow$) The proof is by induction on the length i of the run.

Base $i = 1$: $B^0 \xrightarrow{u^1/z^1} B^1$ implies that there exists p^0, p^1 such that $(p^0, p^1) \in T_N$, $p^0 \in B^0$, $p^1 \in B^1$, $p^1{\downarrow_{\mathcal{X}}} = u^1$, and $p^1{\downarrow_{\mathcal{O}}} = z^1$. By Lemma 2.52, there exists $q^1 \in B^1$ such that $(q^0, q^1) \in T_N$, $q^1{\downarrow_{\mathcal{X}}} = u^1$, and $q^1{\downarrow_{\mathcal{O}}} = z^1$.

I.H.: The hypothesis holds for $k < i$.

I.S.: We need to show how to extend the run in T_N to q^{i+1}. This case is identical to the base case.

($\Leftarrow$) By construction, $(q^i, q^{i+1}) \in T_N$, where $q^i \in B^i$, $q^{i+1} \in B^{i+1}$, $q^{i+1}{\downarrow_{\mathcal{X}}} = u^{i+1}$, and $q^{i+1}{\downarrow_{\mathcal{O}}} = z^{i+1}$, implies $B^i \xrightarrow{u^{i+1}/z^{i+1}} B^{i+1}$ in M^B. ∎

Corollary 2.54 *Let N be a network. Then the initialized I/O runs of M^B and T_N are the same.*

2.5.5 Algorithm for deciding sequential output-stability

In this subsection, we present an algorithm that can determine whether or not a network N is sequentially output-stable. The algorithm takes as input the quotient machine M^B and the set of initial blocks Δ_I corresponding to a network, and returns "sequentially output-stable" or "not sequentially output-stable". The algorithm, named Algorithm C, is shown in Figure 2.39. We assume that the quotient machine has been preprocessed so that every state of M^B is reachable from an initial state.

```
Algorithm C
1     foreach state B of M^B
2       foreach input vector u
3         if there are 2 or more distinct successors of B on u
4           return "not sequentially output-stable"

5     foreach initial block B of Δ_I
6       foreach input vector u
7         S := {B'|B' is a state of M^B and ∃q' ∈ B' s.t. q'↓_X = u and
                                              ∃q ∈ B s.t. (q, q') ∈ T_N}
8         if |S| > 1
9           return "not sequentially output-stable"

10    return "sequentially output-stable"
```

Figure 2.39: The sequential output-stability algorithm.

The test for sequential output-stability simply reduces to checking if the transition relation of M^B is deterministic; that is, if for each input, each state has a unique next state.

As seen in Algorithm C, the test for determinism is broken into two parts. In the first part (lines 1-4) every (reachable) state is checked for determinism for each input valuation.

The second part (lines 5-9) handles a special case. Remember that by the definition of sequential output-stability, there must be a unique output sequence, for each input sequence, starting from any state in a given initial block. This condition is not ensured by part one of the algorithm, because a given initial block is not constrained to lie within a single bisimulation equivalence class. This is obvious because all states of an equivalence class must have the same output component z, whereas the states of an initial block are only required to have the same next state component y. Hence, the second part of the algorithm simply checks that for a given initial block and input, there is at most one successor state M^B on that input. If a network passes both parts of the algorithm, then it is declared "sequentially output-stable".

For a network N, Algorithm C can be executed in time $O(|Q_N|+|T_N|)$. Each edge of the transition graph needs to be examined only a constant number of times.

2.5.5.1 Examples

Example 2.55 [Flip-flop with XOR gate] The Mealy machine in Figure 2.35 passes the first part because there is a unique next state on each input. In this example the only initial block $\{00\cdot 0, 11\cdot 0\}$ corresponds to an equivalence class (because the next state signal is taken as the output), so the first part of the algorithm subsumes the second part. Hence, this network is declared "sequentially output-stable". ∎

Example 2.56 [Flip-flop with OR gate] The Mealy machine in Figure 2.36 clearly passes the first part. By analyzing the Mealy machine, the network in Figure 2.33 is seen to be sequentially output-stable since all transitions of M^B terminate at a single state. ∎

Example 2.57 [RS-latch] The RS-latch fails the first part of the test because state $\{00\cdot 11\}$ has multiple successors on input 00 (see Figure 2.37). Hence, the network is not sequentially output-stable. ∎

2.5.5.2 Proof of correctness of sequential output-stability algorithm

To prove the correctness of Algorithm C, we first define the language of a state. This is just the set of runs from the state, projected onto the inputs and outputs.

$$L(q) = \{u^0z^0, u^1z^1, \ldots \mid \exists \text{ a run } \gamma = q^0, q^1, q^2, \ldots \text{ s.t. } q = q^0\}$$

By identifying states with the same language, we can define the *language equivalence relation.* Namely, $p \simeq^L q$ if $L(p) = L(q)$. It is well known [29] that

1. for deterministic structures, bisimulation and language equivalence coincide, and
2. for nondeterministic structures, bisimulation refines language equivalence (i.e., $p \simeq^B q$ implies $p \simeq^L q$, but the converse is not necessarily true).

We call the language of a state *nondeterministic* if there exists two runs from the state on the same input sequence, which produce different output sequences. The following lemma shows that for nondeterministic transition graphs, if two states are not bisimilar, but are language equivalent, then it must be that the language is nondeterministic. Note that the transition relation T_N of a network N is complete (i.e., for every state q and every input u', there exists a next state q' with input component u'), because the network must do something on each input.

Lemma 2.58 *Let N be a network, and let p and q be states of the corresponding transition graph G_N. If $p \not\simeq^B q$ and $p \simeq^L q$, then $L(p)$ is nondeterministic.*

Proof For sake of contradiction, suppose $L(p)$ is deterministic, and assume $L(p) = L(q)$. We show by induction that $p \simeq_i^B q$ for all $i \geq 0$, and hence that $p \simeq^B q$.
Base: Since p and q each have a unique output, and $L(p) = L(q)$, then the outputs of p and q must be equal. Hence, $p \simeq_0^B q$.
I.H.: If $L(p) = L(q)$ and $L(p)$ is deterministic, then $p \simeq_k^B q$, for all $0 \leq i \leq k$.
I.S.: We show that $p \simeq_{k+1}^B q$. First, $p \simeq_k^B q$ is satisfied by induction. Now, let u be some input, and suppose $(p, p') \in T_N$ where $p'{\downarrow_\mathcal{X}} = u$. Since T_N is complete, then there must exist some q' such that $(q, q') \in T_N$ where $q'{\downarrow_\mathcal{X}} = u$. We want to show that $p' \simeq_k^B q'$. Since $L(p)$ is deterministic, then every successor of p on u must have a deterministic language, and furthermore, all of these successor languages must be the same. Likewise, all of the successors of q on u must have the same deterministic language. Hence, if $L(p') \neq L(q')$,

then this would imply that $L(p) \neq L(q)$, a contradiction. Thus, $L(p') = L(q')$, and $L(p')$ is deterministic. Finally, by induction $p' \simeq_k^B q'$. The reverse condition for bisimulation is shown in a similar fashion. ∎

The following theorem asserts the correctness of Algorithm C.

Theorem 2.59 *Let N be a network. N is not sequentially output-stable if and only if Algorithm C returns "not sequentially output-stable".*

Proof ($\Rightarrow$) Suppose N is not sequentially output-stable. This implies there exists

- an input sequence $\alpha = u^1, u^2, \ldots, u^k$ (where k is less than or equal to the number of states), and
- initialized runs $\gamma = q^0, q^1, \ldots, q^k$ and $\hat{\gamma} = \hat{q}^0, \hat{q}^1, \ldots, \hat{q}^k$ on α in T_N,

such that

- q^0 and $\hat{q}^0$ belong to the same initial block,
- $z^i = \hat{z}^i$ for $1 \leq i < k$, and
- $z^k \neq \hat{z}^k$.

That is, there are runs, on the same input sequence, from two states in the same initial block, which have different output sequences. Clearly $q^k \not\simeq^B \hat{q}^k$, since they have different outputs. Now, either

1. $q^i \simeq^B \hat{q}^i$ for some $0 \leq i < k$, or
2. $q^i \not\simeq^B \hat{q}^i$ for all $0 \leq i < k$.

Suppose that the first case is true, and let j be the largest such i. Since $q^{j+1} \not\simeq^B \hat{q}^{j+1}$, then on input u^{j+1}, the equivalence class containing q^j and $\hat{q}^j$ has more than one successor state in the quotient machine, and hence Algorithm C returns "not sequentially output-stable" on line 4.

Now suppose that the second case is true. Since q^0 and $\hat{q}^0$ (which belong to the same initial block) have successors on u^1 that are not bisimilar, then Algorithm C returns "not sequentially output-stable" on line 9.

($\Leftarrow$) Suppose Algorithm C returns "not sequentially output-stable" on line 4. Then there

exists a state B in M^B that has distinct successors B_1 and B_2 on some input u. Let q_1 be the state in B_1 inducing the transition from B to B_1, and let q_2 be the state in B_2 inducing the transition from B to B_2. Since B is reachable by hypothesis, then by Corollary 2.54, there exists initialized runs γ_1 and γ_2, on the same input sequence, whose last input is u, and that terminate at q_1' and q_2', respectively, where $q_1' \simeq^B q_1$ and $q_2' \simeq^B q_2$. Since $q_1 \not\simeq^B q_2$, and hence $q_1' \not\simeq^B q_2'$, there are two possibilities to consider.

1. $L(q_1') \neq L(q_2')$. Since T_N is complete there exists some input sequence on which q_1' and q_2' generate different output sequences. By extending γ_1 and γ_2 by this input sequence, initialized runs can be produced having different output sequences.

2. $L(q_1') = L(q_2')$. By Lemma 2.58, $L(q_1')$ is nondeterministic. Hence, there exists some input sequence on which q_1' can generate two different output sequences. By extending γ_1 and γ_2 by this input sequence, initialized runs can be produced having different output sequences.

The alternative is that Algorithm C returns "not sequentially output-stable" on line 9. Then for some initial block and some input u, both B_1 and B_2 are possible successors in M^B. We can show that N is sequentially output-stable using the same reasoning as above. ∎

Algorithm C could be applied equally well to the quotient machine formed with respect to language equivalence, rather than bisimulation. However, language equivalence on nondeterministic structures is much harder to compute.

We close this subsection with another example that illustrates Algorithm C and the proof of correctness.

Example 2.60 [Sequential output-stability and runs] The network in Figure 2.40 has internal state variables y and z, and two flip-flops, each initialized to 0. The states of the network are 5-tuples $ux_1x_2 \cdot yz$. There is one initial block $\{000 \cdot 00, 001 \cdot 00, 100 \cdot 00, 101 \cdot 00\}$, corresponding to the one external initial state $x_1y = 00$. Figure 2.41 shows the possible runs of the network on input $\alpha = 1, 0, 1, 1, 1$. This example is interesting because even though the output is functionally determined by the flip-flops ($z = x_1x_2$), z is *not* sequentially output-stable. q_8 and q_9 are not bisimilar because they differ in the output z. Consequently, q_6 and q_7 are not bisimilar, because they have non-bisimilar successors on input $u = 1$. Thus, in the quotient machine, the equivalence class containing q_5 has a nondeterministic successor

on $u = 0$, so the network is declared not sequentially output-stable. This example reinforces the fact that sequential output-stability is a property of the runs of a network, and not a local property of the reachable states. ∎

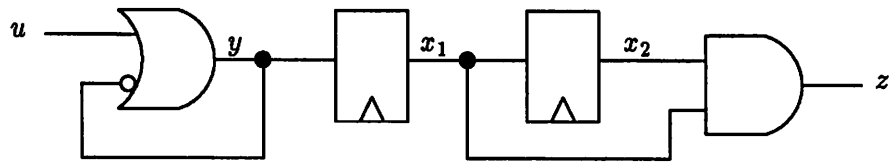


Figure 2.40: Network with external initial state 00 is not sequentially output-stable, even though z is a function of flip-flop outputs.

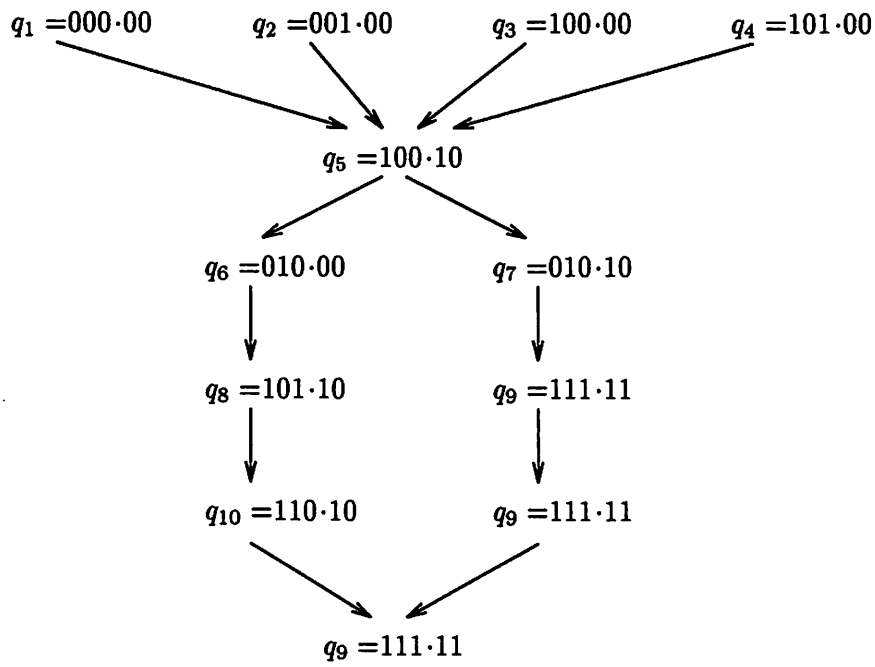


Figure 2.41: Runs of network in Figure 2.40, on sequential input 1, 0, 1, 1, 1.

2.5.6 Extracting an equivalent acyclic implementation

If a network N is sequentially output-stable, then the quotient machine M^B is a deterministic Mealy FSM that has the same I/O behavior as the original circuit. M^B is not state-minimal by construction because bisimularity is *not* an equivalence on languages. That is, it may be that $L(p) = L(q)$ for two states, even though p and q are not bisimilar. Rather than complicating the procedure to produce an equivalence on languages, we just

pass M^B through an FSM state minimizer. Since M^B is deterministic, this can be done in polynomial time.

Example 2.61 [Flip-flop with XOR gate] The quotient machine in Figure 2.35 is in fact state-minimal. The two states of this machine correspond to the two states of the original circuit in Figure 2.26. ▪

Example 2.62 [Flip-flop with OR gate] From Figure 2.36, it can be seen that the two states of the quotient machine can be merged, yielding a one-state FSM that produces 1 on any input. This can be realized by a circuit where the output is driven by a logical 1. ▪

Given a deterministic Mealy FSM M (described for example, by a table of transitions), a synthesis tool (e.g., [30]) can be used to produce a multi-level implementation without combinational cycles. The number of flip-flops in this implementation is not correlated with the number in the original circuit (even excepting the fact that the state encoding method, for example, one-hot versus minimal, influences the number of flip-flops). In general, the implementation may have more flip-flops because the combinational part of the original circuit can store state, whereas the acyclic combinational part of the implementation cannot. However, as seen in Example 2.62, the implementation may actually have fewer flip-flops.

2.5.7 Generating an error trace

If a network is determined to be *not* sequentially output-stable, then we wish to find an input sequence α that violates the condition in Definition 2.41. In addition to α, it is desirable to show two initialized runs γ_1 and γ_2 on α that produce different output sequences. This information can be generated by "unwinding" the fixed point computation for bisimulation.

Algorithm C can return "not sequentially output-stable" by failing the condition on line 3 or line 8. We will concentrate on generating an error trace for the first case; the other case is similar.

If a network fails line 3 of Algorithm C, then we have equivalence classes B, B_1 and B_2, and input u such that B_1 and B_2 are both successors of B on u in M^B. Since B is reachable in M^B, we can produce and initialized run in M^B that terminates at B. By

following the proof of Theorem 2.53, this run can be transformed into an initialized run γ on T_N, having the same I/O.

The run γ can be extended by one state in two different ways. The first is to a state q_1 in B_1, on input u, and the second way is to a state q_2 in B_2, again on input u. Call these extensions γ_1 and γ_2 respectively. We now show how γ_1 and γ_2 can be extended on the same sequence of inputs, to yield different output sequences.

If two states p and q are not bisimilar, then either they have different outputs, or there exists $i > 0$ such that $p \not\simeq^B_{i+1} q$ and $p \simeq^B_i q$. Considering the latter case, this implies that there exists an input u such that

1. there exists p' such that $(p, p') \in T_N$ and $p'{\downarrow_{\mathcal{X}}} = u$, and for all q' such that $(q, q') \in T_N$ and $q'{\downarrow_{\mathcal{X}}} = u$, $p' \not\simeq^B_i q'$, or

2. there exists q' such that $(q, q') \in T_N$ and $q'{\downarrow_{\mathcal{X}}} = u$, and for all p' such that $(p, p') \in T_N$ and $p'{\downarrow_{\mathcal{X}}} = u$, $p' \not\simeq^B_i q'$.

For every pair of states (p, q) that have the same outputs but are not bisimilar, we record during the bisimulation computation, the input u and the next state (p' for condition 1, q' for condition 2) that demonstrates why p and q are not bisimilar.

Returning to the scenario above, we have initialized runs γ_1 and γ_2 leading to q_1 and q_2, respectively, where $q_1 \not\simeq^B q_2$. If q_1 and q_2 have different outputs, then we are done generating the error trace. Otherwise, for some i, $q_1 \not\simeq^B_{i+1} q_2$ and $q_1 \simeq^B_i q_2$, and we recall the input u' and the condition that demonstrates why $q_1 \not\simeq^B_{i+1} q_2$. Suppose that it is condition 1, where the next state q_1' of q_1 on input u' cannot be matched by q_2. Then, we extend γ_1 by q_1', and extend γ_2 by an arbitrary next state q_2' of q_2, on input u'. By hypothesis, $q_1' \not\simeq^B_i q_2'$. Now, either q_1' and q_2' have different outputs, or we again recall the input and condition that demonstrates $q_1' \not\simeq^B_i q_2'$. Each time that we do this, we decrease i by one, so that eventually we must reach a pair of states that have different outputs. At this point, we will have generated initialized runs γ_1 and γ_2, on the same input sequence α, which have different output sequences.

Example 2.63 Consider the partial graph shown in Figure 2.42. Each state is labeled with its input and output component. Suppose q_0 is reachable from an initial state. Even though $L(q_1) = L(q_2)$, $q_1 \not\simeq^B q_2$. These are the equivalence classes at each step of the bisimulation

fixed point computation:

$$\begin{aligned}
\simeq_0^B &= \{\{q_0\},\{q_1,q_2\},\{q_3,q_4,q_5\},\{q_6,q_8\},\{q_7,q_9\}\} \\
\simeq_1^B &= \{\{q_0\},\{q_1,q_2\},\{q_3\},\{q_4\},\{q_5\},\{q_6,q_8\},\{q_7,q_9\}\} \\
\simeq_2^B &= \{\{q_0\},\{q_1\},\{q_2\},\{q_3\},\{q_4\},\{q_5\},\{q_6,q_8\},\{q_7,q_9\}\} \\
\simeq_3^B &= \{\{q_0\},\{q_1\},\{q_2\},\{q_3\},\{q_4\},\{q_5\},\{q_6\},\{q_8\},\{q_7\},\{q_9\}\}
\end{aligned}$$

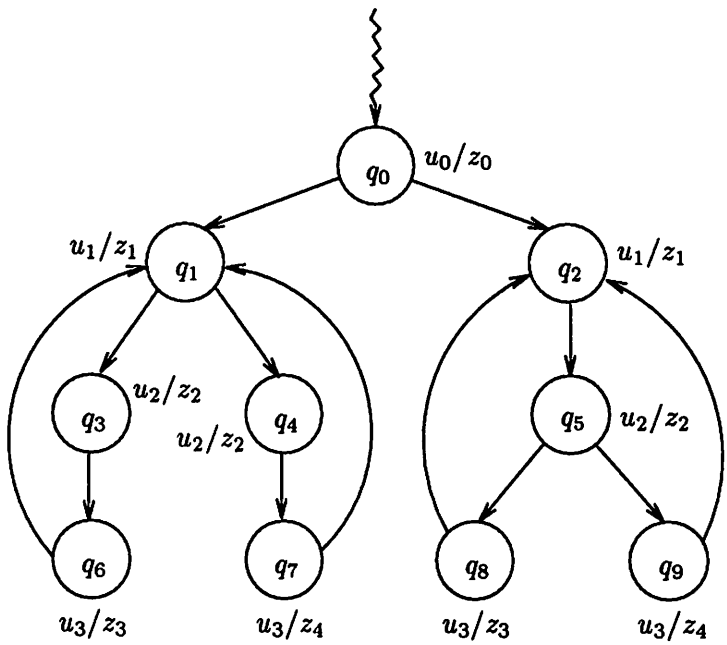


Figure 2.42: Generating an error trace to demonstrate that a network is not sequentially output-stable.

The test for determinism on the quotient machine finds that $\{q_0\}$ has two successors, $\{q_1\}$ and $\{q_2\}$, on input u_1. Thus, we construct runs γ_1 and γ_2 leading to q_1 and q_2, respectively. The states q_1 and q_2 are not $\simeq_2^B$-equivalent because q_1 has a u_2-successor (q_3) that is not $\simeq_1^B$-equivalent to any u_2-successor of q_2. Hence, we extend γ_1 to q_3, and γ_2 to q_5. Now, q_3 and q_5 are not $\simeq_1^B$-equivalent because q_5 has a u_3-successor (q_9) that is not $\simeq_0^B$-equivalent to any u_3-successor of q_3. Extend γ_1 to q_6 and γ_2 to q_9. Finally, q_6 and q_9 are not $\simeq_0^B$-equivalent because they have different outputs. In summary,

$$\begin{aligned}
\gamma_1 &= \ldots q_0 q_1 q_3 q_6, \text{ and} \\
\gamma_2 &= \ldots q_0 q_2 q_5 q_9.
\end{aligned}$$

■

2.5.8 Sequential output-stability in the presence of an environment

The definition of sequential output-stability requires output-stability "for every input sequence $\alpha = u^1, u^2, \ldots$". However, it may be that the environment supplying the inputs only produces a subset of all possible sequences. Thus, we want to redefine sequential output-stability to be with respect to a regular set Θ of input sequences.

An FSM, given at the state transition level, is used to specify Θ. We make two assumptions about this FSM:

1. **The FSM is a Moore machine.** If we were to allow a Mealy machine, then the composition with M^B may not be well-defined because of combinational cycles between M^B and the FSM. If one wanted to model such effects, the FSM could be given at the circuit level, and then the FSM and original circuit could be analyzed together as one circuit, using Algorithm C.

2. **The only inputs to the FSM are from outputs of the circuit under analysis.** Hence, the composition of the FSM and circuit yields a closed system. This assumption is made to ease the presentation.

Let the Moore machine $M_\Theta = \langle S_\Theta, I_\Theta, \Sigma_O, \Sigma_I, O_\Theta, T_\Theta \rangle$ be an environment for network N, where the output alphabet Σ_O of N is the input alphabet of M_Θ, and the input alphabet Σ_I of N is the output alphabet of M_Θ.

The procedure for checking sequential output-stability is modified by taking the product of M_Θ and the quotient Mealy machine M^B to produce a new Mealy machine $\hat{M}_N$, where the transitions and outputs are defined according to the diagram in Figure 2.43. In words, if the present state is $[B, s]$ and s (a state in M_Θ) can produce the output u', then M^B can move to a state B' on input u' and produce output z'. M_Θ then moves along a transition labeled z' to reach the next state s'.

Example 2.64 [RS-latch] An environment M_Θ for the RS-latch is shown in Figure 2.6. Each state is labeled with a single output rs. There is a transition between each pair of states, except from D to A; the absence of this transition prevents the input 00 from following 11. The transitions are not conditional upon the outputs of the RS-latch; they are always enabled. For example, when M_Θ is in state B, it can produce the output 01, and then move to any other state. Every state except A is initial; this prevents 00 as the first input.

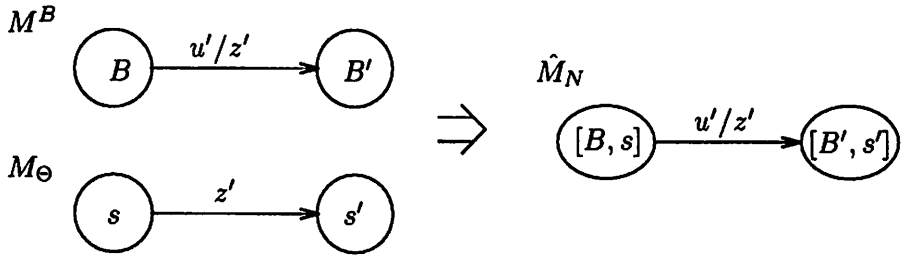


Figure 2.43: Product of edges of quotient machine and environment.

The product of the quotient Mealy machine (Figure 2.37) and the environment yields a Mealy machine with 16 states and 84 transitions. Table 2.1 lists the transitions, where

1. The states 1, 2, 3 and 4 are labels for the quotient states $\{00{\cdot}00, 11{\cdot}00\}$, $\{00{\cdot}11\}$, $\{00{\cdot}10, 01{\cdot}10\}$, and $\{00{\cdot}01, 10{\cdot}01\}$, respectively, and
2. a "-" in a next state label indicates that that component of the state is free to take any value in its domain.

For example, state $4B$ can move to any of the states $3A, 3B, 3C$ or $3D$ on input 01, and produce the output 1.

The initial states and reachable states are marked in the first two columns. The important point is that the problematic states $1A$ and $2A$ are not reachable. The behavior from these two states corresponds to the input 00 at power-up, or the input 00 following 11. ■

To test for sequential output-stability in the presence of an environment, the test for nondeterminism on the original quotient machine is modified. If a state is found in the quotient, for which on a given input there is nondeterministic behavior, then the product machine is checked to see if this state/input combination is reachable. If not, then the observed nondeterminism is ignored.

Example 2.65 [RS-latch] The test for nondeterminism on the quotient machine of Figure 2.37 reveals that states 1 and 2 are nondeterministic on input 00. Input 00 corresponds to state A of the environment. From Table 2.1, we observe that states $1A$ and $2A$ are not reachable. Hence, the RS-latch, in combination with the environment, is declared "sequentially output-stable". ■

initial	reachable	state	next states
		$1A$	1-/0, 2-/1, 3-/1, 4-/0
		$2A$	1-/0, 2-/1, 3-/1, 4-/0
	√	$3A$	3-/1
	√	$4A$	4-/0
√	√	$1B$	3-/1
√	√	$2B$	3-/1
√	√	$3B$	3-/1
√	√	$4B$	3-/1
√	√	$1C$	4-/0
√	√	$2C$	4-/0
√	√	$3C$	4-/0
√	√	$4C$	4-/0
√	√	$1D$	$1\{B,C,D\}/0$
√	√	$2D$	$1\{B,C,D\}/0$
√	√	$3D$	$1\{B,C,D\}/0$
√	√	$4D$	$1\{B,C,D\}/0$

Table 2.1: Transition graph for the RS-latch composed with the environment of Figure 2.6.

In deriving an equivalent acyclic implementation, the unreachable state/input pairs can be used as a source of don't cares to minimize the implementation.

Example 2.66 [RS-latch] Again, refer to the quotient machine of Figure 2.37. For state 1 on input 00, let the next state be 4 and the output be 0. Then states 1 and 4 become equivalent, and can be merged. Likewise, let the next state of 2 on input 00 be 3, with output 1; then states 2 and 3 can be merged. The result is the 2-state FSM shown in Figure 2.7. The left-hand state in the figure remembers that the output q is 0, and the right-hand state remembers 1. Finally, a circuit can be synthesized from this FSM, as shown in Figure 2.8. ∎

2.6 Constructivity

In this section we introduce a third notion of well-behavedness, called constructivity. This name, coined by Berry, is derived from the relationship between constructive logic and this notion of well-behavedness [23]. We will see that constructivity is strictly more con-

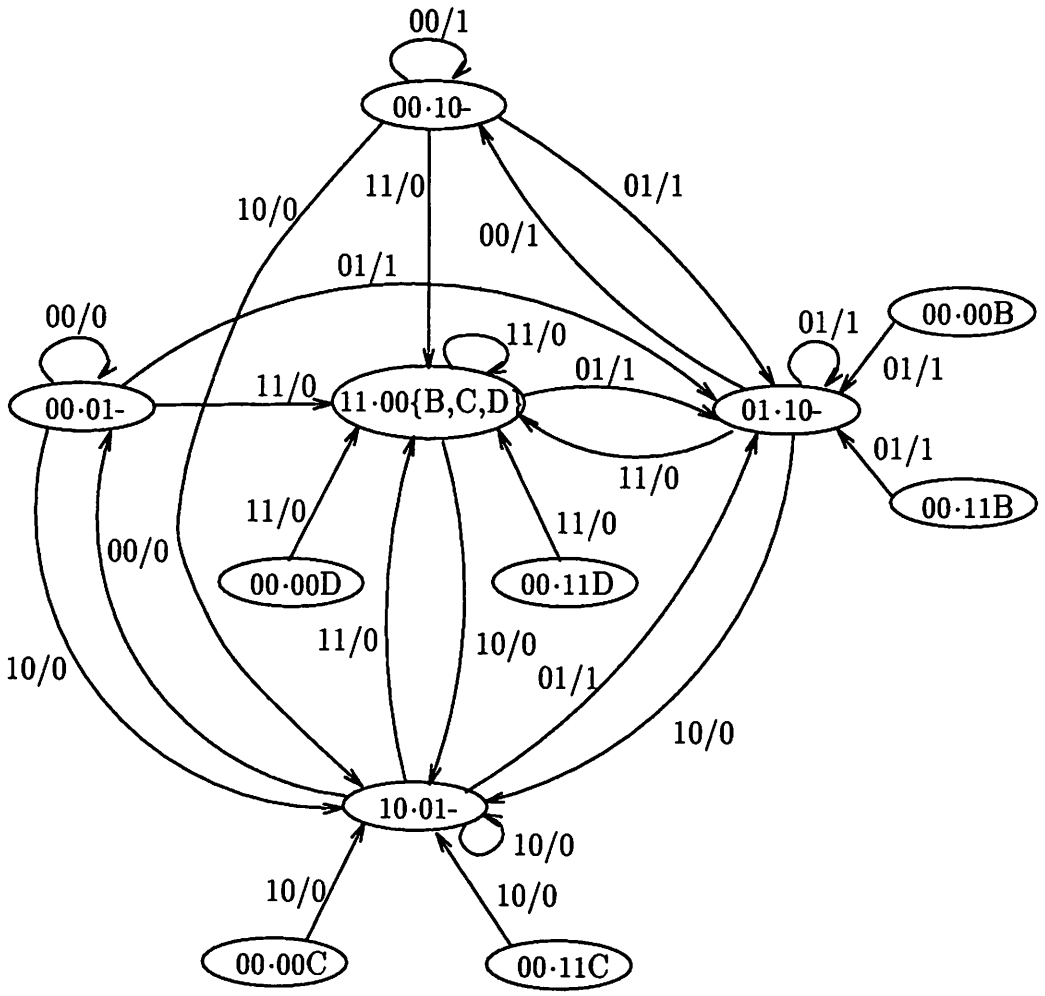


Figure 2.44: Extracted Mealy FSM for the RS-latch composed with an environment.

servative than sequential output-stability, but strictly more permissive than combinational output-stability.

The circuit model and mode of operation used to define constructive networks is identical to that used in Section 2.5 for sequential output-stability, with one major exception. Here, we assume that the internal state is *lost* at each clock tick, whereas for the extended fundamental mode of operation, we assume that the internal state is carried over across clock ticks. We call this the *constructive mode of operation.* Put another way, in the constructive mode, the state of a network is just the value of the flip-flops, whereas for the extended fundamental mode, the state also includes the values of all internal state variables.

One positive consequence of this more conservative operating mode is that it allows us to remove the assumption, made for the extended fundamental mode, that the inputs do not glitch. Hence, a circuit that is well-behaved under the constructive mode of operation is more robust than one well-behaved under the extended fundamental mode, because the former is insensitive to input glitching.

Another difference between constructivity and sequential output-stability is that constructivity requires the flip-flop inputs, as well as the circuit outputs, to be uniquely determined for each input sequence. A consequence of this restriction is that the quotient machine construction of the sequential output-stability test can be replaced by a local check on the reachable states of the transition graph, thus simplifying the test.

The constructive mode of operation is not natural for hardware circuits because combinational wires can in fact store state. Also, the condition that flip-flop inputs must be stable is overly restrictive. Nevertheless, constructivity is useful for hardware because its less aggressive nature allows for more margin of error (e.g., it is insensitive to glitchy inputs), and the algorithm for testing constructivity is more efficient.

The motivating application for constructivity comes from software, in particular from the synchronous language Esterel. The constructive mode is natural in this domain because the state vertices represent the automatic variables (which are initialized on each invocation — they do not remember their previous value) and the flip-flops represent static variables (whose assignment should be unique) This application is thoroughly explored in [23], where Berry defines the *constructive semantics* of pure Esterel. He shows that an Esterel program is constructive (in the sense of these semantics) if and only if the network derived from the program is constructive (in the sense defined here). This "full abstraction theorem" is very powerful because it provides a means of automatically classifying Esterel

programs as legal (i.e., constructive) or illegal. The fact that the theorem connects the abstract world of programs to the (more) concrete world of circuits with delays also points to the universality of constructivity.

2.6.1 Circuit model, mode of operation, and transition graph

The definitions and propositions from Sections 2.5.1 and 2.5.2 for the sequential output-stability case mostly carry over to the constructive case. The similarities and differences are quickly reviewed.

A total state of a network is composed of the same five components:

- u – circuit inputs
- x – flip-flop outputs (present external state)
- w – internal state, less y and z
- y – flip-flop inputs (next external state)
- z – circuit outputs

We require networks to contain at least one flip-flop, because these are the only elements that hold state across clock ticks; it would not make sense to speak of the multi-clock cycle operation of a network with no state elements.

The definition of initial states and initial blocks is identical for the two cases.

As mentioned above, the primary difference is in the mode of operation, and this difference surfaces in the first part of the definition of a run.

Definition 2.67 A *run* on network N (operating in the constructive mode) on input sequence $\alpha = u^1, u^2, \ldots$ starting from state q^0 is a sequence $\gamma = q^0, q^1, \ldots$, where

1. there exists a $\hat{b}_i$ such that $b^{i+1} \in D(a^{i+1}, \hat{b}_i)$-nontransient for all $i \geq 0$, and
2. $x^{i+1} = y^i$ for all $i \geq 0$.

■

Notice that b^{i+1} is completely independent of b^i. The only requirement is that the nontransient state b^{i+1} can be reached from an arbitrary state $\hat{b}$, holding the input constant at a^{i+1}. This weaker condition implies that the set of runs under the constructive mode is a superset of the set of runs under the extended fundamental mode.

The set of runs under the constructive mode is insensitive to the occurrence of glitches on the inputs. Glitches can in fact change the internal state of a network. However, we are already assuming that the internal state is unknown immediately following each clock tick, so glitches cannot make things any worse. The only condition that must be ensured is that enough time is allowed to pass (at least $(2^m-2)D$) after the inputs stabilize, and before the next clock tick occurs. In this case, a^{i+1} in Definition 2.67 is taken as the stabilized value of the vector of inputs.

The definition of the transition graph G_N corresponding to a network N is the same, except that now

$$T_N = \{(q, q') \mid x' = y \text{ and } \exists \hat{b} \in out(R_{a'}(\hat{b}))\}.$$

Example 2.68 [Constructive mode] The transition graph under the constructive mode, for the network of Figure 2.33, is shown in Figure 2.45. The states are valuations of $ux \cdot wy$. The two state-groupings exist solely to simplify the drawing. An edge from a grouping indicates an edge from each state in the grouping; likewise, an incoming edge applies to all states in the grouping.

The one initial block of the network contains the states $10 \cdot 00, 10 \cdot 01, 10 \cdot 10, 10 \cdot 11$, and $11 \cdot 11$. Consider the transitions from $01 \cdot 11$ on input 1. Even though $wz = 11$ in state $01 \cdot 11$, any value for wz is possible in the next state because at each clock tick, the internal state is lost. Thus all states in the top grouping are possible next states of $01 \cdot 11$ on input 1. Compare this to the transition graph in Figure 2.34 for the same network under the extended fundamental mode. In Figure 2.34, from $01 \cdot 11$ on input 1 the only next state is $10 \cdot 11$, because the internal state is preserved across clock ticks. ∎

2.6.2 Definition and properties of constructivity

2.6.2.1 Constructivity

The definition of constructivity is identical to sequential output-stability, except that the set $\mathcal{P}$ of flip-flop inputs must also be stable.

Definition 2.69 A network N is *constructive* (operating in the constructive mode) if for every input sequence $\alpha = u^1, u^2, \ldots$ and for every initial block B, there exists an output sequence $\sigma = z^1, z^2, \ldots$ and a flip-flop sequence $\mu = y^1, y^2, \ldots$ such that for every initialized run γ on α starting from a state in B, $\gamma{\downarrow}^1_{\mathcal{O}} = \sigma$ and $\gamma{\downarrow}^1_{\mathcal{P}} = \mu$. ∎

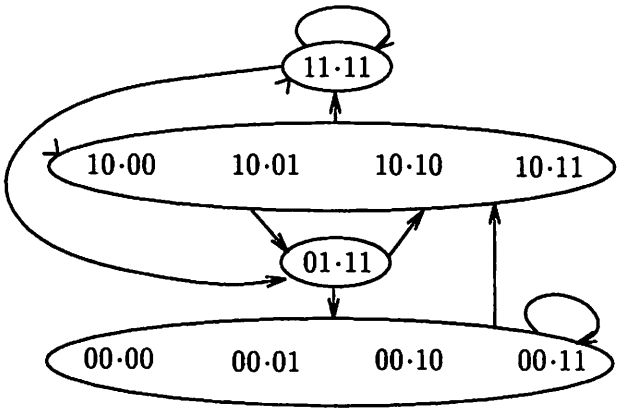


Figure 2.45: Transition graph over states $ux \cdot wz$ of the network in Figure 2.33, under the constructive mode.

As before, the definition itself does not constrain the placement of delay elements, although their placement can affect the property of constructivity. Also, the constructive property is independent of the delay bounds used. Finally, because the constructive mode is insensitive to input glitches, the class of constructive networks is closed under cascade composition. That is, if each of N_1 and N_2 are constructive, then N_1 driving N_2, or N_2 driving N_1, is also constructive. However, connecting N_1 and N_2 in a cycle is not guaranteed to preserve constructivity (consider two NOR gates connected to form an RS-latch).

2.6.2.2 Constructivity versus combinational output-stability and sequential output-stability

Constructivity is more conservative than sequential output-stability for two reasons:

1. the constructive mode allows a superset of runs compared to the extended fundamental mode because the internal state is lost at each clock tick, and

2. the definition of constructivity is stricter than that of sequential output-stability, because of the extra condition on the flip-flop inputs.

The following example shows that this relationship is strict.

Example 2.70 [Sequential output-stability versus constructivity] Consider the network of Figure 2.33. Under the constructive mode on input sequence 1,0,1, both $\sigma_1 = 1,1,0$ and $\sigma_2 = 1,1,1$ are possible output sequences, and hence the network is not constructive. However, as argued in Example 2.47, the network is sequentially output-stable. ∎

Combinational output-stability and constructivity are nearly the same, but combinational output-stability is more conservative because it implicitly assumes that all external states are reachable.

Example 2.71 [Combinational output-stability versus constructivity] Consider the network of Figure 2.46 with initial state 10. The network is not combinationally output-stable because when $x_1x_2 = 11$, then y_2 is unstable. On the other hand, the network is constructive. From Figure 2.47, which shows T_N over the states $x_1x_2 \cdot y_1y_2$, it is evident that starting from $y_1y_2 = 10$, the external state $x_1x_2 = 11$ cannot be reached after the first clock tick. ∎

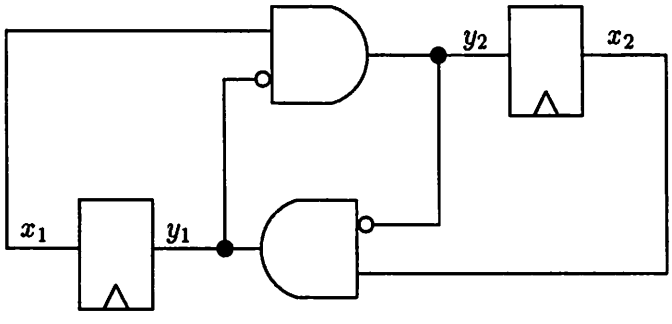


Figure 2.46: Constructive, but not combinationally output-stable.

2.6.2.3 Complexity of deciding constructivity

Theorem 2.72 *Deciding if a network is constructive is PSPACE-hard.*

Proof The same proof as for Theorem 2.44 can be used. The forward implication of the claim holds because if N' is not sequentially output-stable, then it is also not constructive. The backward implication holds because the only oscillation that can arise in N' is the one involving the variables s_2, s_3, and s_4. ∎

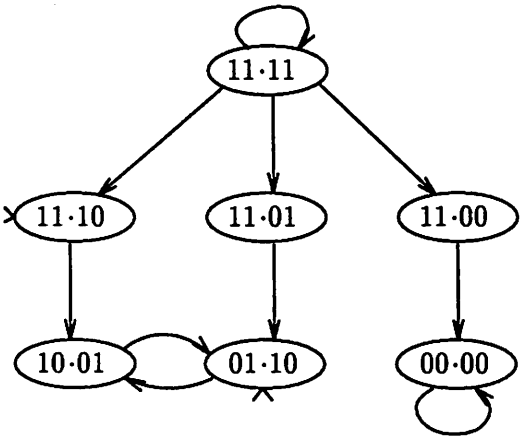


Figure 2.47: Transition graph over states $x_1x_2 \cdot y_1y_2$ for the network of Figure 2.46.

2.6.3 Algorithm for deciding constructivity

The condition for constructivity is just a special case of sequential output-stability, if we include the flip-flop inputs in the circuit output set $\mathcal{O}$.[8] Hence, we could use the sequential output-stability algorithm to decide constructivity. However, we can develop a more efficient algorithm by exploiting the fact that constructivity is a local property on external states, as opposed to the language nature of sequential output-stability.

The algorithm we present for testing constructivity invokes a subroutine for testing combinational output-stability to determine on which combinational inputs a combinational output is unstable. If the network is not complete, then we are obligated to resort to GMW analysis to compute the set of outcome states (refer to Proposition 2.23). However, if the network is complete, then we can leverage the power of ternary simulation. In the interest of efficiency, and since completeness is a reasonable assumption, we assume hereafter that the network is complete.

2.6.3.1 First version of algorithm

A first version of an algorithm for testing constructivity is shown in Figure 2.48. Lines 1-4 compute the set *unstableDomain* of combinational inputs for which a combinational output is unstable. It does this by invoking Algorithm B on each combination ux, with the internal state initialized to Φ^m. Note that initializing the state to Φ^m exactly

[8] Of course, for testing constructivity, we use a different mode of operation, and hence a different transition graph.

corresponds to the assumption that the internal state is lost at each clock tick. If the y or z component is not binary, then ux is added to the *unstableDomain*. Line 5 projects the *unstableDomain* to the flip-flop outputs to yield the set of *unstableStates*.

```
1   unstableDomain := ∅
2   for every ux ∈ B^n × B^l
3     t := AlgB(ux, Φ^m)
4     if t is not binary on yz
5       unstableDomain := unstableDomain ∪ {ux}
6   unstableStates := {x|∃u s.t. ux ∈ unstableDomain}
7   if unstableStates is reachable in ≥ 1 steps from J_N in T_N
8     return "not constructive"
9   else
10    return "constructive"
```

Figure 2.48: Algorithm for deciding constructivity.

From this point, the algorithm is simple. If a state q, where x of q is in *unstableStates*, is reachable from J_N in T_N, then N is not constructive; otherwise, it is constructive. The following argues the correctness of this classification. If a state in *unstableStates* can be reached, then there exists an input $\alpha = u^1, u^2, \ldots, u^k$ and a run $\gamma = q^0, q^1, \ldots, q^k$ on α such that in state q^k, a combinational output is unstable. On the other hand, if *unstableStates* is unreachable, then for every reachable state the combinational outputs are uniquely determined, so the network is constructive. Note that for a constructive network, there cannot be two distinct runs on the same input, but which agree step-by-step on the combinational outputs, because the only state carried across clock ticks is the external state.

2.6.3.2 Modified version of algorithm

The first version of the algorithm requires searching a graph of size 2^{n+l+m}, where n is the number of inputs, l the number of flip-flops, and m the number of input-delay,

flip-flop-delay, gate, and wire vertices. We can do substantially better by searching the projection of this graph onto the external next state component. The projected graph has just 2^l states.

Definition 2.73 Let $G_N = (Q_N, J_N, T_N)$ be the transition graph corresponding to network N. The *reduced transition graph* of G_N is $\breve{G}_N = (\breve{Q}_N, \breve{J}_N, \breve{T}_N)$, where

$$\begin{aligned} \breve{Q}_N &= \mathbb{B}^l \\ \breve{J}_N &= \{y \mid \exists q = uxwyz \in J_N\} \\ \breve{T}_N &= \{(y, y') \mid \exists (q, q') \in T_N\} \end{aligned}$$

■

Example 2.74 [Reduced transition graph] Figure 2.49 shows $\breve{T}_N$ corresponding to T_N of Figure 2.47 and the network of Figure 2.46. For example, the edges $11 \cdot 10 \rightarrow 10 \cdot 01$ and $01 \cdot 10 \rightarrow 10 \cdot 01$ of T_N collapse to the edge $10 \rightarrow 01$ of $\breve{T}_N$. Runs on this graph represent the possible sequences of next external states. ■

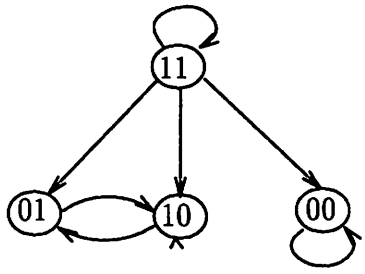


Figure 2.49: Reduced transition graph $\breve{T}_N$, over states $y_1 y_2$, for the network of Figure 2.46.

Example 2.75 Figure 2.50 shows $\breve{T}_N$ corresponding to T_N of Figure 2.45 and the network of Figure 2.33. Since the next state is unconstrained (it is just the input u), $\breve{T}_N$ is complete. ■

The modified algorithm incorporating $\breve{G}_N$ is the same as that of Figure 2.48, except that line 7 is replaced by

7 if *unstableStates* is reachable in ≥ 0 steps from $\breve{J}_N$ in $\breve{T}_N$.

Figure 2.50: Reduced transition graph $\breve{T}_N$, over states u, of Figure 2.45.

Example 2.76 For the network of Figure 2.33, *unstableStates* = $\{0\}$. From $\breve{T}_N$ of Figure 2.50, state 0 is clearly reachable. Recall that this network is sequentially output-stable, because a 1 at z in the first step is "locked in". ∎

Example 2.77 For the network of Figure 2.46, *unstableStates* = $\{11\}$. From $\breve{T}_N$ of Figure 2.49, state 11 is unreachable. Recall that this network is not combinationally output-stable, because combinational output-stability does not take into account reachability. ∎

The complexity of the algorithm is determined by the **for** loop at line 2. There are 2^{n+l} calls to Algorithm B. Using a feedback-vertex set of size k, each call to Algorithm B requires at most k passes, where each pass requires m time (for m total vertices). Thus, the total complexity is $O(km2^{n+l})$.

2.6.3.3 Proof of correctness of the algorithm

Lemma 2.78 *If $q = uxwyz$ is reachable from J_N in T_N, then y is reachable from $\breve{J}_N$ in $\breve{T}_N$.*

Proof Let q be reachable by $\gamma = q^0, q^1, \ldots, q^k$, where $q^0 \in J_N$ and $q^k = q$. Since $(q^i, q^{i+1}) \in T_N$ for $0 \leq i \leq k-1$, then $(y^i, y^{i+1}) \in \breve{T}_N$ for $0 \leq i \leq k-1$. Also, $q^0 \in J_N$ implies $y^0 \in \breve{J}_N$. Hence, $\breve{\gamma} = y^0, y^1, \ldots, y^k$ is a run on $\breve{T}_N$ starting from an initial state y^0 and terminating at $y = y^k$. ∎

Theorem 2.79 *Let N be a complete network. N is not constructive if and only if the algorithm returns "not constructive".*

Proof ($\Rightarrow$) Suppose N is not constructive. This implies there exists

- an input sequence $\alpha = u^1, u^2, \ldots, u^k$ (where k is greater than zero and less than or equal to the number of states), and
- initialized runs $\gamma = q^0, q^1, \ldots, q^k$ and $\hat{\gamma} = \hat{q}^0, \hat{q}^1, \ldots, \hat{q}^k$ on α in T_N,

such that

- q^0 and $\hat{q}^0$ belong to the same initial block,
- $y^i z^i = \hat{y}^i \hat{z}^i$ for $1 \leq i \leq k-1$, and
- $y^k z^k \neq \hat{y}^k \hat{z}^k$.

Since $y^k z^k \neq \hat{y}^k \hat{z}^k$, but $u^k x^k = \hat{u}^k \hat{x}^k$ (since $y^{k-1} = \hat{y}^{k-1}$), then AlgB$(u^k x^k, \Phi^m)$ is not binary on yz. This implies that $u^k x^k \in$ *unstableDomain*, which in turn implies that $x^k = y^{k-1} \in$ *unstableStates*. By Lemma 2.78, since q^{k-1} is reachable in T_N, then y^{k-1} is reachable in $\breve{T}_N$. Hence, the algorithm will return "not constructive" upon reaching y^{k-1}.
($\Leftarrow$) Suppose the algorithm returns "not constructive". This implies there exists $\breve{\gamma} = y^0, y^1, \ldots, y^k$ in $\breve{T}_N$, where $k \geq 0$, $y^0 \in \breve{J}_N$, and $y^k \in$ *unstableStates*. This implies that there exists u such that $uy^k \in$ *unstableDomain*, which in turn implies that AlgB(uy^k, Φ^m) is not binary on yz. This and the existence of $\breve{\gamma}$ implies the existence of two runs

$$\begin{aligned} \gamma &= q^0, q^1, \ldots, q^k, q^{k+1}, \text{ and} \\ \hat{\gamma} &= q^0, q^1, \ldots, q^k, \hat{q}^{k+1} \end{aligned}$$

in T_N such that

- $q^0 \in J_N$,
- $q^i \downarrow_{\mathcal{P}} = y^i$ for $0 \leq i \leq k$,
- $u = u^{k+1} = \hat{u}^{k+1}$, and
- $y^{k+1} z^{k+1} \neq \hat{y}^{k+1} \hat{z}^{k+1}$.

Thus, there exist two runs γ and $\hat{\gamma}$ on $\alpha = u^0, u^1, \ldots, u^k, u^{k+1}$ such that $\gamma \downarrow^1_{\mathcal{O}} \neq \hat{\gamma} \downarrow^1_{\mathcal{O}}$ or $\gamma \downarrow^1_{\mathcal{P}} \neq \hat{\gamma} \downarrow^1_{\mathcal{P}}$, and hence N is not constructive. ∎

2.6.3.4 Symbolic version of algorithm

We can devise a BDD-based implementation of the above algorithm. First, we use Malik's algorithm to implicitly test combinational output-stability for each of the 2^{n+l} combinational inputs. A by-product of this algorithm is a pair of Boolean functions $F_i^1(u,x)$ and $F_i^0(u,x)$ for each combinational output i. Then,

$$unstableDomain(u,x) = \sum_i F_i^{\Phi}(u,x) = \sum_i \overline{(F_i^1(u,x) + F_i^0(u,x))}.$$

The set *unstableStates* is computed using existential quantification:

$$unstableStates(x) = \exists u \; unstableDomain(u,x).$$

The next step is to perform symbolic reachability analysis. We would like to derive a next state *function* for each flip-flop, so that we have the flexibility of employing reachability methods that exploit the determinism of functions (as opposed to the nondeterminism of relations). However, in general, the next states are not functionally determined. For example, in Figure 2.49, state 11 has four possible next states. Nonetheless, there is a way around this contradiction: as long as we limit ourselves to the *stableStates* (the complement of *unstableStates*), then $F^1(u,x)$ gives the correct value for the next state. Thus, we use the function $F^1(u,x)$ corresponding to each flip-flop as the next state function.

Reachability then works as follows. Before each BFS step of reachability, the set of states to be explored is intersected with the set of *unstableStates*. If this intersection is non-empty, then "not constructive" is returned. Otherwise, it is safe to perform the next reachability step. If the fixed point is reached, then "constructive" is returned.

Example 2.80 For the network of Figure 2.46, we use as the next state functions $F_1^1 = \overline{x_1}x_2$ and $F_2^1 = x_1\overline{x_2}$. These functions give the correct values on the *stableStates* $= \{00, 01, 10\}$:

$$\begin{aligned} F_1^1(00)F_2^1(00) &= 00, \\ F_1^1(01)F_2^1(01) &= 10, \text{ and} \\ F_1^1(10)F_2^1(10) &= 01. \end{aligned}$$

■

2.6.4 Extracting an equivalent acyclic implementation

If a network is constructive, then the network never enters the *unstableDomain*. Hence, the positive components F^1 of each output and flip-flop input gives the corresponding Boolean function that needs to be implemented. Since we represent these Boolean functions by BDDs, we can easily translate these BDDs to multi-level circuits.

An important point is that the new circuit contains the same set of flip-flops and uses the same state encoding; the only difference is that the cyclic combinational part has been replaced with an equivalent acyclic implementation. Contrast this to the procedure for sequentially output-stable networks, where the new circuit may have a different number of flip-flops; this is inherent in the fact that in the extended fundamental mode, the combinational part can store state.

2.6.5 Generating an error trace

We wish to generate an input sequence demonstrating why a network is not constructive. If a network is not constructive, then reachability analysis provides a reachable unstable state. We simply work backwards starting from the unstable state to find a sequence of inputs back to an initial state. The process of generating an error trace is the same as that used in formal verification tools [31].

2.6.6 Constructivity in the presence of an environment

To test constructivity for a network N with respect to an environment M_Θ, the reduced transition graph $\check{G}_N$ must be derived from the product of M_Θ and G_N. M_Θ serves to restrict the runs of G_N.

2.7 Proofs

2.7.1 Proof of Theorem 2.24

In this section, we prove that the three statements of Theorem 2.24 are equivalent. Proposition 2.81 proves the equivalence of statements 1 and 2 (where Proposition 2.23 is used for statement 1), and Proposition 2.84 proves the equivalence of statements 2 and 3. Some intermediate lemmas are needed as well to prove the result.

Proposition 2.81 *For complete network N,*

$$\forall a, \textit{there exists a unique } d \textit{ such that } \forall b \in \mathbb{B}^m, \forall b' \in \textit{out}(R_a(b)), b'{\downarrow_{\mathcal{O}}} = d$$

if and only if

$$\forall a, \textit{there exists a unique } d \textit{ such that } \forall b \in \mathbb{B}^m, \mathbf{t}^{\mathbf{B}}_{a,b}{\downarrow_{\mathcal{O}}} = d.$$

Proof We first prove the forward implication of the proposition, except for the uniqueness claim. Let $a \in \mathbb{B}^n$. Choose d to satisfy the hypothesis. Let $b \in \mathbb{B}^m$. By Theorem 2.19, $\mathbf{t}^B_{a,b} =$ *lub out*$(R_a(b))$. Thus $\mathbf{t}^B_{a,b}{\downarrow_{\mathcal{O}}} = (\textit{lub out}(R_a(b))){\downarrow_{\mathcal{O}}}$. By hypothesis, $\forall b' \in \textit{out}(R_a(b)), b'{\downarrow_{\mathcal{O}}} = d$. Hence, $(\textit{lub out}(R_a(b))){\downarrow_{\mathcal{O}}} = d$.

We now prove the reverse implication of the proposition, except for the uniqueness claim. Let $a \in \mathbb{B}^n$. Choose d to satisfy the hypothesis. Let $b \in \mathbb{B}^m$. By Theorem 2.19, $\mathbf{t}^B_{a,b} =$ *lub out*$(R_a(b))$, and thus $d = \mathbf{t}^B_{a,b}{\downarrow_{\mathcal{O}}} = (\textit{lub out}(R_a(b))){\downarrow_{\mathcal{O}}}$. Since d is binary, this implies that $\forall b' \in \textit{out}(R_a(b)), b'{\downarrow_{\mathcal{O}}} = d$.

To show uniqueness in the forward implication, we simply argue by contradiction using the construction given for proving existence in the reverse implication. To show uniqueness in the reverse implication, we do the opposite. ∎

Lemma 2.82 is the key to proving the completeness of statement 3 of Theorem 2.24.

Lemma 2.82 *Let N be a complete network. Then $\forall a$, $\exists b$ such that $AlgA(a, b) = \Phi^m$.*

Proof Let $a \in \mathbb{B}^n$. Recall in Algorithm A that

$$\begin{aligned} \mathbf{s}_i^0 &:= b_i, \text{ and} \\ \mathbf{s}_i^h &:= lub\{\mathbf{s}_i^{h-1}, \mathbf{S}_i(a{\cdot}\mathbf{s}^{h-1})\}. \end{aligned}$$

Construct b as follows. Consider first an input-delay vertex labeled s_i that is driven by input X_j. Let $b_i = \overline{a_j}$. Since $\mathbf{S}_i = X_j = a_j$, then:

$$\begin{aligned} \mathbf{s}_i^0 &= b_i = \overline{a_j}, \text{ and} \\ \mathbf{s}_i^1 &= lub\{\mathbf{s}_i^0, \mathbf{S}_i(a{\cdot}\mathbf{s}^0)\} = lub\{\overline{a_j}, a_j\} = \Phi. \end{aligned}$$

Hence, after the first iteration, each input-delay variable is set to Φ.

Next consider a gate variable s_i driven by wire variables $w_1, w_2, \ldots, w_k$. Choose an arbitrary initial assignment for $w_1, w_2, \ldots, w_k$, say $b_1, b_2, \ldots, b_k$. (Note that w_j, for

$1 \leq j \leq k$, only fans out to gate s_i since N is complete, so no other gate is constraining the value of w_j.) If $\mathrm{S}_i(a{\cdot}b) = \alpha$, then let $b_i = \overline{\alpha}$. Then,

$$\begin{aligned} \mathbf{s}_i^0 &= b_i = \overline{\alpha}, \text{ and} \\ \mathbf{s}_i^1 &= lub\{\mathbf{s}_i^0, \mathrm{S}_i(a{\cdot}\mathbf{s}^0)\} = lub\{\overline{\alpha}, \alpha\} = \Phi. \end{aligned}$$

Hence, after the first iteration, each gate variable is set to Φ.

At this point, we have constructed the initial value b_i for each variable, but we have not shown that the wire variables are forced to Φ in Algorithm A. Consider a wire variable s_i driven by gate variable s_j. Then,

$$\mathbf{s}_i^2 = lub\{\mathbf{s}_i^1, \mathrm{S}_i(a{\cdot}\mathbf{s}^1)\} = lub\{\mathbf{s}_i^1, \mathbf{s}_j^1\} = lub\{\mathbf{s}_i^1, \Phi\} = \Phi.$$

That is, after iteration 1, every gate is driven to Φ, so that after iteration 2, every wire variable is guaranteed to be driven to Φ. ∎

Lemma 2.83 *In Algorithm B, let* $\mathbf{t}_1^0$ *and* $\mathbf{t}_2^0$ *be two different starting points. If* $\mathbf{t}_1^0 \sqsubseteq \mathbf{t}_2^0$, *then* $\mathbf{t}_1^h \sqsubseteq \mathbf{t}_2^h$, *for all* $h \geq 0$, *where* h *is the iteration number in Algorithm B. That is, Algorithm B is monotonic.*

Proof By induction on h. The basis is provided by the hypothesis. Suppose $\mathbf{t}_1^h \sqsubseteq \mathbf{t}_2^h$. Then $\mathbf{t}_1^{h+1} = \mathrm{S}(a{\cdot}\mathbf{t}_1^h) \sqsubseteq \mathrm{S}(a{\cdot}\mathbf{t}_2^h) = \mathbf{t}_2^{h+1}$, where the inequality follows by the monotonicity of S. (Note that Algorithm B is not guaranteed to converge for an arbitrary $\mathbf{t}_1^0$ or $\mathbf{t}_2^0$, because neither $\mathbf{t}_1^0$ nor $\mathbf{t}_2^0$ is assumed to be ternary stable.) ∎

Proposition 2.84 *For complete network* N,

$$\forall a, \textit{there exists a unique } d \textit{ such that } \forall b \in \mathbb{B}^m, \mathbf{t}_{a,b}^{\mathrm{B}}{\downarrow_O} = d$$

if and only if

$$\forall a, \textit{there exists a unique } d \textit{ such that } AlgB(a, \Phi^m){\downarrow_O} = d.$$

Proof We first prove the forward implication of the proposition, except for the uniqueness claim. Let $a \in \mathbb{B}^n$. Choose d to satisfy the hypothesis. Hence, $\forall b \in \mathbb{B}^m, \mathbf{t}_{a,b}^{B}{\downarrow_O} = d$. Choose $\tilde{b}$ to satisfy Lemma 2.82. Thus, $\mathrm{AlgA}(a, \tilde{b}) = \Phi^m$. Hence, $d = \mathbf{t}_{a,\tilde{b}}^{B}{\downarrow_O} = \mathrm{AlgB}(a, \mathrm{AlgA}(a, \tilde{b})){\downarrow_O} = \mathrm{AlgB}(a, \Phi^m){\downarrow_O}$.

We now prove the reverse implication of the proposition, except for the uniqueness claim. Let $a \in \mathbb{B}^n$. Choose d to satisfy the hypothesis. Hence, $\text{AlgB}(a, \Phi^m)\downarrow_{\mathcal{O}} = d$ (Φ^m is ternary stable, so Algorithm B is guaranteed to converge on input a, Φ^m). By Lemma 2.83, $\forall \mathbf{t} \sqsubseteq \Phi^m$, $\text{AlgB}(a, \mathbf{t}) \sqsubseteq \text{AlgB}(a, \Phi)$, which implies $\text{AlgB}(a, \mathbf{t})\downarrow_{\mathcal{O}} \sqsubseteq \text{AlgB}(a, \Phi)\downarrow_{\mathcal{O}} = d$, which implies $\text{AlgB}(a, \mathbf{t})\downarrow_{\mathcal{O}} = d$ since d is binary. Let $b \in \mathbb{B}^m$. Since $\text{AlgA}(a, b) \sqsubseteq \Phi^m$ (trivially), then $\text{AlgB}(a, \text{AlgA}(a, b))\downarrow_{\mathcal{O}} = \mathbf{t}^B_{a,b}\downarrow_{\mathcal{O}} = d$.

To show uniqueness in the forward implication, we simply argue by contradiction using the construction given for proving existence in the reverse implication. To show uniqueness in the reverse implication, we do the opposite. ∎

2.7.2 Proof of Proposition 2.27

Proof Consider Algorithm B for N. If the excitation of $\mathbf{s}_m$ never changes, i.e., if $\mathbf{S}_m(a \cdot \mathbf{t}^i) = \mathbf{t}^i_m = \mathbf{s}_m$, for $0 \leq i \leq B$, then the proposition holds trivially (because the successors of $\mathbf{s}_m$ cannot see a difference). Hence, assume $\mathbf{S}_m(a \cdot \mathbf{t}^i)$ changes for the first time at step $r > 0$. From the monotonicity of Algorithm B, the fact that $\mathbf{S}_m$ does not depend on any input excitations, and the assumption that $\mathbf{s}_m$ was stable in total state $a \cdot \mathbf{s}$, we can conclude that

$$\mathbf{S}_m(a \cdot \mathbf{t}^i) = \begin{cases} \mathbf{s}_m = \Phi & \text{if } i < r, \\ \alpha & \text{if } i \geq r, \text{ where } \alpha \in B. \end{cases}$$

From this and the definition of Algorithm B, it follows that

$$\mathbf{t}^i = \begin{cases} \Phi & \text{if } i < r+1, \\ \alpha & \text{if } i \geq r+1. \end{cases}$$

Clearly, $\mathbf{t}^i_j = \dot{\mathbf{t}}^i_j$ for $1 \leq j \leq m-1$ and $1 \leq i \leq r$, because up until and including round r, there is no difference between $\mathbf{S}_m$ and $\mathbf{t}_m$. Since $\mathbf{S}_m(a \cdot \mathbf{t})$ does not depend on $\mathbf{t}^m$ (by definition of a legal reduction variable), it follows that for $i \leq r$ we have

$$\mathbf{S}_m(a \cdot \dot{\mathbf{t}}^i \cdot \alpha) = \mathbf{S}_m(a \cdot \dot{\mathbf{t}}^i \cdot \Phi) = \mathbf{S}_m(a \cdot \mathbf{t}^i) = \begin{cases} \Phi & \text{if } i < r, \\ \alpha & \text{if } i = r. \end{cases}$$

However, by the monotonicity of Algorithm B, $\dot{\mathbf{t}}^r \sqsupseteq \dot{\mathbf{t}}^i$ for $i \geq r$. Since $\mathbf{S}_m$ is monotone, it follows that

$$\mathbf{S}_m(a \cdot \dot{\mathbf{t}}^i \cdot \alpha) = \begin{cases} \Phi & \text{if } i < r, \\ \alpha & \text{if } i \geq r. \end{cases}$$

Consequently, $\mathbf{t}_m$ "follows" $\mathbf{S}_m$ by at most one round:

$$\mathbf{t}_m^i \sqsupseteq \mathbf{S}_m(a{\cdot}\dot{\mathbf{t}}^i{\cdot}\alpha) \sqsupseteq \mathbf{t}_m^{i+1} \tag{2.1}$$

We now proceed by induction on i to show that

$$\mathbf{t}_j^i \sqsupseteq \dot{\mathbf{t}}_j^i \sqsupseteq \mathbf{t}_j^{i+1}, \text{for } 1 \leq j \leq m-1. \tag{2.2}$$

For the basis, observe that $\mathbf{t}_j^0 = \mathbf{s}_j$ and $\dot{\mathbf{t}}_j^0 = \mathbf{s}_j$ for $1 \leq j \leq m-1$ (i.e., $\mathbf{N}$ and $\dot{\mathbf{N}}$ have the same starting point). By the monotonicity of Algorithm B, $\mathbf{t}_0^i \sqsupseteq \mathbf{t}_j^1$, so the basis holds ($\mathbf{t}_j^0 \sqsupseteq \dot{\mathbf{t}}_j^0 \sqsupseteq \mathbf{t}_j^1$). Assume inductively that $\mathbf{t}_j^i \sqsupseteq \dot{\mathbf{t}}_j^i \sqsupseteq \mathbf{t}_j^{i+1}$, for $1 \leq j \leq m-1$. This hypothesis, the monotonicity of $\mathbf{S}$, and the monotonicity of Algorithm B, yield for $1 \leq j \leq m-1$

$$\begin{array}{rcll}
\mathbf{t}_j^{i+1} & = & \mathbf{S}_j(a{\cdot}\mathbf{t}^i) & \text{(by definition)} \\
& \sqsupseteq & \mathbf{S}_j(a{\cdot}\dot{\mathbf{t}}^i{\cdot}\mathbf{t}_m^i) & \text{(by I.H. and mono. of } \mathbf{S}) \\
& \sqsupseteq & \mathbf{S}_j(a{\cdot}\dot{\mathbf{t}}^i{\cdot}\mathbf{S}_m(a{\cdot}\dot{\mathbf{t}}^i{\cdot}\alpha)) & \text{(by LHS of (2.1) and mono. of } \mathbf{S}) \\
& = & \dot{\mathbf{S}}_j(a{\cdot}\dot{\mathbf{t}}^i) & \text{(by definition)} \\
& \sqsupseteq & \mathbf{S}_j(a{\cdot}\dot{\mathbf{t}}^i{\cdot}\mathbf{t}_m^{i+1}) & \text{(by RHS of (2.1) and mono. of } \mathbf{S}) \\
& \sqsupseteq & \mathbf{S}_j(a{\cdot}\mathbf{t}^{i+1}) & \text{(by RHS of I.H. and mono. of } \mathbf{S}) \\
& = & \mathbf{t}_j^{i+2} & \text{(by definition)}
\end{array}$$

and the induction goes through.

Given (2.2) and the fact that Algorithm B converges, it follows immediately that $\mathbf{t}_j^B = \dot{\mathbf{t}}_j^B$ for $1 \leq j \leq m-1$. ∎

2.7.3 Proof of Proposition 2.29

Proof Consider a vertex v_i of the circuit graph. The proof is by induction on the depth of v_i in $\dot{\mathbf{N}}$.

Base: $depth(v_i) = 0$: Since the depth of v_i is zero, it has a corresponding state variable, say $\mathbf{s}_i$, in $\dot{\mathbf{N}}$. By Proposition 2.28, $\mathbf{t}_i^B = \dot{\mathbf{t}}_i^B$. By Definition 2.8, for a vertex of depth 0, $\dot{\mathbf{t}}_i^B = \dot{\mathbf{F}}_i(a{\cdot}\dot{\mathbf{t}}^B)$.

I.H.: For all $j < k$, where $depth(v_i) = j$, $\mathbf{t}_i^B = \dot{\mathbf{F}}_i(a{\cdot}\dot{\mathbf{t}}^B)$.

I.S.: Suppose vertex v_i has depth k, with vertex function $\mathbf{V}_i$, and corresponding state variable $\mathbf{s}_i$ in $\mathbf{N}$. Since the result of Algorithm B is ternary stable,

$$\mathbf{t}_i^B = \mathbf{S}_i(a{\cdot}\mathbf{t}^B).$$

In a complete network, the excitation function and vertex function are the same for any vertex. Hence,

$$\begin{aligned} \mathbf{t}_i^B &= \mathbf{V}_i(a\cdot\mathbf{t}^B) \\ &= \mathbf{V}_i(a\cdot\mathbf{t}_1^B\cdot\mathbf{t}_2^B\cdot\ldots\cdot\mathbf{t}_m^B). \end{aligned}$$

By the induction hypothesis,

$$\mathbf{t}_i^B = \mathbf{V}_i(a\cdot\dot{\mathbf{F}}_1(a\cdot\dot{\mathbf{t}}^B)\cdot\dot{\mathbf{F}}_2(a\cdot\dot{\mathbf{t}}^B)\cdot\ldots\cdot\dot{\mathbf{F}}_m(a\cdot\dot{\mathbf{t}}^B)).$$

Finally, by the definition of circuit equation,

$$\mathbf{t}_i^B = \dot{\mathbf{F}}_i(a\cdot\dot{\mathbf{t}}^B).$$

■

2.7.4 Proof of Theorem 2.45

Proof Consider a reachable state $a_0\cdot\hat{b}$. Without loss of generality, assume $a_0\cdot\hat{b}$ is stable. Suppose $(a_0\cdot\hat{b}, a_1\cdot b') \in T_N$, where $a_1\cdot b'$ is unstable. Since b' is unstable, $a_1\cdot b'$ must belong to a nontransient cycle $(a_1\cdot b_0, a_1\cdot b_1, \ldots, a_1\cdot b_{k-1})$, where $b' = b_i$ for some $0 \le i \le k-1$, and $b_i R_{a_1} b_{i+1}$ for $0 \le i \le k-1$ (subscripts are always modulo k). (Recall that R_{a_1} is the GMW relation for the input held constant at a_1.) Thus, $(a_0\cdot\hat{b}, a_1\cdot b_i) \in T_N$, for $0 \le i \le k-1$. This is depicted in Figure 2.51a, where the solid arcs indicate GMW transitions, dotted arcs indicate transitions of T_N, and wavy arcs indicate a sequence of GMW transitions, holding the input constant.

Suppose the next input is a_2, and let $a_2\cdot b_{k+i}$ be an outcome state of $a_2\cdot b_i$, for $0 \le i \le k-1$. Hence, with or without ASMP, $(a_1\cdot b_i, a_2\cdot b_{k+i}) \in T_N$. However, without ASMP, when a_1 changes to a_2, it is also possible for b_i to change to b_{i+1} simultaneously. This gives rise to the additional transitions $(a_1\cdot b_i, a_2\cdot b_{k+i+1})$ (see Figure 2.51b).

Since the set of transitions without ASMP contains the set with ASMP, if N is not sequentially output-stable with ASMP, then clearly it is not sequentially output-stable without ASMP.

Hence, it remains to show the converse: the additional transitions alone cannot cause a network to be not sequentially output-stable. Consider the runs of length two starting from state $a_o\cdot\hat{b}$. Let type I runs be those present with or without ASMP. These have the form

$$a_o\cdot\hat{b}, a_1\cdot b_i, a_2\cdot b_{k+i}, \text{ for } 0 \le i \le k-1.$$

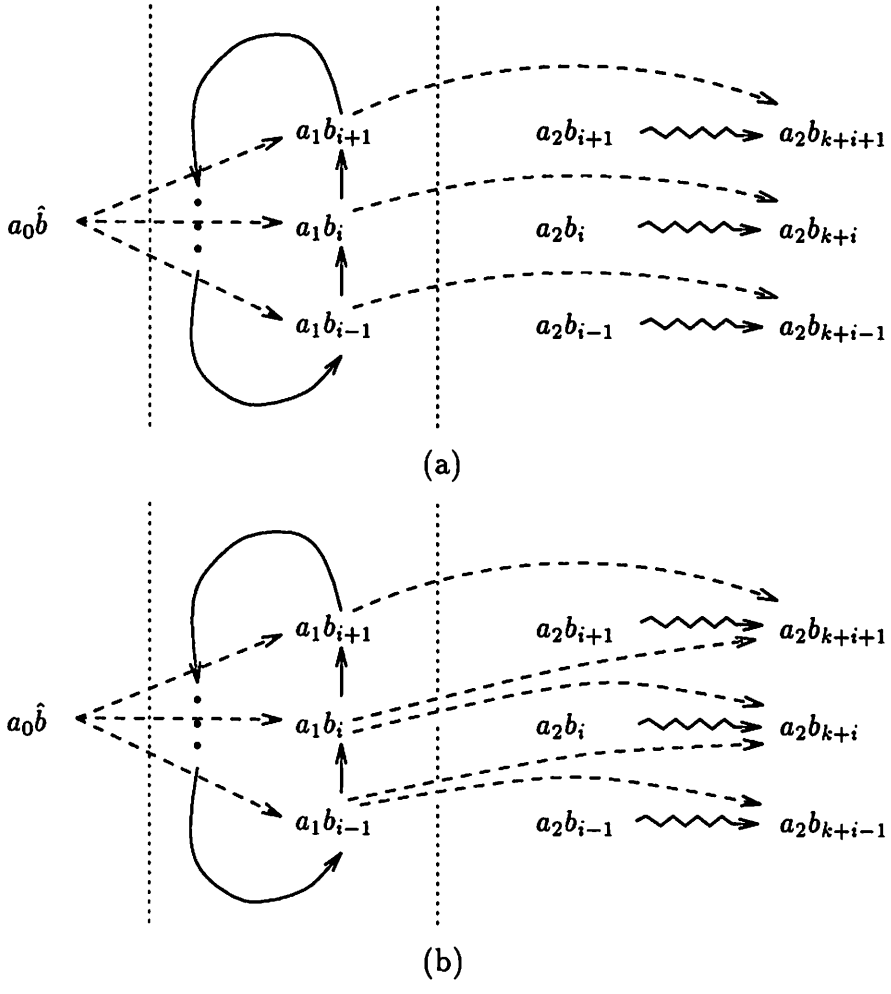


Figure 2.51: a) Runs with ASMP. b) Runs without ASMP.

Let type II runs be those present only without ASMP:

$$a_o \cdot \hat{b}, a_1 \cdot b_i, a_2 \cdot b_{k+i+1}, \text{ for } 0 \leq i \leq k-1.$$

We say that two runs differ if they have different output components on the same clock cycle. Without loss of generality, assume that all internal state variables are outputs. Suppose N without ASMP is not sequentially output-stable. There are three cases to consider.

Case 1: Two type I runs differ. Since N with ASMP contains all type I runs, then N with ASMP is also not sequentially output-stable.

Case 2: A type I run γ_1 differs from a type II run γ_2. Suppose

$$\begin{aligned} \gamma_1 &= a_o \cdot \hat{b}, a_1 \cdot b_i, a_2 \cdot b_{k+i}, \text{ and} \\ \gamma_2 &= a_o \cdot \hat{b}, a_1 \cdot b_j, a_2 \cdot b_{k+j+1}. \end{aligned}$$

Consider also the type I runs

$$\begin{aligned} \gamma_3 &= a_o \cdot \hat{b}, a_1 \cdot b_j, a_2 \cdot b_{k+j}, \text{ and} \\ \gamma_4 &= a_o \cdot \hat{b}, a_1 \cdot b_{j+1}, a_2 \cdot b_{k+j+1}. \end{aligned}$$

If γ_1 and γ_2 differ because $b_i \neq b_j$, then the type I runs γ_1 and γ_3 differ. Likewise, if γ_1 and γ_2 differ because $b_{k+i} \neq b_{k+j+1}$, then γ_1 and γ_4 differ. In either case, this shows that N with ASMP is also not sequentially output-stable.

Case 3: Two type II runs γ_1 and γ_2 differ. Suppose

$$\begin{aligned} \gamma_1 &= a_o \cdot \hat{b}, a_1 \cdot b_i, a_2 \cdot b_{k+i+1}, \text{ and} \\ \gamma_2 &= a_o \cdot \hat{b}, a_1 \cdot b_j, a_2 \cdot b_{k+j+1}. \end{aligned}$$

Consider also the type I runs

$$\begin{aligned} \gamma_3 &= a_o \cdot \hat{b}, a_1 \cdot b_i, a_2 \cdot b_{k+i}, \\ \gamma_4 &= a_o \cdot \hat{b}, a_1 \cdot b_j, a_2 \cdot b_{k+j}, \\ \gamma_5 &= a_o \cdot \hat{b}, a_1 \cdot b_{i+1}, a_2 \cdot b_{k+i+1}, \text{ and} \\ \gamma_6 &= a_o \cdot \hat{b}, a_1 \cdot b_{j+1}, a_2 \cdot b_{k+j+1}. \end{aligned}$$

If γ_1 and γ_2 differ because $b_i \neq b_j$, then the type I runs γ_3 and γ_4 differ. Likewise, if γ_1 and γ_2 differ because $b_{k+i+1} \neq b_{k+j+1}$, then γ_5 and γ_6 differ. In either case, this shows that N with ASMP is also not sequentially output-stable. ∎

2.8 Summary and future work

We have presented a formal classification of synchronous circuits based on their input/output behavior, which takes into account the effects of combinational cycles. This analysis is grounded in the up-bounded inertial delay model.

Generally speaking, a circuit is output-stable if for every input sequence, the circuit produces a unique, stable output sequence. This general notion is formalized in three different classes of circuits, giving a tradeoff on the time to decide the class, versus the permissiveness of the class.

The easiest class to decide, and the most conservative, is combinational output-stability. This class assumes that all flip-flop states are reachable, and it ignores the state-holding ability of the combinational part of the circuit. BDD-based ternary simulation is used to decide this class.

The most difficult class to decide, and the most permissive, is sequential output-stability. This class takes into account the state-holding ability of the combinational part, and distinguishes between behavior in the reachable and unreachable state space. Because of the aggressive nature of this class, we must assume that the inputs do not glitch; consequently, the class of sequentially output-stable circuits is not closed under composition. The decision procedure for this class first constructs the transition graph based on the GMW relation, then builds the quotient machine of the graph with respect to bisimulation, and finally, tests the quotient machine for deterministic behavior.

The last class, constructivity, falls between the other two classes in the time to decide the class, and the degree of permissiveness. It distinguishes between behavior in the reachable and unreachable state space, but does not permit the combinational part to hold state. However, due to this last restriction, the inputs are not required to be glitch-free, and consequently, this class is closed under composition. This class exactly coincides with the class of constructive Esterel programs.

For each of the three classes, if a circuit falls within that class, then we provide a method to produce a new circuit with the same I/O behavior, and without combinational cycles. This is an important feature, as many high-level CAD tools do not accept circuits with combinational cycles.

If a circuit does not fall within a given class, then we provide a procedure to generate an error trace, giving a sequence of inputs that demonstrate why the circuit fails.

Lastly, information about the environment, if known, can be taken into account in a refined analysis.

There are several directions that future work can take. The first is to extend the analysis in the following ways.

1. Allow transistors and tri-state devices.

2. Allow multi-phase clocks.

3. Allow alternate delay models, such as bi-bounded inertial delay and fixed ideal delay.

The second avenue of future work is to refine the methods we have presented. For example, it would be interesting to formulate an implicit procedure for deciding sequential output-stability, possibly employing ternary simulation. Also, for output-stable circuits, it would be useful to derive an acyclic implementation that preserves as much structure of the original circuit as possible, so that the changes are not as drastic.

The third direction is to define new classes of output-stable circuits, with different properties. For example, following the spirit of Singhal's work [32], one could ignore unstable behavior for the first k clock cycles, to allow a circuit enough time to be reset. Another possibility is to define a class strictly larger than constructivity, but which is still insensitive to glitching.

Chapter 3

Formula-Dependent Equivalence for Formal Verification

3.1 Introduction

Formal design verification is the process of verifying that a design has certain properties that the designer intended. A well known verification technique is computation tree logic (CTL) model checking. In this approach, a design is modeled as a finite state machine (FSM), properties are stated using CTL formulas, and a "model checker" is used to prove that the FSM satisfies the given CTL formulas [33]. The complexity of model checking a formula is linear in the number of states of the FSM.

Oftentimes, large designs are constructed by linking together a set of FSMs. The straightforward approach to model checking such a design is to first form the product of the component FSMs to yield a single FSM, and then proceed to model check this single FSM. However, the size of the product machine can be exponential in the number of component machines, and hence the model checker may take exponential time. This is known as the "state explosion problem" when using explicit representations, or the "representation explosion problem" when using implicit representations, like BDDs (see Section 1.1.3). As it turns out, we cannot hope to do better than this in the worst case, because the problem of model checking a system of interacting FSMs is PSPACE-complete [26].

Our goal is to develop an algorithm that alleviates the explosion problem by identifying equivalent states in each component machine. These equivalent states are then used

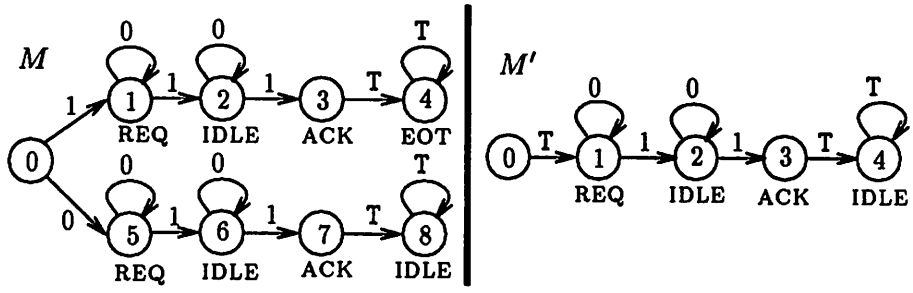


Figure 3.1: Finite state machine M with inputs 0 and 1 and outputs REQ, ACK, IDLE and EOT. The symbol T means "true", the union of all input assignments.

to simplify the components before taking their product, thus leading to a smaller product machine. It is well known that *bisimulation equivalence* is the coarsest (or weakest) equivalence that preserves the truth of *all* CTL formulas [34]. However, in general we are interested in model checking a system with respect to just a few formulas, and hence preserving all CTL formulas is stronger than needed. Thus, we investigate a formula-dependent equivalence that preserves the truth of a particular formula of interest, but possibly not of other formulas. This leads to a coarser equivalence, and thus to a greater opportunity for simplification. If an explicit representation is used for the FSMs, then this equivalence is used to form the quotient machines of the components. If BDDs are used, then the equivalence is used to define a range of permissible transition relations, among which we want to use the one with the smallest BDD.

Consider for example the Moore FSM M described in Figure 3.1. The CTL formula $\phi = \forall G(\text{REQ} \rightarrow \forall F\text{ACK})$ expresses the property that every request is eventually acknowledged. The behaviors from state 1 and 5 are different because states 4 and 8 produce different outputs. However, since there are no behaviors from states 4 and 8 where REQ is produced, then ϕ is always true at these states. Hence, states 1 and 5 can actually be merged, with respect to ϕ. Consequently, M can be replaced by the 5-state machine M': verifying ϕ on a product machine containing the component M is equivalent to verifying ϕ on the product machine with M replaced by M'.

The approach we have developed can be applied to *any* formula of CTL. Thus, we can handle formulas that refer to atomic propositions of any number of the component machines, and the formulas can be nested arbitrarily. The approach is fully automatic and it is exact, in that it returns exactly the set of product states satisfying the formula of interest. Finally, in some cases the approach can detect if a formula passes or fails, without

composing all the component machines.

Section 3.2 discusses related work, and Section 3.3 presents some preliminaries. In Section 3.4 we develop our formula-dependent equivalence, and in Section 3.5 we discuss how this equivalence can be used to simplify compositional model checking. Finally, Section 3.7 mentions future work and gives a summary, and Section 3.6 contains the proofs of the theorems.[1]

3.2 Related work

Other researchers have addressed the problem of reducing the complexity of model checking. As mentioned in the introduction, bisimulation preserves the truth of all CTL formulas, and hence can be used to identify equivalent states to derive smaller component machines. This technique has been used by [36].

Clarke *et al.* presented the *interface rule*, which can be applied when a CTL formula refers to the atomic propositions of just one machine, the "main" machine [37]. In this case, the outputs of the other machines, which cannot be sensed by the main machine, can be "hidden". After hiding such outputs, some states in the other machines may become equivalent, and hence the number of states can be reduced. This technique is orthogonal to our approach, and thus the two approaches could be combined. In general, any output not referred to by the formula, and not observable by other machines, can be hidden.

Grümberg *et al.* defined a subset of CTL, known as ACTL, which permits only universal path quantification, and not existential path quantification [38]. They go on to develop an approach to compositional model checking for ACTL. If an ACTL formula is true of one component in a system, then it is true of the entire system. Thus, in some cases the full product machine can be avoided. However, the formula may be true of the entire system, *without* being true of any one component in isolation, i.e., their approach is conservative, and not exact. In this case, some components must be composed, and the procedure repeated. The user has the option of manually forming abstractions for some of the machines. If the formula is false, then the product machine must always be formed. An asset of this approach is that it handles fairness constraints on the system.

Dams *et al.* have also devised an approach using ACTL [39]. Like our method, they compute an equivalence with respect to a *single* formula. Although they are limited to

[1]This chapter is largely taken from [35].

formulas of ACTL, it may turn out that coarser equivalences are possible by restricting to a subset of CTL. They do not address how their equivalence can be used in *compositional* model checking, where a formula may refer to the atomic propositions of several interacting machines.

Our experience indicates that existential path properties are useful for determining if a system *can* exhibit a certain behavior. This is especially true when ascertaining if the environment for a system has been correctly modeled so that it can produce the stimuli of interest. Hence, we are interested in techniques that can handle *full* CTL.

The work of Chiodo *et al.* [40] has similar aims as ours, and the current work can be seen as an outgrowth of that work. Both approaches are exact, fully automatic, and formula dependent. We have extended Chiodo's method (see Section 3.5.3), and have cast our extension as an equivalence on states.

3.3 Preliminaries

3.3.1 Finite states machines

The systems that we want to verify are synchronous, interacting Moore FSMs, as defined in Section 1.1.2. Each component FSM receives a set of binary-valued inputs, and produces another set of binary-valued outputs. For the purposes of this discussion, we omit the initial states I and the output alphabet Σ_O from the specification of an FSM. Hence, an FSM is a 6-tuple $M = \langle S, \mathcal{X}, \Sigma_I, \mathcal{O}, O, T \rangle$, where, as a reminder,

- S is a finite set of states,
- $\mathcal{X}$ is the set of input signals,
- Σ_I is the set of assignments to $\mathcal{X}$,
- $\mathcal{O}$ is the set of output signals,
- O is the output function, and
- T is the (complete) transition relation.

3.3.2 Computation tree logic

Computation tree logic is a language used to describe properties of state transition systems. We are interested in checking CTL formulas that describe properties of the composition of a set of interacting FSMs. Since the composition of a set of FSMs is again an FSM, we give the syntax and semantics of CTL for a single FSM M. We allow two types of atomic propositions:

1. each output variable is an atomic proposition, and
2. each subset of states is an atomic proposition

The second type arises naturally when recursively checking formulas. With this, the set of CTL formulas is defined inductively as follows.

Definition 3.1 CTL syntax:

- p is a CTL formula, where p is an output variable or a subset of states, and
- if ψ_1 and ψ_2 are CTL formulas, then so are $\neg\psi_1$, $\psi_1 \vee \psi_2$, $\exists X \psi_1$, $\exists G \psi_1$, and $\exists[\psi_1 \; U \; \psi_2]$.

■

Note that inputs are *not* allowed as atomic propositions. However, by modeling an input by an FSM whose output describes the expected behavior of the input, one can implicitly use an input as an atomic proposition.

The semantics of CTL is usually defined on finite Kripke structures, which are directed graphs where each node is labeled by a set of atomic propositions [33]. To extend these semantics to FSMs, we just ignore the labels on the transitions of the FSMs, and we view the outputs as atomic propositions. Let $M = \langle S, \mathcal{X}, \Sigma_I, \mathcal{O}, O, T \rangle$ be an FSM. A *path* from state x_0 is an infinite sequence of states $x_0 x_1 x_2 \ldots$ such that for every i, there exists an $a \in \Sigma_I$ such that $(x_i, a, x_{i+1}) \in T$. The notation $M, x_0 \models \phi$ means that ϕ is true in state x_0 of FSM M. The semantics of CTL is defined inductively as follows.

Definition 3.2 CTL semantics:

- $M, x_0 \models p$, where $p \in \mathcal{O}$, iff $p \in O(x_0)$.
- $M, x_0 \models p$, where $p \subseteq S$, iff $x_0 \in p$.

- $M, x_0 \models \neg\psi_1$ iff $M, x_0 \not\models \psi_1$.
- $M, x_0 \models \psi_1 \vee \psi_2$ iff $M, x_0 \models \psi_1$ or $M, x_0 \models \psi_2$.
- $M, x_0 \models \exists X\psi_1$ iff there exists a path $x_0x_1x_2\ldots$ such that $M, x_1 \models \psi_1$.
- $M, x_0 \models \exists G\psi_1$ iff there exists a path $x_0x_1x_2\ldots$ such that for all i, $M, x_i \models \psi_1$.
- $M, x_0 \models \exists[\psi_1\ U\ \psi_2]$ iff there exists a path $x_0x_1x_2\ldots$ and some $i \geq 0$ such that $M, x_i \models \psi_2$ and for all $j < i$, $M, x_j \models \psi_1$.

■

For example, in machine $M_1 \times M_2$ of Figure 1.2, state $\langle 1, 2'\rangle$ satisfies the formula $\exists G(\neg p \wedge \neg q)$, whereas none of the other states do. The expression $\exists F\psi$ is an abbreviation for $\exists[\mathit{true}\ U\ \psi]$, where *true* is a logical tautology.

Our objective is to solve the following problem.

Definition 3.3 Let $M = \langle S, \mathcal{X}, \Sigma_I, \mathcal{O}, O, T\rangle$ be an FSM, and let ϕ be a CTL formula. The *CTL model checking problem* is to determine all states $x \in S$ such that $M, x \models \phi$. ■

3.4 Formula-dependent equivalence

3.4.1 Overview

Our goal is to define an equivalence on the states of each component machine that is as coarse as possible with respect to a given CTL formula ϕ, while being efficiently computable. Section 3.5 explains how we intend to apply this equivalence to model checking, but the main idea is to merge equivalent states to minimize the size of each component. The minimized machines are then composed. Optionally, the product can be computed incrementally by composing a few of the minimized machines, and then computing a new equivalence for this sub-product. When the top level is reached and just a single machine remains, the usual CTL model checking algorithm is applied to determine the states that satisfy ϕ.

Our formula dependent equivalence can be best explained by comparing it to bisimulation ("strong bisimulation" of Milner [27, p. 88]).

Definition 3.4 Given an FSM $M = (S, \mathcal{X}, \Sigma_I, \mathcal{O}, O, T)$, the *bisimulation equivalence relation*, denoted by $\sim$, is the coarsest equivalence relation satisfying the following:
For all $x, y \in S$, $x \sim y$ implies

- $O(x) = O(y)$, and
- for all $a \in \Sigma_I$
  - whenever $x \xrightarrow{a} t$, then for some w, $y \xrightarrow{a} w$ and $t \sim w$, and
  - whenever $y \xrightarrow{a} w$, then for some t, $x \xrightarrow{a} t$ and $t \sim w$.

■

The soundness of this definition follows from the observation that the class of equivalence relations satisfying the above definition contains the identity, and is closed under union. Intuitively, two states are bisimilar if their corresponding infinite computation trees[2] "match". This means that the two states have the same outputs, and on each input, the two states have next states whose infinite computation trees again match.

We use the notion of *PASS* and *FAIL* states to ease the strict requirement of bisimulation that the infinite computation trees of two states match. Loosely, if a state is a $PASS^\phi$ state with respect to a CTL formula ϕ, then it satisfies ϕ in all environments; likewise, if a state is $FAIL^\phi$, then it does not satisfy ϕ in any environment. Given $PASS^\phi$ and $FAIL^\phi$ states, the first modification to bisimulation we make is that subtrees rooted at $FAIL^\phi$ states are ignored. This means that transitions to $FAIL^\phi$ states from one state need not be matched by the other state. This works because only potential witnesses to a formula need to be preserved. The second modification is that two states are equivalent if they are both $PASS^\phi$ states. A consequence of this is that whereas bisimulation requires the infinite computation trees of next states (for a given input) to match, now it is sufficient that the next states are both $PASS^\phi$ states. This is what we mean by two infinite trees matching *up to* $PASS^\phi$ states. Essentially then, we say that two states are equivalent with respect to ϕ if

1. they are equivalent with respect to the immediate subformulas of ϕ, and

[2]The infinite computation tree of a state is formed by "unrolling" the FSM starting from that state.

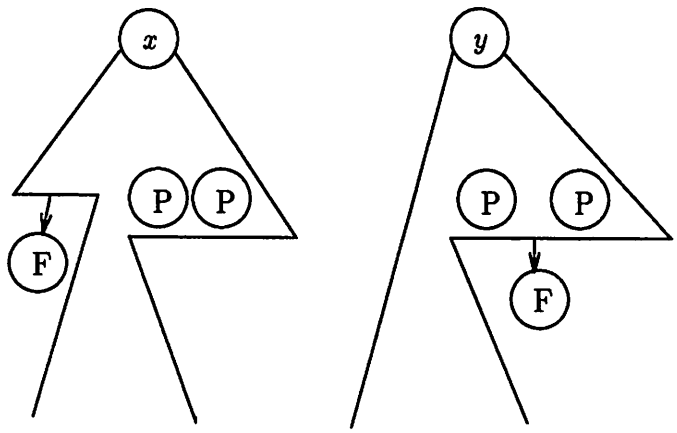


Figure 3.2: Infinite computation trees of states x and y. "P" indicates a $PASS^{\phi}$ state, and "F" indicates a $FAIL^{\phi}$ state.

2. either they are both $PASS^{\phi}$ states or both $FAIL^{\phi}$ states, or the infinite computation trees of the two states match up to $PASS^{\phi}$ states, ignoring all subtrees rooted at $FAIL^{\phi}$ states.

This last point is illustrated in Figure 3.2, which shows the computation trees from two states, x and y. Subtrees rooted at $FAIL^{\phi}$ are ignored, while corresponding subtrees rooted at $PASS^{\phi}$ states are sufficient to declare a match of the subtrees.

3.4.2 $PASS^{\phi}$ and $FAIL^{\phi}$

Before formally defining our equivalence relation, we define the $PASS^{\phi}$ and $FAIL^{\phi}$ sets. For a given formula ϕ, $PASS^{\phi}$ and $FAIL^{\phi}$ sets are defined for each component. In the following definition, we assume a system of just two components, M and M'. In defining the $PASS^{\phi}$ and $FAIL^{\phi}$ sets for M, M' is referenced because the atomic propositions in ϕ may refer to M'. The symbols p_o and p_i are used to distinguish those output atomic propositions produced by M and those produced by M', respectively.

Definition 3.5 Let $M = \langle S, \mathcal{X}, \Sigma_I, \mathcal{O}, O, T \rangle$ and $M' = \langle S', \mathcal{X}', \Sigma_I', \mathcal{O}', O', T' \rangle$ be FSMs, and let ϕ be a CTL formula. Let $p_o \in \mathcal{O}$, $p_i \in \mathcal{O}'$, and $p_s \subseteq S \times S'$. $PASS^{\phi}$ and $FAIL^{\phi}$ for M are subsets of S, as follows:

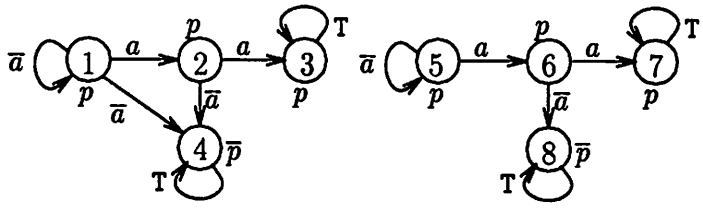


Figure 3.3: Illustrating $PASS^{\phi}$ and $FAIL^{\phi}$, and the fact that $\mathcal{E}^{\phi}$ is coarser than bisimulation.

ϕ		
p_i	$PASS^{\phi}$	$\emptyset$
	$FAIL^{\phi}$	$\emptyset$
p_o	$PASS^{\phi}$	$\{x \in S \mid p_o \in O(x)\}$
	$FAIL^{\phi}$	$\{x \in S \mid p_o \notin O(x)\}$
p_s	$PASS^{\phi}$	$\{x \in S \mid \forall s' \in S', (x,s') \in p_s\}$
	$FAIL^{\phi}$	$\{x \in S \mid \forall s' \in S', (x,s') \notin p_s\}$
$\neg\psi$	$PASS^{\phi}$	$FAIL^{\psi}$
	$FAIL^{\phi}$	$PASS^{\psi}$
$\psi_1 \vee \psi_2$	$PASS^{\phi}$	$PASS^{\psi_1} \cup PASS^{\psi_2}$
	$FAIL^{\phi}$	$FAIL^{\psi_1} \cap FAIL^{\psi_2}$
$\exists X\psi$	$PASS^{\phi}$	$\{x \in S \mid \forall a \in \Sigma_I, \exists t \in PASS^{\psi}$ s.t. $x \xrightarrow{a} t\}$
	$FAIL^{\phi}$	$\{x_0 \in S \mid$ for every path $x_0x_1x_2\ldots, x_1 \in FAIL^{\psi}\}$
$\exists G\psi$	$PASS^{\phi}$	greatest fixed-point of: $R_0 = PASS^{\psi}$; $R_{i+1} = R_i \cap \{x \in S \mid \forall a \in \Sigma_I, \exists t \in R_i$ s.t. $x \xrightarrow{a} t\}$
	$FAIL^{\phi}$	$\{x_0 \in S \mid$ for every path $x_0x_1x_2\ldots$, there exists $i \geq 0$ s.t. $x_i \in FAIL^{\psi}\}$
$\exists[\psi_1 U \psi_2]$	$PASS^{\phi}$	least fixed-point of: $R_0 = PASS^{\psi_2}$; $R_{i+1} = R_i \cup \{x \in S \mid x \in PASS^{\psi_1}$, and $\forall a \in \Sigma_I$, $\exists t \in R_i$ s.t. $x \xrightarrow{a} t\}$
	$FAIL^{\phi}$	$\{x_0 \in S \mid$ for every path $x_0x_1x_2\ldots$, either 1) there exists $i \geq 0$ s.t. $x_i \in FAIL^{\psi_1}$ and $\forall j \leq i, x_j \in FAIL^{\psi_2}$, or 2) $\forall i \geq 0, x_i \in FAIL^{\psi_2}\}$

As an example of $PASS^{\phi}$ and $FAIL^{\phi}$, consider the FSM in Figure 3.3. For $\psi = p$, states 1, 2, 3, 5, 6 and 7 lie in $PASS^{\psi}$ and states 4 and 8 lie in $FAIL^{\psi}$. For $\phi = \exists Gp$, states 3 and 7 lie in $PASS^{\phi}$, while states 4 and 8 lie in $FAIL^{\phi}$, and states 1, 2, 5 and 6 lie in neither. The following proposition says that, indeed, if x is in $PASS^{\phi}$, then any product state with x as a component satisfies ϕ. The proof is given by Proposition 3.14 in Section 3.6.2.

Proposition 3.6 *Let ϕ be a CTL formula, and let x be a state of M. If $x \in$ PASS$^{\phi}$, then for every FSM M', and every state t of M', $M \times M', \langle x,t\rangle \models \phi$. Likewise, if $x \in$ FAIL$^{\phi}$,*

then $M \times M', \langle x, t \rangle \not\models \phi$.

Note that the converse is not true. For example, consider a component M and the formula $\phi = q \wedge \neg q$, where q is an output of some other component. Then $FAIL^{\phi}$ for M is empty (because $FAIL^{q}$ and $PASS^{q}$ are empty by case p_i of Definition 3.5), even though ϕ is not satisfiable (i.e., for any component M', no state in $M \times M'$ satisfies ϕ). However, this weakness in the definition of $FAIL^{\phi}$ makes it tractable to compute. In fact, strengthening the definition of $FAIL^{\phi}$ so that the converse of Proposition 3.6 holds would make $FAIL^{\phi}$ EXPTIME-hard to compute.

Proposition 3.7 *Let ϕ be a CTL formula, and let x be a state of M. Suppose* FAIL$^{\phi}$ *was defined such that then for every FSM M', and every state t of M',*

$$M \times M', \langle x, t \rangle \not\models \phi \text{ iff } x \in \text{FAIL}^{\phi}.$$

Then, FAIL$^{\phi}$ *would be EXPTIME-hard to compute.*

Proof The reduction is from CTL satisfiability, which is known to be EXPTIME-complete [41]. To check if a formula ϕ is satisfiable, compute $FAIL^{\phi}$ for the component M shown in Figure 3.4, which has no outputs, and which is not referred to by ϕ.

Claim: $x \in FAIL^{\phi}$ if and only if ϕ is not satisfiable.

Since M has no outputs, and its one transition is always enabled, then composing M with another component M' does not alter the transition relation of M'. That is, M' and $M \times M'$ are isomorphic. Thus, for any state t of M', and for any formula ϕ,

$$M \times M', \langle x, t \rangle \models \phi \text{ iff } M', t \models \phi.$$

($\Rightarrow$) If $x \in FAIL^{\phi}$, then (by supposition) for any state t of any FSM M', $M \times M', \langle x, t \rangle \not\models \phi$, and hence $M', t \not\models \phi$. Thus, ϕ is not satisfiable.

($\Leftarrow$) If ϕ is not satisfiable, then for any state t of any FSM M', $M', t \not\models \phi$, and hence $M \times M', \langle x, t \rangle \not\models \phi$. Thus, $x \in FAIL^{\phi}$. ∎

Thus, satisfiability could be answered if we could compute $FAIL^{\phi}$ exactly. Similarly, since $x \in FAIL^{\phi}$ if and only if $x \in PASS^{\neg\phi}$, the same reduction shows that $PASS^{\phi}$ would also be EXPTIME-hard to compute.

Figure 3.4: Component machine used to show that computing $FAIL^{\phi}$ exactly is EXPTIME-hard.

3.4.3 Equivalence relation $\mathcal{E}^{\phi}$

Now we formally define our equivalence relation.

Definition 3.8 Let $M = \langle S, \mathcal{X}, \Sigma_I, \mathcal{O}, O, T \rangle$ and $M' = \langle S', \mathcal{X}', \Sigma_I', \mathcal{O}', O', T' \rangle$ be FSMs, and let ϕ be a CTL formula. The equivalence relation $\mathcal{E}^{\phi}$ on the states of FSM M is the coarsest equivalence relation satisfying the following.

For $x, y \in S$, $\mathcal{E}^{\phi}(x, y)$ iff:

Case $\phi = p_i$: $(x, y) \in S \times S$.

Case $\phi = p_o$: $x \in FAIL^{\phi}$ and $y \in FAIL^{\phi}$ or $x \in PASS^{\phi}$ and $y \in PASS^{\phi}$.

Case $\phi = p_s$: for all $s' \in S'$, $(x, s') \in p_s$ iff $(y, s') \in p_s$.

Case $\phi = \neg\psi$: $\mathcal{E}^{\psi}(x, y)$.

Case $\phi = \psi_1 \vee \psi_2$: $\mathcal{E}^{\psi_1}(x, y)$ and $\mathcal{E}^{\psi_2}(x, y)$.

Case $\phi = \exists X \psi$: $\mathcal{E}^{\psi}(x, y)$ and

1. $x \in FAIL^{\phi}$ and $y \in FAIL^{\phi}$, or $x \in PASS^{\phi}$ and $y \in PASS^{\phi}$, or
2. $O(x) = O(y)$, and for all $a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^{\psi}$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{E}^{\psi}(t, w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^{\psi}$, $\exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{E}^{\psi}(t, w)$.

Case $\phi = \exists G \psi$: $\mathcal{E}^{\psi}(x, y)$ and

1. $x \in FAIL^{\phi}$ and $y \in FAIL^{\phi}$, or $x \in PASS^{\phi}$ and $y \in PASS^{\phi}$, or
2. $O(x) = O(y)$, and for all $a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^{\phi}$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{E}^{\phi}(t, w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^{\phi}$, $\exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{E}^{\phi}(t, w)$.

Case $\phi = \exists[\psi_1 \; U \; \psi_2]$: $\mathcal{E}^{\psi_1}(x, y)$ and $\mathcal{E}^{\psi_2}(x, y)$ and

1. $x \in FAIL^{\phi}$ and $y \in FAIL^{\phi}$, or $x \in PASS^{\phi}$ and $y \in PASS^{\phi}$, or
2. $O(x) = O(y)$, and for all $a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^{\phi}$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{E}^{\phi}(t, w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^{\phi}$, $\exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{E}^{\phi}(t, w)$.

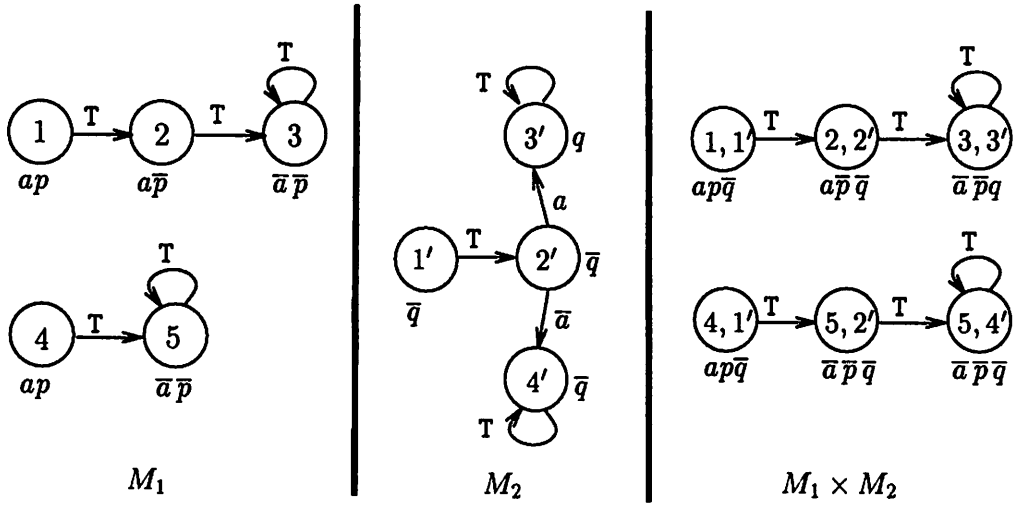


Figure 3.5: Equivalence on subformulas is required. Only the states reachable from $\langle 1, 1' \rangle$ and $\langle 4, 1' \rangle$ are shown in $M_1 \times M_2$.

■

In a manner similar to Milner, we can show that $\mathcal{E}^\phi$ is the maximum fixed-point of a certain functional (see Lemma 3.21 of Section 3.6.3.1). Hence, using a standard fixed-point computation, $\mathcal{E}^\phi$ can be computed in polynomial time.

Notice that $\mathcal{E}^\phi$ requires equivalence on all subformulas. As the following example shows, this requirement is warranted. Consider M_1 in Figure 3.5. For $\phi = \exists F(p \wedge \exists F(\overline{p} \wedge q))$, states 2, 3 and 5 lie in $FAIL^\phi$, as detailed in Table 3.1. So with respect to ϕ, the infinite computation trees of 1 and 4 match when $FAIL^\phi$ successors are ignored, and if we did not require equivalence on subformulas, they would be $\mathcal{E}^\phi$-equivalent. However, if we were to compose M_1 with M_2, we see that ϕ holds in state $\langle 1, 1' \rangle$ but does not hold in state $\langle 4, 1' \rangle$. Thus, it would be wrong to have 1 and 4 be $\mathcal{E}^\phi$-equivalent. Requiring equivalence on all subformulas fixes this problem (in particular, 2 and 5 are distinguished by $\exists F(\overline{p} \wedge q)$ because they differ on an output, namely a, and this in turn causes 1 and 4 to be distinguished).

3.4.4 Properties of $\mathcal{E}^\phi$

Since we define CTL so that formulas may refer directly to states via atomic propositions, then any formula-*independent* equivalence (e.g., bisimulation) will distinguish every pair of states, whereas $\mathcal{E}^\phi$ may make some states equivalent. However, even if we could not refer to states, $\mathcal{E}^\phi$ is still coarser than bisimulation. As stated earlier, one reason

	ϕ	$PASS^\phi$	$FAIL^\phi$	equiv classes
1	p	$\{1,4\}$	$\{2,3,5\}$	$\{1,4\},\{2,3,5\}$
2	$\overline{p}$	$\{2,3,5\}$	$\{1,4\}$	$\{1,4\},\{2,3,5\}$
3	q	$\emptyset$	$\emptyset$	$\{1,2,3,4,5\}$
4	$\overline{p} \wedge q$	$\emptyset$	$\{1,4\}$	$\{1,4\},\{2,3,5\}$
5	$\exists F(\overline{p} \wedge q)$	$\emptyset$	$\emptyset$	$\{1\},\{4\},\{2,3,5\}$
6	$p \wedge \exists F(\overline{p} \wedge q)$	$\emptyset$	$\{2,3,5\}$	$\{1\},\{4\},\{2,3,5\}$
7	$\exists F(p \wedge \exists F(\overline{p} \wedge q))$	$\emptyset$	$\{2,3,5\}$	$\{1\},\{4\},\{2,3,5\}$

Table 3.1: Equivalence classes for M_1 of Figure 3.5 on $\exists F(p \wedge \exists F(\overline{p} \wedge q))$.

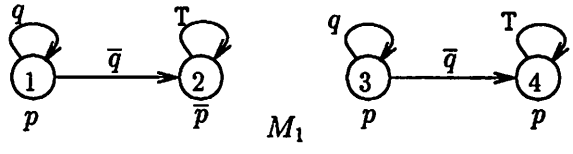


Figure 3.6: $\mathcal{E}^\phi$ equivalence is incomplete. The input to M_1 is q, and the output is p. States 1 and 3 can be safely merged with respect to the formula $\phi = \exists Gq$.

for this is that the subtrees rooted at $FAIL^\phi$ states are ignored. This is illustrated in Figure 3.3: if $\phi = \exists Gp$, then 4 is a $FAIL^\phi$ state, and thus 1 and 5 are $\mathcal{E}^\phi$-equivalent; however, they are not bisimilar.

On the other hand, there are cases where $\mathcal{E}^\phi$ distinguishes two states that could actually be merged. Consider the FSM M_1 in Figure 3.6 and the formula $\phi = \exists Gq$, where q is an output of some component not shown. Since q is an input to M_1, the sets $PASS^\phi$ and $FAIL^\phi$ are empty, and hence $\mathcal{E}^\phi$ reduces to bisimulation. States 1 and 3 are not bisimilar because 2 and 4 have different outputs, and thus 1 and 3 are not $\mathcal{E}^\phi$-equivalent. However, q must be false to reach states 2 and 4, and thus the difference between states 1 and 3 does not affect the validity of ϕ. Hence, states 1 and 3 could be merged with respect to ϕ, but $\mathcal{E}^\phi$ will not merge them.

The following proposition says that $\mathcal{E}^\phi$-equivalent states cannot be distinguished, with respect to ϕ, by any environment. This is key in proving Theorem 3.11, the theorem of correctness. The proof is given by Proposition 3.14 in Section 3.6.3.2.

Proposition 3.9 *Let ϕ be a CTL formula, and let x and y be states of M such that $\mathcal{E}^\phi(x,y)$. Then for any state t of any FSM M': $M \times M', \langle x,t\rangle \models \phi$ iff $M \times M', \langle y,t\rangle \models \phi$.*

As an aside, note that the converse of Proposition 3.9 is not true. In fact, just because two states cannot be distinguished with respect to ϕ by any environment, this does not imply that they can be merged. For example, consider M_1 in Figure 3.5, and the formula $\phi = \exists F(p \wedge \exists F(\overline{p} \wedge q))$. As stated earlier, states 2 and 5 lie in $FAIL^{\phi}$, and thus for any state t of any FSM M', $M_1 \times M', \langle 2, t\rangle \models \phi$ iff $M_1 \times M', \langle 5, t\rangle \models \phi$ (i.e., by Proposition 3.6, neither $\langle 2, t\rangle$ nor $\langle 5, t\rangle$ satisfies ϕ). However, if we were to merge states 2 and 5 into a single state, states 1 and 4 would become equivalent. But, as discussed earlier, it would be wrong to have 1 and 4 be $\mathcal{E}^{\phi}$-equivalent.

3.5 Application of $\mathcal{E}^{\phi}$ to model checking

3.5.1 Compositional model checking

The equivalence relation $\mathcal{E}^{\phi}$ can be used to manage the size of the transition relations encountered in compositional model checking. The assumptions are that each component machine is relatively small and easy to manipulate, and that the full product machine is too large to build and manipulate. The general idea is to minimize each component machine, with respect to $\mathcal{E}^{\phi}$, before composing it with other machines. We can incrementally build the product machine by composing machines into clusters, and again applying minimization to each cluster. When just one machine remains, we apply a standard CTL model checker to determine the final result. Figure 3.7 outlines a procedure for this approach.

The question of how to minimize a component with respect to $\mathcal{E}^{\phi}$ depends on what sort of data representation is used for the transition relations.

3.5.1.1 Use of $\mathcal{E}^{\phi}$ with explicit representations

If an explicit representation is used (e.g., adjacency lists), then minimization is simply a matter of forming the quotient machine of each component with respect to $\mathcal{E}^{\phi}$.

Definition 3.10 Let $M = \langle S, \mathcal{X}, \Sigma_I, \mathcal{O}, O, T\rangle$ and let ϕ be a CTL formula. Impose an arbitrary total ordering on the elements of S. Let the equivalence classes of the equivalence relation $\mathcal{E}^{\phi} \subseteq S \times S$ be denoted by $S^{\phi} = \{c_1, c_2, ..., c_r\}$, where $c_i \subseteq S$. Define $\tilde{c}_i$ to be the *representative* of c_i, where $\tilde{c}_i$ is the least element of c_i. Then the *quotient machine* of M with respect to $\mathcal{E}^{\phi}$ is $M^{\phi} = (S^{\phi}, \mathcal{X}, \Sigma_I, \mathcal{O}, O^{\phi}, T^{\phi})$, where

```
compositionalModelChecker(φ, M_1, ..., M_n)
  if (n = 1)
    return modelChecker(φ, M_1);
  for (i = 1; i ≤ n; i++)
    M_i^* = minimize(M_i, φ);
  M'_1, ..., M'_l = formClusters(M_1^*, ..., M_n^*);
  return compositionalModelChecker(φ, M'_1, ..., M'_l);
```

Figure 3.7: Outline of procedure for compositional model checking: minimize and form product incrementally.

- $O^\phi : S^\phi \to 2^\mathcal{O}$ such that $O^\phi(c) = O(\tilde{c})$, and
- $T^\phi \subseteq S^\phi \times \Sigma_I \times S^\phi$ such that $(c, a, d) \in T^\phi$ iff $\exists x \in c$ and $\exists y \in d$ such that $(x, a, y) \in T$.

■

Note that the output of c depends on the ordering of the states of M. However, in the proofs to follow, we never assume anything about the ordering; that is, the output of c can be chosen to be the output of any state in the equivalence class c. Note also that we use $[s]$ to denote the equivalence class of s.

As described in the algorithm for the compositional model checker, we use the quotient machine of each FSM in place of the original component. The following theorem asserts that doing this does not alter the result returned by the model checker. The proof is given by Corollary 3.29 in Section 3.6.3.2.

Theorem 3.11 *Let ϕ be a CTL formula, and let $M_1, \ldots, M_n$ be FSMs. Let M_i^ϕ be the quotient of M_i with respect to $\mathcal{E}_i^\phi$, and let $[s_i]$ denote the equivalence class of $\mathcal{E}_i^\phi$ containing s_i. Then for all product states $\langle s_1, \ldots, s_n \rangle$,*

$$M_1 \times \ldots \times M_n, \langle s_1, \ldots, s_n \rangle \models \phi \quad \text{iff} \quad M_1^\phi \times \ldots \times M_n^\phi, \langle [s_1], \ldots, [s_n] \rangle \models \phi.$$

After the model checker is applied to the product of the quotient machines, Theorem 3.11 can be directly applied to recover the product states in the original state space that satisfy ϕ.

3.5.1.2 Use of $\mathcal{E}^\phi$ with implicit representations

If an implicit representation is used, then minimization becomes more complicated. We focus on the case where BDDs are used. There is no correlation between the size of the BDD for a transition relation, and the number of transitions in the relation. Thus, the idea behind minimization in this case is to use $\mathcal{E}^\phi$ to define a range of transition relations, any of which can be used in place of the original transition relation, and then choose the relation in this range with the smallest BDD. It should be noted however, that smaller component BDDs do not *guarantee* a smaller product BDD—this is only a heuristic.

For a component M, we take the upper bound of the range to be T^{max}, which is the relation formed by adding to T any transition between two states for which there exists a transition between equivalent states:

$$T^{max}(x,a,x') = T(x,a,x') \cup \{(x,a,x') | (s,a,s') \in T \text{ and } \mathcal{E}^\phi(x,s) \text{ and } \mathcal{E}^\phi(x',s')\}.$$

The lower bound is T itself. Given these bounds, a heuristic like *restrict* [42] is used to find a small BDD between T and T^{max}. Theorem 3.30 of Section 3.6.3.2 shows that any transition relation between T and T^{max} can be used without altering the result returned by the model checker. Alternatively, instead of looking for a small relation between T and T^{max}, we can just use T^{min}, which is the transition relation of the quotient machine, if it turns out that T^{min} is small.

3.5.2 Early pass/fail detection

Sometimes the model checking problem is posed as: given a formula ϕ and a subset of product states Q, is Q contained in the set of states satisfying ϕ? For example, Q may be the set of initial states.[3] Since our method returns all states satisfying ϕ, a simple containment check answers the question. However, in some cases, we may be able to answer

[3] If a set of initial states is known, then we can restrict our attention to the reachable state space. In this case, we can apply known techniques for exploiting the unreachable states, such as minimizing the transition relation with respect to unreachable states; these techniques are orthogonal to those discussed in this paper.

the question without composing all the machines, yielding a further savings in time. This is known as early pass/fail detection.

Let $Q = \{q^1, q^2, \ldots, q^m\}$, where q^j is the product state $\langle s_1^j, s_2^j, \ldots, s_n^j \rangle$, and let $FAIL_i^\phi$ be the $FAIL^\phi$ states in component i. If $s_i^j \in FAIL_i^\phi$, then any product state $\langle t_1, \ldots, t_{i-1}, s_i^j, t_{i+1}, \ldots, t_n \rangle$ does not satisfy ϕ, so in particular, q^j does not satisfy ϕ. Hence, the answer to the above question is "no". So in summary, if for any i, $FAIL_i^\phi$ intersects the ith state component of the set Q, then the answer is "no".

On the other hand, to reach an early "yes" answer, we need each state in Q to be "covered" by at least one $PASS^\phi$ state. If $s_i^j \in PASS_i^\phi$, then every state in Q with s_i^j as its ith component is guaranteed to satisfy ϕ. So in summary, if for every state in Q, at least one of its component states is a $PASS^\phi$ state, then the answer is "yes".

3.5.3 Processing subformulas

As the number of subformulas in ϕ increases, the equivalence $\mathcal{E}^\phi$ becomes finer because equivalence on all subformulas is required. However, if some of the subformulas of ϕ are first replaced by fresh atomic propositions representing the product states satisfying the subformulas, then this may lead to a coarser equivalence. This follows since knowing which product states satisfy a subformula adds information to what was originally known, information that can be used at the component level in computing $\mathcal{E}^\phi$ (for the new ϕ).

This is illustrated by the system in Figure 1.2, where $\phi = (\exists G(p \wedge q)) \wedge Q$, and Q is the set $\{\langle 1, 1' \rangle, \langle 2, 1' \rangle\}$ of product states. Lines 1 through 6 of Table 3.2 show the equivalence classes calculated for M_1 on the subformulas of ϕ. The end result (line 6) is that no states are equivalent; hence, we have gained nothing. Instead of processing all of ϕ, we could stop after computing the equivalence for $\exists G(p \wedge q)$. In this case, states 2 and 3 are equivalent (line 4), and thus a smaller machine can be built for M_1. When this quotient machine is composed with M_2 and the model checker is applied, we discover that no product states satisfy $\exists G(p \wedge q)$. At this point, we can create a fresh atomic proposition, Q', to represent this (empty) set of states. Then when we calculate the equivalence on M_1 for $Q' \wedge Q$ (which is the same as the original ϕ), we see that states 1 and 2 are now equivalent (line 8), so we can again construct a smaller machine for M_1.

Thus, we may want to follow a strategy where a nested formula is recursively decomposed into simpler subformulas, and the compositional model checker of Figure 3.7

	ϕ	$PASS^\phi$	$FAIL^\phi$	equiv classes
1	q	$\emptyset$	$\emptyset$	$\{1,2,3\}$
2	p	$\{2,3\}$	$\{1\}$	$\{1\},\{2,3\}$
3	$p \wedge q$	$\emptyset$	$\{1\}$	$\{1\},\{2,3\}$
4	$\exists G(p \wedge q)$	$\emptyset$	$\{1\}$	$\{1\},\{2,3\}$
5	Q	$\emptyset$	$\{3\}$	$\{1,2\},\{3\}$
6	$(\exists G(p \wedge q)) \wedge Q$	$\emptyset$	$\{1,3\}$	$\{1\},\{2\},\{3\}$
7	Q'	$\emptyset$	$\{1,2,3\}$	$\{1,2,3\}$
8	$Q' \wedge Q$	$\emptyset$	$\{1,2,3\}$	$\{1,2\},\{3\}$

Table 3.2: Equivalence classes for M_1 of Figure 1.2 on $(\exists G(p \wedge q)) \wedge Q$.

is applied to each subformula. Note that whereas Chiodo *et al.* [40] always recursively decompose a formula into its *immediate* subformulas, we can decompose a formula into *arbitrary* subformulas, since our equivalence works on nested formulas.

Of course, even though we may be able to compute coarser equivalences with this strategy, the drawback is that a reduced product machine must be constructed for each subformula. Experiments are required to determine how to decompose a formula to achieve a balance between these conflicting demands.

3.6 Proofs

3.6.1 Preliminaries

In the definitions, theorems and proofs that follow, x is used synonymously with x_0. For example, when we say "there exists a path $x_0 \rightarrow x_1 \rightarrow \ldots$", it is implicit that $x = x_0$. Also, $(x, y) \in \mathcal{R}^\phi$ means the same thing as $\mathcal{R}^\phi(x, y)$.

Fact 3.12 *Let M_1 and M_2 be two FSMs. If the path $\langle x_0, s_0\rangle \rightarrow \langle x_1, s_1\rangle \rightarrow \ldots$ exists in $M_1 \times M_2$, then the path $x_0 \rightarrow x_1 \rightarrow \ldots$ exists in M_1.*

3.6.2 $PASS^\phi$ and $FAIL^\phi$

This section establishes the truth of Proposition 3.6 of Section 3.4.2. For easy reference, we give the sets $\neg PASS^\phi$ and $\neg FAIL^\phi$.

Fact 3.13 *Let $M = \langle S, \mathcal{X}, \Sigma_I, \mathcal{O}, O, T\rangle$ and $M' = \langle (S', \mathcal{X}', \Sigma_I', \mathcal{O}', O', T'\rangle$ be FSMs, and let ϕ be a CTL formula.*

ϕ	$\neg PASS^\phi$	$\neg FAIL^\phi$
p_i	S	S
p_o	$\{x \in S \mid p \notin O(x)\}$	$\{x \in S \mid p \in O(x)\}$
p_s	$\{x \in S \mid \exists s' \in S', (x,s') \notin p_s\}$	$\{x \in S \mid \exists s' \in S', (x,s') \in p_s\}$
$\neg\psi$	$\neg FAIL^\psi$	$\neg PASS^\psi$
$\psi_1 \vee \psi_2$	$\neg PASS^{\psi_1} \cap \neg PASS^{\psi_2}$	$\neg FAIL^{\psi_1} \cup \neg FAIL^{\psi_2}$
$\exists X\psi$	$\{x \in S \mid \exists a \in \Sigma_I$ s.t. $\forall t, x \xrightarrow{a} t$ implies $t \in \neg PASS^\psi\}$	$\{x_0 \in S \mid$ there exists a path $x_0x_1x_2\ldots$, s.t. $x_1 \notin FAIL^\psi\}$
$\exists G\psi$	(not needed)	$\{x_0 \in S \mid$ there exists a path $x_0x_1x_2\ldots$, s.t. $\forall i \geq 0$, $x_i \notin FAIL^\psi\}$
$\exists[\psi_1 \; U \; \psi_2]$	(not needed)	$\{x_0 \in S \mid$ there exists a path $x_0x_1x_2\ldots$ and $\exists i \geq 0$ s.t. $x_i \notin FAIL^{\psi_2}$ and $\forall j < i$, $x_j \notin FAIL^{\psi_1}\}$

Also, it is worth keeping in mind the following two equivalences:

$\neg\exists[\psi_1 \; U \; \psi_2] \;\Leftrightarrow\; \forall([\neg\psi_2 \; U (\neg\psi_1 \wedge \neg\psi_2)] \vee G\neg\psi_2)$ *(not a CTL formula)*

$\neg\forall[\psi_1 \; U \; \psi_2] \;\Leftrightarrow\; \exists[\neg\psi_2 \; U (\neg\psi_1 \wedge \neg\psi_2)] \vee \exists G\neg\psi_2$

Proposition 3.14 *(Proposition 3.6 of Section 3.4.2) Let ϕ be a CTL formula, and let x be a state of M. If $x \in$ PASS$^\phi$, then for every FSM M', and every state t of M', $M \times M', \langle x,t\rangle \models \phi$. Likewise, if $x \in$ FAIL$^\phi$, then $M \times M', \langle x,t\rangle \not\models \phi$.*

Proof (by induction on the structure of ϕ)

Case $\phi = p_i$

Since $PASS^\phi = FAIL^\phi = \emptyset$, this case is vacuously true.

Case $\phi = p_o$

PASS: $x \in PASS^\phi$ implies $p_o \in O(x)$, which implies $p_o \in O(\langle x,t\rangle)$, which implies $\langle x,t\rangle \models \phi$.

FAIL: $x \in FAIL^\phi$ implies $p_o \notin O(x)$, which implies $p_o \notin O(\langle x,t\rangle)$, which implies $\langle x,t\rangle \not\models \phi$.

Case $\phi = p_s$

PASS: $x \in PASS^\phi$ implies $\forall s' \in S', (x,s') \in p_s$, which implies $(x,t) \in p_s$ which implies

$\langle x,t\rangle \models \phi$.

FAIL: $x \in FAIL^{\phi}$ implies $\forall s' \in S', (x,s') \notin p_s$, which implies $(x,t) \notin p_s$ which implies $\langle x,t\rangle \not\models \phi$.

Case $\phi = \neg\psi$

PASS: $x \in PASS^{\phi}$ implies $x \in FAIL^{\psi}$, which by the I.H. implies $\langle x,t\rangle \not\models \psi$, which implies $\langle x,t\rangle \models \phi$.

FAIL: $x \in FAIL^{\phi}$ implies $x \in PASS^{\psi}$, which by the I.H. implies $\langle x,t\rangle \models \psi$, which implies $\langle x,t\rangle \not\models \phi$.

Case $\phi = \psi_1 \vee \psi_2$

PASS: $x \in PASS^{\phi}$ implies $x \in PASS^{\psi_1}$ or $x \in PASS^{\psi_2}$, which by the I.H. implies $\langle x,t\rangle \models \psi_1$ or $\langle x,t\rangle \models \psi_2$, which implies $\langle x,t\rangle \models \psi_1 \vee \psi_2$.

FAIL: $x \in FAIL^{\phi}$ implies $x \in FAIL^{\psi_1}$ and $x \in FAIL^{\psi_2}$, which by the I.H. implies $\langle x,t\rangle \not\models \psi_1$ and $\langle x,t\rangle \not\models \psi_2$, which implies $\langle x,t\rangle \not\models \psi_1 \vee \psi_2$.

For the remaining formula types, the intuition is as follows. For the PASS case, if x has a "PASS path" in M, then in the presence of any environment, x will still have a PASS path, since the environment and M are required to be complete. For the FAIL case, if all the paths from x in M are "FAIL paths", then composing M with some environment may remove some paths from x, but whichever remain are still FAIL paths.

Case $\phi = \exists X\psi$

PASS: Assume $x \in PASS^{\phi}$. We must show that there exists a next state $\langle x',t'\rangle$ of $\langle x,t\rangle$ such that $\langle x',t'\rangle \models \psi$. By the completely specified assumption, every state has a next state, so let $\langle x',t'\rangle$ be a next state of $\langle x,t\rangle$. It remains to show that $\langle x',t'\rangle \models \psi$. Suppose $\langle x',t'\rangle \not\models \psi$, and suppose $x \xrightarrow{a} x'$. $x \in PASS^{\phi}$ implies there exists x'' such that $x \xrightarrow{a} x''$ and $x'' \in PASS^{\psi}$. By the I.H., $\langle x'',s\rangle \models \psi$, for all $s \in S'$. If $x'' = x'$, we have a contradiction. If $x'' \neq x'$, then $\langle x'',t'\rangle$ must also be a next state of $\langle x,t\rangle$. Since $\langle x'',t'\rangle \models \psi$, we are done.

FAIL: Assume $x \in FAIL^{\phi}$. Let $\langle x',t'\rangle$ be a next state of $\langle x,t\rangle$. Since $x \in FAIL^{\phi}$, then $x' \in FAIL^{\psi}$, which by the I.H. implies $\langle x',t'\rangle \not\models \psi$, which finally implies $\langle x,t\rangle \not\models \phi$.

Case $\phi = \exists G\psi$

PASS: Assume $x \in PASS^{\phi}$, and suppose that $\langle x,t\rangle \not\models \exists G\psi$. This implies that for every path $\langle x_0,t_0\rangle \rightarrow \langle x_1,t_1\rangle \rightarrow \ldots$, there exists k such that $\langle x_k,t_k\rangle \not\models \psi$. By the completely specified assumption, there must exist at least one such path. Let K be the maximum of all such k. The following claim shows that this leads to a contradiction, where $N = K$.

Claim: If $x \in PASS^{\phi}$, then for any $N \geq 0$, there exists a path[4] $\langle x_0,t_0\rangle \rightarrow \langle x_1,t_1\rangle \rightarrow \ldots \rightarrow \langle x_N,t_N\rangle$ such that for all $i \leq N$, $\langle x_i,t_i\rangle \models \psi$ and $x_i \in PASS^{\phi}$.

Base, N=0: We are given that $x \in PASS^{\phi}$. This implies $x \in PASS^{\psi}$, which by the I.H. of the proposition implies that $\langle x,s\rangle \models \psi$, for all $s \in S'$. Thus, $\langle x,t\rangle \models \psi$.

I.H.: For $k < N$, there exists a path $\langle x_0,t_0\rangle \rightarrow \langle x_1,t_1\rangle \rightarrow \ldots \rightarrow \langle x_k,t_k\rangle$ such that for all $i \leq k$, $\langle x_i,t_i\rangle \models \psi$ and $x_i \in PASS^{\phi}$.

I.S.: By the completely specified assumption, there exists a next state $\langle x_{k+1},t_{k+1}\rangle$ of $\langle x_k,t_k\rangle$. Suppose $\langle x_{k+1},t_{k+1}\rangle \not\models \psi$, and suppose $x_k \xrightarrow{a} x_{k+1}$. $x_k \in PASS^{\phi}$ implies there exists x'_{k+1} such that $x_k \xrightarrow{a} x'_{k+1}$ and $x'_{k+1} \in PASS^{\phi}$, and hence that $x'_{k+1} \in PASS^{\psi}$. By the I.H., $\langle x'_{k+1},s\rangle \models \psi$, for all $s \in S'$. If $x'_{k+1} = x_{k+1}$, we have a contradiction. If $x'_{k+1} \neq x_{k+1}$, then $\langle x'_{k+1},t_{k+1}\rangle$ is also a next state of $\langle x_k,t_k\rangle$. Since $\langle x'_{k+1},t_{k+1}\rangle \models \psi$, we are done. This finishes the claim, and hence proves that $\langle x,t\rangle \models \exists G\psi$.

FAIL: Let $\langle x_0,t_0\rangle \rightarrow \langle x_1,t_1\rangle \rightarrow \ldots$ be a path. Since $x \in FAIL^{\phi}$, there exists an $i \geq 0$ such that $x_i \in FAIL^{\psi}$. By the I.H., this implies that $\langle x_i,t_i\rangle \not\models \psi$. Since the path was arbitrary, then $\langle x,t\rangle \not\models \phi$.

Case $\phi = \exists[\psi_1 \; U \; \psi_2]$

PASS: Assume $x \in PASS^{\phi}$. Define $\tilde{R}_0 = R_0$, and for $i > 0$, $\tilde{R}_i = R_i \setminus R_{i-1}$, where R_i refers to the definition of $PASS^{\phi}$. Then for some i, $x \in \tilde{R}_i$. Suppose the fixed-point is reached at iteration N. Then for $i \leq N$, the following claim proves the proposition.

Claim: For all $x \in \tilde{R}_i$, and for all $t \in S'$, $\langle x,t\rangle \models \phi$.

[4] *Paths* were defined earlier to be infinite. However, in a few cases we also use the same term to refer to finite paths; the correct interpretation will be obvious from the context.

Base, i=0: $x \in \tilde{R}_0$ implies $x \in PASS^{\psi_2}$. By the I.H. of the proposition, this implies $\langle x,t\rangle \models \psi_2$ which in turn implies $\langle x,t\rangle \models \phi$.

I.H.: For $k < i$, for all $x \in \tilde{R}_k$, and for all $t \in S'$, $\langle x,t\rangle \models \phi$.

I.S.: $x \in \tilde{R_{k+1}}$ implies $x \notin R_0$, which implies $x \in PASS^{\psi_1}$. By the I.H. of the proposition, this implies $\langle x,t\rangle \models \psi_1$. By the completely specified assumption, there exists a next state $\langle x',t'\rangle$ of $\langle x,t\rangle$. If $x' \in R_k$, then $x' \in \tilde{R}_j$ for some $j < k$. Hence, by the I.H. $\langle x',t'\rangle \models \phi$, and therefore $\langle x,t\rangle \models \phi$. If $x' \notin R_k$, then suppose $x \xrightarrow{a} x'$. Since $x \in \tilde{R_{k+1}}$, then there exists $x'' \in R_k$ such that $x \xrightarrow{a} x''$, and therefore $\langle x'',t'\rangle$ is also a next state of $\langle x,t\rangle$. Since $x'' \in R_k$, then applying the I.H. as above, we have $\langle x,t\rangle \models \phi$.

FAIL: Let $\langle x_0,t_0\rangle \rightarrow \langle x_1,t_1\rangle \rightarrow \ldots$ be a path. Since $x \in FAIL^{\phi}$, either

1. there exists $i \geq 0$ such that $x_i \in FAIL^{\psi_1}$ and for all $j \leq i$, $x_j \in FAIL^{\psi_2}$, or
2. for all $i \geq 0$, $x_i \in FAIL^{\psi_2}$.

By the I.H., this implies

1. there exists $i \geq 0$ such that $\langle x_i,t_i\rangle \not\models \psi_1$ and for all $j \leq i$, $\langle x_j,t_j\rangle \not\models \psi_2$, or
2. for all $i \geq 0$, $\langle x_i,t_i\rangle \not\models \psi_2$.

Since the path was arbitrary, then $\langle x,t\rangle \not\models \phi$.

■

A state cannot be in both $PASS^{\phi}$ and $FAIL^{\phi}$.

Corollary 3.15 *$PASS^{\phi}$ and $FAIL^{\phi}$ are disjoint.*

Proof For the sake of contradiction, suppose there exists a state x such that $x \in PASS^{\phi} \cap FAIL^{\phi}$. Then by Proposition 3.14, for all t, $M \times M', \langle x,t\rangle \models \phi$ and $M \times M', \langle x,t\rangle \not\models \phi$. Since a formula and its negation cannot be true at the same state (because $x \models \neg\psi$ iff $x \not\models \psi$), this is a contradiction. ■

3.6.3 Formula-dependent equivalence relation

3.6.3.1 Formula-dependent bisimulation

We defined $\mathcal{E}^\phi$ in Definition 3.8 as "the coarsest equivalence relation" satisfying a certain property. However, it is not immediately obvious that this definition is sound, i.e., that there exists a unique such relation. This section establishes that the definition is indeed sound.

Proving this fact is not straightforward. Our development parallels that of Milner's development [27] showing that bisimulation is also "the coarsest equivalence relation" satisfying a certain property. We start by defining a *formula-dependent bisimulation* (FDB) in Definition 3.16, which looks like $\mathcal{E}^\phi$, but provides only a one-way implication. Lemma 3.17 then establishes an important relationship between FDBs and the sets of $PASS^\phi$ and $FAIL^\phi$ states. Lemma 3.19 proves that the identity relation is an FDB, and FDBs are closed under inverses, composition, and union. $\mathcal{E}^\phi$ is then actually defined as the union of all FDBs, and Lemma 3.21 shows that $\mathcal{E}^\phi$ is the largest FDB, and is an equivalence relation. Finally, Proposition 3.23 shows that a two-way implication holds for $\mathcal{E}^\phi$, by proving that $\mathcal{E}^\phi$ satisfies the converse of Definition 3.16.

Definition 3.16 Let $M = \langle S, \mathcal{X}, \Sigma_I, \mathcal{O}, O, T\rangle$ and $M' = \langle (S', \mathcal{X}', \Sigma_I', \mathcal{O}', O', T'\rangle$ be FSMs, and ϕ a CTL formula. A binary relation $\mathcal{R}^\phi \subseteq S \times S$ is a *formula-dependent bisimulation* (FDB) if $\mathcal{R}^\phi(x, y)$ implies:

Case $\phi = p_i$
$(x, y) \in S \times S$.

Case $\phi = p_o$
$x \in FAIL^\phi$ and $y \in FAIL^\phi$, or $x \in PASS^\phi$ and $y \in PASS^\phi$.

Case $\phi = p_s$
for all $s' \in S'$, $(x, s') \in p_s$ iff $(y, s') \in p_s$.

Case $\phi = \neg\psi$
there exists an FDB $\mathcal{R}^\psi$ such that $\mathcal{R}^\psi(x, y)$.

Cases $\phi = \psi_1 \vee \psi_2$ and $\phi = \psi_1 \wedge \psi_2$
there exist FDBs $\mathcal{R}^{\psi_1}$ and $\mathcal{R}^{\psi_2}$ such that $\mathcal{R}^{\psi_1}(x,y)$ and $\mathcal{R}^{\psi_2}(x,y)$.

Case $\phi = \exists X\psi$
there exists an FDB $\mathcal{R}^{\psi}$ such that $\mathcal{R}^{\psi}(x,y)$ and

1. $x \in FAIL^{\phi}$ and $y \in FAIL^{\phi}$, or $x \in PASS^{\phi}$ and $y \in PASS^{\phi}$, or
2. $O(x) = O(y)$, and for all $a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^{\psi}$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{R}^{\psi}(t,w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^{\psi}$, $\exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{R}^{\psi}(t,w)$.

Cases $\phi = \exists G\psi$ and $\phi = \exists F\psi$
there exists an FDB $\mathcal{R}^{\psi}$ such that $\mathcal{R}^{\psi}(x,y)$ and

1. $x \in FAIL^{\phi}$ and $y \in FAIL^{\phi}$, or $x \in PASS^{\phi}$ and $y \in PASS^{\phi}$, or
2. $O(x) = O(y)$, and for all $a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^{\phi}$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{R}^{\phi}(t,w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^{\phi}$, $\exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{R}^{\phi}(t,w)$.

Case $\phi = \exists[\psi_1 \; U \; \psi_2]$
there exist FDBs $\mathcal{R}^{\psi_1}$ and $\mathcal{R}^{\psi_2}$ such that $\mathcal{R}^{\psi_1}(x,y)$ and $\mathcal{R}^{\psi_2}(x,y)$ and

1. $x \in FAIL^{\phi}$ and $y \in FAIL^{\phi}$, or $x \in PASS^{\phi}$ and $y \in PASS^{\phi}$, or
2. $O(x) = O(y)$, and for all $a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^{\phi}$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{R}^{\phi}(t,w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^{\phi}$, $\exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{R}^{\phi}(t,w)$.

■

The following lemma implies that if the state pair (x,y) is in $\mathcal{R}^{\phi}$ and one of the states is in $PASS^{\phi}$, then the other state must also be in $PASS^{\phi}$; likewise for $FAIL^{\phi}$.

Lemma 3.17 *Let ϕ be a CTL formula, and suppose that $\mathcal{R}^{\phi}$ is an FDB such that $\mathcal{R}^{\phi}(x,y)$. Then $x \in$ PASS$^{\phi}$ implies $y \in$ PASS$^{\phi}$, and $x \in$ FAIL$^{\phi}$ implies $y \in$ FAIL$^{\phi}$.*

Proof (by induction on the structure of ϕ)

Case $\phi = p_i$
Since $PASS^\phi = FAIL^\phi = \emptyset$, this case is vacuously true.

Case $\phi = p_o$
$\mathcal{R}^\phi(x,y)$ implies $x \in PASS^\phi$ and $y \in PASS^\phi$, or $x \in FAIL^\phi$ and $y \in FAIL^\phi$. Since $x \in PASS^\phi$, then $x \in PASS^\phi$ and $y \in PASS^\phi$, which implies $y \in PASS^\phi$. Similarly if $x \in FAIL^\phi$.

Case $\phi = p_s$
$\mathcal{R}^\phi(x,y)$ implies that for all $s' \in S', (x,s') \in p_s$ iff $(y,s') \in p_s$. $x \in PASS^\phi$ implies $\forall s' \in S', (x,s') \in p_s$. Thus, $\forall s' \in S', (y,s') \in p_s$, and hence $y \in PASS^\phi$. Similarly if $x \in FAIL^\phi$.

Case $\phi = \neg\psi$
$\mathcal{R}^\phi(x,y)$ implies there exists an FDB $\mathcal{R}^\psi$ such that $\mathcal{R}^\psi(x,y)$. $x \in PASS^\phi$ implies $x \in FAIL^\psi$, which by the I.H. implies $y \in FAIL^\psi$, which then implies $y \in PASS^\phi$. Similarly if $x \in FAIL^\phi$.

Case $\phi = \psi_1 \vee \psi_2$
PASS: $\mathcal{R}^\phi(x,y)$ implies there exist FDBs $\mathcal{R}^{\psi_1}$ and $\mathcal{R}^{\psi_2}$ such that $\mathcal{R}^{\psi_1}(x,y)$ and $\mathcal{R}^{\psi_2}(x,y)$. $x \in PASS^\phi$ implies $x \in PASS^{\psi_1}$ or $x \in PASS^{\psi_2}$, which by the I.H. implies $y \in PASS^{\psi_1}$ or $y \in PASS^{\psi_2}$, which then implies $y \in PASS^\phi$.

FAIL: $x \in FAIL^\phi$ implies $x \in FAIL^{\psi_1}$ and $x \in FAIL^{\psi_2}$, which by the I.H. implies $y \in FAIL^{\psi_1}$ and $y \in FAIL^{\psi_2}$, which then implies $y \in FAIL^\phi$.

For the remaining cases, we proceed as follows. Assume that $\mathcal{R}^\phi(x,y)$ and $x \in PASS^\phi$—we want to show $y \in PASS^\phi$. $\mathcal{R}^\phi(x,y)$ holds because $x \in FAIL^\phi$ and $y \in FAIL^\phi$, or $x \in PASS^\phi$ and $y \in PASS^\phi$, or by condition 2 of Definition 3.16. By Corollary 3.15, $x \in PASS^\phi$ implies $x \notin FAIL^\phi$, and hence it cannot be the case that $x \in FAIL^\phi$ and $y \in FAIL^\phi$. If $\mathcal{R}^\phi(x,y)$ holds because $x \in PASS^\phi$ and $y \in PASS^\phi$, then $y \in PASS^\phi$ and we are done. Hence, we assume $y \notin PASS^\phi$, which implies that $\mathcal{R}^\phi(x,y)$ holds by condition 2, and proceed to show a contradiction. We proceed in a similar fashion to show that $\mathcal{R}^\phi(x,y)$

and $x \in FAIL^{\phi}$ implies $y \in FAIL^{\phi}$.

Case $\phi = \exists X \psi$

PASS: Assume $x \in PASS^{\phi}$ and $y \notin PASS^{\phi}$. Since $y \notin PASS^{\phi}$, there exists $a \in \Sigma_I$ such that whenever $y \xrightarrow{a} w$, then $w \notin PASS^{\psi}$. Let a' be such an a. Since $x \in PASS^{\phi}$, there exists t such that $x \xrightarrow{a'} t$ and $t \in PASS^{\psi}$. Since $\mathcal{R}^{\phi}(x, y)$ holds by condition 2 and $t \notin FAIL^{\psi}$, then there exists an FDB $\mathcal{R}^{\psi}$ and there exists w such that $y \xrightarrow{a'} w$ and $\mathcal{R}^{\psi}(t, w)$. But this implies, by the I.H., that $w \in PASS^{\psi}$, a contradiction. Hence, $y \in PASS^{\phi}$.

FAIL: Assume $x \in FAIL^{\phi}$ and $y \notin FAIL^{\phi}$. Since $y \notin FAIL^{\phi}$, there exists a next state w of y on a such that $w \notin FAIL^{\psi}$. Since $\mathcal{R}^{\phi}(x, y)$, there exists a next state t of x on a and an FDB $\mathcal{R}^{\psi}$ such that $\mathcal{R}^{\psi}(t, w)$. Since $x \in FAIL^{\phi}$, then $t \in FAIL^{\psi}$. But $\mathcal{R}^{\psi}(t, w)$ and $t \in FAIL^{\psi}$ imply, by the I.H., that $w \in FAIL^{\psi}$, a contradiction. Hence, $y \in FAIL^{\phi}$.

Case $\phi = \exists G \psi$

PASS: Assume $x \in PASS^{\phi}$ and $y \notin PASS^{\phi}$. The claim below shows that $y \in R_i, \forall i \geq 0$, where R_i refers to the definition of $PASS^{\phi}$. Since the fixed point is reached in a finite number of steps, then $y \in PASS^{\phi}$, which is a contradiction.

Claim: If $x \in PASS^{\phi}$ and there exists an FDB $\mathcal{R}^{\phi}$ such that $\mathcal{R}^{\phi}(x, y)$, then for all $i \geq 0$, $y \in R_i$.

Base, i=0: $\mathcal{R}^{\phi}(x, y)$ implies there exists an FDB $\mathcal{R}^{\psi}$ such that $\mathcal{R}^{\psi}(x, y)$. Also, $x \in PASS^{\phi}$ implies that $x \in PASS^{\psi}$. Thus, by the I.H. of the lemma, these facts imply that $y \in PASS^{\psi}$, which implies $y \in R_0$.

I.H.: For $k < i$, if $x \in PASS^{\phi}$ and there exists an FDB $\mathcal{R}^{\phi}$ such that $\mathcal{R}^{\phi}(x, y)$, then $y \in R_k$.

I.S.: To show that $y \in R_{k+1}$, we need to show

1. $y \in R_k$, and
2. $\forall a \in \Sigma_I, \exists w$ s.t. $y \xrightarrow{a} w$ and $w \in R_k$.

The first part follows by the I.H., since $x \in PASS^{\phi}$ and there exists an FDB $\mathcal{R}^{\phi}$ such that $\mathcal{R}^{\phi}(x, y)$. For the second part, let $a \in \Sigma_I$. Since $x \in PASS^{\phi}$, then there exists a next state t of x on a such that $t \in PASS^{\phi}$. If $y \in PASS^{\phi}$, then automatically $y \in R_{k+1}$, so we can assume that $\mathcal{R}^{\phi}(x, y)$ holds by condition 2. Hence, there exists a next state w of y on a such that $\mathcal{R}^{\phi}(t, w)$. Since $t \in PASS^{\phi}$ and there exists an FDB $\mathcal{R}^{\phi}$ such that $\mathcal{R}^{\phi}(t, w)$, then by the I.H. $w \in R_k$. Hence, $y \in R_{k+1}$.

FAIL: Assume $x \in FAIL^{\phi}$ and $y \notin FAIL^{\phi}$. Since $y \notin FAIL^{\phi}$, there exists a path $y_0 \xrightarrow{a_1} y_1 \xrightarrow{a_2} \ldots$ such that $y_i \notin FAIL^{\psi}$ for all i, which in turn implies that $y_i \notin FAIL^{\phi}$ for all i.

Since $x \in FAIL^{\phi}$, then every path $x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} \ldots$ from x must eventually reach a state x_k such that $x_k \in FAIL^{\psi}$. This implies that all states leading up to x_k must also be $FAIL^{\phi}$ states (because if not, then there would be a path from x that never reaches a $FAIL^{\psi}$ state). Since $\mathcal{R}^{\phi}(x, y)$ holds by condition 2 and $y_i \notin FAIL^{\phi}$ for all i, then repeated application of condition 2 shows that one of the paths $x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} \ldots$ is such that $\mathcal{R}^{\phi}(x_i, y_i)$ holds for all $i \leq k$, where $x_k \in FAIL^{\psi}$. But $\mathcal{R}^{\phi}(x_k, y_k)$ implies that there exists an FDB $\mathcal{R}^{\psi}$ such that $\mathcal{R}^{\psi}(x_k, y_k)$. But since $x_k \in FAIL^{\psi}$, then by the I.H., $y_k \in FAIL^{\psi}$, which is a contradiction.

Case $\phi = \exists[\psi_1 \; U \; \psi_2]$

PASS: Assume $x \in PASS^{\phi}$ and $y \notin PASS^{\phi}$. Since $x \in PASS^{\phi}$, then there exists $i \geq 0$ such that $x \in R_i$, where R_i refers to the definition of $PASS^{\phi}$. Therefore, by the following claim, $y \in R_i$, which implies $y \in PASS^{\phi}$, a contradiction.

Claim: For all $i \geq 0$, if $\mathcal{R}^{\phi}(x, y)$ and $x \in R_i$, then $y \in R_i$.

Base, i=0: $x \in R_0$ implies $x \in PASS^{\psi_2}$. $\mathcal{R}^{\phi}(x, y)$ implies there exists an FDB $\mathcal{R}^{\psi_2}$ such that $\mathcal{R}^{\psi_2}(x, y)$. Then by the I.H. of the lemma, $y \in PASS^{\psi_2}$, which implies $y \in R_0$.

I.H.: For $k < i$, if $\mathcal{R}^{\phi}(x, y)$ and $x \in R_k$, then $y \in R_k$.

I.S.: $x \in R_{k+1}$ implies one of the following:

- $x \in R_k$

- $x \in PASS^{\psi_1}$, and $\forall a \in \Sigma_I, \exists t$ s.t. $x \xrightarrow{a} t$ and $t \in R_k$.

If $x \in R_k$, then by the I.H. $y \in R_k$ and we are done. So assume $x \notin R_k$. To show $y \in R_{k+1}$, we will show that $y \in PASS^{\psi_1}$, and $\forall a \in \Sigma_I, \exists w$ s.t. $y \xrightarrow{a} w$ and $w \in R_k$. Since $x \in PASS^{\psi_1}$ and there exists an FDB $\mathcal{R}^{\psi_1}$ such that $\mathcal{R}^{\psi_1}(x, y)$, then by the I.H. $y \in PASS^{\psi_1}$. Now, let $a \in \Sigma_I$. Since $x \in R_{k+1}$, then there exists t s.t. $x \xrightarrow{a} t$ and $t \in R_k$. If $y \in R_{k+1}$ we are done, so assume that $\mathcal{R}^{\phi}(x, y)$ holds by condition 2 of the definition of FDB. Since $t \in R_k$, then $t \in PASS^{\phi}$, so condition 2 implies there exists w s.t. $y \xrightarrow{a} w$ and $\mathcal{R}^{\phi}(t, w)$. Thus, since $t \in R_k$ and $\mathcal{R}^{\phi}(t, w)$, then by the I.H. $w \in R_k$. Hence $y \in R_{k+1}$.

FAIL: Assume $x \in FAIL^{\phi}$ and $y \notin FAIL^{\phi}$. Since $y \notin FAIL^{\phi}$, there exists a finite path $y_0 \xrightarrow{a_1} y_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} y_n$ such that $y_i \notin FAIL^{\psi_1}$ for all $i < n$, and $y_n \notin FAIL^{\psi_2}$, which in turn implies $y_i \notin FAIL^{\phi}$ for all $i \leq n$.

Since $x \in FAIL^{\phi}$, then for every path $x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} \ldots$, one of the following must be true:

1. there exists $k \geq 0$ such that $x_k \in FAIL^{\psi_1}$ and $x_k \in FAIL^{\psi_2}$, and for all $i < k$, $x_i \in FAIL^{\psi_2}$, or
2. for all $i \geq 0$, $x_i \in FAIL^{\psi_2}$.

Since $\mathcal{R}^{\phi}(x, y)$ holds by condition 2 and $y_i \notin FAIL^{\phi}$ for all $i \leq n$, then repeated application of condition 2 shows that one of the paths $x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} \ldots$ is such that one of the following is true:

1. $\mathcal{R}^{\phi}(x_i, y_i)$ holds for all $i \leq k$. This implies that there exist FDBs $\mathcal{R}^{\psi_1}$ and $\mathcal{R}^{\psi_2}$ such that $\mathcal{R}^{\psi_1}(x_i, y_i)$ and $\mathcal{R}^{\psi_2}(x_i, y_i)$ for all $i \leq k$. We must consider 3 different ranges for k:
   (a) $k < n$: $x_k \in FAIL^{\psi_1}$ implies $x_i \in FAIL^{\psi_1}$ for *some* $i < n$, which by the I.H. implies $y_i \in FAIL^{\psi_1}$ for some $i < n$, which is a contradiction.
   (b) $k = n$: $x_n \in FAIL^{\psi_2}$ implies by the I.H. $y_n \in FAIL^{\psi_2}$, a contradiction.
   (c) $k > n$: Since for all $i \leq n$ $x_i \in FAIL^{\psi_2}$, by the I.H., $y_n \in FAIL^{\psi_2}$, a contradiction.
2. $\mathcal{R}^{\phi}(x_i, y_i)$ holds for all $i \leq n$. This implies that there exists an FDB $\mathcal{R}^{\psi_2}$ such that $\mathcal{R}^{\psi_2}(x_n, y_n)$. But since $x_n \in FAIL^{\psi_2}$, then by the I.H., $y_n \in FAIL^{\psi_2}$, which is a contradiction.

In each case we have a contradiction, and hence $y \in FAIL^{\phi}$.

■

Definition 3.18 Given binary relations $\mathcal{R}_i$ $(i = 1, 2, \ldots)$ over a set S, define:

1. $Id_S = \{(x, x) | x \in S\}$ (identity)
2. $\mathcal{R}_i^{-1} = \{(y, x) | (x, y) \in \mathcal{R}_i\}$ (inverse)
3. $\mathcal{R}_1\mathcal{R}_2 = \{(x, z) |$ for some $y, (x, y) \in \mathcal{R}_1$ and $(y, z) \in \mathcal{R}_2\}$ (composition)
4. $\mathcal{R}_1 \cup \mathcal{R}_2 = \{(x, y) | (x, y) \in \mathcal{R}_1$ or $(x, y) \in \mathcal{R}_2\}$ (union)

■

The following lemma shows that the identity relation is an FDB, and FDBs are closed under inverses, composition, and union.

Lemma 3.19 *Assume that each of $\mathcal{R}_i^{\phi}$ $(i = 1, 2, \ldots)$ is an FDB. Then the following relations are all FDBs:*

1. Id_S^{ϕ}
2. $\mathcal{R}_i^{\phi\,-1}$
3. $\mathcal{R}_1^{\phi}\mathcal{R}_2^{\phi}$.
4. $\bigcup_{i \in I} \mathcal{R}_i^{\phi}$, *for some index set I.*

Proof (by induction on the structure of ϕ)

Case $\phi = p_i$

1. Let $(x, x) \in Id_S^{\phi}$. Then $(x, x) \in S \times S$.
2. Let $(y, x) \in \mathcal{R}_i^{\phi\,-1}$. Then $(x, y) \in \mathcal{R}_i^{\phi}$, which implies that $(x, y) \in S \times S$, which implies that $(y, x) \in S \times S$.
3. Let $(x, z) \in \mathcal{R}_1^{\phi}\mathcal{R}_2^{\phi}$. Then for some y, we have $(x, y) \in \mathcal{R}_1^{\phi}$ and $(y, z) \in \mathcal{R}_2^{\phi}$. This implies that for some y, $(x, y) \in S \times S$ and $(y, z) \in S \times S$, which implies that $(x, z) \in S \times S$.

4. Let $(x,y) \in \bigcup_{i\in I} \mathcal{R}_i^\phi$. Then for some i, $(x,y) \in \mathcal{R}_i^\phi$, which implies that $(x,y) \in S \times S$.

Case $\phi = p_o$

1. Let $(x,x) \in Id_S^\phi$. If $x \in PASS^\phi$, then $x \in PASS^\phi$ and $x \in PASS^\phi$. Likewise if $x \in FAIL^\phi$.

2. Let $(y,x) \in \mathcal{R}_i^{\phi\,-1}$. Then $(x,y) \in \mathcal{R}_i^\phi$, which implies $x \in PASS^\phi$ and $y \in PASS^\phi$, or $x \in FAIL^\phi$ and $y \in FAIL^\phi$. By symmetry of conjunction, this implies $y \in PASS^\phi$ and $x \in PASS^\phi$, or $y \in FAIL^\phi$ and $x \in FAIL^\phi$.

3. Let $(x,z) \in \mathcal{R}_1^\phi \mathcal{R}_2^\phi$. Then for some y, we have $(x,y) \in \mathcal{R}_1^\phi$ and $(y,z) \in \mathcal{R}_2^\phi$. This implies that for some y, $x \in PASS^\phi$ and $y \in PASS^\phi$, or $x \in FAIL^\phi$ and $y \in FAIL^\phi$, and $y \in PASS^\phi$ and $z \in PASS^\phi$ or $y \in FAIL^\phi$, and $z \in FAIL^\phi$. This implies $x \in PASS^\phi$ and $y \in PASS^\phi$ and $z \in PASS^\phi$, or $x \in FAIL^\phi$ and $y \in FAIL^\phi$ and $z \in FAIL^\phi$, which in turn implies $x \in PASS^\phi$ and $z \in PASS^\phi$, or $x \in FAIL^\phi$ and $z \in FAIL^\phi$.

4. Let $(x,y) \in \bigcup_{i\in I} \mathcal{R}_i^\phi$. Then for some i, $(x,y) \in \mathcal{R}_i^\phi$, which implies that $x \in PASS^\phi$ and $y \in PASS^\phi$, or $x \in FAIL^\phi$ and $y \in FAIL^\phi$.

Case $\phi = p_s$

1. Let $(x,x) \in Id_S^\phi$. Trivially, for all $s' \in S', (x,s') \in p_s$ iff $(x,s') \in p_s$.

2. Let $(y,x) \in \mathcal{R}_i^{\phi\,-1}$. Then $(x,y) \in \mathcal{R}_i^\phi$, which implies for all $s' \in S', (x,s') \in p_s$ iff $(y,s') \in p_s$. By symmetry of "iff", this implies for all $s' \in S', (y,s') \in p_s$ iff $(x,s') \in p_s$.

3. Let $(x,z) \in \mathcal{R}_1^\phi \mathcal{R}_2^\phi$. Then for some y, we have $(x,y) \in \mathcal{R}_1^\phi$ and $(y,z) \in \mathcal{R}_2^\phi$. This implies that for some y, for all $s' \in S', (x,s') \in p_s$ iff $(y,s') \in p_s$, and for all $s' \in S', (y,s') \in p_s$ iff $(z,s') \in p_s$. By transitivity of "iff", this implies for all $s' \in S', (x,s') \in p_s$ iff $(z,s') \in p_s$.

4. Let $(x,y) \in \bigcup_{i\in I} \mathcal{R}_i^\phi$. Then for some i, $(x,y) \in \mathcal{R}_i^\phi$, which implies that for all $s' \in S', (x,s') \in p_s$ iff $(y,s') \in p_s$.

Case $\phi = \neg\psi$

1. Let $(x,x) \in Id_S^\phi$. Since $Id_S^\phi = Id_S^\psi$, then $(x,x) \in Id_S^\psi$. Thus, Id_S^ψ serves as the needed FDB $\mathcal{R}^\psi$.

2. Let $(y,x) \in \mathcal{R}_i^{\phi\,-1}$. Then $(x,y) \in \mathcal{R}_i^\phi$, which implies that there exists an FDB $\mathcal{R}_i^\psi$ such that $\mathcal{R}_i^\psi(x,y)$. This in turn implies $\mathcal{R}_i^{\psi\,-1}(y,x)$. By the I.H., $\mathcal{R}_i^{\psi\,-1}$ is an FDB, and hence serves as the needed FDB.

3. Let $(x,z) \in \mathcal{R}_1^\phi \mathcal{R}_2^\phi$. Then for some y, we have $(x,y) \in \mathcal{R}_1^\phi$ and $(y,z) \in \mathcal{R}_2^\phi$. This implies that there exist FDBs $\mathcal{R}_1^\psi$ and $\mathcal{R}_2^\psi$ such that $\mathcal{R}_1^\psi(x,y)$ and $\mathcal{R}_2^\psi(y,z)$. This in turn implies $(x,z) \in \mathcal{R}_1^\psi \mathcal{R}_2^\psi$. By the I.H., $\mathcal{R}_1^\psi \mathcal{R}_2^\psi$ is an FDB, and hence serves as the needed FDB.

4. Let $(x,y) \in \bigcup_{i\in I} \mathcal{R}_i^\phi$. Then for some i, $(x,y) \in \mathcal{R}_i^\phi$, which implies that there exists an FDB $\mathcal{R}_i^\psi$ such that $\mathcal{R}_i^\psi(x,y)$. Thus, $\mathcal{R}_i^\psi$ serves as the needed FDB.

Case $\phi = \psi_1 \vee \psi_2$

1. Let $(x,x) \in Id_S^\phi$. Since $Id_S^\phi = Id_S^{\psi_1} = Id_S^{\psi_2}$, then $(x,x) \in Id_S^{\psi_1}, Id_S^{\psi_2}$. Thus, $Id_S^{\psi_1}$ and $Id_S^{\psi_2}$ serve as the needed FDBs.

2. Let $(y,x) \in \mathcal{R}_i^{\phi\,-1}$. Then $(x,y) \in \mathcal{R}_i^\phi$, which implies that there exist FDBs $\mathcal{R}_i^{\psi_1}$ and $\mathcal{R}_i^{\psi_2}$ such that $\mathcal{R}_i^{\psi_1}(x,y)$ and $\mathcal{R}_i^{\psi_2}(x,y)$. This in turn implies $\mathcal{R}_i^{\psi_1\,-1}(y,x)$ and $\mathcal{R}_i^{\psi_2\,-1}(y,x)$. By the I.H., $\mathcal{R}_i^{\psi_1\,-1}$ and $\mathcal{R}_i^{\psi_2\,-1}$ are FDBs, and hence serve as the needed FDBs.

3. Let $(x,z) \in \mathcal{R}_1^\phi \mathcal{R}_2^\phi$. Then for some y, we have $(x,y) \in \mathcal{R}_1^\phi$ and $(y,z) \in \mathcal{R}_2^\phi$. This implies that there exist FDBs $\mathcal{R}_1^{\psi_1}$ and $\mathcal{R}_1^{\psi_2}$, and $\mathcal{R}_2^{\psi_1}$ and $\mathcal{R}_2^{\psi_2}$, such that $\mathcal{R}_1^{\psi_1}(x,y)$ and $\mathcal{R}_1^{\psi_2}(x,y)$, and $\mathcal{R}_2^{\psi_1}(y,z)$ and $\mathcal{R}_2^{\psi_2}(y,z)$. This in turn implies $(x,z) \in \mathcal{R}_1^{\psi_1}\mathcal{R}_2^{\psi_1}$ and $(x,z) \in \mathcal{R}_1^{\psi_2}\mathcal{R}_2^{\psi_2}$. By the I.H., $\mathcal{R}_1^{\psi_1}\mathcal{R}_2^{\psi_1}$ and $\mathcal{R}_1^{\psi_2}\mathcal{R}_2^{\psi_2}$ are FDBs, and hence serve as the needed FDBs.

4. Let $(x,y) \in \bigcup_{i\in I} \mathcal{R}_i^\phi$. Then for some i, $(x,y) \in \mathcal{R}_i^\phi$, which implies that there exist FDBs $\mathcal{R}_i^{\psi_1}$ and $\mathcal{R}_i^{\psi_2}$ such that $\mathcal{R}_i^{\psi_1}(x,y)$ and $\mathcal{R}_i^{\psi_2}(x,y)$. Thus, $\mathcal{R}_i^{\psi_1}$ and $\mathcal{R}_i^{\psi_2}$ serve as the needed FDBs.

Case $\phi = \exists X \psi$

We first must show that "there exists an FDB $\mathcal{R}^\psi$ such that $\mathcal{R}^\psi(x,y)$". This is done exactly as for the case $\phi = \neg\psi$. As a reminder, the needed FDB for each case is:

1. Id_S^ψ
2. $\mathcal{R}_i^{\psi\,-1}$
3. $\mathcal{R}_1^\psi \mathcal{R}_2^\psi$
4. $\mathcal{R}_i^\psi$

It remains to show that condition 1 or 2 is satisfied. We break this into case a, where $x \in FAIL^\phi$, and case b, where $x \notin FAIL^\phi$ and $x \notin PASS^\phi$. The case where $x \in PASS^\phi$ is similar to case a.

1. Let $(x, x) \in Id_S^\phi$.
   (a) $x \in FAIL^\phi$ implies $x \in FAIL^\phi$ and $x \in FAIL^\phi$.
   (b) $O(x) = O(x)$. Let a and t be such that $x \xrightarrow{a} t$ and $t \notin FAIL^\psi$. Then trivially, a and t are such that $x \xrightarrow{a} t$, and $Id_S^\psi(t, t)$.
2. Let $(y, x) \in \mathcal{R}_i^{\phi\,-1}$. Then $(x, y) \in \mathcal{R}_i^\phi$.
   (a) By Lemma 3.17, this implies $y \in FAIL^\phi$. Hence, $y \in FAIL^\phi$ and $x \in FAIL^\phi$.
   (b) This implies $O(x) = O(y)$, and $\forall a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^\psi$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{R}_i^\psi(t, w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^\psi$, $\exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{R}_i^\psi(t, w)$.

   By symmetry of equality and conjunction, this implies $O(y) = O(x)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^\psi$, $\exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{R}_i^\psi(t, w)$, and
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^\psi$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{R}_i^\psi(t, w)$.

   Finally, $\mathcal{R}_i^\psi(t, w)$ implies $\mathcal{R}_i^{\psi\,-1}(w, t)$. Hence, we have $O(y) = O(x)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^\psi$, $\exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{R}_i^{\psi\,-1}(w, t)$, and
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^\psi$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{R}_i^{\psi\,-1}(w, t)$.
3. Let $(x, z) \in \mathcal{R}_1^\phi \mathcal{R}_2^\phi$. Then for some y, we have $(x, y) \in \mathcal{R}_1^\phi$ and $(y, z) \in \mathcal{R}_2^\phi$.
   (a) By Lemma 3.17, we have $y \in FAIL^\phi$, and a second application of Lemma 3.17 gives $z \in FAIL^\phi$. Hence, $x \in FAIL^\phi$ and $z \in FAIL^\phi$.
   (b) $(x, y) \in \mathcal{R}_1^\phi$ and $x \notin FAIL^\phi$ and $x \notin PASS^\phi$ imply $O(x) = O(y)$, and $\forall a \in \Sigma_I$

- whenever $x \xrightarrow{a} t$ and $t \notin FAIL^{\psi}$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{R}_1^{\psi}(t, w)$, and
- whenever $y \xrightarrow{a} w$ and $w \notin FAIL^{\psi}$, $\exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{R}_1^{\psi}(t, w)$.

By Lemma 3.17, we have $y \notin FAIL^{\phi}$ and $y \notin PASS^{\phi}$. This and $(y, z) \in \mathcal{R}_2^{\phi}$ imply $O(y) = O(z)$, and

- whenever $y \xrightarrow{a} w$ and $w \notin FAIL^{\psi}$, $\exists v$ s.t. $z \xrightarrow{a} v$ and $\mathcal{R}_2^{\psi}(w, v)$, and
- $\forall a, v$ s.t. $z \xrightarrow{a} v$ and $v \notin FAIL^{\psi}$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{R}_2^{\psi}(w, v)$.

By transitivity of equality, $O(x) = O(z)$. To show the second part, let a and t be such that $x \xrightarrow{a} t$ and $t \notin FAIL^{\psi}$. This implies the existence of w such that $y \xrightarrow{a} w$ and $\mathcal{R}_1^{\psi}(t, w)$. By Lemma 3.17, we have $w \notin FAIL^{\psi}$. This in turn implies the existence of v such that $z \xrightarrow{a} v$ and $\mathcal{R}_2^{\psi}(w, v)$. Therefore, $\mathcal{R}_1^{\psi}(t, w)$ and $\mathcal{R}_2^{\psi}(w, v)$ imply $\mathcal{R}_1^{\psi}\mathcal{R}_2^{\psi}(t, v)$. Putting this all together, we have that $x \xrightarrow{a} t$ and $t \notin FAIL^{\psi}$ imply that there exists v such that $z \xrightarrow{a} v$ and $\mathcal{R}_1^{\psi}\mathcal{R}_2^{\psi}(t, v)$. Likewise, it can be shown that a next state v of z on a implies that there exists a next state t of x on a such that $\mathcal{R}_1^{\psi}\mathcal{R}_2^{\psi}(t, v)$.

4. Let $(x, y) \in \bigcup_{i \in I} \mathcal{R}_i^{\phi}$. Then for some i, $(x, y) \in \mathcal{R}_i^{\phi}$.

   (a) By Lemma 3.17, this implies $y \in FAIL^{\phi}$. Hence, $x \in FAIL^{\phi}$ and $y \in FAIL^{\phi}$.

   (b) We need to show that

   $O(x) = O(y)$, and $\forall a \in \Sigma_I$

   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^{\psi}$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{R}_i^{\psi}(t, w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^{\psi}$, $\exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{R}_i^{\psi}(t, w)$.

   But this is exactly what $\mathcal{R}_i^{\phi}(x, y)$ implies.

Case $\phi = \exists G\psi$

We first must show that "there exists an FDB $\mathcal{R}^{\psi}$ such that $\mathcal{R}^{\psi}(x, y)$". This is done exactly as for the case $\phi = \neg\psi$.

It remains to show that condition 1 or 2 is satisfied. We break this into case a, where $x \in FAIL^{\phi}$, and case b, where $\not x \in FAIL^{\phi}$ and $\not x \in PASS^{\phi}$. Case a is exactly the same as for $\phi = \exists X\psi$. Case b is nearly the same (just replace ψ by ϕ everywhere ψ occurs in $FAIL^{\psi}$ and $\mathcal{R}^{\psi}$).

Case $\phi = \exists[\psi_1 \; U \; \psi_2]$

The existence of FDBs $\mathcal{R}^{\psi_1}$ and $\mathcal{R}^{\psi_2}$ such that $\mathcal{R}^{\psi_1}(x,y)$ and $\mathcal{R}^{\psi_2}(x,y)$ can be shown as for case $\phi = \psi_1 \vee \psi_2$. The rest of the proof is the same as for the case $\phi = \exists G\psi$.

■

The formal definition of $\mathcal{E}^\phi$ is deceptively simple.

Definition 3.20 Define $\mathcal{E}^\phi = \bigcup\{\mathcal{R}^\phi | \mathcal{R}^\phi \text{ is an FDB}\}$. ■

Lemma 3.21 *1. $\mathcal{E}^\phi$ is the largest FDB.*

2. $\mathcal{E}^\phi$ is an equivalence relation.

Proof

1. By Lemma 3.19(4), $\mathcal{E}^\phi$ is an FDB and includes any other such.

2. refl.: For any $x \in S$, $\mathcal{E}^\phi(x,x)$ by Lemma 3.19(1), since $\mathcal{E}^\phi$ includes Id_S^ϕ.

   sym.: If $\mathcal{E}^\phi(x,y)$, then $\mathcal{R}^\phi(x,y)$ for some FDB $\mathcal{R}^\phi$. Hence $\mathcal{R}^{\phi\,-1}(y,x)$, and so $\mathcal{E}^\phi(y,x)$ by Lemma 3.19(2).

   trans.: If $\mathcal{E}^\phi(x,y)$ and $\mathcal{E}^\phi(y,z)$ then $\mathcal{R}_1^\phi(x,y)$ and $\mathcal{R}_2^\phi(y,z)$ for some FDBs $\mathcal{R}_1^\phi$ and $\mathcal{R}_2^\phi$. So $\mathcal{R}_1^\phi\mathcal{R}_2^\phi(x,z)$, and so $\mathcal{E}^\phi(x,z)$ by Lemma 3.19(3).

■

We have shown that $\mathcal{E}^\phi$ is an FDB. Now we want to show that $\mathcal{E}^\phi$ also satisfies the converse of Definition 3.16. Following Milner, we define a new relation $\mathcal{T}^\phi$ in terms of $\mathcal{E}^\phi$. Then we show that $\mathcal{T}^\phi$ and $\mathcal{E}^\phi$ are in fact equivalent.

Definition 3.22 $\mathcal{T}^\phi(x,y)$ iff:

Case $\phi = p_i$

$(x,y) \in S \times S$.

Case $\phi = p_o$

$x \in PASS^\phi$ and $y \in PASS^\phi$, or $x \in FAIL^\phi$ and $y \in FAIL^\phi$.

Case $\phi = p_s$
for all $s' \in S', (x, s') \in p_s$ iff $(y, s') \in p_s$.

Case $\phi = \neg\psi$
$\mathcal{E}^{\psi}(x, y)$.

Cases $\phi = \psi_1 \vee \psi_2$ and $\phi = \psi_1$ and ψ_2
$\mathcal{E}^{\psi_1}(x, y)$ and $\mathcal{E}^{\psi_2}(x, y)$.

Case $\phi = \exists X \psi$
$\mathcal{E}^{\psi}(x, y)$ and

1. $x \in FAIL^{\phi}$ and $y \in FAIL^{\phi}$, or $x \in PASS^{\phi}$ and $y \in PASS^{\phi}$, or
2. $O(x) = O(y)$, and $\forall a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^{\psi}, \exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{E}^{\psi}(t, w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^{\psi}, \exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{E}^{\psi}(t, w)$.

Cases $\phi = \exists G \psi$ and $\phi = \exists F \psi$
$\mathcal{E}^{\psi}(x, y)$ and

1. $x \in FAIL^{\phi}$ and $y \in FAIL^{\phi}$, or $x \in PASS^{\phi}$ and $y \in PASS^{\phi}$, or
2. $O(x) = O(y)$, and $\forall a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^{\phi}, \exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{E}^{\phi}(t, w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^{\phi}, \exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{E}^{\phi}(t, w)$.

Case $\phi = \exists[\psi_1 \; U \; \psi_2]$
$\mathcal{E}^{\psi_1}(x, y)$ and $\mathcal{E}^{\psi_2}(x, y)$ and

1. $x \in FAIL^{\phi}$ and $y \in FAIL^{\phi}$, or $x \in PASS^{\phi}$ and $y \in PASS^{\phi}$, or
2. $O(x) = O(y)$, and $\forall a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^{\phi}, \exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{E}^{\phi}(t, w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^{\phi}, \exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{E}^{\phi}(t, w)$.

■

Proposition 3.23 *$\mathcal{E}^\phi(x,y)$ iff $\mathcal{T}^\phi(x,y)$.*

Proof ($\Rightarrow$) For the base cases, $\mathcal{E}^\phi(x,y)$ directly implies $\mathcal{T}^\phi(x,y)$. Of the remaining cases, we show the case $\phi = \exists G\psi$ in detail; the rest of the cases are similar.

Case $\phi = \exists G\psi$
Since $\mathcal{E}^\phi$ is an FDB, $\mathcal{E}^\phi(x,y)$ implies:
there exists an FDB $\mathcal{R}^\psi$ such that $\mathcal{R}^\psi(x,y)$ and

1. $x \in FAIL^\phi$ and $y \in FAIL^\phi$, or $x \in PASS^\phi$ and $y \in PASS^\phi$, or
2. $O(x) = O(y)$, and $\forall a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^\phi$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{E}^\phi(t,w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^\phi$, $\exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{E}^\phi(t,w)$.

Let $\mathcal{R}'^\psi$ be such an FDB. Since $\mathcal{E}^\phi$ includes all FDBs, then $\mathcal{R}'^\psi(x,y)$ implies $\mathcal{E}^\psi(x,y)$. Hence, we have:
$\mathcal{E}^\psi(x,y)$ and

1. $x \in FAIL^\phi$ and $y \in FAIL^\phi$, or $x \in PASS^\phi$ and $y \in PASS^\phi$, or
2. $O(x) = O(y)$, and $\forall a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^\phi$, $\exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{E}^\phi(t,w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^\phi$, $\exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{E}^\phi(t,w)$.

By Definition 3.22, this implies $\mathcal{T}^\phi(x,y)$.
($\Leftarrow$) It suffices to show that $\mathcal{T}^\phi$ is an FDB, since $\mathcal{E}^\phi$ includes all FDBs. For the base cases, the definition for $\mathcal{T}^\phi$ directly implies the FDB definition. Of the remaining cases, we show the case $\phi = \exists G\psi$ in detail; the rest of the cases are similar.

Case $\phi = \exists G\psi$
By Definition 3.22, $\mathcal{T}^\phi(x,y)$ implies:
$\mathcal{E}^\psi(x,y)$ and

1. $x \in FAIL^\phi$ and $y \in FAIL^\phi$, or $x \in PASS^\phi$ and $y \in PASS^\phi$, or
2. $O(x) = O(y)$, and $\forall a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^\phi, \exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{E}^\phi(t, w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^\phi, \exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{E}^\phi(t, w)$.

Since $\mathcal{E}^\psi$ is an FDB, then this implies that there exists an FDB $\mathcal{R}^\psi$ such that $\mathcal{R}^\psi(x, y)$. Also, by the forward implication of this proposition, $\mathcal{E}^\phi(t, w)$ implies $\mathcal{T}^\phi(t, w)$. Thus, $\mathcal{T}^\phi(x, y)$ implies the following, and hence is an FDB:
there exists an FDB $\mathcal{R}^\psi$ such that $\mathcal{R}^\psi(x, y)$ and

1. $x \in FAIL^\phi$ and $y \in FAIL^\phi$, or $x \in PASS^\phi$ and $y \in PASS^\phi$, or
2. $O(x) = O(y)$, and $\forall a \in \Sigma_I$
   - whenever $x \xrightarrow{a} t$ and $t \notin FAIL^\phi, \exists w$ s.t. $y \xrightarrow{a} w$ and $\mathcal{T}^\phi(t, w)$, and
   - whenever $y \xrightarrow{a} w$ and $w \notin FAIL^\phi, \exists t$ s.t. $x \xrightarrow{a} t$ and $\mathcal{T}^\phi(t, w)$.

■

3.6.3.2 $\mathcal{E}^\phi$ preserves CTL

This section demonstrates that two states related by $\mathcal{E}^\phi$ have the same truth on ϕ, and that the quotient with respect to ϕ can be used in place of M in compositional model checking.

Fact 3.24 *For the cases $\phi = \exists G\psi$, and $\phi = \exists[\psi_1 \, U \, \psi_2]$, those states related by condition 2 of the definition of $\mathcal{E}^\phi$ are computed by the following greatest fixed-point computation:*

$$
\begin{aligned}
Q_0 &= \{(x, y) \in S \times S | O(x) = O(y)\} \\
Q_{i+1} &= Q_i \cap \{(x, y) \in S \times S | \forall a \in \Sigma_I, \\
&\quad (\forall t \in S \text{ s.t. } x \xrightarrow{a} t \text{ and } t \notin \mathrm{FAIL}^\phi, \exists w \in S \text{ s.t. } y \xrightarrow{a} w \text{ and } (t, w) \in Q_i), \text{ and} \\
&\quad (\forall w \in S \text{ s.t. } y \xrightarrow{a} w \text{ and } w \notin \mathrm{FAIL}^\phi, \exists t \in S \text{ s.t. } x \xrightarrow{a} t \text{ and } (t, w) \in Q_i)\}
\end{aligned}
$$

Proposition 3.25 *(Proposition 3.9 of Section 3.4.4) Let ϕ be a CTL formula, and let x and y be states of M such that $\mathcal{E}^\phi(x, y)$. Then for any state t of any FSM M',*

$$M \times M', \langle x, t \rangle \models \phi \text{ iff } M \times M', \langle y, t \rangle \models \phi.$$

Proof The proof is by induction on the structure of ϕ. We show that $M \times M', \langle x,t\rangle \models \phi$ implies $M \times M', \langle y,t\rangle \models \phi$. The converse holds by symmetry.

Case $\phi = p_i$

$\langle x,t\rangle \models p_i$ implies $\langle z,t\rangle \models p_i$ for all $z \in S$, which in turn implies $\langle y,t\rangle \models p_i$.

Case $\phi = p_o$

$\langle x,t\rangle \models p_o$ implies $p_o \in O(x)$. $\mathcal{E}^{\phi}(x,y)$ implies $p_o \in O(x)$ iff $p_o \in O(y)$, and thus $p_o \in O(y)$. Finally, this implies $\langle y,t\rangle \models p_o$.

Case $\phi = p_s$

$\langle x,t\rangle \models p_s$ implies $(x,t) \in p_s$. $\mathcal{E}^{\phi}(x,y)$ implies for all $s' \in S', (x,t') \in p_s$ iff $(y,t') \in p_s$. Thus, $(y,t) \in p_s$, which in turn implies $\langle y,t\rangle \models p_s$.

Case $\phi = \neg\psi$

$\langle x,t\rangle \models \phi$ implies $\langle x,t\rangle \not\models \psi$. $\mathcal{E}^{\phi}(x,y)$ implies $\mathcal{E}^{\psi}(x,y)$, which by the I.H. implies $\langle y,t\rangle \not\models \psi$, which in turn implies $\langle y,t\rangle \models \phi$.

Case $\phi = \psi_1 \vee \psi_2$

$\langle x,t\rangle \models \phi$ implies $\langle x,t\rangle \models \psi_1$ or $\langle x,t\rangle \models \psi_2$. $\mathcal{E}^{\phi}(x,y)$ implies $\mathcal{E}^{\psi_1}(x,y)$ and $\mathcal{E}^{\psi_2}(x,y)$, which by the I.H. implies $\langle y,t\rangle \models \psi_1$ or $\langle y,t\rangle \models \psi_2$, which in turn implies $\langle y,t\rangle \models \phi$.

Case $\phi = \exists X \psi$

$\langle x,t\rangle \models \phi$ implies that there exists a next state $\langle x',t'\rangle$ of $\langle x,t\rangle$ such that $\langle x',t'\rangle \models \psi$. Suppose $a \in \Sigma_I$ such that $x \xrightarrow{a} x'$ and $a = O'(s)$. Now, since $\langle x,t\rangle \models \phi$, then by Proposition 3.14, $x \notin FAIL^{\phi}$. Also, if $y \in PASS^{\phi}$, then the lemma follows, so assume $y \notin PASS^{\phi}$. Thus, $\mathcal{E}^{\phi}(x,y)$ holds by condition 2, and hence $O(x) = O(y)$ and there exists y' such that $y \xrightarrow{a} y'$ and $\mathcal{E}^{\psi}(x',y')$. This implies that $\langle y,t\rangle \rightarrow \langle y',t'\rangle$, and by the I.H. $\langle y',t'\rangle \models \psi$. Finally, this implies $\langle y,t\rangle \models \phi$.

Case $\phi = \exists G \psi$

$\langle x,t\rangle \models \phi$ implies that there exists a path $\langle x_0,t_0\rangle \rightarrow \langle x_1,t_1\rangle \rightarrow \ldots$ such that $\langle x_i,t_i\rangle \models \psi$ for all i. This implies by Proposition 3.14 that $x_i \notin FAIL^{\psi}$ for all i, which in turn implies

$x_i \notin FAIL^{\phi}$ for all i. Suppose $a_i \in \Sigma_I$ such that $x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} \ldots$ and $a_i = O'(t_{i-1})$.

For the sake of contradiction, suppose $\langle y, t\rangle \not\models \exists G\psi$. This implies that for every path $\langle y_0, s_0\rangle \rightarrow \langle y_1, s_1\rangle \rightarrow \ldots$, there exists k such that $\langle y_k, s_k\rangle \not\models \psi$. For each path, we are interested in the minimum such k. Since $\langle y_k, s_k\rangle \not\models \psi$, this implies that for all $i \leq k$, $\langle y_i, s_i\rangle \not\models \phi$, which by Proposition 3.14 implies $y_i \notin PASS^{\phi}$.

Since $\mathcal{E}^{\phi}(x, y)$ holds by condition 2 and $x_i \notin FAIL^{\phi}$ for all i, then repeated application of condition 2 shows that one of the paths $y_0 \xrightarrow{a_1} y_1 \xrightarrow{a_2} \ldots$ is such that $O(x_i) = O(y_i)$ and $\mathcal{E}^{\phi}(x_i, y_i)$ holds for all $i \leq k$. This implies that $\langle y_0, t_0\rangle \rightarrow \langle y_1, t_1\rangle \rightarrow \ldots \rightarrow \langle y_k, t_k\rangle$ is a path. By assumption, $\langle y_k, t_k\rangle \not\models \psi$. $\mathcal{E}^{\phi}(x_k, y_k)$ implies $\mathcal{E}^{\psi}(x_k, y_k)$; since $\langle x_k, t_k\rangle \models \psi$, then by the I.H., $\langle y_k, t_k\rangle \models \psi$, which is a contradiction. Hence, $\langle y, t\rangle \models \phi$.

Case $\phi = \exists[\psi_1 \; U \; \psi_2]$

$\langle x, t\rangle \models \phi$ implies that there exists a path $\langle x_0, t_0\rangle \rightarrow \langle x_1, t_1\rangle \rightarrow \ldots$ and some k such that $\langle x_k, t_k\rangle \models \psi_2$ and for all $j < k$, $\langle x_j, t_j\rangle \models \psi_1$. This implies by Proposition 3.14 that $x_k \notin FAIL^{\psi_2}$ and for all $j < k$, $x_j \notin FAIL^{\psi_1}$, which in turn implies $x_i \notin FAIL^{\phi}$ for all $i \leq k$. Suppose $a_i \in \Sigma_I$ such that $x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} \ldots$ and $a_i = O'(t_{i-1})$.

For the sake of contradiction, suppose $\langle y, t\rangle \not\models \exists[\psi_1 \; U \; \psi_2]$. This implies that for every path $\langle y_0, s_0\rangle \rightarrow \langle y_1, s_1\rangle \rightarrow \ldots$, one of the following is true:

1. $\exists l \geq 0$ such that $\langle y_l, s_l\rangle \not\models \psi_1 \vee \psi_2$ and $\forall j < l, \langle y_j, s_j\rangle \not\models \psi_2$ (for each path, we are interested in the minimum such l).

2. $\forall i \geq 0$, $\langle y_i, s_i\rangle \not\models \psi_2$.

This implies that for each path, one of the following is true:

1. $\exists l \geq 0$ such that $\forall j \leq l$, $y_j \notin PASS^{\phi}$,

2. $\forall i \geq 0$, $y_i \notin PASS^{\phi}$.

Since $\mathcal{E}^{\phi}(x, y)$ holds by condition 2 and $x_i \notin FAIL^{\phi}$ for all i, then repeated application of condition 2 shows that one of the paths $y_0 \xrightarrow{a_1} y_1 \xrightarrow{a_2} \ldots$ is such that one of the following is true:

1. If $l < k$, then $O(x_i) = O(y_i)$ and $\mathcal{E}^{\phi}(x_i, y_i)$ holds for all $i \leq l$. This implies that $\langle y_0, t_0\rangle \rightarrow \langle y_1, t_1\rangle \rightarrow \ldots \rightarrow \langle y_l, t_l\rangle$ is a path. By assumption, $\langle y_l, t_l\rangle \not\models \psi_1$. $\mathcal{E}^{\phi}(x_l, y_l)$

implies $\mathcal{E}^{\psi}(x_l, y_l)$; since $\langle x_l, t_l \rangle \models \psi_1$, then by the I.H., $\langle y_l, t_l \rangle \models \psi_1$, which is a contradiction.

If $l \geq k$, then $O(x_i) = O(y_i)$ and $\mathcal{E}^{\phi}(x_i, y_i)$ holds for all $i \leq k$, where $y_k \notin PASS^{\psi_2}$. This implies that $\langle y_0, t_0 \rangle \rightarrow \langle y_1, t_1 \rangle \rightarrow \ldots \rightarrow \langle y_k, t_k \rangle$ is a path. By assumption, $\langle y_k, t_k \rangle \not\models \psi_2$. $\mathcal{E}^{\phi}(x_k, y_k)$ implies $\mathcal{E}^{\psi}(x_k, y_k)$; since $\langle x_k, t_k \rangle \models \psi_2$, then by the I.H., $\langle y_k, t_k \rangle \models \psi_2$, which is a contradiction.

2. Same as the case for $l \geq k$ immediately above.

In each case, we have a contradiction. Hence, $\langle y, t \rangle \models \phi$.

■

Definition 3.26 Let $M = \langle S, \mathcal{X}, \Sigma_I, \mathcal{O}, O, T \rangle$, let ϕ be a CTL formula, and let $M^{\phi} = (S^{\phi}, \mathcal{X}, \Sigma_I, \mathcal{O}, O^{\phi}, T^{\phi})$ be the quotient machine of M induced by $\mathcal{E}^{\phi}$. Define $\hat{M} = (\hat{S}, \mathcal{X}, \Sigma_I, \mathcal{O}, \hat{O}, \hat{T})$ as follows:

- $\hat{S} = S \cup S^{\phi}$,
- $\hat{T} \subseteq \hat{S} \times \Sigma_I \times \hat{S}$ such that $(s, a, t) \in \hat{T}$ iff $(s, a, t) \in T$ or $(s, a, t) \in T^{\phi}$,
- $\hat{O} : \hat{S} \rightarrow 2^{\mathcal{O}}$ such that $\hat{O}(s) = O(s)$ if $s \in S$, and $\hat{O}(s) = O^{\phi}(s)$ if $s \in S^{\phi}$.

■

Intuitively, $\hat{M}$ is derived by "placing M and M^{ϕ} side-by-side", and considering the result as a single FSM. Let $\hat{\mathcal{E}}^{\phi}$ denote the equivalence computed on $\hat{M}$ with respect to ϕ. The following lemma says that states s and $[s]$ of $\hat{M}$ are related by $\hat{\mathcal{E}}^{\phi}$.

Lemma 3.27 *Let $M = \langle S, \mathcal{X}, \Sigma_I, \mathcal{O}, O, T \rangle$ and $M' = \langle (S', \mathcal{X}', \Sigma_I', \mathcal{O}', O', T' \rangle$ be FSMs, and let ϕ be a CTL formula. Let $\hat{M}$ refer to the machine defined in Definition 3.26. Then for all states $s \in S$, $(s, [s]) \in \hat{\mathcal{E}}^{\phi}$.*

Proof (by induction on the structure of ϕ)

For this lemma, we carefully state the inductive hypothesis: for every subformula ψ of ϕ, and for all $s \in S$, $(s, [s]) \in \hat{\mathcal{E}}^{\psi}$, where $[s]$ is always the equivalence class of s with respect to ϕ.

Case $\phi = p_i$

Since $\hat{\mathcal{E}}^{\phi}$ contains all pairs of states in $\hat{M}$, it contains $(s, [s])$.

Case $\phi = p_o$

If $p_o \in O(s)$, then for all t such that $\mathcal{E}^{\phi}(s,t)$, $p_o \in O(t)$. Hence, regardless of the representative used for $[s]$, we have $p_o \in O^{\phi}([s])$. Therefore, $(s,[s]) \in \hat{\mathcal{E}}^{\phi}$.

Likewise, if $p_o \notin O(s)$, then for all t such that $\mathcal{E}^{\phi}(s,t)$, $p_o \notin O(t)$. Hence, regardless of the representative used for $[s]$, we have $p_o \notin O^{\phi}([s])$. Therefore, $(s,[s]) \in \hat{\mathcal{E}}^{\phi}$.

Case $\phi = p_s$

Since $p_s \subseteq S \times S'$, we must first extend p_s to $\hat{p}_s \subseteq \hat{S} \times S'$. If $\langle s, s' \rangle \in S \times S'$, then $\langle s, s' \rangle \in \hat{p}_s$ iff $\langle s, s' \rangle \in p_s$. If $\langle [s], s' \rangle \in S^{\phi} \times S'$, then $\langle [s], s' \rangle \in \hat{p}_s$ iff for all $t \in S$ such that $\mathcal{E}^{\phi}(s,t)$, $\langle t, s' \rangle \in p_s$.

Now we are ready to prove this case. For all states $t \in S$, if $\mathcal{E}^{\phi}(s,t)$, then for all $s' \in S'$, $\langle s, s' \rangle \in p_s$ iff $\langle t, s' \rangle \in p_s$. Hence, $(s,[s]) \in \hat{\mathcal{E}}^{\phi}$.

Case $\phi = \neg\psi$

By the I.H., $(s,[s]) \in \hat{\mathcal{E}}^{\psi}$. But, since $\hat{\mathcal{E}}^{\psi} = \hat{\mathcal{E}}^{\phi}$, then trivially $(s,[s]) \in \hat{\mathcal{E}}^{\phi}$.

Case $\phi = \psi_1 \vee \psi_2$

By the I.H., $(s,[s]) \in \hat{\mathcal{E}}^{\psi_1}$ and $(s,[s]) \in \hat{\mathcal{E}}^{\psi_2}$. Hence $(s,[s]) \in \hat{\mathcal{E}}^{\phi}$.

We break each remaining case into three mutually exclusive subcases: $s \in FAIL^{\phi}$, $s \in PASS^{\phi}$, and $s \notin FAIL^{\phi} \cup PASS^{\phi}$.

Case $\phi = \exists X \psi$

$\underline{s \in FAIL^{\phi}}$: Let $a \in \Sigma_I$. and let $[t]$ be such that $[s] \xrightarrow{a} [t]$ in M^{ϕ}. We need to show that $[t] \in FAIL^{\psi}$. Since $[s] \xrightarrow{a} [t]$, there exists x and y such that $\mathcal{E}^{\phi}(s,x)$ and $\mathcal{E}^{\phi}(t,y)$ and $x \xrightarrow{a} y$ in M. Since $\mathcal{E}^{\phi}(s,x)$ and $s \in FAIL^{\phi}$, then by Lemma 3.17, $x \in FAIL^{\phi}$ and hence $y \in FAIL^{\psi}$. Then $\mathcal{E}^{\phi}(t,y)$ implies $\mathcal{E}^{\psi}(t,y)$, which implies $t \in FAIL^{\psi}$. By the I.H., $(t,[t]) \in \hat{\mathcal{E}}^{\psi}$, and thus $[t] \in FAIL^{\psi}$, which implies that $[s] \in FAIL^{\phi}$. Hence, $(s,[s]) \in \hat{\mathcal{E}}^{\phi}$.

$\underline{s \in PASS^{\phi}}$: Let $a \in \Sigma_I$. Then since $s \in PASS^{\phi}$, there exists t such that $s \xrightarrow{a} t$ and $t \in PASS^{\psi}$. Therefore, there exists $[t]$ such that $[s] \xrightarrow{a} [t]$. By the I.H., $(t,[t]) \in \hat{\mathcal{E}}^{\psi}$, and thus $[t] \in PASS^{\psi}$. Hence, $(s,[s]) \in \hat{\mathcal{E}}^{\phi}$.

$s \notin FAIL^\phi \cup PASS^\phi$: In this case, every state equivalent to s under $\mathcal{E}^\phi$ is related by condition 2 of the definition of $\mathcal{E}^\phi$. Therefore, for all x such that $\mathcal{E}^\phi(x,s)$, $O(x) = O(s)$, and thus $O(s) = O^\phi([s])$, regardless of the representative chosen for $[s]$.

Let $a \in \Sigma_I$, and let t be such that $s \xrightarrow{a} t$ and $t \notin FAIL^\psi$. Then there exists $[t]$ such that $[s] \xrightarrow{a} [t]$, and by the I.H., $(t,[t]) \in \hat{\mathcal{E}}^\psi$. This satisfies the first half of condition 2.

On the other hand, let $[t]$ be such that $[s] \xrightarrow{a} [t]$ and $[t] \notin FAIL^\psi$. Then there exists x and y such that $\mathcal{E}^\phi(s,x)$ and $\mathcal{E}^\phi(t,y)$ and $x \xrightarrow{a} y$. By the I.H., $(t,[t]) \in \hat{\mathcal{E}}^\psi$. Then $\mathcal{E}^\phi(t,y)$ implies $\hat{\mathcal{E}}^\psi(t,y)$, which implies by transitivity that $\hat{\mathcal{E}}^\psi([t],y)$. Then by Lemma 3.17, $y \notin FAIL^\psi$. Now we apply condition 2 to show that there exists v such that $s \xrightarrow{a} v$ and $\hat{\mathcal{E}}^\psi(y,v)$. By transitivity, $\hat{\mathcal{E}}^\psi([t],v)$. Hence $(s,[s]) \in \hat{\mathcal{E}}^\phi$.

Case $\phi = \exists G\psi$

$s \in FAIL^\phi$: For sake of contradiction, assume that $[s] \notin FAIL^\phi$. This implies that there exists a path $[w_0] \xrightarrow{a_1} [w_1] \xrightarrow{a_2} \ldots$, where $w_0 = s$, such that for all $i \geq 0$, $[w_i] \notin FAIL^\psi$. By the following claim, this implies that that there exists a path $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots$, where $s_0 = s$, such that for all $i \geq 0$, $s_i \notin FAIL^\psi$. But this contradicts the fact that $s \in FAIL^\phi$, and hence $[s] \in FAIL^\phi$.

Claim: For all $j \geq 0$, if $s_0 \in FAIL^\phi$ and there exists a path $[w_0] \xrightarrow{a_1} [w_1] \xrightarrow{a_2} \ldots \xrightarrow{a_j} [w_j]$, (where $w_0 = s_0$), such that for all $0 \leq i \leq j$, $[w_i] \notin FAIL^\psi$, then there exists a path $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_j} s_j$, where $s_0 = s$, such that for all $0 \leq i \leq j$, $s_i \notin FAIL^\psi$ and $\mathcal{E}^\phi(s_i, w_i)$.

Base, j=0: $[w_0] \notin FAIL^\psi$ implies $[s_0] \notin FAIL^\psi$, since $w_0 = s_0$. By the I.H. of the lemma, $(s_0,[s_0]) \in \hat{\mathcal{E}}^\psi$, and hence $s_0 \notin FAIL^\psi$. Since $w_0 = s_0$, then trivially $\mathcal{E}^\phi(s_0, w_0)$.

I.H.: For $k < j$, if $s_0 \in FAIL^\phi$ and there exists a path $[w_0] \xrightarrow{a_1} [w_1] \xrightarrow{a_2} \ldots \xrightarrow{a_k} [w_k]$, (where $w_0 = s_0$), such that for all $0 \leq i \leq k$, $[w_i] \notin FAIL^\psi$, then there exists a path $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_k} s_k$, where $s_0 = s$, such that for all $0 \leq i \leq k$, $s_i \notin FAIL^\psi$ and $\mathcal{E}^\phi(s_i, w_i)$.

I.S.: $[w_k] \xrightarrow{a_{k+1}} [w_{k+1}]$ implies there exists x and y such that $\mathcal{E}^\phi(x, w_k)$ and $\mathcal{E}^\phi(y, w_{k+1})$ and $x \xrightarrow{a_{k+1}} y$. Since $s_0 \in FAIL^\phi$ and for all $0 \leq i \leq k$, $s_i \notin FAIL^\psi$, then $s_k \in FAIL^\phi$. By transitivity, $\mathcal{E}^\phi(x, s_k)$, and hence $x \in FAIL^\phi$. Also, $\mathcal{E}^\phi(x, s_k)$ implies $\mathcal{E}^\psi(x, s_k)$, which

implies $x \notin FAIL^{\psi}$. Since M is completely specified, there exists s_{k+1} such that $s_k \xrightarrow{a_{k+1}} s_{k+1}$.

Now, since $x \notin FAIL^{\psi}$ and $x \in FAIL^{\phi}$, then $y \in FAIL^{\phi}$. Likewise, since $s_k \notin FAIL^{\psi}$ and $s_k \in FAIL^{\phi}$, then $s_{k+1} \in FAIL^{\phi}$. This implies that $\mathcal{E}^{\phi}(y, s_{k+1})$, which by transitivity, implies that $\mathcal{E}^{\phi}(s_{k+1}, w_{k+1})$. By the I.H. of the lemma $(w_{k+1}, [w_{k+1}]) \in \hat{\mathcal{E}}^{\psi}$, which implies that $w_{k+1} \notin FAIL^{\psi}$. Finally, by Lemma 3.17, this implies that $s_{k+1} \notin FAIL^{\psi}$.

$s \in PASS^{\phi}$: We show that $[s] \in PASS^{\phi}$ by proving the following claim, where R_i refers to the definition of $PASS^{\phi}$.

Claim: If $s \in PASS^{\phi}$, then for all $i \geq 0$, $[s] \in R_i$.

Base, i=0: $s \in PASS^{\phi}$ implies $s \in PASS^{\psi}$. By the I.H. of the lemma, $(s, [s]) \in \hat{\mathcal{E}}^{\psi}$, and hence $[s] \in PASS^{\psi}$. Thus, $[s] \in R_0$.

I.H.: For $k < i$, if $s \in PASS^{\phi}$, then $[s] \in R_k$.

I.S.: By the I.H., $[s] \in R_k$. Let $a \in \Sigma_I$. Since $s \in PASS^{\phi}$, then there exists t such that $s \xrightarrow{a} t$ and $t \in PASS^{\phi}$. Therefore $[s] \xrightarrow{a} [t]$, and since $t \in PASS^{\phi}$, by the I.H., $[t] \in R_k$.

$s \notin FAIL^{\phi} \cup PASS^{\phi}$: The following claim shows that $(s, [s]) \in \hat{\mathcal{E}}^{\phi}$ by showing that for all $i \geq 0$, $(s, [s]) \in Q_i$, where Q_i refers to the fixed point computation for $\hat{\mathcal{E}}^{\phi}$. Remember that $\mathcal{E}^{\phi}$ is already known.

Claim: For all $i \geq 0$, $(s, [s]) \in Q_i$.

Base, i=0: Since all the states in the equivalence class of s have the same outputs, then $O(s) = O^{\phi}([s])$. Hence, $(s, [s]) \in Q_0$.

I.H.: For $k < i$, $(s, [s]) \in Q_k$.

I.S.: By the I.H., $(s, [s]) \in Q_k$. Let $a \in \Sigma_I$. Let t be such that $s \xrightarrow{a} t$ and $t \notin FAIL^{\phi}$. Then there exists $[t]$ such that $[s] \xrightarrow{a} [t]$, and by the I.H., $(t, [t]) \in Q_k$. This satisfies the first half

of condition 2.

On the other hand, let $[t]$ be such that $[s] \xrightarrow{a} [t]$ and $[t] \notin FAIL^{\phi}$. Then there exists x and y such that $\mathcal{E}^{\phi}(s,x)$ and $\mathcal{E}^{\phi}(t,y)$ and $x \xrightarrow{a} y$. We need to show that $y \notin FAIL^{\phi}$ in order to apply condition 2. If $y \in FAIL^{\phi}$, then by Lemma 3.17, $t \in FAIL^{\phi}$. Then applying the above subcase, this implies that $[t] \in FAIL^{\phi}$, which is a contradiction. Hence, we can assume that $y \notin FAIL^{\phi}$. Applying condition 2, there exists v such that $s \xrightarrow{a} v$ and $\mathcal{E}^{\phi}(y,v)$, which by transitivity implies that $\mathcal{E}^{\phi}(t,v)$. This trivially implies that $(t,v) \in Q_k$, which combined with the I.H. that $(t,[t]) \in Q_k$, gives that $([t],v) \in Q_k$.

Case $\phi = \exists[\psi_1 \; U \; \psi_2]$

$\underline{s \in FAIL^{\phi}}$: (This proof is similar to the corresponding case for Lemma 3.17.) For sake of contradiction, assume that $[s] \notin FAIL^{\phi}$. This implies that there exists a path $[w_0] \xrightarrow{a_1} [w_1] \xrightarrow{a_2} \ldots$, where $w_0 = s$, and there exists $j \geq 0$ such that $[w_j] \notin FAIL^{\psi_2}$ and for all $i < j$, $[w_i] \notin FAIL^{\psi_1}$. The following claim shows that a path of all non-$FAIL^{\psi_1}$ states from $[w_0]$ can be matched by a path of all non-$FAIL^{\psi_1}$ states from s_0. Then we consider the fact that there exists j such that $[w_j] \notin FAIL^{\psi_2}$. By the I.H. of the lemma, $(w_j,[w_j]) \in \hat{\mathcal{E}}^{\psi_2}$, and hence $w_j \notin FAIL^{\psi_2}$. By the claim, $\mathcal{E}^{\phi}(s_j,w_j)$, which implies $\mathcal{E}^{\psi_2}(s_j,w_j)$, which in turn implies $s_j \notin FAIL^{\psi_2}$. This contradicts the fact that $s \in FAIL^{\phi}$, and hence $[s] \in FAIL^{\phi}$.

Claim: For all $j \geq 0$, if $s_0 \in FAIL^{\phi}$ and there exists a path $[w_0] \xrightarrow{a_1} [w_1] \xrightarrow{a_2} \ldots \xrightarrow{a_j} [w_j]$, (where $w_0 = s_0$), such that for all $0 \leq i \leq j$, $[w_i] \notin FAIL^{\psi_1}$, then there exists a path $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_j} s_j$, where $s_0 = s$, such that for all $0 \leq i \leq j$, $s_i \notin FAIL^{\psi_1}$ and $\mathcal{E}^{\phi}(s_i,w_i)$.

The proof is exactly the same as for the corresponding claim in the case $\phi = \exists G\psi$, except that ψ is replaced by ψ_1.

$\underline{s \in PASS^{\phi}}$: Since $s \in PASS^{\phi}$, then there exists $i \geq 0$ such that $s \in R_i$, where R_i refers to the definition of $PASS^{\phi}$. We show that $[s] \in PASS^{\phi}$ by proving the following claim.

Claim: For all $i \geq 0$, if $s \in R_i$, then $[s] \in R_i$.

$\underline{\text{Base, i=0:}}$ $s \in R_0$ implies $s \in PASS^{\psi_2}$. By the I.H. of the lemma, $(s,[s]) \in \hat{\mathcal{E}}^{\psi_2}$, and hence $s \in PASS^{\psi_2}$. Thus, $[s] \in R_0$.

I.H.: For $k < i$, if $s \in R_k$, then $[s] \in R_k$.

I.S.: To show that $[s] \in R_{k+1}$, we need to show that:

- $[s] \in R_k$, or
- $[s] \in PASS^{\psi_1}$, and $\forall a \in \Sigma_I, \exists [t]$ s.t. $[s] \xrightarrow{a} [t]$ and $[t] \in R_k$.

Let $s \in R_{k+1}$. If $s \in R_k$, then by the I.H., $[s] \in R_k$. So assume $s \notin R_k$. First, by the I.H. of the lemma, $(s, [s]) \in \hat{\mathcal{E}}^{\psi_1}$, which implies $[s] \in PASS^{\psi_1}$. Next, let $a \in \Sigma_I$. Since $s \notin R_k$, there exists t such that $s \xrightarrow{a} t$ and $t \in R_k$. Therefore $[s] \xrightarrow{a} [t]$, and since $t \in R_k$, by the I.H., $[t] \in R_k$.

$s \notin FAIL^{\phi} \cup PASS^{\phi}$: Same as for the corresponding subcase for the case $\phi = \exists G \psi$.

■

The following theorem states that M can be replaced by its quotient M^{ϕ} in compositional model checking.

Theorem 3.28 *Let $M = \langle S, \mathcal{X}, \Sigma_I, \mathcal{O}, O, T \rangle$ and $M' = \langle (S', \mathcal{X}', \Sigma_I', \mathcal{O}', O', T' \rangle$ be FSMs, and let ϕ be a CTL formula. Then for all product states $\langle s, s' \rangle$,*

$$M \times M', \langle s, s' \rangle \models \phi \text{ iff } M^{\phi} \times M', \langle [s], s' \rangle \models \phi.$$

Proof By Lemma 3.27, $(s, [s]) \in \hat{\mathcal{E}}^{\phi}$. Then applying Proposition 3.25, we have

$$\hat{M} \times M', \langle s, s' \rangle \models \phi \text{ iff } \hat{M} \times M', \langle [s], s' \rangle \models \phi.$$

Since M and M^{ϕ} do not interact with each other, we have

$$M \times M', \langle s, s' \rangle \models \phi \text{ iff } \hat{M} \times M', \langle s, s' \rangle \models \phi,$$

and

$$\hat{M} \times M', \langle [s], s' \rangle \models \phi \text{ iff } M^{\phi} \times M', \langle [s], s' \rangle \models \phi.$$

Combining this series of equivalences gives the theorem. ■

Note that a formula ϕ may directly refer to the states S of a component machine M via an atomic proposition $p_s \subseteq S \times S'$. If we want to model check ϕ on a system where M has been replaced by its quotient M^{ϕ}, then we must first modify ϕ by replacing p_s with a new atomic proposition p_s^{ϕ} that refers to the states in S^{ϕ} rather than S. In particular, $\langle [s], s' \rangle \in p_s^{\phi}$ iff for all $t \in S$ such that $\mathcal{E}^{\phi}(s, t)$, $\langle t, s' \rangle \in p_s$.

Corollary 3.29 *(Theorem 3.11 of Section 3.5.1.1) Let ϕ be a CTL formula, and let $M_1, \ldots, M_n$ be FSMs. Let M_i^ϕ be the quotient of M_i with respect to $\mathcal{E}_i^\phi$, and let $[s_i]$ denote the equivalence class of $\mathcal{E}_i^\phi$ containing s_i. Then for all product states $\langle s_1, \ldots, s_n \rangle$,*

$$M_1 \times \ldots \times M_n, \langle s_1, \ldots, s_n \rangle \models \phi \text{ iff } M_1^\phi \times \ldots \times M_n^\phi, \langle [s_1], \ldots, [s_n] \rangle \models \phi.$$

Proof If we can show that for any i

$$M_1 \times \ldots \times M_{i-1} \times M_i \times M_{i+1} \times \ldots \times M_n, \langle s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n \rangle \models \phi \text{ iff}$$
$$M_1 \times \ldots \times M_{i-1} \times M_i^\phi \times M_{i+1} \times \ldots \times M_n, \langle s_1, \ldots, s_{i-1}, [s_i], s_{i+1}, \ldots, s_n \rangle \models \phi,$$

then by applying this fact successively n times, the corollary is proved.

By associativity and commutativity of FSM composition, we have

$$M_1 \times \ldots \times M_{i-1} \times M_i \times M_{i+1} \times \ldots \times M_n, \langle s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n \rangle \models \phi \text{ iff}$$
$$(M_1 \times \ldots \times M_{i-1} \times M_{i+1} \times \ldots \times M_n) \times M_i, \langle (s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n), s_i \rangle \models \phi.$$

Then applying Theorem 3.28 gives

$$(M_1 \times \ldots \times M_{i-1} \times M_{i+1} \times \ldots \times M_n) \times M_i, \langle (s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n), s_i \rangle \models \phi. \text{ iff}$$
$$(M_1 \times \ldots \times M_{i-1} \times M_{i+1} \times \ldots \times M_n) \times M_i^\phi, \langle (s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n), [s_i] \rangle \models \phi,$$

and finally associativity and commutativity gives

$$(M_1 \times \ldots \times M_{i-1} \times M_{i+1} \times \ldots \times M_n) \times M_i^\phi, \langle (s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n), [s_i] \rangle \models \phi, \text{ iff}$$
$$M_1 \times \ldots \times M_{i-1} \times M_i^\phi \times M_{i+1} \times \ldots \times M_n, \langle s_1, \ldots, s_{i-1}, [s_i], s_{i+1}, \ldots, s_n \rangle \models \phi,$$

■

The following theorem shows that any transition relation between T and T^{max} can be used in place of T for compositional model checking.

Theorem 3.30 *Let ϕ be a CTL formula and let x be a state of M. Let $\tilde{M}$ be an FSM with the same specification as M, except that $T \subseteq \tilde{T} \subseteq T^{max}$. Then for any state t of any FSM M',*

$$M \times M', \langle x, t \rangle \models \phi \text{ iff } \tilde{M} \times M', \langle x, t \rangle \models \phi.$$

Proof (by induction on the structure of ϕ)

Case $\phi = p_i$

p_i is independent of M, and hence of $\tilde{M}$, so there is no change.

Case $\phi = p_o$

The outputs of M and $\tilde{M}$ are the same for each state, so there is no change.

Case $\phi = p_s$

The state space of M and $\tilde{M}$ are the same, so there is no change.

Case $\phi = \neg\psi$

$M \times M', \langle x, t\rangle \models \phi$ if and only if $M \times M', \langle x, t\rangle \not\models \psi$, which by induction is true if and only if $\tilde{M} \times M', \langle x, t\rangle \not\models \psi$, which is true if and only if $\tilde{M} \times M', \langle x, t\rangle \models \phi$.

Case $\phi = \psi_1 \vee \psi_2$

$M \times M', \langle x, t\rangle \models \phi$ if and only if $M \times M', \langle x, t\rangle \models \psi_1$ or $M \times M', \langle x, t\rangle \models \psi_2$, which by induction is true if and only if $\tilde{M} \times M', \langle x, t\rangle \models \psi_1$ or $\tilde{M} \times M', \langle x, t\rangle \models \psi_2$, which is true if and only if $\tilde{M} \times M', \langle x, t\rangle \models \phi$.

Case $\phi = \exists X \psi$

($\Rightarrow$) Trivial, since if a witness path exists in T, then it must also exist in $\tilde{T}$.

($\Leftarrow$) $\tilde{M} \times M', \langle x, t\rangle \models \phi$ implies that there exists a next state $\langle x', t'\rangle$ of $\langle x, t\rangle$ in $\tilde{M} \times M'$ such that $\tilde{M} \times M', \langle x', t'\rangle \models \psi$. Suppose $a \in \Sigma_I$ such that $x \xrightarrow{a} x'$ in $\tilde{T}$ and $a = O'(s)$. Then by the definition of T^{max}, there exists y, y' such that $y \xrightarrow{a} y'$ in T and $\mathcal{E}^{\phi}(x, y)$ and $\mathcal{E}^{\phi}(x', y')$.

By induction, $M \times M', \langle x', t'\rangle \models \psi$, which implies that $x' \notin FAIL^{\psi}$. Since $\mathcal{E}^{\phi}(x', y')$, which implies that $\mathcal{E}^{\psi}(x', y')$, we have that $y' \notin FAIL^{\psi}$.

Now suppose that $\mathcal{E}^{\phi}(x, y)$ by Condition 1 of the definition of $\mathcal{E}^{\phi}$. Since $y' \notin FAIL^{\psi}$, this implies that $y \notin FAIL^{\phi}$, which in turn implies that $x \notin FAIL^{\phi}$ (by Lemma 3.17), which finally implies that $x \in PASS^{\phi}$. Hence, $M \times M', \langle x, t\rangle \models \phi$, for any M' and t.

Now suppose that $\mathcal{E}^{\phi}(x, y)$ by Condition 2. Since $y' \notin FAIL^{\psi}$, this implies that there exists x'' such that $x \xrightarrow{a} x''$ in T and $\mathcal{E}^{\psi}(y', x'')$. Since $\mathcal{E}^{\psi}(x', y')$ and $M \times M', \langle x', t'\rangle \models \psi$, then by Proposition 3.25, $M \times M', \langle y', t'\rangle \models \psi$. Hence, again by Proposition 3.25, $M \times M', \langle x'', t'\rangle \models \psi$, which finally implies that $M \times M', \langle x, t\rangle \models \phi$.

Case $\phi = \exists G\psi$

($\Rightarrow$) Trivial, since if a witness path exists in T, then it must also exist in $\tilde{T}$.

($\Leftarrow$) $\tilde{M} \times M', \langle x, t\rangle \models \phi$ implies that there exists a path $\langle x_0, t_0\rangle \rightarrow \langle x_1, t_1\rangle \rightarrow \ldots$ in $\tilde{M} \times M'$, where $x_0 = x$ and $t_0 = t$, such that $\tilde{M} \times M', \langle x_i, t_i\rangle \models \psi$ for all $i \geq 0$. By induction, this implies that for all i, $M \times M', \langle x_i, t_i\rangle \models \psi$, which by Proposition 3.14 implies that $x_i \notin FAIL^{\psi}$, which in turn implies $x_i \notin FAIL^{\phi}$. Suppose $a_i \in \Sigma_I$ such that $x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} \ldots$ in $\tilde{T}$, and $a_i = O'(t_{i-1})$.

We now prove the following claim by induction.

Claim: There exists a path $x'_0 \xrightarrow{a_1} x'_1 \xrightarrow{a_2} \ldots$ in T, where $x_0 = x$, and for all $i \geq 0$, $O(x'_i) = O(x_i)$ and $\mathcal{E}^{\phi}(x'_i, x_i)$; or, there exists a path $x'_0 \xrightarrow{a_1} x'_1 \xrightarrow{a_2} \ldots \xrightarrow{a_k} x'_k$ in T, where $x_0 = x$, and for all $i < k$, $O(x'_i) = O(x_i)$ and $\mathcal{E}^{\phi}(x'_i, x_i)$, and $x'_k \in PASS^{\phi}$.

Base, i=0: Suppose $x_0 = x'_0 \notin PASS^{\phi}$. Trivially, $O(x'_0) = O(x_0)$ and $\mathcal{E}^{\phi}(x'_0, x_0)$.

I.H.: For $l < i$, there exists a path $x'_0 \xrightarrow{a_1} x'_1 \xrightarrow{a_2} \ldots \xrightarrow{a_l} x'_l$ in T, where $x_0 = x$, and for all $i < l$, $O(x'_i) = O(x_i)$, and for all $i \leq l$, $\mathcal{E}^{\phi}(x'_i, x_i)$.

I.S.: If $x'_l \in PASS^{\phi}$, then the claim is proven. Suppose $x'_l \notin PASS^{\phi}$. We must extend the path from x'_l to x'_{l+1}. Now, $x_l \xrightarrow{a_{l+1}} x_{l+1}$ in $\tilde{T}$ implies that there exists y, z such that $y \xrightarrow{a_{l+1}} z$ and $\mathcal{E}^{\phi}(x_l, y)$ and $\mathcal{E}^{\phi}(x_{l+1}, z)$. By I.H., $\mathcal{E}^{\phi}(x'_l, x_l)$, and hence by transitivity, $\mathcal{E}^{\phi}(x'_l, y)$.

Since $x_l \notin FAIL^{\phi}$ and $\mathcal{E}^{\phi}(x'_l, x_l)$, then $x'_l \notin FAIL^{\phi}$. Thus, $\mathcal{E}^{\phi}(x'_l, y)$ holds by Condition 2 of the definition of $\mathcal{E}^{\phi}$. First, this implies that $O(x'_l) = O(y) = O(x_l)$. Next, $x_{l+1} \notin FAIL^{\phi}$ implies $z \notin FAIL^{\phi}$ (by Lemma 3.17), and thus $y \xrightarrow{a_{l+1}} z$ and $\mathcal{E}^{\phi}(x'_l, y)$ imply that there exists x'_{l+1} such that $x'_l \xrightarrow{a_{l+1}} x'_{l+1}$ in T and $\mathcal{E}^{\phi}(x'_{l+1}, z)$. By transitivity, this implies $\mathcal{E}^{\phi}(x'_{l+1}, x_{l+1})$. This completes the claim.

Now the proposition follows. If the claim is true by the first part, then since the outputs of x_i and x'_i agree, there exists a path $\langle x'_0, t_0\rangle \rightarrow \langle x'_1, t_1\rangle \rightarrow \ldots$ in $M \times M'$. Since $\mathcal{E}^{\phi}(x'_i, x_i)$, then $\mathcal{E}^{\psi}(x'_i, x_i)$. Since $M \times M', \langle x_i, t_i\rangle \models \psi$, then by Proposition 3.25, $M \times M', \langle x'_i, t_i\rangle \models \psi$, for all $i \geq 0$. Hence, $M \times M', \langle x_0, t_0\rangle \models \phi$.

If the claim is true by the second part, then the reasoning is the same except that $x'_k \in PASS^{\phi}$ implies that $M \times M', \langle x'_k, t_k\rangle \models \phi$, which along with $M \times M', \langle x'_i, t_i\rangle \models \psi$ for $i < k$, implies $M \times M', \langle x_0, t_0\rangle \models \phi$.

Case $\phi = \exists[\psi_1 \; U \; \psi_2]$

($\Rightarrow$) Trivial, since if a witness path exists in T, then it must also exist in $\tilde{T}$.

($\Leftarrow$) $\tilde{M} \times M', \langle x, t\rangle \models \phi$ implies that there exists a path $\langle x_0, t_0\rangle \rightarrow \langle x_1, t_1\rangle \rightarrow \ldots \rightarrow \langle x_k, t_k\rangle$ in $\tilde{M} \times M'$, where $x_0 = x$ and $t_0 = t$, such that for all $i < k$, $\tilde{M} \times M', \langle x_i, t_i\rangle \models \psi_1$, and $\tilde{M} \times M', \langle x_k, t_k\rangle \models \psi_2$. By induction, this implies that for all $i < k$, $M \times M', \langle x_i, t_i\rangle \models \psi_1$, which by Proposition 3.14 implies that $x_i \notin FAIL^{\psi_1}$, which in turn implies $x_i \notin FAIL^{\psi_1}$. Likewise, $x_k \notin FAIL^{\psi_2}$. Suppose $a_i \in \Sigma_I$ such that $x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} \ldots \xrightarrow{a_k} x_k$ in $\tilde{T}$, and $a_i = O'(t_{i-1})$.

The rest of the proof follows the case for $\phi = \exists G\psi$. ∎

3.7 Summary and future work

We have presented a formula-dependent equivalence that can be used to manage the size of the transition relations encountered in compositional CTL model checking. We have yet to implement the method, and the ultimate effectiveness of the method can be confirmed only by experimentation. Given an arbitrary CTL formula ϕ, the method works by first computing an equivalence, which preserves ϕ, on the states of each component machine. If an explicit representation for transition relations is used, then the quotient machine is constructed for each component, and the quotient machines are used to build a smaller product machine.

If BDDs are used, then the equivalence for each component is used to determine a range of permissible transition relations. More work remains to derive a procedure for efficiently choosing a relation from this range that will ultimately lead to a smaller product machine.

Our approach can be applied incrementally to build the product machine by clustering some minimized machines, forming their product, and repeating the equivalence computation. Research is needed to understand how best to cluster the components to achieve the smallest sub-products. Also, we outlined how our approach can be applied to the subformulas of a formula, to achieve a coarser equivalence. We need to devise a heuristic to intelligently decompose a formula into subformulas to take advantage of this.

An important part of a CTL model checker is the ability to generate counter-examples. Since we are altering the product machine, a counter-example in the altered product may not actually exist in the full product. A method needs to be developed to

handle this. Finally, we plan to extend our method to fair-CTL model checking, and we would like to apply similar ideas to the language containment paradigm.

Chapter 4

Deciding State Reachability for Large FSMs

4.1 Introduction

We are concerned with the problem of determining if there exists a path, in the state transition graph of a system of interacting FSMs, from a given set of initial states I to a final set F. We call this the *state reachability problem.* This problem is more specific than the usual problem of determining the set of states R reachable from I. Obviously, if R is known, then by checking if R and F intersect, the state reachability problem can be answered. However, it may not be necessary to compute R to decide state reachability.

Finding efficient algorithms to solve state reachability is crucial because several problems in verification, logic synthesis and testing can be efficiently reduced to the state reachability problem. For example, to determine if two FSMs M_1 and M_2 are equivalent, we can define the set of initial states I to be those product states that are initial in both M_1 and M_2, and F to be those products states where the output values of M_1 and M_2 differ. Then, M_1 and M_2 are equivalent if and only if I cannot reach F in the product of M_1 and M_2. As another example, checking safety properties specified as automata over finite strings can be reduced to state reachability. In particular, if a monitor T is defined that enters a BAD state when a property is violated, then an FSM M satisfies the property if and only if the BAD states cannot be reached from the initial states, in the product of M and T. In summary, an efficient solution to the state reachability problem would provide

efficient solutions to a host of other CAD problems.

When an FSM is described as a set of interacting FSMs, the state reachability problem is PSPACE-complete [26]. Even so, algorithms based on symbolic breadth-first traversal using BDDs can handle FSMs with several hundred flip-flops. However, on larger examples, the standard approaches start to falter because

1. the BDD representing the set of states reached at an intermediate step grows too large, or

2. the image of a given set of states cannot be computed.

This work does not address the first problem directly, but instead focuses on the second problem; in doing so, we aim to increase the size of FSMs that can be analyzed.

To understand the idea behind our approach, consider the state transition graph G of an FSM M, representing a set of interacting FSMs. We assume that G is too large to build and analyze directly. Instead, we make a series of over- and under-approximations to G, where with each approximation, we attempt to narrow in on a path from I to F, or prove that such a path cannot exist. An over-approximation of G is a graph containing a superset of the edges[1] of G, and an under-approximation of G is a graph containing a subset.

Consider an over-approximation V to G, and restrict V to those transitions lying on a path from I to F. If there is a path in G from I to F, then this path must exist in the restricted V. Now, consider an under-approximation U. Denote by I' all those states that are reachable from I in U, and by F' all those states that can reach F in U. If I' and F' intersect, then certainly I can reach F in G, because, by definition, all of the transitions in U are in G. On the other hand, if I' and F' do not intersect, then we try to extend the frontier of I' by looking for true transitions (i.e., those in G) among those in V that lead from states in I'. If no such transition can be found in V, then we have proven that I cannot reach F in G. We also try to work backwards from F' at the same time, in a similar manner. In summary, V is used to guide the search in U for a path from I to F.

The feasibility of our approach is predicated upon finding approximations to G that have reasonable BDD sizes, but yet are close enough approximations to G to permit useful information to be derived. To this end, we introduce two new as BDD operators,

[1]The terms *edge* and *transition* are used interchangeably.

called *bddOverApprox* and *bddUnderApprox*. Consider the BDD F representing a function f. *BddUnderApprox* selectively replaces some subgraphs in F by the constant ZERO, yielding a new BDD G representing the function g, such that G has fewer nodes than F, and $g \subseteq f$. Nodes are selected for replacement based on a cost function that takes a parameter that controls the tradeoff between reducing the BDD node count, and reducing the onset size. In a similar manner, *bddOverApprox* adds minterms to a function by replacing BDD nodes by the constant ONE. Ravi and Somenzi [4] independently and concurrently formulated the same BDD approximation problem, although they apply it in a different situation.

The main contribution of this work is a BDD-based algorithm to solve the state reachability problem, via a series of under- and over-approximations to the state transition graph. To our knowledge, the idea of using both under- and over- approximations in this domain is novel. Also, we define, and give a heuristic solution to, the new problem of approximating Boolean functions to yield a small BDD.

4.2 The state reachability problem

FSMs were defined in Section 1.1.2. For the state reachability problem, the outputs of an FSM are of no interest, and hence we omit them from the specification of an FSM. Thus, an FSM is specified as a 5-tuple, $M = \langle S, I, \mathcal{X}, \Sigma_I, T \rangle$, where:

- S is the state space of size 2^l, spanned by the binary variables $x = [x_1, x_2, \ldots, x_l]$. We also introduce a second set of variables $y = [y_1, y_2, \ldots, y_l]$ to denote the next state.
- I is a subset of S, denoting the initial states.
- $\mathcal{X}$ is the set of n inputs, with associated binary variables $u = [u_1, u_2, \ldots, u_n]$.
- $\Sigma_I = \mathbb{B}^n$ is the input alphabet.
- T is the next state function, $T : S \times \Sigma_I \rightarrow S$. T is presented as a vector of l Boolean functions, $\delta = [\delta_1, \delta_2, \ldots, \delta_l]$, where δ_i is the next state function of the state variable x_i. $T_i = (y_i \equiv \delta_i(u, x))$ gives the corresponding transition relation of x_i, and $T = \prod_{i=1}^{l} T_i$. δ is typically given as a multi-level logic network.

The *state transition graph* $G(x, y)$ of M is a binary relation over S defined by

$$G(x, y) = \exists u \prod_{i=1}^{l} T_i(x, u, y_i).$$

That is, $G(x,y) = 1$ if and only if there exists an input u such that from state $x = [x_1, x_2, \ldots, x_l]$, δ_i evaluates to y_i, for all $1 \leq i \leq l$. The state sequence $\pi = x^0 x^1 \ldots$ is a *run* of M if $(x^i, x^{i+1}) \in G$, for all $i \geq 0$. The run π is *initialized* if $x^0 \in I$.

Definition 4.1 An instance of the *state reachability problem* consists of an FSM $M = \langle S, I, \mathcal{X}, \Sigma_I, T \rangle$ and a subset of states $F \subseteq S$. The answer to the state reachability problem is YES if there exists an initialized run $x^0 x^1 \ldots x^r$ in the state transition graph of M such that $x^r \in F$, and NO otherwise. ∎

As defined, the next state behavior of an FSM must be deterministic. However, sometimes FSMs are specified with nondeterminism, meaning that for a given input and present state, there may be more than one next state. In this case, a relation, rather than a function, is needed to specify the next state behavior. Since we are only interested in the state transition graph G derived from the relation T, nondeterminism does not pose a problem; we stick with determinism for clarity.

Finally, we are interested in analyzing a system of interacting machines, but our definition of the problem is with respect to a single FSM. This limitation is easily overcome by realizing that any system of FSMs can be thought of as a single FSM by amassing the next state functions of all the state variables of the system into a single vector of next state functions.

4.3 Related work

The problem of traversing the state graph of an FSM, whether to compute the set of reachable states or to determine if a specific subset of states is reachable, has been the object of intensive research over the last decade. A breakthrough occurred in 1989 when Coudert, Berthet and Madre proposed using BDDs to perform symbolic breadth-first traversal of state graphs [42]. With this approach, the number of states in the graph is no longer the principal limitation (as in depth-first traversal); instead the "complexity" of the Boolean functions defining the underlying circuit govern the efficiency.

Since symbolic traversal was proposed, many researchers have suggested various heuristics in a quest to traverse ever larger and complex FSMs. These heuristics have been shown to be effective in some cases, and not so in others. Here we review some of the previous work, and how it relates to our research.

4.3.1 Image computation

Image computation is the central task in symbolic traversal. Given a set of states A, we want to determine the successors of A in the graph G of an FSM. The image can be computed as

$$image(y) = \exists x(G(x,y) \cdot A(x)).$$

That is, a state y is a successor of A if there exists a state x in A such that there is an edge from x to y in G. Substituting for G from above,

$$image(y) = \exists x \exists u(\prod_{i=1}^{l} T_i(x,u,y_i) \cdot A(x)).$$

The naive approach of first taking the product of the T_i's and A, and then quantifying x and u, is ill-conceived because often the intermediate product is large even though the final result is not so large. In general, the full product must be computed because existential quantification does not distribute over Boolean conjunction. However, a special case can be exploited where it does distribute. Namely, the equation

$$\exists x(f(x,y) \cdot g(y))$$

can be rewritten as

$$\exists x(f(x,y)) \cdot g(y).$$

Several researchers have used this fact to quantify some variables before the entire product is formed, in an attempt to avoid the intermediate blowup in the overall computation [43, 44, 45, 46, 7].

Another technique for simplifying image computation is to use certain sets of states as don't cares to simplify the BDDs of the set A of states and the individual transition relations T_i. In particular, suppose that the set R of states has already been reached, and during the previous image computation, the set B was reached for the first time. For the next image computation step, there is no harm in re-exploring states in R that are not in B. Thus, any set C such that $B \subseteq C \subseteq R$ is suitable for the next image computation. In addition, we can arbitrarily choose the behavior of T_i on any state x not in C, since such states are disregarded when the eventual product with C is formed. The operators *constrain* and *restrict* implement the simplification of BDDs using don't care sets [6, 7, 8].

Cabodi *et al.* [47] introduce the *existsCofactor*, which is similar to the constrain operator, but allows existential quantification to distribute over conjunction. In particular, they are able to rewrite

$$\exists x(f(x,y) \cdot g(x,y))$$

as

$$\exists x f(x,y) \cdot \exists g'(x,y)),$$

where $g'(x,y)$ is the *existsCofactor* of g with respect to f.

All three of the techniques discussed in this subsection are orthogonal to our approach, and in fact can be used in combination with our approach

4.3.2 Exact state reachability

Balarin introduced an algorithm to test for language emptiness of automata over infinite strings [48]. For ease of presentation, we describe his algorithm for the simpler case of automata over finite strings. An automaton over finite strings has a designated set F of accepting, or final, states. The language of such an automaton is not empty if and only if there exists a path from I (the initial states) to F; thus, state reachability provides the answer to language emptiness.

To solve this problem, Balarin makes a series of successively finer over-approximations to the state graph G of the automaton, in an attempt to determine if I can reach F. Each over-approximation V is analyzed to determine if there is a path from I to F. There are two cases to consider.

1. There is a path from I to F in V: If this path exists in G (G is analyzed for this single path), then the language is not empty, and the algorithm terminates. Otherwise, a new over-approximation is constructed that eliminates this path in V, and the procedure is repeated.

2. There is not a path from I to F in V: Then the language is empty, and the algorithm terminates.

Balarin's algorithm partly served as inspiration for our approach. However, whereas Balarin analyzes a single path from I to F in V at a given iteration, we analyze all of the paths from I to F to guide the search for a real path in the under-approximation U.

The work of Cabodi *et al.* [49] is similar in spirit to ours. They first compute an over-approximation of the states reachable from I, and then use this information to constrain an exact backward search from F. If I is reached in the backward search from F, the the state reachability problem is answered in the affirmative.

The work of Courcoubetis *et al.* [50] addresses the problem of not being able to build the transition relation for an FSM M. Instead of employing BDDs and performing BFS on the state graph, they use DFS, building up the graph one transition at a time. Since this method is explicit, they are limited in time to exploring roughly 10^8 states. This method is referred to as "on-the-fly", because states are checked to see if they belong to F *while* the graph is being built.

4.3.3 Approximate state reachability

The set of *reachable states* is the set of states R that can be reached from the initial states I. Obviously, if R is known, then the set I can reach F if and only if R and F have a non-empty intersection. Even if R is not known, sometimes an approximation to R can be used to answer the state reachability problem. If an under-approximation R^- and F intersect, then clearly R and F intersect, and the answer to state reachability is "YES". On the other hand, if an over-approximation R^+ and F do not intersect, then R and F do not intersect, and the answer is "NO". In the cases where R^+ or R^- cannot be used to answer the state reachability problem, they could be used as a starting point to focus the search for a path from I to F.

Cho *et al.* [51] have devised various techniques for over-approximating R. The first step of these techniques is to partition the flip-flops of the FSM to yield a set of k interacting sub-FSMs. This partitioning is done so that flip-flops with strong interaction tend to be placed in the same sub-FSM. Next, the set R_i of reachable states of sub-FSM$_i$ is computed for each i. This computation does not consider the full, dynamic interaction of sub-FSM$_i$ with the other sub-FSMs, but instead considers some partial constraints on the values of the inputs of sub-FSM$_i$ driven by the other sub-FSMs. This yields an over-approximation of the state reachable in sub-FSM$_i$. Finally, an over-approximation R^+ of the entire reached set is given by the Cartesian product $R^+ = R_1 \times \ldots \times R_k$. They use the complement of R^+ as an under-approximation of the unreachable states to perform logic minimization.

Ravi and Somenzi [4] propose a technique to under-approximate R. Their al-

gorithm proceeds with the usual symbolic BFS, but when the set of states A to explore becomes too large (in terms of BDD size), they continue the search from only a subset of A. This subset is heuristically chosen to have a small BDD while retaining as many states as possible from A. As mentioned previously, this problem of approximating a set using a small BDD is the same problem we have formulated. As such, we could employ the heuristics proposed by Ravi and Somenzi in our work.

As mentioned earlier, Courcoubetis *et al.* [50] perform state graph traversal using depth-first traversal, and not using BDDs. To represent the set of states visited thus far in a traversal, they use a data structure whose size is directly proportional to the size of the set of states. The problem is that this set may become too large to represent. To combat this problem, they use a hash table without collision chains, which hashes a state to a single bit indicating if the state has already been visited. Since collisions may occur in this hash table, it may be incorrectly deduced that a state has already been visited, when in fact it has not been. This may in turn lead to visiting only a subset of the total set of reachable states, thus yielding an under-approximation. They use probabilistic analysis to quantify the probability of collisions for a randomly chosen hash function.

4.4 Algorithm to decide state reachability

Figure 4.1 gives an outline of our algorithm to solve the state reachability problem for the FSM M and final state set F. The algorithm also takes $0 \leq \alpha \leq 1$ as input, which controls the degree of approximation used during the algorithm. Here we discuss the top-level control of the algorithm and illustrate the algorithm in detail with an example. In Section 4.6, we discuss each of the subroutines in detail.

We assume that the graph $G(x, y)$ of M is too big to build and manipulate. Instead, we construct a series of approximations to $G(x, y)$ that are small enough to manipulate efficiently. We are willing to trade off execution time in favor of memory savings, in an attempt to handle very large FSMs.

The initial step is to compute an over-approximation $V(x, y) \supseteq G(x, y)$ (line 1), and an under-approximation $U(x, y) \subseteq G(x, y)$ (line 2). G is a set of directed edges of a graph, so V is a superset and U is a subset of this set of edges. The goal is to choose $V(x, y)$ and $U(x, y)$ so that they are close to $G(x, y)$, but have much smaller representations.

The algorithm then iterates, adding edges to $U(x, y)$ and removing edges from

```
   stateReachability(M, F, α)

 1 V := InitialOverApprox(M, α)
 2 U := InitialUnderApprox(M, α)

 3 I := initial states I of M
 4 F := input parameter F

 5 while (TRUE)
 6   I := I + ForwardReachability(U, I)   /* states reachable from I in U */
 7   F := F + BackwardReachability(U, F)   /* states which can reach F in U */

 8   if (I intersects F)
 9     return "yes, I can reach F"

10   V := RestrictToFinal(V, I, F)   /* restr. V to edges on paths from I to F */

11   if (I cannot reach F in V)
12     return "no, I cannot reach F"

13   toF := V(x,y) · ¬F(x) · F(y)   /* transitions to F in V */
14   fromI := V(x,y) · I(x) · ¬I(y)   /* transitions from I in V */

15   falseToF(x,y) := ApproxFalseEdges(M, toF, V, α)   /* ⊆ false edges in toF */
16   trueToF(x,y) := ApproxTrueEdges(M, toF, V, α)   /* ⊆ true edges in toF */
17   falseFromI(x,y) := ApproxFalseEdges(M, fromI, V, α)   /* ⊆ false edg. in fromI */
18   trueFromI(x,y) := ApproxTrueEdges(M, fromI, V, α)   /* ⊆ true edg. in fromI */

19   if (falseToF AND trueToF AND falseFromI AND trueFromI are empty)
20     trueToF(x,y) := ExactTrueEdges(M, toF)
21     trueFromI(x,y) := ExactTrueEdges(M, fromI)
22     if (trueToF OR trueFromI is empty)
23       return "no, I cannot reach F"
24     else
25       falseToF(x,y) := toF(x,y) - trueToF(x,y)
26       falseFromI(x,y) := fromI(x,y) - trueFromI(x,y)

27   V := V - (falseToF + falseFromI)   /* remove false edges from V */
28   U := U + (trueToF + trueToI)   /* add true edges to U */
```

Figure 4.1: Algorithm to decide state reachability.

$V(x,y)$, until it is determined whether or not there exists a path from I to F in G. The search is conducted working forward from I and backward from F. In particular, each iteration starts by performing, in U, forward reachability from I and backward reachability from F (lines 6 and 7). Reachability is carried to a fixed point. Any states reached from I are added to I, and any states that can reach F are added to F. Since all edges in U are present in G, all states in I and F can also be reached in G. If at any time I and F intersect, then we know that a path exists from the original I to the original F in G (lines 8 and 9).

The next step is to restrict V to those edges that lie on some path from I to F in V (line 10). If there does indeed exist a path from I to F in G, then it must lie in the restricted V. Hence, if it is discovered that there is no path from I to F in V, then we can immediately conclude that no such path exists in G (lines 11 and 12). This restriction is done on each iteration, because as we will see, we eliminate some edges from V on each iteration (line 27). Remember that we assume that the representation for V is small enough so that we can do reachability on V efficiently.

Since we carried reachability in U to a fixed point (lines 6 and 7), by definition, there are no edges in U leaving I or entering F. Hence, to continue the search in U, we need to find edges of G leaving I and edges entering F. Where do we look for such edges? Naturally, we look for them in V. In particular, V focuses our search for a path from I to F, since if such a path exists in G, it must exist in V. Furthermore, V contains *only* those edges lying on a path from I to F in V (because of line 10). This is a key point. Thus, in lines 13 and 14, we restrict V to those edges entering F (*toF*) and those exiting I (*fromI*). Our intention is to determine which edges in these sets are "true" (exist in G) and which are "false" (do not exist in G). Just answering the question for one edge is already NP-complete (reduction from SAT). Hence, we try to approximate the true and false sets (lines 15-18). If all of the approximations are empty, then we must do *exact* analysis (lines 20 and 21) on the true edges in order to draw any conclusions. If exact analysis does not find any true edges, then we can conclude (line 23) that no path exists from I to F in G.

If at least one of *falseToF*, *trueToF*, *falseFromI*, and *trueFromI* is non-empty, then we remove false edges from V, and add true edges to U (lines 27 and 28). Removing false edges from V is important because it may further narrow the search. At this point, we repeat the entire loop.

There are three major subroutines in this procedure:

1. initial over-approximation of G (line 1),

2. initial under-approximation of G (line 2), and

3. approximation of edges from I and to F (lines 15-18).

Each of these subroutines involves approximating a set of edges represented by a BDD. For this task, we make extensive use of the BDD approximation operators. Section 4.5 addresses the BDD approximation problem.

4.4.1 Example

We illustrate the algorithm with a detailed example. Figure 4.2 shows the graph $G(x, y)$ of a 21-state FSM. The states are labeled for reference. The set of initial states is $I = \{3\}$, and the set of final states is $F = \{13\}$. To emphasize, we do not have direct access to this graph in the algorithm; it is shown here to clarify the operation of the algorithm.

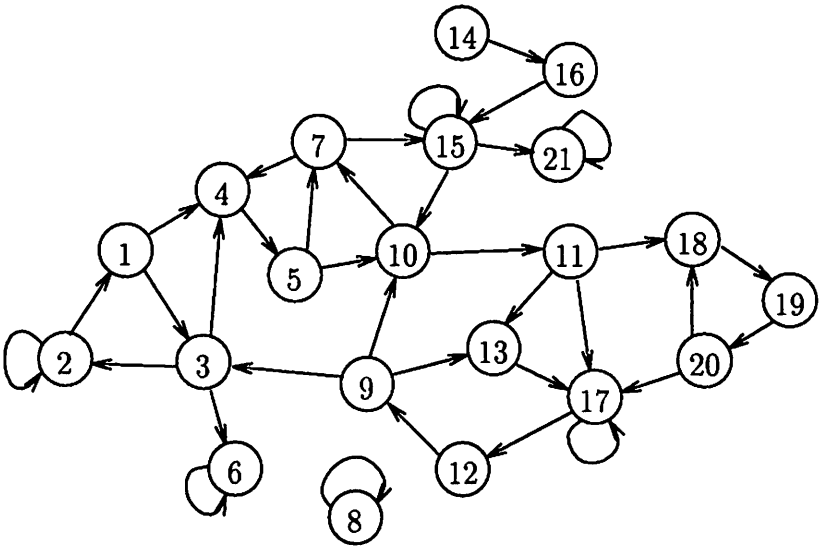


Figure 4.2: Graph G of a 21-state FSM. The initial state is 3 and the final state is 13.

The initial over-approximation V_1 and under-approximation U_1 are shown in Figures 4.3 and 4.4, respectively. Throughout the example, the current sets of states I and F are each indicated by a dotted region. In addition, the false edges of the over-approximations (although not known a priori by the algorithm) are indicated by dots.

Iteration 1 Performing reachability in U_1 (lines 6, 7), I is enlarged to $\{2,3\}$ (because of the edge 3→2) and F is enlarged to $\{9,13\}$ (because of 9→13). Since $I \cap F = \emptyset$, we proceed to restrict V_1 (line 10) to edges on paths from I to F, to yield V_1' in Figure 4.5.[2] For example, edges 3→6 and 16→15 are removed. With careful analysis, even edges 9→3 and 1→3 can be removed, since their removal does not affect the reachability question. In addition, we remove the self-loops in V_1. Since I cannot reach F in V_1', the condition at line 11 if false.

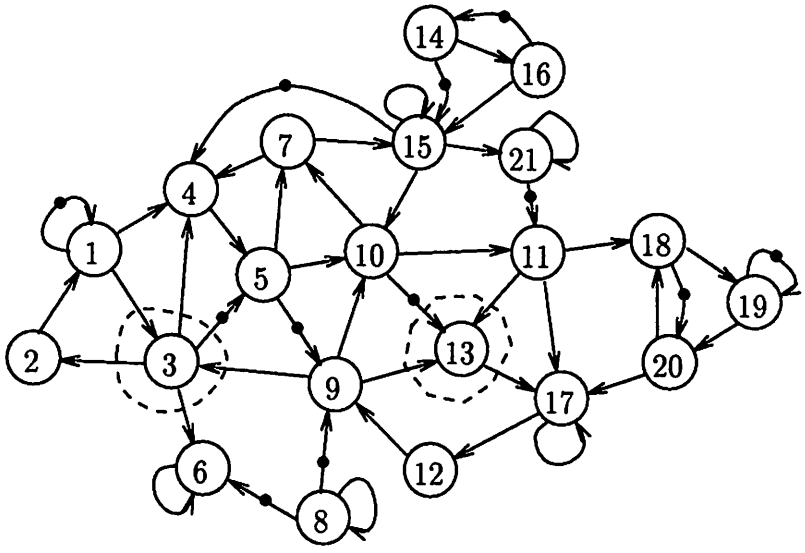


Figure 4.3: Initial over-approximation V_1. The sets I and F are indicated by the dotted region. False edges are indicated by a dot.

As a side note, as the algorithm is presented, whenever we can remove edges from V_1, we do so. In reality, removing such edges may make the BDD for V needlessly larger. Hence, these edges should be treated as don't cares and used to minimize the BDD size of V. In addition, edges between two states in I, or between two states in F, can be treated as don't cares in both V and U.

Continuing with the algorithm, line 13 identifies those edges of V_1' entering F,

$$toF = \{5\rightarrow 9, 10\rightarrow 13, 11\rightarrow 13, 12\rightarrow 9\}.$$

and line 14 identifies the edges of V_1' leaving I,

$$fromI = \{2\rightarrow 1, 3\rightarrow 4, 3\rightarrow 5\}.$$

[2]Now the new I and F are shown. Edges contained within I or within F are not drawn.

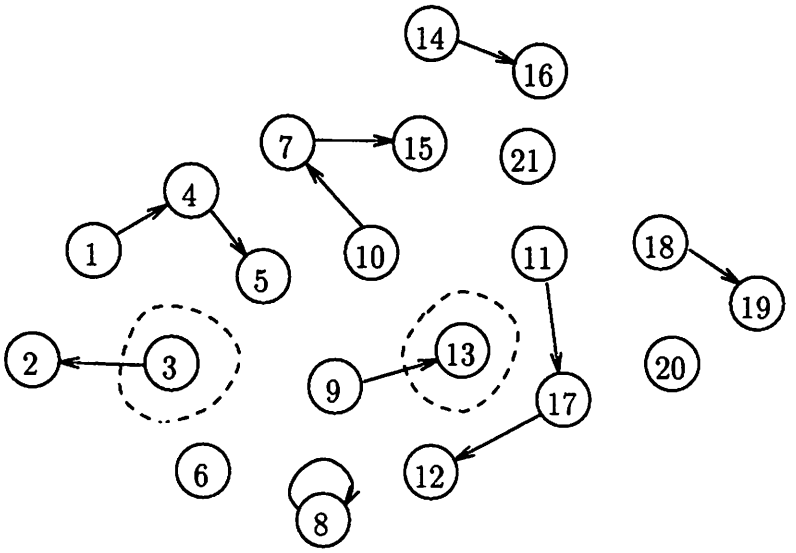


Figure 4.4: Initial under-approximation U_1.

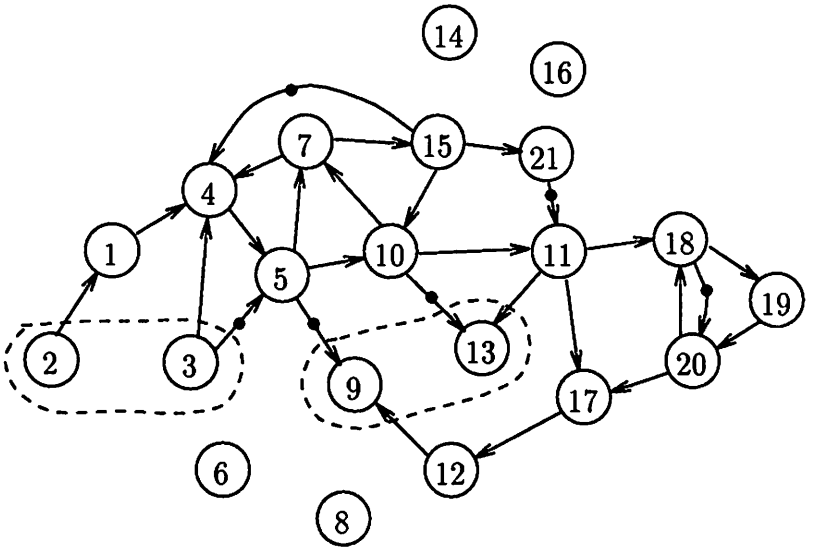


Figure 4.5: Over-approximation V_1', formed by restricting V_1 to edges on paths from I to F. Edges contained within I or within F are not drawn.

Since these edges belong to an over-approximation, we do not know a priori which edges are true and which are false. Suppose lines 15 to 18 make the following approximations:

$$\begin{aligned} \mathit{falseToF} &= \emptyset, \\ \mathit{trueToF} &= \{12 \rightarrow 9\}, \\ \mathit{falseFromI} &= \{3 \rightarrow 4\}, \\ \mathit{trueFromI} &= \{2 \rightarrow 1\}. \end{aligned}$$

Since not all of these are empty, the algorithm proceeds to lines 27 and 28, where edge $3 \rightarrow 4$ is removed from V_1' to yield V_2 (Figure 4.6), and edges $12 \rightarrow 9$ and $2 \rightarrow 1$ are added to U_1 to yield U_2 (Figure 4.7).

Iteration 2 Reachability on U_2 expands I to $\{1, 2, 3, 4, 5\}$, and expands F to $\{9, 11, 12, 13, 17\}$. Notice that pre-existing edges in the under-approximation (e.g., $1 \rightarrow 4$, $4 \rightarrow 5$) allow the reachability computation to progress beyond those edges just added to the under-approximation.

I and F still do not intersect, so we restrict V_2 (for example, by removing all edges to and from states 18, 19 and 20), to yield V_2' (Figure 4.8). I can still reach F in V_2' (line 11), so we compute

$$\begin{aligned} \mathit{toF} &= \{5 \rightarrow 9, 10 \rightarrow 13, 10 \rightarrow 11, 21 \rightarrow 11\}, \text{ and} \\ \mathit{fromI} &= \{5 \rightarrow 7, 5 \rightarrow 9, 5 \rightarrow 10\}. \end{aligned}$$

Suppose lines 15 to 18 make the following approximations:

$$\begin{aligned} \mathit{falseToF} &= \{10 \rightarrow 13, 21 \rightarrow 11\}, \\ \mathit{trueToF} &= \emptyset, \\ \mathit{falseFromI} &= \{5 \rightarrow 9\}, \\ \mathit{trueFromI} &= \{5 \rightarrow 7\}. \end{aligned}$$

Then the false edges are removed from V_2' to yield V_3 (Figure 4.9), and the true edge is added to U_2 to yield U_3 (Figure 4.10).

Iteration 3 Reachability on U_3 adds states 7 and 15 to I. I and F do not intersect. Restricting V_3 removes edges $10 \rightarrow 7$ and $15 \rightarrow 21$, to yield V_3' (Figure 4.11). Attention is

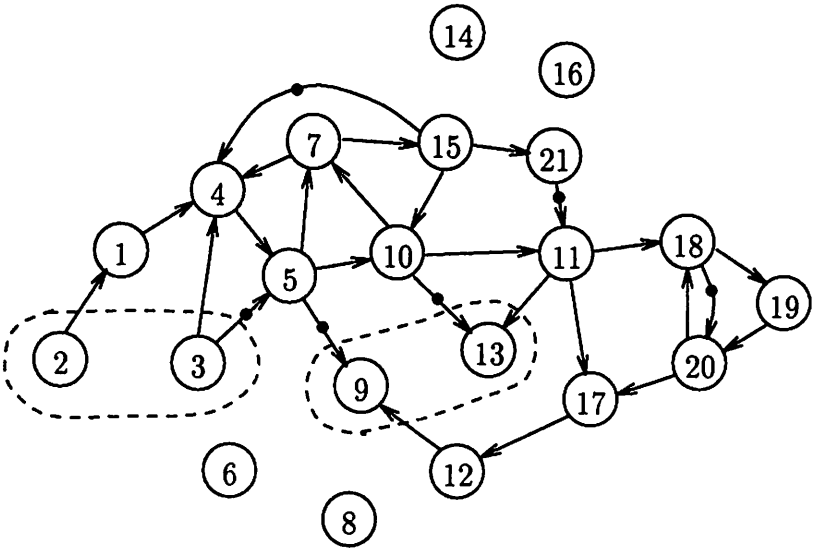


Figure 4.6: Over-approximation V_2, formed by removing edge 3→4 from V_1'.

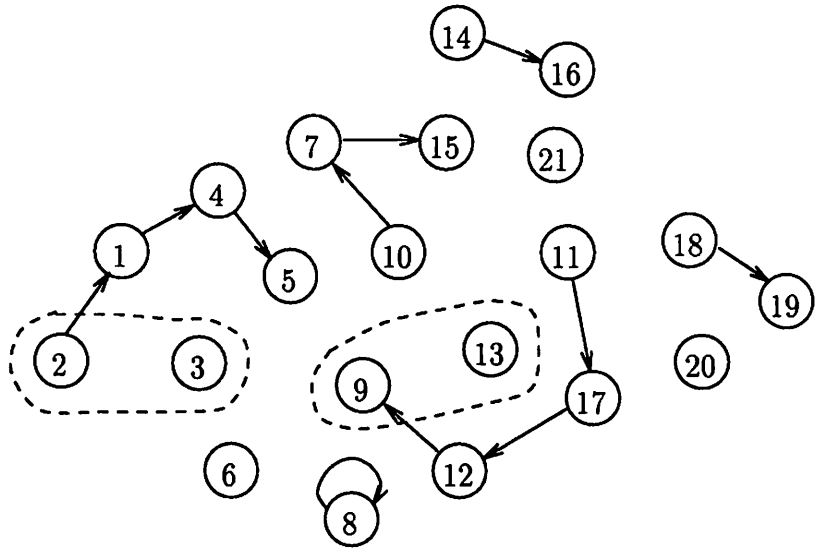


Figure 4.7: Under-approximation U_2, formed by adding edges 12→9 and 2→1 to U_1.

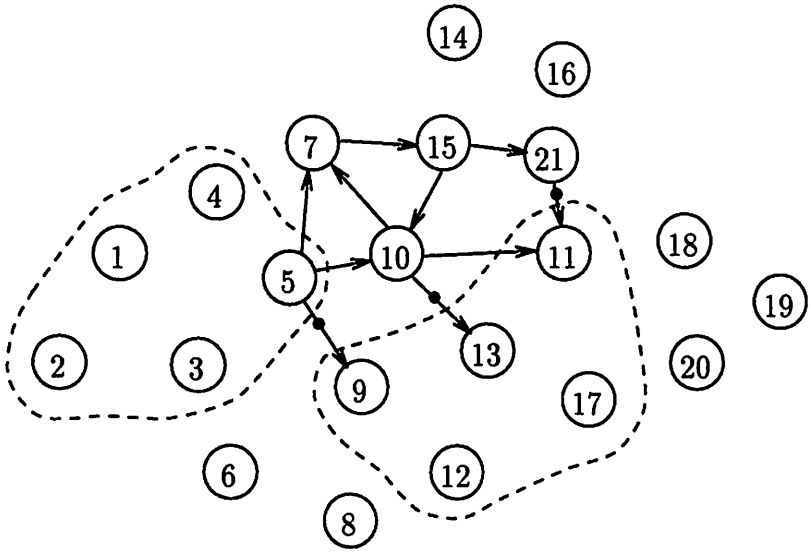


Figure 4.8: Over-approximation V_2', formed by restricting V_2 to edges on paths from I to F.

focused on

$$\begin{aligned} toF &= \{10 \rightarrow 11\}, \text{ and} \\ fromI &= \{5 \rightarrow 10, 15 \rightarrow 10\}. \end{aligned}$$

Suppose lines 15 to 18 make the following approximations:

$$\begin{aligned} falseToF &= \emptyset, \\ trueToF &= \{10 \rightarrow 11\}, \\ falseFromI &= \emptyset, \\ trueFromI &= \{15 \rightarrow 10\}. \end{aligned}$$

These two true edges are added to U_3 to yield U_4 (Figure 4.12).

Iteration 4 Reachability in U_4 adds state 10 to I, and also to F. At this point, I and F intersect, and the algorithm returns "yes, I can reach F."

4.5 Approximating Boolean functions

Our algorithm to decide state reachability efficiently is predicated upon being able to find close approximations to sets of edges (e.g., the graph $G(x, y)$, and the sets $toF(x, y)$

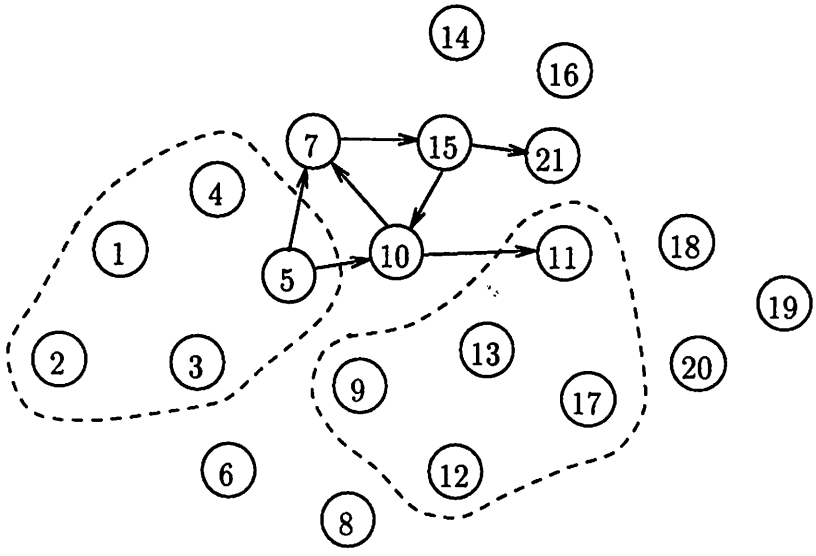


Figure 4.9: Over-approximation V_3, formed by removing edges 10→13, 21→11 and 5→9 from V_2'.

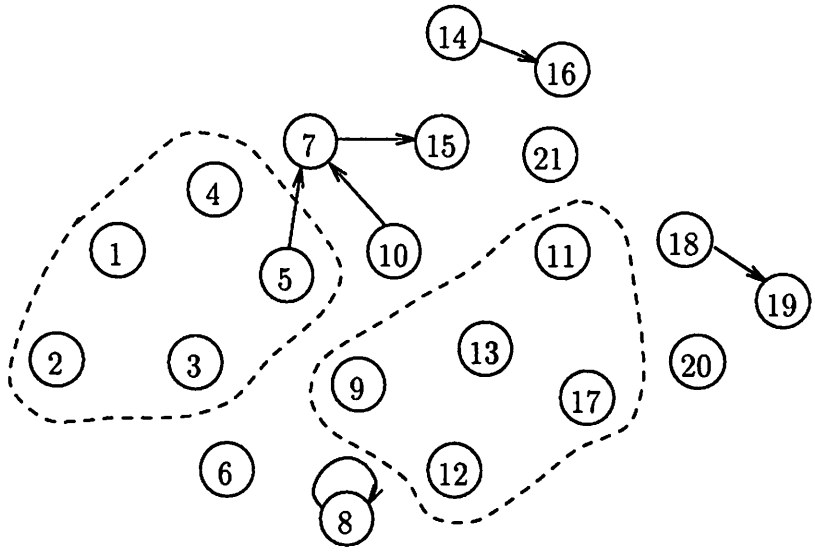


Figure 4.10: Under-approximation U_3, formed by adding edge 5→7 to U_2.

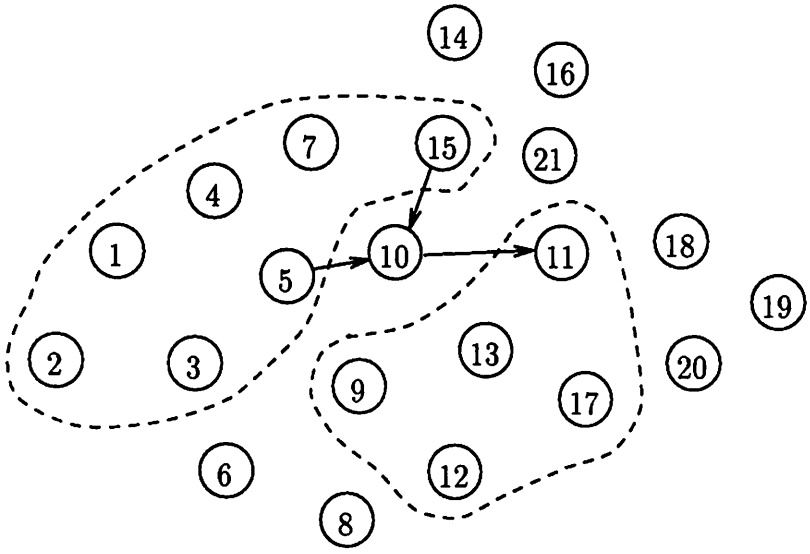


Figure 4.11: Over-approximation V_3', formed by restricting V_3 to edges on paths from I to F.

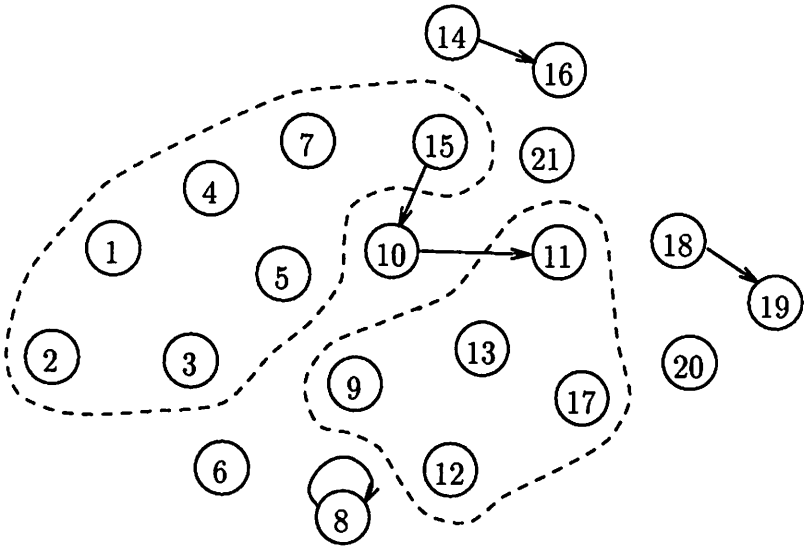


Figure 4.12: Under-approximation U_4, formed by adding edges 10→11 and 15→10 to U_3.

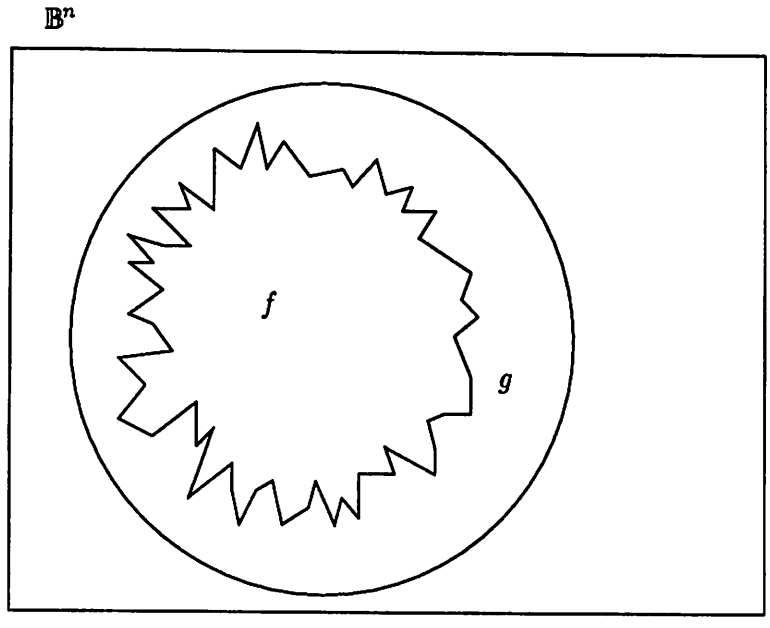


Figure 4.13: The BDD over-approximation problem.

and $fromI(x, y)$), which have small BDDs. In this section, we define a general problem whose solution can be used to approximate sets of edges. We discuss the related work of Ravi and Somenzi, and then offer our own heuristic to solve the problem.

4.5.1 Statement of the problem

Given a Boolean function f, we say that the Boolean function g if an *over-approximation* of f is $g \supseteq f$. This definition can be extended to relations by considering the characteristic functions of relations. We want to find an over-approximation g of f such that g just "barely" contains f, and yet the BDD for g is much smaller that the BDD for f (under a fixed variable ordering). This problem is illustrated in Figure 4.13, where the function f is a "complicated" function with a large BDD, and g is a "simple" function with a small BDD, derived from f by adding some minterms to the onset.

The demands of having a close approximation and yet having a small BDD are sometimes conflicting. There are two extreme approximations we could consider. The first is the function f itself; this approximation is exact, however, by assumption, this function has an unwieldy BDD. The second approximation is the tautology; this approximation has a BDD of size 1, however, it is unlikely to be useful since it does not contain any information.

These conflicting demands lead us to the following optimization problem.

Definition 4.2 The *BDD over-approximation problem* is, given the BDD for a function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ and $0 \leq \alpha \leq 1$, find $g \supseteq f$ such that the *cost* of g is minimized, where

$$cost(g) = \alpha(\log_2^2 |\text{onset}(g)|) + (1 - \alpha)|\text{BDD}(g)|$$

and $|\text{onset}(g)|$ is the size of the onset of g, and $|\text{BDD}(g)|$ is the size of the BDD for g. ∎

Several remarks regarding this problem are in order.

1. The parameter α appearing in the cost function allows us to control the relative weight between finding a close approximation and finding an approximation with small BDD size. We see that when $\alpha = 1$, the minimum cost solution is f itself, and when $\alpha = 0$, the minimum cost solution is the tautology.

2. Since for an over-approximation g of f, $|\text{onset}(g)| \geq |\text{onset}(f)|$, if $cost(g) < cost(f)$, then this implies that $|\text{BDD}(g)| < |\text{BDD}(f)|$.

3. The logarithm of the onset size is used to balance the two terms being summed. Even though both $|\text{onset}(g)|$ and $|\text{BDD}(g)|$ can be exponential in n, functions we can handle typically have exponential size onsets but polynomial size BDDs. Therefore, so that the onset size term does not dominate the BDD size term, we take the logarithm squared of the onset size term.

4.5.1.1 Complexity of the BDD over-approximation problem

The decision problem corresponding to the state reachability problem is in NP.
Instance: A function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ represented by a BDD, $0 \leq \alpha \leq 1$, and $K < |\text{BDD}(f)|$.
Question: Does there exist $g : \mathbb{B}^n \rightarrow \mathbb{B}$ such that $g \supseteq f$ and $cost(g) \leq K$?

Proposition 4.3 *The above problem is in NP.*

Proof We must verify in time polynomial in $|\text{BDD}(f)|$ whether or not a guess g is a solution to the problem. If $|\text{BDD}(g)| > |\text{BDD}(f)|$, then we can immediately dismiss g as a potential solution, by the second remark above. Otherwise, we traverse BDD(g) to determine $|\text{onset}(g)|$; this can be done in time $O(|\text{BDD}(g)|)$. Then we compute $cost(g)$, and verify whether or not $cost(g) \leq K$. If so, we verify that $g \supseteq f$; this can be done by checking that $f \cdot \overline{g} = 0$, which can be done in time $O(|\text{BDD}(f)| \cdot |\text{BDD}(g)|)$. ∎

4.5.1.2 Minterms versus BDD size

The drawing in Figure 4.13 is meant to suggest that just by adding a "few" minterms to the onset of f, the BDD size can be drastically reduced. Unfortunately, the truth is not so ideal. We now show that adding one minterm to the onset of f cannot reduce the BDD size of f by more than n, where n is the number of variables. Thus, if $|\text{BDD}(f)|$ is exponential in n, then to get an exponential reduction in the BDD size, we must add an exponential number of minterms, which can no longer be construed as a "close" approximation.

Let $x_1, x_2, \ldots, x_n$ span the space $\mathbb{B}^n$. Without loss of generality, assume the BDD variable ordering is $x_1 < x_2 < \ldots < x_n$, with x_1 being the top variable.

Lemma 4.4 *Let $f : \mathbb{B}^n \rightarrow \mathbb{B}$ and let $m \in \mathbb{B}^n$ be a minterm in the offset of f. Then $|BDD(f + m)| \leq |BDD(f)| + n$. That is, adding a minterm to the onset of a function cannot increase the size of its BDD by more than n.*

Proof We argue that the number of nodes at level i cannot increase by more than 1. Since there are at most n levels, the total size cannot increase by more than n.

Let $b_j \in \mathbb{B}$ denote an assignment to x_j, and let m be the minterm $b'_1, \ldots, b'_n$. Consider the cofactors of f on all combinations of $b_1, \ldots, b_{i-1}$. Partitions these cofactors into equivalence classes based on equality. The number of classes whose representative depends on x_i gives the number of nodes at level i in the BDD for f.

Now consider the cofactors of $f + m$ on all combinations of $b_1, \ldots, b_{i-1}$. First, we note that cofactoring distributes over disjunction, so

$$(f + m)_{b_1,\ldots,b_{i-1}} = f_{b_1,\ldots,b_{i-1}} + m_{b_1,\ldots,b_{i-1}}.$$

The cofactor of m by $b_1, \ldots, b_{i-1}$ is 0 for every combination of $b_1, \ldots, b_{i-1}$, except for $b'_1, \ldots, b'_{i-1}$. Thus, all but one of the cofactors of $f + m$ are the same as the corresponding cofactors of f. Hence, each cofactor remains in the same equivalence class, with the exception of the cofactor by $b'_1, \ldots, b'_{i-1}$. In the case that this cofactor forms its own class, and is dependent on x_i, the number of nodes at level i will increase by one. In all other cases (the cofactor joins another class, or forms its own class but is independent of x_i), the number of nodes at level i does not increase. ∎

Lemma 4.5 *Let $f : \mathbb{B}^n \rightarrow \mathbb{B}$ and let $m \in \mathbb{B}^n$ be a minterm in the onset of f. Then $|BDD(f \cdot \overline{m})| \leq |BDD(f)| + n$. That is, removing a minterm from the onset of a function cannot increase the size of its BDD by more than n.*

Proof The proof is similar to Lemma 4.4. In this case, we have

$$(f \cdot \overline{m})_{b_1,\ldots,b_{i-1}} = f_{b_1,\ldots,b_{i-1}} \cdot \overline{m}_{b_1,\ldots,b_{i-1}}.$$

The only combination of $b_1, \ldots, b_{i-1}$ where $\overline{m}_{b_1,\ldots,b_{i-1}} \neq 1$ is $b'_1, \ldots, b'_{i-1}$. Thus, all but one of the cofactors of $f \cdot \overline{m}$ are the same as the corresponding cofactors of f. The rest of the proof is the same. ∎

Theorem 4.6 *Let $f : \mathbb{B}^n \rightarrow \mathbb{B}$ and let $m \in \mathbb{B}^n$ be a minterm. Changing the value of f on m cannot change the size of the BDD for f by more than n.*

Proof

Case 1: m is in the offset of f. By Lemma 4.4, $|\text{BDD}(f+m)| \leq |\text{BDD}(f)| + n$. We must show that $|\text{BDD}(f+m)| \geq |\text{BDD}(f)| - n$. For sake of contradiction, suppose $|\text{BDD}(f+m)| < |\text{BDD}(f)| - n$. Let $g = f + m$. By Lemma 4.5, $|\text{BDD}(g \cdot \overline{m})| \leq |\text{BDD}(g)| + n$. Since

$$g \cdot \overline{m} = (f + m)\overline{m} = f \cdot \overline{m} = f,$$

then substituting for g gives $|\text{BDD}(f)| \leq |\text{BDD}(f+m)| + n$. By hypothesis, $|\text{BDD}(f+m)| < |\text{BDD}(f)| - n$, which implies $|\text{BDD}(f)| < |\text{BDD}(f)| - n + n = |\text{BDD}(f)|$, an obvious contradiction. Thus, adding a minterm to f cannot change the BDD size by more than n.

Case 2: m is in the onset of f. By Lemma 4.5, $|\text{BDD}(f \cdot \overline{m})| \leq |\text{BDD}(f)| + n$. To show that $|\text{BDD}(f \cdot \overline{m})| \geq |\text{BDD}(f)| - n$, we proceed exactly as in Case 1, using Lemma 4.4 this time. Thus, removing a minterm from f cannot change the BDD size by more than n. ∎

Thus, we see that the effect on BDD size of adding minterms to a function is somewhat gradual. Adding k minterms can reduce the BDD size by at most kn. Of course, adding minterms can also *increase* the BDD size, so choosing which minterms to add requires judiciousness. As a side note, the above lemmas and theorem can be easily generalized to adding and removing cubes, rather than just minterms.

4.5.1.3 Under-approximations

So far we have discussed only the problem of finding good over-approximations. However, we are also interested in finding good under-approximations. The formal statement of the BDD under-approximation problem follows.

Definition 4.7 The *BDD under-approximation problem* is, given $f : \mathbb{B}^n \rightarrow \mathbb{B}$ and $0 \leq \alpha \leq 1$, find $g \subseteq f$ such that the *cost* of g is minimized, where

$$cost(g) = \alpha(\log_2^2(|\text{offset}(g)|)) + (1 - \alpha)|\text{BDD}(g)|.$$

■

Thus, to minimize the cost, we want to minimize the size of the BDD and the size of the offset, subject to the constraint that $g \subseteq f$. Note that when $\alpha = 1$, the minimum cost solution is f itself, and when $\alpha = 0$, the minimum cost solution is the zero function.

4.5.2 The subsetting problem of Ravi and Somenzi

Ravi and Somenzi independently and concurrently formulated a problem, termed the *subsetting problem*, which is nearly identical to our BDD under-approximation problem [4]. They employ subsetting to compute an under-approximation of the set of reachable states of an FSM, as explained in Section 4.3. Here, we explain the subsetting problem, and the heuristics they propose for solving this problem.

Definition 4.8 The *subsetting problem* is, given a BDD for $f : \mathbb{B}^n \rightarrow \mathbb{B}$ and a threshold $K < |\text{BDD}(f)|$, find a function g such that $g \subseteq f$, $|\text{BDD}(g)| \leq K$, and the number of minterms in the onset of g is maximum. ■

This problem is nearly identical to ours, the only difference being that Ravi and Somenzi use the threshold K to control the degree of approximation, whereas we use the parameter α.

The first heuristic they propose is termed *heavy branch subsetting*. This method starts at the root of the BDD for f and follows a single path through the BDD, setting to the constant ZERO the side branches along this path. For a given node on this path, it always sets to ZERO that child "holding" the lesser number of minterms in the onset of f (the "light" child), and keeping the other (the "heavy" child). The procedure terminates when

enough nodes have been eliminated so that the total BDD size falls below the threshold K. The procedure keeps track of how many nodes are being eliminated by computing, in a preprocessing step, the number of nodes "held" by each light child, exclusive of its corresponding child (called the *differential_node_count*). Using this technique, the total runtime is linear in |BDD(f)|. The result of this procedure is a BDD with a string of nodes at the top, each with one child pointing to ZERO.

The second heuristic is called *short path subsetting*. The idea here is to keep just those short paths from the root to the constant ONE, because they hold a large number of minterms but cost little in terms of the number of BDD nodes. The procedure first labels each node with the sum of its shortest distance from the root, and its shortest distance to the constant ONE; this is called the *path_length*. Then, based on the threshold K, it determines a maximum value for *path_length* such that removing all nodes with a *path_length* greater than the maximum will yield a BDD of size less that K. The resulting BDD may have many disjoint paths, and consequently little sharing of BDD nodes.

Experiments were conducted to compute under-approximations of the set of reachable states for several large FSMs. These experiments validated the utility of subsetting. As a side note, neither heuristic was shown to be superior to the other.

4.5.3 Heuristic for the BDD under-approximation problem

For a function f with k minterms in its onset, there are 2^k functions $g \subseteq f$. Since we want to find an approximation g with lower cost than f, g must have fewer BDD nodes. We try to find such a g by replacing some subgraphs of f by the constant ZERO; this is also the general approach of Ravi and Somenzi. This is guaranteed to reduce the number of nodes, while meeting the condition that $g \subseteq f$. However, depending on the value of α, replacing a subgraph by ZERO may actually increase the cost. The challenge is to determine which set of subgraphs to replace by ZERO in order to maximize the cost reduction.

Although we have not been able to determine a lower bound on the complexity of the BDD under-approximation problem, it seems likely that solving the problem exactly would be prohibitive. In fact, since we want to repeatedly apply the approximation operator on BDDs of tens of thousands of nodes, we require an algorithm that is linear, or nearly so. Because of this, we take a very greedy approach.

The basic idea is to visit the nodes of the BDD for f on a level-by-level basis, from

top to bottom. Within a level, the nodes are visited in an arbitrary order. When a node v is visited, we compute

- *numOnset*(v), which is the number of minterms in the onset of f that would be removed if all edges pointing to v were redirected to ZERO, and
- *nodeSavings*(v), which is the number of nodes in the subgraph rooted at v that would be saved if v were replaced by ZERO. Note that some nodes in the subgraph of v are shared by other parts of the BDD, and hence do not contribute to *nodeSavings*(v).

Given these two measures, we can determine whether or not replacing v by ZERO will increase or decrease the overall cost; if it will decrease, then we greedily make the replacement. We continue processing each node in turn until all non-constant nodes have been processed.

We now detail the four major steps of the algorithm, and illustrate it on the BDD in Figure 4.14. To simplify the presentation, we assume that complement pointers are not used in the BDD. However, the implementation must ultimately take into account complement pointers, because all present-day BDD packages use them. The main complication is that replacing a node v by ZERO will actually *add* minterms to the onset of f, if v can be reached by an odd number of complement pointers.

4.5.3.1 Step 1: Compute the *onsetFraction* of each node v in f

For the Boolean function rooted at v, *onsetFraction*(v) gives the ratio of the size of the onset to the size of the entire Boolean space. This figure can be computed for all nodes of f in linear time by applying DFS from the root of f. The terminal cases of the recursion are

$$\begin{aligned} \textit{onsetFraction}(\text{ONE}) &= 1, \text{ and} \\ \textit{onsetFraction}(\text{ZERO}) &= 0. \end{aligned}$$

The *onsetFraction* of a non-constant node is computed in terms of the *onsetFraction* of its two children:

$$\textit{onsetFraction}(v) = \frac{1}{2}\textit{onsetFraction}(v.\textit{left}) + \frac{1}{2}\textit{onsetFraction}(v.\textit{right}).$$

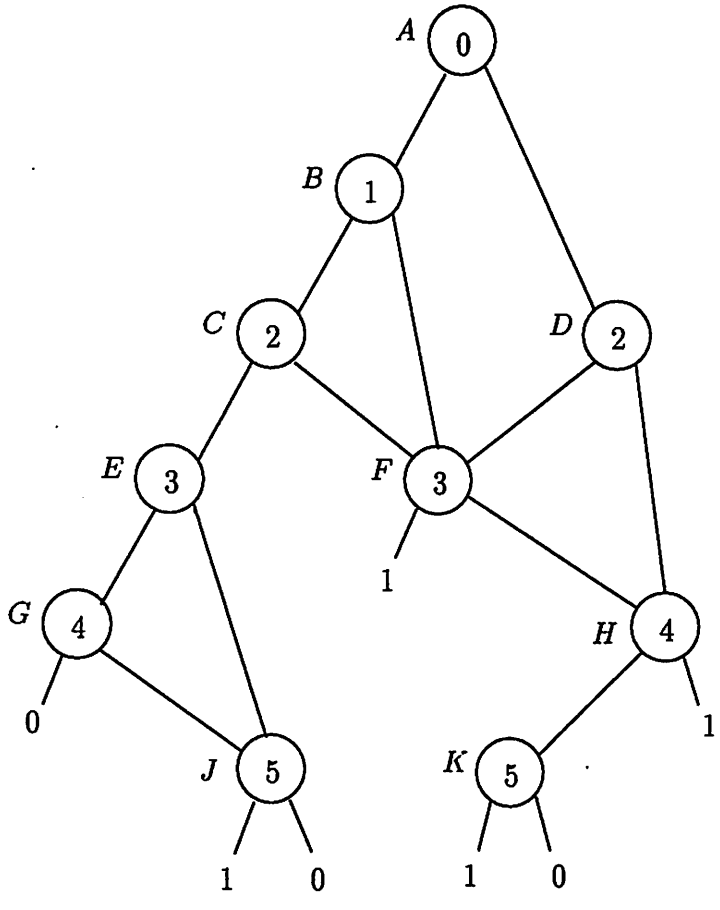


Figure 4.14: BDD used to illustrate the *bddUnderApprox* algorithm.

The *onsetFraction* for each node in our example BDD is shown in Table 4.1. For example,

$$onsetFraction(C) = \frac{1}{2} \cdot \frac{3}{8} + \frac{1}{2} \cdot \frac{7}{8} = \frac{5}{8}.$$

node	*onset-Fraction*	*function-RefCount*	*node-Sav-ings*	*num-Min-terms*	*num-Onset*	*costBenefit*, α					
						0	.2	.4	.6	.8	1
A	25/32	1	10	64	50	10	3.7	-2.6	-8.9	-15.2	-21.5
B	3/4	1	5	32	24	5	1.4	-2.2	-5.8	-9.4	-13.0
C	5/8	1	4	16	10	4	1.9	-0.2	-2.3	-4.4	-6.5
D	13/16	1	1	32	26	1	-2.0	-4.9	-7.9	-10.8	-13.8
E	3/8	1	3	8	3	3	2.0	0.9	-0.1	-1.2	-2.2
F	7/8	3	1	40	35	1	-2.6	-6.2	-9.8	-13.4	-17.0
G	1/4	1	1	4	1	1	0.6	0.3	-0.1	-0.4	-0.8
H	3/4	2	2	36	27	2	-1.2	-4.5	-7.7	-11.0	-14.2
J	1/2	2	1	6	3	1	0.4	-0.3	-0.9	-1.6	-2.2
K	1/2	1	1	18	9	1	-0.4	-1.8	-3.2	-4.6	-6.0

Table 4.1: The *bddUnderApprox* algorithm applied to the BDD of Figure 4.14.

4.5.3.2 Step 2: Compute the *functionRefCount* of each node v in f

functionRefCount(v) gives the number of edges pointing to v from within the function f; it excludes pointers from other functions within the same BDD manager. This figure can be computed for all nodes of f in linear time by performing BFS from the root. The *functionRefCount* of each node is initialized to 0. Then, for each node visited, the *functionRefCount* of each of its children is incremented by one. The *functionRefCount* of each node of our example BDD is shown in Table 4.1.

4.5.3.3 Step 3: Approximate the BDD

This step is the heart of the procedure. The nodes are visited via BFS. The subgraph rooted at a node is replaced by ZERO if this reduces the overall cost of the solution. When this happens, the *functionRefCount*s of some nodes in the subgraph are decremented.

The details of this step are now given. For each node v visited that has a non-zero *functionRefCount*, the following three actions are performed.

Action 1: Compute *nodeSavings*(v), the number of nodes that would be eliminated in f if just the subgraph rooted at v was replaced by ZERO. This can be computed by performing a local BFS starting from v. Each node has a *localRefCount*, which is initialized to *functionRefCount* each time a local BFS is commenced. When a node u is visited during a BFS, if its *localRefCount* is non-zero, then u is not explored further, and it does not contribute to *nodeSavings* of v; a non-zero *localRefCount* indicates that such a node is being shared by other parts of the BDD for f. On the other hand, if the *localRefCount* of u is zero, then *nodeSavings*(v) is incremented, and the *localRefCounts* of u's two children are decremented by one.

Consider computing *nodeSavings* for node B in our example. By definition, B itself contributes 1 to *nodeSavings*(B). The *localRefCount* of the children of B are decremented: for C, 1 is decremented to 0, and for F, 3 is decremented to 2. Next, C is visited, and since its *localRefCount* is now 0, *nodeSavings*(B) is incremented (to 2), and the *localRefCounts* of E and F are decremented, to 0 and 1, respectively. Say F is visited next. Its *localRefCount* is not 0, so we skip over F and proceed to E. Its *localRefCount* is 0, and proceeding in this fashion, we see that E, G and F all contribute to *nodeSavings*(B). Hence, *nodeSavings*(B) is 5. The *nodeSavings* for other nodes is shown in Table 4.1.

This step, repeated for each node, can lead to overall quadratic running time (consider a BDD that is just a single chain of nodes; BFS from each node will explore the rest of the chain). However, because the local BFS search from a node is pruned at nodes whose *localRefCount* is non-zero, the running time in practice should be nearly linear.

Action 2: Compute *numOnset*(v), the number of minterms in the onset of f that would be removed if v was replaced by ZERO. This can be computed by multiplying *onsetFraction*(v) by *numMinterms*(v). *NumMinterms*(v) records how many of the 2^n minterms of the Boolean space "pass through" v. For the root of f, *numMinterms* is 2^n. As each node u (that is not replaced by ZERO) is visited in the global BFS of Step 3, *numMinterms* of each child of u is incremented by one-half of *numMinterms*(u).

In our example,

$$
\begin{aligned}
\mathit{numMinterms}(A) &= 2^6 + 64, \\
\mathit{numMinterms}(B) &= \frac{1}{2}\mathit{numMinterms}(A) = 32, \\
\mathit{numMinterms}(F) &= \frac{1}{2}\mathit{numMinterms}(B) + \frac{1}{2}\mathit{numMinterms}(C) + \frac{1}{2}\mathit{numMinterms}(D)
\end{aligned}
$$

$$= \quad 16 + 8 + 16 = 40$$

Hence,

$$\begin{aligned} numOnset(F) &= onsetFraction(F) \cdot numMinterms(F) \\ &= \frac{7}{8} \cdot 40 = 35. \end{aligned}$$

For each node, *numOnset* can be computed in constant time.

Action 3: Compute *costBenefit*(v), which measures the change in cost of the solution if v were replaced by ZERO, to yield the function f_{new}. This is computed as follows.

$$\begin{aligned} costBenefit &= cost(f) - cost(f_{new}) \\ &= [\alpha(\log_2^2 |offset(f)|) + (1-\alpha)|\mathrm{BDD}(f)|] \\ &\quad -[\alpha(\log_2^2 |offset(f_{new})|) + (1-\alpha)|\mathrm{BDD}(f_{new})|] \\ &= \alpha[\log_2^2 |offset(f)| - \log_2^2 |offset(f_{new})|] \\ &\quad +(1-\alpha)(|\mathrm{BDD}(f)| - |\mathrm{BDD}(f_{new})|) \\ &= \alpha[\log_2^2(|offset(f)|) - \log_2^2(|offset(f)| + numOnset(v))] \\ &\quad +(1-\alpha)nodeSavings(v) \end{aligned}$$

If *costBenefit*(v) is greater than zero, then the flag *replaceByZero*(v) is set, and the *functionRefCount* of v's two children are decremented by one. If this causes *functionRefCount* of a child to fall to zero, then the *functionRefCount*s are recursively decremented. For example, if *costBenefit*(B) is greater than zero, *functionRefCount*(F) will fall to one, and *functionRefCount* of C, E and G will fall to zero.

If *costBenefit*(v) is less than or equal to zero, then the flag *replaceByZero*(v) is reset, and *numMinterms* of each child of v is incremented by one-half of *numMinterms*(v).

The *costBenefit* computed for a node v is affected by which other nodes have been marked for replacement by ZERO. In particular, *numMinterms*(v) may decrease, and *nodeSavings*(v) may increase, as nodes at or above the level of v are marked for replacement by ZERO (of course, if *functionRefCount*(v) falls to zero, then *costBenefit*(v) is irrelevant). By processing the nodes in a top-down fashion, *costBenefit*(v) needs to be computed just once, when v is considered for replacement.

Even though *costBenefit*(v) is affected by the actions above v, the values for *costBenefit* in Table 4.1 are computed, for illustration purposes only, assuming that no nodes

processed before a given node are marked for replacement. Also, since *costBenefit* is a function of α, the value of *costBenefit* is shown for 6 different values of α. As expected, as α tends to one, the *costBenefit* becomes negative, meaning replacement by ZERO is undesirable.

4.5.3.4 Step 4: Build the new BDD

This process starts from the root and proceeds recursively in DFS fashion. If the constants ZERO or ONE are reached, then that constant is returned. If a node marked *replaceByZero* is reached, then ZERO is returned. Otherwise, for a node labeled by variable x, a new node is created labeled with x and with children formed by the recursive building process.

For our example, suppose $\alpha = 0.4$. Then the first node processed with positive *costBenefit* is E. In fact, this is the only node replaced by ZERO (G becomes irrelevant once E is replaced). The new BDD is shown in Figure 4.15. Whereas the original BDD had 50 onset minterms and 10 nodes, the new BDD has 47 onset minterms and 7 nodes.

4.5.3.5 Discussion

Greedily choosing one node at a time for replacement by ZERO may lead to a suboptimal solution. Consider the partial BDD shown in Figure 4.16. Both of the children of node A are shared by other parts of the function, so *nodeSavings*(A) is one (node A itself). Hence, unless α is nearly zero, A probably will not be replaced by ZERO. Likewise, *nodeSavings*(B) is one and B probably will not be replaced. However, if we considered replacing A and B simultaneously by ZERO, we would find that *nodeSavings*($\{A, B\}$) is $K+2$, where K is the number of nodes in the common subgraph of A and B. If K is large, this may trigger replacement.

The algorithm could be modified easily to consider pairs of nodes for replacement. However, this would increase the complexity of the algorithm, which may make it impractical for intermediate to large BDDs. Also, instead of pairs of nodes, we might want to consider larger sets for replacement.

Another limitation with our approach comes in selecting the value for α. In our example, for $\alpha = 0$, all nodes qualify for replacement, whereas at $\alpha = 0.6$, none of the nodes qualify. Combining the onset size and the BDD size terms in the same equation makes it

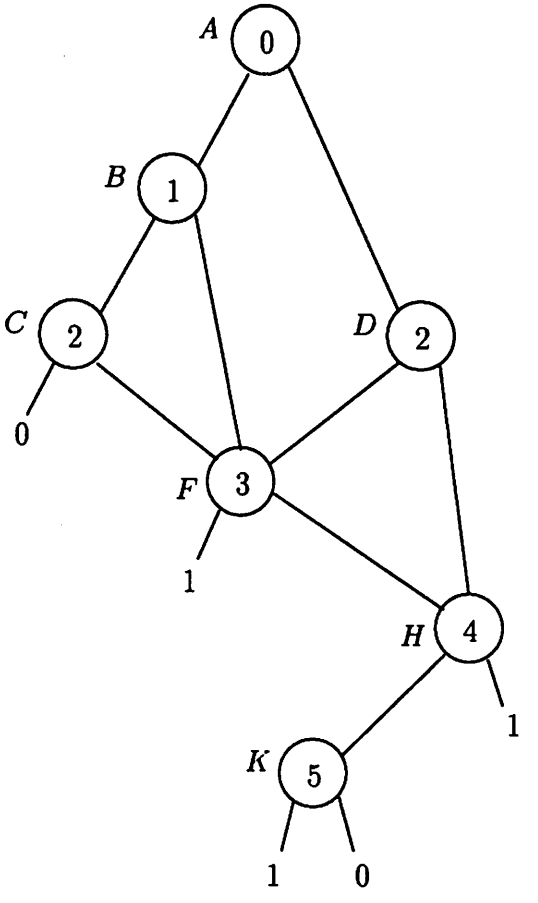


Figure 4.15: The result of *bddUnderApprox* applied to the BDD of Figure 4.14. E is replaced by ZERO.

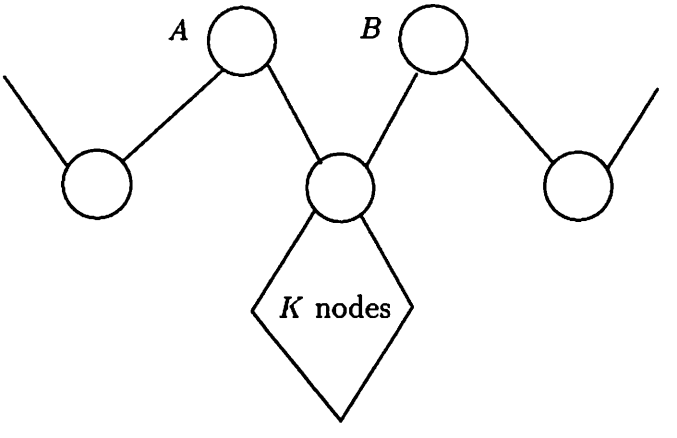


Figure 4.16: Neither A nor B will be replaced by ZERO when considered individually, but may be replaced by ZERO if considered simultaneously.

difficult to select a precise value of α that distinguishes "good" replacements from "bad" replacements. Possibly a threshold-based approach, like that of Ravi and Somenzi, might be more robust.

4.5.4 Application to binary Boolean operations

In our algorithm for deciding reachability, we frequently want to find a good approximation to the Boolean combination of a pair of functions, for example, the conjunction $f \cdot g$. We *could* define a new BDD operator that takes as input two functions and returns an approximation to their conjunction. Instead, we take an alternate approach where we form the conjunction exactly, and then approximate the result. This approach begs the question if we are able to form the conjunction exactly. We can, as long as we keep small the intermediate BDDs of the reachability computation. In other words, if we apply *bddApprox* to all intermediate BDDs, then we should be able to performs local computations exactly. A benefit of this approach is that we can concentrate on developing heuristics for just a single problem, the BDD approximation problem.

4.6 Approximating sets of edges

We concluded Section 4.4 by listing the three major subroutines of our algorithm to decide reachability. Each of these subroutines involves approximating a set of edges; we now discuss each in detail.

4.6.1 Initial over-approximation of G

The goal of this subroutine is to find a superset V of the edges of G such that V has low *cost*, as defined in Definition 4.2. As a reminder, the function we wish to approximate is

$$G(x, y) = \exists u \prod_{i=1}^{l} T_i(x, u, y_i).$$

The first step is to build each T_i. Then, some u variables are "cut" to partition the T_i's into a set of clusters. Next, each cluster is built separately, and finally the clusters are conjuncted to yield the over-approximation V. At each step of this process, *bddOverApprox* is used to control the size of the BDDs.

4.6.1.1 Building each T_i

As stated earlier, $T_i = (y_i \equiv \delta_i(x, u))$, where δ_i is the next state function of the ith flip-flop. The BDD for δ_i may be too large to build. In this case, T_i can be represented by the conjunction of a set of smaller terms by introducing intermediate variables.[3]

Specifically, starting at the combinational inputs x and u and proceeding in topological order through the combinational network, we begin by constructing the BDD for each network node in terms of the combinational inputs. However, if the BDD for the function g_j of a given node v_j exceeds a user-settable threshold, then an intermediate variable p_j is introduced at node v_j. Then the BDDs of nodes in the fanout of v_j are built in terms of p_j. T_i can then be expressed as the conjunction of the terms $(p_j \equiv g_j)$, with the intermediate variables existentially quantified. Each of these terms can be over-approximated using *bddOverApprox* so that their product has a reasonable BDD size. In the sequel, we refer to the result of this step as $T_i(x, u, y_i)$, regardless of whether or not T_i has in fact been approximated.

4.6.1.2 Cutting u variables

The next step is to "cut" some u variables to partition the T_i's into a set of clusters. This approximation relies on the observation that $(\exists x f) \cdot (\exists x g) \supseteq \exists x (f \cdot g)$. The idea is to cut some of the u variables by moving them into the product. For example, we might cut u_1 by replacing

$$\exists u_1, u_2 \, [T_1(x, u_1, u_2, y_1) \cdot T_2(x, u_1, y_2)]$$

by

$$(\exists u_1, u_2 \, T_1(x, u_1, u_2, y_1)) \cdot (\exists u_1 \, T_2(x, u_1, y_2)).$$

Equivalently, the problem is to cluster the T_i's; any u variables passing between clusters are cut. We want to minimize the number of cut variables, so that the amount of over-approximation is minimized. We formulate the problem as a traditional graph partitioning problem on hypernets. In particular, we create an undirected graph, where each flip-flop is represented by a vertex, and there exists an edge labeled by u_k between vertices i and j if T_i and T_j both depend on u_k. Then we successively apply graph bipartitioning (using, for example, the Fiduccia-Mattheyses algorithm), minimizing the number of u variables cut.

[3] This technique has been used by others in a variety of settings, e.g., [52, 53, 54].

The size of each partition is limited by a user provided parameter, giving the maximum of the sum of BDD sizes for each partition.[4]

4.6.1.3 Building each cluster

At this point, we must construct the graph $C_j(x, y)$ for each cluster:

$$C_j(x, y) = \exists u \prod_{i \in J} T_i(x, u, y_i)$$

where J is the set of flip-flops in the jth partition. First, we find a schedule for the conjunctions and quantifications (for example, using the techniques in [45]). In general, this may be in the form of a tree. Then we build C_j according to this schedule, but we apply *bddOverApprox* to intermediate results to avoid large BDDs. In particular, there are two types of intermediate computations.

1. Conjunction: form the conjunction exactly and then apply *bddOverApprox* to the result.

2. Existential quantification: apply *bddOverApprox* to the results of the intermediate disjunctions (i.e., $f_x + f_{\overline{x}}$), and to the final result of existential quantification.

4.6.1.4 Conjuncting the clusters

The last step is to take the product of the clusters C_j. Again, we form each conjunction exactly, and then apply *bddOverApprox* to the result. The final result is the approximation V.

V should have no more than approximately 10,000 BDD nodes, so that we can manipulate it efficiently. Hence, we need some dynamic control to make sure that V does not exceed this limit. This could take the form of stopping the computation when the limit is exceeded, and restarting it with a lower value of α (i.e., a worse approximation, but smaller BDD size); or we could just restart the phase of conjuncting clusters with a lower value of α.

[4]The flip-flop partitioning technique of Cho *et al.* [55] could also be applied to the present problem.

4.6.2 Initial under-approximation of G

The goal of this subroutine is to find a subset U of the edges of G such that U has low *cost*, as defined in Definition 4.7. Computing U follows the same outline as computing the over-approximation V.

We cut the u variables using the same partition found in computing V. However, rather than *existentially* quantifying the cut variables, we now *universally* quantify them, relying on the fact that $(\forall x f) \cdot (\forall x g) \subseteq \exists x (f \cdot g)$. To build each cluster, we use under-approximation on intermediate results, rather than over-approximation.

4.6.3 ApProxIMaTion of edges from I in V

The variable *fromI* in the reachability algorithm contains those edges in the current over-approximation V that pass from a state in I to a state not in I. The set *fromI* can be partitioned into two sets.

1. True edges: these are edges that exist in the exact graph G and that, when added to the under-approximation U (line 28), allow the forward traversal in U to progress (line 6).

2. False edges: these are edges that do *not* exist in G, and that, when removed from V (line 27), further restrict the set of potential paths from I to F (line 10).

The set of true edges E is the set of *all* edges from I in V, restricted to the exact graph $G(x, y)$:

$$\begin{aligned} E(x,y) &= G(x,y) \cdot \mathit{fromI}(x,y) \\ &= (\exists u \prod_{i=1}^{l} T_i(x,u,y_i)) \cdot \mathit{fromI}(x,y) \end{aligned}$$

The set of false edges is then just the set difference of E from V, $V(x,y) \setminus E(x,y)$.

Ideally, we would like to determine the partition exactly. Unfortunately, this problem is hard, as deciding if just a single edge is true or false is already NP-complete. Thus, we settle for approximating the sets of true and false edges. We want to find some edges from I that are definitely false (line 17), and some that are definitely true (line 18); the status of the remainder of the edges in *fromI* will be unknown.

Clearly, an under-approximation of E yields an approximation to the true edges, and an over-approximation of E yields an approximation to the false edges. To compute

these approximations of E, we would like to rewrite the equation for E so that it has the same form as the equation for G, thus permitting the application of the procedures outlined in Sections 4.6.1 and 4.6.2. This can be done simply:

$$\begin{aligned} E(x,y) &= \exists u[(\prod_{i=1}^{l} T_i(x,u,y_i)) \cdot \mathit{fromI}(x,y)] \\ &= \exists u[\prod_{i=1}^{l} (T_i(x,u,y_i) \cdot \mathit{fromI}(x,y))] \\ &= \exists u \prod_{i=1}^{l} T_i'(x,u,y_i) \end{aligned}$$

where $T_i' = T_i \cdot \mathit{fromI}$.[5] Thus, we have expressed the set E of true edges from I in V as the product of individual transition relations, and we can apply the procedures of Sections 4.6.1 and 4.6.2 to compute an over-approximation and under-approximation, respectively. Figure 4.17 illustrates the various sets involved in the above computation.

$$\begin{aligned} G &= \text{exact graph} \\ V &= \text{current over-approximation of } G \\ U &= \text{current under-approximation of } G \\ \mathit{fromI} &= \text{edges from } I \text{ in } V \\ E &= G \cap \mathit{fromI} \\ V' &= \text{over-approximation of } E \\ U' &= \text{under-approximation of } E \\ \text{approx. false edges} &= \mathit{fromI} \setminus V' \\ \text{approx. true edges} &= U' \end{aligned}$$

Note that V does not necessarily contain G, because some true edges are removed from V at line 10.

The procedure for approximating the true and false edges to F follows analogously.

[5] Alternatively, we could define $T_i' = \mathit{restrict}(T_i, \mathit{fromI})$, using the *restrict* operator of [6] . Then $\exists u \prod T_i'$ is no longer exactly $E(x,y)$, but can still be used to form approximations.

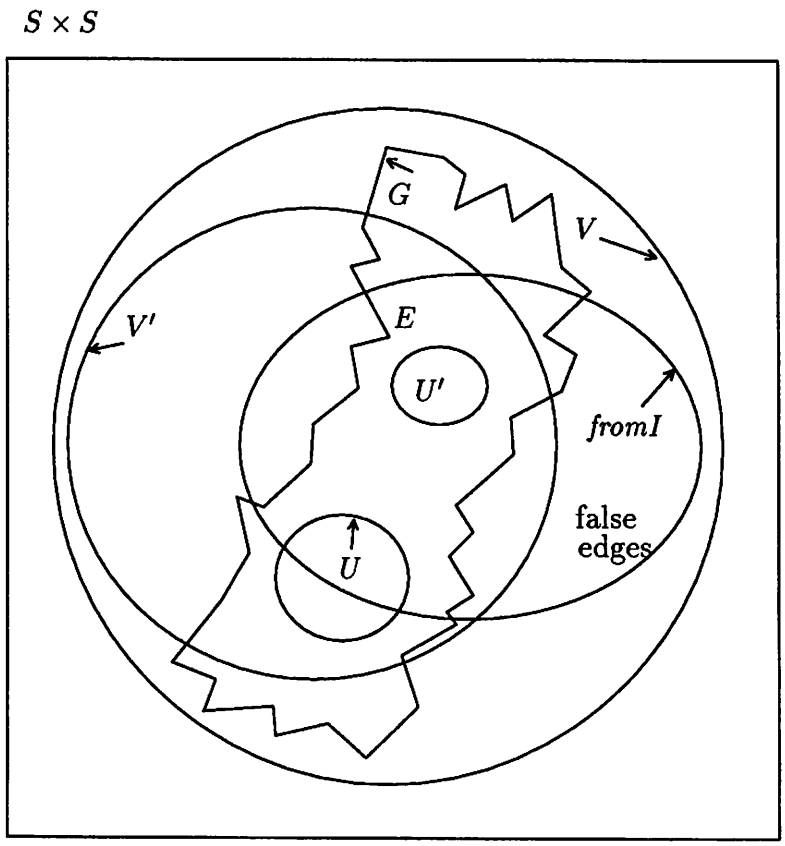


Figure 4.17: Diagram showing over- and under-approximations to $E(x, y)$.

4.7 Summary and future work

We have presented a technique for deciding state reachability for large FSMs. Specifically, we seek to answer if there exists a path from a set I of initial states to a set of F of final states in an FSM. Several problems in logic synthesis, formal verification, and testing can be reduced to this question, and hence an efficient algorithm for solving this problem would have great benefit.

Our approach constructs an over-approximation V, and an under-approximation U, to the state transition graph G. Then, the potential witness paths from I to F in V are used to guide the search for a true path in U from I to F.

The success of our approach hinges on the quality of the approximations that we construct. A good approximation is one that retains most of the original information, yet has a small representation. We use BDDs to represent the set of edges of a state graph. We have formulated a general optimization problem, called the *bddApprox* problem, which seeks to find a set representing a close approximation of another set, and yet having a small BDD representation. We presented a heuristic for solving the *bddApprox* problem. Ravi and Somenzi formulated the same problem, and presented several heuristics. We suspect that others will find applications for the *bddApprox* problem, and will develop more heuristics for its solution.

As with any heuristic for solving a hard problem, our approach can ultimately be validated only by implementing the algorithms and testing them on a set of examples. This remains as future work.

Chapter 5

Summary

This dissertation has addressed three problems concerning the formal analysis of synchronous circuits and FSMs. The first problem is that of analyzing the logical behavior of synchronous circuits containing combinational cycles. We formalized what it means for a circuit to be output-stable, and provided decision procedures to classify circuits based on their output behavior.

Next, a new heuristic for model checking CTL formulas on a system of interacting FSMs was presented. This method defines a formula-dependent equivalence relation on the states of the component FSMs, and uses this equivalence to simplify the components before forming their product.

Finally, the problem of state reachability in FSMs, or that of determining if one set of states can reach another set, was addressed. As part of the solution presented to this problem, we defined a new problem, the BDD approximation problem.

The second and third problems have existed for many years, and this work attempts to provide new solutions to them to extend the size of FSMs that can be handled. The first problem has been formulated here, and algorithms are provided to solve it; now, circuits with combinational cycles need not be summarily rejected. Taken together, the work presented on these problems should advance the analysis of synchronous circuits.

Appendix A

VIS: Verification Interacting with Synthesis

A.1 Introduction

This manual provides a brief overview of the architecture of VIS. The first section looks at VIS as a whole, and subsequent sections cover the major components of VIS.

VIS was designed to be modular and lightweight. By understanding the architecture of the system, future VIS developers can work to maintain these attributes.

A.2 VIS

VIS is partitioned into three main components:

1. VIS-F — The front end. It provides the ability to read and write BLIF-MV files, and supports a hierarchical data structure mimicking the constructs of BLIF-MV.
2. VIS-V — The verification system. This provides facilities for combinational and sequential equivalence checking, fair CTL model checking, and cycle-based simulation.
3. VIS-S — The synthesis system. This provides state minimization, variable encoding, and hierarchical restructuring capabilities.

Figure A.1 is a block diagram showing how the three components interact. The packages that constitute each component are listed along with edges denoting dependencies

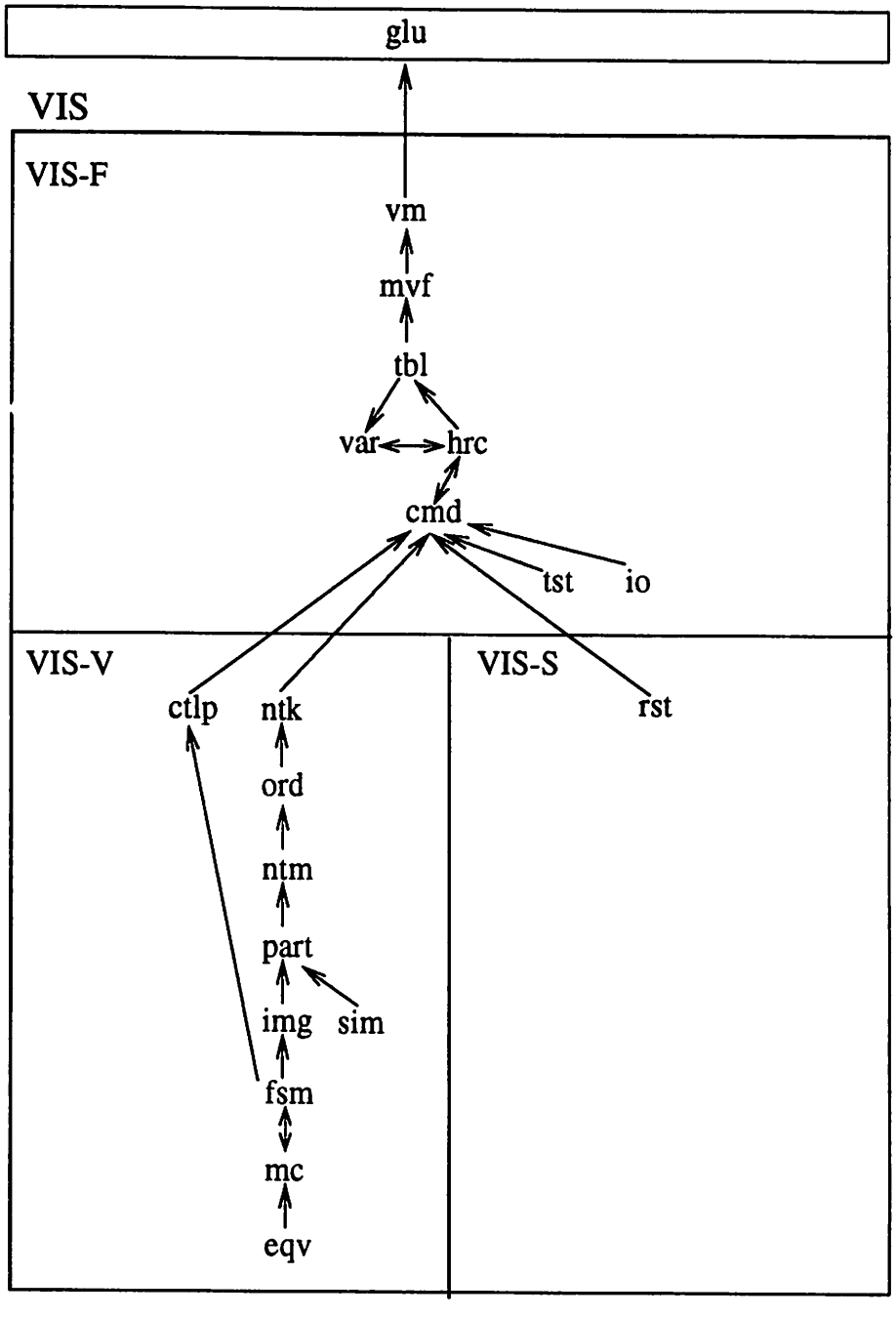


Figure A.1: Components and packages of VIS. An edge from package A to B denotes that A depends on B (edges implied by transitivity are not shown).

among the packages. `glu` is the Generic Libraries Utility, which contains utility packages such as `array`, `list`, and `bdd`. Note that

- VIS-F does not depend on VIS-V or VIS-S, and
- VIS-V and VIS-S are independent.

The first point allows VIS to be easily compiled leaving out VIS-V, VIS-S, or both, to produce an executable containing a subset of the capabilities. The second point forces communication between verification and synthesis to occur via the front end, rather than directly.

The division of VIS into the three components is not reflected in the directory structure of the source code. Instead, all packages are kept within a single directory named `src`.

A.3 VIS-F: Front End

VIS-F is the front end. It provides an in-memory representation of BLIF-MV. This hierarchical representation can be traversed and manipulated. VIS-F consists of the following packages:

- `vm` — Contains the `main()` function, and provides the compilation date, version number, and the location of the VIS library.[1]
- `cmd` — The interactive command interface. Provides a global table to store values for user-settable variables (e.g., the value `autoexec`). Also provides the system level commands like `help`, `alias`, and `set`.[2]
- `mvf` — A data structure to represent multi-valued input, multi-valued output functions, based on BDDs.
- `tbl` — A data structure to represent multi-valued relations, in particular the `.table` construct in BLIF-MV.
- `var` — A data structure to represent multi-valued variables, in particular the `.mv` construct in BLIF-MV.

[1] Largely borrowed from the `main` package of SIS.
[2] Largely borrowed from the `command` package from SIS.

- `hrc` — Data structures to represent a hierarchical design, in particular the `.model`, `.subckt`, and `.latch` constructs in BLIF-MV.
- `io` — Routines to read and write BLIF-MV and BLIF files.
- `tst` — A package template that can be used as the starting point for the creation of new packages.

When a BLIF-MV file is parsed, a directed acyclic graph of models is created by the `io` package, corresponding to the hierarchy given in the file. The DAG is then transformed by `io` into a tree by creating separate nodes for each instantiation of a model. Traversal and manipulation of the hierarchy takes place on the tree, and not the DAG, using routines provided by `hrc`.

The `Hrc_Node_t` data structure provides a lookup table for applications (e.g., VIS-V and VIS-S) to store data associated with a node in the hierarchy. In this manner, VIS-F can remain independent of VIS-V and VIS-S.

A.4 VIS-V: Verification

VIS-V provides analysis capabilities for designs. From any node in the hierarchy (the *current* node), executing the command `flatten_hierarchy` causes a flattened *network* to be created, representing everything from the current node down to the leaves of the hierarchy. Having a flattened representation in which all combinational "gates" and latches exist in a single network allows for the global analysis of that part of the design encompassed by the current node of the hierarchy. The packages of VIS-V are:

- `ntk` — A directed graph representation, where the vertices are "gates," inputs and latches. Combinational cycles are not precluded by the network data structure, but many of the packages assume the absence of combinational cycles.
- `ord` — Routines to order the MDD variables of a network, based on the structure of the network. Also provides an interface to dynamic ordering of variables.
- `ntm` — A routine to build the `Mvf_Function_ts` of the roots of an arbitrary region of a network, in terms of the leaves of the region. The leaves can be treated as variables or as specific constants.

- `part` — Routines to build an MVF representation of a network. The MVFs are stored at the vertices of a DAG, where the sinks correspond to the combinational outputs of the network, and the sources to the combinational inputs. In general, intermediate vertices can be introduced to control the size of the MVFs.

- `sim` — A cycle-based network simulator. Simulation is performed by evaluating the MVFs provided by the `part` package. Simulation vectors can be provided by the application, or random simulation can be performed.

- `img` — Generic routines for performing forward and backward image computation. The routines work off the graph of MVFs provided by the `part` package, and have no direct knowledge of the `ntk` or `fsm` packages. Since this is an active area of research, a generic interface has been designed to easily allow the addition of new computation methods.

- `fsm` — An abstraction of a network. The FSM does not actually store the next state functions of the FSM — these are provided by `part`. It does store the vectors of present state and next state variables, reachability information, fairness constraints-related information, and image computation information.

- `mc` — A fair CTL model checker and debugger for FSMs.

- `eqv` — Routines for performing combinational equivalence between regions of two networks, and for performing sequential equivalence between two FSMs.

- `ctlp` — A CTL parser.

The nodes of a network have a single output. The function of a combinational node of a network is represented by a `Tbl_Table_t`. A k-output table in the hierarchy is represented by k combinational nodes in the corresponding network, where each of the k nodes points to the same table, but are distinguished by which output column they represent. This splitting is done by the flattening routine in the `ntk` package.

Throughout VIS-V, it is assumed that the combinational outputs are completely specified and deterministic. Non-determinism is introduced via *pseudo-inputs*. A pseudo-input is like a non-deterministic constant that can update its value on each clock cycle. See the documentation for the `ntk` package for more information on pseudo-inputs.

Because the emphasis of VIS-V is on analysis, the ability to modify the network data structure is not provided. It is assumed that once a network is created from the hierarchy, the network will be unchanged until it is destroyed.

Notice that the `ntk` package is independent of all other packages in VIS-V. This independence is maintained by allowing applications (e.g., `fsm`, `part`) to store information associated with a network in a lookup table. It is important to maintain this independence so that the `ntk` data structures do not become cluttered.

A.5 VIS-S: Synthesis

Only one package has been written thus far for this component. It is anticipated that packages will be added to support state minimization, variable encoding, and other operations. However, note that VIS-F already allows a BLIF file to be written, which can be massaged by the sequential synthesis system SIS and read back into the hierarchy. The packages of VIS-S are:

- `rst` — Routines for restructuring the hierarchy.

A.6 Possible Improvements

1. The `mvf` package could be located in the Generic Libraries Utility.

2. One existing complication in removing the restriction that networks can't be modified is that the `Tbl_Table_t` and `Var_Variable_t` data within a network are owned by (and hence freed by) the hierarchy manager (`hrc` package). This could be resolved by creating a global manager for tables and a global manager for variables.

Bibliography

[1] Sharad Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, July 1994.

[2] Leon Stok. False loops through resource sharing. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 345–348, November 1992.

[3] Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits.* Springer-Verlag, 1995.

[4] Kavita Ravi and Fabio Somenzi. High-density reachability analysis. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 154–158, November 1995.

[5] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, August 1986.

[6] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of sequential machines using Boolean functional vectors. In *Proceedings of the IFIP International Workshop, Applied Formal Methods for Correct VLSI Design*, November 1989.

[7] Hervé J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 130–133, November 1990.

[8] Thomas R. Shiple, Ramin Hojati, Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton. Heuristic minimization of BDDs using don't cares. In *Proc. 31st Design Automat. Conf.*, pages 225–231, San Diego, CA, June 1994.

[9] Yosinori Watanabe and Robert K. Brayton. The maximum set of permissible behaviors for FSM networks. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 316–320, November 1993.

[10] Thomas R. Shiple, Vigyan Singhal, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Analysis of combinational cycles in sequential circuits. In *Proc. Int'l Symposium on Circuits and Systems*, pages 592–595, vol. IV, May 1996.

[11] Thomas R. Shiple, Gérard Berry, and Hervé Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference*, pages 328–333, March 1996.

[12] Jerry R. Burch. David Dill, Elizabeth Wolf, and Giovanni De Micheli. Modeling hierarchical combinational circuits. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 612–617, November 1993.

[13] Oded Maler and Amir Pnueli. Timing analysis of asynchronous circuits using timed automata. In Paolo E. Camurati and Hans Eveking, editors, *Proceedings of the Conference on Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, pages 189–205, Frankfurt/Main, Germany, October 1995. Springer-Verlag.

[14] Sharad Malik. Analysis of cyclic combinational circuits. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 618–625, November 1993.

[15] Nicholas Halbwachs and Florence Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Euromicro'95*, September 1995. Como, Italy.

[16] Randal E. Bryant. Extraction of gate level models from transistor circuits by four-valued symbolic analysis. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 350–353, November 1991.

[17] Randal E. Bryant. Boolean analysis of MOS circuits. *IEEE Trans. Computer-Aided Design*, 6(4):634–649, July 1987.

[18] Kanwar Jit Singh and P. A. Subrahmanyam. Extracting RTL models from transistor netlists. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 11–15, November 1995.

[19] Manish Pandey, Alok Jain, Randal E. Bryant, Derek Beatty, Gary York, and Samir Jain. Extraction of finite state machines from transistor netlists by symbolic simulation. In *Proc. Int'l Conf. on Computer Design*, pages 596–601, October 1995.

[20] Timothy Kam and P. A. Subrahmanyam. Comparing layouts with HDL models: A formal verification technique. *IEEE Trans. Computer-Aided Design*, 14(4):503–509, April 1995.

[21] M. J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, New York, 1979.

[22] G. D. Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[23] Gérard Berry. The constructive semantics of pure Esterel. *To Appear*, 1996.

[24] William H. Kautz. The necessity of closed circuit loops in minimal combinational circuits. *IEEE Trans. Comput.*, 19(2):162–164, February 1970.

[25] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer-Verlag, 1993.

[26] Adnan Aziz and Robert K. Brayton. Verifying interacting finite state machines. Technical Report UCB/ERL M93/52, Electronics Research Laboratory, U.C. Berkeley, July 1993.

[27] Robin Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.

[28] R. Paige and R.E. Tarjan. Three partition-refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[29] R. J. van Glabbeek. The linear time - branching time spectrum. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90, Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, August 1990.

[30] Ellen M. Sentovich, Kanwar Jit Singh, Cho Moon, Hamid Savoj, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proc. Int'l Conf. on Computer Design*, October 1992.

[31] Ramin Hojati, Robert K. Brayton, and Robert P. Kurshan. BDD-based debugging of designs using language containment and fair CTL. In Costas Courcoubetis, editor, *Proceedings of the Conference on Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 41–58. Springer-Verlag, June 1993.

[32] Vigyan Singhal, Carl Pixley, Adnan Aziz, and Robert K. Brayton. Exploiting power-up delay for sequential optimization. In *European Design Automation Conference*, Brighton, Great Britain, September 1995.

[33] Edmund M. Clarke, E. Allen Emerson, and Aravinda Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.

[34] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Characterizing Kripke structures in temporal logic. Technical Report CS 87-104, Department of Computer Science, Carnegie Mellon University, 1987.

[35] Adnan Aziz, Thomas R. Shiple, Vigyan Singhal, and Alberto L. Sangiovanni-Vincentelli. Formula-dependent equivalence for compositional CTL model checking. In David L. Dill, editor, *Proceedings of the Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 324–337, Stanford, CA, June 1994. Springer-Verlag.

[36] A. Bouajjani, J-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18(3):247–271, 1992.

[37] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *4th Annual Symposium on Logic in Computer Science*, Asilomar, CA, June 1989.

[38] Orna Grumberg and David E. Long. Model checking and modular verification. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR '91, International Conference*

on Concurrency Theory, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1991.

[39] Dennis Dams, Orna Grumberg, and Rob Gerth. Generation of reduced models for checking fragments of CTL. In Costas Courcoubetis, editor, *Proceedings of the Conference on Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 479–490. Springer-Verlag, June 1993.

[40] Massimiliano Chiodo, Thomas R. Shiple, Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton. Automatic compositional minimization in CTL model checking. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 172–178, November 1992.

[41] E. Allen Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier Science Publishers B.V., 1990.

[42] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, June 1989.

[43] Jerry R. Burch, Edmund M. Clarke, and David E. Long. Representing circuits more efficiently in symbolic model checking. In *Proc. 28th Design Automat. Conf.*, pages 403–407, June 1991.

[44] Daniel Geist and Ilan Beer. Efficient model checking by automated ordering of transition relation partitions. In David L. Dill, editor, *Proceedings of the Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 299–310, Stanford, CA, June 1994. Springer-Verlag.

[45] Ramin Hojati, Sriram C. Krishnan, and Robert K. Brayton. Early quantification and partitioned transition relations. In *Proc. Int'l Conf. on Computer Design*, October 1996.

[46] Rajeev K. Ranjan, Adnan Aziz, Robert K. Brayton, Bernard Plessier, and Carl Pixley. Efficient BDD algorithms for FSM synthesis and verification. In *International Workshop on Logic Synthesis*, pages 3–27 – 3–34, May 1995.

[47] Gianpiero Cabodi and Paolo E. Camurati. Exploiting cofactoring for efficient FSM symbolic traversal based on the transition relation. In *Proc. Int'l Conf. on Computer Design*, pages 299–303, October 1993.

[48] Felice Balarin. *Iterative Methods for Formal Verification of Digital Systems.* PhD thesis, University of California, Berkeley, 1994.

[49] Gianpiero Cabodi, Paolo E. Camurati, and Stefano Quer. Efficient state space pruning in symbolic backward traversal. In *Proc. Int'l Conf. on Computer Design*, pages 230–235, October 1994.

[50] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, October 1992.

[51] Hyunwoo Cho, Gary D. Hachtel, Enrico Macii, Bernard Plessier, and Fabio Somenzi. Algorithms for approximate FSM traversal. In *Proc. 30th Design Automat. Conf.*, pages 25–30, June 1993.

[52] Patrick C. McGeer, Kenneth L. McMillan, Alexander Saldanha, Alberto L. Sangiovanni-Vincentelli, and Patrick Scaglia. Fast discrete function evaluation using decision diagrams. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 402–407, November 1995.

[53] Jawahar Jain, Amit Narayan, Claudionor Coelho, Sunil P. Khatri, Alberto L. Sangiovanni-Vincentelli, Robert K. Brayton, and Masahiro Fujita. Combining Top-down and Bottom-up Approaches for ROBDD Construction. Technical Report UCB/ERL M95/30, Electronics Research Laboratory, U.C. Berkeley, April 1995.

[54] Rajeev K. Ranjan. Private communication, 1996.

[55] Hyunwoo Cho, Gary D. Hachtel, Enrico Macii, Massimo Poncino, and Fabio Somenzi. A structural approach to state space decomposition for approximate reachability analysis. In *Proc. Int'l Conf. on Computer Design*, pages 236–239, September 1994.