HIERARCHICAL MODELS OF SYNCHRONOUS CIRCUITS FOR FORMAL VERIFICATION AND SUBSTITUTION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Elizabeth Susan Wolf
September 1995

Abstract

As industrial circuit designs become larger and more complex, the use of simulation as the sole means for verification of their correctness no longer suffices. One of the potential methods to complement simulation is formal verification, in which mathematical methods are applied to prove that desired properties hold of circuit models.

In this thesis, we develop a mathematical model of synchronous sequential circuits that supports both automated formal hierarchical verification and substitution. In order to facilitate hierarchical verification, we model synchronous circuit specifications and implementations uniformly. Each of these descriptions provides both a behavioral and a structural view of the circuit or specification being modeled. For formal verification, our framework provides a means for comparison of the behavior of a circuit model to a requirements specification in order to determine whether the circuit is an acceptable implementation of the specification. For substitution, and to support a modular verification process, it provides a structural view of a circuit and the capability to plug in one component in place of another in a circuit model. This allows us to determine whether the new component constitutes an acceptable substitution in terms of the desired behavior of the full circuit. In addition, our model supports nondeterministic specifications, which capture the minimum requirements of a circuit without forcing us to overspecify by including irrelevant details.

Hierarchical descriptions of combinational circuits may often contain apparent loops. Previous existing formalisms have relied on syntactic methods for distinguishing apparent from actual unlatched feedback loops in hardware designs. However, these methods do not work correctly for nondeterministic models. Our model of the behavior of a synchronous circuit within a single clock cycle correctly handles such cyclic dependencies even in the presence of nondeterminism, by providing a semantic method to describe them.

In addition to developing a theoretical framework to support behavioral and structural comparison of synchronous circuit models at various levels of detail, we have implemented and proved the correctness of automatic decision procedures for both formal verification and substitution using these models. Our main substitution result is the derivation of a closed-form expression for the most general specification of the allowed substitutions for a component in a circuit, against which candidate components may be compared via the behavior comparison algorithms developed for formal verification. We describe our software implementation of these procedures.

Acknowledgements

I would like to thank my advisor, David Dill, for his careful and critical reading of the dissertation, and for his advice, collaboration and support throughout this research. I would also like to thank the other members of my reading committee, Jerry Burch, Carolyn Talcott, and Vaughan Pratt. Jerry Burch collaborated with me closely, and qualifies as co-advisor. His collaboration was critical to formulating many of the ideas presented in this thesis. Carolyn Talcott was invaluable in helping to smooth out the presentation of the formal theorems and proofs, and in drawing connections between this work and other ideas in computer science. Vaughan Pratt provided the NP-completeness proof that convinced us of the need to change definitions.

Even with all their help, I would not have finished this dissertation without the support of many other people. Thanks to Steve Nowick, Alan Hu, Robert Jones, Richard Ho and Jules Bergmann for technical help and advice. I would also like to thank Lydia Kavraki and Hugh McGuire for invaluable thesis-writing advice, all of the math library staff, officemates present and past, Janet Murdock and Marcia Derr, and of course my family.

Finally, I am grateful for external financial support from a National Science Foundation Graduate Fellowship and for support from Xerox Corporation through their fellowship program. The latter was arranged by Carolyn Tajnai. David Dill supported this work financially during summers and through my final year in the Ph.D. program, through a grant from Mitsubishi Electric Research Laboratories, Inc. and through NSF grant MIP-8858807. This work would not have been completed without their financial support.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Formal hardware verification is the use of mathematical methods to prove that desired properties hold of circuit models. It has not yet been fully incorporated into the industrial design process, but it is beginning to be explored in industry as a complementary validation method to simulation.

In current standard industrial practice, hardware designs are verified by means of extensive simulation. This method may provide extensive coverage but it is not exhaustive: modern circuit designs are simply too complex to be exhaustively simulated. Instead, an attempt is made to determine the critical input sequences for testing, and the circuit design is simulated on these test vectors. In contrast, formal verification provides assurance that the properties verified hold of the circuit model no matter what the input combination sequence it is given, and no matter how long that sequence is.

The goal of this thesis is to provide a mathematical model of synchronous sequential circuits that supports formal *hierarchical* verification. Our model must support formal substitution analysis, and must allow a specification to admit multiple correct implementations. In order to support all these objectives simultaneously, our framework supports the *expression* of non-cascade composition, and both structural and behavioral views of a synchronous circuit. We have also developed a self-contained model for *combinational* circuits that meets all these objectives.

In hierarchical verification, we do not distinguish between implementation models and requirements specifications, but instead allow a model to express a range of behavioral and structural constraints. At one end of this spectrum lie the models which allow arbitrary behavior and have no internal structural constraints at all, and at the other are those models so behaviorally and structurally constrained that they each model a unique gate-level circuit. We define a particular mathematical relation to hold between two models if the first is a correct implementation of the other (considered as a specification). We may verify that one such specification model meets the requirements imposed by another specification, and from this conclude that any circuit model which correctly implements the first specification also correctly implements the second. The process of designing such a sequence of models, each of which is more structurally and behaviorally constrained than the one before, is called refinement. In addition, a hierarchical verification framework must have the following *compositionality property:* a structurally composite specification is automatically satisfied by the composition of any correct implementations of its component specifications. Hence we may determine implementations that satisfy the component specifications independently of each other, a verification process that we call *piecewise verification.* In this way, hierarchical verification supports a modular design process.

For both substitution and *hierarchical* verification, we require the ability to express structural constraints on a circuit model. In other words, we want to be able to define a piece of a circuit and to talk about its connections to the rest of the circuit of which it is a part. Therefore our formal semantics of circuits must allow us to define primitive circuits and operations on them that correspond to wiring them together (circuit composition) and hiding wires (ignoring wires that are not primary outputs). This will allow us to construct behavioral models from the behavior of primitive or previously constructed components according to the structure of a composite circuit. We utilize the circuit algebra framework of [61] in order to provide this capability.

A central problem in modeling the synchronous behavior of clocked hardware is the question of how to model the inherently asynchronous behavior exhibited by such hardware during each clock cycle. In a Moore machine representation of a synchronous circuit or specification, *all* parts of the circuitry are assumed to be latched, so that output value combinations are completely determined by the current state [110]. In a Mealy machine model, the current output value combination is determined by both the current state and the current input value combination, so that this determination is assumed to be instantaneous [101]. We choose to use the Mealy machine model, in order to allow the composition of arbitrary forms of synchronous hardware, including the special case of pure combinational logic, to form more complex synchronous circuits.

However, manifest in this approach is the question of recognizing combinational feedback loops. If a model may be nondeterministic, and composition is an algebraic operator, then arbitrary models meeting given *syntactic* constraints must be composable. This allows the inadvertent creation of unlatched (combinational) feedback loops. The traditional approach to this problem has been to detect such loops by syntactic means. However, these syntactic detection methods are not satisfactory for models that admit multiple correct responses to the same input values, like our specification models. Therefore we take a different approach, and require our mathematical framework to incorporate the ability to express such potentially oscillating loops, and detect them semantically. The means by which we handle such ostensibly zero-delay cycles is one of the contributions of this thesis.

1.2 Formal Hardware Verification and Logic Synthesis

1.2.1 Introduction

The work presented in this thesis falls into two main areas in the domains of hardware design and validation. The first of these areas is formal verification. As mentioned earlier, this is a validation approach distinct from the traditional method of hardware design validation by simulation. The second area that provides context for our results falls within the domain of logic synthesis: the substitution results presented in this thesis are most closely related to the area of logic optimization.

In this section, we present a brief description of how our work fits into the general picture in these two areas. We refer the reader to the surveys of others for summaries of the full field.

1.2.2 Formal hardware verification

Formal verification is the use of mathematical methods to prove that desired properties hold of system models. Most approaches to formal hardware verification fall into one of three general categories: model checking, model comparison methods, and theorem proving.

Our framework supports a model comparison approach to formal verification. In particular, we concentrate on a form of formal verification in which the requirements specification and the implementation behavior are modeled uniformly. As described in Section 1.1, this is a necessary condition for the support of hierarchical verification. Hierarchical verification supports piecewise verification and refinement, which requires that specifications can be used as implementation descriptions in larger descriptions. Therefore specifications and implementations must be the same *kind* of models. Our use of a uniform model comparison method contrasts in particular with model-checking and non-uniform model comparison methods, which we define below.

A general-purpose theorem prover can be used to prove that particular mathematical properties are implied by a logical theory. A logical theory is a set of axioms whose intended semantics is a particular system model or class of models. Alternatively, theorem proving may be used to prove that one object or class of objects in such a logical theory is in a particular mathematical relation to another. For example, Bronstein has used the Boyer-Moore theorem prover to show that certain properties hold of some benchmark circuits [26, 24] and has also used the same theorem prover and logical theory of synchronous circuit behavior to verify that pipelined and non-pipelined versions of the same circuit exhibit effectively equivalent behavior [25, 24]. Both these approaches have also been used with the HOL system and with other theorem provers to verify hardware designs [65, 128, 129].

The use of a general-purpose theorem prover makes this methodology highly expressive. The exact expressiveness depends on the underlying logic of the particular theorem prover, and determines the range of expressiveness of both specifications and system descriptions. However, the verification process itself requires a great deal of human intervention [65, 128]. For this reason it is often referred to as a *mechanized* method, rather than an *automated* one.

Model checking, in contrast, is the use of special-purpose programs to verify that certain properties hold of a particular model [48, 29]. The system model is manipulated explicitly rather than being described axiomatically. The requirements specification is given as a list of *conditions* which the circuit model is verified to meet.

Once these conditions and the circuit model have been formulated, the model-checking verification process is fully *automatic.* A model checker builds or accepts a finite-automaton model of the system and checks whether or not the specified property holds of the model. If it does, the model checker informs the user that the property has been verified to hold of the model. If it does not, the model checker returns a failure trace. This is a sequence of steps allowed by the model that will lead to a situation illustrating a violation of the desired property.

Practical model checking was first introduced with specifications given in a propositional branching time temporal logic and models given as Kripke structures [48, 29] or a particular form of Petri net [118]. Highly efficient *automatic* methods, that take time asymptotically linear in the size of this automaton and in the size of the temporal logic formula being verified, have been developed to check of such a model that a particular property holds. Later developments in symbolic representation of state sets and the transition relation of the automata models have kept model checking in the running as a potentially viable approach to formal verification of real systems. Binary Decision Diagrams (BDDs) are a (potentially) compact and efficiently manipulable representation of Boolean functions [33]. An equivalent canonical representation called Typed Decision Graphs (TDGs) was independently developed [16]. In 1990, several groups independently proposed their use to represent states and transition relations of sequential systems in model checking [99, 17, 53]. The incorporation of this technology into model checkers expanded their capability to the modeling of systems several orders of magnitude larger than had previously been feasible [40, 100].

In the model comparison approach to formal verification, both implementation and specification are system models. A mathematical "correct implementation" relation is defined over pairs of models. This relation holds between two models if and only if the former is considered an adequate implementation of the latter. Usually, the implementation and specification models are described in the same language or formalism, but this need not be the case. An example of a model comparison method in which specifications and implementations are not modeled uniformly is AT&T Bell Lab's COSPAN [87, 88]. The most common use of the model comparison method is equivalence checking, in which the "correct implementation" relation is input-output-behavior equivalence [59, 51, 52, 60]. Equivalence checking is particularly useful *during* the hardware design process, because it can be used to check equivalence between the input- and output-models of each stage of the hardware synthesis process (see next section).

In our framework the implementation and specification to be compared are models of the same kind: ours is a *uniform* model comparison method. Our "correct implementation" relation, however, is more complex than simple equivalence, in order to provide for specifications that allow multiple correct implementations.

An in-depth discussion of the various approaches to hardware verification is beyond the scope of this section. The interested reader may consult [65] for an excellent and comprehensive survey of current approaches to formal hardware verification; a less comprehensive tutorial that contains many explanatory examples of the basic techniques can be found in [98]. Information on theorem proving for formal hardware verification appears in [65, 128, 129, 86].

1.2.3 Logic synthesis

In this section, we provide a short overview of hardware synthesis and describe how our substitution results fit into this area.

Hardware synthesis is the process of refining an abstract model of a circuit until it becomes sufficiently detailed that all the information necessary for its fabrication is present [103]. This process is in practice broken up into several stages: architectural synthesis (also called high-level synthesis or structural synthesis), logic synthesis, and geometrical-level synthesis (also called physical design). Our substitution results are relevant to the logic synthesis stage.

In architectural synthesis, a behavioral description of the circuit is refined into a register-transfer level (RTL) model. In an RTL model, the macroscopic structure of the circuit (functional units, data paths) is clarified and the control unit functionality is described. Architectural synthesis consists of identifying the hardware *resources* that can implement the circuit's operations, *scheduling* the execution of these operations, and *binding* the scheduled operations to the resources, while attempting to minimize the estimated area of the circuit and the time required for the execution of each operation. Further discussion of architectural synthesis can be found in [103, ch. 4-6].

In logic synthesis, an RTL model is transformed into a fully structural representation of the circuit, usually a gate-level netlist (a list of logic gates that includes information on the connections between these gates). Optimization is a significant part of this process: attempts are made to minimize the area of the circuit, as well as propagation delay (in the case of combinational logic), and clock-cycle length and latency (the number of clock cycles required for a circuit operation) in the case of synchronous logic. Our substitution results are relevant to the optimization problem in logic synthesis.

The physical design stage of hardware synthesis generates the layout of a circuit from the netlist created during logic synthesis, by determining the *placement* of the gates and the wiring between them for the full circuit. The output of this stage provides all the information necessary for fabrication of the physical circuit.

For the interested reader, the book [103] provides an overview of the synthesis process and further detail on architectural-level synthesis and logic synthesis for both combinational and synchronous circuits. This book provides additional references for those interested in the physical design stage of hardware synthesis. The book [58] concentrates on combinational logic synthesis alone. This book emphasizes the interrelation between delay, area, and testability of synthesized combinational circuits. The reference [5] concentrates on synchronous logic synthesis.

Our substitution results provide a unifying framework for the optimization problem in the logic synthesis stage of hardware synthesis. They apply equally to combinational and to synchronous logic synthesis.

1.3 Asynchronous Trace Theory

The work presented in this thesis is based on an approach to formal verification of hierarchical *asynchronous* circuits that was presented by Dill [61], called *asynchronous trace theory.* While Dill attacked the problem for asynchronous circuits only, his work incorporated some critical insights that we utilize in a synchronous framework.

Standard gate-level models of hardware make an assumption of "unidirectionality". That is, they assume a unidirectional flow of information through the circuit, or at the very least, a unidirectional electron flow. The use of Boolean functions to model gates, which constitutes a fundamental use of this abstraction, appears as a standard model in elementary electrical engineering texts.

The first of the insights of asynchronous trace theory is that in order to talk about whether or not a given specification is implementable, it is necessary to distinguish between the input and output wires of a circuit. This distinction is about more than just the standard unidirectionality abstraction employed in gate-level modeling. A circuit enforces constraints on the relation between the values on its input wires and the values on its output wires; however, it cannot control the values that appear on its input wires. This point has been ignored in the simple "language inclusion" approach to verification, in which as long as the possible behaviors of a model are a *subset* of those of another model, the first is considered to be a correct implementation of the second [1, 126, 127, 46, 47, 73].

Asynchronous trace theory thus requires that a model account for all possible input values at all times; a model's set of "possible behaviors" must always allow yet another input wire value change to occur. In our synchronous trace theory, this corresponds to admitting all input value combinations on each clock cycle. Thus a model may not *disallow* any input values.

We may wish a specification to guarantee certain behavior of every correct implementation – but only under certain input conditions. If these input constraints are not met, the specification makes no guarantees at all. Asynchronous trace theory allows specifications this expressiveness by distinguishing those behaviors of a circuit which are *possible but undesirable.*

Thus a circuit model incorporates two distinct sets of behaviors: those that are possible behaviors of the circuit being modeled, and a distinguished subset of these possible behaviors, those that follow an undesirable input event. Once an undesirable input event has occurred, anything goes: by definition of the input event's having been undesirable, the model makes no guarantees about what may or may not subsequently happen. Under these circumstances we say that the model has gone into failure mode. The extent of a specification's failure mode is a statement about the environments in which its implementation circuits will function correctly.

Asynchronous trace theory recognizes this aspect of specification in its definition of the mathematical relation between a specification and its correct implementations. First, it assumes that the set of potential environments of a circuit C is simply a set of circuits with complementary inputs and outputs to C *(i.e.,* the input wires of each of these environment circuits are precisely the output wires of C, and the output wires of each environment are precisely the input wires of C). Therefore an environment is just a circuit, and may also have a failure mode: the relation between a circuit and its environment is symmetric. Then trace theory defines a model A to be a correct implementation of another model B if and only if A may be *safely substituted* for B in any environment. A can correctly implement B only if A and B have the same input wires and output wires (and hence the same environments). We say that A correctly implements B if and only if for *every* environment E of A and B, if we may place B in E without the composition of E and B going into failure-mode, then we may place A in E without that composition going into failure-mode. Of course, it may be the case that A can be placed in more environments without activitating its failure-mode than can B; in this case B is not a correct implementation of A, even though A is a correct implementation of B.

We incorporate these insights of asynchronous trace theory into our own model of *synchronous* sequential circuits and their specifications. For its structural view of a circuit model, we also utilize the circuit algebra framework of [61], which is explained in the following chapter.

1.4 Zero-Delay Cycles

1.4.1 Introduction

In order to model synchronous circuits it is necessary to model their behavior during each clock cycle. Thus the modeling of *combinational* circuitry is a necessary step in modeling *synchronous* circuits.

The standard assumption in electronic circuit design is that one does not want to create unlatched combinational feedback loops (except for very specific purposes, such as the creation of a precisely timed oscillator). An exception to this rule was noted by Kautz, who pointed out that use of such loops may be necessary in order to minimize the number of gates in a circuit designed to meet a combinational specification [81]. However, standard logic synthesis systems and timing analyzers are not able to handle combinational loops [130, 92, 131]. For example, a common source of combinational feedback loops in modern designs is resource sharing algorithms run during the high-level synthesis of data and control paths. Even combinational loops that are created through resource sharing that are in fact never exercised (false paths) are considered undesirable, and models have been developed to detect them during high-level synthesis and so avoid their construction [130].

Unfortunately, the property of containing no feedback loops is not automatically preserved when two combinational circuits are wired together. Any formalism which allows the composition of circuit *models* into more complex ones must either apply some mechanism in order to ensure that such loops are not inadvertently created via composition, or must knowingly allow their creation.

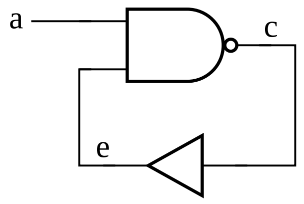

Figure 1.1: The composition of C_1 and C_2

Consider the following simple example. Let C_1 be a nand-gate with input wires labeled **a** and **b** and output wire labeled **c**. Let C_2 be a non-inverting buffer whose input wire is labeled **d** and whose output wire is labeled **e**. If C_1 and C_2 are wired together so that **c** and **d** are tied together, and similarly **e** and **b**, the result is a combinational feedback loop (see Figure 1.1). If M_1 models C_1 and M_2 models C_2 in some formalism that allows the composition of models, then that formalism must either define or disallow the composition of M_1 and M_2 into a model of the composite circuit just described.

We identify four classes of solution employed by existing formal models of combinational and synchronous circuits that include a composition operator.

- Models which ignore the problem, leading to anomalies.
- Moore machine models, in which zero-delay cycles cannot occur.
- Use of functional dependency tracking to disallow the composition of models into combinational feedback loops.
- Ternary simulation methods.

The first approach is not a solution at all, and in fact very few existing formalisms make this mistake. In Section 1.4.2, we illustrate how indiscriminate use of the standard Mealy machine model of synchronous circuit behavior may lead to anomalies. Most formalisms that incorporate a Mealy machine model of synchronous circuit behavior avoid machine compositions that create zero-delay cycles. We present a Mealy-machine based formalism that has neglected to address the issue of zero-delay cycles and which consequently exhibits anomalies.

The second approach constrains circuit models to be Moore machines, which are automata in which the current output is determined solely by the current state (irrespective of the current input value combination). Moore machines can be used to model purely combinational logic. They can also be used to model synchronous circuits in which *every* component is latched. However, it is not clear how a Moore machine could be used to model both latched and unlatched circuitry simultaneously. When used to model purely combinational logic, Moore machines do allow combinational feedback loops to be modeled without the disappearing behavior phenomenon observed in the case of indiscriminate use of Mealy machines. When used to model synchronous circuits, they do not allow the creation of combinational feedback loops, because every component must be latched. In Section 1.4.3, we expand on the brief discussion of Moore machines that appears in Section 1.1. In particular, we explain why this model is inadequate for our modeling needs. We describe the Moore machine model of combinational circuitry and explain how it neatly avoids the zero-delay cycle problem. We also briefly describe some well-known formalisms that have not addressed the issue of zero-delay cycles because they utilize an underlying Moore-machine model of causality between events.

In the third approach, functional dependency tracking methods are employed in an attempt to identify problematic applications of the composition operator. The intent is that such applications be identified and subsequently disallowed. The dependencies in question are situations in which the value on a particular output wire depends on the value on a particular input wire. When this information is maintained accurately, it is called *dynamic* dependency information. A useful approximation to maintaining and computing complete dynamic dependency information is the maintenance of *static* dependency information, usually in the form of a graph. This set of dependencies reflects the topology of the circuit, and maintains that every output of a primitive circuit component is dependent on every input of that component. In the presence of nondeterministic or black box models, only static dependency information may be available. However, it may be overly conservative. In Section 1.4.4, we describe these methods in more detail, and illustrate the problems that arise when they are used in the presence of nondeterministic models.

Our solution falls into the fourth class, that of ternary simulation methods. This method consists of positing a third wire value, in addition to the digital 0 and 1. Such a third value may be used for various purposes; we focus on its relevance for the detection of combinational feedback loops, or zero-delay cycles. Our solution utilizes it merely to detect the presence of such a loop; related research currently in progress attempts to utilize these methods to distinguish problematic (oscillatory) combinational loops from 'harmless' ones. In Section 1.4.5, we describe this and other work using ternary methods, and compare it with our own.

In the following subsections, we describe all four of these approaches in more detail, and categorize many existing formalisms according to them.

1.4.2 Zero-delay models that ignore the problem

A fairly standard approach to the logical, or functional, modeling of combinational logic is to abstract away from propagation delays, effectively assuming all combinational components to be zero-delay.

Mealy-machine models of synchronous hardware behavior fall into this category, as do standard relational and functional models of combinational logic behavior.

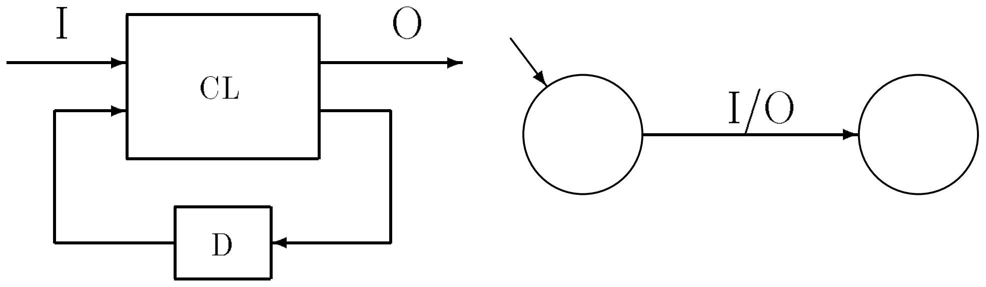

Figure 1.2: Finite-state machine representation of synchronous circuit

Figure 1.2 presents a typical block diagram depiction of a synchronous circuit, and a piece of the corresponding finite-state automaton representation of its behavior. The current output value combination and next state of the circuit are determined by the current state and the current input value combination. Therefore the digital behavior of this circuit can be described by a Mealy machine: an automaton whose transitions are labeled with input-output value combinations. This abstracts away from any propagation delay in the combinational logic that determines the current output values, effectively modeling zero delay between [the presentation of] input and output. Note that we do not assume that Mealy machines must be deterministic in their input value combinations: there may be two outedges from a single state which are labeled with the same input value combination, but distinct output value combinations. This is in contrast to the original Mealy machine definition, which allowed for input don't cares, but did not allow nondeterminism of this sort [101].

In synchronous circuit design, it is assumed that no matter what the cumulative delay in the combinational logic, the clock cycle is made long enough to allow for it. Thus it appears that we can separate the logical (functionality) analysis and the timing analysis of such circuits. A zero-delay view of combinational logic is consistent with this separation: these models are intended for performing logical analysis only.

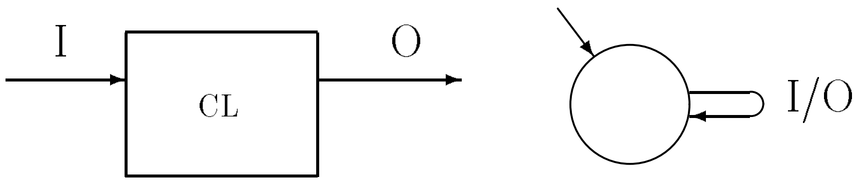

Figure 1.3: Finite-state machine representation of combinational circuit

Of course, a combinational circuit is a special case of a synchronous one: it just happens to exhibit the same behavior during each clock cycle. Its behavior can be described by a Mealy machine of the form shown in Figure 1.3. Equivalently, such a Mealy machine can be represented by the set of its edge-labels: this is a functional or Boolean relation model of purely combinational circuits [44, 22].

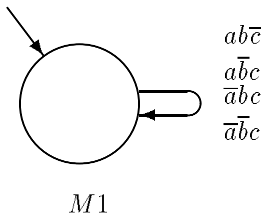

Figure 1.4: Mealy machine representation of C_1

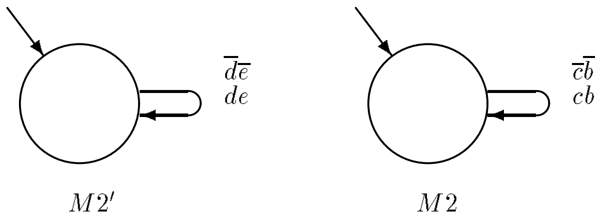

Figure 1.5: Mealy machine representation of C_2

Consider the example described in Section 1.4.1. A simple Mealy machine model of the nand-gate C_1 is M_1 of Figure 1.4. M_1 is a single-state automaton (all of whose transitions are from this single state to itself, and) whose edge-label set is precisely the Boolean function $\mathbf{c} = \overline{(\mathbf{a} \wedge \mathbf{b})}$. Similarly, the non-inverting buffer C_2 is most simply modeled by the Mealy machine M_2' shown in Figure 1.5. We rename the wires in the model of C_2 in order that composition of the models correctly reflect how we have wired together C_1 and C_2 to produce the circuit depicted in Figure 1.1 (page 8). The model M_2 of the relabeled circuit also appears in Figure 1.5.

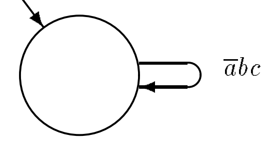

Figure 1.6: Composition of the Mealy machines M_1 and M_2

Standard finite-state-automaton composition of M_1 and M_2 produces the Mealy machine shown in Figure 1.6. Note that it *disallows* any situation in which the input wire **a** is held high, effectively placing demands on the environment of the composite circuit. (Any subsequent composition of this model with any environment model will preclude that environment's holding **a** high). Thus it is an *inadequate* model of the composite circuit, because it precludes some of that circuit's actual behavior. In Chapter 3 we will discuss how and why this phenomenon occurs; basically, this disappearing behavior is a result of oscillation that cannot be described as a single digital value.

Mealy machines appear regularly in the CAD literature as specifications or descriptions of synchronous circuits. Here we are interested in formalisms that allow the *composition* of two Mealy machines to form a model of the composition of the circuits (or requirements specifications) represented by the operand Mealy machines, because this is where the anomalies occur. The problem is exemplified by Lin *et al.*'s model of synchronous circuits as presented in [91]. This model allows the composition of arbitrary models of appropriate input/output type, despite the modeling degradation caused whenever oscillating combinational feedback loops are created.

Lin *et al.*'s model [91] is based on the principles and algebraic structure of asynchronous trace theory [61], and is intended to solve the substitution specification problem which our model correctly handles. However, the authors do not deal with the zero-delay cycle problem.

In order to demonstrate that their model leads to anomalies, we illustrate how it handles the example of Section 1.4.1. Following their definitions, we extend the Mealy machine M_1 of Figure 1.4 by a transition labeled by $(\mathbf{c} = (\mathbf{a} \wedge \mathbf{b}))$ from its already-existing state to an "X"-state whose only out-edge is a self-loop labeled `--/-` to indicate that this state is a sink state (that is, once control is in this state, it will stay there no matter what input-output combinations may occur – and all input/output combinations are allowed). Similarly, we add an "X" sink state to M_2 (of Figure 1.5) and label a transition from the previously existing state to this "X"-state with $(\mathbf{b} = \overline{\mathbf{c}})$. Despite these cosmetic additions, the composition operation of Lin *et al.* specifies that only the traces $(\overline{\mathbf{a}}\mathbf{b}\mathbf{c})^*$ (the set of all sequences of arbitrary finite length which repeat the combinational behavior $\overline{\mathbf{a}}\mathbf{b}\mathbf{c}$ in every clock cycle) are successful behaviors of the composite circuit – which is erroneous. Despite their claim that it does so, Lin *et al.*'s model does not correctly handle combinational feedback loops.

1.4.3 Moore machine models of clocked behavior

When modeled as a Moore machine, a system's output values are completely determined by its current state, irrespective of its current input values [110]. As a model of *synchronous* circuits, therefore, Moore machines allow only latched components, for which the response to the current input values does not occur until the next clock cycle. Moore machines may also model combinational circuits, but in that case they model propagation delays for every component, and therefore do not allow latched components at all. It is not clear how one could use Moore machines to model both latched and unlatched components in the same system. Because we would like to use a single formalism to model both pure combinational logic *and* components that contain latches, we choose to use a Mealy machine model of synchronous circuit behavior.

However, it is the case that Moore machine models of combinational circuits avoid the zero-delay cycle problem and can successfully model combinational feedback loops. This is because they allow

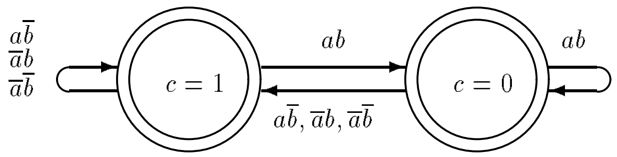
$a\overline{b}$
$\overline{a}b$
$\overline{a}\overline{b}$
$c = 1$
ab
$c = 0$
ab
$a\overline{b}, \overline{a}b, \overline{a}\overline{b}$

Figure 1.7: Moore machine representation of C_1 (nand-gate)

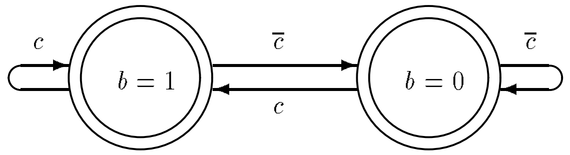

Figure 1.8: Moore machine representation of C_2 (non-inverting buffer)

no zero-delay paths: some propagation delay is incorporated into this model for every component of a combinational circuit. We illustrate how the problem is avoided by examining how Moore machines would be used to model the combinational example of Section 1.4.1. Figure 1.7 contains a Moore machine model of the nand-gate C_1 and Figure 1.8 contains a Moore machine model of the non-inverting buffer C_2. Their composition appears in Figure 1.9. It correctly reflects the fact that the only stable state of the composite circuit of Figure 1.1 is one in which the input wire **a** is held low, in which case the wires **b** and **c** stabilize to the value 1. In addition, however, it correctly reflects the oscillatory behavior of this circuit in the case that **a** is held high. No zero-delay cycle is exhibited here, because the model does not admit any component having a zero propagation delay.

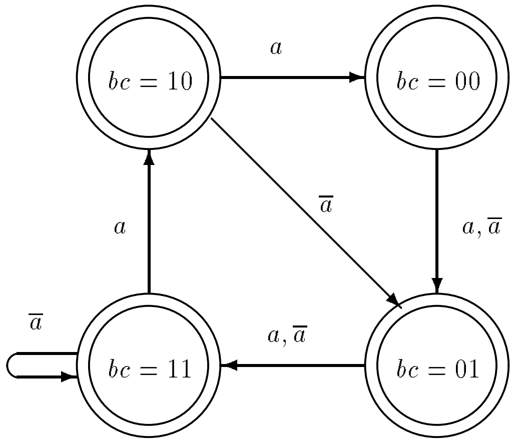

Figure 1.9: Composition of the Moore machine representations of C_1 and C_2

In the remainder of this section, we discuss some well-known synchronous models of communication and verification which use a Moore machine model. Although billed as synchronous formalisms, some of these models are used indiscriminately to model combinational circuits as well, and in that case they model propagation delays. When they model synchronous circuits, they cannot create combinational feedback loops because all components are assumed to be latched. Hence in both cases, they avoid the need to concern themselves with the modeling of zero-delay cycles.

Note that by *synchronous* models we mean clocked models: this is in contrast to an approach to the modeling of communicating processes in which the word 'synchronous' is sometimes used to refer to *synchronization* of communication, usually via common variables (wire or action names). In fact all the formalisms we discuss utilize common or matched-pair names to indicate synchronization or coordination of communication between subprocesses. This is immediate if we consider such a name to denote a specific wire, as we do in our hardware model.

Two classic approaches to the modeling of synchronization in the context of asynchronous processes are Milner's CCS [106, 108] and Hoare's CSP [72]. Both of these process algebras have been extended to *synchronous* versions: CCS was defined in terms of Synchronous CCS (SCCS) by Milner in [107]; CSP is extended to Synchronous CSP (SCSP) by Barnes in [8]. The operational semantics defined for CCS and SCCS posit a branching structure of time, in which the decision to choose one particular course of future action over another is as critical as the possible action sequences themselves. The denotational semantics defined for CSP and SCSP do not take this into account. Nevertheless, the two synchronous formalisms SCCS and SCSP agree on their choice of a fundamental underlying Moore discipline. That is, in both cases the action model chosen was Moore machines. Fundamentally, this is because both SCCS and SCSP assume all synchronous input and output values of a process (values on distinct wires occuring during the same clock cycle) to be independent of each other, except for those action names (or action-coaction pairs) common to multiple processes, which may coordinate communication between them. Thus they cannot support dependency chains of combinational logic that manifest within a single clock cycle.

In SCCS, a process (essentially, its behavior) is represented by an agent. Agents are created from actions and agents using five basic operators. This model certainly allows the expression of processes which do not correspond directly to hardware. However, it may be used to describe hardware, if the actions are appropriately interpreted and their use constrained.

Verification consists of checking that a particular mathematical relation holds between two agents (where one is presumably considered an implementation and the other a specification). In order that this relation be appropriately defined, it is necessary that the recursion operator not introduce any ambiguous agents, in the following sense. An algebraic expression created using only the five operators must (modulo choice in determining which disjunct to activate) indicate unique behavior: recursion may be used only if a unique least fixpoint exists for the resulting expression. For example, this says we may not define an agent to be equal to itself, with no further qualification, because then the agent could be anything, and the equation would still hold. The conditions Milner chooses as sufficient to guarantee the existence of such a unique least fixpoint effectively constrain the allowed agents to behave as Moore machines [107, pp. 285–286].

The underlying philosophy of CCS and SCCS dictates that the set of actions occuring during a single clock cycle, except for those action-coaction pairs that are explicitly listed as restricted, must be independent. In SCCS, we may restrict an agent so as to allow only a certain set of actions — but then *any* subset of this set may occur within each clock cycle [over which the agent is defined]. Similarly, SCSP is based on the assumption that "*events observed simultaneously occur independently;* the performance of one event at a particular time cannot preempt another event at the same time" (emphasis mine) [8, p. 9]. That is, "a process cannot offer its environment the choice between two events without being able to offer both together" [8, p. 9]. This automatically enforces a Moore discipline, in which the performance of events depends only on the current state rather than on any additional events occuring during the same clock cycle.

In SCSP, a process is again represented by an algebraic expression created from other processes and events using five basic operators. As in SCCS, in order to use this system for verification it is necessary to prove that use of the recursion operator does not lead to ill-defined processes that do not correspond to specific behavior options. The proof given (actually, cited) depends on the fact that a process name P only appears in its own definition in a 'guarded' position, that is, *following* the first clock cycle in which the process is active. This restricts every process P to behave as a Moore machine.

Thus both SCCS and SCSP have avoided the need to handle the potential creation of zero-delay cycles via composition, by enforcing a Moore discipline in which such cycles cannot be created. In general, one of the concerns of this family of algebraic frameworks is that delays of all kinds be modeled. Positing that only independent events may be simultaneous, and that the model most explicitly distinguish the occurence times of non-simultaneous events, these synchronous models constrain the set of events that occur during a single clock cycle to contain *only* pairwise-independent events.

CIRCAL is a formalism based on many of the concepts of Milner's CCS, and applicable to the modeling of hardware [104, 105]. In contrast to SCCS and SCSP, it has been implemented in software, as a verification tool called the Circal System [7]. In CIRCAL a hardware component is represented by an agent, which is described by the actions it is expected to perform. Agent comparison via an operational semantics is the basis for verification using this model [104, 109]. CIRCAL differs from CCS primarily in how it handles nondeterminism. It is also sufficiently flexible that a particular model may be synchronous, asynchronous, or both. As in SCCS, only independent events may occur during the same clock cycle, and so CIRCAL also utilizes a Moore machine model for synchronous behavior [104].

Compositional SML (CSML) is an extension to the language SML that allows the modular description of finite-state machines and their composition [50]. SML was developed to support the description of circuit models for model checking. A model written in SML can be automatically translated into a finite-state machine which can then be verified using the EMC model checker

[27, 29, 28]. CSML is an extension to SML that was developed to support *compositional* model checking. In compositional model checking, the user separately specifies individual components (modules) of the full model. During subsequent verification, a smaller "interface process" may be substituted for a module by hiding all wires irrelevant to the property being checked of the full composite model, in order to avoid the state-explosion problem [49]. Thus CSML allows the description of component finite-state machines of the full model and their composition to form this full model. The semantics of CSML support both asynchronous and synchronous hardware descriptions – the latter as Moore machines only.

AT&T's COSPAN verification system utilizes a selection/resolution model of communication between subprocesses [87]. Composition of processes is defined via a selection/resolution paradigm for communication between them. Constraints are placed on the form of the processes to be composed, that suffice to avoid the creation of any zero-delay cycles via composition. Two alternate sets of constraints are available for this purpose. One set effectively constrains the component processes to be Moore machines. The other utilizes the dependency tracking methods that will be described in the following section (Section 1.4.4).

The selection/resolution framework describes the activity that occurs during a single clock cycle. First, each component process *selects* a partial transition label *(i.e.,* values for each of its *output* wires) from among those full transition labels appearing on the outedges of its current state. This restricts its set of candidate next states, but not necessarily to a unique next state. After each component process has selected its transition label, their conjunction (the *current global selection)* is visible to each of the processes, and based on this conjunction each process *resolves* the global selection to determine its own next state (from among those remaining available to it since its own selection) by choosing the target state of one of its outedges whose label is compatible with the current global selection. The global next state (that is, the next state of the composite process) is the combination of all the local states so chosen.

Effectively, the selection/resolution procedure describes the composition of Mealy machines. However, the zero-delay cycle problem is avoided in COSPAN by placing constraints on the form of allowed processes.

Kurshan defines two sets of conditions sufficient to guarantee that at least one global next state always be defined using the selection/resolution definition of composition [87, Chapter 7]. Each of these sets of constraints suffices to disallow the creation of zero-delay cycles in the resulting composite process. The first option is to require that all subprocesses be independent (no joint output wires), lockup-free (all states have at least one outedge) Moore processes (the full outedge label set of every state allows *all* input combinations to occur together with any allowed output combination). An alternative sufficient set of requirements is that all subprocesses be independent and lockup-free and that the *static dependency graph* which encodes all potential functional dependencies between the subprocesses be acyclic. Thus the COSPAN system utilizes both a Moore discipline and a static dependency graph method to ensure that zero-delay cycles will not be created by composition of processes.

1.4.4 Functional-dependency tracking methods

1.4.4.1 Introduction

Another approach to the modeling of zero-delay cycles is to anticipate potentially problematic applications of the composition operator and to subsequently *disallow* those compositions. In this way, the creation of zero-delay cycles can be avoided altogether. In this section we describe various methods that have been employed in order to detect these problematic compositions prior to their actual application. We explore the standard methods used for hardware design, as well as some related applications in the software domain. We may classify all of these approaches as *syntactic* methods for the detection of zero-delay feedback. The method we have in fact employed in the framework described in this thesis allows the *semantic* detection of such loops after they have been created, because we have concluded that none of the methods described in this section are applicable in the presence of the wide variety of properties we wish our models to incorporate. In addition to discussing these syntactic detection mechanisms, we discuss why they are inapplicable in our framework.

CAD tools employ dependency tracking methods to detect combinational feedback paths. Two basic dependency-tracking methods are known for combinational circuits: static dependency graphs and Boolean differences. Boolean differences reflect *dynamic* dependency information: they specify the situations (value combinations on the other input wires) in which the Boolean value on a particular output wire depends only on the Boolean value on a particular input wire. Static dependencies reflect the topology of the circuit. Both of these syntactic tracking methods are overly conservative in the presence of nondeterministic models [41]. In practice, the use of these methods in our framework would lead to unacceptable constraints on the allowed compositions.

In this section we also discuss some related work in the software domain. The *synchrony hypothesis* that forms the basis of the semantics of many synchronous reactive specification languages corresponds precisely to positing a zero-delay reaction in response to an input event. Hence the zero-delay cycle problem arises in this context as well. In Section 1.4.4.3, we describe the algorithms their compilers use to detect – and reject – programs that contain actual “instantaneous” feedback loops. In their own terminology, such programs are *non-causal.*

In practice, many formalisms do not incorporate an explicit mechanism for the detection and hence prevention of combinational feedback loops. Instead, the relevant publications simply state that the definition of a combinational circuit requires it to be acyclic. The mechanism by which this is to be guaranteed is left implicit. In the case of gate-level formalisms, static dependency tracking is easily implemented. In other cases, it is not as clear what mechanism is intended to enforce this constraint, but the formalism in any case is not intended for direct implementation. Much of the logic synthesis work that addresses optimization in multi-level logic networks falls into the first class; the Ruby formalism, for example, falls into the second [79, 123, 80, 78].

1.4.4.2 Dependency tracking for combinational circuits

As stated in the previous subsection, two basic dependency-tracking methods are known for combinational circuits. They are static dependency graphs and Boolean differences. The former is sufficient to ensure that composition of circuits not create combinational feedback loops, but it is overly conservative in the presence of black-box descriptions for which a unique gate-level representation is not available [41]. The latter is generally assumed to be sufficient to ensure that composition of circuits not create combinational loops that exhibit anomalous behavior. In fact, it has recently been pointed out that this is not a reliable assumption [92]. In any case the Boolean differences method is inapplicable in the case of nondeterministic models [41].

In the first of these methods, a static dependency graph is maintained for every circuit model. This graph incorporates the maximal possible set of dependencies between input- and output-wires of every component. The dependency graph of the result of composing two circuits is the union of the dependency graphs of the component circuits. We define the dependency graph of a primitive (black box) component with input wires I and primary output wires O to be the directed graph $\langle V, E\rangle$ whose vertex-set V is the externally-visible wire set of the component ($V = I \cup O$) and whose edge-set E indicates that the value on each output wire is dependent on the values on all the input wires of the component ($E = \{\langle x, y\rangle \mid x \in I, y \in O\}$). The union of two graphs $\langle V_0, E_0\rangle$ and $\langle V_1, E_1\rangle$ is $\langle V_2, E_2\rangle = \langle (V_0 \cup V_1), (E_0 \cup E_1)\rangle$. Note that V_0 and V_1 need not be disjoint.

This dependency graph provides a sufficient condition to determine that composing together two combinational circuits will not lead to a combinational feedback loop: if the union of their dependency graphs contains no cycle, then their composition does not contain a loop. Assuming the primitive circuits for which the dependency graphs were created are gates, this union dependency graph contains a cycle only if the resulting circuit contains a topological loop. However, as shown in [41], if all or some of the 'primitives' for which dependency graphs are created are high-level descriptions of complex circuits, a cycle may occur in the union dependency graph without an actual combinational feedback loop appearing in the corresponding composite circuit. That is, a cycle in this graph may be an artifact of the high-level description of one or both of the components.

This situation is illustrated by the block diagram in Figure 1.10. The bidirectional communication between the high-order 4-bit adder and the carry look-ahead generator (CLG) in this carry look-ahead adder is an example of an "apparent loop" [10] (also called a "pseudo-cycle" [4]). There are no loops or cycles in the circuit when it is described at the gate level, but this information is lost in the block diagram of the circuit. In this case, an acyclic block diagram could be formed by splitting the CLG into two smaller blocks [10]. Thus the cycle in the corresponding static dependency graph is an artifact of the high-level description of the CLG.

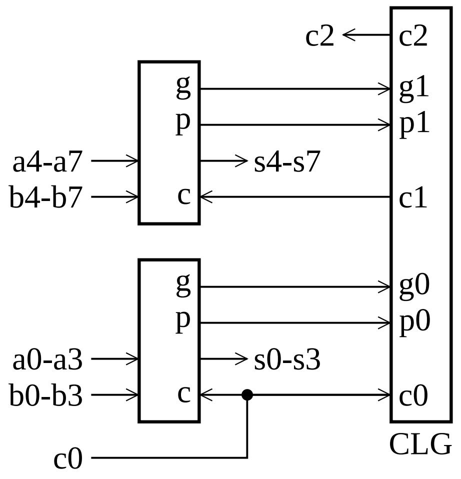

Figure 1.10: Eight-bit carry look-ahead adder

Static dependency tracking is sufficient but not necessary for feedback detection. It detects topological loops, but it cannot distinguish those topological loops that actually constitute feedback situations from those that do not. *Dynamic* dependencies identify input-value combinations for which a gate ignores one of its (other) inputs in computing its output value. For example, if one input to an *and*-gate is held low, then the output of the gate holds value 0 no matter what the value on its other input wire. Similarly, if one input to an *or*-gate is held high, then the output of the gate holds value 1 no matter what the value on its other input wire. Dynamic dependency information of this sort can be propagated from gates to full circuits.

It has generally been assumed that this information can be used to distinguish between combinational feedback loops and *false path* loops. A false path is a path through a circuit that is never exercised. An example of such a path is the loop through the logic blocks F and G in the circuit illustrated in Figure 1.11. The circuit contains a topological loop, but it computes the feedback-free function z = if c then $G(F(x))$ else if not(c) then $F(G(x))$. If c always holds one of the Boolean values 0 or 1, then the topological loop is never exercised.

Boolean differences formalize dynamic dependencies [121, 96, 45]. They were originally used in the hardware design community to derive information for fault testing. They are also used in the computation of observability don't cares (see Section 1.5.2). In acyclic combinational circuits, the Boolean difference provides insight into the conditional functional dependency of a component's output value on any particular input x.

The Boolean difference is defined as $\partial F/\partial x = F_x \oplus F_{\overline{x}}$, where the component in question computes the function F ($z = F(y_0, y_1, \ldots, y_m, x, y_{m+1}, \ldots, y_n)$) and $\oplus$ denotes exclusive-or. F_x and $F_{\overline{x}}$ are parts of the Shannon decomposition $F = x \cdot F_x + \overline{x} \cdot F_{\overline{x}}$, and are defined as follows. If $z = F(y_0, y_1, \ldots, y_m, x, y_{m+1}, \ldots, y_n)$ then

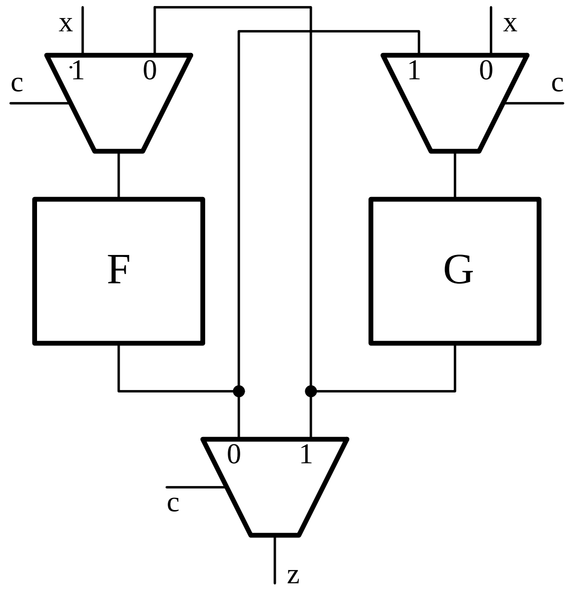

Figure 1.11: False path combinational loop (Figure 2 of [92])

- $F_x = F(y_0, y_1, \ldots, y_m, 1, y_{m+1}, \ldots, y_n)$ and
- $F_{\overline{x}} = F(y_0, y_1, \ldots, y_m, 0, y_{m+1}, \ldots, y_n)$

The Boolean difference $\partial z/\partial x$ for a circuit with input wire x and output wire z is a Boolean expression describing the Boolean value combinations on the other inputs of the circuit such that in their presence, the Boolean value on z varies with the Boolean value on x. That is, for a circuit all of whose wires under all circumstances stabilize to steady 0 or steady 1, the Boolean difference $\partial z/\partial x$ describes the conditions under which z is functionally dependent on x. For such a circuit, $\partial z/\partial x = 0$ if and only if there are no circumstances in which the value on z depends on the Boolean value on x. If $\partial z/\partial x = 0$ for an acyclic combinational circuit, then the zero-delay path between x and z is a false path.

Unfortunately, Malik has noted that the Boolean difference may incorrectly indicate a lack of dependence between the values on two wires when in fact subsequent composition forms a loop that may oscillate [92]. In the case he identifies, the Boolean difference incorrectly indicates a lack of dependence between the values on two wires ($\partial z/\partial x = 0$), when in fact the value on one (z) is dependent on whether or not the other (x) is oscillating. Because the possibility of oscillation is not taken into account by the Boolean difference, which represents only *Boolean* behaviors *(i.e.,* stabilization to steady 0 or steady 1), this kind of dependency cannot be represented nor detected by use of Boolean differences. Thus Boolean differences cannot be used to detect combinational feedback, because such detection requires that transient non-Boolean states be taken into account.

In addition, the use of Boolean differences is inappropriate in the presence of *nondeterministic* specifications. A nondeterministic component may well lead to an overly conservative dependency set, relative to the dynamic dependency sets corresponding to each of its potential implementations [41]. Consider the specification of a circuit with two output wires and one input wire, as shown

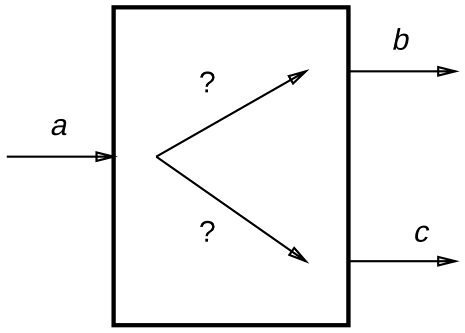

Figure 1.12: Specification: $(b = a \wedge c = 0) \vee (c = a \wedge b = 0)$

in Figure 1.12. This specification may be correctly implemented by the two distinct functions $(b = a$ and $c = 0)$ and $(c = a$ and $b = 0)$. In the former, $\partial b/\partial a = 1$ and $\partial c/\partial a = 0$, and in the latter, $\partial b/\partial a = 0$ and $\partial c/\partial a = 1$. To prevent zero-delay cycles, we would be forced to assume that both b and c depend on a, even though this can never occur. That is, we must conclude for this specification that $\partial b/\partial a = 1$ and $\partial c/\partial a = 1$. Maximizing the dependency set associated with a specification in this manner will rapidly lead to a situation in which all compositions involving the specification are disallowed. Hence we consider the method of Boolean differences to be inapplicable in the case of such nondeterministic specifications.

In much of the logic optimization work for multi-level logic networks and for synchronous logic networks (see Section 1.5), the mechanism by which acyclicity is to be preserved is not made explicit. In the case of gate-level formalisms, static dependency tracking is easily implemented. In his work on deriving all the degrees of freedom available for the correct implementation of one finite-state machine (FSM) in a set of interacting FSMs (see Section 1.5.3), Watanabe utilizes Boolean differences to determine whether or not a particular deterministic FSM M_1' can be implemented so that it does not form a combinational feedback loop when composed with the other FSMs [136]. Malik's results imply this is not reliable [92].

Because we require our framework to allow the expression of nondeterministic models (so that we may describe specifications that allow for multiple correct implementations) and of high-level black-box circuit models and specifications (in order to support hierarchical verification), we cannot use either of these two syntactic methods for tracking "dependencies" to detect and disallow zero-delay cycles.

1.4.4.3 Detection of zero-delay feedback loops in software

Synchronous languages [11, 66] are a class of executable specification languages designed to support the rigorous specification of synchronous reactive systems [114, 70]. Their semantics are all based on the *synchrony hypothesis,* which states that all subprocesses of a system produce their outputs and internal signals simultaneously with the arrival of the input signals to which they are responding [11, 66]. Hence the execution of a program divides time up into "instants." During an instant, every parallel subprocess proceeds as far as possible in its program in response to the signals broadcast by all the others and by the environment *(i.e.,* in response to its input signals). An instant terminates when all processes can only await further environmental interaction. Each such instant is assumed to take no time with respect to the external environment [15]. Thus the synchrony hypothesis is similar to the zero-delay assumption underlying the Mealy machine approach to modeling synchronous circuits.

Under the synchrony hypothesis, the communication mechanism between subprocesses is the instantaneous broadcasting of signals, so that "all processes share the same vision of their environment and of each other"[15, p. 91]. The assumption of instantaneity of response applies to each subprocess. In all of these synchronous languages, the presence or absence of a signal in an "instant" may be detected. Therefore an instantaneous reaction chain between subprocesses may lead to *inconsistent* information about the presence or absence of various signals. (That is, the same signal may be required to be both present and absent at the same time). It can also happen that multiple sets of output signals are consistent with the reactions of the individual subprocesses. If for every set of input signals from the environment the resulting reaction chains between subprocesses can only lead to a *unique* and consistent set of signals being broadcast during that instant, and this holds for every possible sequence of instants allowed by a program, then the program is called *causal.* (Of course, a non-causal program contains a zero-delay cycle). Restrictions on the allowed constructs of this form are enforced by the compilers for the respective languages; these restrictions suffice to disallow all non-causal programs. Because the problem is undecidable in general, they may disallow some causal programs as well. In the remainder of this section we describe these synchronous programming languages and discuss the mechanisms their compilers employ to detect – and reject – those programs that are non-causal.

Esterel [15, 18, 14] is an imperative synchronous programming language. It features various control structures, one of which is a parallel operator for creating subprocesses, and three ways to stop a subprocess: traps, interrupts, and natural termination. As in all synchronous languages, communication between subprocesses is achieved via the broadcasting of signals: a signal is broadcast by the subprocess which "emits" it, and is instantaneously perceived to be "present" by *all* subprocesses. In Esterel, however, a signal may have a value in addition to simply being "present" or absent. In fact, the same signal may be emitted multiple times in the same "instant" and need not have the same value every time. In this case, an operator for combining the various values the signal is given (emitted with) is determined at declaration of the signal, and subsequent reads of the signal will read this combined value. The execution semantics enforced by the V3 compiler guarantee that no intermediate value of a signal can be read.

ESTEREL has a behavioral semantics, and requires that its progams all be deterministic (that is, that a program respond to each sequence of sets of input signals with a *unique* sequence of sets of output signals). The behavioral semantics do not provide a full operational definition of how to find this unique fixpoint response. In addition, they do not enforce the determinism constraint: there exist ESTEREL programs for which the behavioral semantics allow multiple sequences of sets of output signals in response to the same sequence of sets of input signals. In order to enforce the determinism constraint, and to provide an operational semantics, an *execution* semantics has been defined for the language [15]. These semantics enforce determinism by treating each signal as a shared-memory address, with initial value "undefined", and by imposing a discipline in which a signal cannot be read if it can still be written in the current "instant". Thus, no intermediate values of a signal can be read, and "one may not conclude that a signal is absent as long as its emission by some process remains possible" [66, Chapter 5]. If all subprocesses block because each one is waiting to read or test for the presence of a signal that may yet be written (emitted) in the current instant by another subprocess, the program is declared *non-causal* and consequently disallowed. Unfortunately, the execution semantics are overly conservative, and there are cases in which they disallow programs that are in fact causal.

LUSTRE[43, 112, 67, 42] and SIGNAL[12, 64, 13] are declarative synchronous programming languages based on a data-flow model [66]. In these languages, a program is a set of recurrence equations that describes a set of *flows.* A flow is a sequence that denotes the behavior of a particular signal at each tick of a clock *(i.e.,* its value when it is present). Multiple clocks may be in effect, some of them defined in terms of each other. Each flow X defines a clock C_X that denotes those instants at which the signal is present. The variables in a set of recurrence equations represent flows; the operators include arithmetic operators and comparison relations that are applied pointwise (per relevant clock tick) to the flows, and language-specific operators that allow one flow to be defined in terms of another by defining a new clock or applying a time shift, etc. LUSTRE and SIGNAL vary in the specifics of their operators and the definitions they allow. In LUSTRE, each variable may only appear on the left-hand side of a single equation, and it is completely defined by its declaration and this equation. (This is referred to as the *definition principle).* Thus LUSTRE is a functional language, in which every operator defines a function from input sequences to output sequences. In SIGNAL, the equations impose a set of constraints on the relative speeds of the various clocks. In particular, "the way an output flow is used may constrain the input flows of the operator that produces it" [66, Section 4.3]. In both cases, the compiler enforces the constraint that a unique output sequence be defined as the response to each input sequence. (Every program must be deterministic). In addition, the compilers must check that the clock definitions and constraints contained in a program are not inconsistent.

They must ensure, for example, that there exists at least one clock that is an infinite sequence of instants at which arbitrary input signals may be received. The clock constraints for each signal that are specified by a program are encoded in the *clock calculus* of the relevant language. LUSTRE allows the programmer less freedom in manipulating clocks than does SIGNAL. In particular, in LUSTRE no information may be inferred from the way a variable (a signal and its associated clock) is used. Thus a simpler clock calculus suffices for these checks in LUSTRE than in SIGNAL.

In order to enforce determinism within an instant, the LUSTRE compiler employs a single-pass static dependency check [43, 67]. Thus it is very conservative, and disallows as non-causal even those feedback loops which are false paths. Data dependency checking and clock consistency checking are pursued independently. Clock consistency is a simple syntactic check that strictly enforces LUSTRE's *definition principle* via rewriting.

The SIGNAL compiler employs a conditional-dependency check for data dependencies. Because it allows the clock of a variable to be inferred from the use of that variable, checking for data dependencies and checking for clock consistency are interrelated [12, 64]. Analysis of each program builds both a conditional dependency graph and an encoding in clock calculus equations of the constraints the program places on the relation between the clocks of the signals in the program. This procedure is overly conservative, and may cause rejection of programs that are in fact causal. For example, SIGNAL cannot reason about non-Boolean data types, and so it may reject a program as placing constraints on its inputs when in fact the constraints partition the space of possible values. On the other hand, it may allow programs that are intuitively non-reactive, or non-causal, in that their outputs may drive the frequency of their inputs [66, Section 4.3][12].

We briefly mention a formalism called STATECHARTS that belongs to the class of languages inspired by the ideas of [70] and the synchrony hypothesis, but which does not utilize syntactic methods to disallow problematic applications of the composition operator. A simpler language based on the ideas of STATECHARTS, called ARGOS, does successfully utilize an exact syntactic check in order to disallow compositions that lead to problematic zero-delay cycles.

STATECHARTS [69] is a graphical specification language whose intended semantics obey the synchrony hypothesis. However, instead of syntactically disallowing those compositions that may create problematic "instantaneous" reaction chains, the intent is that compilers for this visual language implement some operational semantics for such cases [71, 76, 116], and force a run-time error whenever such a chain is determined to lead to a contradiction [116]. Earlier versions of the semantics allowed inconsistencies in the set of signals produced during the instant [71, 75, 76] rather than causing a run-time error in the case of the operational semantics leading to an inconsistency. (Recall that an inconsistency is a situation in which some signal s must be both present and absent in the same instant). Thus those semantics are clearly unacceptable for our framework. The later semantics do not allow nondeterminism, and we therefore find them unacceptable as well.

ARGOS [94, 95] is a graphical formalism based on STATECHARTS that has simpler graphical syntax and hence simpler semantics. As in STATECHARTS, extensions to the standard graphical notation for Mealy machines allow the expression of implicit composition (without the attendant explosion in size of the graphical representation) and of *hierarchy.* In order to determine whether or not a particular composition is allowed, the compiler for this language executes an exhaustive search for feedback loops in "instantaneous" chains of reaction [66, Section 5.2]. This syntactic check is exact, and allows all and only the applications of the composition operator that do not lead to potentially nondeterministic or inconsistent behavior. However, it is not applicable to our framework because it disallows nondeterminism.

The syntactic methods that the compilers for these synchronous specification languages employ in order to detect and hence reject non-causal programs are overly conservative for our framework. We require that nondeterministic models be supported even as zero-delay cycles are detected, and none of the syntactic techniques we have just described allow that. Therefore we cannot utilize any of these syntactic methods. In the following section, we describe a *semantic* approach to the zero-delay cycle problem.

1.4.5 Ternary simulation methods

1.4.5.1 Introduction

In this section we investigate the application of ternary simulation techniques to model circuit behavior in the presence of zero-delay cycles. Ternary simulation is a hardware simulation methodology in which the circuit model is assumed to recognize three distinct wire values: the usual digital **0** and **1**, and a third value commonly called **X**. This third value has historically been used in various ways. Depending on the particular context, **X** may denote confusion on input wires and multiple possible correct responses on output wires [23]. Recently, its use on input wires to denote multiple potential input values has been advocated as well [37, 36]. In order that the appearance of **X** on an output wire provide meaningful feedback from a simulation run, certain mathematical conditions must hold, which constrain how the **X** value may be used by the circuit model that is being simulated. These constraints require the components of the model to be functions that are monotonic in a particular partial order over the three-value domain of wire values.

It turns out that use of such a third value in a manner consistent with its use in ternary simulation (restriction to monotonic functions) suffices to model the behavior and presence of combinational feedback loops. If a fully-specified combinational circuit produces **X** on an output wire in response to a Boolean input value combination (that is, one that does not contain any **X**'s) then there must be a loop in the circuit. Thus ternary simulation may be applied to *test* for the presence of a loop. If we define the behavioral model of a logic gate to be a monotonic extension of the corresponding Boolean function, then the composition of models reflects the wiring together of the logic gates, and the resulting composite behavioral model correctly reflects the information we could derive via ternary simulation. In other words, it provides a *semantics* for such loops.

Our own work uses a third wire value in addition to digital **0** and **1**, to denote oscillation and stabilization to an intermediate voltage. This idea follows the tradition of ternary simulation to some extent. However, we are the first to use a third value to model zero-delay cycles in nondeterministic models. In addition, we have identified a condition that is weaker than functional monotonicity and yet suffices to guarantee the soundness of our formal verification results using these models.

Recently some researchers have addressed the problem of analyzing circuits that contain combinational feedback loops which are "harmless" while still detecting and disallowing those that may cause anomalies [68, 92, 124]. Some of them are using ternary simulation strategies in their classification procedures [92, 124].

Their work seeks to clarify distinctions that we have not found to be relevant for our purposes. Our use of a third wire value guarantees the *marking* of every combinational feedback loop, so that the presence of such a loop may be detected from its behavior. Subsequently the user or designer may choose whether to keep the flagged loop or to remove it from the design. We do not seek to make judgements as to which loops are harmful and which not, but rather to provide the user with the tools to make these decisions for him or herself. In particular we are concerned that the behavior and presence of loops be made explicit in order to ensure the accuracy of our formal verification methods. Our use of a third wire value suffices to eliminate the disappearing behavior observed for example in the formalism of [91], which may lead to invalid results in formal verification.

In this section we describe the traditional uses of a third wire value, and contrast them with our own. We also examine the recent work classifying combinational feedback loops into those that are considered harmful and those that are not, and informally describe those situations in which our formalism does implicitly conclude that a combinational feedback loop is "not harmful."

1.4.5.2 Traditional ternary simulation

In order to contrast our use of a third wire value to denote oscillation or intermediate voltage with the ways in which this value has been used in the past, we first provide a short history and description of ternary simulation.

Gate-level ternary simulation was introduced in the 60's as a technique for detecting hazards in combinational logic, and races and oscillations in asynchronous circuits [62, 77]. In [62], Eichelberger defined the following ternary simulation algorithm, that was used to approximate the circuit's response to a new input value combination, starting at a stable state for a known previous input value combination. First, every input wire whose value changes from **0** to **1** or from **1** to **0** in the transition from the previous input value combination to the new one, is set to the third value **X** (which he called $\frac{1}{2}$). The circuit is then simulated using the *standard ternary extension* to each gate's Boolean function representation. As many passes as necessary are made until no internal wires change value; the input wires are held constant through all these passes. (Jephson, McQuarrie and Vogelsberg extend this algorithm to correctly handle explicit delay elements [77]). At stabilization, some internal wires may hold the value **X**, and others digital value **0** or **1**. Then the new Boolean input value combination is applied to this intermediate stable state, and again the circuit is simulated repeatedly until quiescence.

Any remaining **X** values indicate the possible presence of a critical race, hazard, or oscillation; modulo the accuracy of the Boolean function model of the gates, any resulting **0** and **1** values are guaranteed to be the response of the corresponding circuit of gates for any combination of logic gate propagation delays [62]. It was soon noticed in the hardware community that this technique may return an overly pessimistic response in some cases, as certain combinations of propagation delays are highly unlikely. Subsequently this technique was extended to allow the initial steady state to contain **X** values on wires as an indication of digital but unknown voltage [23].

Current ternary simulation algorithms initialize all internal nodes of the circuit to value **X**. They then compute the steady-state response function of the circuit. The steady-state response on a wire is the value (**0**, **1** or **X**) that wire would attain if the input value combination to the circuit were held fixed long enough for the wires to stabilize. For this computation, the behavior of each logic gate is represented by the standard ternary extension of the corresponding Boolean function. For the purposes of this computation, the behavior of a circuit composed of Boolean logic gates is adequately represented by the composition of the standard ternary extensions of the Boolean function representations of those gates.

In [30, 31], Bryant suggested extending these methods to the analysis of MOS circuits. He developed a mathematical switch-level model which is most recently implemented in the COSMOS tool [31, 35, 34, 38]. Other work on related switch-level models is referenced in [39, Section 4]. The basic idea implemented in COSMOS is that various attributes of MOS transistors (such as relative conductance and capacitance of the transistors in the circuit) can be encoded into Boolean equations to be evaluated during simulation, just as the standard ternary extensions to the Boolean function representations of the component gates of a circuit are evaluated in gate-level ternary simulation. During simulation of MOS circuits using this Boolean equation representation, the value **X** on a node denotes "an indeterminate voltage between low and high indicating an uninitialized network state or an error condition caused by a short circuit or charge sharing" [35, p. 637].

Later Bryant extended these ideas to the formal verification of circuits [32, 36, 37]. He proposed that such circuits can be formally verified via ternary simulation. More precisely, he noted that if one can find a sufficient set of assertions on a finite number of bounded-length input-output behaviors of a circuit, or on a finite number of bounded-length sequences that incorporate its input-output behaviors and the values on its latches, such that a circuit that satisfies these assertions must correctly implement a specification, then one can formally verify the circuit by applying ternary simulation methods *(i.e.,* running it through a ternary simulator) to verify that the circuit does indeed satisfy these assertions. In this work, he allows the behavior of circuit components to be represented by arbitrary monotonic extensions of Boolean functions, rather than requiring that the *standard* ternary extension be used. The property of monotonicity of these functions ($x \leq y \Rightarrow f(x) \leq f(y)$) over the *information ordering* $X < 0$ and $X < 1$ on $\{0, 1, X\}$ is necessary for the soundness of the verification results. He introduces the use of the third value **X** to represent multiple input sequences at once, a technique which he calls *input weakening* [32, 37, 36]. Because the value **X** on a wire constitutes less information than if it were to hold the value **0** or **1**, an input vector containing **X** as the value of a particular wire allows the simulation of both possibilities at once. If an input vector v containing **X**'s leads to an output vector containing **0**'s and **1**'s on those wires we are interested in, then we have effectively performed the simulation of multiple input vectors at once to derive the same result. This trick may significantly enhance the efficiency of the method.

Thus, the use of a third wire value to represent non-digital behavior is not new. It has been used in various ways to represent value confusion, ill-defined behavior and oscillation in (deterministic) circuit models. Our particular use of this mechanism to represent stabilization to an intermediate value (a particular form of ill-defined behavior) or oscillation is only unique in that we are using it for a new purpose, the modeling of zero-delay cycles in the presence of nondeterminism. In addition, we have adapted the functional monotonicity constraint (used to guarantee the soundness of ternary simulation) to a partial-monotonicity condition that is more suitable for nondeterministic models. Our new condition suffices to guarantee the soundness of our verification results.

1.4.5.3 Detection and classification of combinational feedback loops

As stated earlier, use of a third wire value in a particular consistent manner leads to the flagging of all combinational feedback loops in a behavioral circuit model. It turns out that identifying the presence of such loops suffices for our purpose, which is the development of sound, composable models for formal verification.

In [24], Bronstein has proposed a third value, which he calls "?" to denote the unknown (undefined) value. Within his logical theory of synchronous circuit behavior, he proves that if a circuit contains no unlatched loops, then the circuit responds to all Boolean input vector sequences with Boolean values on *all* its wires, including those that are not designated as visible primary output wires. This is precisely the same claim we make for our own framework, stated contrapositively. Namely, we claim that if some Boolean input vector sequence (to a deterministic circuit model) leads to an output sequence that contains the third wire value, we may conclude that the circuit contains a combinational feedback loop.

Some other researchers have gone in a slightly different direction: their work addresses the problem of analyzing circuits that contain combinational feedback loops which are "harmless" while still detecting and disallowing those that may cause anomalies [68, 92, 124]. Some of them use ternary simulation strategies in their classification procedures [92, 124].

Kautz [81], Cerny and Marin [44], and Malik [92] have all identified the same criterion for circuits that despite containing unlatched feedback loops, nevertheless exhibit acceptable combinational behavior. Their criterion for this category of harmless loops is the following: a circuit is "combinational" if and only if it produces a unique output value combination in response to each input value combination, and the output value combination produced does *not* depend on the circuit's history. Acyclic circuits containing only combinational components are clearly "combinational". The question that Malik [92] addresses is: which circuits that do contain unlatched feedback loops are nevertheless "combinational"? He proposes a general solution to this problem that uses ternary simulation techniques. Halbwachs and Maraninchi address the same question, but without taking propagation delays into account, and with history encoded as local variables [68]. They do not employ ternary value techniques for their solution. Shiple *et al.* [124] have extended the question to circuits that contain latches, in an attempt to develop a less conservative algorithm for zero-delay cycle detection in ESTEREL. They extend Malik's techniques to this case.

We do not intend our models to address this question. However, it is the case that combinational feedback loops among non-primary output wires of a circuit that do not adversely affect the behavior on the primary output wires of that circuit do become invisible in our framework. Thus although we do not seek to make explicit the distinction between anomalous and irrelevant combinational feedback loops, some such distinctions do result automatically from our use of the circuit algebra framework (which enables us to "hide" non-primary outputs in order to indicate that they do not participate in the observable input-output behavior of the circuit).

1.4.6 Summary

In order to support a *hierarchical* approach to formal hardware verification, we require the ability to model combinational feedback loops that may inadvertently be created via model composition. The work described in this thesis successfully applies ternary simulation modeling techniques to correctly model the functional behavior of combinational feedback loops. A feature of our model is its ability to *express* the presence of such cycles, without determining any specific course of action for eliminating them from a hardware design. In addition, our framework supports the expression of nondeterministic models, which the available syntactic methods for combinational feedback loop detection, for example, do not.

In this section, we contrasted our approach with others based primarily on how they handle – or do not handle – the modeling of combinational feedback loops. Various approaches have been proposed to model the creation of unlatched feedback loops via circuit composition. We have examined formalisms that handle the problem gracefully as well as some that do not, and have compared our own approach to all of them.

We have explained why we advocate an approach that models the behavior of these loops as zero-delay cycles, despite the attendant modeling problems. Moore machine models of circuit behavior can easily express combinational feedback loops. However, they do not easily allow the modeling of clocked hardware *and* purely combinational circuitry within the same formalism.

We have discussed syntactic approaches to avoiding the zero-delay cycle problem as well as approaches closely related to our own semantic approach. Static dependency tracking is a well-known syntactic method for avoiding the creation of unlatched feedback loops that is sound but overly conservative. In compilation for programming languages that support zero-delay communication between subprocesses, control flow information may be utilized in a sound fashion to determine a topologically sorted order for signal emissions or value determination. However, this too is overly conservative in the programs it allows.

Attempts are currently being made to utilize methods from ternary simulation to detect feedback less conservatively. Our method is a conservative use of ternary simulation that allows the *semantic* detection of topological (structural) loops; it does not address the question of distinguishing harmless ("combinational", or "causal") loops from those exhibiting anomalous behavior. Of course, in many cases we can in fact identify loops that are quite clearly harmless.

1.5 Related Substitution Work

1.5.1 Introduction

The substitution results presented in this thesis are most closely related to the area of logic optimization, which falls in the general area of logic synthesis. Our main substitution result is the derivation of a closed-form expression that precisely specifies all and only the allowed substitute circuitry in a given location in a given circuit or partially structurally-specified model. Logic optimization is the study of how a partially-structured representation of a circuit may be rearranged so as to optimize it in terms of some metric such as area or delay. It traditionally focuses on local transformations of the current representation of the hardware design, a problem to which our results provide the complete solution space.

In this section, we discuss past and current work in logic optimization, from the perspective of our own substitution results. We focus primarily on the extent to which the various formalisms identify degrees of freedom available for optimization.

Our work is particularly relevant to the area of logic optimization in multi-level logic networks, and to global optimization approaches for sequential logic optimization. The area of redesign, or resynthesis, is also a relevant application: many of the same techniques used for multi-level logic optimization have been applied to logic redesign. We describe related work in these areas, and compare it to our own.

1.5.2 Combinational logic synthesis and optimization

In combinational logic optimization, a distinction is made between two-level logic minimization and multi-level minimization. In two-level minimization, the objective is to minimize the area required for a two-level (AND-OR, NOR-NOR, or other two-level form) representation of a given set of logic functions. The correspondence of various easily-measured metrics to the area of the final circuit is well known, and the methods for two-level minimization are well understood. However, better results may often be obtained via a multi-level approach, which can exploit further degrees of freedom in implementing multiple logic functions. In multi-level optimization, a single- or multiple-output logic function may be implemented by a circuit of arbitrary finite depth. In practice the depth is bounded by the objective of optimizing with respect to propagation delay as well as area. Our substitution approach identifies *all* the degrees of freedom available for implementation, and hence applies to the more general multi-level approach to the optimization of combinational logic.

The standard strategy for multi-level logic synthesis and optimization divides the process into two stages. The first stage is independent of the precise circuit components and technology to be used in the final implementation. It transforms an abstract model of the logic block, called a *logic network,* into a final form that more closely reflects the structure of the desired implementation. The second stage is called *technology mapping* or *library binding.* It *binds* each subsection of the final logic network to an actual circuit component available in a given technology-dependent database, according to the intended functionality of that section of the network. We concentrate on the technology-independent first stage of this process.

Techniques proposed for the technology-independent stage of multi-level logic optimization include logic transformations, algebraic approaches, and Boolean don't-care (DC) methods [20, 58, 103]. Our substitution results generalize don't-care methods.

The fundamental idea behind don't-care (DC) techniques for combinational logic optimization is to exploit the degrees of freedom available in the implementation of a logic function f. For $\mathcal{B}$ the domain of Boolean values $\{0, 1\}$, an incompletely specified Boolean function $f : \mathcal{B}^n \rightharpoonup \mathcal{B}$ may be described by three sets: its on-set f^{on} (these are the input value combinations that f maps to 1), its off-set f^{off} (the input value combinations that f maps to 0), and its don't-care (DC) set f^{dc} (the input value combinations for which the value of f is not specified). If a function g is completely specified then $g^{dc} = \emptyset$. An incompletely specified logic function f may be correctly implemented by any completely specified function g such that $f^{on} \subseteq g^{on} \subseteq f^{on} \cup f^{dc}$ (where $\subseteq$ denotes the usual set inclusion). Different choices among the allowed Boolean functions g lead to different costs (area and delay) in actual circuit implementation. Combinational synthesis and optimization seek to take advantage of this by choosing optimal or near-optimal g according to the relevant cost metrics.

Don't-care (DC) information may be provided with the specification for a logic block or it may be *derived* from the multi-level representation of a partially optimized version of that logic block. The former class of DCs are called *external* DCs; the latter fall into two distinct subclasses, called *satisfiability* don't cares and *observability* don't cares. In order to explain these distinctions, it is necessary to define a logic network.

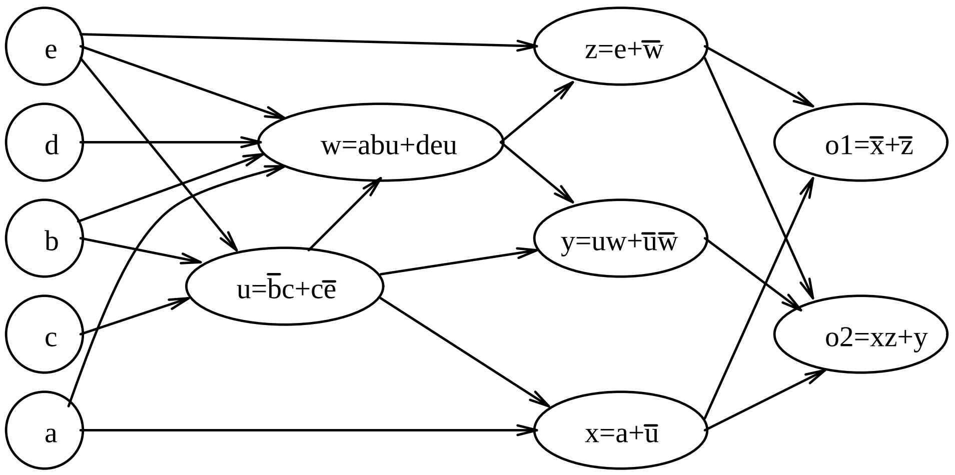
e
d
b
c
a
w=abu+deu
u=$\overline{b}$c+c$\overline{e}$
z=e+$\overline{w}$
y=uw+$\overline{u}\overline{w}$
x=a+$\overline{u}$
o1=$\overline{x}$+$\overline{z}$
o2=xz+y

Figure 1.13: A combinational logic network (Figure 2.1 of [55])

A logic network [19, 9, 20] is a directed, acyclic graph (DAG) whose vertices (also called nodes) are annotated with single-output Boolean functions and whose edges represent the wires that must be present to correctly connect the components that implement these functions. The source nodes of the network represent the primary input wires of the circuit being synthesized and the sink nodes represent the circuit's primary output wires. Each internal node represents an intermediate function in the computation of one or more of the Boolean functions computed by the circuit and appearing on its output wires. An edge appears between two internal nodes if the function computed by one is required input for the other. Thus a logic network is a hybrid structural/behavioral representation of a circuit. In the process of logic optimization the structure is expected to change and the function for each new node to become less complex or closer to those component functionalities available in the library.

An internal node represents a single-output Boolean function, which is ostensibly completely specified. However, its location in the logic network determines some don't-cares in its actual implementation [9, 20, 55]. The don't cares which may be identified for a single node fall into two classes. The first is the class of *satisfiability* don't cares (SDCs). SDCs reflect the correlations we may assume between distinct inputs to this node based on the functions (associated with other nodes) that compute the values on these input wires. The second class are the *observability* don't cares (ODCs) of the node. They are the result of propagating external don't care information backwards through the network from its primary outputs. Even if there are no external don't cares specified for the network, the backward propagation computation itself accumulates information about the circumstances (on the other wires) under which the output of a particular node has no effect and hence is a don't care.

The full don't-care set of a node captures all the degrees of freedom available for its implementation assuming the other nodes remain stable. Techniques have also been developed for computing *compatible* don't-care sets for a set of nodes, that identify some degrees of freedom in the implementation of each that do not affect the identified degrees of freedom available for implementation of any of the other nodes in the set [111, 57, 120]. However, even such sets of compatible don't cares cannot express correlation of outputs. It may be the case that in the presence of a particular input-value combination to the logic network, a pair of nodes in a network can both output 0 or both output 1, and in either case the network will produce the correct output values. Don't cares cannot reflect this information, because they cannot allow these two combinations (00 and 11) as the outputs of the two nodes without allowing 01 and 10 as well.

Observability relations, or Boolean relations, improve on don't care sets by allowing the expression of correlation of outputs [44, 22]. A Boolean relation specification of a subcircuit or a set of nodes is a set of input-output value combinations that describes precisely which output value combinations may occur in response to each input value combination. It is considered to be correctly implemented by any completely specified (multi-output) Boolean function that is *compatible* with it, that is, that when considered itself as a set of input-output value combinations is seen to be a subset of the Boolean relation. The use of Boolean relations to describe the degrees of freedom available for implementation of a subcircuit within a combinational circuit has been explored by [44, 22, 120, 134]. Boolean relations (like DCs) are defined for logic networks, which by definition can represent only acyclic combinational circuits. Our substitution results provide a specification of *all* the degrees of freedom available in implementing an internal node or set of nodes. Thus we address the same problem as do DC-sets and Boolean relations. However, our substitution results are more general than Boolean relations (which are in turn more general than DCs), because we solve the problem in a more general context than logic networks. Our circuit semantics allow the expression of combinational circuits that contain (unlatched) feedback loops, and therefore among the degrees of freedom we identify as available for the correct implementation of a logic network node or set of nodes is the possibility of its containing feedback.

We mention one additional approach to the expression of the degrees of freedom available for the implementation of a node or set of nodes in a logic network. This approach is called Boolean unification, and it has been applied by Fujita *et al.* to both multi-level logic optimization and logic redesign [63, 84]. It provides a methodology for deriving the most general specification of a node or set of nodes in a combinational logic circuit, based on E-unification [83, 6] for E the equational theory of Boolean rings [97]. This method is not applicable to combinational circuits that contain feedback, because its soundness depends on the Boolean *exclusive-or* and *and* operators forming a Boolean ring. In the presence of combinational feedback, the Boolean ring axioms do not hold for these operators. Therefore (as for Boolean-relation based methods) our substitution results extend this method because they describe all degrees of freedom in a more general context.

As just indicated, another relevant application of the above techniques is in the area of *logic redesign.* Redesign is the problem of rectifying incorrect designs to meet a given specification [135, 85]. This problem may arise in various ways. In some cases a design error has been found which we hope can be corrected by modifying a restricted part of the existing design (resynthesis). In other cases, the existing hardware design was correct but the specification has subsequently been modified (engineering changes). The former case obviously allows the application of logic optimization techniques, while the applicability of these techniques to the latter may require some explanation.

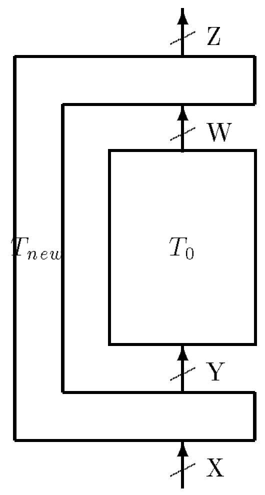

Figure 1.14: Rectification of an existing circuit T_0 by the addition of modification circuitry T_{new}

In the case of an existing hardware design and a rectified specification, we may attempt to modify the existing hardware by adding external circuitry to the design. If possible, the desired correction is achieved by the addition of such modification circuitry solely on the input wires of the existing circuit or solely on its outputs. As described above, Boolean unification techniques are applicable only to these cases [63, 84]. However, in order to achieve the desired correction to the functionality of the new composite circuit, it may sometimes be necessary to add modification circuitry to both the inputs and the outputs of the original circuit, as illustrated in Figure 1.14. This case has been addressed in [135]. In principle, the problem of determining all the degrees of freedom available for the correct implementation of T_{new} in Figure 1.14 is precisely the problem of deriving a Boolean relation or observability relation for a subcircuit of a circuit or for a set of nodes in a logic network, as addressed by [44, 22, 120, 134]. However, their work assumes an acyclic structure for the resulting composite circuit, as reflected in the acyclicity requirement for a logic network. Therefore, in order to avoid the inadvertent creation of combinational feedback loops by composition of the existing circuitry T_0 and the newly created modification circuitry T_{new}, Watanabe *et al.* take a different approach to the derivation of correct T_{new}. They present a conservative relational specification for T_{new} which does not identify all the degrees of freedom available for its correct implementation (not even of those T_{new} that would *not* lead to the creation of a combinational feedback loop), but which allows the later addition of latches to modify the composite circuit somewhat in order to break any combinational feedback loops [135]. (Details are provided in Example 4.9 in Chapter 3). In our own approach to this redesign problem, the creation of combinational feedback loops is not a concern. Hence our substitution results, which address the same problem as [44, 22, 120, 134], may be applied directly to this redesign problem without modification in order to correctly identify all the degrees of freedom available for the correct implementation of T_{new}.

1.5.3 Sequential logic synthesis and optimization

In performing logic synthesis for synchronous circuits, the standard strategy is to first determine a binary encoding for each state of the circuit [5]. This is called *state encoding.* One may then pursue further optimization and refinement of the resulting synchronous logic network [103]. Subsequently, technology mapping is applied in a manner similar to the combinational case. In some cases, optimization may be interleaved with state encoding (reencoding), to the extent that state encoding effectively incorporates the technology mapping step [103].

State encoding transforms a RTL model into a *synchronous logic network* (SLN). We describe this abstract circuit model for the case in which the circuit is clocked by a single-phase clock (that is, all rising clock edges are modeled abstractly as equivalent clock ticks) and all state is stored in edge-triggered synchronous delay elements (edge-triggered D-flipflops, also called registers) [102]. Synchronous logic networks are similar to the logic networks described in the previous section, except that they are intended to model synchronous circuits rather than purely combinational circuitry. Every vertex is associated with a single-output Boolean (combinational) function, as in the case of a logic network. The synchronous delay elements are modeled as positive weights on the edges of the SLN. An edge that corresponds to a wire connecting two combinational logic blocks with no intervening delay element has weight zero. The underlying graph need not be acyclic; however, every cycle in the SLN *must* have a positive total edge weight. Therefore this model does not allow *unlatched* feedback loops. In addition, because a vertex of a SLN may be associated with a Boolean function that depends on the function output value of another vertex at different instances of time, a SLN may contain multiple edges between the same two vertices, each labeled with a distinct weight.

Various methods may be employed for state encoding [5]. We concentrate on those approaches that utilize don't-care techniques, as those are most closely related to our own approach.

During state encoding, the RTL model may be decomposed into a set of interacting finite-state machines whose states are encoded symbolically (rather than being fully determined at the bit level). Don't-care methods may then be applied in an attempt to minimize both the number of bits necessary for binary state encoding and the next-state logic necessary to implement the transition relations of these component machines [5, 21]. Our substitution results provide a methodology for deriving all the degrees of freedom available for the correct implementation of an arbitrary subcircuit. Because these don't-care methods address a similar problem for the case of a component finite-state machine, we will briefly discuss these methods and contrast their expressiveness with our own.

For the case of *cascaded* finite-state machines (FSMs), in which each FSM drives the next in series, the behavior of each component machine may determine *input don't-care sequences* for the machine that it drives [82, 119]. The derivation of input don't-care sequences has been extended to the case in which feedback is allowed between the two machines [132]. Similarly, researchers have addressed the problem of deriving the *output don't-care sequences* for the driving machine from examination of the driven machine [119, 133]. This problem is much more difficult, and a formalism even more expressive than Boolean relations is needed to fully describe the derived degrees of freedom available for correct implementation, even in this simple cascade configuration [133, 21]. A sufficiently expressive formalism has been identified in [122, 21] where it is called *multiple Boolean relations* (MBRs). This formalism also suffices to describe the output don't-care sequences when feedback is allowed between the two machines [133].

These methods do not combine the computation of input don't-care sequences and output don't-care sequences in the presence of feedback between machines, and therefore they do not compute all the degrees of freedom available for correct implementation. It is not known just how much information is lost, because it is not known how to iterate these methods to correctly derive all possible degrees of freedom. Methods for deriving an automaton specification that incorporates all the degrees of freedom available for the correct implementation of one FSM in a set of interacting FSMs have been identified by [136, 54]. However, in both of these formalisms the combination of the other interacting FSMs must be presented as a single *deterministic* FSM – that is, one that allows only a single input-value combination in response to each output-value combination from each state. In contrast, our approach allows the derivation of an automaton specification even when the other interacting FSMs do not form a single deterministic machine.

Multiple Boolean relations, or MBRs, were introduced in [122], where they were defined to be sets of maximal Boolean relations. This formalism allows the expression of two Boolean relations without implying that their union also describes an allowed solution. For example, if one Boolean relation part of an MBR allows the output value combinations 010 and 101 in response to the input value 0, and the output value combination 111 in response to input 1, and another allows 000 in response to 0 and 110 and 011 in response to 1, that does *not* imply that an implementation that produces 101 in response to 0 and 110 in response to 1 is a correct, allowed implementation. Such a constraint cannot be expressed by a single Boolean relation. In [21] it was noted that MBRs can be expressed as Mealy machines that allow multiple output-value combinations in response to the same input-value combination from a given state. We call such machines *nondeterministic* Mealy machines. Every nondeterministic Mealy machine describes an MBR. However, multiple distinct *minimal* nondeterministic Mealy machines can express the same MBR. Therefore MBRs are not as expressive as nondeterministic Mealy machines, which are the formalism we employ to identify all the degrees of freedom available for the correct implementation of a subcircuit.

Various methods may be applied to a synchronous logic network in order to refine it into an optimized form to which technology mapping can be applied. One may apply combinational logic optimization methods to the combinational blocks of logic that do not cross register boundaries. A method which leads to better optimization results is to alternate such combinational optimization with *retiming* [90], in which registers are moved across blocks of logic in order to minimize their number or to shorten the minimum length of the clock cycle (by minimizing the propagation delay through any contiguous combinational blocks) [90, 102, 93]. Our substitution results are most closely related to the more expressive approach to optimization and refinement of a synchronous logic network which we describe below.

An approach to the optimization of a synchronous logic network that can express better results than what can be achieved by retiming and combinational optimization alone is described in [56, 57, 55]. This approach considers the *traces* (input-output sequences) that constitute the behavior of a synchronous logic network and utilizes don't care techniques to identify degrees of freedom available for the implementation of an arbitrary subnetwork. *Synchronous recurrence equations* describe the behavior of a synchronous logic network in equational form, using variables that are wire names indexed by relative time offsets to indicate the value on a particular wire during a particular clock cycle. Don't-care information can be derived from these equations.

In the case of a synchronous logic network that is acyclic, all degrees of freedom available for the correct implementation of an arbitrary subnetwork can be derived by this method. MBRs are required to express these degrees of freedom [55, Section 5.4]. In the case of a synchronous logic network that contains feedback, the network is partitioned into an acyclic part and a set of feedback connections. Synchronous recurrence equations that express input don't cares and output don't cares of the acyclic part of the network are derived iteratively. It is not known whether or not in all cases the particular partition chosen is irrelevant, and hence it is not known how much of the total don't-care information is derived by this method.

We mention one other recent addition to the literature on degrees of freedom in implementing synchronous logic. This work differs from the rest by not assuming that the initial state of the representation is known and reachable. This corresponds to not assuming full reset capability for the hardware. Singhal and Pixley *et al.* have proposed a method for optimizing the implementation for a traditional (deterministic) Mealy machine by replacing the original Mealy machine with one having fewer states, when both the original and the replacement may power up in any state [113]. They present a formal definition of a "safe replaceability" relation $\preceq$ between two Mealy machines that takes into account the ability of both machines to power up in any state [125]. They clarify that the $\preceq$ relation implies safe replaceability with respect to *all* environments: if $D_1 \preceq D_0$, then D_1

may safely replace D_0 in *any* environment [113, p. 445]. Therefore, no don't-care information may be extracted from the environment of the intended circuit. Instead, their definitions identify degrees of freedom available for the implementation of the states themselves: if D_0 may be safely replaced by a Mealy machine with an order of magnitude fewer states, then fewer registers are required to implement these states.

1.6 Contributions of the thesis

We have developed a mathematical model of synchronous sequential circuits that supports both automated formal hierarchical verification and substitution. In order to facilitate hierarchical verification, we model synchronous circuit specifications and implementations uniformly. Each of these descriptions provides both a behavioral and a structural view of the circuit or specification being modeled. We define primitive behavioral models and operations on them that correspond to wiring together circuits and to hiding non-primary output wires in order to explicitly state that the values on these wires do not participate in the observable input-output behavior of the circuit or specification. We prove that our models and these operations form a *circuit algebra* [61]. In order to correctly model the behavior of a synchronous circuit during a single clock cycle, we adopt a modification of traditional Mealy machines as the basis for our behavioral model of synchronous circuits. In addition, our model supports nondeterministic specifications, which capture the minimum requirements of a circuit without forcing us to overspecify by including irrelevant implementation details. All models are finitely representable and all operations are effective.

For formal verification, our framework provides a means for comparison of the behavior of a circuit model to a requirements specification in order to determine whether the circuit is an acceptable implementation of the specification. One model correctly implements another in this framework if it may be safely substituted for it in any environment. The resulting behavior comparison relation forms a preorder over the class of models with the same input- and output-wires. In order to determine whether this relation holds in any given case, we have developed algorithms for *canonicalizing* a model to enable its automatic comparison to a candidate implementation. This extends the framework of asynchronous trace theory to the case of synchronous circuits.

For substitution, and to support a modular verification process, our framework provides a structural view of a circuit and the capability to plug in one component in place of another in a circuit model. This allows us to determine whether the new component constitutes an acceptable substitution in terms of the desired behavior of the full circuit. In fact, we have derived a closed-form expression for the most general specification of the allowed substitutions for a component in a circuit, against which candidate components may be compared via the behavior comparison algorithms developed for formal verification. Our solution to this problem provides a more *general* solution than has previously been available for existing problems in the areas of logic optimization and rectification. We have developed automatic procedures for the computation of this most general specification and for comparing it to candidate substitute components.

We have developed a similar, fully self-contained model of combinational circuits and their specifications. All of the results for the synchronous case have been derived for the combinational model as well, including fully automatic decision procedures for determining when one model correctly implements another (considered as a specification), and all the substitution results. The combinational framework can be embedded in the synchronous one, because a combinational behavioral model may be considered to be a synchronous model of a particular form: a synchronous model of a combinational circuit or specification simply repeats its combinational behavior during each clock cycle.

Hierarchical descriptions of combinational circuits may often contain apparent loops. This is because the considerations involved in decomposing a circuit design into its component blocks are primarily *functional* considerations rather than structural ones. Therefore apparent loops are ubiquitous in a block diagram. In the presence of black-box behavioral descriptions for which structural information is not available, apparent loops and actual combinational feedback loops in a hardware design are not readily distinguishable. Previous existing formalisms have relied on syntactic methods for distinguishing them. However, these methods are not satisfactory for nondeterministic models. Our model of the behavior of a synchronous circuit within a single clock cycle correctly handles such cyclic dependencies even in the presence of nondeterminism, by providing a semantic method to describe them. As a result, we can also correctly model those actual combinational feedback loops that may be inadvertently created via circuit model composition.

The semantic method we use to describe combinational feedback is the addition of a third wire value to denote non-Boolean behavior. Traditionally, in order to ensure that such an addition leads to sound results, a circuit has been represented by a monotonic function. This approach is not directly applicable to nondeterministic models. Instead, we have identified a new constraint on the use of the third wire value, that is sufficient to ensure the soundness of our verification and substitution results.

In the safe substitution framework, the asymptotic complexity of determining whether or not one circuit model correctly implements another is dominated by the asymptotic complexity of determining of another, derived model whether or not this constraint holds of it. Our choice of this new constraint leads to a conformance check of lesser asymptotic complexity than would be the case if we were to extend the monotonicity condition to nondeterministic models in various other (obvious) ways.

In addition to developing a theoretical framework to support behavioral and structural comparison of synchronous circuit models at various levels of detail, we have developed and implemented automatic decision procedures for both formal verification and substitution using these models.

1.7 Overview of the thesis

In this chapter we have provided an overview of work related to our own. Hopefully, we have also succeeded in communicating an overview of what we have achieved and its significance.

In the following chapter, we explain the circuit algebra framework on which our model is based. In Chapters 3 and 4, we develop a fully self-contained model of *combinational* circuits and their specifications, and develop the theoretical underpinnings of the automated verification procedures and substitution results for these models. In Chapter 5, we develop the full model of *synchronous* circuits and their specifications and develop the theoretical underpinnings of the automated verification procedures and substitution results for these models. They are an extension of the combinational circuit models described in Chapters 3 and 4: the development and many of the proofs are essentially identical to those in the previous two chapters. Chapter 6 describes the algorithms used in our software implementation of the automatic formal verification procedures for models of synchronous circuits. Combinational circuit models are a special case of these circuits, with only trivial syntactic extensions, and can be handled by the same verification tool. Finally, Chapter 7 contains our conclusions.

Finally, in an attempt to clearly delineate that portion of the work presented in this thesis which is my own, I must clearly attribute the contributions of others. The use of a third wire value to model feedback effects was suggested by Jerry Burch, and further developed in joint work among myself, Jerry Burch and David Dill. The current definitions for the required constraints on its use arose in the course of this joint work. However, the canonicalization definitions, theorems, proofs and algorithms, and the software implementation and its design, are my own.

Chapter 2

Circuit Algebra and Other Math

2.1 Introduction

Circuit algebra was developed in [61] in order to formalize the notion of circuit *structure* in a mathematical model of circuits. It identifies the properties of circuits that should be reflected in a formal system for hierarchically describing their structure. Circuit algebra defines a set of structural operators that may be applied to circuit models, and places constraints on how these operators may interact with each other. The basic idea is that any reasonable definition of circuit models, and of what these operators do to them, must identify different ways of building the same structure. The rules of circuit algebra formalize mathematically when two distinct circuit algebra expressions denote the same circuit. When circuit models are behavioral descriptions, as in our case, the circuit algebra framework guarantees that every circuit (irrespective of the order of its construction) corresponds to a unique behavioral model, and that every circuit structure that may be built using the circuit algebra operators and primitive or previously constructed behavioral models is a legitimate behavioral circuit model. That is, different behaviors are not assigned to the same circuit structure, and every circuit may be structurally constructed from its primitive behavioral components as a circuit algebra expression.

In this chapter, we list the definitions of circuit algebra and summarize its results. For a full exposition of circuit algebra see [61, Chapter 2]. The *rules* of circuit algebra axiomatize the framework; when they hold of a model of circuits and its operators, then the results of circuit algebra follow for that model. When we present our formal models for combinational and synchronous circuits, we will prove that they obey the circuit algebra rules and thus form a circuit algebra.

In this chapter we also present mathematical notation and definitions that we will use in the remainder of the thesis.

2.2 Mathematical Preliminaries

2.2.1 Introduction

In this section we present some of the notational conventions that are used in the remainder of this and subsequent chapters. Basic notation for functions and sets is used in the presentation of circuit algebras in the following section. Relations are used in describing our behavioral models of combinational and synchronous circuits. Sequences may be used to denote behavior over time, and are therefore relevant for the modeling of sequential circuits. Therefore we are interested in mathematical notation and background relating to finite sequences and regular languages. In this section we present this notation and background, in addition to defining our notational conventions for the presentation of functions, sets, relations and general structures.

2.2.2 Functions, sets and relations

We use the standard notation $\subseteq$ for set inclusion: $C \subseteq A$ if every element of the set C is also an element of the set A. The sentence $C \subset A$ means that C is a proper subset of A, *i.e.,* $C \subseteq A$ and there exists at least one element of A that is not in C. The number of elements in A is $|A|$.

We say two sets are *disjoint* if they have no element in common: Y and Z are disjoint if $(Y \cap Z) = \emptyset$. The cross-product of two sets is the following set of pairs: $H \times G = \{\langle h, g\rangle \mid h \in H \text{ and } g \in G\}$. We may sometimes refer to $(H \times H)$ as H^2. Note that the order of the elements of a pair matters: $\langle h, h'\rangle = \langle h_1, h'_1\rangle$ if and only if $h = h_1$ and $h' = h'_1$.

We write $f : A \longrightarrow B$ to indicate that f is a total function from domain A to codomain B. A total function assigns an element of its codomain to *every* element of its domain: $\forall a \in A.f(a)$ is defined. If f is a partial function that is not total we write $f : A \rightharpoonup B$. The set of all total functions from domain A to codomain B may be written $[A \longrightarrow B]$. Equivalently, we may write it as B^A. If $A = \emptyset$, then B^A contains a single element, which we call the *empty function.* The identity function over domain A is denoted by $\mathbf{1}_A$. Functions may be composed: if $f_2 : A \longrightarrow B$ and $f_1 : C \longrightarrow D$ such that $f_2(A) \subseteq C$, then the composition of f_1 and f_2 is the function $(f_1 \circ f_2)$, which is defined by $\forall a \in A.(f_1 \circ f_2)(a) = f_1(f_2(a))$. Finally, two functions are equal if they map identical elements to identical elements: $(f_1 : A \longrightarrow B) = (f_2 : A \longrightarrow C)$ if and only if $\forall a \in A.f_1(a) = f_2(a)$.

For any $C \subseteq A$, $f(C) = \{f(a) \mid a \in C\}$. Use of this notation is introduced by saying that "f is extended naturally to sets." Later we will discuss the natural extension of a function to other types of objects as well. It follows directly from this definition that if $C_1 \subseteq A$ and $C_2 \subseteq A$, then $f(C_1) \cup f(C_2) = f(C_1 \cup C_2)$.

For any $C \subseteq A$ and $f : A \longrightarrow B$, we write $f_{|C}$ to denote the *restriction* of f to C. This function is defined as follows: $f_{|C} : C \longrightarrow B$ and $\forall c \in C.f_{|C}(c) = f(c)$. We use the notation $f : x \longmapsto y$ to indicate that $f(x) = y$. If we are only concerned with the effect of f on a small number of elements, we may denote f by its effect on those elements, *e.g.,* as $[a \longmapsto b, c \longmapsto d]$.

A function is *injective* if it assigns a distinct value from the codomain to each element in its domain. Formally, $f : A \longrightarrow B$ is injective if and only if $\forall x, y \in A.[f(x) = f(y) \Longrightarrow x = y]$. We may also state that a function is injective *over* C, for $C \subseteq A$, and in that case we mean that $\forall x, y \in C.[f(x) = f(y) \Longrightarrow x = y]$. (Or in other words, $f_{|C}$ is injective).

Given a set B and two disjoint sets Y and Z, and two functions $f_1 \in B^Y$ and $f_2 \in B^Z$, we define $(f_1 \cup f_2) \in B^{(Y \cup Z)}$ to be the function f such that for every $c \in (Y \cup Z)$,

$$f(c) = \begin{cases} f_1(c) & \text{if } c \in Y \\ f_2(c) & \text{if } c \in Z \end{cases}$$

Clearly this function union operation is commutative ($f_1 \cup f_2 = f_2 \cup f_1$) and associative ($(f_1 \cup f_2) \cup f_3 = f_1 \cup (f_2 \cup f_3)$).

A relation is a way of pairing together elements of two sets: a relation R over two sets A and B is a set of pairs $R \subseteq (A \times B)$. We may sometimes write aRb to denote $\langle a, b\rangle \in R$. If A and B are the same set, that is, $R \subseteq A^2$, then we may say that R is a relation over A.

An equivalence relation over a set D is a relation that is transitive, reflexive, and symmetric. Transitivity is the property that $\forall a, b, c \in D.[aRb \text{ and } bRc \Longrightarrow aRc]$. Reflexivity is the property that $\forall a \in D.aRa$. Symmetry is the property that $\forall a, b \in D.[aRb \Longrightarrow bRa]$. An equivalence relation partitions its underlying set into a set of disjoint classes, the union of which constitutes the full original set. For $d \in D$, we denote by $[d]_{R_0}$ the equivalence class of d induced by the equivalence relation R_0.

We say a relation $R \subseteq D^2$ is a *preorder* if it is transitive and reflexive. We say that a relation $R \subseteq D^2$ is a *partial order* if it is transitive, reflexive, and anti-symmetric. Anti-symmetry is the property that $\forall a, b \in D.[aRb \text{ and } bRa \Longrightarrow a = b]$. Every preorder over a set D induces a partition of D into equivalence classes, and a partial order over the set of these equivalence classes. The preorder $R \subseteq D^2$ induces an equivalence relation R_0 over D, defined as follows: $\forall a, b \in D.[aR_0b \Longleftrightarrow (aRb \text{ and } bRa)]$. According to this definition, whenever aR_0b and aRc then bRc as well. R_0 in turn induces a partial order R' over $\{[d]_{R_0} \mid d \in D\}$, which is defined as follows: $[a]_{R_0}R'[b]_{R_0}$ if and only if $\forall a \in [a]_{R_0}, b \in [b]_{R_0}.aRb$. By properties of preorders and equivalence relations, this occurs if and only if $\exists a \in [a]_{R_0}, b \in [b]_{R_0}.aRb$. The relation R' is a partial order.

For any set Y, the *pointwise extension* of any partial order R over a set D to a partial order R_Y over $[Y \longrightarrow D]$ is defined as follows: $\forall f, g : Y \longrightarrow D.[fR_Yg \Longleftrightarrow \forall d \in D.\langle f(d), g(d)\rangle \in R]$.

For a partial order R over D, and a subset $C \subseteq D$, we define the *least upper bound* of C in D, written $lub(C)$, to be $d \in D$ such that $\forall c \in C.cRd$ and $\forall d' \in D.[[\forall c \in C.cRd'] \Longrightarrow dRd']$. There need not exist a least upper bound in all cases, but if there is one then it is unique. If $lub(C) \in C$, we may omit mention of D.

A monotonic function $f : A \longrightarrow B$ with respect to particular partial orders $R_A \subseteq A^2$ and

$R_B \subseteq B^2$ is a function from A to B that has the following *monotonicity* property:

$$\forall a, a' \in A.[aR_Aa' \Longrightarrow f(a)R_Bf(a')]$$

Following [61], we use the notational convention that subscripts and superscripts are inherited by the components of a composite description. If structures of a particular kind are defined to contain fields named A, B, and C, then the structure named H_0 will have fields named A_0, B_0 and C_0, and the structure named H' will have fields named A', B' and C', etc.

2.2.3 Finite sequences and regular languages

ω is the natural numbers. A finite sequence w of length $n \in \omega$ on some set C is a function $\{i \in \omega \mid 0 < i \leq n\} \longrightarrow C$. Given a sequence w, we write $w[i]$ to denote the element in the i'th position of w. (That is, $w[i]$ is $w(i)$). We denote by $len(w)$ the length of w. We define $w[0] = \varepsilon$, the empty sequence, for all sequences w. ε has length 0. If C does not contain sequences, the set of all finite-length sequences over C is written C^*. The set of all sequences of length n over C is written C^n. Thus $C^* = \bigcup_{n\in\omega} C^n$.

The concatenation of two sequences w and z is the sequence written $w \cdot z$. It is defined as follows. If $len(w) = n$ and $len(z) = m$, then $y = (w \cdot z)$ if and only if for every $i \in \omega$ such that $0 < i \leq n$, $y[i] = w[i]$, and for every $j \in \omega$ such that $n < j \leq (n+m)$, $y[j] = z[j-n]$. The same notation is used if either of w or z is an element of C : in that case we do not distinguish it from an element of C^1. Concatenation may be extended naturally from sequences to sets of sequences: for $Y, W \subseteq C^*$, $Y \cdot W = \{(y \cdot w) \mid y \in Y \text{ and } w \in W\}$. Concatenation is associative: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$. Therefore we may omit the parentheses and write simply $x \cdot y \cdot z$. The natural extension of a function $f : C \longrightarrow C'$ to sequences on C is defined recursively: $f(\varepsilon) = \varepsilon$, and for $c \in C$ and $w \in C^*$, $f(c \cdot w) = f(c) \cdot f(w)$. It is clear from the definition that any function defined as such an extension distributes over sequence concatenation: $f(w \cdot z) = f(w) \cdot f(z)$. The *pointwise extension* of any partial order R over a set D to a partial order R' over D^* is defined as follows: $\forall y, z \in D^*.[yR'z \Longleftrightarrow [len(y) = len(z) \textit{ and } \forall i \in \omega.[0 < i \leq len(y) \Longrightarrow \langle y[i], z[i]\rangle \in R]]]$.

We define notation to describe the prefixes of a sequence. A sequence $w \in C^*$ is a *prefix* of a sequence $y \in C^*$ if and only if there exists a sequence $z \in C^*$ such that $y = w \cdot z$. Thus $pref(y) = \{w \in C^* \mid \exists z \in C^*.y = w \cdot z\}$ is the set of all prefixes of y. The $pref$ function is extended to sets as follows: for $Y \subseteq C^*$, $pref(Y) = \bigcup_{y\in Y} pref(y)$. We say a set Y is *prefix-closed* if and only if $pref(Y) \subseteq Y$.

We define notation to describe the extensions of a sequence in a given set of sequences. A sequence $w \in C^*$ is an extension of a sequence $y \in C^*$ in a set $W \subseteq C^*$ if and only if $(y \cdot w) \in W$. Formally, for $y \in C^*$ and $W \subseteq C^*$, $ext(y, W) = \{w \in C^* \mid (y \cdot w) \in W\}$. More specific notation allows us to specify the length of the extensions in which we are interested. For each $n \in \omega$,

$ext_n(y, W) = \{w \in C^n \mid (y \cdot w) \in W)\}$. The *ext* and ext_n functions are extended to sets as follows: for $Y, W \subseteq C^*$, $ext(Y, W) = \bigcup_{y \in Y} ext(y, W)$ and $ext_n(Y, W) = \bigcup_{y \in Y} ext_n(y, W)$.

Finally, we present our notation for regular languages and finite-state automata. Most of the definitions and results that we describe can be found in any introductory automata theory book, for example [74]. We are especially interested in Mealy machines [101], which are a particular kind of finite-state automaton. We define our own variant of Mealy machines, which we use in our synchronous circuit models.

A *finite-state automaton* M is a five-tuple $\langle \Sigma, Q, Q_0, Q_F, \delta \rangle$, where Σ is a finite set of symbols, Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $Q_F \subseteq Q$ is a set of final states and $\delta \subseteq (Q \times \Sigma \times Q)$ is a transition relation. M may also be called a *finite-state machine (FSM).* M is a *deterministic* FSM if and only if δ in fact forms a function $\delta : (Q \times \Sigma) \longrightarrow Q$ and Q_0 contains only one element $q_0 \in Q$. If M is not a deterministic FSM we say it is a *nondeterministic* FSM.

Each finite-state automaton denotes a particular set of sequences on Σ, as follows. A *run* of an FSM M on a sequence $w \in \Sigma^*$ is a sequence ρ on Q such that $\rho[1] \in Q_0$ and for every $n \in \omega$ such that $0 < n < len(\rho)$, $\langle \rho[n], w[n], \rho[n+1] \rangle \in \delta$. If there exists a run ρ on $w \in \Sigma^*$, we may write $\rho[1] \overset{M,w}{\Longrightarrow} \rho[len(\rho)]$, or simply $\rho[1] \overset{w}{\Rightarrow} \rho[len(\rho)]$. In addition, if there exists a run ρ of M on some $w \in \Sigma^*$, we say that $\rho[len(\rho)] \in Q$ is *reachable.* We say the run is *accepting* if $\rho[len(\rho)] \in Q_F$. If there exists an accepting run on $w \in \Sigma^*$, we say that M accepts w. Note that if M is a deterministic FSM, then the definition of a run ρ on w becomes $\rho[1] = q_0$ and $\forall n \in \omega$ such that $0 < n < len(w)$, $\delta(\rho[n], w[n]) = \rho[n+1]$. It follows that a deterministic FSM M accepts w if and only if there is a *unique* accepting run on w. We define the *language* $\mathcal{L}(M)$ of a FSM M to be the set of all sequences accepted by M. There exists an effective procedure for transforming any nondeterministic FSM M into a deterministic FSM M' such that $\mathcal{L}(M') = \mathcal{L}(M)$.

It turns out that there exists an alternative characterization of the expressiveness of FSMs. This characterization is in terms of *regular languages.* A *regular language* or *regular set* $L \subseteq A^*$ is a set of sequences described by a *regular expression.* Regular expressions over a predefined alphabet A are defined recursively as follows. Every element of A is a regular expression, the *union* of two regular expressions is a regular expression (written $\alpha + \beta$ for regular expressions α and β), the *concatenation* of two regular expressions is a regular expression (written $\alpha \cdot \beta$ or $\alpha\beta$), and the *Kleene closure* of a regular expression is a regular expression (written α^*). The details of the correspondence of a regular expression to a regular set can be found in [74]. Regular languages are precisely the sets L of sequences for which there exist finite-state automata M such that $\mathcal{L}(M) = L$. Therefore we may say that a set of sequences is a regular set in order to indicate that it can be expressed by a finite-state automaton, without making explicit the automaton itself.

Regular sets are closed under union, concatenation, Kleene closure, intersection, set complement (within A^*), and substitutions. The Kleene closure of a set L of finite sequences is defined as $\bigcup_{n \in \omega} L^n$ where $L^0 = \{\varepsilon\}$ and $L^{(n+1)} = L \cdot L^n$. The Kleene closure of a single finite sequence w is the Kleene closure of the set $\{w\}$. A substitution is a function from an alphabet (set of symbols) A to a set of regular sets. It maps each symbol $a \in A$ to a regular set R_a. We extend it naturally to sequences, and then via union to sets of sequences: for $W \subseteq A^*$, $s(W) = \bigcup_{w \in W} s(w)$. Therefore it distributes over sequence and set concatenation. If X is a regular set, then so are $\mathit{pref}(X)$, $\mathit{ext}(w, X)$, and $\mathit{ext}_n(w, X)$.

A Mealy machine is a particular form of FSM in which the symbols Σ have a specific kind of internal structure [101]. Specifically, Σ is $B^{(I \cup O)}$ where B is the set of values that a wire may hold, and I and O are the input- and output-wires, respectively, of a hardware system. Traditional Mealy machines [101] are deterministic in the following sense: for every state $q \in Q$ and input-value combination $x \in B^I$, there exist a unique output-value combination $y \in B^O$ and a unique next-state $q' \in Q$ such that $\langle q, (x \cup y), q' \rangle \in \delta$. If a Mealy machine is deterministic in this sense, we say that it is a deterministic Mealy machine. A deterministic Mealy machine must be a deterministic FSM. However, a Mealy machine can be a deterministic FSM and yet not be a deterministic Mealy machine. We will work with a variant of traditional Mealy machines, that need not be deterministic in the sense just described. They are defined precisely as are finite-state automata, except that Σ must have the additional structure we have described ($\Sigma = B^{(I \cup O)}$ for some disjoint sets I and O), and every input-value combination must be represented on the out-edges of every state:

$$\forall q \in Q, x \in B^I . \exists y \in B^O, q' \in Q . \langle q, (x \cup y), q' \rangle \in \delta$$

2.3 Circuit Algebra

Circuit algebra formalizes the notion of circuit *structures.* It defines the constraints we expect to hold of mathematical operations (on circuit *models)* that are supposed to reflect the physical wiring together or packaging of actual circuits. In this section, we present the mathematical operators for building hierarchical circuit models, and present the axioms of circuit algebra, also called its *rules.* We then summarize the theorems of circuit algebra from [61, Chapter 2].

In this framework, a circuit description must have associated with it two finite disjoint sets I and O. I is the set of its input-wire names, and O is the set of its output-wire names. Following our naming convention, a circuit description C has associated sets I and O, a circuit description C' has associated sets I' and O', etc. We also say that $A = (I \cup O)$, $A' = (I' \cup O')$, etc.

Three operators are defined for circuit descriptions: composition (written $\|$), hiding (del), and renaming (ren). They are intended to reflect the operations of wiring two circuits together, hiding some of the output wires of a circuit in order to indicate that these wires cannot be wired up to any further wires in other circuits, and making explicit when two circuits share a wire, respectively. Composition ($\|$) combines two circuit descriptions into a single circuit description by identifying wires that have the same name. Outputs may not be wired together, so $C'' = C \parallel C'$ is only defined if $(O \cap O') = \emptyset$. The circuit algebra framework also requires that $O'' = (O \cup O')$ and

$I'' = ((I \cup I') - O'')$. The renaming operator *ren* changes wire names. $C' = ren(r)(C)$ is only defined if r is a function that is injective over A. In that case, $I' = r(I)$ and $O' = r(O)$. The hiding operator *del* hides output wires of the circuit so that they cannot be connected to other wires (via composition). $C' = del(D)(C)$ is only defined if $D \subseteq O$. In that case, $I' = I$ and $O' = (O - D)$.

For a discussion of these restrictions and their ramifications, see [61, Section 2.3]. There it is shown that the requirements that $(I \cap O) = \emptyset$ and that outputs not be wired together do not place unrealistic constraints on the circuits that can be modeled.

The following rules constitute the axioms of circuit algebra. Any circuit modeling framework in which the circuit models include disjoint I and O components and in which the operators obey the constraints listed above and for which the following equations hold – is a circuit algebra. In this presentation of the rules of circuit algebra, we assume that T, T_1, T_2 and T_3 are circuit descriptions.

C1: If O_1, O_2, and O_3 are pairwise disjoint, then $T_1 \parallel (T_2 \parallel T_3) = (T_1 \parallel T_2) \parallel T_3$.

C2: If $(O_1 \cap O_2) = \emptyset$, then $T_1 \parallel T_2 = T_2 \parallel T_1$.

C3: If r' is injective over A and r is injective over $r'(A)$, then $ren(r)(ren(r')(T)) = ren(r \circ r')(T)$.

C4: If $(O_1 \cap O_2) = \emptyset$ and r is injective over $(A_1 \cup A_2)$, then

$$ren(r)(T_1 \parallel T_2) = ren(r)(T_1) \parallel ren(r)(T_2)$$

C5: $ren(\mathbf{1}_A)(T) = T$

C6: If $(D_1 \cap D_2) = \emptyset$ and $(D_1 \cup D_2) \subseteq O$, then $del(D_1)(del(D_2)(T)) = del(D_1 \cup D_2)(T)$.

C7: $del(\emptyset)(T) = T$.

C8: If $(O_1 \cap O_2) = \emptyset$, $(D_1 \cap A_2) = (D_2 \cap A_1) = \emptyset$, $D_1 \subseteq O_1$ and $D_2 \subseteq O_2$, then

$$del(D_1)(T_1) \parallel del(D_2)(T_2) = del(D_1 \cup D_2)(T_1 \parallel T_2)$$

C9: If $D \subseteq O$ and $r_{|(A-D)} = r'_{|(A-D)}$ and r' is injective over A, then

$$ren(r)(del(D)(T)) = del(r'(D))(ren(r')(T))$$

In [61, Chapter 2], the following results are derived for circuit algebras. First, a definition is given for the *structural equivalence* of two circuit structures. Two circuit structures are structurally equivalent if they have the same input wires and the same output wires, and corresponding basic components which are connected in the same way. It is then proved that for a minimally informative model of circuit structures, the *equivalence classes* of structurally equivalent circuit structures form a circuit algebra.

It is proved that every circuit algebra expression is algebraically equivalent (i.e., equivalent by the rules of circuit algebra) to a *normal-form expression.* A normal-form expression contains only a single application of the hiding operator, which is the outermost operator. This operator is applied to an expression that is the composition of multiple subexpressions, each of which consists of the application of a renaming operator to a primitive circuit structure S_i. In other words, a normal-form expression has the following form:

$$del(D)[ren(r_1)(S_1) \parallel ren(r_2)(S_2) \parallel \ldots \parallel ren(r_n)(S_n)]$$

This result is key to proving that if two expressions describe structurally equivalent circuits, the expressions are algebraically equivalent.

As a direct result, any behavioral circuit model that forms a circuit algebra describes all circuits and assigns the same behavior to structurally equivalent circuits. The axioms of circuit algebra hold of both our model of combinational circuits and our model of synchronous circuits. Therefore these models adequately capture the structural view of such circuits.

2.4 Vectors and Sequences to Denote Circuit Behavior

We are concerned in this thesis with the behavior of circuits. Therefore we name their wires and refer to the values held on each wire during each clock cycle. A vector is an assignment of values to the elements of a known set of wires. A sequence of vectors, in contrast, refers to the evolution over time of the values on a known set of wires, *i.e.,* one vector per clock cycle. These mathematical objects may be defined using standard mathematical notation for functions and for regular sets.

$\mathcal{B}$ denotes the digital domain of wire values $\{0, 1\}$. A single-output Boolean function $f : \mathcal{B}^n \longrightarrow \mathcal{B}$ assigns an element of $\mathcal{B}$ to every element in a set of n wires. In our own work, we will always make explicit the particular set of wires to which assignments are made. Thus assignments of Boolean values to a particular set of wires will take the form $\mathcal{B}^A$, where A is a predefined set of wire names. The elements of $\mathcal{B}^A$ are *Boolean vectors.* Recall that $\mathcal{B}^A$ also denotes the set of functions from domain A to codomain $\mathcal{B}$. This notation is entirely consistent with the explanation we have just given.

In general, for any domain $\mathcal{V}$ of wire values, and any predefined set of wire names A, every element of $\mathcal{V}^A$ is a *vector.* Renaming may be naturally extended to vectors: if $r : A' \longrightarrow B$ is injective over $A \subseteq A'$, then $r : \mathcal{V}^A \longrightarrow \mathcal{V}^{r(A)}$ is defined by $\forall w \in \mathcal{V}^A.\forall a \in A.(r(w))(r(a)) = w(a)$. The *pointwise extension* of any partial order R over a domain of wire values $\mathcal{V}$ to vectors of these values is simply the pointwise extension of R to functions: per set of wires H, $\forall x, y \in \mathcal{V}^H.[xR_Hy \Longleftrightarrow \forall h \in H.\langle x(h), y(h)\rangle \in R]$. If $\mathcal{V}$ contains only one element $v \in \mathcal{V}$, then v^A denotes the unique function in $\mathcal{V}^A$.

We have referred to the *standard ternary extension* of a Boolean function. We will now define this term precisely. Let $\mathcal{B}_X = \{0, 1, \mathbf{X}\}$ be the ternary domain of wire values used for ternary simulation (see Section 1.4.5). We extend the partial order *information ordering* $\leq$, in which $\mathbf{X} < 0$ and $\mathbf{X} < 1$ and in which 0 and 1 are incomparable [37], pointwise to vectors of wire values.

The standard ternary extension of a single-output Boolean function $f : \mathcal{B}^C \longrightarrow \mathcal{B}$ is the function $g : (\mathcal{B}_X)^C \longrightarrow \mathcal{B}_X$ such that $g = lub\{h : (\mathcal{B}_X)^C \longrightarrow \mathcal{B}_X \mid h_{|\mathcal{B}^C} = f$ and h is monotonic$\}$. For example, the standard ternary extension of the Boolean *or*-function maps the input vectors $1\mathbf{X}$ and $\mathbf{X}1$ to 1, but maps the input vectors $0\mathbf{X}$, $\mathbf{X}0$ and $\mathbf{XX}$ to $\mathbf{X}$.

This definition extends to multi-output Boolean functions $f : \mathcal{B}^C \longrightarrow \mathcal{B}^D$ via a transformation of the *type* of f. We note that $f : \mathcal{B}^C \longrightarrow \mathcal{B}^D$ may be represented by a vector of single-output Boolean functions $f_d : \mathcal{B}^C \longrightarrow \mathcal{B}$, one for each element $d \in D$. That is, $f : \mathcal{B}^C \longrightarrow \mathcal{B}^D$ contains information equivalent to that contained by the function $f' \in [D \longrightarrow [\mathcal{B}^C \longrightarrow \mathcal{B}]]$, that maps every $d \in D$ to f_d such that for every $x \in \mathcal{B}^C$, $f_d(x) = (f(x))(d)$. Then the standard ternary extension of $f : \mathcal{B}^C \longrightarrow \mathcal{B}^D$ is the result of taking the standard ternary extension g_d of each f_d. In other words, $g : (\mathcal{B}_X)^C \longrightarrow (\mathcal{B}_X)^D$ is just $g' \in [D \longrightarrow [(\mathcal{B}_X)^C \longrightarrow \mathcal{B}_X]]$ (where $g'(d) = g_d$ for each $d \in D$), considered as a multi-output function. With respect to the partial order information ordering $\leq$ over $\mathcal{B}_X$, and its extension pointwise to vectors of values in $(\mathcal{B}_X)^C$ and $(\mathcal{B}_X)^D$, the standard ternary extension of every Boolean function is a monotonic function.

The transformation between types illustrated in the above discussion is an example of an *isomorphism.* Two sets A and B are *isomorphic* if their elements are in one-to-one correspondence. In other words, there exists an injective function $h_1 : A \longrightarrow B$ such that $h_1(A) = B$. We may refer to h_1 as an *isomorphism.* If the elements of the sets A and B have known internal structure (like f and f' in the above discussion), isomorphism also requires that the functions in which we are interested – extended in the standard ways to these elements – map corresponding elements of A and B to elements that also correspond. In the above discussion, for example, h_1 maps f to f' and g to g', and the function in which we are interested is the one that maps a Boolean function to its standard ternary extension.

In the remainder of this section, we present our definitions in terms of an unspecified domain $\mathcal{V}$ of wire values. We are concerned with vectors of values over this domain ($x \in \mathcal{V}^C$) and with sequences of such vectors ($w \in (\mathcal{V}^C)^*$).

In preparation for defining our behavioral circuit models, we define the functions del and del^{-1} over these domains. For $D \subseteq A$ and $x \in \mathcal{V}^A$, $del(D)(x)$ is defined to be the unique $y \in \mathcal{V}^{(A-D)}$ for which there exists $z \in \mathcal{V}^D$ such that $x = (y \cup z)$. This definition extends naturally to sets of vectors, and to sequences of vectors and thence to sets of sequences of vectors. For D disjoint from A, and $y \in \mathcal{V}^A$, $del^{-1}(D)(y)$ is defined to be the set $\{(y \cup z) \mid z \in \mathcal{V}^D\}$. This definition may be naturally extended to sequences of vectors. We extend this definition to sets of vectors, and to sets of sequences of vectors, as follows: for $Y \subseteq \mathcal{V}^A$ or for $Y \subseteq (\mathcal{V}^A)^*$, $del^{-1}(D)(Y) = \bigcup_{y \in Y} del^{-1}(D)(y)$.

These functions have the following useful properties.

Property 2.1 *If $D_2 \subseteq A$ and $D_1 \subseteq (A - D_2)$, and $W \subseteq \mathcal{V}^A$ or $W \subseteq (\mathcal{V}^A)^*$, then*

$$del(D_1)[del(D_2)(W)] = del(D_1 \cup D_2)(W)$$

Property 2.2 *If $D_1 \subseteq A$ and $(D_2 \cap A) = \emptyset$, and $W \subseteq \mathcal{V}^A$ or $W \subseteq (\mathcal{V}^A)^*$, then*

$$del(D_1)[del^{-1}(D_2)(W)] = del^{-1}(D_2)[del(D_1)(W)]$$

Property 2.3 *If A and D_1 and D_2 are pairwise disjoint, and $W \subseteq \mathcal{V}^A$ or $W \subseteq (\mathcal{V}^A)^*$, then*

$$del^{-1}(D_1)[del(D_2)^{-1}(W)] = del^{-1}(D_1 \cup D_2)(W)$$

Property 2.4 *If $(D \cap A) = \emptyset$, and $W, Z \subseteq \mathcal{V}^A$ or $W, Z \subseteq (\mathcal{V}^A)^*$, then*

$$del^{-1}(D)(W \cap Z) = del^{-1}(D)(W) \cap del^{-1}(D)(Z)$$

Chapter 3

Combinational circuit models

3.1 Introduction

In this chapter we develop relational models for combinational circuits. We intend that a behavior of a synchronous circuit be modeled as a sequence of combinational steps, each of which takes place within a single clock cycle. In order to better understand the properties of such a sequence, we first explore the internal structure of a combinational step. We proceed from the assumption that this exploration is best pursued via a full study of the combinational case, considered independently of the fact that our final goal in developing combinational circuit models is to [understand the combinational case sufficiently well that we can] embed aspects of these models into full sequential circuit models.

Combinational circuits are a special case of clocked sequential circuits: they just happen to exhibit the same output values in response to the same input values during every clock cycle. Therefore even after we expand the combinational circuit model that we develop in this chapter into a model of clocked sequential circuits and specifications, we will still need to be able to model combinational circuits. In order to enable the expansion of our combinational circuit model to model clocked sequential models *as well as* the purely combinational circuitry which we tackle in this chapter, we adopt a modified Mealy machine model of synchronous circuit behavior. Thus our model of combinational circuit behavior posits that there is zero delay between the arrival of the input values and the computation of the resulting output values during the current clock cycle. In other words, we use a zero-delay model of combinational behavior, rather than a model that incorporates propagation delay.

Our goal is to develop a model useful for both substitution and formal hierarchical verification of combinational circuits. In order to meet this goal, our model should support hierarchical construction and modular description of circuits. We also want to support nondeterminism, in order to enable the expression of a specification which allows for multiple correct implementations. The circuit algebra framework presented in the previous chapter provides hierarchical construction and modular description capabilities. However, as discussed in Chapter 1, we still need to think about how our model handles the combinational feedback loops that may be created by indiscriminate application of the composition operator. As discussed in Section 1.4, our desire to support nondeterministic models and modular 'black-box' descriptions and to support both the expression of purely combinational circuitry and clocked sequential behavior in a single *sequential* model means that existing known solutions to this problem are not applicable in our framework.

The organization of this chapter is as follows. In the following section, we describe our solution to the problem of modeling unlatched feedback loops. We also present the formal definition of a combinational relation structure, which is our representation of a specification or implementation of a combinational circuit. The model incorporates our solution to the zero-delay cycle modeling problem. In Section 3.3, we define the circuit algebra operations for these models and prove the closure of the class of combinational relation structures under these algebraic operations. In each section, we provide examples to illustrate the theory.

3.2 The combinational circuit model

3.2.1 Introduction

We seek to develop models of combinational circuits that support hierarchical construction and modular description of such circuits, and that support nondeterminism. The latter enables the expression of a specification which captures the minimum requirements of a circuit instead of requiring that we overspecify by including irrelevant implementation details.

The circuit algebra framework provides the first two of these desired properties. We utilize a *relational* model of combinational behavior in order to provide the third. In the relational model, combinational behavior is represented by a set of input-output value combinations that the circuit may produce or that the specification allows.

In modeling synchronous behavior as a sequence of clock cycles, we ignore the passing of time within a single clock cycle, and only examine the final digital values of the wires after stabilization within the cycle. However, in some cases not all wires need stabilize. Normally, combinational circuits that are modeled usefully at a digital level of abstraction are assumed to be wired together only into topologies with unidirectional flow of information. That is, they are constructed from well-behaved digital parts using only *cascade composition,* so that the resulting network forms an acyclic directed graph. If we restrict ourselves to cascade composition, we avoid all cases in which wires need not stabilize. Thus in this case the binary digital abstraction for wire values suffices.

However, as discussed in Section 1.4, the standard syntactic methods for detecting non-cascade composition do not work in the presence of nondeterministic models and black-box behavioral descriptions for which no gate-level structural description is available. Therefore we are obliged to model combinational feedback loops. Our relational model of combinational circuit behavior utilizes semantic means in order to allow the expression and hence detection of such zero-delay cycles.

In order to meet the goal that our models support hierarchical verification, we require the ability to model the possible *environments* of a circuit as circuit models as well. This leads to the need to distinguish two kinds of possible behaviors of a circuit: those which are desirable and those which are possible but undesirable. The necessity for a two-language model of combinational circuits and their specifications is elucidated in Section 4.2, where we develop the notion of a circuit's environment as a tool for determining the correct implementations of a requirements specification.

In the remainder of this section, we first show why the binary digital abstraction does not suffice for modeling combinational feedback loops. We then present our solution: the addition of a third digital value which is intended to denote a situation in which a wire does not stabilize to either of the Boolean values 0 or 1 by the end of the clock cycle. Finally, we present our formal model for combinational circuits and their specifications, and provide examples.

3.2.2 The ternary domain of wire values

If we restrict ourselves to cascade composition, we avoid all cases in which wires need not stabilize to either of the digital Boolean values 0 or 1. Thus in this case the binary digital abstraction for wire values suffices. Boolean relations handle this case, and from them we learn that we must use relations rather than functions to support nondeterminism [44, 22].

In this section, we explain the additional expressiveness of Boolean relations over Boolean functions, and show why they are nevertheless inadequate to model non-cascade composition. We then present our solution to modeling wire values in the presence of non-cascade composition.

In modeling synchronous behavior as a sequence of clock cycles, we only examine the final digital values of the nodes after stabilization within a clock cycle. If we restrict the unlatched circuits we model to those involving only cascade composition, such a circuit may be modeled as a Boolean function from input-value combinations (values on input wires) to output-value combinations (values on all other nodes). Thus the simplest formal model of such a piece of combinational logic having input wire set I and output wire set O is a mapping from input vectors to output vectors, $f : \mathcal{B}^I \longrightarrow \mathcal{B}^O$. However, this functional representation is inadequate in the presence of choice.

If we allow nondeterminism in order to handle requirements specifications, we can no longer model everything we want with a function. Consider the following potential extension to the function model of circuit behavior, which we do not adopt. We might consider modeling a circuit by a set of mappings $f_j : \mathcal{B}^I \longrightarrow \mathcal{B}^{o_j}$, one for each $o_j \in O$. In order to handle the case in which an output wire's value in response to some particular set of input-value combinations is arbitrary, we could allow each f_j to be partial. In such a model, for any j and $w \in \mathcal{B}^I$ such that $f_j(w)$ is undefined, we would understand that the output wire o_j may take arbitrary value under input-value combination w. The proposed formalism, however, does not allow the expression of all the degrees of freedom a requirements specification might require.

The above formalism does not support correlated outputs: it cannot express the requirement that (for some given input value combination w) one output wire, o_{j_1}, may have the value 1 only if another output wire, o_{j_2}, has the value 0. In other words, we may wish to express constrained choice, in which some input-value combination may result in any of several specific output-value combinations, but not in a completely arbitrary output-value combination. These are the *conditioned vertex equivalence classes* of [22], which cannot be expressed by a set of mappings $f_j : \mathcal{B}^I \longrightarrow \mathcal{B}^{o_j}$.

Thus in order to use our model to express a specification, with all its potential degrees of freedom, a functional representation is not sufficient. Instead, we model a combinational circuit or circuit specification as a *relation* between input-value combinations and output-value combinations. This allows multiple alternative output vectors as the result of a single input-value combination. Note that this representation allows the expression of correlated outputs in the presence of choice. It corresponds to the output characteristic function of Cerny and Marin [44] and the Boolean relation representation of Brayton and Somenzi [22].

However, Boolean relations are not an adequate representation for combinational circuitry in the presence of non-cascade composition, as they do not provide appropriate semantics for combinational feedback loops. The following example demonstrates the inadequacy of the binary digital value domain $\mathcal{B}$ in the presence of non-cascade composition.

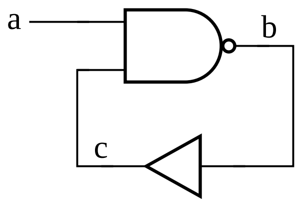

Figure 3.1: Gated ring oscillator

Consider the circuit illustrated in Figure 3.1. It consists of a nand-gate with inputs labeled a and c and output wire b composed together with a non-inverting buffer whose input wire is the wire labeled b and whose output is the wire labeled c. While both of these components can be modeled correctly by the obvious Boolean relations (which happen to be functions), their composition yields the Boolean relation $R \subseteq \mathcal{B}^{\{a,b,c\}}$ that contains only the single vector that assigns 1 to a, 0 to b, and 0 to c. This relation does not admit the possibility that a take the value 1. This is clearly incorrect since a is an input: its value cannot be controlled by the circuit.

In reality, if this circuit receives as input the value 1, then either b and c will oscillate, or they will both get stuck at some intermediate voltage which is neither a digital 1 nor a digital 0. Neither of b or c will stabilize to a recognized digital value. Such non-digital behavior cannot be expressed by a Boolean relation, and so this behavior simply disappears from the representation. Disappearing behavior can cause consistency problems, and is clearly inappropriate.

The following question is often raised: does disappearing loop behavior always manifest itself as a Boolean relation that does not admit all possible input value combinations? If so, the disappearing behavior phenomenon is easily detected, and compositions that lead to it disallowed. Unfortunately, this is not the case in general. In Section 3.3.5 we present an example Boolean circuit that contains a combinational feedback loop for which the Boolean relation representation admits all possible input value combinations and yet fails to include all possible behaviors of the circuit.

This particular case also illustrates the type of consistency problems that may arise from disappearing behavior in the model. The Boolean relation for the gated ring oscillator, with b considered a primary output and c only an internal node, is compatible with the Boolean relation for an inverter, which is $R \subseteq \mathcal{B}^{\{a,b\}}$ that contains two vectors: one assigns 0 to a and 1 to b, and the other assigns 1 to a and 0 to b. This incorrectly implies that the gated ring oscillator is a correct implementation of an inverter specification. (Recall from Section 1.5.2 that a Boolean relation specification is correctly implemented by any Boolean relation circuit model that is compatible with it, which means that when considered as a set of input-output value combinations, the latter is a subset of the former). Thus the disappearing behavior of the gated ring oscillator has led to a situation in which verification is not sound. An unsound formal verification result of this sort, which claims a circuit to be a correct implementation of a requirements specification when in fact it is not, is called a *false positive* result. Models which lead to false positive results are clearly unacceptable for a formal verification framework.

In order to solve the expressiveness dilemma just illustrated, we work within a ternary framework. We call our ternary set of digital values $\mathcal{T} = \{0, 1, \perp\}$. The value $\perp$, pronounced “bottom,” is intended to represent a lack of convergence to either of the other two values, or an *undefined* value. It represents oscillation or stabilization to an intermediate voltage. Note that we do not intend $\perp$ to denote the value on a wire that has either the value 0 or the value 1, but for which the correct value is unknown. That situation is handled in our formalism by including both possibilities in the nondeterministic relation representing the circuit's possible behaviors. Therefore we distinguish $\mathcal{T}$ from the more general domain $\mathcal{B}_X$.

We will demonstrate that the addition and appropriate use of the new third wire value suffice to solve the problem illustrated in the example above. The ternary-domain models we use in this demonstration meet some but not all of the formal constraints we will place on our final models. However, they suffice to illustrate how the third wire value may be used to flag the presence of combinational feedback loops, thereby eliminating the class of false positives in formal verification that can be caused by the disappearing behavior phenomenon just shown.

We model the non-inverting buffer and the nand-gate by the standard ternary extensions of their Boolean relation representations (which happen to be functions). The addition of information about the behavior of each of these circuits in the presence of a non-Boolean value on an input wire suffices to guarantee that *some* behavior will remain in the composite model, enough to indicate

a	c	b
$\perp$	$\perp$	$\perp$
$\perp$	0	1
$\perp$	1	$\perp$
0	$\perp$	1
0	0	1
0	1	1
1	$\perp$	$\perp$
1	0	1
1	1	0

nand(a,c)=b

b	c
$\perp$	$\perp$
0	0
1	1

buf(b)=c

a	b	c
$\perp$	$\perp$	$\perp$
0	1	1
1	$\perp$	$\perp$

gated ring oscillator

Figure 3.2: Ternary-domain relation models for nand-gate, buffer, and their composition

a	b
$\perp$	$\perp$
0	1
1	$\perp$

(A)

a	b
$\perp$	$\perp$
0	1
1	0

(B)

Figure 3.3: Ternary-domain relation models for (A) oscillator and (B) inverter

the presence of a feedback loop. This is illustrated in Figure 3.2, where the ternary-domain models for the nand-gate and the non-inverting buffer, and the resulting composite model, all appear in tabular form. This composite model correctly reflects the fact that in the composite circuit, when a is held high, b and c will both oscillate or will settle at an intermediate voltage. Extending this example further, we illustrate in Figure 3.3 the ternary-domain relation representation of the gated-ring oscillator with b considered a primary output and c only an internal node. This figure also depicts the ternary-domain relation model of an inverter. We see that the former is *not* compatible with the latter, which correctly reflects the fact that the composite circuit depicted in Figure 3.1 does not correctly implement the specification for an inverter.

We have not yet discussed the formal constraints we place on a valid combinational circuit model, nor given the precise definition of $\perp$'s appropriate use. In the remainder of this chapter, we will fully develop the above example in the new ternary domain of wire values, along with the relevant formal definitions. The development of this illustrative example will occur as we present the relevant parts of our framework for the formal verification of combinational circuits. Correct use of the ternary domain of wire values always yields models for combinational logic (whether in cascade or in non-cascade composition topologies) that explicitly represent the behavior of the circuitry in the presence of any input value combination. Proof of this fact appears later in this chapter.

3.2.3 Combinational relation structures

In this section we present our formal model of combinational circuits and their specifications. We call these models *combinational relation structures.* The class of combinational relation structures forms a circuit algebra under algebraic operations that correspond to renaming of wires, wiring together of two circuits to form a composite circuit, and packaging of a wire as an internal node rather than a primary output.

Combinational relation structures are defined over the ternary domain of wire values. They also distinguish between two kinds of possible behaviors of the circuit being modeled: those which are desirable, or intended, and those which are possible but undesirable. The usefulness of this distinction will become clear in Section 4.2, in which we develop the notion of a circuit's environment as a tool for determining correct implementations.

In addition, we place restrictions on the form of these two sets of behaviors that are based on an underlying partial order $\leq$ in the set $\mathcal{T}$, the ternary wire domain: $\perp < 1$ and $\perp < 0$, and 0 and 1 are incomparable. This is the *information ordering* of [37]. The definedness ordering $\leq$ on $\mathcal{T}$ may be extended pointwise to a partial order on vectors of wire values.

Intuitively, a realistic circuit model respects this underlying partial order: if an output wire has a well-defined 0 or 1 value despite some input wire's holding an undefined value, then this output wire does not change value nor "become undefined" *(i.e.,* stabilize differently or not at all) if that input wire instead stabilizes at a defined value. Traditionally this intuition has been reflected in a requirement of functional monotonicity: a requirement that the set of possible behaviors of a circuit model constitute a function that is monotonic in the set of possible input-value combinations to the circuit. However, it is not clear what the appropriate definition of monotonicity should be for a relation. Among the options available, we have chosen one that suffices to provide us with sound verification results. This is the upward-chains property, which we introduce below.

The new value $\perp$ may represent oscillation. There are many different wave patterns that oscillation may exhibit, including patterns that may be indistinguishable from a steady 1 or a steady 0 except for intermittent glitches. Therefore we require that the set of possible behaviors of a circuit and the set of failure behaviors of a circuit each be modeled as an *input-downward-closed* set. That is, if an input value combination w admits an output value combination z, then all input value combinations that are the same as w except for some replacements of 0 and 1 values by $\perp$ values must also admit z. This is because a $\perp$ value on one of its input wires *may* appear to a circuit as a 0 or 1, if the wire happens to hold one of these Boolean values for a sufficiently long time prior to the clock tick.

The formal definition of the input-downward-closure constraint uses the definedness ordering $\leq$ over vectors of wire values. The set of input wires to a model is I and its set of output wires is O.

Thus its set of possible behaviors and its set of failure behaviors are subsets of $\mathcal{T}^{(I \cup O)}$. Formally, the *input-downwards-closure* (*IDC*) property holds of the set $W \subseteq \mathcal{T}^{(I \cup O)}$ if and only if

$$\forall x, x' \in \mathcal{T}^I . \forall y \in \mathcal{T}^O . (x' \leq x \wedge (x \cup y) \in W) \Longrightarrow ((x' \cup y) \in W)$$

A realistic circuit model cannot refuse any input value combinations. Mathematically, this constraint is formalized as a *totality* constraint on the set of possible behaviors of a circuit: a set $R \subseteq \mathcal{T}^{(I \cup O)}$ is *total* if and only if

$$\forall x \in \mathcal{T}^I . \exists y \in \mathcal{T}^O . (x \cup y) \in R$$

Unfortunately, as we have already seen in previous sections of this thesis, totality is not preserved by the composition operator. This is true even when we use the ternary domain of wire values, and even when the set of possible behaviors of the model obeys the input-downward-closure constraint.

In order to guarantee the preservation of totality under the algebraic operations, we impose an additional constraint on how the third wire value $\perp$ may be used in our models. This constraint is called the *receptiveness* condition, and it must hold of the set of possible behaviors of every model. It is formalized in terms of the upward-chains property, which is defined as follows. We say that a set $C \subseteq \mathcal{T}^{(I \cup O)}$ has the upward-chains property if and only if

$$\forall x, x' \in \mathcal{T}^I . \forall y \in \mathcal{T}^O . [[(x \cup y) \in C \wedge x \leq x'] \Longrightarrow \exists y' \in \mathcal{T}^O . y \leq y' \wedge (x' \cup y') \in C]$$

Formally, the receptiveness condition is defined as follows. A set $P \subseteq \mathcal{T}^{(I \cup O)}$ is *receptive* if and only if there exists a subset C of P which is total and which has the upward-chains property. Intuitively, the reason we do not require that the full set of possible behaviors have this property is that a specification model may allow input-output value combinations that need not be exhibited by any correct implementation. As described above, the upward-chains property is a *relational* analog of functional monotonicity. It suffices to preserve totality under the algebraic operations.

Our use of the receptiveness constraint suffices to guarantee closure of the class of models we consider under the algebraic operations: it is preserved over the algebraic operations of composition, renaming, and deletion (as is proved in a subsequent section of this chapter). The totality condition guarantees the ability of every relation structure to respond no matter what inputs it encounters, and the upward-chains component of the receptiveness constraint guarantees the preservation of totality under the algebraic operations.

Combinational relation structures are our model of combinational circuits and their requirements specifications. Formally, we define a *combinational relation structure* (sometimes referred to as a relation structure) to be a quadruple $T = (I, O, S, F)$ such that

- I and O are disjoint finite sets,

- $F \subseteq \mathcal{T}^{(I \cup O)}$ is a (possibly empty) set of circuit behaviors, known as the *failure set* of the relation structure, which obeys the *input-downwards-closure* constraint,
- $S \subseteq \mathcal{T}^{(I \cup O)}$ is a (possibly empty) set of circuit behaviors, known as the *success set* of the relation structure, and
- $P = S \cup F$, the set of *possible behaviors* of T, obeys both the input-downwards-closure constraint and the receptiveness constraint.

I and O are intended to represent the input- and output-wire sets, respectively, of the circuit or specification being modeled. As part of our standard notation, we define $A = (I \cup O)$ to be the alphabet of the relation structure. Note that P must be non-empty, as otherwise it cannot be receptive.

In developing the theory of combinational relation structures in the remainder of this chapter, we may wish to emphasize various aspects of these models at the expense of others. For example, in discussing the receptiveness constraint we may wish to ignore the precise contents of the S and F-sets of a relation structure, and only consider its full P-set. In discussing the preservation of input-downward-closure under the algebraic operations, we may wish to concentrate on the F and P-sets of a relation structure and to ignore the contents of its S-set. The following definition provides a tool we will use in this spirit in later sections of this chapter.

We define a preorder $\sqsubseteq$ on combinational relation structures having the same I and O sets:

$$(I, O, S, F) \sqsubseteq (I, O, S', F') \text{ if and only if } [(F \subseteq F') \text{ and } (P \subseteq P')]$$

Later we will extend the relation $\sqsubseteq$ to structures $(I, O, \emptyset, \emptyset)$ as well (see page 90 in Chapter 4).

This preorder induces a partial order among its equivalence classes. If $F = F'$ and $P = P'$ then $[T]_{\sqsubseteq} = [T']_{\sqsubseteq}$, and we write $T \sim_{\sqsubseteq} T'$. Essentially, T and T' are in the same $\sqsubseteq$-equivalence class if they vary only in the amount of overlap between their S and F sets. Hence setting $S = (P - F)$ defines a unique representative of the $\sqsubseteq$-equivalence class (I, O, F, P), and we may sometimes use the ambiguous notation $T = (I, O, F, P)$ in place of (I, O, S, F). In addition, when discussing receptiveness alone, we may sometimes use the notation $T = (I, O, P)$, ignoring the F-set of the relation structure altogether.

In the following section, we present the formal definitions of the algebraic operations of composition, renaming, and deletion. We also prove that the class of combinational relation structures together with these operations forms a circuit algebra. We conclude the current section with some examples.

3.2.4 Examples

In this subsection we present examples of combinational relation structures that illustrate our modeling technique and further explain how to use our two-language model. We also use these examples to introduce some more notational conventions.

In presenting a combinational relation structure's S, F, and P-sets, we utilize the following notation: the literal a indicates that the value on node a is 1, the literal $\overline{a}$ indicates that the value on node a is 0, and the literal $\perp_a$ indicates that the value on node a is $\perp$. Concatenations of such literals are called monomials. Monomials represent conjunctions of such statements, as expected, and sets of monomials represent the disjunction of their member monomials.

Example 3.1 *We represent an inverter with input wire labeled a and output wire labeled b as the following combinational relation structure:*

$$T_{inv-ab} = (I = \{a\}, O = \{b\}, S_{inv-ab}, F = \emptyset)$$

where

$$S_{inv-ab} = \{\overline{a}b, a\overline{b}, \perp_a\perp_b, \perp_a b, \perp_a\overline{b}\}$$

Note that the success set of the inverter is simply the smallest input-downward-closed set that contains the standard ternary extension of the Boolean function representation of this gate. The F-set is effectively not in use in this example.

Example 3.2 *We represent a non-inverting buffer with input wire labeled b and output wire labeled c as the following combinational relation structure:*

$$T_{buf-bc} = (I = \{b\}, O = \{c\}, S_{buf-bc}, F = \emptyset)$$

where

$$S_{buf-bc} = \{\overline{b}\overline{c}, bc, \perp_b\perp_c, \perp_b c, \perp_b\overline{c}\}$$

As in the previous example, the set of possible behaviors of the circuit model is simply the standard ternary extension of the Boolean function representation for the gate, with the addition of those behaviors required to make the relation input-downward-closed.

Example 3.3 *We represent a nand-gate with input wires labeled a and c and output wire labeled b as the following combinational relation structure:*

$$T_{nand-acb} = (I = \{a, c\}, O = \{b\}, S_{nand-acb}, F = \emptyset)$$

where $S_{nand-acb}$ is the relation

$$\{ac\overline{b}, a\overline{c}b, \overline{a}cb, \overline{a}\,\overline{c}b, \perp_a c\overline{b}, a\perp_c\overline{b}, \perp_a\overline{c}b, a\perp_c b, \perp_a cb, \overline{a}\perp_c b, a\perp_c\perp_b, \perp_a c\perp_b, \perp_a\perp_c b, \perp_a\perp_c\overline{b}, \perp_a\perp_c\perp_b\}$$

In this case the gate being modeled has two inputs. The set of possible behaviors is again the standard ternary extension of the Boolean function for the gate, with the addition of those behaviors necessary to make it input-downward-closed.

The following examples illustrate use of the F-set. Although the set of possible behaviors of a circuit must allow for any input value combination, the set of *desirable* behaviors need not include all such combinations. Example 3.4 illustrates use of the F-set to explicitly maintain information about the environments in which we expect those circuits implementing the specification to function correctly. Example 3.5 further clarifies the potential of this explicitly-delineated subset of a circuit's possible behaviors to make a requirements specification more expressive.

In the following example we introduce the notation X_e to indicate that the wire labeled e may take any of the values $0, 1$, or $\perp$. In other words, $X_e = \{e, \overline{e}, \perp_e\}$. For example, the extended monomial $a\overline{b}X_e$ is shorthand for the set of monomials $\{a\overline{b}e, a\overline{b}\overline{e}, a\overline{b}\perp_e\}$.

Example 3.4 *The following combinational relation structure is a specification for an inverter that we expect to place only in an environment in which its input stabilizes. Its input wire is labeled a and its output wire labeled b.*

$$T_0 = (I = \{a\}, O = \{b\}, S = \{a\overline{b}, \overline{a}b\}, F = \{\perp_a X_b\})$$

In fact, F-sets allow *more* degrees of freedom in specification than does a simple statement of the environments in which we expect those circuits implementing a specification to function correctly. Example 3.5 illustrates that the F-set of a combinational relation structure may contain a certain input-output value combination and yet not contain *all* output-value combinations as possible responses to that input-value combination.

Example 3.5 *In this example we consider a specification that maintains information about expected environments in its F-set, but which does not allow all output value combinations in response to an unexpected input value combination.*

The following combinational relation structure is a specification for an and-gate with input wires labeled a and b and output wire c, that we expect to place only in an environment in which the input value combination $a\overline{b}$ does not occur.

$$T_1 = (\{a, b\}, \{c\}, S_1, F_1)$$

where

$$S_1 = \{abc, \overline{a}b\overline{c}, \overline{a}\overline{b}\overline{c}, \perp_a bX_c, \overline{a}\perp_b\overline{c}, \perp_a\overline{b}\overline{c}, a\perp_b c, a\perp_b\perp_c, \perp_a\perp_b X_c\}$$

and

$$F_1 = \{a\overline{b}\overline{c}, \perp_a\overline{b}\overline{c}, a\perp_b\overline{c}, a\perp_b\perp_c, \perp_a\perp_b\overline{c}, \perp_a\perp_b\perp_c\}$$

Note that there is some overlap between S_1 and F_1. The following input-output value combinations appear in both: $a\perp_b\perp_c, \perp_a\overline{b}\overline{c}, \perp_a\perp_b\overline{c}, \perp_a\perp_b\perp_c$. A viable modeling alternative would be to eliminate the two combinations $a\perp_b\perp_c$ and $\perp_a\perp_b\perp_c$ from F_1. Note that the second may not be eliminated without the first being eliminated as well, as F_1 must remain input-downward-closed. If $a\perp_b\perp_c$ is maintained as an element of F_1, then it can be eliminated from S_1. However, the receptiveness constraint on $P_1 = F_1 \cup S_1$ requires that this input-output value combination appear in at least one of F_1 or S_1. In any case, F_1 specifies more than just a set of environments in which we would not necessarily expect the implementations of T_1 to function correctly: it also places constraints on their behavior even in these unintended environments.

3.3 The algebraic operations: combining relation structures

3.3.1 Introduction

We have defined a combinational relation structure to be a tuple (I, O, S, F) such that I and O are disjoint and such that certain properties hold of the sets S, F and $P = (S \cup F)$. In general, we will define some relation structures that correspond to specifications or to primitive gates, and build from them more complex specifications or circuits. In order to do that, we need formal mathematical definitions of operations that correspond to combination and manipulation of the actual circuits that are represented by our already-defined relation structures. We also need to know that the resulting mathematical objects are themselves combinational relation structures, since they are intended to represent circuits as well.

The following algebraic operations on combinational relation structures correspond to the manipulation and combination of combinational circuits to produce more complex or further-packaged circuitry. As required by the circuit algebra framework, these operations are:

- Composition ($\|$), which corresponds to wiring two circuits together.

  Formally this operation is defined in terms of two others: intersection ($\cap$) and inverse deletion (del^{-1}). Composition unifies the values on each wire that is common to the two component circuit representations. Intersection composes two combinational relation structures with precisely the same wire sets but no joint output wires, and inverse deletion "prepares" a relation structure for composition via the intersection operation by adding to it input wires whose values it ignores.

- Deletion or hiding (*del*), which makes explicit which output wires of a circuit are its intended primary outputs. It hides some of the output wires of a circuit in order to indicate that these wires do not participate in the observable input-output behavior of the newly-packaged circuit, and therefore cannot be further connected to the wires of any other circuit.
- Renaming (*ren*), which allows us to rename the wires of a circuit (without wiring together any previously distinct wires), in order to facilitate composition in any desired topology.

In the following subsection, we present the formal definitions of the algebraic operations on combinational relation structures. In Section 3.3.3, we prove that the results of applying these operations to combinational relation structures are themselves combinational relation structures. In Section 3.3.4, we prove that the class of combinational relation structures together with the algebraic operations forms a circuit algebra. Finally, in Section 3.3.5, we present some examples of composite circuit models in order to illustrate how the formal operations implement the intended actions on the circuits being modeled.

3.3.2 The algebraic operations

In this section we present the formal definitions of the algebraic operations on combinational relation structures.

- Composition ($\|$) corresponds to wiring two circuits together:

  Circuit composition is defined formally in terms of the simpler *intersection* and *inverse deletion* operators:

  - The *intersection* operator performs the composition of two combinational relation structures whose alphabets are identical but whose output-wire sets are disjoint.

    If (I, O, S, F) and (I', O', S', F') are combinational relation structures such that

    $$(I \cup O) = (I' \cup O') \text{ and } (O \cap O') = \emptyset$$

    then

    $$(I, O, S, F) \cap (I', O', S', F') = ((I \cap I'), (O \cup O'), (S \cap S'), ((P \cap F') \cup (F \cap P')))$$

    Note that the P-set of the result is the set $(P \cap P')$.
  - The *inverse deletion* operator adds new input wires to a combinational relation structure, and expands the sets of its behaviors to ignore the new input wires' values. This operator is used to unify the wire sets of two circuits before composition.

If (I, O, S, F) is a combinational relation structure, and D a set such that $(D \cap (I \cup O)) = \emptyset$, then

$$del^{-1}(D)((I, O, S, F)) = (I \cup D, O, del^{-1}(D)(S), del^{-1}(D)(F))$$

Recall from Chapter 2 that for $Y \subseteq \mathcal{T}^A$, $del^{-1}(D)(Y) = \bigcup_{y \in Y} del^{-1}(D)(y)$, and for $y \in \mathcal{T}^A$, $del^{-1}(D)(y) = \{(y \cup z) \mid z \in \mathcal{T}^D\}$.

We define circuit composition in terms of the above two operators. The composition operator first adds the appropriate wires to the input wire sets of the two component combinational relation structures, in order that they both have the same alphabet $A'' = (I'' \cup O'')$, and then takes their intersection. This corresponds to wiring two component circuits together in the obvious way. Note that we may *not* wire together outputs.

Let $T = (I, O, S, F)$ and $T' = (I', O', S', F')$ be combinational relation structures such that $(O \cap O') = \emptyset$. Let $A = (I \cup O)$ and $A' = (I' \cup O')$. Then

$$T \parallel T' = del^{-1}(A' - A)(T) \cap del^{-1}(A - A')(T')$$

- Deletion or hiding (del) hides the indicated output wires of a circuit so that these wires may no longer participate in the observable input-output behavior of the newly-packaged circuit:

  Let (I, O, S, F) be a combinational relation structure. If $D \subseteq O$, then

  $$del(D)((I, O, S, F)) = (I, (O - D), del(D)(S), del(D)(F))$$

  Recall from Chapter 2 that for $Y \subseteq \mathcal{T}^A$, $del(D)(Y) = \{del(D)(y) \mid y \in Y\}$, and for $y \in \mathcal{T}^A$, $del(D)(y)$ is the unique $x \in \mathcal{T}^{(A-D)}$ for which there exists $z \in \mathcal{T}^D$ such that $y = (x \cup z)$. Therefore, $del(D)(Y) = \{w \in \mathcal{T}^{(A-D)} \mid \exists z \in \mathcal{T}^D.(w \cup z) \in Y\}$.

  Note that even if S and F are disjoint, $del(D)(S)$ and $del(D)(F)$ need not be.

- Renaming (ren) allows us to rename the wires of a circuit (without wiring together any previously distinct wires), in order to facilitate composition in any desired topology. It is used to "name together" two wires in separate circuits that are to be combined into one by composition.

  Let r be an injective function from an alphabet A to a set B. We extend the function r naturally to sets of elements of A, and to vectors of values on A and to sets of such vectors. Thus if $C \subseteq A$, we define $r(C) = \{r(c) \mid c \in C\}$. If $w \in \mathcal{T}^A$, we define $w' = r(w)$ to be the unique element of $\mathcal{T}^{r(A)}$ such that for each $a \in A$, $w'(r(a)) = w(a)$. Similarly, if $W \subseteq \mathcal{T}^A$, then $r(W) = \{r(w) \mid w \in W\}$.

Let $T = (I, O, S, F)$ be a combinational relation structure. Let r be an injective function from $A = (I \cup O)$ to a set B. Then

$$ren(r)((I, O, S, F)) = (r(I), r(O), r(S), r(F))$$

In the following sections we prove that the class of combinational relation structures is closed under application of these algebraic operators (Section 3.3.3) and that together they form a circuit algebra (Section 3.3.4).

3.3.3 Closure under the algebraic operations

In this section we prove that the results of applying these operations to combinational relation structures are themselves combinational relation structures. This is done by proving that the required property of input-downwards-closure does indeed hold of both the resulting structure's F-set and its P-set (Section 3.3.3.1), and that the receptiveness constraint is also preserved by all the algebraic operations (Section 3.3.3.2). These proofs suffice to prove the closure of the class of combinational relation structures under the algebraic operations.

3.3.3.1 Preservation of the input-downward-closure constraint

In this section, we prove that the input-downwards-closure constraint (IDC) is preserved under the algebraic operations.

The proofs are straightforward for all the algebraic operations. It is obvious that IDC is preserved under renaming. The proofs for the remaining operations are as follows.

- IDC is preserved by the hide operation:

  Let $T = (I, O, F, P)$ and $D \subseteq O$. Let $T' = del(D)(T) = (I, O - D, del(D)(F), del(D)(P))$.

  We will prove that $del(D)(P)$ is IDC. Let $x, x' \in \mathcal{T}^I$ and $y \in \mathcal{T}^{(O-D)}$ such that $(x \cup y) \in del(D)(P)$ and $x' \leq x$. We must prove that $(x' \cup y) \in del(D)(P)$.

  By definition of T', there exists some $z \in \mathcal{T}^D$ such that $(x \cup (y \cup z)) \in P$. But then by IDC of P, it must be the case that $(x' \cup (y \cup z)) \in P$ as well.

  Therefore $(x' \cup y) \in del(D)(P)$.

  The same proof can be used to show that $del(D)(F)$ is IDC. ∎

- IDC is preserved by inverse deletion:

  Let $T = (I, O, F, P)$ and $(D \cap (I \cup O)) = \emptyset$.
  Let $T' = del^{-1}(D)(T) = ((I \cup D), O, del^{-1}(D)(F), del^{-1}(D)(P))$.

  We will prove that $del^{-1}(D)(P)$ is IDC. Let $x, x' \in \mathcal{T}^{(I \cup D)}$ and $y \in \mathcal{T}^O$ such that $(x \cup y) \in del^{-1}(D)(P)$ and $x' \leq x$. We must prove that $(x' \cup y) \in del^{-1}(D)(P)$.

By definition of T', there must exist some $z, z' \in \mathcal{T}^I$ and $v, v' \in \mathcal{T}^D$ such that $x = (z \cup v)$, $x' = (z' \cup v')$, and $(z \cup y) \in P$. Since $x' \leq x$, we know $z' \leq z$ and $v' \leq v$. But then by IDC of P, it must be the case that $(z' \cup y) \in P$ as well.

Therefore $(x' \cup y) \in del^{-1}(D)(P)$.

The same proof can be used to show that $del^{-1}(D)(F)$ is IDC. ∎

- IDC is preserved by the intersection operation:

  Let $T = (I, O, F, P)$ and $T' = (I', O', F', P')$ be combinational relation structures such that $(I \cup O) = (I' \cup O')$ and $(O \cap O') = \emptyset$. Then $(T \cap T')$ is well-defined. Let $T'' = (T \cap T') = (I'', O'', F'' = ((P \cap F') \cup (P' \cap F)), P'' = (P \cap P'))$.

  – Let $x, x' \in \mathcal{T}^{I''}, y \in \mathcal{T}^{O''}, x' \leq x$, and $(x \cup y) \in P''$.

    We must prove that $(x' \cup y) \in P''$.

    We first note that $y = (y_1 \cup y_2)$ for some $y_1 \in \mathcal{T}^{O'}$ and $y_2 \in \mathcal{T}^O$.

    We know that $(x \cup y) = ((x \cup y_1) \cup y_2) \in P$ and $(x \cup y) = ((x \cup y_2) \cup y_1) \in P'$.

    Because $x' \leq x$, therefore $(x' \cup y_1) \leq (x \cup y_1)$ and $(x' \cup y_2) \leq (x \cup y_2)$.

    Thus by IDC of P and P', $(x' \cup y) = ((x' \cup y_1) \cup y_2) \in P$ and $(x' \cup y) = ((x' \cup y_2) \cup y_1) \in P'$.

    Therefore $(x' \cup y) \in (P \cap P') = P''$. ∎

  – Let $x, x' \in \mathcal{T}^{I''}, y \in \mathcal{T}^{O''}, x' \leq x$, and $(x \cup y) \in F''$.

    We must prove that $(x' \cup y) \in F''$.

    If $(x \cup y) \in (F \cap P')$ then the argument above may be applied with P replaced by F, to prove that $(x' \cup y) \in (F \cap P') \subseteq F''$. A symmetric argument may be used if $(x \cup y) \in (F' \cap P)$. ∎

Thus input-downwards-closure of F and P is preserved by the algebraic operations.

3.3.3.2 Preservation of the receptiveness condition

In this section, we prove that the receptiveness condition of combinational relation structures is preserved by the algebraic operations. The proof of its preservation under the renaming operator is trivial and will not be given here. The proofs for the remaining operators appear below.

Because only the P-set of a combinational relation structure need be receptive, we can ignore the S and F-sets of the combinational relation structures in these proofs. Therefore we refer to a relation structure as $T = (I, O, P)$ (rather than $T = (I, O, S, F)$) in the proofs that follow.

- Receptiveness is preserved by the hide operation:

  Let $T = (I, O, P)$ and $D \subseteq O$. Let $C \subseteq P$ be total in $\mathcal{T}^I$ and have the upward-chains property.

  Let $T' = del(D)(T) = (I, O - D, del(D)(P))$.

Let $C' = del(D)(C)$. Clearly, $C' \subseteq del(D)(P)$. Because C is total in $\mathcal{T}^I$, so is C'. We proceed to prove that C' has the upward-chains property.

Let $x, x' \in \mathcal{T}^I$ such that $x \leq x'$. Let $y \in \mathcal{T}^{(O-D)}$ such that $(x \cup y) \in C'$.

We must prove that there exists $y' \in \mathcal{T}^{(O-D)}$ such that $y \leq y'$ and $(x' \cup y') \in C'$.

By definition of C', there exists $q \in \mathcal{T}^D$ such that $(x \cup (y \cup q)) \in C$. But then by the upward-chains property of C, there exists $m' \in \mathcal{T}^O$ such that $(y \cup q) \leq m'$ and $(x' \cup m') \in C$.

Let $y' = del(D)(m')$. Then we have $y' \in \mathcal{T}^{(O-D)}$ such that $y \leq y'$ and $(x' \cup y') \in del(D)(C) = C'$, so C' has the upward-chains property. ∎

- Receptiveness is preserved by inverse deletion:

  Let $T = (I, O, P)$ and $(D \cap (I \cup O)) = \emptyset$. Let $C \subseteq P$ be total (in $\mathcal{T}^I$) and have the upward-chains property.

  Let $T' = del^{-1}(D)(T) = (I \cup D, O, del^{-1}(D)(P))$. Let $C' = del^{-1}(D)(C)$. Then $C' \subseteq del^{-1}(D)(P)$. Clearly C' is total, because C is total in $\mathcal{T}^I$ and del^{-1} preserves totality. We proceed to prove that C' has the upward-chains property.

  Let $x, x' \in \mathcal{T}^{(I \cup D)}$ such that $x \leq x'$. Let $y \in \mathcal{T}^O$ such that $(x \cup y) \in C'$.

  We must prove that there exists $y' \in \mathcal{T}^O$ such that $y \leq y'$ and $(x' \cup y') \in C'$.

  By definition of T', there exist $z, z' \in \mathcal{T}^I$ and $q, q' \in \mathcal{T}^D$, such that $x = (z \cup q)$ and $x' = (z' \cup q')$. By definition of C', $(z \cup y) \in C$. Because $x \leq x'$, so too $z \leq z'$. Therefore by the upward-chains property of C, there exists $y' \in \mathcal{T}^O$ such that $y \leq y'$ and $(z' \cup y') \in C$. But then $(x' \cup y') \in C'$.

  Thus C' has the upward-chains property. ∎

- Receptiveness is preserved by intersection:

  Let $T = (I, O, P)$ and $T' = (I', O', P')$ be combinational relation structures such that $(I \cup O) = (I' \cup O')$ and $(O \cap O') = \emptyset$. Let $C \subseteq P$ be total in $\mathcal{T}^I$ and have the upward-chains property. Let $C' \subseteq P'$ be total in $\mathcal{T}^{I'}$ and have the upward-chains property.

  Let $T'' = T \cap T' = (I'', O'', P'')$. Let $C'' = C \cap C'$. Clearly, $C'' \subseteq P''$.

  We will prove that C'' is total in $\mathcal{T}^{I''}$ and has the upward-chains property. The proof that C'' is *total* is by induction over the definedness ordering $\leq$ on $\mathcal{T}^{I''}$. The induction step of this proof is the proof that C'' has the upward-chains property.

  - **Base case:** We must prove that there exists $y \in \mathcal{T}^{O''}$ such that $(\perp^{I''} \cup y) \in C''$.

    By hypothesis, there exist $y_1^{(0)} \in \mathcal{T}^{O'}$ such that $(\perp^{I'} \cup y_1^{(0)}) \in C'$, and $y_2^{(0)} \in \mathcal{T}^O$ such that $(\perp^I \cup y_2^{(0)}) \in C$.

Rearranging the presentation of each of these elements,

$$(\bot^{I''} \cup (\bot^{O} \cup y_1^{(0)})) \in C'$$

and

$$(\bot^{I''} \cup (\bot^{O'} \cup y_2^{(0)})) \in C$$

Because $\bot^{O} \leq y_2^{(0)}$ and $\bot^{O'} \leq y_1^{(0)}$, the upward-chains properties of C and C' guarantee that there exist $y_1^{(1)} \geq y_1^{(0)}$ and $y_2^{(1)} \geq y_2^{(0)}$ such that

$$(\bot^{I''} \cup (y_2^{(0)} \cup y_1^{(1)})) \in C'$$

and

$$(\bot^{I''} \cup (y_1^{(0)} \cup y_2^{(1)})) \in C$$

We can continue indefinitely in this fashion, incrementing the superscripts of the new $y_1^{(n)}$ and $y_2^{(m)}$. However, their respective domains are finite, and hence eventually we will encounter some $n \in \omega$ such that $y_1^{(n+1)} = y_1^{(n)}$ and some $m \in \omega$ such that $y_2^{(m+1)} = y_2^{(m)}$. Let $k = \min(n, m)$. Then $(\bot^{I''} \cup (y_2^{(k+1)} \cup y_1^{(k+1)})) \in (C \cap C') = C''$.

Let $y = (y_2^{(k+1)} \cup y_1^{(k+1)})$. Then $y \in \mathcal{T}^{O''}$ and $(\bot^{I''} \cup y) \in C''$. ∎

– **Induction step:** This is simply the proof that C'' has the upward-chains property. It provides the induction step for the proof that C'' is *total*.

Let $x, x' \in \mathcal{T}^{I''}$ such that $x \leq x'$. Let $y \in \mathcal{T}^{O''}$ such that $(x \cup y) \in C''$.

We must prove that there exists $y' \in \mathcal{T}^{O''}$ such that $y \leq y'$ and $(x' \cup y') \in C''$.

By definition of T'', there exist $y_1^{(0)} \in \mathcal{T}^{O'}$ and $y_2^{(0)} \in \mathcal{T}^{O}$ such that $y = (y_1^{(0)} \cup y_2^{(0)})$. By definition of C'', $((x \cup y_1^{(0)}) \cup y_2^{(0)}) \in C$ and $((x \cup y_2^{(0)}) \cup y_1^{(0)}) \in C'$.

Because $x \leq x'$, we know that $(x \cup y_1^{(0)}) \leq (x' \cup y_1^{(0)})$ and $(x \cup y_2^{(0)}) \leq (x' \cup y_2^{(0)})$. Therefore, by the upward-chains properties of C and C', there exist $y_1^{(1)} \geq y_1^{(0)}$ and $y_2^{(1)} \geq y_2^{(0)}$ such that $((x' \cup y_1^{(0)}) \cup y_2^{(1)}) \in C$ and $((x' \cup y_2^{(0)}) \cup y_1^{(1)}) \in C'$.

Ostensibly, we can continue indefinitely in this fashion, finding $y_1^{(n+1)} \geq y_1^{(n)}$ and $y_2^{(n+1)} \geq y_2^{(n)}$ such that $((x' \cup y_1^{(n)}) \cup y_2^{(n+1)}) \in C$ and $((x' \cup y_2^{(n)}) \cup y_1^{(n+1)}) \in C'$. The domains $\mathcal{T}^{O}$ and $\mathcal{T}^{O'}$ are finite, so eventually each of the increasing chains $y_i^{(0)} \leq y_i^{(1)} \leq \ldots \leq y_i^{(n)} \leq \ldots$ must reach a fixpoint. That is, there must exist minimal $m_0 \in \omega$ such that $y_1^{(m_0)} = y_1^{(m_0+1)}$ and minimal $n_0 \in \omega$ such that $y_2^{(n_0)} = y_2^{(n_0+1)}$.

Let $k = \min(m_0, n_0)$. Then $((x' \cup y_1^{(k+1)}) \cup y_2^{(k+1)}) \in C$ and $((x' \cup y_2^{(k+1)}) \cup y_1^{(k+1)}) \in C'$.

Let $y' = (y_1^{(k+1)} \cup y_2^{(k+1)})$. Then $y \leq y'$ and $(x' \cup y') \in (C \cap C') = C''$.

Therefore C'' has the upward-chains property. ∎

This concludes our proof that the class of combinational relation structures is closed under the algebraic operations.

3.3.4 Combinational relation structures form a circuit algebra

In this section we prove that the class of combinational relation structures together with the algebraic operations of composition, renaming, and deletion forms a circuit algebra. We do so by proving that all nine of the circuit algebra axioms hold for combinational relation structures.

C1 follows from Properties 2.3 and 2.4. **C2** follows from commutativity of set union and set intersection. **C3** follows from properties of function composition, because application of r and r' has been extended naturally from wire names to sets of wire names, vectors of named-wire values, and sets of such vectors.

The proof of **C4** requires the following lemmas:

Lemma 3.1 *If $x \in \mathcal{T}^{A_1}$ and $y \in \mathcal{T}^{A_2}$, and $(A_1 \cap A_2) = \emptyset$, and r is injective over $(A_1 \cup A_2)$, then*

$$r(x \cup y) = r(x) \cup r(y)$$

Proof: Obvious.

Lemma 3.2 *If $(A \cap D) = \emptyset$ and r is injective over $(A \cup D)$ and $W \subseteq \mathcal{T}^A$, then*

$$r(del^{-1}(D)(W)) = del^{-1}(r(D))(r(W))$$

Proof: Both parts of this proof rely on Lemma 3.1.

- $r(del^{-1}(D)(W)) \subseteq del^{-1}(r(D))(r(W))$:

  Let $x \in r(del^{-1}(D)(W))$. Then there exists $y \in del^{-1}(D)(W)$ such that $x = r(y)$, and there exist $v \in W$ and $z \in \mathcal{T}^D$ such that $y = (v \cup z)$.

  But then $r(y) = r(v) \cup r(z)$, and $r(v) \in r(W)$ and $r(z) \in \mathcal{T}^{r(D)}$.
  Therefore $x = r(y) \in del^{-1}(r(D))(r(W))$. ∎

- $del^{-1}(r(D))(r(W)) \subseteq r(del^{-1}(D)(W))$:

  Let $x \in del^{-1}(r(D))(r(W))$. Then there exist $v \in r(W)$ and $z \in \mathcal{T}^{r(D)}$ such that $x = (v \cup z)$. But then there exist $v' \in W$ and $z' \in \mathcal{T}^D$ such that $r(v') = v$ and $r(z') = z$. Clearly $(v' \cup z') \in del^{-1}(D)(W)$.

  But then $x = (v \cup z) = (r(v') \cup r(z')) = r(v' \cup z') \in r(del^{-1}(D)(W))$. ∎

C4 follows from Lemma 3.2 and the fact that renaming distributes across set difference, set intersection and set union of basic sets, as well as across intersection and union of sets of vectors. **C5** and **C7** are obvious. **C6** follows from Property 2.1.

The proof of **C8** relies on the following lemmas:

Lemma 3.3 *If T is a combinational relation structure, $(D \cap E) = (A \cap D) = \emptyset$ and $E \subseteq O$, then*

$$del(E)(del^{-1}(D)(T)) = del^{-1}(D)(del(E)(T))$$

This follows from Property 2.2.

Lemma 3.4 *If $W_1 \subseteq \mathcal{T}^A$ and $W_2 \subseteq \mathcal{T}^{(A \cup D)}$ and $(A \cap D) = \emptyset$, then*

$$W_1 \cap del(D)(W_2) = del(D)(del^{-1}(D)(W_1) \cap W_2)$$

Proof:

- $W_1 \cap del(D)(W_2) \subseteq del(D)(del^{-1}(D)(W_1) \cap W_2)$:

  Let $x \in W_1 \cap del(D)(W_2)$. Then there exists $z \in \mathcal{T}^D$ such that $(x \cup z) \in W_2$. Clearly $(x \cup z) \in del^{-1}(D)(W_1)$ as well. Therefore $(x \cup z) \in (del^{-1}(D)(W_1) \cap W_2)$, and so $x \in del(D)(del^{-1}(D)(W_1) \cap W_2)$. ▮

- $del(D)(del^{-1}(D)(W_1) \cap W_2) \subseteq W_1 \cap del(D)(W_2)$:

  Let $x \in del(D)(del^{-1}(D)(W_1) \cap W_2)$. Then there exists $z \in \mathcal{T}^D$ such that

  $$(x \cup z) \in (del^{-1}(D)(W_1) \cap W_2)$$

  But then $x \in del(D)(W_2)$ and $x \in W_1$. Therefore $x \in (W_1 \cap del(D)(W_2))$. ▮

Lemma 3.5 *If T_1 and T_2 are combinational relation structures, and $(A_1 \cap D_2) = (A_2 \cap D_1) = \emptyset$ and $D_1 \subseteq O_1$ and $D_2 \subseteq O_2$ and $(A_1 - D_1) = (A_2 - D_2)$, then*

$$del(D_1)(T_1) \cap del(D_2)(T_2) = del(D_1 \cup D_2)(T_1 \parallel T_2)$$

Proof:

Both sides of this equation have I-set $(I_1 \cap I_2)$ and O-set $((O_1 \cup O_2) - (D_1 \cup D_2))$. We must prove that their S-sets are identical and that their F-sets are identical. We will first prove that the S-sets are identical:

$$(del(D_1)(S_1) \cap del(D_2)(S_2)) = del(D_1 \cup D_2)(del^{-1}(A_2 - A_1)(S_1) \cap del^{-1}(A_1 - A_2)(S_2))$$

This proof may then be applied with S_1 replaced by F_1 and S_2 replaced by P_1, and again with S_1 replaced by P_1 and S_2 replaced by F_2. Because deletion distributes over the union of sets of vectors, this suffices to prove that the F-sets of the two sides of the equation are identical as well.

We will now prove that the S-sets are indeed identical:

$$\begin{array}{rcll} (del(D_1)(S_1) \cap del(D_2)(S_2)) & = & del(D_2)[del^{-1}(D_2)(del(D_1)(S_1)) \cap S_2] & Lemma\ 3.4 \\ & = & del(D_2)[del(D_1)(del^{-1}(D_2)(S_1)) \cap S_2] & Property\ 2.2 \\ & = & del(D_2)[del(D_1)[del^{-1}(D_2)(S_1) \cap del^{-1}(D_1)(S_2)]] & Lemma\ 3.4 \\ & = & del(D_1 \cup D_2)(del^{-1}(D_2)(S_1) \cap del^{-1}(D_1)(S_2)) & Property\ 2.1 \end{array}$$

By the assumptions of the lemma, $(A_2 - A_1) = D_2$ and $(A_1 - A_2) = D_1$. Therefore this is equivalent to $del(D_1 \cup D_2)(del^{-1}(A_2 - A_1)(S_1) \cap del^{-1}(A_1 - A_2)(S_2))$. ∎

Using these lemmas, we will now prove **C8**.

Let $T'' = del(D_1)(T_1) \parallel del(D_2)(T_2)$. By definition of composition,

$T'' = del^{-1}((A_2 - D_2) - (A_1 - D_1))(del(D_1)(T_1)) \cap del^{-1}((A_1 - D_1) - (A_2 - D_2))(del(D_2)(T_2))$.

By the assumptions of the theorem, $(A_1 \cap D_2) = \emptyset$ and $(A_2 \cap D_1) = \emptyset$. Thus $((A_2 - D_2) - (A_1 - D_1)) = (A_2 - (A_1 \cup D_2))$ and $((A_1 - D_1) - (A_2 - D_2)) = (A_1 - (A_2 \cup D_1))$, and Lemma 3.3 is applicable. By Lemma 3.3, $T'' = del(D_1)(del^{-1}(A_2 - (A_1 \cup D_1))(T_1)) \cap del(D_2)(del^{-1}(A_1 - (A_2 \cup D_2))(T_2))$. By Lemma 3.5, this is equivalent to

$$del(D_1 \cup D_2)[del^{-1}(A_2 - (A_1 \cup D_1))(T_1) \parallel del^{-1}(A_1 - (A_2 \cup D_2))(T_2)]$$

which, by definition of composition, is equivalent to $del(D_1 \cup D_2)(T_1 \parallel T_2)$. ∎

The proof of **C9** relies on the following lemma:

Lemma 3.6 *If $D \subseteq A$ and r' is injective over A and $r_{|(A-D)} = r'_{|(A-D)}$ and $W \subseteq \mathcal{T}^A$, then*

$$r(del(D)(W)) = del(r'(D))(r'(W))$$

Proof:

- $r(del(D)(W)) \subseteq del(r'(D))(r'(W))$:

  Let $x \in r(del(D)(W))$. Then there exists $y \in del(D)(W)$ such that $r(y) = x$. But then there exists $z \in \mathcal{T}^D$ such that $(y \cup z) \in W$.

  Thus $r'(y \cup z) \in r'(W)$, and $r'(z) \in \mathcal{T}^{r'(D)}$. Because $r'(y \cup z) = r'(y) \cup r'(z)$, we know that $r'(y) \in del(r'(D))(r'(W))$. We also know that $x = r(y) = r'(y)$.

  Therefore $x \in del(r'(D))(r'(W))$. ∎

- $del(r'(D))(r'(W)) \subseteq r(del(D)(W))$:

  Let $x \in del(r'(D))(r'(W))$. Then there exists $z \in \mathcal{T}^{r'(D)}$ such that $(x \cup z) \in r'(W)$. Because r' is injective over A, there exist unique $x' \in \mathcal{T}^{(A-D)}$ and $z' \in \mathcal{T}^D$ such that $r'(x') = x$ and $r'(z') = z$, and there exists a unique $v' \in \mathcal{T}^A$ such that $r'(v') = (x \cup z)$. Because $r'(x' \cup z') = (r'(x') \cup r'(z')) = (x \cup z)$, it must be the case that $v' = (x' \cup z')$.

  We know that $v' \in W$, because $r'(v') = r'(x' \cup z') = (r'(x') \cup r'(z')) \in r'(W)$ and v' is unique.

Therefore $x = r'(x') \in del(r'(D))(r'(W))$. ∎

C9 follows from Lemma 3.6 and the fact that $r(O - D) = r'(O - D) = r'(O) - r'(D)$.

This concludes our proof that the class of combinational relation structures together with the algebraic operations of composition, renaming, and deletion forms a circuit algebra.

3.3.5 Examples

In this section we present some examples of composite relation structures, in order to illustrate how the formal operations on circuit models reflect the intended actions on the circuits being modeled. We begin with the gated ring oscillator of Figure 3.1, and then proceed to the promised example (page 55) of a combinational circuit containing a feedback loop for which the Boolean relation is total in the set of all input value combinations. This is the circuit of Figure 6 of [92], which appears here in Figure 3.4. These examples show how the composition operator reflects the wiring together of two circuits, and illustrate the use of the deletion operator to indicate that certain nodes are not intended to be primary outputs. The second example also illustrates the usefulness of the renaming operator.

In the third example, we present requirements specifications with non-empty F-sets and show how the F-set of their composition reflects the environments in which those circuits that are implementations are expected to function correctly.

Example 3.6 Gated ring oscillator:

The circuit depicted in Figure 3.1 (page 54) consists of a nand-gate and a non-inverting buffer wired together so that the composite circuit has only a single input wire. Combinational relation structures representing the two components are presented in Examples 3.2 and 3.3 (Section 3.2.4). Their composition is the following relation structure:

$$T_{gro} = (I = \{a\}, O = \{b, c\}, S = \{\overline{a}bc, \perp_a bc, a\perp_b\perp_c, \perp_a\perp_b\perp_c, \perp_a\perp_b c\}, F = \emptyset)$$

In the discussion of the Boolean relation representation of this example circuit, the gated ring oscillator was compared to an inverter specification with input wire a and output wire b. In order to make this comparison using combinational relation structures, we must declare the wire labeled c to be an internal node rather than a primary output. This operation results in the following relation structure representation of the newly packaged circuit:

$$del(\{c\})(T_{gro}) = (\{a\}, \{b\}, \{\overline{a}b, \perp_a b, a\perp_b, \perp_a\perp_b\}, \emptyset)$$

Malik points out that the circuit depicted in Figure 3.4 may stabilize or oscillate depending on the relative delays of its components, if the input wire a has value 0 [92]. (If the input wire a has value 1, the output b must take value 1, and the other nodes of the ciruit must stabilize as well). The same observation applies if the internal wires of the circuit are initialized to $\perp$. Thus we expect a combinational relation structure representation of this circuit to admit both possibilities, and indeed this is what happens. As a result, the combinational relation structure we derive for it as Example 3.7 admits all possible input-value combinations and *in addition* represents the oscillatory behavior that would disappear in the Boolean relation representation.

Recall that on page 55 we claimed that a Boolean relation may fail to represent oscillatory behavior of a circuit containing a feedback cycle even though the relation is total in the input value combinations for the circuit. This circuit proves that claim, as the Boolean relation representation of the circuit is simply the set of those elements of the P-set of the combinational relation structure (computed in Example 3.7) which contain only Boolean wire values (0 and 1).

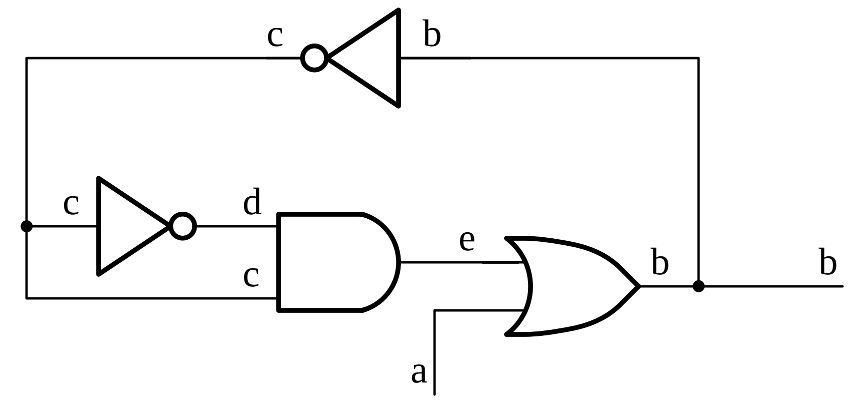

Figure 3.4: Combinational feedback loop (Fig. 6 of [92])

Example 3.7 Another combinational feedback loop (Fig. 6 of [92]):

We consider the circuit of Figure 3.4. As is clear from the structure of this circuit, if b stabilizes then so must the rest of the output nodes. Hence we would like to concentrate on the circuit's behavior by considering all the output nodes other than b to be internal nodes rather than primary outputs. And in order to simplify the representation of the circuit as we construct it from its components, we will project away (package as internal nodes) each of these nodes as early as possible. (The reader may be interested to examine the oscillation as it ripples through these internal nodes; in order to do so, one must compose the circuit representation without projecting away the internal nodes. The representation quickly becomes overly cumbersome for legible textual presentation, however, and hence is not presented here).

The standard combinational relation structure representations of our circuit's components follow.

- *or-gate with inputs labeled a and e and output labeled b :*

$$T_{or-aeb} = (\{a, e\}, \{b\}, S_{or-aeb}, \emptyset)$$

where S_{or-aeb} is the relation

$$\{\overline{a}\,\overline{e}\overline{b}, \overline{a}eb, a\overline{e}b, aeb, \overline{a}\bot_e X_b, \bot_a\overline{e}X_b, a\bot_e b, \bot_a eb, \bot_a\bot_e X_b\}$$

- *inverter with input labeled b and output labeled c :*

$$T_{inv-bc} = (\{b\}, \{c\}, \{b\overline{c}, \overline{b}c, \bot_b X_c\}, \emptyset)$$

- *inverter with input labeled c and output labeled d (presented using the ren operator):* $T_{inv-cd} = ren(r)T_{inv-bc}$, *where r is a function mapping c to d and b to c.*

- *and-gate with inputs labeled c and d and output labeled e :*

$$T_{and-cde} = (\{c, d\}, \{e\}, S_{and-cde}, \emptyset)$$

where $S_{and-cde}$ is the relation

$$\{\overline{c}\overline{d}\overline{e}, \overline{c}d\overline{e}, c\overline{d}\overline{e}, cde, c\bot_d X_e, \bot_c dX_e, \overline{c}\bot_d\overline{e}, \bot_c\overline{d}\overline{e}, \bot_c\bot_d X_e\}$$

The components may be composed in arbitrary order to obtain the same final result, because composition is associative and commutative. We have randomly chosen the following order of two-component compositions, each followed by the relevant deletion.

- $T_1 = del(\{d\})(T_{inv-cd} \,\|\, T_{and-cde}) = (\{c\}, \{e\}, \{\overline{c}\,\overline{e}, c\overline{e}, \bot_c X_e\}, \emptyset)$
- $T_2 = del(\{c\})(T_1 \,\|\, T_{inv-bc}) = (\{b\}, \{e\}, \{\overline{b}\overline{e}, b\overline{e}, \bot_b X_e\}, \emptyset)$
- $T_3 = T_2 \,\|\, T_{or-aeb} = (\{a\}, \{b, e\}, S_3, \emptyset)$ *where*

$$S_3 = \{\overline{a}\overline{b}\overline{e}, \bot_a\overline{b}\overline{e}, ab\overline{e}, \bot_a b\overline{e}, \bot_a\bot_b\overline{e}, \overline{a}\bot_b\bot_e, \bot_a\bot_b\bot_e\}$$

- $T_4 = del(\{e\})(T_3) = (\{a\}, \{b\}, \{ab, \overline{a}\overline{b}, \overline{a}\bot_b, \bot_a X_b\}, \emptyset)$

T_4 is our combinational relation structure representation of the circuit of Figure 3.4. It incorporates both the oscillating and the stabilizing possibilities of the circuit, as discussed above.

In the following example we illustrate the behavior of F-sets under composition of relation structures.

Example 3.8 *Consider the specification of Example 3.4. It describes an inverter that we expect to place only in an environment in which its input stabilizes. This expectation is indicated by its F-set.*

$$T_0 = (\{a\}, \{b\}, S_0 = \{a\overline{b}, \overline{a}b\}, F_0 = \{\bot_a X_b\})$$

We wish to specify a circuit having input wire labeled a and output wire labeled c, which behaves like two such inverters in series.

We expect such a specification to be identical to that of a stable-input expecting non-inverting buffer. The buffer specification is represented by the following combinational relation structure:

$$T_{buf} = (\{a\}, \{c\}, S_{buf} = \{ac, \overline{a}\,\overline{c}\}, F_{buf} = \{\perp_a X_c\})$$

In order to derive our specification, we compose T_0 above with $T_1 = ren(r)(T_0)$, where the function r maps the wirename b to c and the wirename a to b, and subsequently hide the b wire so as to indicate it is not to be considered a primary output:

$$T_1 = ren(r)(T_0) = (\{b\}, \{c\}, S_1 = \{b\overline{c}, \overline{b}c\}, F_1 = \{\perp_b X_c\})$$

$$T_{Spec} = del(\{b\})(T_0 \parallel T_1) = (\{a\}, \{c\}, S_{Spec} = \{ac, \overline{a}\,\overline{c}\}, F_{Spec} = \{\perp_a X_c\})$$

As we expect, $T_{Spec} = T_{buf}$.

Now consider what happens when we compose T_0 with T_{buf}, suitably renamed so that the composition forms a loop. First we formalize the required renaming:

$$ren(r'')(T_{buf}) = (I = \{b\}, O = \{a\}, S = \{ba, \overline{b}\overline{a}\}, F = \{\perp_b X_a\})$$

The composition described yields the following combinational relation structure:

$$T_0 \parallel ren(r'')(T_{buf}) = (I = \emptyset, O = \{a, b\}, S = \emptyset, F = \{\perp_a \perp_b\})$$

This composition forms what would be a simple ungated ring oscillator specification – except for the fact that the only possible behavior of the circuit, which is to oscillate, is a failure behavior. This correctly reflects the fact that neither of the two component specifications specifies a circuit that is expected to function correctly in an environment in which its input does not stabilize.

Chapter 4

Verification and Substitution

4.1 Introduction

In this chapter we present the theory that underlies our algorithms for formal hierarchical verification and substitution of combinational circuits. The theoretical framework provides an overview of the verification procedures, but does not delve into all of the details necessary to implement them. The missing algorithms are special cases or parts of the corresponding algorithms for formal verification of synchronous circuits. These algorithms will be presented in Chapter 6.

In this chapter, we define a formal relation between combinational models that corresponds to one being a correct implementation of the other (considered as a specification). We define a combinational circuit representation to correctly implement a specification if the implementation can be safely substituted for the specification. A $\perp$ value on an output wire of a specification under a particular set of input circumstances actually allows the implementation to output *any* value on that wire under those circumstances. We provide a decision procedure for the formal relation that holds between a specification and a correct implementation thereof. We also use this relation to fully characterize the set of allowed substitutions for a subcircuit in a combinational circuit or specification, a problem relevant to the logic synthesis domain.

The organization of the chapter is as follows. In section 4.2, we present the full definition of when a combinational relation structure describes an acceptable implementation of a specification, and outline a decision procedure for this relation. In Section 4.3 we use these semantics to fully characterize the set of allowed substitutions for a subcircuit in a combinational circuit or specification. Examples are provided at each step of the development.

4.2 Verification for combinational circuit models

4.2.1 Introduction

In this section we define what it means for a circuit to correctly implement a specification. A combinational relation structure is a correct implementation of another (considered as a specification) if the formal relation of *conformance* holds between them: one relation structure *conforms* to another if the first may be safely substituted for the second in any context. The notion of failure behaviors is critical to the concept of safe substitution, and ties together our understanding of correct implementation and expected environments.

We are interested in hierarchical verification. Thus we require that the conformance relation meet the following *compositionality condition:* if two circuit components conform to two requirements specifications ($T_1 \preceq T_1'$ and $T_2 \preceq T_2'$), then it must be the case that the composition of the implementations conforms to the composition of the specifications ($T_1 \parallel T_2 \preceq T_1' \parallel T_2'$). Similarly, it must be the case that the conformance relation is preserved under hiding of output wires. These properties are called *monotonicity* of the operations with respect to conformance.

In the following subsections, we define safe substitutability for combinational relation structures and outline a decision procedure for the conformance relation. We define the maximal safe environment of a circuit specification, and show that its use in deciding conformance guarantees the correct interpretation of a $\perp$ value on any of the output wires of the specification. The method applies both to verifying that a circuit (represented by a combinational relation structure) is a correct implementation of a specification relation structure (Section 4.2.5), and to verifying that substituting a circuit into a given location in a predetermined circuit results in a full circuit that exhibits only desired behavior (Section 4.3).

4.2.2 Correct implementation: the conformance relation

We say that a combinational relation structure *conforms* to another if and only if the first may be safely substituted for the second in any legal environment. In order to define this formally, we first define an expression context. An expression context is a circuit algebra expression with a single free variable α [61]. The (I, O)-type of α must be known, and is referred to as (I_α, O_α). If we replace α in this expression by a combinational relation structure $(I_\alpha, O_\alpha, S, F)$, the result denotes a combinational relation structure.

If $T = (I, O, S, F)$ and $T' = (I, O, S', F')$, we say that T conforms to T' if and only if T may be *safely substituted* for T' in every expression context such that $I_\alpha = I$ and $O_\alpha = O$. The formal definition follows.

We say a combinational relation structure $T = (I, O, S, F)$ is *failure-free* if $F = \emptyset$. Safe substitution is substitution that preserves failure freedom [61]:

Let $T = (I, O, S, F)$ and $T' = (I, O, S', F')$ both be combinational relation structures. Formally, we say that T conforms to T', written $T \preceq T'$, if and only if for all expression contexts $E[\alpha_{I,O}]$,

$$E[T'] \text{ is failure-free } \Longrightarrow E[T] \text{ is failure-free}$$

As in asynchronous trace theory, we can reduce this definition to an equivalent one that uses only a restricted form of expression context. In order to simplify use of this equivalent definition of conformance, we first introduce some further notation.

We say that $E = (I_E, O_E, S_E, F_E)$ is a *legal environment of* a combinational relation structure $T = (I, O, S, F)$ if and only if E is a combinational relation structure and $I = O_E$ and $O = I_E$.

Theorem 4.1 *Let $T = (I, O, S, F)$ and $T' = (I, O, S', F')$ be combinational relation structures. Then $T \preceq T'$ if and only if for all legal environments $E = (O, I, S_E, F_E)$ of T and T',*

$$(T' \cap E) \textit{ is failure-free } \Longrightarrow (T \cap E) \textit{ is failure-free}$$

Proof: Exactly as for the corresponding theorem in asynchronous trace theory: Lemma 4.2 on p. 58 of [61]. This proof only uses the circuit algebra axioms and lemmas. ∎

Yet another condition equivalent to conformance uses composition ($\|$) rather than intersection ($\cap$). The statement of this condition's equivalence to the other is Theorem 4.4 below. We first state and prove some supporting lemmas.

Lemma 4.2 *Let $T_1 = (I_1, O_1, S_1, F_1)$ and $T_2 = (I_2, O_2, S_2, F_2)$ be combinational relation structures, and let $D \subseteq O_2$, such that $(I_1 \cup O_1) = ((I_2 \cup O_2) - D)$ and $(O_1 \cap O_2) = \emptyset$. Then*

$$del^{-1}(D)(T_1) \cap T_2 \textit{ is failure-free} \Longleftrightarrow T_1 \cap del(D)(T_2) \textit{ is failure-free}$$

Proof:

In order to prove this lemma, it suffices to prove that for all sets $X \subseteq \mathcal{T}^A$ and $Y \subseteq \mathcal{T}^{(A \cup D)}$, where A and D are disjoint,

$$(X \cap del(D)(Y)) = \emptyset \Longleftrightarrow (del^{-1}(D)(X) \cap Y) = \emptyset$$

(Just substitute F_1 for X and P_2 for Y to get half the lemma, and substitute P_1 for X and F_2 for Y to get the other half). But this follows from Lemma 3.4 and the fact that for $W \subseteq \mathcal{T}^A$, $del(D)(W) = \emptyset \Longleftrightarrow W = \emptyset$. (Recall that if $W \neq \emptyset$ and $A = D$, then $del(D)(W)$ contains one element, the empty function). ∎

Lemma 4.3 *Let $T = (I, O, S, F)$ and $T' = (I, O, S', F')$ be combinational relation structures. Let $(D \cap (I \cup O)) = \emptyset$. Then*

$$T \preceq T' \Longleftrightarrow del^{-1}(D)(T) \preceq del^{-1}(D)(T')$$

Proof:

- $(\Longrightarrow)$: Let $T \preceq T'$.

  Let E be a combinational relation structure such that $I_E = O$ and $O_E = (I \cup D)$ and such that $del^{-1}(D)(T') \cap E$ is failure-free. We must prove that $del^{-1}(D)(T) \cap E$ is failure-free.

  Let $E_0 = del(D)(E)$, which is a legal environment of T'. By Lemma 4.2, $T' \cap E_0$ is failure-free.

  By assumption, $T \preceq T'$. Therefore, $T \cap E_0$ is failure-free. By Lemma 4.2, therefore,

  $$del^{-1}(D)(T) \cap E \text{ is failure-free}$$

  Therefore, $del^{-1}(D)(T) \preceq del^{-1}(D)(T')$. ∎

- $(\Longleftarrow)$: Let $del^{-1}(D)(T) \preceq del^{-1}(D)(T')$.

  Let $E = (I_E, O_E, S_E, F_E)$ be a legal environment of T and T' such that $T' \cap E$ is failure-free. We must prove that $T \cap E$ is failure-free as well.

  Let $E_1 = (I_E, O_E \cup D, del^{-1}(D)(S_E), del^{-1}(D)(F_E))$, which is not only a combinational relation structure, but a legal environment of $del^{-1}(D)(T')$ as well. Note that despite its S and F-set definitions, E_1 is *not* identical to $del^{-1}(D)(E)$, because its new wires D are output wires rather than input wires. However, $del(D)(E_1) = E$.

  Set manipulations involving the F-set of E_1 reveal that $del^{-1}(D)(T') \cap E_1$ is failure-free. By assumption, $del^{-1}(D)(T) \preceq del^{-1}(D)(T')$. Therefore, $del^{-1}(D)(T) \cap E_1$ is failure-free. By Lemma 4.2 (because $del(D)(E_1) = E$), $del^{-1}(D)(T) \cap E_1$ is failure-free if and only if $T \cap E$ is failure-free. Therefore, $T \cap E$ must be failure-free.

  Therefore, $T \preceq T'$. ∎

Theorem 4.4 *Let $T = (I, O, S, F)$ and $T' = (I, O, S', F')$ be combinational relation structures. Then $T \preceq T'$ if and only if for all combinational relation structures $E = (I_E, O_E, S_E, F_E)$ such that $(O \cap O_E) = \emptyset$,*

$$(T' \parallel E) \textit{ is failure-free} \Longrightarrow (T \parallel E) \textit{ is failure-free}$$

Proof:

- $(\Longrightarrow)$: Let $T \preceq T'$.

  Let E be a combinational relation structure such that $(O_E \cap O) = \emptyset$. Let $T' \parallel E$ be failure-free.

  We must prove that $T \parallel E$ is failure-free as well.

  Let $E_0 = del^{-1}(A - A_E)(E)$. By definition of the composition operator,

  $$(T' \parallel E) = del^{-1}(A_E - A)(T') \cap E_0$$

By Lemma 4.3, $del^{-1}(A_E - A)(T) \preceq del^{-1}(A_E - A)(T')$. Therefore, $del^{-1}(A_E - A)(T) \cap E_0$ is failure-free. But $del^{-1}(A_E - A)(T) \cap E_0 = T \parallel E$.

Therefore $T \parallel E$ is failure-free. ∎

- $(\Longleftarrow)$: Assume that for all combinational relation structures $E = (I_E, O_E, S_E, F_E)$ such that $(O \cap O_E) = \emptyset$, if $(T' \parallel E)$ is failure-free then $(T \parallel E)$ is failure-free as well.

  Let E be a combinational relation structure such that $I_E = O$ and $O_E = I$, and such that $T' \cap E$ is failure-free. Then $T' \cap E = T' \parallel E$, and so $T \parallel E = T \cap E$ is failure-free as well.

  Therefore $T \preceq T'$. ∎

We will use both of these equivalent definitions of conformance in the proofs that follow.

The algebraic operators rename, hide and composition are all monotonic with respect to conformance, as required for sound hierarchical verification. Monotonicity is obvious for the case of the renaming operator. We will prove it for the other two.

Theorem 4.5 *Let $T = (I, O, S, F)$ and $T' = (I, O, S', F')$ be combinational relation structures. Let $D \subseteq O$. Then*

$$T \preceq T' \Longrightarrow del(D)(T) \preceq del(D)(T')$$

Proof: Let $T \preceq T'$. Then for all legal environments $E = (O, I, S_E, F_E)$ of T and T', $(T' \cap E)$ being failure-free implies that $(T \cap E)$ is as well.

Let $E' = (O - D, I, S_{E'}, F_{E'})$ be a combinational relation structure such that $(del(D)(T') \cap E')$ is failure-free. We must prove that $(del(D)(T) \cap E')$ is failure-free.

Let $E_0 = del^{-1}(D)(E')$. Because E' is a combinational relation structure, E_0 is as well. By Lemma 4.2, because $(del(D)(T') \cap E')$ is failure-free, it must be the case that $(T' \cap E_0)$ is failure-free as well. But then by definition of $T \preceq T'$ it must be the case that $(T \cap E_0)$ is failure-free. Invoking Lemma 4.2 again, we conclude that $(del(D)(T) \cap E')$ is failure-free.

As E' was an arbitrary legal environment of $del(D)(T)$ and $del(D)(T')$, we have proved by Theorem 4.1 that $del(D)(T) \preceq del(D)(T')$. ∎

Theorem 4.6 *Let $T = (I, O, S, F)$ and $T' = (I, O, S', F')$ be combinational relation structures. Let T'' be any combinational relation structure such that $(O'' \cap O) = \emptyset$. Then*

$$T \preceq T' \Longrightarrow T \parallel T'' \preceq T' \parallel T''$$

Proof: Let $T \preceq T'$. Then for all combinational relation structures E such that $(O_E \cap O) = \emptyset$, $(T' \parallel E)$ being failure-free implies that $(T \parallel E)$ is as well.

Let $E' = (O_{E'}, I_{E'}, S_{E'}, F_{E'})$ be a combinational relation structure such that $((T' \parallel T'') \parallel E')$ is failure-free. We must prove that $((T \parallel T'') \parallel E')$ is failure-free.

Let $E_0 = T'' \parallel E'$. Because E' and T'' are both combinational relation structures, E_0 is also a combinational relation structure. By associativity of composition (a circuit algebra axiom), $((T' \parallel T'') \parallel E') = (T' \parallel E_0)$. But then by definition of $T \preceq T'$ it must be the case that $(T \parallel E_0)$ is failure-free. Invoking associativity of composition again, we conclude that $((T \parallel T'') \parallel E')$ is failure-free. As E' was an arbitrary combinational relation structure, we have proved by Theorem 4.4 that $T \parallel T'' \preceq T' \parallel T''$. ∎

Recall the preorder $\sqsubseteq$ over combinational relation structures:

$$(I, O, F, P) \sqsubseteq (I, O, F', P') \text{ iff } ((F \subseteq F') \wedge (P \subseteq P'))$$

This relation is stronger than conformance.

Lemma 4.7 *If T and T' are both combinational relation structures, then $T \sqsubseteq T' \Longrightarrow T \preceq T'$.*

Proof: Follows from Theorem 4.1 by set manipulations.

4.2.3 Conformance equivalence

Conformance is a preorder on combinational relation structures. It is not a partial order. We define $T \sim T'$ to mean that $T \preceq T'$ and $T' \preceq T$. This equivalence ("conformance equivalence") induces a partial order on equivalence classes of combinational relation structures.

Lemma 4.8 *If T and T' are both combinational relation structures, then*

$$T \sim_{\sqsubseteq} T' \Longrightarrow T \sim T'$$

Proof: Two applications of Lemma 4.7.

We seek a unique representative for each conformance equivalence class. Such a representative will enable us to treat $\preceq$ as a partial order. More precisely, we seek a procedure that when applied to any element of any conformance equivalence class will produce that class' unique representative. The form of such a procedure will aid us in clarifying the extent of each conformance equivalence class.

Consider the case of a combinational relation structure T that can force *every one* of its legal environments into a failure. As far as safe substitution is concerned, we might as well simply mark all entries of T's P-set as failures. This is because T will have a non-failure-free composition with every one of its legal environments whether or not we make this addition to the F-set of T.

Similarly, we note that if a relation structure is going to force every environment into a failure, its F-set might as well be universal. Again, this is because the relation structure T will have a non-failure-free composition with every one of its legal environments irrespective of the particular behaviors that the composite model exhibits. These observations lead to the definition of combinational autofailures and the autofailure manifestation process.

We say a relation structure $T = (I, O, S, F)$ is a *combinational autofailure* if and only if it has no failure-free composition with any of its legal environments. Formally, for all combinational relation structures $E = (O, I, S_E, F_E)$, $E \cap T$ is not failure-free. Because the set of all legal environments of T includes those with empty F-sets, this condition is equivalent to requiring that for all legal environments E of T, $(F \cap P_E)$ is not empty. In contrast to the definition of autofailures in asynchronous trace theory, there is an extra complication because the environment of a combinational relation structure can respond instantaneously. (Note also that a combinational autofailure is a relation structure, whereas in asynchronous trace theory, an autofailure is an element of the P-set of a trace structure). There is, however, an effective procedure for determining whether or not T is a combinational autofailure. That procedure is described in Chapter 6.

We define the operation of *autofailure manifestation* as follows:

$$afm((I, O, S, F)) = \begin{cases} (I, O, S, \mathcal{T}^{(I \cup O)}) & \text{if } T \text{ is a combinational autofailure} \\ (I, O, S, F) & \text{otherwise} \end{cases}$$

The *afm* operator expands the F-set of its combinational autofailure operand to the maximal set $\mathcal{T}^{(I \cup O)}$. However, an operand relation structure that is not a combinational autofailure is left unchanged by the operator.

Lemma 4.9 *Let T be a combinational relation structure. Then $afm(T)$ is too.*

Proof:

Let $T = (I, O, S, F)$ be a combinational relation structure. If T is not a combinational autofailure, then $afm(T) = T$ is a combinational relation structure by assumption.

If T is a combinational autofailure, then $S, F \subseteq \mathcal{T}^{(I \cup O)}$, and the F and P-sets of $afm(T)$ are both $\mathcal{T}^{(I \cup O)}$. Clearly $\mathcal{T}^{(I \cup O)}$ is both input-downward-closed and receptive. ∎

Theorem 4.10 *Let T be a combinational relation structure. Then $T \sim afm(T)$.*

Proof:

If T is not a combinational autofailure, then $afm(T) = T$ and so clearly $T \sim afm(T)$.

We concentrate on the case in which T is a combinational autofailure. For every combinational relation structure T, $T \sqsubseteq afm(T)$. Thus by Lemma 4.7, $T \preceq afm(T)$.

We must prove that $afm(T) \preceq T$ for T a combinational autofailure.

By definition of a combinational autofailure, there exists no legal environment E of T such that $T \cap E$ is failure-free. By Theorem 4.1, it is thus vacuously true that $afm(T) \preceq T$.

Thus $afm(T) \preceq T$ and so we have proved that $afm(T) \sim T$. ∎

A combinational relation structure may have S and F sets which are not disjoint. An input-output value combination which appears in both these sets represents a behavior that is nondeterministically either a success or a failure. Composition with any other combinational relation structure admitting this behavior will be non-failure-free irrespective of whether the behavior is also in S. Hence in attempting to delete extraneous information from T in such a way as to maintain the soundness of our verification, we require that such an input-output value combination be considered solely as a failure.

We call the process of setting $S_{new} = (S - F)$ *failure-exclusion*, and the resulting relation structure $fe(T)$.

Lemma 4.11 *Let $T = (I, O, S, F)$ be a combinational relation structure. Then $fe(T)$ is too.*

Proof:

Neither the F-set nor the P-set of T is affected by the fe operator. Therefore the required properties of these two sets are preserved by application of this operator. ∎

Theorem 4.12 *Let $T = (I, O, S, F)$ be a combinational relation structure. Then $T \sim fe(T)$.*

Proof:

Neither the F-set nor the P-set of T is affected by the fe operator. Therefore, $T \sqsubseteq fe(T)$ and $fe(T) \sqsubseteq T$. Thus, by Lemma 4.8, $T \sim fe(T)$. ∎

We define output-upwards-closure in the obvious way by analogy to input-downwards-closure (IDC). Given predetermined input and outputs sets I and O, respectively, we say that a set $W \subseteq \mathcal{T}^{(I \cup O)}$ is *output-upwards-closed* precisely when

$$\forall x \in \mathcal{T}^I . \forall y, y' \in \mathcal{T}^O . [[(x \cup y) \in W \wedge y \leq y'] \Longrightarrow (x \cup y') \in W]$$

We say a combinational relation structure $T = (I, O, F, P)$ is output-upwards-closed if and only if both its F and P sets are.

In addition to the *property* of output-upward-closure we define an *operator* OUC on combinational relation structures. If $T = (I, O, S, F)$ is a combinational relation structure, we define

$$OUC(T) = (I, O, OUC_{I,O}(S), OUC_{I,O}(F))$$

where $OUC_{I,O}(W)$ is the set $W \subseteq \mathcal{T}^{(I \cup O)}$ together with the minimal set of additional behaviors (elements of $\mathcal{T}^{(I \cup O)}$) necessary to make the resulting set output-upwards-closed, for W any of S, F, or P. Clearly, $OUC_{I,O}(S) \cup OUC_{I,O}(F) = OUC_{I,O}(P)$. We prove that the result of applying this operator to a combinational relation structure is itself a combinational relation structure, and that all conformance equivalence classes are closed under this operation.

Lemma 4.13 *If T is a combinational relation structure, then $OUC(T)$ is too.*

Proof:

Let $T = (I, O, S, F)$ be a combinational relation structure. We prove that $OUC_{I,O}(P)$ is receptive, and that $OUC_{I,O}(F)$ and $OUC_{I,O}(P)$ are input-downward-closed:

- $OUC_{I,O}(P)$ is receptive:

  By definition of the $OUC_{I,O}$ operator, $P \subseteq OUC_{I,O}(P)$. Hence we can simply maintain the old C, a subset of P with the upward-chains property, as the new C (for $OUC(T)$). Therefore $OUC_{I,O}(P)$ is receptive. ▮

- $OUC_{I,O}(F)$ and $OUC_{I,O}(P)$ are input-downward-closed:

  We prove that application of the $OUC_{I,O}$ operator to any input-downward-closed set which is a subset of $\mathcal{T}^{(I \cup O)}$ *preserves* that set's input-downward-closure property.

  Let $Q \subseteq \mathcal{T}^{(I \cup O)}$ be input-downward-closed. Let $x, x' \in \mathcal{T}^I$ such that $x' \leq x$. Let $y \in \mathcal{T}^O$ such that $(x \cup y) \in OUC_{I,O}(Q)$.

  By definition of the $OUC_{I,O}$ operator, there exists $y' \leq y$ such that $(x \cup y') \in Q$. But then by input-downward-closure of Q, $(x' \cup y') \in Q$, and therefore $(x' \cup y) \in OUC_{I,O}(Q)$. ▮

Theorem 4.14 *Let T be a combinational relation structure. Then $T \sim OUC(T)$.*

Proof:

By definition of the OUC operator, $T \sqsubseteq OUC(T)$. Thus by Lemma 4.7, $T \preceq OUC(T)$.

We must prove that $OUC(T) \preceq T$.

Let $E = (O, I, F_E, P_E)$ be a legal environment of T such that $T \cap E$ is failure-free. We must prove that $OUC(T) \cap E$ is also failure-free.

Say it is not. Then there exists w an element of the F-set of $OUC(T) \cap E$, that is,

$$w \in ((OUC_{I,O}(F) \cap P_E) \cup (OUC_{I,O}(P) \cap F_E))$$

Assume without loss of generality that $w \in (OUC_{I,O}(F) \cap P_E)$.

Let $x \in \mathcal{T}^I$ and $y \in \mathcal{T}^O$ such that $w = (x \cup y)$. Then by definition of the $OUC_{I,O}$ operator, there exists $y' \leq y$ such that $(x \cup y') \in F$. By input-downward-closure of P_E, $(x \cup y') \in P_E$ as well. But then $(x \cup y') \in (F \cap P_E) \subseteq ((F \cap P_E) \cup (P \cap F_E))$, which is $(T \cap E)$'s failure-set, which by assumption is empty. Therefore there can be no such $w \in (OUC_{I,O}(F) \cap P_E)$, and so $(OUC_{I,O}(F) \cap P_E) = \emptyset$.

By the same argument, there can be no $w \in (OUC_{I,O}(P) \cap F_E)$, and so $(OUC_{I,O}(P) \cap F_E) = \emptyset$. Therefore $((OUC_{I,O}(F) \cap P_E) \cup (OUC_{I,O}(P) \cap F_E)) = \emptyset$. In other words, $OUC(T) \cap E$ is failure-free.

Therefore $OUC(T) \preceq T$, and so $T \sim OUC(T)$. ▮

We say that a combinational relation structure $T = (I, O, S, F)$ is *canonicalized* if $S \cap F = \emptyset$, F and P are output-upward-closed, and either $F = \mathcal{T}^{(I \cup O)}$ or T is not a combinational autofailure. In section 4.2.5, we will prove that a canonicalized combinational relation structure is unique in its conformance equivalence class. For now, we prove only the fairly obvious statement that the procedures outlined above, if applied in such an order that no operator undoes the desired effects of any other operator previously applied, do indeed result in a canonicalized relation structure. In addition, we prove that application of these operators to an already canonicalized relation structure T has no effect.

Theorem 4.15 *Let T be a combinational relation structure. Then*

$$fe(afm(OUC(T))) \text{ and } fe(OUC(afm(T)))$$

are both canonicalized combinational relation structures that are conformance equivalent to T.

Proof:
Let T be a combinational relation structure. Let $T_1 = OUC(T), T_2 = afm(T_1)$, and $T_3 = fe(T_2)$.
Let $T_4 = afm(T), T_5 = OUC(T_4)$, and $T_6 = fe(T_5)$.
We must prove

- T_3 is a combinational relation structure, $T_3 \sim T$, and T_3 is canonicalized, and
- T_6 is a combinational relation structure, $T_6 \sim T$, and T_6 is canonicalized.

The proofs follow.

- By Lemma 4.13, T_1 is a combinational relation structure. Hence by Lemma 4.9, T_2 is too. Thus, by Lemma 4.11, T_3 is a combinational relation structure.

  We utilize the transitivity of $\sim$ to prove that $T_3 \sim T$. By Theorem 4.14, $T \sim T_1$. By Theorem 4.10, $T_1 \sim T_2$. By Theorem 4.12, $T_2 \sim T_3$. Thus by transitivity of the relation $\sim$, we have proved that $T \sim T_3$.

  In order to prove that T_3 is canonicalized, we prove that each of the relevant criteria holds:

  - F_3 and P_3 are output-upwards-closed:

    By definition, T_1 is output-upwards-closed. If T_1 is a combinational autofailure, then $F_2 = P_2 = \mathcal{T}^{(I \cup O)}$, which is certainly output-upwards-closed. If T_1 is not a combinational autofailure, then $F_2 = F_1$ and $P_2 = P_1$, so T_2 is output-upwards-closed because T_1 is. Finally, F_3 and P_3 are output-upward-closed because by definition of failure exclusion, $F_3 = F_2$ and $P_3 = P_2$. ∎
  - $(S_3 \cap F_3) = \emptyset$:

    This follows directly from the definition of failure exclusion. ∎
  - Either $F_3 = \mathcal{T}^{(I \cup O)}$ or T_3 is not a combinational autofailure:

    By Theorem 4.1, because $T \sim T_3$ it must be the case that T is a combinational autofailure if and only if T_3 is. Similarly, T is a combinational autofailure if and only if T_1 is.

    Thus, if T_3 is a combinational autofailure, then so is T_1. Hence in this case, by definition of autofailure manifestation, $F_2 = P_2 = \mathcal{T}^{(I \cup O)}$. Because failure exclusion does not affect the F-set of its operand relation structure, $F_3 = \mathcal{T}^{(I \cup O)}$ as well. ∎

- By Lemma 4.9, T_4 is a combinational relation structure. Hence by Lemma 4.13, T_5 is too. Thus, by Lemma 4.11, T_6 is a combinational relation structure.

  We utilize the transitivity of $\sim$ to prove that $T_6 \sim T$. By Theorem 4.10, $T \sim T_4$. By Theorem 4.14, $T_4 \sim T_5$. By Theorem 4.12, $T_5 \sim T_6$. Thus by transitivity of the relation $\sim$, we have proved that $T \sim T_6$.

  In order to prove that T_6 is canonicalized, we prove that each of the relevant criteria holds:

  - F_6 and P_6 are output-upwards-closed:

    By definition, T_5 is output-upwards-closed. By definition of failure exclusion, $F_6 = F_5$ and $P_6 = P_5$. Therefore F_6 and P_6 are output-upward-closed ∎

  - $(S_6 \cap F_6) = \emptyset$:

    This follows directly from the definition of failure exclusion. ∎

  - Either $F_6 = \mathcal{T}^{(I \cup O)}$ or T_6 is not a combinational autofailure:

    By Theorem 4.1, because $T \sim T_6$ it must be the case that T is a combinational autofailure if and only if T_6 is.

    Thus, if T_6 is a combinational autofailure, then so is T. Hence in this case, by definition of autofailure manifestation, $F_4 = P_4 = \mathcal{T}^{(I \cup O)}$. Because $OUC(\mathcal{T}^{(I \cup O)}) = \mathcal{T}^{(I \cup O)}$, $F_5 = \mathcal{T}^{(I \cup O)}$ too. And finally, because failure exclusion does not affect the F-set of its operand relation structure, $F_6 = \mathcal{T}^{(I \cup O)}$ as well. ∎

QED Theorem 4.15

In addition to being canonicalized and conformance equivalent to T, these two derived relation structures are identical to each other and to T *if T was already canonicalized.*

Lemma 4.16 *If T is a canonicalized combinational relation structure, then*

$$fe(afm(OUC(T))) = fe(OUC(afm(T))) = T$$

Proof: Let T be a canonicalized combinational relation structure. Then F and P are output-upward-closed (that is, $F = OUC_{I,O}(F)$ and $P = OUC_{I,O}(P)$). Therefore, $T = OUC(T)$.

Similarly, because T is canonicalized, either $F = \mathcal{T}^{(I \cup O)}$ or T is not a combinational autofailure. Therefore $afm(T) = T$. And finally, because T is canonicalized, its S and F-sets are disjoint. Therefore $fe(T) = T$.

Because of these three facts, $fe(afm(OUC(T))) = fe(afm(T)) = fe(T) = T$ and

$$fe(OUC(afm(T))) = fe(OUC(T)) = fe(T) = T.$$ ∎

4.2.4 The maximal safe environment of a circuit

In order to derive a decision procedure for conformance, we define the maximal environment with which a circuit (represented by a combinational relation structure) may be safely composed. However, not all combinational relation structures have maximal environments that are themselves combinational relation structures. There exists a particular class of combinational relation structures whose maximal environment has an empty P-set, and therefore is not receptive. We require an exception to the conformance decision procedure to handle the case of those combinational relation structures whose maximal safe environment is not a combinational relation structure.

In this section, we define the maximal safe environment of a combinational relation structure and begin to clarify how we will decide conformance if one or both of the combinational relation structures being compared does not have a maximal safe environment which is a combinational relation structure. In Section 4.2.5, we will present the final theorems necessary to support our conformance decision procedure, and discuss the decision procedure itself.

We define a mirroring operation over the *canonicalized* combinational relation structures only. Mirroring a combinational relation structure involves swapping its inputs and outputs, complementing its F and P sets, and then swapping them. The S-set is left intact. Formally, for $T = (I, O, S, F)$ a canonicalized combinational relation structure, the mirror of T is

$$mir(T) = (O, I, S, (\mathcal{T}^{(I \cup O)} - P)) = (O, I, S, \overline{P})$$

Note that the P-set of $mir(T)$ is $(S \cup \overline{P}) = \overline{F}$: the equality holds because $(S \cap F) = \emptyset$ for canonicalized T.

Intuitively, $mir(T)$ is the maximal environment of canonicalized T such that $T \parallel mir(T)$ is failure-free. However, if $S = \emptyset$ then $F = P$, so that T must be a combinational autofailure. In this case $mir(T)$ is not a combinational relation structure, as it has an empty P-set (which is therefore not receptive). We note that the mirror of a canonicalized combinational relation structure whose S-set is nonempty is itself a canonicalized combinational relation structure (Lemma 4.17), and that their composition is failure free (Lemma 4.18).

Lemma 4.17 *Let $T = (I, O, S, F)$ be a canonicalized combinational relation structure such that $S \neq \emptyset$. Then $mir(T)$ is also a canonicalized combinational relation structure.*

Proof: Let T be a canonicalized combinational relation structure. Let $T' = mir(T)$. We must prove that T' is a combinational relation structure and that it is canonicalized.

As a preliminary to proving that $P' = \overline{F}$ obeys the receptiveness constraint, we note that for every combinational relation structure $T_0 = (I, O, F_0, P_0)$ and legal environment $E = (O, I, F_E, P_E)$ of T_0,

$$\begin{aligned}(T_0 \cap E) \text{ is failure-free} &\iff (P_0 \cap F_E) = \emptyset \wedge (F_0 \cap P_E) = \emptyset \\ &\iff F_E \subseteq \overline{P_0} \wedge P_E \subseteq \overline{F_0}\end{aligned}$$

The equations above state that

$$T \cap E \text{ is failure-free if and only if } F_E \subseteq \overline{P} \text{ and } P_E \subseteq \overline{F}$$

We must prove that there exists $C \subseteq \overline{F}$ that is total (in $\mathcal{T}^O$) and that has the upward-chains property (in (O, I)). The proof is by contradiction. Say that there exists no such $C \subseteq \overline{F}$. Then clearly no $X \subseteq \overline{F}$ can contain such a C either. Thus, by this containment-upward-closure property of the receptiveness constraint, there exists no legal environment E of T such that $T \cap E$ is failure-free. This is because $T \cap E$ is failure-free only if $P_E \subseteq \overline{F}$. Thus there can exist no $\mathcal{T}^O$-total $C \subseteq P_E$ having the upward-chains property in (O, I). Therefore P_E does not obey the receptiveness constraint, and so E is not a combinational relation structure after all. By this argument, there exists no legal environment E of T such that $T \cap E$ is failure-free, and hence T is a combinational autofailure. But then, since T is canonicalized by assumption, $F = \mathcal{T}^{(I \cup O)}$. Since $S \cap F = \emptyset$, it must also be the case that $S = \emptyset$. But by assumption, $S \neq \emptyset$. Therefore there must exist appropriate $C \subseteq \overline{F} = P'$.

Finally, the proof that $P' = \overline{F}$ and $F' = \overline{P}$ are input-downward-closed (in (O, I)) follows from the fact that F and P are output-upwards-closed (in (I, O)) :

Let $y, y' \in \mathcal{T}^O$ and $x \in \mathcal{T}^I$ such that $(y \cup x) \in \overline{F}$ and $y' \leq y$. We must prove that $(y' \cup x) \in \overline{F}$. By assumption, T is canonicalized. Therefore F is output-upwards-closed. Hence, because $(y \cup x) \notin F$, it must be the case that $(y' \cup x) \notin F$ as well. But then $(y' \cup x) \in \overline{F}$. The same argument may be repeated with P in place of F.

This concludes our proof that $T' = mir(T)$ is a combinational relation structure.

We proceed to prove that $T' = mir(T)$ is canonicalized:

- F' and P' are output-upwards-closed:

  $F' = \overline{P}$ is output-upwards-closed in (O, I) because P is input-downward-closed in (I, O), and $P' = \overline{F}$ is output-upwards-closed in (O, I) because F is input-downward-closed in (I, O) :

  Let $x, x' \in \mathcal{T}^I$ and $y \in \mathcal{T}^O$ such that $(y \cup x) \in \overline{P}$ and $x \leq x'$. We must prove that $(y \cup x') \in \overline{P}$. By assumption, P is input-downward-closed. Therefore, because $(y \cup x) \notin P$, it must be the case that $(y \cup x') \notin P$ as well. But then $(y \cup x') \in \overline{P} = F'$. The same argument may be repeated with F in place of P. ∎

- $S' \cap F' = \emptyset$:

  $S' = S$ and $F' = \overline{P} = \overline{(S \cup F)} \subseteq \overline{S} = \overline{S'}$. Thus $w \in S' \Longrightarrow w \notin F'$ and $w \in F' \Longrightarrow w \notin S'$. ∎

- If T' is a combinational autofailure then $F' = \mathcal{T}^{(I \cup O)}$:

  T is a legal environment of T'. Thus if T' is a combinational autofailure, then $T \cap T'$ is not failure-free.

But the F-set of $(T \cap T')$ is $(F' \cap P) \cup (P' \cap F) = (\overline{P} \cap P) \cup (\overline{F} \cap F) = \emptyset$. Therefore it must be the case that T' is not a combinational autofailure. Therefore, the condition holds vacuously.

QED Lemma 4.17

Lemma 4.18 *If T is a canonicalized combinational relation structure with nonempty S set, then*

$$T \cap mir(T) \text{ is failure free}$$

Proof: Let $T = (I, O, S, F)$ be a canonicalized combinational relation structure. By Lemma 4.17, if $S \neq \emptyset$, then $mir(T)$ is a combinational relation structure. Hence we can compose T and $mir(T)$, and the F-set of their composition is $((P \cap \overline{P}) \cup (F \cap \overline{F}))$, which is empty by definition of set complement. Hence $T \cap mir(T)$ is failure free. ∎

In order to define the maximal environment with which a combinational relation structure T may be safely composed, we preprocess T so as to enable application of the mirror operator, and then take its mirror. Our preprocessing consists of *canonicalizing* T as described in the previous section. We arbitrarily pick one (unambiguous) version of this process, which (by Theorem 4.15) is guaranteed to result in a combinational relation structure conformance equivalent to the original: by Theorem 4.15, $canon(T) = fe(afm(OUC(T)))$ is a combinational relation structure conformance equivalent to the original T :

$$canon((I, O, S, F)) = \begin{cases} (I, O, \emptyset, T^{(I \cup O)}) & \text{if } T \text{ is a combinational autofailure} \\ (I, O, OUC_{I,O}(S) - OUC_{I,O}(F), OUC_{I,O}(F)) & \text{otherwise} \end{cases}$$

Note that by Lemma 4.16, the *canon* operator is idempotent.

If T is a combinational relation structure, then the maximal safe environment of T is

$$T^{MaxEnv} = mir(canon(T))$$

The following theorems assure us that this formal definition does indeed yield the $\sqsubseteq$-maximal environment of T (of those whose S and F sets are disjoint) with which the original combinational relation structure T can be composed failure-free. This justifies calling T^{MaxEnv} the maximal safe environment of T. We also show that the *MaxEnv* operator provides a semi-decision procedure for conformance.

Lemma 4.19 *Let $T = (I, O, S, F)$ be a combinational relation structure such that the S-set of $canon(T)$ is non-empty. Then T^{MaxEnv} is a canonicalized combinational relation structure.*

Proof: By Theorem 4.15, $canon(T)$ is a canonicalized combinational relation structure. By Lemma 4.17, therefore, $mir(canon(T)) = T^{MaxEnv}$ is a canonicalized combinational relation structure as long as the S-set of $canon(T)$ is nonempty. ∎

Note that T^{MaxEnv} need not be a combinational relation structure. This occurs precisely when $canon(T)$ has an empty S-set: in that case, $mir(canon(T))$ has an empty P-set and hence does not obey the receptiveness constraint on P. This is because the S-set of $canon(T)$ is empty if and only if the F and P-sets of $canon(T)$ are equal: by definition of the canonicalization process and by Theorem 4.15, this in turn occurs exactly when T is a combinational autofailure. Hence a combinational relation structure T whose maximal safe environment is not a combinational relation structure is a combinational autofailure. Similarly, every combinational autofailure has a maximal safe environment that is not a combinational relation structure. This point will be important for understanding the interaction between the notions of maximal safe environment and conformance.

Lemma 4.20 *If T is a combinational relation structure, then T is a combinational autofailure if and only if T^{MaxEnv} is not a combinational relation structure.*

We extend the domain of the relation $\sqsubseteq$ to structures of the form $(I, O, \emptyset, \emptyset)$. These structures have the *form* of combinational relation structures, but they are not combinational relation structures because their P-set is empty (and hence not receptive). We maintain the requirement that only structures of the same (I, O)-type are comparable using this preorder. Clearly, for all combinational relation structures $T = (I, O, S, F)$,

$$T \not\sqsubseteq (I, O, \emptyset, \emptyset) \text{ and } (I, O, \emptyset, \emptyset) \sqsubseteq T \text{ and } (I, O, \emptyset, \emptyset) \sqsubseteq (I, O, \emptyset, \emptyset)$$

This extended definition is used by the following lemma, which holds for combinational relation structures T for which T^{MaxEnv} is not a combinational relation structure as well as for those such that T^{MaxEnv} is a combinational relation structure.

Lemma 4.21 *Let $T = (I, O, S, F)$ and $E = (O, I, S_E, F_E)$ be combinational relation structures. Then*

$$(T \cap E) \textit{ is failure-free if and only if } E \sqsubseteq T^{MaxEnv}$$

Proof:

As discussed above, and formalized in Lemma 4.20, T^{MaxEnv} is not a combinational relation structure if and only if the F and P-sets of $canon(T)$ are equal, which occurs if and only if T is a combinational autofailure. By definition of autofailures, this occurs precisely when there exists no legal environment E of T such that $T \cap E$ is failure-free.

If we extend the relation $\sqsubseteq$ to such structures $(I, O, \emptyset, \emptyset)$, as described above, we derive that if T is a combinational autofailure then there exists no combinational relation structure E such that $E \sqsubseteq T^{MaxEnv}$ (because there exists no non-empty P_E such that $P_E \subseteq \emptyset$). Hence if T^{MaxEnv} is not a combinational relation structure, then for *all* environments E of T, $(T \cap E)$ is not failure free *and* $E \not\sqsubseteq T^{MaxEnv}$. Hence in this case, $T \cap E$ is failure-free if and only if $E \sqsubseteq T^{MaxEnv}$.

If T^{MaxEnv} is a combinational relation structure then by Theorems 4.1 and 4.15,

$$(T \cap E) \text{ is failure free if and only if } (canon(T) \cap E) \text{ is failure free}$$

Let $T_0 = (I, O, F_0, P_0) = canon(T)$. Then

$$\begin{aligned}(T \cap E) \text{ is failure free} &\iff (T_0 \cap E) \text{ is failure free}\\ &\iff (P_0 \cap F_E) = \emptyset \wedge (F_0 \cap P_E) = \emptyset\\ &\iff F_E \subseteq \overline{P_0} \wedge P_E \subseteq \overline{F_0}\\ &\iff E \sqsubseteq mir(T_0) = T^{MaxEnv}\end{aligned}$$

∎

Therefore T^{MaxEnv} is the $\sqsubseteq$-maximal safe environment of T having disjoint S and F sets. We proceed to use this fact to derive a semi-decision procedure for conformance.

Lemma 4.22 *Let T be a combinational relation structure that is not a combinational autofailure. Then*

$$(T^{MaxEnv})^{MaxEnv} = canon(T)$$

Proof: Let $T = (I, O, F, P)$ be a combinational relation structure that is not a combinational autofailure. By Theorem 4.15, $canon(T)$ is a canonicalized combinational relation structure. Because T is not a combinational autofailure, the S-set of $canon(T)$ is non-empty. Thus (by Lemma 4.17) $mir(canon(T)) = T^{MaxEnv}$ is also a canonicalized combinational relation structure.

Hence in this case

$$\begin{aligned}(T^{MaxEnv})^{MaxEnv} &= mir(canon(T^{MaxEnv}))\\ &= mir(T^{MaxEnv}) && \text{by Lemma 4.16}\\ &= mir(mir(canon(T)))\\ &= canon(T)\end{aligned}$$

∎

Theorem 4.23 *Let $T = (I, O, F, P)$ and $T' = (I, O, F', P')$ be combinational relation structures such that the S-set of $canon(T')$ is nonempty. Then*

$$T \cap (T')^{MaxEnv} \text{ is failure-free} \Longrightarrow T \preceq T'$$

Proof: Let $T = (I, O, F, P)$ and $T' = (I, O, F', P')$ be combinational relation structures such that the the S-set of $canon(T')$ is non-empty. By Lemma 4.21, $T \cap (T')^{MaxEnv}$ is failure free if and only if $T \sqsubseteq ((T')^{MaxEnv})^{MaxEnv}$. By Lemma 4.22, $((T')^{MaxEnv})^{MaxEnv} = canon(T')$.

Let $T \cap (T')^{MaxEnv}$ be failure free. Then by Lemmas 4.21 and 4.22, $T \sqsubseteq canon(T')$. Thus by Lemma 4.7, $T \preceq canon(T')$. By Theorem 4.15, $T' \sim canon(T')$. Therefore $T \preceq T'$. ∎

As discussed previously, although T^{MaxEnv} is the $\sqsubseteq$-maximal safe canonicalized environment of T *if* it is a combinational relation structure, it need not in fact be a combinational relation structure. This occurs precisely when T is a combinational autofailure (Lemma 4.20).

By a vacuous application of the definition of conformance, every combinational relation structure of the appropriate (I, O)-type conforms to a combinational autofailure. Hence a combinational relation structure T whose maximal safe environment is not a combinational relation structure is a valid specification for *all* combinational relation structures that have the same I and O sets as T. The proof of these facts follows.

Theorem 4.24 *Let $T = (I, O, S, F)$ and $T' = (I, O, S', F')$ be combinational relation structures. If $(T')^{MaxEnv}$ is not a combinational relation structure, then $T \preceq T'$.*

Proof: Let $T' = (I, O, S', F')$ be a combinational relation structure such that $(T')^{MaxEnv}$ is not a combinational relation structure. By Lemma 4.20, this is equivalent to the statement that T' is a combinational autofailure.

By definition of a combinational autofailure, for all legal environments $E = (O, I, S_E, F_E)$ of T', $T' \cap E$ is not failure-free.

But then for all combinational relation structures $T = (I, O, S, F)$ it is trivially the case that for every legal environment $E = (O, I, S_E, F_E)$ of T and T', $T' \cap E$ is failure-free implies that $T \cap E$ is failure-free.

Hence $T \preceq T'$. ∎

Thus all combinational relation structures of the same (I, O)-type whose maximal safe environments are not combinational relation structures are conformance equivalent.

Lemma 4.25 *Let $T = (I, O, F, P)$ and $T' = (I, O, F', P')$ be combinational relation structures such that T^{MaxEnv} and $(T')^{MaxEnv}$ are not combinational relation structures. Then $T \sim T'$.*

Proof: Two applications of Theorem 4.24.

The preceding lemmas and theorems provide an effective semi-decision procedure for conformance, assuming one has an algorithm for determining whether or not a combinational relation structure is a combinational autofailure. As stated earlier, we have developed such an algorithm and it will be presented in Chapter 6.

If T' is a combinational autofailure, then we know that $(T')^{MaxEnv}$ is not a combinational relation structure, and so by Theorem 4.24, $T \preceq T'$. If T' is not a combinational autofailure, we may compute $(T')^{MaxEnv}$, and we know it will be a combinational relation structure. Then we may check whether or not $T \cap (T')^{MaxEnv}$ is failure-free. If it is failure-free, then by Theorem 4.23 it must be the case that $T \preceq T'$. However, if $(T')^{MaxEnv}$ is a combinational relation structure and $T \cap (T')^{MaxEnv}$ is not failure-free, we do not (yet) know whether or not $T \preceq T'$.

In the following section, we present the final theorems that describe a full decision procedure for conformance, by filling in this final case.

4.2.5 A decision procedure for conformance

4.2.5.1 Introduction

In this section, we prove that there is a unique canonicalized combinational relation structure in each conformance equivalence class. This defines a canonical element of each such class, and allows the possibility of working with a reduced circuit algebra consisting only of canonicalized combinational relation structures.

Furthermore, we prove that for non-autofailure T', $T \preceq T' \iff (T \cap (T')^{MaxEnv})$ is failure-free (Theorem 4.31, below) and $T \preceq T' \iff T \sqsubseteq canon(T')$ (Theorem 4.32,below). These results provide a decision procedure for conformance. In addition, the latter result clarifies how conformance handles a $\perp$ value on an output wire of a specification.

If we work only with canonical relation structures as circuit models and requirements specifications, these results reduce to the following simple statements:

$$T \preceq T' \iff (T \cap mir(T')) \text{ is failure-free } \iff T \sqsubseteq T'$$

While maintaining only canonical representations complicates the implementation of the algebraic operators, the above result leads to a *conceptually* simpler decision procedure for conformance.

4.2.5.2 Canonical elements

In this section, we prove that there is a unique canonicalized combinational relation structure in each conformance equivalence class. This defines a canonical element of each such class, and allows the possibility of working with a reduced circuit algebra consisting only of canonicalized combinational relation structures.

Lemma 4.26 *Every conformance equivalence class contains a canonicalized combinational relation structure.*

Proof:

By Theorem 4.15, for all combinational relation structures T it is the case that $canon(T)$ is canonicalized and is in the same conformance equivalence class as T. Therefore every conformance equivalence class contains at least one canonicalized combinational relation structure. ∎

Theorem 4.27 *Every conformance equivalence class contains a unique canonicalized combinational relation structure.*

Proof:

By Lemma 4.26, every conformance equivalence class contains at least one canonicalized combinational relation structure.

Let $T = (I, O, F, P)$ and $T' = (I, O, F', P')$ be distinct canonicalized combinational relation structures. Say without loss of generality that P' contains at least one input-output combination that is not contained in P (formally, $(P' - P) \neq \emptyset$), or F' contains at least one input-output combination that is not contained in F (formally, $(F' - F) \neq \emptyset$). Then $S \neq \emptyset$, as $F = P = \mathcal{T}^{(I \cup O)}$ would not allow $P \subset P'$ or $F \subset F'$. We must prove that $T \not\sim T'$.

We will prove that $T' \not\preceq T$, by exhibiting a legal environment E of T and T' such that $E \cap T$ is failure-free but $E \cap T'$ is not failure-free.

Let $E = mir(T)$. By Lemma 4.17 and because $S \neq \emptyset$, E is a combinational relation structure. By Lemma 4.18, $T \cap E$ is failure-free.

But because $(P' - P) \neq \emptyset$ or $(F' - F) \neq \emptyset$, we know that $\overline{P} \cap P' \neq \emptyset$ or $\overline{F} \cap F' \neq \emptyset$. Hence $T' \cap E$ is not failure-free.

Thus there exists at least one legal environment of T and T' violating the definition of T' conforming to T. Therefore $T' \not\preceq T$ and so $T' \not\sim T$. ∎

We call this unique canonicalized element in a conformance equivalence class its *canonical* element. The following fact clarifies the relation between the $\sqsubseteq$ ordering and these canonical elements.

Lemma 4.28 *Every conformance equivalence class contains a unique $\sqsubseteq$-maximal element having disjoint S and F sets, and that element is its canonical element.*

Proof:

By Theorem 4.27, every conformance equivalence class contains a unique canonicalized element. Therefore it suffices to prove that every $\sqsubseteq$-maximal element of every conformance equivalence class whose S and F sets are disjoint is canonicalized.

Let T be a combinational relation structure whose S and F-sets are disjoint. Assume that T is a maximal element in its conformance equivalence class. By construction, $T \sqsubseteq canon(T)$, and by Theorem 4.15, $T \sim canon(T)$. Therefore, $T = canon(T)$. ∎

Because each conformance equivalence class contains a unique canonical element, we hope to be able to work solely with these unique representatives. However, in order to do so we need to redefine our algebraic operators so that the class of canonical combinational relation structures becomes closed under their application. We do this by redefining the operators to first apply the operation as previously defined and then canonicalize the result:

- Renaming does not change: if the operand relation structure is canonicalized, the result is as well.
- Hiding preserves output-upward-closure, but it may convert a non-autofailure combinational relation structure into a combinational autofailure. Hence after application of the old deletion operator, we apply autofailure manifestation and failure exclusion to the result.
- Inverse deletion preserves the canonicality of its operand relation structure. Hence it need not be modified (see next item).

- Composition may destroy output-upward-closure; it may also create a combinational autofailure from non-autofailure component relation structures. Hence after application of the old composition operator, we apply the full canonicalization sequence to the result.

  Note that although composition is defined to apply both the inverse deletion and the intersection operators, we need recanonicalize only after application of intersection, the final step of the previously defined composition operator, as inverse deletion preserves canonicality.

We know that canonical relation structures are isomorphic to full conformance equivalence classes: each conformance equivalence class and its canonical element are in one-to-one correspondence. We prove that conformance equivalence is a congruence for circuit algebras. This allows us to define the quotient of the circuit algebra of combinational relation structures with respect to conformance equivalence, and tells us it is also a circuit algebra.

Theorem 4.29 *The class of canonical combinational relation structures, together with the new operator definitions, forms a circuit algebra.*

Proof:

We first prove that conformance equivalence is a congruence for circuit algebras. This proof is precisely the proof of Lemma 4.25 on p. 69 of [61]:

We must prove that if T_1, T_1', T_2 and T_2' are combinational relation structures such that $T_1 \sim T_1'$ and $T_2 \sim T_2'$, then $(T_1 \parallel T_2) \sim (T_1' \parallel T_2')$, $del(D)(T_1) \sim del(D)(T_1')$, and $ren(r)(T_1) \sim ren(r)(T_1')$. The first follows from four applications of Theorem 4.6, the second follows from two applications of Theorem 4.5, and the third follows from two applications of their obvious counterpart for the renaming operator. ∎

By results of algebra, the fact that $\sim$ is a congruence allows us to define the quotient of the circuit algebra of combinational relation structures, and guarantees that the result is also a circuit algebra. More concretely, this proves that we may work solely with canonical combinational relation structures, while still depending on the truth of the circuit algebra axioms.

4.2.5.3 The conformance check

We now derive two distinct presentations of a decision procedure for conformance. The former will prove to be most useful for deciding conformance relative to an environment (Section 4.3); the latter will clarify how conformance handles a $\perp$ value on an output wire of a specification.

Lemma 4.30 *Let T' be a canonical combinational relation structure. Then*

$$T \preceq T' \Longrightarrow T \sqsubseteq T'$$

Proof:

The lemma holds trivially if the S-set of T' is empty, because in that case T' is a combinational autofailure and so its F and P-sets are universal ($F' = P' = \mathcal{T}^{(I \cup O)}$).

We now consider the case in which the S-set of T' is nonempty:

Let $T = (I, O, F, P)$ and $T' = (I, O, F', P')$ be combinational relation structures such that $T' = canon(T')$ and $S' \neq \emptyset$. Let $T \preceq T'$. Then for all legal environments $E = (O, I, F_E, P_E)$ of T and T', $T' \cap E$ being failure free implies that $T \cap E$ is failure-free.

By Lemma 4.17, $mir(T')$ is both canonical and a combinational relation structure. Therefore it is a legal environment of T'.

By Lemma 4.18, $T' \cap mir(T')$ is failure-free. Therefore by Theorem 4.1, $T \cap mir(T')$ is failure-free as well. By Lemma 4.21, it must therefore be the case that $T \sqsubseteq (mir(T'))^{MaxEnv}$.

$$\begin{aligned}(mir(T'))^{MaxEnv} &= mir(canon(mir(T'))) \\ &= mir(mir(T')) && \text{by Lemmas 4.16 and 4.17} \\ &= T'\end{aligned}$$

Thus $T \sqsubseteq T'$. Therefore $T \preceq T' \Longrightarrow T \sqsubseteq T'$. ∎

Theorem 4.31 *If $T = (I, O, S, F)$ and $T' = (I, O, S', F')$ are both combinational relation structures, and T' is not a combinational autofailure, then*

$$T \preceq T' \Longleftrightarrow (T \cap (T')^{MaxEnv}) \textit{ is failure-free}$$

Proof: Let $T = (I, O, S, F)$ be a combinational relation structure. Let $T' = (I, O, S', F')$ be a non-autofailure combinational relation structure.

By Theorem 4.23, if $(T \cap (T')^{MaxEnv})$ is failure-free then $T \preceq T'$.

We must prove that if $T \preceq T'$, then $(T \cap (T')^{MaxEnv})$ is failure-free.

Let $T \preceq T'$. By Theorem 4.15, $T' \sim canon(T')$ and $canon(T')$ is indeed canonical.

Let $T'_0 = canon(T')$. By Lemma 4.30, $T \sqsubseteq T'_0$.

By Lemma 4.22, $T'_0 = ((T')^{MaxEnv})^{MaxEnv}$.

Therefore by Lemma 4.21, $T \sqsubseteq T'_0$ implies that $T \cap (T')^{MaxEnv}$ is failure-free.

Therefore $T \preceq T' \Longrightarrow T \cap (T')^{MaxEnv}$ is failure-free. ∎

The preceding theorem provides a decision procedure for checking conformance: it fills in the missing piece from the semi-decision procedure described at the end of Section 4.2.4. This decision procedure requires that we be able to effectively determine of a relation structure T' whether or not it is a combinational autofailure. We have not yet presented an effective algorithm for doing so; however, we have developed such an algorithm. It will be presented in Chapter 6.

The decision procedure for checking conformance that has been completed by the preceding theorem is the following. In order to check whether or not $T \preceq T'$, we first determine whether or not T' is a combinational autofailure. If T' is a combinational autofailure, then $T \preceq T'$ by Theorem 4.24.

Otherwise, we compute $(T')^{MaxEnv}$. We then check for emptiness of the F-set of $T \cap (T')^{MaxEnv}$: if it is empty, $T \preceq T'$, and otherwise not.

The following theorem provides an alternate presentation of the same decision procedure. It clarifies precisely how conformance handles a $\perp$ value on an output wire of a specification, as explained below.

Theorem 4.32 *Let $T = (I, O, P)$ and $T' = (I, O, P')$ be combinational relation structures. Then*

$$T \preceq T' \iff T \sqsubseteq canon(T')$$

Proof:

- $(\Longrightarrow)$: Let $T \preceq T'$. By Theorem 4.15, $T' \sim canon(T')$. Hence by Theorem 4.15 and Lemma 4.30, $T \sqsubseteq canon(T')$. ▮
- $(\Longleftarrow)$: Let $T \sqsubseteq canon(T')$. By Lemma 4.7, $T \preceq canon(T')$. But by Theorem 4.15, $canon(T') \sim T'$. Therefore $T \preceq T'$. ▮

The preceding theorem appears to provide an alternate decision procedure for checking conformance. However, the two procedures are effectively equivalent. Computing $(T')^{MaxEnv}$ requires us to compute $T'_0 = canon(T')$, and the method by which we check whether $T \sqsubseteq canon(T') = T'_0$ is to check whether $P \cap \overline{P'_0} = \emptyset$ and $F \cap \overline{F'_0} = \emptyset$, that is to say, whether $T \cap (T')^{MaxEnv}$ is failure-free. Hence the two theorems emphasize distinct views of the same conformance decision procedure.

The second theorem clarifies that the appearance of a $\perp$ value on an output wire in a specification indicates that any of the values $0, 1$ or $\perp$ may appear in its place in the allowed implementations. Because $canon(T')$ is output-upward-closed, a $\perp$-value on an output wire of T' cannot be distinguished from the availability of all three wire value options on that wire in T', when deciding whether or not $T \preceq T'$.

By Theorem 4.29, the class of canonical relation structures together with the adapted algebraic operators forms a circuit algebra. Thus we may choose to maintain all combinational relation structures in canonicalized form, in which case the results of Theorems 4.31 and 4.32 reduce to the following simple statements:

$$T \preceq T' \iff (T \cap mir(T')) \text{ is failure-free } \iff T \sqsubseteq T'$$

While maintaining only canonical representations complicates the implementation of the algebraic operators, the above result leads to a *conceptually* simpler decision procedure for conformance.

4.2.6 Examples

In this section we present some examples which illustrate the conformance relation and its use in hierarchical verification.

The first example (Example 4.1) continues our development of the gated ring oscillator example, introduced in Section 3.2.2. It substantiates the claim made there: combinational relation structures can represent the gated ring oscillator of Figure 3.1 sufficiently accurately to avoid the false positive verification result we derived using the Boolean relation representation.

In Example 4.2 we illustrate how a nondeterministic specification allows multiple correct implementations. We provide a nondeterministic specification and enumerate its correct implementations. In Example 4.4, we sketch a small example of the use of *hierarchical* verification.

In Example 4.3 we discuss how to most accurately represent a selector that is implemented by pass transistors. Combinational relation structures are not intended to handle such switch-level concerns as the bidirectional electron flow which may occur as two nodes stabilize to a newly-unified unipotential region. However, that concern arises in considering this circuit. This example graphically illustrates the fundamental need for F-sets in our models.

Example 4.1 Gated ring oscillator:

Consider again the circuit depicted in Figure 3.1 on page 54. This circuit is a gated ring oscillator that consists of a nand-gate and a non-inverting buffer composed together into a loop. In Example 3.6 on page 72, we provided a combinational relation structure representation of this circuit:

$$T_{gro} = (I = \{a\}, O = \{b, c\}, S = \{\overline{a}bc, \bot_a bc, a\bot_b\bot_c, \bot_a\bot_b\bot_c, \bot_a\bot_b c\}, F = \emptyset)$$

In our discussion of the Boolean relation representation of this circuit, we claimed that the combinational relation structure representation can be used to correctly show that this circuit, packaged so that the wire labeled c is an internal node rather than a primary output, is not a correct implementation of an inverter specification. However, only now do we have the tools to show that this is the case.

We retrieve from Example 3.6 the combinational relation structure representation of the newly packaged circuit that we previously constructed:

$$T_{Circuit} = del(\{c\})(T_{gro}) = (\{a\}, \{b\}, \{\overline{a}b, \bot_a b, a\bot_b, \bot_a\bot_b\}, \emptyset)$$

A combinational relation structure representation of the inverter specification appears in Example 3.1:

$$T_{Spec} = T_{inv-ab} = (\{a\}, \{b\}, \{\overline{a}b, a\overline{b}, \bot_a\bot_b, \bot_a b, \bot_a\overline{b}\}, \emptyset)$$

In order to check whether or not $T_{Circuit} \preceq T_{Spec}$, we canonicalize T_{Spec} and then determine whether $T_{Circuit} \sqsubseteq canon(T_{Spec})$. In this particular case, both the specification and its potential *implementation have empty F-sets. Therefore they cannot be combinational autofailures, and so $(T_{Spec})^{MaxEnv}$ is a combinational relation structure. In addition, $T' = canon(T_{Spec}) = T_{Spec}$, as T_{Spec} is already output-upward-closed.*

Because the behavior $\mathbf{a}\bot_b$ appears in $P_{Circuit}$ but not in P', the set of possible behaviors of T', we conclude that $T_{Circuit} \not\sqsubseteq T'$, and hence $T_{Circuit} \not\preceq T_{Spec}$. In other words, this circuit is not a correct implementation of the inverter specification. This result is accurate for precisely the reason implied by the combinational relation structure representations: in the actual gated loop circuit, on input value $\mathbf{a} = 1$ the node $\mathbf{b}$ oscillates rather than stabilizing to the value 0 as required by the inverter specification.

In the following examples we utilize a succinct notation for relations between wire values that is less unwieldy than the explicit lists of extended monomials used in previous examples. In this notation, we present the relation as an equation or set of equations between wire values. The value on node e is represented by v_e. In these equations, conjunction is implicit, and binds tighter than disjunction, which is represented by the symbol $+$. Other standard Boolean logic symbols may also appear in these equations: $\oplus$ for exclusive-or, $\overline{\oplus}$ for equivalence (NXOR), etc.

The relation refers only to Boolean values; the intent is to extrapolate to our ternary domain of wire values by expanding the indicated Boolean relation to its standard ternary extension, and then expanding the relation further by adding those behaviors necessary to make it input-downward-closed. By analogy to the $OUC_{I,O}$ operator, we define an $IDC_{I,O}$ operator that adds to its operand set precisely those elements necessary to make the set input-downward-closed. In the case of a combinational relation structure with a non-empty F-set, we independently apply the $IDC_{I,O}$ operator to the standard ternary extension of the relations given for the F-set and the P-set. Then we assign to the new S-set all those elements of the newly expanded P-set which do not appear in the newly expanded F-set. Because subsequent canonicalization would delete the other new elements from S (via failure exclusion), there is no loss of generality in this approach. Of course, not all combinational relation structures can be described in this way, as this is not the only option available in creating combinational relation structure representations of specifications and circuits. In Examples 3.4 and 3.5, for example, F-sets are larger than indicated by the above mapping process. However, we have just outlined one possible systematic way to create combinational relation structure models from Boolean descriptions.

We may mix and match the new and old notations, representing some portion of a ternary-domain relation as an explicit set of extended monomials and another part as an equation. Note that where we mix the equation notation and the monomial notation in a behavioral description, we intend to denote the input-downward-closure of the standard ternary extension of the relation described by their union.

In addition, we may sometimes employ a shorthand in presentation of the relations between wire values, in which wire *value* indicators are replaced by the relevant wire *names.* For example, if v_e

denotes the value on the node labeled e, the equation $e = ab + cd$ is shorthand for the relation $v_e = v_a v_b + v_c v_d$.

Example 4.2 *The specification*

$$T_{Spec} = (I = \{a, b, c, d\}, O = \{e\}, S = ((e = ab + cd) \cup \{\overline{a}\overline{b}\overline{c}\overline{d}e\}), F = \emptyset)$$

allows the output wire e to stabilize to either 0 or 1 if the input wires all have value 0. However, an implementation satisfying this specification may be deterministic, and settle to a specific predetermined value in this case. A correct implementation $T_{Impl} = (I, O, S_{Impl}, F_{Impl}) \preceq T_{Spec}$ must have $F_{Impl} = \emptyset$, but may take any of the three forms

- $S_{Impl} = S_{Spec}$
- $S_{Impl} = (e = ab + cd)$, *or*
- $S_{Impl} = (e = ab + cd + \overline{a}\overline{b}\overline{c}\overline{d})$.

For example, $T \preceq T_{Spec}$ for the combinational relation structure

$$T = del(\{f, g\})(T_{nand-abf} \parallel T_{nand-cdg} \parallel T_{nand-fge})$$

[which represents the circuit] illustrated in Figure 4.1.

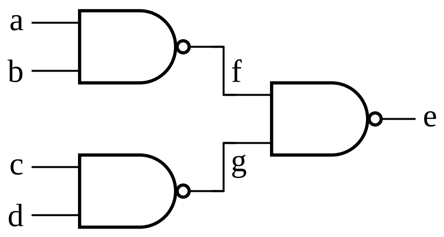

Figure 4.1: One possible implementation of an example nondeterministic specification

In our next example we consider the case of a selector with two select lines and two input data lines. Under accepted normal operating conditions, such a circuit should never receive the value 1 on both its select lines simultaneously. If the circuit employs pass transistors, the effect of holding both select lines high may be to unify the `data_in` lines and the `data_out` line into a single node (so that their values stabilize as a unipotential region). This can have undesirable effects on the logic ostensibly driving the `data_in` lines. Such a situation may cause actual circuit behavior that is inconsistent with our model, so it is necessary to disallow it in order to preserve the integrity of the model.

Stating the allowed environments of a circuit can be a critical part of its modeling. At the same time we realize that a circuit model does not have the authority to control its environment – only to state explicitly the class of environments over which an acceptable implementation is guaranteed to behave according to the specification. The F-set expresses this restriction without implying that a circuit may control its own environment.

Use of the F-set to express the situation described above is akin to its use in the asynchronous trace theory to express hazards [61]. In both cases, the F-set makes explicit the limits of the level of abstraction employed by the model. Its use indicates where the model breaks down because of an inability to describe certain low-level electrical phenomena.

Note also that this use of the F-set clarifies why we have defined composition to maximize the F-set of a composite relation structure. When we compose a relation structure representation of the selector together with a relation structure representation of any logic driving the select lines, the F-set of the composite relation structure should and does contain all value combinations in which both select lines are held high. Thus failure marking is propagated out to the periphery of the composite circuit.

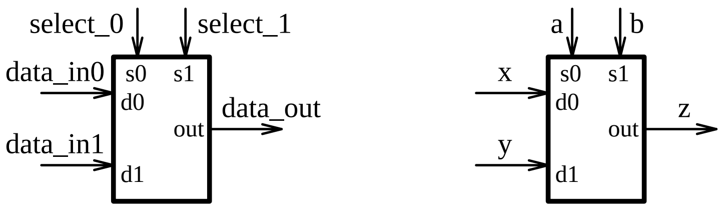

Figure 4.2: Selector circuit and relabelled version

Example 4.3 *The following combinational relation structure is a requirements specification for a selector with two select lines and two input-data lines, as illustrated in Figure 4.2. In order to slightly compress its presentation, we have labeled the* `select_0` *line as the shorter a,* `select_1` *as b,* `data_in0` *as x,* `data_in1` *as y, and* `data_out` *as z.*

$$T_{sel-spec} = (\{a, b, x, y\}, \{z\}, S_{sel-spec}, F_{sel-spec})$$

where

$$S_{sel-spec} = (z = a\overline{b}x + \overline{a}by) \cup \{\overline{a}\overline{b}X_xX_yX_z\}$$

and

$$F_{sel-spec} = \{abX_xX_yX_z\}$$

In reading this description, recall that where we mix the equation notation and the extended monomial notation in a behavioral description, we intend to denote the input-downward-closure of the standard ternary extension of the relation described by their union. In this case, for example,

$$\{\overline{a}\bot_bX_xX_yX_z, \bot_a\overline{b}X_xX_yX_z\} \subseteq S_{sel-spec}$$

because these terms are in the input downward closure of $\{\overline{a}\overline{b}X_xX_yX_z\} \subseteq S_{sel-spec}$*. Later we will see examples in which the standard ternary extension of the parts of* S *denoted via distinct kinds of notation is a proper superset of the union of the two parts' standard ternary extensions. As explained earlier, in this case we intend our mixed notation to denote the larger set.*

An implementation may make an arbitrary choice about what value to output on the wire z *when both select lines are low. This is because any environment in which such an implementation is placed should not sample the data output by the selector under those circumstances. This is expressed by the inclusion of* $\overline{a}\overline{b}X_xX_yX_z$ *in* $S_{sel-spec}$*.*

A selector should never be asked to handle a situation in which both select lines are held high. Fundamentally, the specification of a selector function disallows this situation, even though technically a circuit cannot control its environment. As discussed above, actual selector circuits may not be physically equipped to handle the consequences of both select lines being held high; despite this they are considered correct implementations of this specification. Thus all behaviors in which both select lines are high may be failure behaviors, and so $F_{sel-spec} = \{abX_xX_yX_z\}$*.*

The selector specification should admit implementations that make arbitrary choices about what to do when both the select lines are held low. In addition, an implementation may in fact be able to handle both select lines being high, if the values on the two input data lines are the same. For example, when both select lines are held low, an implementation may set $z = x$ *or* $z = y$ *or* $z = 0$ *or* $z = 1$*. When both select lines are held high and* $x = y$*, an implementation may set* $z = x$*, and contain* $abxyz$ *and* $ab\overline{x}\overline{y}\overline{z}$ *in its* S*-set (or it may contain them in its* F*-set). All eight of these possible implementations are allowed by the specification as given.*

Finally, we present a small example of hierarchical verification.

Example 4.4 *We would like to develop a new kind of selector, one that sets its* `data_out` *value to the value on* `data_in0` *if the input wires* c *and* d *have the same value, and sets its* `data_out` *value to the value on* `data_in1` *if the input wires* c *and* d *have distinct values. We can easily construct such a circuit by hooking up some simple combinational logic to any correct implementation of the selector specification described in the previous example. Thus the following combinational relation structure expresses our new specification:*

$$T_{spec} = del\{a, b\}(T_{xor-cdb} \parallel T_{inv-ba} \parallel T_{sel-spec})$$

where $T_{xor-cdb}$ *is a combinational relation structure representing an exclusive-or gate with inputs* c *and* d *and output wire* b*, and* T_{inv-ba} *is a combinational relation structure representing an inverter with input wire labeled* b *and output wire labeled* a*. Recall that*

$$T_{sel-spec} = (\{a, b, x, y\}, \{z\}, S_{sel-spec}, F_{sel-spec})$$

is the selector specification of Example 4.3, in which the wire names a and b are shorthand for the pins `select_0` *and* `select_1`*, respectively, the wire names x and y are shorthand for the pins* `data_in0` *and* `data_in1`*, respectively, and the wire name z is shorthand for the* `data_out` *pin.*

In hierarchical verification, we verify that our implementations of T_{inv-ba}, $T_{xor-cdb}$, and the selector all conform to their respective specifications. We may then conclude that their composition, with the wires labeled a and b packaged as internal nodes rather than primary outputs, meets the specification given above. If we later choose to exchange the selector circuit we chose for another, $T_{sel-impl-2}$, it suffices to verify that $T_{sel-impl-2} \preceq T_{sel-spec}$ in order to conclude that the full new circuit is a correct implementation of the full specification.

4.3 Substitution for combinational circuit models

4.3.1 Introduction

Many hardware design problems require the designer to determine the advisability of replacing one subcircuit by another. Problems such as optimization of a multi-level logic network during logic synthesis, or rectification of an already-existing circuit to meet an altered specification, fall into this class.

In this section we address the problem of determining the *correctness* of replacing one subcircuit by another (or of designing a particular subcircuit for which there is no predecessor), relative to a given specification for the behavior of the full circuit. We utilize our previous results to derive a full general characterization of *all* allowed substitutions for a subcircuit or subcircuit specification.

The notion of the maximal safe environment of a combinational relation structure can be used to determine those relation structures that can be safely substituted for a component of a predetermined circuit. We seek to determine the correctness of a circuit with respect to, or relative to, a given predefined environment.

We define correctness with respect to an environment using expression contexts as defined early in Section 4.2.2. An expression context is a circuit algebra expression with a single free variable α [61]. The (I, O)-type of α must be known, and is referred to as (I_α, O_α). If we replace α in this expression by a combinational relation structure $(I_\alpha, O_\alpha, S, F)$, the result denotes a combinational relation structure.

Formally, we define correctness with respect to an environment as follows. For combinational relation structures $T' = (I, O, S', F')$ and $T'' = (I, O, S'', F'')$, we say that T' is a correct substitution for T'' with respect to an environment E, written $T' \preceq_E T''$, if and only if $E[\alpha]$ is a legitimate expression context such that $I_\alpha = I$ and $O_\alpha = O$, and $E[T'] \preceq E[T'']$.

If $T = E[T'']$ is the original circuit, and T'' the part thereof for which we wish to substitute something new, we seek a decision procedure that will tell us whether T' is a correct substitute for T'' with respect to E. The following subsection provides such a decision procedure.

4.3.2 The closed-form solution

In this section we present a closed form solution to a slightly more general problem than that described above. We require only that the specification have the same (I, O)-type as the surrounding context of the subcircuit to be replaced. This more general problem definition lends itself to a recursive solution, of which our original problem's solution is a special case.

In order to define this closed-form solution we utilize the following recursive function f. This function effectively peels away the surrounding environment specification from around a subcircuit, the allowed replacements for which we seek to characterize. It recursively derives a most general characterization of the allowed substitutions into the designated "location" in the environment that is its first argument, such that the substitution meets the specification which is its second argument. Theorem 4.33 below states that the function defined below does indeed correctly derive the result just described.

The auxiliary function f is inductively defined as follows.

- $f(\alpha, T) = T$
- $f(del(D)(T_2[\alpha]), T) = f(T_2[\alpha], del_O^{-1}(D)(T))$
- $$f(T_3 \parallel T_2[\alpha], T) = \begin{cases} (I_\alpha, O_\alpha, \mathcal{T}^{A_\alpha}, \mathcal{T}^{A_\alpha}) & \text{if } T \text{ is a combinational autofailure (CAF)} \\ (I_\alpha, O_\alpha, \emptyset, \emptyset) & \text{if } del(A - A_2)(T^{MaxEnv} \parallel T_3) \text{ is a CAF} \\ f(T_2[\alpha], (del(A - A_2)(T^{MaxEnv} \parallel T_3))^{MaxEnv}) & \text{otherwise} \end{cases}$$

where $A_x = I_x \cup O_x$. In the third case, the intent is that the earliest applicable option among those listed be taken.

The operation $del_O^{-1}(D)(T)$ adds the wires D to T as new *output* wires taking arbitrary values. In effect, it allows for uncontrolled output values. Note that although we did not previously give it a name, we have already used this construction: in the proof of Lemma 4.3. Clearly the result of applying this new operator to a combinational relation structure is also a combinational relation structure. In fact, it preserves canonicality.

In order to interpret the results of applying f, we extend the domain of the $\preceq$ relation, specifying that for all combinational relation structures $T = (I, O, S, F)$,

$$T \not\preceq (I, O, \emptyset, \emptyset)$$

Note that this is consistent with the extension we made to the domain of the $\sqsubseteq$ relation in Section 4.2.4.

This extension is necessary to handle the third induction case of the theorem below, in which it can happen that $(del(A - A_2)(T^{MaxEnv} \parallel T_3))^{MaxEnv}$ is not a combinational relation structure because $del(A - A_2)(T^{MaxEnv} \parallel T_3)$ is a combinational autofailure. In that case, the inductive definition effectively states that there exists no combinational relation structure T' meeting the criterion that $T_0[T'] \preceq T$ in the original statement of the theorem. The intuitive reason that such a case might arise is that the global specification T and that part of the full circuit whose form is already determined (T_3) may be mutually incompatible. Given the extension defined above, the induction result can state this cleanly: in such a case, the end result of applying the function f is the structure $(I, O, \emptyset, \emptyset)$, which indicates unambiguously that there is no possible T' that can be substituted as described.

Theorem 4.33 *If $T = (I, O, S, F)$ and $T' = (I_\alpha, O_\alpha, S', F')$ are combinational relation structures, and $T_0[\alpha]$ is an expression context of type (I, O), then*

$$T_0[T'] \preceq T \textit{ iff } T' \preceq f(T_0[\alpha], T)$$

Proof: The proof is by induction on the structure of T_0. Some of the lemmas on which it depends have not yet been presented; their statements and proofs appear in Section 4.3.3.

Base case: Let $T_0[\alpha] = \alpha$. Then the statement of the theorem reduces to

$$T' \preceq T \text{ iff } T' \preceq f(T_0[\alpha], T) = f(\alpha, T) = T$$

which is self-evident.

Induction cases:

Case: $T_0[\alpha] = del(A_1 - A_0)(E_1[\alpha])$

The induction hypothesis is that

$$\forall T'', T'. E_1[T'] \preceq T'' \text{ iff } T' \preceq f(E_1[\alpha], T'')$$

Now

$$\begin{array}{rcll} T_0[T'] \preceq T & \Longleftrightarrow & del(A_1 - A_0)(E_1[T']) \preceq T & \text{Note that } A_0 = A \\ & \Longleftrightarrow & E_1[T'] \preceq del_O^{-1}(A_1 - A)(T) & \text{by Lemma 4.40} \\ & \Longleftrightarrow & T' \preceq f(E_1[\alpha], del_O^{-1}(A_1 - A)(T)) & \text{by the induction hypothesis} \\ & & \quad = f(del(A_1 - A)(E_1[\alpha]), T) & \\ & & \quad = f(T_0[\alpha], T) & \end{array}$$

Case: $T_0[\alpha] = T_2 \parallel (E_1[\alpha])$

The induction hypothesis is that

$$\forall T'', T'. E_1[T'] \preceq T'' \text{ iff } T' \preceq f(E_1[\alpha], T'')$$

If T is a combinational autofailure, then by Lemma 4.20, T^{MaxEnv} is not a combinational relation structure. Therefore, by Theorem 4.24, $T_0[T'] \preceq T$. In that case

$$f(T_0[\alpha], T) = f(T_2 \parallel (E_1[\alpha]), T) = (I_\alpha, O_\alpha, \mathcal{T}^{A_\alpha}, \mathcal{T}^{A_\alpha})$$

and so (again by Theorem 4.24) $T' \preceq f(T_0[\alpha], T)$.

If T is not a combinational autofailure, and $(del(A-A_1)(T^{MaxEnv} \parallel T_2))^{MaxEnv}$ is a combinational relation structure, then

$$\begin{array}{rcll}
T_0[T'] \preceq T & \equiv & T_2 \parallel (E_1[T']) \preceq T & \\
& \Longleftrightarrow & T^{MaxEnv} \cap (T_2 \parallel (E_1[T'])) \text{ is failure-free (FF)} & \text{by Theorem 4.31} \\
& \Longleftrightarrow & T^{MaxEnv} \parallel (T_2 \parallel (E_1[T'])) \text{ is FF} & \text{Note: } A = A_2 \cup A_1 \\
& \Longleftrightarrow & (T^{MaxEnv} \parallel T_2) \parallel (E_1[T']) \text{ is FF} & \text{composition axioms} \\
& \Longleftrightarrow & (del(A-A_1)(T^{MaxEnv} \parallel T_2)) \cap (E_1[T']) \text{ is FF} & \text{by Lemma 4.2} \\
& \Longleftrightarrow & (canon(del(A-A_1)(T^{MaxEnv} \parallel T_2))) \cap (E_1[T']) \text{ is FF} & \text{by Lemma 4.41} \\
& \Longleftrightarrow & ((del(A-A_1)(T^{MaxEnv} \parallel T_2))^{MaxEnv})^{MaxEnv} \cap (E_1[T']) & \\
& & \text{is failure free} & \text{by Lemma 4.22} \\
& \Longleftrightarrow & E_1[T'] \preceq (del(A-A_1)(T^{MaxEnv} \parallel T_2))^{MaxEnv} & \text{by Theorem 4.31} \\
& \Longleftrightarrow & T' \preceq f(E_1[\alpha], (del(A-A_1)(T^{MaxEnv} \parallel T_2))^{MaxEnv}) & \text{induction hypothesis}
\end{array}$$

The use of hiding (del) in this case is legal, because $(A - A_1) \subseteq O_{(T^{MaxEnv} \parallel T_2)}$. This follows from the original statement that $T_2 \parallel (E_1[T']) \preceq T$, which implies that $A = (A_2 \cup A_1)$ and $I = ((I_1 \cap I_2) \cup (A_1 - A_2) \cup (A_2 - A_1))$. The result follows specifically because $O_{(T^{MaxEnv} \parallel T_2)} = O_2 \cup I$ and $A - A_1 = A_2 - A_1 = (O_2 - I_1) \cup (I_2 - A_1)$: because of these facts, we only need to prove that $I_2 - A_1 \subseteq O_2 \cup I$, which is clearly equivalent to $I_2 - A_1 \subseteq I$, which follows from the expansion of I, above.

It can happen that $(del(A-A_1)(T^{MaxEnv} \parallel T_2))^{MaxEnv}$ is not a combinational relation structure because $del(A - A_1)(T^{MaxEnv} \parallel T_2)$ is a combinational autofailure. The proof for that case follows.

If T is a combinational relation structure that is not a combinational autofailure, but

$$del(A - A_1)(T^{MaxEnv} \parallel T_2)$$

is a combinational autofailure, then

$$\begin{array}{rcll}
T_0[T'] \preceq T & \equiv & T_2 \parallel (E_1[T']) \preceq T & \\
& \Longleftrightarrow & T^{MaxEnv} \cap (T_2 \parallel (E_1[T'])) \text{ is failure-free (FF)} & \text{by Theorem 4.31} \\
& \Longleftrightarrow & T^{MaxEnv} \parallel (T_2 \parallel (E_1[T'])) \text{ is FF} & \text{Note that } A = A_2 \cup A_1 \\
& \Longleftrightarrow & (T^{MaxEnv} \parallel T_2) \parallel (E_1[T']) \text{ is FF} & \text{composition axioms} \\
& \Longleftrightarrow & (del(A-A_1)(T^{MaxEnv} \parallel T_2)) \cap (E_1[T']) \text{ is FF} & \text{by Lemma 4.2}
\end{array}$$

If $del(A - A_1)(T^{MaxEnv} \parallel T_2)$ is a combinational autofailure, then there exists no such $E_1[T']$, *i.e.*, no $E_1[T']$ such that $(del(A-A_1)(T^{MaxEnv} \parallel T_2)) \cap (E_1[T'])$ is failure-free. Therefore, $T_0[T'] \not\preceq T$.

In addition,

$$(del(A - A_1)(T^{MaxEnv} \parallel T_2)) \cap (E_1[T']) \text{ is failure-free} \Longleftrightarrow T' \preceq (I_\alpha, O_\alpha, \emptyset, \emptyset)$$

because both sides of this "if and only if" statement are false.

But of course in this case

$$f(T_2 \parallel E_1[\alpha], T) = (I_\alpha, O_\alpha, \emptyset, \emptyset)$$

Therefore

$$T_0[T'] \preceq T \Longleftrightarrow T' \preceq f(T_2 \parallel E_1[\alpha], T)$$

and so the theorem is proved for this final case as well.

Hence in all cases, $T_0[T'] \preceq T$ if and only if $T' \preceq f(T_0[\alpha], T)$. ∎

We may now use the function f to derive the solution to our original subcircuit substitution problem. The following theorem states that those T' such that $T' \preceq f(E[\alpha], E[T''])$ are precisely the acceptable substitutes for the subcircuit T''.

Theorem 4.34 *If $E[\alpha]$ is an expression context and $T'' = (I_\alpha, O_\alpha, S'', F'')$ is a combinational relation structure, then for all combinational relation structures $T' = (I_\alpha, O_\alpha, S', F')$,*

$$T' \preceq_E T'' \Longleftrightarrow T' \preceq f(E[\alpha], E[T''])$$

Proof: Let $T_0 = E$ and $T = E[T'']$. The result then follows from Theorem 4.33. ∎

Note that it follows from Theorem 4.33 that it cannot happen that $f(E[\alpha], E[T'']) = (I_\alpha, O_\alpha, \emptyset, \emptyset)$. This is because such a case arises only if some component of $E[\alpha]$ is incompatible with the global specification, which in this case is $E[T'']$. But if T'' is a combinational relation structure of the correct (I_α, O_α)-type, and $E[\alpha]$ really is a legitimate expression context, then $E[T'']$ must be compatible with the requirements imposed by its own components. Hence, if it does happen that $f(E[\alpha], E[T'']) = (I_\alpha, O_\alpha, \emptyset, \emptyset)$, then $E[T'']$ is not a combinational relation structure. In that case, $T' \not\preceq_E T''$ for *all* relation structures T'. In particular, in this case T'' may not be substituted for itself in E, because either E is not a legal expression context (and so $\preceq_E$ is not defined), or T'' is not a combinational relation structure.

In the remainder of this section, we present and prove the remaining supporting lemmas used in the proof of Theorem 4.33 (Section 4.3.3), and provide further discussion and examples of relevant applications of this theorem (Section 4.3.4).

4.3.3 Supporting lemmas

In this section, we present and prove Lemmas 4.40 and 4.41, the remaining supporting lemmas used in the proof of Theorem 4.33 in Section 4.3.2. Their own supporting lemmas precede them.

Lemma 4.35 *Let $T = (I, O, S, F)$ be a combinational relation structure. Let $(D \cap (I \cup O)) = \emptyset$. Then there exists a combinational relation structure $E = (O, I, S_E, F_E)$ such that $T \cap E$ is failure free if and only if there exists a combinational relation structure $E' = (O \cup D, I, S_{E'}, F_{E'})$ such that $del_O^{-1}(T) \cap E'$ is failure free.*

Proof: Let $T = (I, O, S, F)$ be a combinational relation structure and $(D \cap (I \cup O)) = \emptyset$.

- $(\Longrightarrow)$:

  Let $E = (O, I, S_E, F_E)$ be a legal environment of T such that $T \cap E$ is failure free.

  Let $E' = del^{-1}(D)(E)$. We proceed to prove that $del_O^{-1}(D)(T) \cap E'$ is failure free.

  The F-set of $del_O^{-1}(D)(T) \cap E'$ is

  $$[(del^{-1}(D)(F) \cap del^{-1}(D)(P_E)) \cup (del^{-1}(D)(P) \cap del^{-1}(D)(F_E))]$$

  Say $z \in (del^{-1}(D)(F) \cap del^{-1}(D)(P_E))$. Then $z = (w \cup q)$ for some $w \in \mathcal{T}^{(I \cup O)}$ and $q \in \mathcal{T}^D$. Thus $w \in (F \cap P_E)$. But by assumption, $T \cap E$ is failure-free. Therefore there exists no such w, and hence no such z. Because the identical argument works for $z' \in (del^{-1}(D)(P) \cap del^{-1}(D)(F_E))$, we conclude that

  $$[(del^{-1}(D)(F) \cap del^{-1}(D)(P_E)) \cup (del^{-1}(D)(P) \cap del^{-1}(D)(F_E))] = \emptyset$$

  Therefore $del_O^{-1}(D)(T) \cap E'$ is failure free.

  Thus we have proved that there exists a legal environment E' of $del_O^{-1}(D)(T)$ such that $del_O^{-1}(D)(T) \cap E'$ is failure free. ∎

- $(\Longleftarrow)$:

  Let $E' = (O \cup D, I, S_{E'}, F_{E'})$ be a combinational relation structure such that $del_O^{-1}(D)(T) \cap E'$ is failure free.

  Let $E = (O, I, del(D)(S_{E'}), del(D)(F_{E'}))$. Note that $E \neq del(D)(E')$. However, a simple calculation, based on the assumption that E' is a combinational relation structure, shows that E is a combinational relation structure: it is input-downward-closed and has a receptive P-set.

  We proceed to prove that $T \cap E$ is failure free.

  Say not. Then there must exist $z \in ((F \cap P_E) \cup (P \cap F_E))$. Say without loss of generality that $z \in (F \cap P_E)$. Because $z \in P_E$, there must exist $q \in \mathcal{T}^D$ such that $(z \cup q) \in P_{E'}$. Since for *any* $q_0 \in \mathcal{T}^D$ it is the case that $(z \cup q_0) \in del^{-1}(D)(F)$, it is true for $q_0 = q$ as well. Therefore $(z \cup q) \in (del^{-1}(D)(F) \cap P_{E'})$. But then $(z \cup q)$ is in the F-set of $del_O^{-1}(D)(T) \cap E'$, which by assumption is empty. Therefore there can be no such z, and so $(F \cap P_E) = \emptyset$. A symmetric argument shows that $(P \cap F_E) = \emptyset$ as well, and so $T \cap E$ is failure free.

Thus we have proved that there exists a legal environment E of T such that $T \cap E$ is failure free. ∎

QED Lemma 4.35

Lemma 4.36 *Let $T = (I, O, S, F)$ be a combinational relation structure. Let $(D \cap (I \cup O)) = \emptyset$. Then T is a combinational autofailure if and only if $del_O^{-1}(D)(T)$ is a combinational autofailure.*

Proof: This follows directly from Lemma 4.35.

Lemma 4.37 *Let $T = (I, O, S, F)$ be a combinational relation structure, and let $(D \cap (I \cup O)) = \emptyset$. Then*

$$del_O^{-1}(D)(OUC(T)) = OUC(del_O^{-1}(D)(T))$$

Proof:

Let $q \in del^{-1}(D)(OUC_{I,O}(F))$. Then $q = (x \cup y)$ for some $x \in \mathcal{T}^I$ and $y \in \mathcal{T}^{(O \cup D)}$. Let $z = del(D)(y)$ and $w = del(O)(y)$. Then $(x \cup z) \in OUC_{I,O}(F)$. Therefore there must exist $z' \leq z$ such that $(x \cup z') \in F$. But then $(x \cup (w \cup z')) \in del^{-1}(D)(F)$, and

$$q = (x \cup y) = (x \cup (w \cup z)) \in OUC_{I,(O \cup D)}(del^{-1}(D)(F))$$

Therefore $del^{-1}(D)(OUC_{I,O}(F)) \subseteq OUC_{I,(O \cup D)}(del^{-1}(D)(F))$.

Let $s \in OUC_{I,(O \cup D)}(del^{-1}(D)(F))$. Then $s = (x \cup y)$ for some $x \in \mathcal{T}^I$ and $y \in \mathcal{T}^{(O \cup D)}$. Furthermore, there exists $y' \leq y$ such that $(x \cup y') \in del^{-1}(D)(F)$.

Let $z' = del(D)(y')$. Then $(x \cup z') \in F$. Let $z = del(D)(y)$ and $w = del(O)(y)$. Because $y' \leq y$, it must be the case that $z' \leq z$. Therefore $(x \cup z) \in OUC_{I,O}(F)$. But then

$$s = (x \cup y) = (x \cup (z \cup w)) \in del^{-1}(D)(OUC_{I,O}(F))$$

Therefore $OUC_{I,(O \cup D)}(del^{-1}(D)(F)) \subseteq del^{-1}(D)(OUC_{I,O}(F))$.

Therefore $del^{-1}(D)(OUC_{I,O}(F)) = OUC_{I,(O \cup D)}(del^{-1}(D)(F))$. The same proof holds with F replaced by S or P.

Hence

$$\begin{aligned} OUC(del_O^{-1}(D)(T)) &= (I, O \cup D, OUC_{I,(O \cup D)}(del^{-1}(D)(S)), OUC_{I,(O \cup D)}(del^{-1}(D)(F))) \\ &= (I, O \cup D, del^{-1}(D)(OUC_{I,O}(S)), del^{-1}(D)(OUC_{I,O}(F))) \\ &= del_O^{-1}(D)(OUC(T)) \end{aligned}$$

QED Lemma 4.37

Lemma 4.38 *Let $T = (I, O, F, P)$ be a combinational relation structure, and let $(D \cap (I \cup O)) = \emptyset$. Then*

$$del_O^{-1}(D)(canon(T)) = canon(del_O^{-1}(D)(T))$$

Proof:

We consider two cases. In the first, both T and $del_O^{-1}(D)(T)$ are combinational autofailures. In this case,

$$\begin{aligned}
del_O^{-1}(D)(canon(T)) &= del_O^{-1}(D)((I, O, F = \mathcal{T}^A, P = \mathcal{T}^A)) \\
&= (I, O \cup D, F = \mathcal{T}^{(A \cup D)}, P = \mathcal{T}^{(A \cup D)}) \\
&= canon(del_O^{-1}(D)(T))
\end{aligned}$$

By Lemma 4.36, the sole remaining case is that in which neither T nor $del_O^{-1}(D)(T)$ is a combinational autofailure. In this case,

$$\begin{aligned}
del_O^{-1}(D)(canon(T)) &= (I, O \cup D, del^{-1}(D)(OUC_{I,O}(F)), del^{-1}(D)(OUC_{I,O}(P))) \\
&= del_O^{-1}(D)(OUC(T)) \\
&= OUC(del_O^{-1}(D)(T)) \qquad \text{by Lemma 4.37} \\
&= (I, O \cup D, OUC_{I,(O \cup D)}(del^{-1}(D)(F)), OUC_{I,(O \cup D)}(del^{-1}(D)(P))) \\
&= canon(del_O^{-1}(D)(T))
\end{aligned}$$

Note that the S sets of both $del_O^{-1}(D)(canon(T))$ and $canon(del_O^{-1}(D)(T))$ are equal to their common P set minus their common F set, and are therefore equal to each other. Thus there was no need to mention failure-exclusion explicitly in the equations above. ∎

Lemma 4.39 *Let $T = (I, O, F, P)$ be a canonical combinational relation structure, and let $(D \cap (I \cup O)) = \emptyset$. Then*

$$del^{-1}(D)(mir(T)) = mir(del_O^{-1}(D)(T)$$

Proof:

$$\begin{aligned}
del^{-1}(D)(mir(T)) &= del^{-1}(D)((O, I, \overline{P}, \overline{F})) \\
&= (O \cup D, I, \overline{del^{-1}(D)(\overline{P})}, \overline{del^{-1}(D)(\overline{F})}) \\
&= (O \cup D, I, \overline{del^{-1}(D)(P)}, \overline{del^{-1}(D)(F)}) \\
&= mir((I, O \cup D, del^{-1}(D)(F), del^{-1}(D)(P)) \\
&= mir(del_O^{-1}(D)(T))
\end{aligned}$$

∎

Lemma 4.40 *Let $T = (I, O, F, P)$ be a combinational relation structure. Let $D \subseteq O$. Let $T' = (I, (O - D), F', P')$ be a combinational relation structure. Then*

$$del(D)(T) \preceq T' \Longleftrightarrow T \preceq del_O^{-1}(D)(T')$$

Proof:

If $(T')^{MaxEnv}$ is not a combinational relation structure, then by Theorem 4.24, $del(D)(T) \preceq T'$. At the same time, T' must be a combinational autofailure. But then by Lemma 4.36, so is

$del_O^{-1}(D)(T')$. Therefore $(del_O^{-1}(D)(T'))^{MaxEnv}$ is also not a combinational relation structure, and so (again by Theorem 4.24), $T \preceq del_O^{-1}(D)(T')$. Therefore in this case $del(D)(T) \preceq T' \iff T \preceq del_O^{-1}(D)(T')$.

If, on the other hand, $(T')^{MaxEnv}$ is a combinational relation structure, our reasoning is as follows. According to Lemma 4.2, $del^{-1}(D)((T')^{MaxEnv}) \cap T$ is failure-free if and only if $(T')^{MaxEnv} \cap del(D)(T)$ is failure-free. By Theorem 4.31, since $(T')^{MaxEnv}$ is a combinational relation structure, $(T')^{MaxEnv} \cap del(D)(T)$ is failure-free if and only if $del(D)(T) \preceq T'$.

Because $(T')^{MaxEnv}$ is a combinational relation structure, T' is not a combinational autofailure. Hence by Lemma 4.36, $del_O^{-1}(D)(T')$ is also not a combinational autofailure. Therefore $(del_O^{-1}(D)(T'))^{MaxEnv}$ is a combinational relation structure too. Therefore, again by Theorem 4.31, $(del_O^{-1}(D)(T'))^{MaxEnv} \cap T$ is failure-free if and only if $T \preceq del_O^{-1}(D)(T')$.

Therefore in order to prove that $del(D)(T) \preceq T' \iff T \preceq del_O^{-1}(D)(T')$, it suffices to prove that $del^{-1}(D)((T')^{MaxEnv}) = (del_O^{-1}(D)(T'))^{MaxEnv}$.

$$\begin{array}{rcll} del^{-1}(D)((T')^{MaxEnv}) & = & del^{-1}(D)(mir(canon(T'))) & \\ & = & mir(del_O^{-1}(D)(canon(T'))) & \text{by Lemma 4.39} \\ & = & mir(canon(del_O^{-1}(D)(T'))) & \text{by Lemma 4.38} \\ & = & (del_O^{-1}(D)(T'))^{MaxEnv} & \end{array}$$

Therefore in all cases, $del(D)(T) \preceq T' \iff T \preceq del_O^{-1}(D)(T')$. ∎

Lemma 4.41 *Let T and T' be combinational relation structures such that $T \parallel T'$ is well-defined. Then*

$$T \parallel T' \textit{ is failure-free} \iff canon(T) \parallel T' \textit{ is failure-free}$$

Proof:

By Theorem 4.15, $T \sim canon(T)$. Therefore, by two applications of Theorem 4.4, for every T' a combinational relation structure such that $O \cap O' = \emptyset$, $T \parallel T'$ is failure-free if and only if $canon(T) \parallel T'$ is failure-free. ∎

4.3.4 Applications and Examples

In this section, we have addressed the problem of determining the *correctness* of replacing one subcircuit by another (or of designing a particular subcircuit for which there is no predecessor), relative to a given specification for the behavior of the full circuit. Our solution to this problem provides a more *general* solution than has previously been available for existing problems in the areas of logic synthesis and rectification.

In this subsection, we provide examples of how our results can be applied to known problems in hardware design, and how they generalize the known solutions to these problems. The main result of the previous subsections provides a fully general behavioral characterization of the allowed substitute circuitry for a subcircuit. We show how this information is useful for logic synthesis (Section 4.3.4.1) and rectification (Section 4.3.4.2).

4.3.4.1 Logic Synthesis

The optimization problem for logic synthesis in which we are interested is that of determining the precise parameters of an acceptable substitution for a subnetwork. Our formalism turns out to provide an exact method for the determination of *don't cares* in multiple-vertex optimization, that takes into account output correlations. In addition, the same description indicates what the substitution circuitry *must* do.

Given a logic network representing a multiple-level combinational circuit, the logic optimization problem is to optimize one or more nodes of the network according to cost metrics that correspond approximately to the actual area, delay, testability or power consumption of the subsequent bound version of the network. Currently, the only general methods for defining the parameters of an acceptable substitution (without taking the cost metrics into account) are don't care sets [20] and Boolean relations [22]. Both of these have been presented only in the context of acyclic networks and subnetworks.

Our method allows the expression of the precise parameters of an acceptable substitution for the most general multiple-vertex case: because we allow cycles in our circuit descriptions, an arbitrary collection of nodes of the logic network may comprise the subnetwork under investigation. In addition, the logic network itself may contain cycles.

Formally, given a logic network and an arbitrary marked set of nodes (a subnetwork) within it represented as relation structures, we seek to indicate the most general behavioral specification for acceptable substitute circuitry to replace the indicated subnetwork. But this is precisely the problem of conformance relative to a specific environment. Therefore, we may apply Theorem 4.34 directly to derive this most general specification: If $E[T'']$ is the original logic network, and T'' the subnetwork to be optimized, Theorem 4.34 tells us precisely which relation structures T' may be safely substituted for T'' in the network.

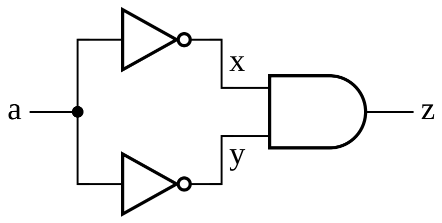

Figure 4.3: Substitution example

Example 4.5 *Consider the circuit depicted in Figure 4.3. We wish to determine the full range of possible substitutions for the and-gate, such that the circuit maintains its current input-output* *behavior.*

The following combinational relation structure represents the circuit in its current form:

$$T = T_{inv-ax} \parallel T_{inv-ay} \parallel T_{and-xyz}$$

Its designated components T_{inv-ax}, T_{inv-ay}, and $T_{and-xyz}$ take the obvious forms shown in previous examples.

We seek to find the allowed substitutions T' for $T_{and-xyz}$ in T. According to Theorem 4.34, any T' that conforms to $T'_{max} = f((T_{inv-ax} \parallel T_{inv-ay} \parallel \alpha_{\{x,y\},\{z\}}), T)$ may be so substituted.

$$f((T_{inv-ax} \parallel T_{inv-ay} \parallel \alpha_{\{x,y\},\{z\}}), T) = f((T_{inv-ay} \parallel \alpha_{\{x,y\},\{z\}}), (T^{MaxEnv} \parallel T_{inv-ax})^{MaxEnv})$$

is equal to

$$f(\alpha_{\{x,y\},\{z\}}, (del(\{a\})(((T^{MaxEnv} \parallel T_{inv-ax})^{MaxEnv})^{MaxEnv} \parallel T_{inv-ay}))^{MaxEnv})$$

which is equal to

$$(del(\{a\})(((T^{MaxEnv} \parallel T_{inv-ax})^{MaxEnv})^{MaxEnv} \parallel T_{inv-ay}))^{MaxEnv}$$

which (by Lemma 4.22) is equivalent to

$$(del(\{a\})(canon(T^{MaxEnv} \parallel T_{inv-ax}) \parallel T_{inv-ay}))^{MaxEnv}$$

which reduces to the following combinational relation structure:

$$T'_{max} = (\{x, y\}, \{z\}, S = \{xyz, \overline{x}\,\overline{y}\,\overline{z}\} \cup (x \neq y), F = \emptyset)$$

Note that expansion and input-downward-closure of the part of S that is expressed in equational notation result in the inclusion of the extended monomial $\perp_x \perp_y X_z$ in S.

Thus the set of relation structures T' that may be substituted for the and-gate in this circuit includes all those whose F-set is empty and whose S-set is one of $z = x$, $z = y$, or $z = x + y$, among other possibilities. These correctly indicate that the and-gate may be replaced by an or-gate, or a wire from one of the nodes x or y, without affecting the input-output behavior of the full circuit.

Example 4.6 *Consider the circuit in Figure 4.4. We wish to determine the full range of substitutions for the composite subcircuit consisting of the and- and or-gates in this circuit.*

A naive approach to the problem would be to define the following combinational relation structure

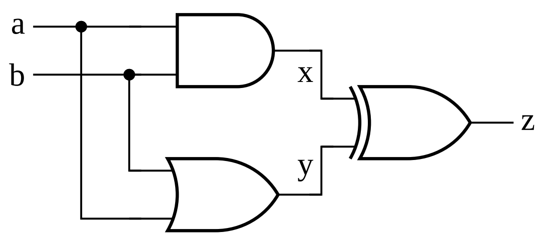

Figure 4.4: Substitution and rectification example

to represent the full circuit:

$$T_{full} = (\{a, b\}, \{x, y, z\}, S_{full} = ((z = (x \oplus y) \wedge (y = a + b) \wedge (x = ab)), F_{full} = \emptyset)$$

However, if we use this circuit representation we derive that the current implementation is the only possible substitution for itself. If we are to derive any benefit from this analysis, we must make explicit the fact that we are concerned only with the values on the input wires a and b and the output wire z. We achieve this effect by hiding x and y :

$$T = del(\{x, y\})(T_{full}) = (\{a, b\}, \{z\}, S_{full} = (z = (a + b) \oplus ab), F_{full} = \emptyset)$$

We may now compute a meaningful "upper bound" (in $\preceq$) for the allowed substitute circuitry for the indicated subcircuit.

$$T'_{max} = f(del(\{x, y\})(T_{xor-xyz} \parallel \alpha_{\{a,b\},\{x,y\}}), T)$$

Expansion as per the definition of f leads to the following result:

$$T'_{max} = (del(\{z\})((del_O^{-1}(\{x, y\})(T))^{MaxEnv} \parallel T_{xor-xyz}))^{MaxEnv} = (\{a, b\}, \{x, y\}, S'_{\mathrm{max}}, \emptyset)$$

where

$$S'_{max} = \{\overline{a}\overline{b}\overline{x}\,\overline{y}, \overline{a}\overline{b}xy, ab\overline{x}\,\overline{y}, abxy, \overline{a}b\overline{x}y, \overline{a}bx\overline{y}, a\overline{b}\overline{x}y, a\overline{b}x\overline{y}, \perp_a X_b X_x X_y\,, X_a \perp_b X_x X_y\}$$

In other words, any subcircuit which enforces the constraint that **a** *and* **b** *have the same Boolean value if and only if* **x** *and* **y** *have the same Boolean value may be substituted for the and- and or-gate combination in the current circuit. For example, an implementation in which* **x=a** *and* **y=b** *would be a correct substitute $T' \preceq T'_{max}$.*

4.3.4.2 Rectification

Our results also provide a new, fully general solution to the rectification problem. As illustrated clearly by Theorem 4.33, the results of Section 4.3.2 can also be applied when the specification for the full circuit is not a model of an already-existing circuit. Therefore we can apply this theorem directly to the logic redesign problem introduced in Section 1.5.2, where it provides a more general solution than has previously been available.

Example 4.7 *Consider again the circuit of Figure 4.4. Assume the specification for this circuit has been changed. We consider two distinct modified specifications. In both cases, we wish to keep the and- and or-gates in the circuit unchanged, and to resynthesize the xor-gate to meet the new modified specification.*

In our first rectification example, the specification has been changed to

$$T_{new} = T_{and-abz} = (\{a, b\}, \{z\}, S = (z = ab), F = \emptyset)$$

As stated above, we are interested in meeting this new specification by modifying only the xor-gate part of the current circuit. Thus we seek to characterize the circuitry T' that may be substituted for the xor-gate in order to meet the new specification. The most general characterization of T' is

$$T'_{max} = f(del(\{x, y\})(T_{and-abx} \parallel T_{or-aby} \parallel \alpha_{\{x,y\},\{z\}}), T_{new})$$

Expansion of this term according to the definition of f leads to the following combinational relation structure:

$$T'_{max} = (del(\{a, b\})((del_O^{-1}(\{x, y\})(T_{new}))^{MaxEnv} \parallel (T_{and-abx} \parallel T_{or-aby})))^{MaxEnv}$$

which reduces to

$$T'_{max} = (\{x, y\}, \{z\}, S'_{max}, \emptyset)$$

where

$$S'_{max} = \{xyz, \overline{x}y\overline{z}, x\overline{y}X_z, \overline{x}\,\overline{y}\,\overline{z}, \bot_x yX_z, x\bot_y X_z, \bot_x\overline{y}X_z, \overline{x}\bot_y\overline{z}, \bot_x\bot_y X_z\}$$

Thus for example $T'_i \preceq T'_{max}$ for the relation structures

$$T'_1 = (\{x, y\}, \{z\}, (z = xy), \emptyset)$$

and

$$T'_2 = (\{x, y\}, \{z\}, (z = x), \emptyset)$$

Either of these may be substituted for the xor-gate in the circuit of Figure 4.4 in order to meet the new specification for the full circuit.

Next we consider the case in which the specification has been changed to

$$T_{new} = (\{a, b\}, \{z\}, S = (z = a\overline{b}), F = \emptyset)$$

Again,

$$T'_{max} = f(del(\{x,y\})(T_{and-abx} \parallel T_{or-aby} \parallel \alpha_{\{x,y\},\{z\}}), T_{new})$$

which would expand to

$$T'_{max} = (del(\{a,b\})((del_O^{-1}(\{x,y\})(T_{new}))^{MaxEnv} \parallel (T_{and-abx} \parallel T_{or-aby})))^{MaxEnv}$$

except that $T_0 = del(\{a,b\})((del_O^{-1}(\{x,y\})(T_{new}))^{MaxEnv} \parallel (T_{and-abx} \parallel T_{or-aby}))$ *is a combinational autofailure and hence instead* $T'_{max} = (I, O, \emptyset, \emptyset)$. *Specifically,* T_0 *has the form* $(\{z\}, \{x,y\}, S_0, F_0)$ *where* $S_0 = (\{X_x X_y X_z\} - F_0)$, *and*

$$F_0 = \{xyz, xy\bot_z, \overline{x}yX_z, \overline{x}\,\overline{y}z, \overline{x}\,\overline{y}\bot_z, \bot_x yz, \bot_x y\bot_z, \overline{x}\bot_y z, \overline{x}\bot_y\bot_z\}$$

Because every legal environment of this relation structure must admit some output value combination on input value combination $\overline{x}\mathbf{y}$, *there is no legal environment of* T_0 *whose composition with* T_0 *is failure-free. Hence* T_0 *is a combinational autofailure.*

Thus we see that it is possible to modify a specification to such an extent that if we restrict ourselves to only modifying a certain piece of the existing circuit, without changing any of the wires of the circuit, there may be no rectification solution. Our method does identify such situations, as we have just shown.

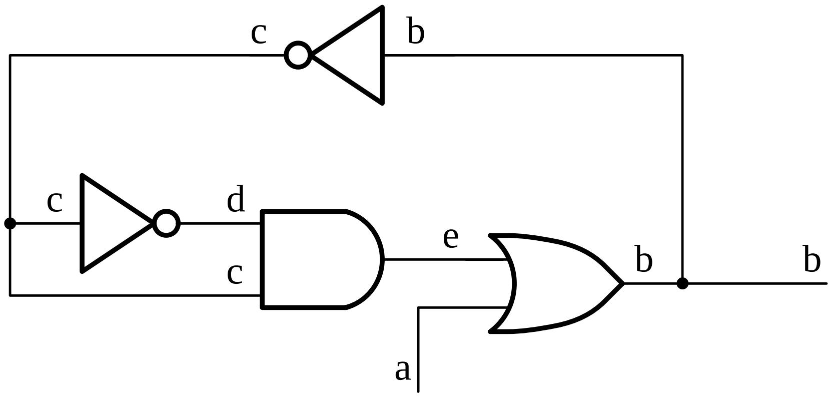

Figure 4.5: Combinational feedback loop (Fig. 6 of [92])

Example 4.8 *Consider the circuit illustrated in Figure 3.4 on page 73, which is repeated here in Figure 4.5 [92]. We will use this circuit to illustrate the capability of combinational relation structures to handle multi-vertex optimization even in the case that the vertex-set chosen for resynthesis is part of a combinational feedback loop.*

In Example 3.7 on page 73, we determined that the following combinational relation structure represents this circuit:

$$T = (\{a\}, \{b\}, \{ab, \overline{a}\overline{b}, \overline{a}\bot_b, \bot_a b, \bot_a \overline{b}, \bot_a \bot_b\}, \emptyset)$$

We wish to determine the class of all possible substitutions for the subcircuit consisting of both the inverter with input wire labeled **b** *and output wire labeled* **c** *and the and-gate. Thus we seek to determine the class of combinational relation structures T' that conform to*

$$T'_{max} = f(del(\{c,d,e\})(T_{or-aec} \parallel T_{inv-cd} \parallel \alpha_{\{b,d\},\{c,e\}}), T)$$

Expansion of this function application according to the definition of f yields the following term:

$$T'_{max} = (del(\{a\})((del_O^{-1}(\{c,d,e\})(T))^{MaxEnv} \parallel (T_{or-aec} \parallel T_{inv-cd})))^{MaxEnv}$$

which reduces to

$$T'_{max} = (\{b,d\}, \{c,e\}, S'_{max}, F'_{max})$$

where

$$S'_{max} = \{X_b\overline{d}c\overline{e}, X_b d\overline{c}\overline{e}, X_b X_d \bot_c \overline{e}, b\overline{d}ce, bd\overline{c}e, bX_d\bot_c e, X_b\overline{d}c\bot_e, X_b d\overline{c}\bot_e, X_b X_d \bot_c \bot_e\}$$

(which is simply $del^{-1}(\{b,e\})S_{inv-cd} \cap del^{-1}(\{c,d\})\{X_b\overline{e}, be, X_b\bot_e\}$), and

$$F'_{max} = \overline{S'_{max}} = \{X_b X_d X_c X_e\} - S'_{max}$$

Thus appropriate T' include, among others, replacement of the inverter with input wire **b** *and output wire* **c** *with a wire (i.e., we may combine the nodes* **b** *and* **c** *into a single equipotential region without adversely affecting the behavior of the circuit).*

We also note that according to T'_{max}, it is permissible to substitute an or-gate for the and-gate in the current circuit. Tracing the consequences of this substitution in the circuit itself, we see that it allows output value **b** $= 1$ *on input value* **a** $= 0$*, which is not a possible behavior of the original circuit. The reason for this apparent counterintuitive result is output-upward-closure of the specification: when we canonicalized T we added the behavior* $\overline{\mathbf{a}}\mathbf{b}$ *to its S-set via output-upward-closure of its P-set. This is consistent with our assumption that the appearance of a $\bot$ value on an output wire in a specification indicates that any of the values $0, 1$ or $\bot$ may appear in its place in the allowed implementations.*

Example 4.9 *In their mixed-mode approach to combinational rectification, Watanabe and Brayton present the problem of constructing additional circuitry to be added around already-existing circuitry*

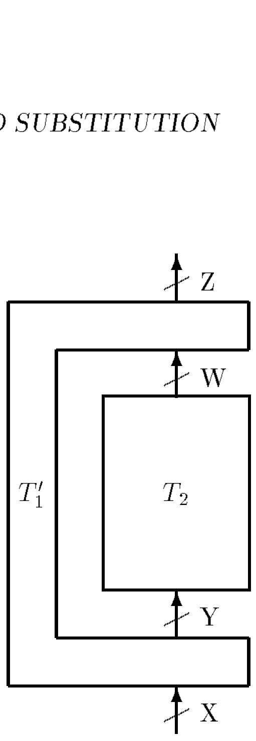

Figure 4.6: Mixed-mode rectification

in order to meet a modified specification [135]. This approach is illustrated in Figure 4.6. If $T_2 = (W, Y, S_2, F_2)$ *represents the original existing circuitry then* $T_1' = ((X \cup Y), (W \cup Z), S_1', F_1')$ *represents the circuitry added in order that the new combined circuit meet the modified specification. Formalized in terms of combinational relation structures,* $T = del(Y \cup W)(T_1' \parallel T_2)$ *represents the rectified circuit that meets the modified specification.*

In our hierarchical approach to the problem, we consider T_2 *to be the original specification of the problem, and* T_1' *to be the most general characterization of allowed additions to correct implementations of* T_2 *such that the composition of* T_1' *and* T_2 *(with communication wires suitably hidden) is a combinational relation structure representation of the rectified specification. That is, if* T *is the rectified specification, we derive* $T_1' = f(del(Y \cup W)(T_2 \parallel \alpha_{(X \cup W),(Y \cup Z)}), T)$*, which is the most general characterization of allowed correction circuitry. Then any* $T' \preceq T_1'$ *is a correct implementation for the rectification circuitry, just as any implementation* $T_0 \preceq T_2$ *is a correct implementation of the original specification. Note that if* T_2 *is a deterministic implementation, the problem we address is equivalent to that discussed above; in this case as well, however, our solution is more general than theirs.*

The solution T_1' *derived in [135] is not the most general possible solution Boolean relation. Instead, in order to support the removal of combinational feedback loops from the composition of* T_2 *and* T_1' *implementations, Watanabe and Brayton present a more conservative estimate of allowed rectification circuitry. In their own notation, if* H *is the Boolean relation representing the modified specification and* T_2 *the Boolean relation representing the already-existing circuitry, they define the*

Boolean relation corresponding to T_1' to be G such that

$$(X \cup Y \cup W \cup Z) \in G \Longleftrightarrow [[(X \cup Z) \in H \ \textit{and} \ (Y \cup W) \in T_2] \ \textit{or} \ \forall Y'.(Y' \cup W) \notin T_2]$$

However, the most general solution in their terms would be G such that

$$(X \cup Y \cup W \cup Z) \in G \Longleftrightarrow [(X \cup Z) \in H \ \textit{or} \ (Y \cup W) \notin T_2]$$

which corresponds precisely to our solution

$$T_1' = f(\,del(Y \cup W)(T_2 \parallel \alpha_{(X \cup W),(Y \cup Z)}), T) = (\,del_O^{-1}(Y \cup W)(T))^{MaxEnv} \parallel T_2)^{MaxEnv}$$

Note that our T is their H, and that the $\parallel$ operator corresponds to conjunction of Boolean relations and the MaxEnv operator corresponds to complementation of Boolean relations. Strip away the alphabet information (del_O^{-1}) and apply De Morgan's Law once, and the correspondence is immediate.

Thus we see that allowing the creation of combinational feedback loops leads to more flexibility in characterizing allowed rectification circuitry than is provided by approaches in which disallowing such loops must be a concern.

Chapter 5

Synchronous circuit models

5.1 Introduction

In this chapter, we extend the theory we have developed for combinational circuits to full synchronous, sequential designs. Recall that our original purpose was to develop a model of synchronous circuits that would support both formal verification and substitution of such circuits. We determined that in order to meet these goals, our model should support nondeterminism, and hierarchical construction and modular description of synchronous circuits. In the previous chapter, we were able to apply the circuit algebra framework of Chapter 2 to achieve these goals in modeling combinational circuits. This required incorporating a new ternary domain of wire values into the model, in order to allow both nondeterminism and an algebraic composition operator. We also identified constraints on the use of the new wire value, $\perp$, sufficient to guarantee that any combinational model be receptive to any vector of input values. In this chapter, we incorporate those results into a trace theoretic model of synchronous circuits.

Our model of the behavior of a synchronous circuit or specification consists of a prefix-closed set of sequences of combinational circuit behavioral descriptions, expressed as a possibly nondeterministic Mealy machine. Latches are primitive models; they cannot be created from combinational circuitry.

As in the combinational theory, we define a formal relation between models that corresponds to one being a correct implementation of the other (considered as a specification). Because of the prefix-closed nature of our models, this relation only deals with *safety properties* of the specification and implementation. While the constraints on a model can be modified to allow the expression of arbitary *liveness properties* as well, we have not pursued that extension in this thesis.

We define a sequential circuit representation to correctly implement a specification if the implementation can be safely substituted for the specification. In addition, a $\perp$ value on an output wire of a specification under particular circumstances actually allows the implementation to output *any* value on that wire under those circumstances. We provide a decision procedure for the formal relation that holds between a specification and any correct implementation thereof. As in the combinational case, we may use this relation to fully characterize the set of allowed substitutions for a subcircuit.

The organization of this chapter is as follows. In the following section, we present the formal definition of a sequential trace structure, which is our representation of a specification or implementation of a synchronous circuit. In Section 5.3, we define the algebraic operations for these models and prove the closure of the class of sequential trace structures under these operations, and show that this class together with these operators forms a circuit algebra. In section 5.4, we present the full definition of when a sequential trace structure describes an acceptable implementation of a specification, and outline a decision procedure for this relation. Finally, in Section 5.5 we discuss our substitution results for synchronous circuits. Examples are provided at each step of the development.

5.2 The synchronous circuit model

5.2.1 Introduction

We seek to develop models of sequential, synchronous circuits that support hierarchical construction and modular description of such circuits, and that support nondeterminism. The latter enables the expression of a specification which captures the minimum requirements of a circuit instead of requiring that we overspecify by including irrelevant implementation details. We saw in Chapter 4 that we can achieve all these goals for *combinational* circuits using relational models. In this section, we show how these ideas can be extended to provide a trace theoretic model for *synchronous* circuits.

We model the set of input-output behaviors of a synchronous circuit or circuit specification as a regular, prefix-closed set of finite sequences of combinational circuit behavioral descriptions. Digital hardware is inherently finite-state. Therefore the restriction to regular sets, which are precisely those sets of finite sequences that can be expressed by a finite-state automaton, is reasonable. The prefix-closure constraint, and the fact that we allow only finite-length sequences, mean that our specifications can express only safety properties.

Safety properties state what an implementation may not do, but cannot say anything about what it *must* do [89]. In a clocked system, this distinction breaks down somewhat, as a deterministic Mealy machine certainly specifies required responses. However, it cannot express *unbounded* response properties, which state that a certain reaction must occur some *unspecified* time in the future.

Liveness properties state what an implementation *must* do. For example, the unbounded-response property 'if R then eventually P' is a liveness property. (This propositional temporal logic sentence represents a specification requirement such as 'if a request is made for a resource then the resource is eventually granted to the requestor'). The key to the additional expressiveness of this sentence over safety properties is its use of the word "eventually". Such properties are useful for high-level system specifications, in which the relative speeds of the subprocesses are as yet undetermined.

We can express all *safety* properties of circuits using finite traces. Semantically, safety properties are those properties which hold of an infinite sequence if and only if every prefix of it can be extended to a sequence with the property [2, 3]. In other words, all *partial behaviors* obey the constraint. Therefore prefix-closure is an appropriate *syntactic* constraint on a set of finite sequences representing a safety property.

The state of a sequential circuit is the vector of values currently held on its latches. However, the value to which a primary output stabilizes within the clock cycle may depend on the current input values as well as on the values at the state-holding nodes of the circuit. Therefore we model these circuits using Mealy machines.

5.2.2 Sequential trace structures

In this section we present our formal model of synchronous circuits and their specifications. We call these models *sequential trace structures.*

Sequential trace structures differ from combinational relation structures only in that their S, F and P-sets are sets of *traces,* or sequences of input-output value combinations. A trace is intended to model a possible behavior of a synchronous circuit over time. Each input-output value combination in a trace models the synchronous circuit's behavior during a single clock cycle, and the sequence of such value combinations denotes its behavior during consecutive clock cycles. Each trace can be thought of as the circuit's response to a given sequence of input-value combinations.

The constraints we place on the F and P-sets of a sequential trace structure reflect the same concerns as in the combinational case. The combinational receptiveness constraint is enforced *per clock cycle* (to ensure that the model correctly reflects receptiveness of a circuit to all input-value combinations during each clock cycle): following any allowed behavior of arbitrary finite length the model is receptive to all of the input vectors of $\mathcal{T}^I$. Input-downward closure now allows for the propagation through time *(i.e.,* during this *and* subsequent clock cycles) of the result of having mistaken a $\perp$ value on an input wire of a circuit for digital 0 or digital 1 during the current clock cycle.

Recall that $\mathcal{T}^Y$ is the set of all possible assignments to elements of Y of values from $\mathcal{T}$. We define $\mathcal{B}(Y) = (\mathcal{T}^Y)^*$ to be the set of all finite sequences of such assignments. Note that $(\mathcal{T}^Y)^* = \bigcup_{n\in\omega}(\mathcal{T}^Y)^n$ is isomorphic to $\bigcup_{n\in\omega}(\mathcal{T}^n)^Y$, because $(\mathcal{T}^Y)^n$ is isomorphic to $(\mathcal{T}^n)^Y$. (The isomorphism maps $w = (w_1 \cdot w_2 \cdot \ldots \cdot w_n) \in (\mathcal{T}^Y)^n$ to $w' : Y \longrightarrow \mathcal{T}^n$ such that $w'(y) = (w_1(y) \cdot w_2(y) \cdot \ldots \cdot w_n(y))$). We will often interchange these two types as the definition of $\mathcal{B}(Y)$.

The function union operator extends naturally to sequences: the function union of two sequences $b \in \mathcal{B}(B)$ and $c \in \mathcal{B}(C)$ of equal length n is $(b \cup c) \in \mathcal{B}(B \cup C)$ of length n such that $(b \cup c)[i] = (b[i] \cup c[i])$ for every positive integer $i \leq n$.

We extend the definedness order $\leq$ pointwise from vectors of wires to sequences of vectors of wires. Recall that for $w, w' \in \mathcal{T}^B$, we say that $w \leq w'$ if and only if $\forall b \in B . w(b) \leq w'(b)$, where for $x, x' \in \mathcal{T}$, $x \leq x'$ if and only if $x = x'$ or $x = \perp$. We now define the pointwise extension of the partial order $\leq$ to finite sequences of such vectors. If $w, w' \in \mathcal{B}(B)$ are of equal length n, then we say that $w \leq w'$ if and only if $\forall i \in \{1, 2, \ldots, n\}.w[i] \leq w'[i]$. Note that because $w \leq w'$ cannot hold for w and w' of unequal length, the statement that $w \leq w'$ *implies* that $len(w) = len(w')$.

Using this notation, we now define the *input-downwards-closure* and *receptiveness* constraints for sets of sequential behaviors. As in the combinational case, we assume that I are the input wires of the circuit or specification and that O are its primary output wires, and we define $A = (I \cup O)$. Formally, the *input-downwards-closure (IDC)* property holds of the set $W \subseteq \mathcal{B}(I \cup O)$ (written $IDC(W)$) if and only if

$$\forall n \in \omega . \forall x, x' \in (\mathcal{T}^I)^n . \forall y \in (\mathcal{T}^O)^n . (x' \leq x \wedge (x \cup y) \in W) \Longrightarrow ((x' \cup y) \in W)$$

The receptiveness condition is defined as follows. A set $P \subseteq \mathcal{B}(I \cup O)$ is *receptive* if and only if for every $w \in P$, there exists $C \subseteq ext_1(w, P)$ such that

- C is total:
$$\forall x \in \mathcal{T}^I . \exists y \in \mathcal{T}^O . (x \cup y) \in C$$
- C has the upward-chains property:
$$\forall x, x' \in \mathcal{T}^I . \forall y \in \mathcal{T}^O . [[(x \cup y) \in C \wedge x \leq x'] \Longrightarrow \exists y' \in \mathcal{T}^O . y \leq y' \wedge (x' \cup y') \in C]$$

Formally, we define a *sequential trace structure* (sometimes referred to simply as a trace structure) to be a quadruple $T = (I, O, S, F)$ such that

- I and O are disjoint finite sets,
- $F \subseteq \mathcal{B}(I \cup O)$ is a regular (possibly empty) set of circuit behaviors ("traces"), known as the *failure set* of the trace structure, which obeys the *input-downwards-closure* constraint,
- $S \subseteq \mathcal{B}(I \cup O)$ is a prefix-closed, regular (possibly empty) set of circuit behaviors ("traces"), known as the *success set* of the trace structure, and
- $P = S \cup F$, the set of *possible traces* of T, is prefix-closed, regular, and non-empty, and obeys both the input-downwards-closure constraint and the receptiveness constraint.

Note that, in contrast to the receptiveness condition for combinational relation structures, the sequential receptiveness condition does *not* preclude P's being the empty set. Hence it is necessary to maintain the explicit requirement that P be non-empty.

We define a preorder $\sqsubseteq$ on sequential trace structures having the same I and O sets:

$$(I, O, S, F) \sqsubseteq (I, O, S', F') \text{ iff } ((F \subseteq F') \wedge (P \subseteq P'))$$

Later we will extend this preorder to structures $(I, O, \emptyset, \emptyset)$ as well (see page 147 in Section 5.4.4). This preorder induces a partial order among its equivalence classes. If $F = F'$ and $P = P'$ then $[T]_{\sqsubseteq} = [T']_{\sqsubseteq}$, and we write $T \sim_{\sqsubseteq} T'$. Essentially, T and T' are in the same $\sqsubseteq$-equivalence class if they vary only in the amount of overlap between their S and F sets.

We may sometimes use the ambiguous notation

$$T = (I, O, F, P)$$

in place of (I, O, S, F). In this case it is assumed that S, although not specified, is prefix-closed. Note that we cannot simply assume that setting $S = (P - F)$ defines a unique representative of the $\sqsubseteq$-equivalence class (I, O, F, P), as this set need not be prefix-closed. In addition, when discussing receptiveness alone, we may sometimes use the notation $T = (I, O, P)$, ignoring the F-set of the trace structure altogether.

In the following section, we present the formal definitions of the algebraic operations of composition, renaming, and projection. We also prove that the class of sequential trace structures together with these operations forms a circuit algebra. We conclude the current section with some examples.

5.2.3 Examples

In this subsection we present examples of sequential trace structures that illustrate our modeling technique and further explain how to use the two-language model. We also use these examples to introduce some more notational conventions. In addition to the notational conventions introduced in the previous chapter for presentation of combinational relation structures, we utilize the usual graphical conventions for representation of finite-state automata.

Our first example presents a basic component which will appear in many of the examples in this chapter: the edge-triggered D-flipflop. Following [90], our basic circuit and specification components are combinational parts and D-flipflops, which simply delay by one clock cycle the flow of their input signal. Our second example illustrates that a sequential trace structure representation of a combinational component with an empty F-set has S- (and P-) set consisting of arbitrary finite numbers of repetitions of its combinational behavior. Formally, its S-set is the language defined by a single-state automaton with a self-loop labeled with the traces of a combinational relation structure representation of the component. In our third example, we illustrate how this generalizes to combinational parts with non-empty F-sets.

Finally, as the fourth example in this subsection, we provide a sequential trace structure representation of the specification for a clocked SR flip-flop. This example illustrates meaningful use of the F-set to denote the possibility of asynchronous hazard behavior in a known simple sequential circuit.

We present the S-, F- and P- sets of a sequential trace structure, all of which are regular sets, by

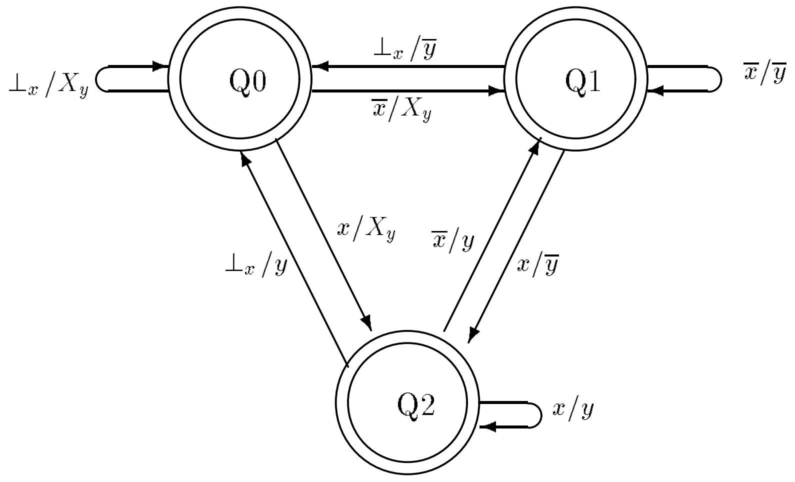

Figure 5.1: Automaton representation of the behavior of a D-flipflop

presenting finite-state automata which accept them. We utilize the usual graphical conventions in this presentation: concentric circles indicate final states, an arrow-head pointing to a state indicates it is a start state, and edge labels indicate the "symbols" of the accepted traces (each symbol is a vector). Note that we add to the notational conventions introduced in the previous chapter, by separating the input wire values from the output wire values of a single combinational behavior by the symbol "/" as is usual in the representation of Mealy machines. In addition, we may sometimes attach a list of distinct labels to a single edge, as shorthand for multiple edges with the same source and target states. In this case these labels will either appear on separate lines or will be separated by commas.

We may sometimes represent both the S- and F-sets by a single automaton. In this case, each final state is marked with S or F (or both) to indicate that it is a final state in the automaton representation of S or F, respectively. We call such an automaton representing both the S-set and the F-set of a sequential trace structure a *combined automaton.*

Example 5.1 *The following sequential trace structure represents a D-flipflop. Following Bronstein [24], we define these single-bit registers according to the value they present on their output wire in the first clock cycle. R_b is the register with output wire labeled b whose first output is the value* 1*. If its first output is the value* 0*, we call it $R_{\overline{b}}$. Similarly, $R_{\perp_b}$ produces as its first output the value* $\perp$ *. (Of course, this value may be perceived as a* 0 *or a* 1*, depending on the actual oscillation pattern exhibited). On subsequent cycles, these registers pass through to their output wire b whatever value was received (after stabilization, if relevant) on the input wire in the previous clock cycle.*

The automaton of Figure 5.1 represents the potential P-sets of a one-bit register whose input

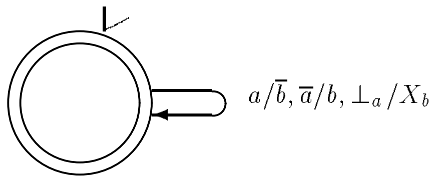

Figure 5.2: Automaton representation of the sequential behavior of an inverter

wire is labeled x and whose output wire is labeled y. Note that such a circuit has an empty F-set.

The diagram may represent any of R_y, $R_{\overline{y}}$ and $R_{\perp_y}$, depending on the state chosen as the start state. R_y has start state $Q2$ and $R_{\overline{y}}$ has start state $Q1$, while $R_{\perp_y}$ is defined with start state $Q0$. Note that the state $Q0$ represents a state of the circuit in which the previous cycle's input value is unknown as well as that in which it is undefined. Because of the input-downward-closure constraint, these two states are essentially indistinguishable. Thus if we choose to not identify the first output value of our register R, we may simply set the start state to $Q0$.

The following example illustrates how the sequential trace structure representation of a combinational component may be based on the combinational relation structure representation of its behavior.

Example 5.2 *The following sequential trace structure represents an inverter whose input wire is labeled* **a** *and whose output wire is labeled* **b**:

$$T_{inv-ab} = (I = \{a\}, O = \{b\}, S_{inv-ab}, F = \emptyset)$$

where S_{inv-ab} is the language accepted by the automaton of Figure 5.2.

Because it preserves no state, the sequential behavior of this circuit consists simply of unbounded repetition of its combinational behavior. Hence the minimal automaton representation of its sequential behavior contains a single state, which is both its start state and its final state, and accepts arbitrary-length finite sequences of combinational inverter behaviors.

Example 5.2 illustrated how to produce a sequential trace structure representation of a combinational circuit with an empty F-set. We have not yet shown how to handle a circuit specification with a non-empty F-set. In the following example we show that such combinational specifications are also easily extended to sequential trace structures.

Example 5.3 *The following sequential trace structure is a specification for an inverter that we expect to place only in an environment in which its input stabilizes. Its input wire is labeled* **a** *and its output wire labeled* **b**.

$$T_1 = (\{a\}, \{b\}, S_1, F_1)$$

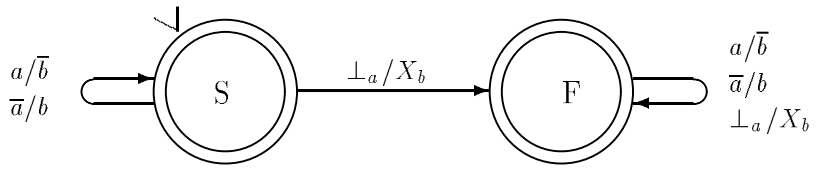

Figure 5.3: Dual-automaton representation of the sequential behaviors of a non-failure-free inverter

where S_1 and F_1 are the languages accepted by the combined automaton of Figure 5.3.

The combinational version of this specification was discussed in Example 3.4 in Chapter 3, where we provided the following combinational relation structure representation for it:

$$T_0 = (I = \{a\}, O = \{b\}, S = \{a\overline{b}, \overline{a}b\}, F = \{\perp_a X_b\})$$

From this description it is clear how we have constructed the sequential trace structure T_1 from the combinational relation structure T_0. Note that if a run of this deterministic automaton over a sequence accepts it as a failure trace, the sequence may not be extended to any success trace, because the S-set must be prefix-closed.

In the final example of this section, we present a sequential trace structure representation of the specification for a clocked SR-flipflop. Here the specification's F-set is not a simple extension of any combinational circuit's F-set. In the following sections we will discuss the construction of possible implementations for this specification in our model.

Example 5.4 *In this example we present a sequential trace structure which represents the specification for a clocked SR-flipflop. The excitation table for this latch specifies that when the S and R lines are both held low at the clock tick, the state of the latch, as reflected in its output line Q, does not change. If S is held high, the latch is set: Q goes high. If R is held low, the latch is reset: Q goes low. The situation in which both S and R are held high at the clock tick is disallowed.*

This specification can be represented by the sequential trace structure

$$T_{SRff} = (\{s, r\}, \{q\}, S_{SRff}, F_{SRff})$$

where S_{SRff} and F_{SRff} are described by the combined automaton of Figure 5.4.

5.3 The algebraic operations for sequential trace structures

In this section we discuss the circuit algebra operations for sequential trace structures. The results of applying these operations to sequential trace structures are themselves sequential trace structures.

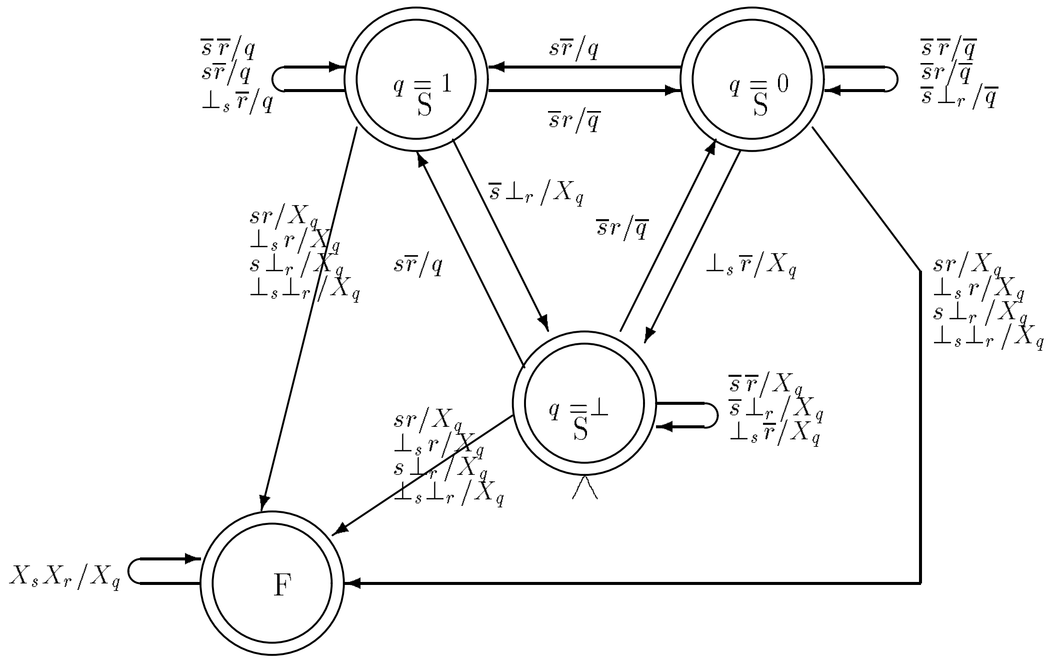
$\overline{s}\,\overline{r}/q$
$s\overline{r}/q$
$\perp_s\overline{r}/q$
$q = 1$
S
$s\overline{r}/q$
$\overline{s}r/\overline{q}$
$q = 0$
S
$\overline{s}\,\overline{r}/\overline{q}$
$\overline{s}r/\overline{q}$
$\overline{s}\perp_r/\overline{q}$
$\overline{s}\perp_r/X_q$
$\overline{s}r/\overline{q}$
sr/X_q
$\perp_s r/X_q$
$s\perp_r/X_q$
$\perp_s\perp_r/X_q$
$s\overline{r}/q$
$\perp_s\overline{r}/X_q$
sr/X_q
$\perp_s r/X_q$
$s\perp_r/X_q$
$\perp_s\perp_r/X_q$
$q = \perp$
S
$\overline{s}\,\overline{r}/X_q$
$\overline{s}\perp_r/X_q$
$\perp_s\overline{r}/X_q$
sr/X_q
$\perp_s r/X_q$
$s\perp_r/X_q$
$\perp_s\perp_r/X_q$
X_sX_r/X_q
F

Figure 5.4: Behavioral specification of a clocked SR-flipflop

The class of sequential trace structures together with the algebraic operations forms a circuit algebra. In Section 5.3.2, we present some examples of composite circuit models in order to illustrate how the formal operations implement the intended actions on the circuits being modeled.

5.3.1 Definitions, closure, and circuit algebra rules

The formal definitions of the algebraic operations on sequential trace structures are identical to those of combinational relation theory (see Section 3.3.2), except that the type of the behavior sets S and F to which the operators are applied differs. In order to define renaming, it is necessary to use the natural extension of the renaming function $r : A \longrightarrow B$ from vectors to sequences of vectors and to sets of such sequences.

We will now prove that the class of sequential trace structures is closed under application of these algebraic operators and that together they form a circuit algebra. Regularity of each of S, F, and P is preserved by the algebraic operations, by closure properties of regular languages. Prefix-closure of S and P is also preserved by the operations, as is non-emptiness of P ($\varepsilon \in P$ is preserved by all the operations). The proof that following the application of any of the algebraic operations, the required property of input-downwards-closure holds of both the resulting structure's F-set and its P-set, duplicates the proof for the combinational case: just replace $\mathcal{T}^Y$ by $\mathcal{B}(Y)$ for every set Y in the proof of Section 3.3.3.1.

We present our proof that the receptiveness constraint is preserved by all the algebraic operations. It is nearly identical to the proof for the combinational case (see Section 3.3.3.2). Because only the P-set of a sequential trace structure need be receptive, we can ignore the S and F-sets of the sequential trace structures in these proofs. Therefore we refer to a relation structure as $T = (I, O, P)$. The proof that receptiveness is preserved under the renaming operator is trivial and will not be given here. The proofs for the remaining operators appear below.

- Receptiveness is preserved by the hide operation:

  Let $T = (I, O, P)$ and $D \subseteq O$. For each $w \in P$, let $C_w \subseteq \mathit{ext}_1(w, P)$ be total (in $\mathcal{T}^I$) and have the upward-chains property.

  Let $T' = \mathit{del}(D)(T) = (I, O - D, \mathit{del}(D)(P))$.

  Let $w' \in \mathit{del}(D)(P)$. We must prove that there exists $C'_{w'} \subseteq \mathit{ext}_1(w', \mathit{del}(D)(P))$ that is total in $\mathcal{T}^I$ and has the upward-chains property.

  Pick some $w_0 \in \mathit{del}^{-1}(D)(\{w'\}) \cap P$ (by definition, there must exist such w_0).
  Let $C'_{w'} = \mathit{del}(D)(C_{w_0})$. Clearly, $C'_{w'} \subseteq \mathit{ext}_1(w', \mathit{del}(D)(P))$.

  The proof that $C'_{w'}$ is total in $\mathcal{T}^I$ and that $C'_{w'}$ has the upward-chains property may be derived by replacing C by C_w and C' by $C'_{w'}$ in the proof (for the combinational case) that C' is total in $\mathcal{T}^I$ and has the upward-chains property, which appears in Section 3.3.3.2. ∎

- Receptiveness is preserved by inverse deletion:

  Let $T = (I, O, P)$ and $(D \cap (I \cup O)) = \emptyset$. For each $w \in P$, let $C_w \subseteq \mathit{ext}_1(w, P)$ be total (in $\mathcal{T}^I$) and have the upward-chains property.

  Let $T' = \mathit{del}^{-1}(D)(T) = (I \cup D, O, \mathit{del}^{-1}(D)(P))$. For each $w' \in \mathit{del}^{-1}(D)(P)$, let $C'_{w'} = \mathit{del}^{-1}(D)(C_w)$ where $w' \in \mathit{del}^{-1}(D)(w)$. Clearly, $C'_{w'} \subseteq \mathit{ext}_1(w', \mathit{del}^{-1}(D)(P))$.

  We must prove that these $C'_{w'}$ are each total in $\mathcal{T}^{(I \cup D)}$ and each have the upward-chains property. Clearly, each $C'_{w'}$ is total, because every C_w is total in $\mathcal{T}^I$ and del^{-1} preserves totality. We proceed to prove that each $C'_{w'}$ has the upward-chains property.

  Let $w' \in \mathit{del}^{-1}(D)(P)$. By definition of T', there exists $w \in P$ such that $w' \in \mathit{del}^{-1}(D)(w)$. The rest of the proof is identical to the proof for the combinational case (in Section 3.3.3.2), with C_w substituted for C and $C'_{w'}$ substituted for C'. ∎

- Receptiveness is preserved by intersection:

  Let $T = (I, O, P)$ and $T' = (I', O', P')$ be sequential trace structures such that $(I \cup O) = (I' \cup O')$ and $(O \cap O') = \emptyset$. For each $w \in P$, let $C_w \subseteq \mathit{ext}_1(w, P)$ be total in $\mathcal{T}^I$ and have the upward-chains property. For each $w' \in P'$, let $C'_{w'} \subseteq \mathit{ext}_1(w', P')$ be total in $\mathcal{T}^{I'}$ and have the upward-chains property.

Let $T'' = T \cap T' = (I'', O'', P'')$. For each $w'' \in P''$, let $C''_{w''} = C_{w''} \cap C'_{w''}$. Clearly, $C''_{w''} \subseteq ext_1(w'', P'')$.

We will prove that these $C''_{w''}$ are each total in $\mathcal{T}^{I''}$ and each have the upward-chains property.

Let $w'' \in P'' = (P \cap P')$. The proof that $C''_{w''}$ is total in $\mathcal{T}^{I''}$ and that it has the upward-chains property is identical to the proof for the combinational case (in Section 3.3.3.2), with $C_{w''}$ substituted for C, $C'_{w''}$ substituted for C', and $C''_{w''}$ substituted for C''. ∎

Because we know that the regularity and input-downwards-closure of F and P, the regularity of S, the prefix-closure of S and P, and the non-emptiness of P are also preserved by the algebraic operations, this concludes our proof that the class of sequential trace structures is closed under the algebraic operations.

The class of sequential trace structures together with the algebraic operations of composition, renaming, and projection forms a circuit algebra. The proofs are identical to those for the combinational case, except that the type of the elements in S and F is different. We must extend renaming functions r to sequences of vectors and to sets of such sequences. However, in contrast to the situation in asynchronous trace theory, all operations are length-preserving (clock cycles cannot be collapsed), and so all the operations distribute over sequence concatenation and the concatenation of sets of sequences. All the lemmas of Section 3.3.4 hold in the sequential case as well.

5.3.2 Examples

In this section we present some examples of composite circuit models in order to illustrate the effects of the circuit algebra operations. We begin by walking through the composition of a simple circuit consisting of two gates and a latch. We then present models of some candidate implementation circuits for the SR flipflop specification given in the previous section. These circuits and their models illustrate that X-effects of gate-level ternary simulation are an accurate reflection of the intended semantics of the $\perp$ symbol.

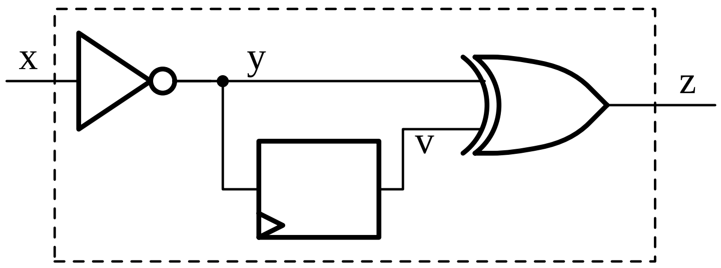

Figure 5.5: Simple circuit containing a latch

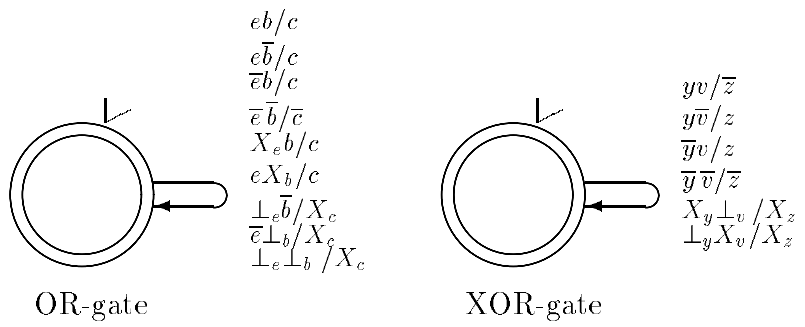

Figure 5.6: Automata representations of the sequential behavior of two gates

Example 5.5 *Consider the circuit illustrated in Figure 5.5. Its primary output is the wire labeled z, and its input is labeled x. Note that the wires y and v are hidden; they are not primary outputs of this circuit.*

We model this circuit as

$$T = del(\{y, v\})(INV1 \parallel (L1 \parallel XOR1))$$

where sequential trace structure representations of the component gates INV1 and XOR1 and the component latch L1 are as follows:

- *$INV1 = ren(r)(T_{inv-ab})$ is the inverter model of Example 5.2 (page 126) in Section 5.2.3, appropriately renamed.*
- *$XOR1 = (\{y, v\}, \{z\}, S_{xor-yvz}, \emptyset)$ – see Figure 5.6 for an automaton representation of $S_{xor-yvz}$.*
- *$L1 = ren(r')(T_{Dff-xy})$, is the $R_{\bot_x}$ model of Example 5.1 in Section 5.2.3, appropriately renamed.*

Automaton representations of the behavior of each of these components or their renaming appear in Figures 5.2, 5.6, and 5.1, respectively. They all have empty F-sets.

The creation of the composite model for this circuit proceeds in several stages. We present the intermediate sequential trace structures derived during one possible version of this process. Note that the composition may be done in any order, as composition is associative and commutative, and that a wire may be hidden as soon as it is no longer required to participate in any future compositions. We choose to order our operations as indicated by the definition of the full circuit T, above.

$$T_1 = (L1 \parallel XOR1) = (\{y\}, \{v, z\}, S_1, \emptyset)$$

where S_1 is the language accepted by the automaton of Figure 5.7.

$$T_2 = (INV1 \parallel T_1) = (\{x\}, \{y, v, z\}, S_2, \emptyset)$$

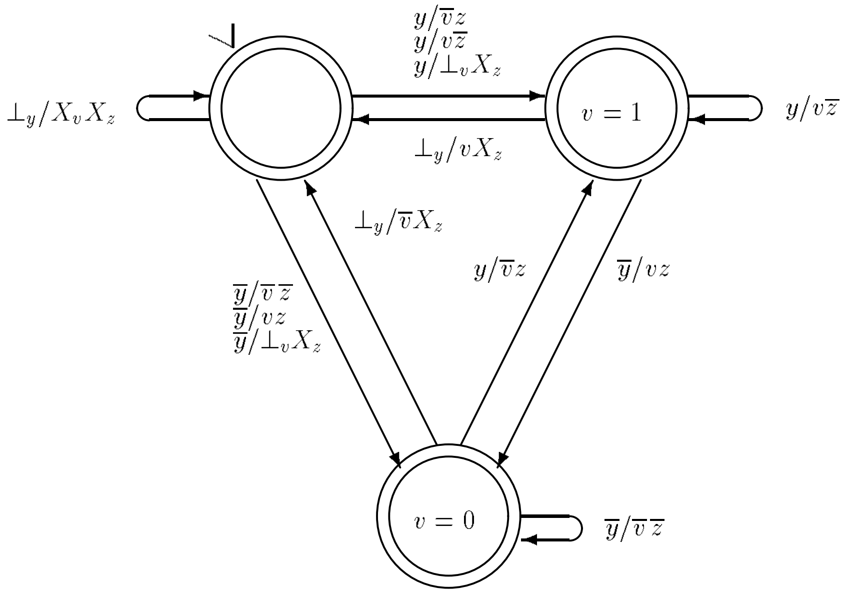

Figure 5.7: Automaton representation of the sequential behavior of T_1 of Example 5.5

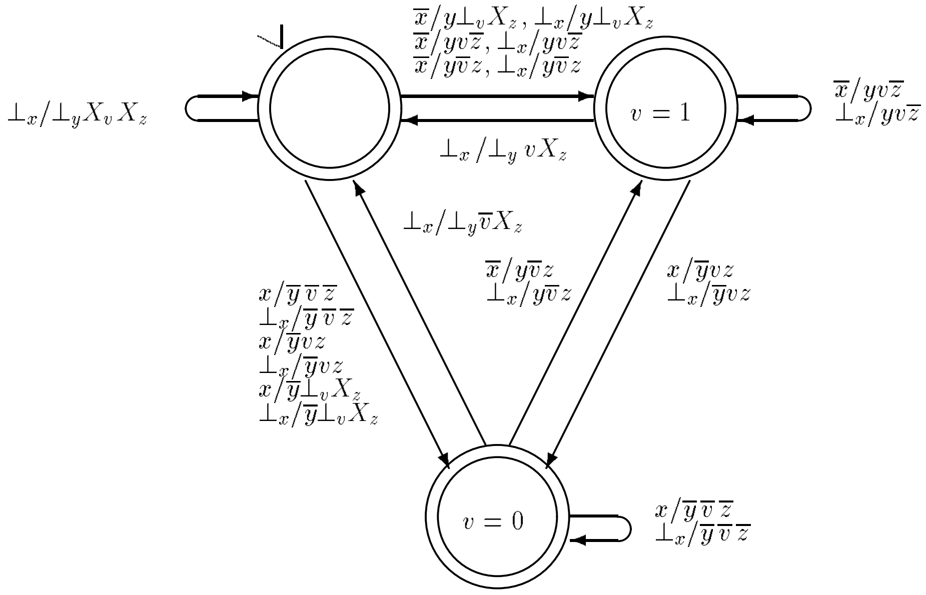
$\overline{x}/y\perp_v X_z, \perp_x/y\perp_v X_z$
$\overline{x}/yv\overline{z}, \perp_x/yv\overline{z}$
$\overline{x}/y\overline{v}z, \perp_x/y\overline{v}z$
$\perp_x/\perp_y X_v X_z$
$v=1$
$\overline{x}/yv\overline{z}$
$\perp_x/yv\overline{z}$
$\perp_x/\perp_y vX_z$
$\perp_x/\perp_y\overline{v}X_z$
$\overline{x}/y\overline{v}z$
$\perp_x/y\overline{v}z$
$x/\overline{y}vz$
$\perp_x/\overline{y}vz$
$x/\overline{y}\,\overline{v}\,\overline{z}$
$\perp_x/\overline{y}\,\overline{v}\,\overline{z}$
$x/\overline{y}vz$
$\perp_x/\overline{y}vz$
$x/\overline{y}\perp_v X_z$
$\perp_x/\overline{y}\perp_v X_z$
$v=0$
$x/\overline{y}\,\overline{v}\,\overline{z}$
$\perp_x/\overline{y}\,\overline{v}\,\overline{z}$

Figure 5.8: Automaton representation of the sequential behavior of T_2 of Example 5.5

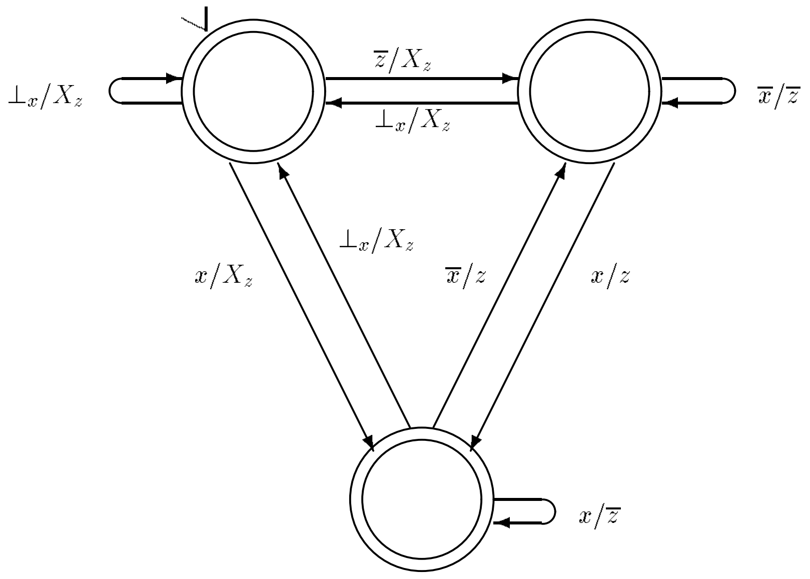

Figure 5.9: Automaton representation of the sequential behavior of the full circuit of Example 5.5

where S_2 is accepted by the automaton of Figure 5.8. And finally,

$$T = del(\{y, v\})(T_2) = (\{x\}, \{z\}, S, \emptyset)$$

where S is accepted by the automaton of Figure 5.9.

In the preceding example, we demonstrated how to use the algebraic operators to create a composite-circuit model from the models of its components. The following example highlights the role of "X-effects," or "X-confusion," in sequential trace theory. This name was coined to describe an undesirable artifact of the X-value in gate-level ternary simulation. However, in our model the identical effects *accurately* reflect the intended semantics of the "$\perp$" symbol.

Example 5.6 *In this example, we consider four distinct circuits, each of which contains a single latch and some combinational logic. For the sake of brevity, we do not walk the reader through the composition process, as we did in the previous two examples of this section. Instead, we describe the sequential trace structure representations of the primitive components of each circuit, and provide circuit algebraic formulas which correspond to these composite circuits. We then provide an automaton representation of the behavior of three of these full circuits.*

Two points are of interest in this example, beyond its role in further clarifying the algebraic operators. The first is that we obtain identical sequential trace structures for the two circuits C_6 and C_7. The second is the manifestation of "X-effects" in the presence of reconvergent fanout in T_6, T_7

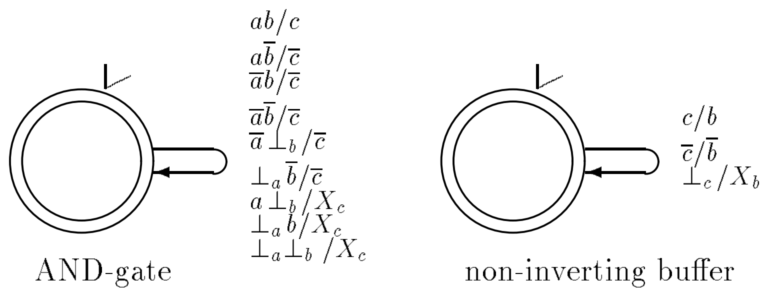
ab/c
$a\overline{b}/\overline{c}$
$\overline{a}b/\overline{c}$
$\overline{a}\overline{b}/\overline{c}$
$\overline{a}\perp_b/\overline{c}$
$\perp_a\overline{b}/\overline{c}$
$a\perp_b/X_c$
$\perp_a b/X_c$
$\perp_a\perp_b/X_c$
AND-gate
c/b
$\overline{c}/\overline{b}$
$\perp_c/X_b$
non-inverting buffer

Figure 5.10: Automata for the sequential behavior of an AND-gate and a non-inverting buffer

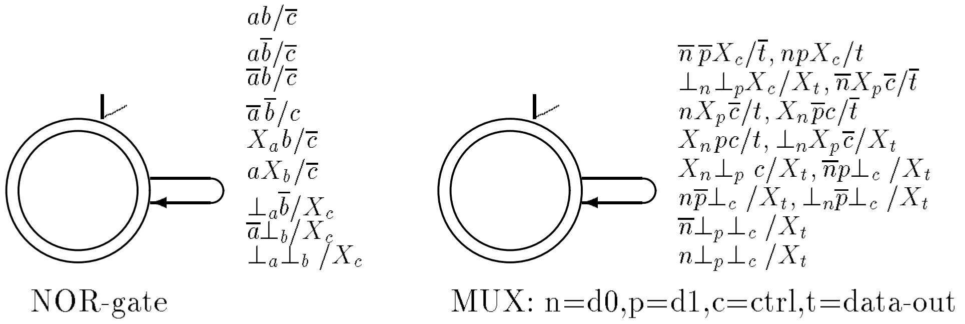
$ab/\overline{c}$
$a\overline{b}/\overline{c}$
$\overline{a}b/\overline{c}$
$\overline{a}\overline{b}/c$
$X_a b/\overline{c}$
$aX_b/\overline{c}$
$\perp_a\overline{b}/X_c$
$\overline{a}\perp_b/X_c$
$\perp_a\perp_b/X_c$
NOR-gate
$\overline{n}\,\overline{p}X_c/\overline{t}, npX_c/t$
$\perp_n\perp_p X_c/X_t, \overline{n}X_p\overline{c}/\overline{t}$
$nX_p\overline{c}/t, X_n\overline{p}c/\overline{t}$
$X_n pc/t, \perp_n X_p\overline{c}/X_t$
$X_n\perp_p c/X_t, \overline{n}p\perp_c/X_t$
$n\overline{p}\perp_c/X_t, \perp_n\overline{p}\perp_c/X_t$
$\overline{n}\perp_p\perp_c/X_t$
$n\perp_p\perp_c/X_t$
MUX: n=d0,p=d1,c=ctrl,t=data-out

Figure 5.11: Automata representations of the sequential behavior of a NOR-gate and a MUX

and T_8. Reconvergent fanout refers to a situation in a circuit in which a value computed by one component propagates through two or more distinct routes to another component. X-effects occur when a node holding value $\perp$ is not interpreted to have the same value by all gate-models of which it is an input wire.

In our model X-effects accurately reflect the intended semantics of the symbol "$\perp$". The X-value of gate-level ternary simulation is intended to model two cases, that in which a node has value 0 *and that in which the same node has value* 1*. It is not intended to model a situation in which the Boolean value of a node is ambiguous, but only one in which it is unknown. Hence "X-confusion" constitutes a loss of information in that model. However, in our model the third wire value $\perp$ represents a [possibly unstable] voltage which is not unambiguously interpretable as a* 0 *or as a* 1*; in this case, it is entirely appropriate that the same node be interpreted variously as having several values.*

Before we present the four circuits and their sequential trace structure representations, we provide a list of gate and latch models for reference. All of these models have empty F-sets; that is, their P-set and their S-set are identical. Let T_{Dff-xy} be the D-flipflop model whose P-set is defined by the automaton of Figure 5.1 with initial state $Q0$. Let T_{inv-ab} be the inverter model whose P-set is shown in Figure 5.2. Let T_{or-ebc} and $T_{xor-yvz}$ be the or-gate and xor-gate models, respectively, whose P-sets are shown in Figure 5.6. Let $T_{and-abc}$ be the and-gate model whose P-set is shown in Figure 5.10. Let $T_{nor-abc}$ and $T_{mux-npct}$ be the nor-gate and MUX models, respectively, whose

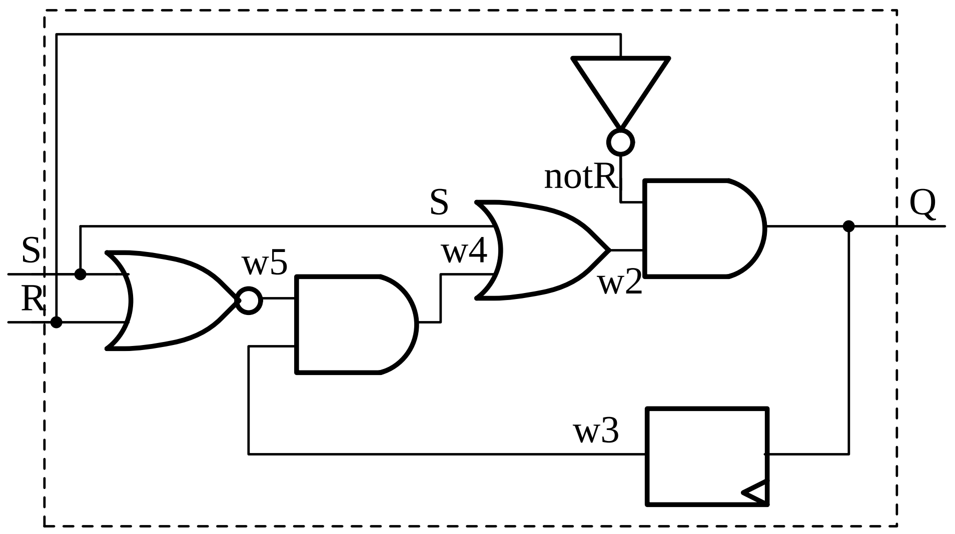

Figure 5.12: The circuit C_6

P-sets are shown in Figure 5.11. (Note that in the MUX model, n and p refer to the respective data_in lines d0 (negative) and d1 (positive), c is the control line, and t is the data_out line). Models for additional two-input gates may be created by composing the appropriate inverter model or models with the appropriate two-input gate model.

We first consider the circuit in Figure 5.12. A trace structure representation of this circuit may be described by the circuit algebra expression

$$T_6 = del(\{notR,w2,w3,w4,w5\})(INV6 \,||\, (AND6a \,||\, (OR6 \,||\, (AND6b \,||\, (NOR6 \,||\, L6)))))$$

Next, we consider the circuit in Figure 5.13. A trace structure representation of this circuit may be described by the expression

$$T_7 = del(\{notR,d0,d1,ctrl\})(INV7 \,||\, (AND7 \,||\, (NOR7 \,||\, (MUX7 \,||\, L7))))$$

Computation of T_6 and T_7 reveals that they are identical. Note that this need not have been the case if we had defined our primitive gate models differently: the two expressions are not algebraically equivalent. *Figure 5.14 contains the minimal automaton that accepts their P-set. As discussed previously, this P-set exhibits X-effects; this will be discussed in more detail in Example 5.8 of Section 5.4.5.*

Now we consider the circuit of Figure 5.15. Note that this circuit is almost identical to the circuit C_7 of Figure 5.13; they differ only in that the nor-gate in C_7 has been replaced by a nxor-gate in C_8. Hence the circuit C_8 may be modeled by the sequential trace structure described by the following

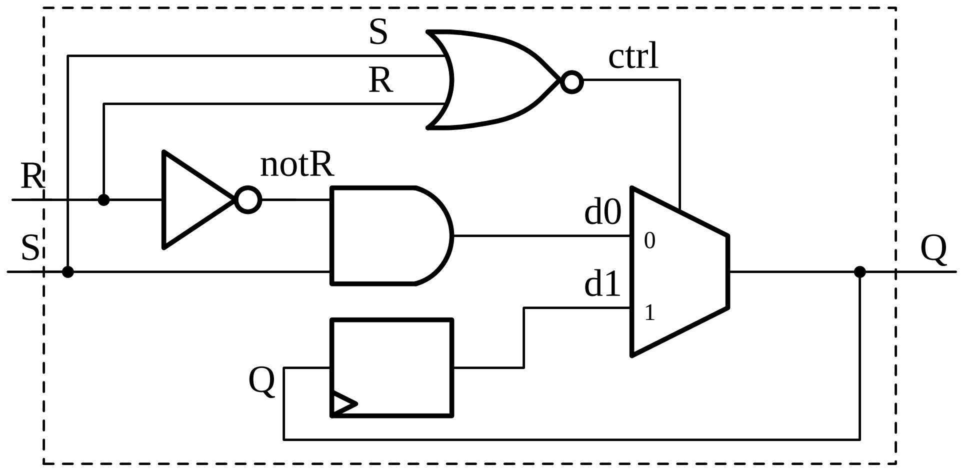

Figure 5.13: The circuit C_7

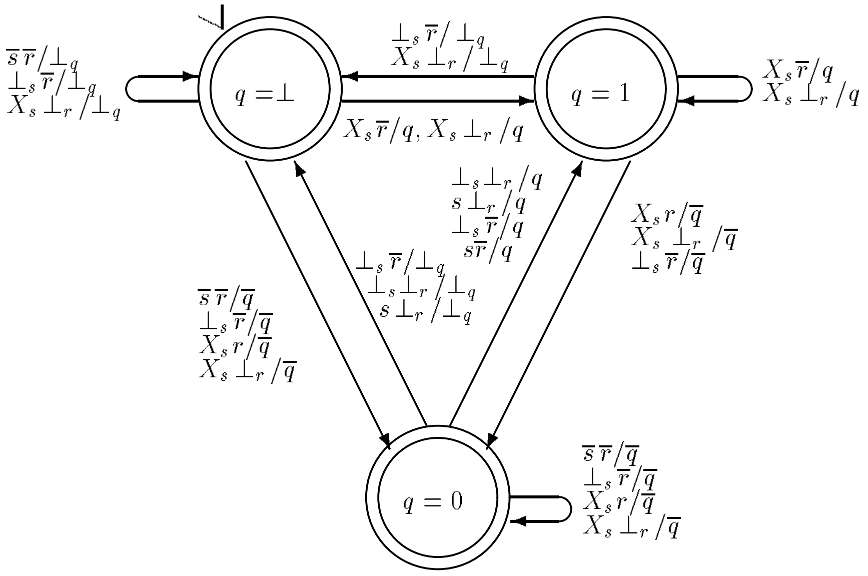
$q = \perp$
$q = 1$
$q = 0$
$\overline{s}\,\overline{r}/\perp_q$
$\perp_s \overline{r}/\perp_q$
$X_s \perp_r/\perp_q$
$\perp_s \overline{r}/\perp_q$
$X_s \perp_r/\perp_q$
$X_s \overline{r}/q, X_s \perp_r/q$
$X_s \overline{r}/q$
$X_s \perp_r/q$
$\perp_s \perp_r/q$
$s \perp_r/q$
$\perp_s \overline{r}/q$
$s\overline{r}/q$
$X_s r/\overline{q}$
$X_s \perp_r/\overline{q}$
$\perp_s \overline{r}/\overline{q}$
$\perp_s \overline{r}/\perp_q$
$\perp_s \perp_r/\perp_q$
$s \perp_r/\perp_q$
$\overline{s}\,\overline{r}/\overline{q}$
$\perp_s \overline{r}/\overline{q}$
$X_s r/\overline{q}$
$X_s \perp_r/\overline{q}$
$\overline{s}\,\overline{r}/\overline{q}$
$\perp_s \overline{r}/\overline{q}$
$X_s r/\overline{q}$
$X_s \perp_r/\overline{q}$

Figure 5.14: Automaton representation of the behavior of T_6 and T_7

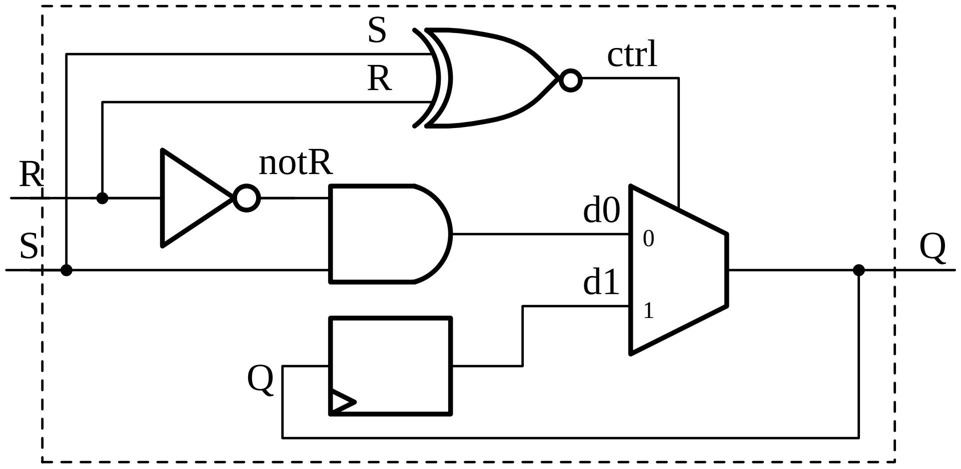

Figure 5.15: The circuit C_8

circuit algebra expression:

$$T_8 = del(\{notR, d0, d1, ctrl\})(INV7 \,\|\, (AND7 \,\|\, (NXOR8 \,\|\, (MUX7 \,\|\, L7))))$$

One might expect that T_8 *be identical to* T_7*. However, a xor-gate has no controlling values: unless both of its input wires hold known Boolean values, its output value is arbitrary. Hence* T_8 *exhibits even worse uncertainty than that exhibited by* T_7*. In fact, the P-set of* T_8 *contains that of* T_7 *as a proper subset.*

Finally, we present a circuit whose Boolean behavior is identical to that of C_6 *and* C_7*, but which does not exhibit X-effects. It avoids X-effects because it does not contain reconvergent fanout. This circuit,* C_9*, appears in Figure 5.16. The minimal automaton representation of the P-set of* T_9 *appears in Figure 5.17.*

We have provided sequential trace structure representations for four distinct circuits, two of which turned out to be identical given our gate and latch models, and three of which exhibit X-effects because of reconvergent fanout. In Section 5.4.5 we will examine all four of these trace structures again, and determine which of them correctly implement an SR-flipflop specification.

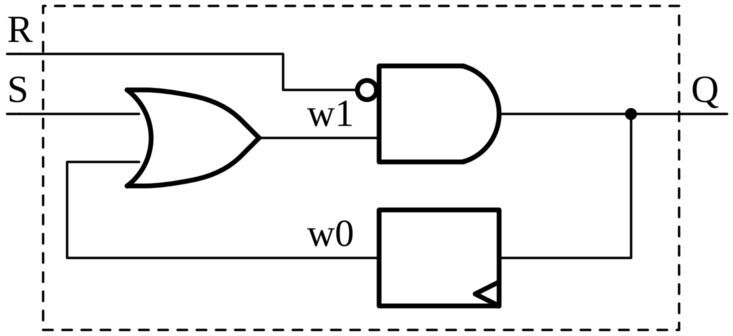

Figure 5.16: The circuit C_9

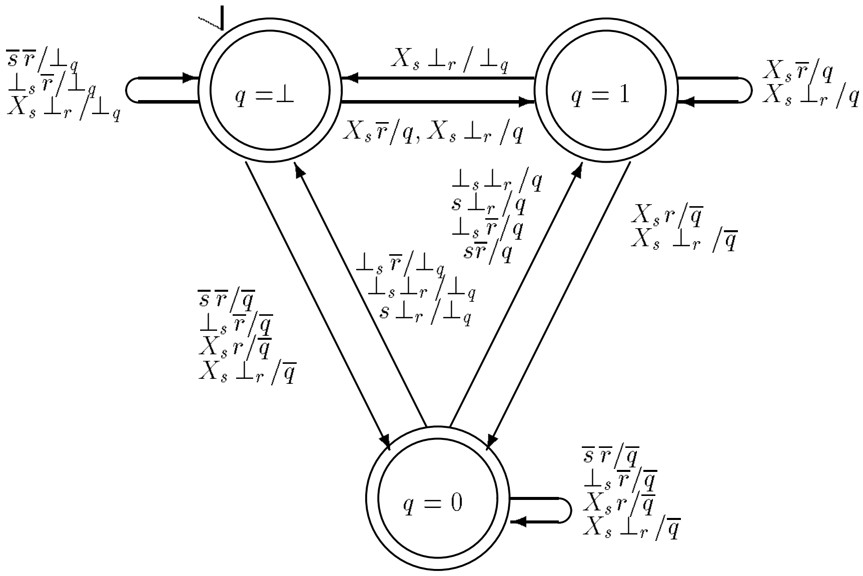
$q = \perp$
$q = 1$
$q = 0$
$\overline{s}\,\overline{r}/\perp_q$
$\perp_s\overline{r}/\perp_q$
$X_s\perp_r/\perp_q$
$X_s\perp_r/\perp_q$
$X_s\overline{r}/q, X_s\perp_r/q$
$X_s\overline{r}/q$
$X_s\perp_r/q$
$\perp_s\perp_r/q$
$s\perp_r/q$
$\perp_s\overline{r}/q$
$s\overline{r}/q$
$X_s r/\overline{q}$
$X_s\perp_r/\overline{q}$
$\perp_s\overline{r}/\perp_q$
$\perp_s\perp_r/\perp_q$
$s\perp_r/\perp_q$
$\overline{s}\,\overline{r}/\overline{q}$
$\perp_s\overline{r}/\overline{q}$
$X_s r/\overline{q}$
$X_s\perp_r/\overline{q}$
$\overline{s}\,\overline{r}/\overline{q}$
$\perp_s\overline{r}/\overline{q}$
$X_s r/\overline{q}$
$X_s\perp_r/\overline{q}$

Figure 5.17: Automaton representation of the behavior of T_9

5.4 Verification for synchronous circuit models

5.4.1 Introduction

In this section we define what it means for a synchronous circuit to correctly implement a requirements specification. The conformance and substitution results from combinational relation structures extend intact to sequential trace structures, except that canonicalization is more involved. Procedures that previously were applied to sets of input-output value combinations must now be applied to sets of sequences of such value combinations. More precisely, procedures must now be applied *per state* of a finite-state automaton representation.

We define failure-freedom, safe substitution, and the conformance relation $\preceq$ for sequential trace structures exactly as for combinational relation structures. A sequential trace structure is a correct implementation of another (considered as a specification) if and only if it *conforms* to the specification trace structure. Most of the other definitions are the same as well. There are three exceptions: first, to compute the mirror of a sequential trace structure, we complement its F and P-sets in the domain $\mathcal{B}(A)$. Second, the OUC property and operator are defined for sets of sequences of vectors. Third, the concept of an autofailure must be redefined for sequential trace structures. We still define $canon(T) = fe(afm(OUC(T)))$, and it is still the case that $canon(T)$ is the unique canonicalized element of the conformance-equivalence class of T. However, the definitions of autofailure manifestation (afm), the OUC operator, and the precise definition of a *canonicalized* sequential trace structure, differ from the combinational case.

In this section, we sketch the derivation of our decision procedure for sound hierarchical formal verification of synchronous circuits, following the basic outline of the presentation for the combinational case, which appears in Section 4.2. The substitution results for sequential trace structures are presented in Section 5.5.

5.4.2 Correct implementation: the conformance relation

The definition of when one sequential trace structure is a correct implementation of another (considered as a specification) is precisely the same as in the combinational case: one trace structure conforms to another if the first may be safely substituted for the second in any context. As in the combinational case, this conformance relation obeys the *compositionality condition,* because the algebraic operations are monotonic with respect to conformance.

More formally, we define the legal environment of a sequential trace structure T to be those sequential trace structures whose I and O-sets complement those of T, as in the combinational case. We can then prove, using precisely the same proofs as in Section 4.2.2, the equivalence of the alternate characterizations of $\preceq$ in terms of trace structure intersection and composition rather than expression contexts, and the relevant supporting lemmas. Again using the same proofs, we can prove the monotonicity of the algebraic operations with respect to $\preceq$ for sequential trace structures.

In other words, the analogs of Theorems and Lemmas 4.1 through 4.6 hold for sequential trace structures. These results allow us to utilize the alternate equivalent characterizations of $\preceq$ in later proofs. They also confirm the soundness of *hierarchical* verification for the synchronous case.

Conformance is a preorder rather than a partial order in the synchronous case as well. Again, we define $T \sim T'$ to mean that $T \preceq T'$ and $T' \preceq T$, and this equivalence relation induces a partial order on conformance-equivalence classes of sequential trace structures. The preorder $\sqsubseteq$ over sequential trace structures is stronger than the conformance relation: the analogs of Lemmas 4.7 and 4.8 hold for sequential trace structures as well.

5.4.3 Conformance equivalence classes and canonicalization

In this section, we present in full detail the theoretical underpinnings of the canonicalization process for sequential trace structures.

If we represent a trace structure T by a deterministic combined automaton that accepts both its F-set and its P-set, we can talk about T's *control* being in a state q of this automaton. This corresponds to the automaton's having just accepted a partial behavior $w \in P$ such that $q_0 \overset{w}{\Rightarrow} q$ (where q_0 is the initial state of this deterministic automaton). If it is the case that T can force *every one* of its legal environments into a failure state following the arrival of its control at the state q, then we might as well simply mark q as an F-final state. This is because T will have a non-failure-free composition with every legal environment that contains w in its own P-set whether or not we make this addition to the F-set of T.

Similarly, we note that if a trace structure has already failed (control has already entered a F-final state q' such that $q_0 \overset{w}{\Rightarrow} q'$) then it does not matter what it does afterwards. Again, this is because the trace structure T will have a non-failure-free composition with every legal environment that allows T to reach this F-final state ($w \in (P \cap P_E)$), irrespective of what the composition may or may not do afterwards. These observations lead to the definition of autofailures and the autofailure manifestation process.

We say a trace structure $T = (I, O, S, F)$ is *failure-forcing* if and only if it has no failure-free composition with any of its legal environments. Formally, for all legal environments $E = (O, I, S_E, F_E)$ of T, $E \cap T$ is not failure-free. For a trace structure $T = (I, O, S, F)$ and $w \in P$, we define $T_w = (I, O, ext(w, S), ext(w, F))$. Using this property and new definition, we present the formal definition of the *autofailures* of a trace structure $T = (I, O, S, F)$:

$$af(T) = \{w \in P \mid T_w \text{ is failure-forcing}\}$$

Clearly, $af(T)$ is the set of all $w \in P$ such that every legal environment that admits w in its own P-set can force T beyond (or at) w into the F-set of the composition. More specifically, the definition above includes precisely those $w \in P$ beyond (or at) which they can all force control into

$(F \cap P_E)$. (The "or at" means that $F \subseteq af(T)$). Because the set of all legal environments admitting w includes those with empty F-sets, this condition is equivalent to stating that all legal environments could force T beyond w into $((F \cap P_E) \cup (P \cap F_E))$, which is the full failure-set of the composition. In contrast to the definition of autofailures in asynchronous trace theory, there is an extra complication because the environment of a sequential trace structure can respond instantaneously. (Note also that in contrast to the definition of a *combinational* autofailure, which is a combinational relation structure, the autofailures we have just defined are elements of the P-set of a trace structure, as in the asynchronous case). There is, however, an effective procedure for determining $w \in af(T)$. That procedure is described in Chapter 6.

Lemma 5.1 *Let $T = (I, O, F, P)$ be a sequential trace structure. Then $F \subseteq af(T)$.*

Proof: Obvious, especially from the discussion.

We define the operation of *autofailure manifestation* to be the addition of all autofailures and their extensions to the F-set of a sequential trace structure:

$$afm((I, O, S, F)) = (I, O, S, af(T) \cdot \mathcal{B}(A))$$

Lemma 5.2 *Let T be a sequential trace structure. Then $afm(T)$ is too.*

Proof:

Let $T = (I, O, S, F)$ be a sequential trace structure. Then S and F are regular sets. Therefore $P = (S \cup F)$ is a regular set, and may be represented by a deterministic finite-state automaton. The definition of $af(T)$ makes it clear that it defines a subset of the states in that deterministic automaton. Therefore the set $af(T) \cdot \mathcal{B}(A)$ is regular as well.

Clearly, $af(T) \cdot \mathcal{B}(A)$ is input-downward-closed if $af(T)$ is. We prove that $af(T)$ is input-downward-closed because F and P are:

Let $x, x' \in (\mathcal{T}^I)^n$ such that $x' \leq x$, and $y \in (\mathcal{T}^O)^n$ such that $(x \cup y) \in P$. Then by input-downward-closure of P and F, respectively,

$$ext((x \cup y), P) \subseteq ext((x' \cup y), P) \text{ and } ext((x \cup y), F) \subseteq ext((x' \cup y), F)$$

Thus $T_{(x \cup y)} \sqsubseteq T_{(x' \cup y)}$. But then by the sequential analog of Lemma 4.7, $T_{(x \cup y)} \preceq T_{(x' \cup y)}$.

Let $(x \cup y) \in af(T)$. Because for every legal environment E of T, $E \cap T_{(x \cup y)}$ is *not* failure-free, the same must hold of $E \cap T_{(x' \cup y)}$. Therefore $(x' \cup y) \in af(T)$ as well. Therefore $af(T)$ is input-downward-closed.

Because $F \subseteq af(T)$ (by Lemma 5.1) and $af(T) \subseteq P$, it must be the case that $(S \cup F) = (S \cup af(T))$. Because $P = (S \cup F)$ is input-downward-closed, $(S \cup af(T) \cdot \mathcal{B}(A))$ is also.

Similarly, $(S \cup af(T) \cdot \mathcal{B}(A))$ is non-empty if P is.

And finally, $(S \cup af(T) \cdot \mathcal{B}(A))$ obeys the receptiveness constraint if P does: for every $w \in P$ there is an appropriate $C_w \subseteq ext_1(w, P) \subseteq ext_1(w, (S \cup af(T) \cdot \mathcal{B}(A)))$, and for every $w \in ((S \cup af(T) \cdot \mathcal{B}(A)) - P)$, $w \cdot \mathcal{B}(A)$ is contained in $(S \cup af(T) \cdot \mathcal{B}(A))$ – so that clearly in this case as well there exists an appropriate C_w. ∎

Lemma 5.3 *Let T be a sequential trace structure. Let $T' = afm(T) = (I, O, S', F')$. Then $F' = F' \cdot \mathcal{B}(A)$ and $af(T') \subseteq F'$.*

Proof:

That $F' = F' \cdot \mathcal{B}(A)$ is obvious from the definition of the *afm* operator.

In order to prove that $af(T') \subseteq F'$ it suffices to note that behaviors in S of which no element of $af(T)$ is a prefix cannot be made into autofailures (of T') by autofailure manifestation. ∎

Theorem 5.4 *Let T be a sequential trace structure. Then $T \sim afm(T)$.*

Proof:

By Lemma 5.1, $F \subseteq af(T)$. Therefore $T \sqsubseteq afm(T)$. Thus by the sequential analog of Lemma 4.7, $T \preceq afm(T)$.

We must prove that $afm(T) \preceq T$.

Let $E = (O, I, S_E, F_E)$ be a legal environment such that $T \cap E$ is failure-free. Let $w \in (P \cap P_E) = P''$. Then $T_w = (I, O, ext(w, F), ext(w, P))$ and $E_w = (O, I, ext(w, F_E), ext(w, P_E))$ are sequential trace structures such that $(T_w \cap E_w)$ is failure-free. Therefore T_w is not failure-forcing, and so $w \notin af(T)$. Thus

$$w \in P'' \Longrightarrow w \notin af(T)$$

Because $af(T) \subseteq P$, therefore $(af(T) \cap P_E) = \emptyset$. Because P_E is prefix-closed, it is also the case that

$$(af(T) \cdot \mathcal{B}(A) \cap P_E) = \emptyset$$

Therefore, $((af(T) \cdot \mathcal{B}(A) \cap P_E) \cup ((P \cup (af(T) \cdot \mathcal{B}(A)) \cap F_E)) = \emptyset$, and so $afm(T) \cap E$ is also failure-free.

Thus $afm(T) \preceq T$, and so we have proved that $afm(T) \sim T$. ∎

A sequential trace structure may have S and F sets which are not disjoint. A trace which appears in both these sets represents a behavior that is nondeterministically either a success or a failure. Composition with any other sequential trace structure admitting this trace will be non-failure-free irrespective of whether the trace is also in S. Hence in attempting to delete extraneous information from T in such a way as to maintain the soundness of our verification, we require that such a trace be considered solely as a failure.

We call the process of setting $S_{new} = (S - F)$ *failure-exclusion*, and the resulting trace structure $fe(T)$. Note that failure-exclusion need not in the general case result in a sequential trace structure, as the resulting S-set need not be prefix-closed. However, if we have just applied autofailure-manifestation to T, then $(S - F)$ is guaranteed to be prefix-closed.

Lemma 5.5 *Let $T = (I, O, S, F)$ be a sequential trace structure. If $F = F \cdot \mathcal{B}(A)$ then $(S - F)$ is prefix-closed.*

Proof: Obvious.

Lemma 5.6 *Let $T = (I, O, S, F)$ be a sequential trace structure such that $F = F \cdot \mathcal{B}(A)$. Then $fe(T)$ is too.*

Proof:

Neither the F-set nor the P-set of T is affected by the fe operator. Therefore we need only check for the required properties of the new S-set: regularity and prefix-closure.

Because S and F are regular, so is $S - F = S \cap \overline{F}$, the new S-set. And by Lemma 5.5, $(S - F)$ is also prefix-closed. ∎

Theorem 5.7 *Let $T = (I, O, S, F)$ be a sequential trace structure such that $F = F \cdot \mathcal{B}(A)$. Then $T \sim fe(T)$.*

Proof:

Neither the F-set nor the P-set of T is affected by the fe operator. Therefore, $T \sqsubseteq fe(T)$ and $fe(T) \sqsubseteq T$. Thus, by the sequential analog of Lemma 4.8, $T \sim fe(T)$. ∎

We define output-upwards-closure for sets of sequences of vectors in the obvious way, by analogy to input-downwards-closure (IDC). Given predetermined input and output sets I and O respectively, we say that a set $W \subseteq \mathcal{B}(I \cup O)$ is *output-upwards-closed* ($OUC(W)$) precisely when

$$\forall n \in \omega . \forall x \in (\mathcal{T}^I)^n . \forall y, y' \in (\mathcal{T}^O)^n . [[(x \cup y) \in W \wedge y \leq y'] \Longrightarrow (x \cup y') \in W]$$

We say a sequential trace structure $T = (I, O, F, P)$ is output-upwards-closed if and only if both its F and P sets are.

In addition to the *property* OUC we define an *abstract operator* OUC on sequential trace structures. If $T = (I, O, S, F)$ is a sequential trace structure, we define

$$OUC(T) = (I, O, OUC_{I,O}(S), OUC_{I,O}(F))$$

where $OUC_{I,O}(W)$ is the set $W \subseteq \mathcal{B}(I \cup O)$ together with the minimal set of additional behaviors (elements of $\mathcal{B}(I \cup O)$) necessary to make the resulting set output-upwards-closed, for W any of S, F, or P. Clearly $OUC_{I,O}(S) \cup OUC_{I,O}(F) = OUC_{I,O}(P)$. We prove that the result of applying this operator to a sequential trace structure is itself a sequential trace structure, and that all conformance equivalence classes are closed under application of this operator.

In order to prove regularity of the S and F sets of $OUC(T)$, it is necessary to provide details of our implementation of this operator. In the lemmas immediately following the implementation description, we show that applying the operator does indeed result in a sequential trace structure that is output-upwards-closed.

We implement the abstract OUC operator by a concrete operator OUC_{impl} : for every sequential trace structure $T = (I, O, S, F)$ we define

$$OUC_{impl}(T) = (I, O, OUC_{impl}(S), OUC_{impl}(F))$$

The concrete operator OUC_{impl} adds edges to existing finite-state automata that represent S and F and P. Let

$$M = \langle \Sigma = \mathcal{T}^{(I \cup O)}, Q, q_0, \mathit{FinalStates}, \delta \rangle$$

be a finite state-automaton such that $L(M)$ is one of S, F or P. For every state $q \in Q$ in this automaton, and every $x \in \mathcal{T}^I$ and $y, y' \in \mathcal{T}^O$ such that $y \leq y'$ and $\langle q, (x \cup y), q' \rangle \in \delta$ for some $q' \in Q$, the OUC_{impl} operator adds the edge $\langle q, (x \cup y'), q' \rangle$ into δ. Thus the transition relation of M is expanded while its other parts (including $\mathit{FinalStates}$) are left untouched. It may be the case that the S-, F- and P-sets of T are all represented by a single combined-automaton, in which case the addition of edges need only be done to this single automaton. The proof that this construction is correct follows.

Lemma 5.8 *Let T be a sequential trace structure. Then $OUC_{impl}(T)$ is the $\sqsubseteq$-minimal output-upwards-closed upper bound of T.*

Proof:

The above description of the concrete OUC_{impl} operator describes a specific enhancement to S and to F. We mathematically characterize this enhancement and prove that it is equivalent to our previous abstract description of the $OUC_{I,O}$ operator.

The enhancement of S and F implemented by the operation described above results in automata that define the sets $OUC_{impl}(S)$ and $OUC_{impl}(F)$, respectively, here defined for a set $W \subseteq \mathcal{B}(I \cup O)$:

$$OUC_{impl}(W) = \bigcup_{i \in \omega} OUC_{i,impl}(W)$$

where

- $OUC_{0,impl}(W) = W$ and
- $OUC_{(n+1),impl}(W) = \{(w \cdot (x \cup y') \cdot z) \mid w, z \in \mathcal{B}(I \cup O) \wedge x \in \mathcal{T}^I \wedge y' \in \mathcal{T}^O \wedge$ $\exists y \leq y' . (w \cdot (x \cup y) \cdot z) \in OUC_{n,impl}(W)\}$

We prove that this enhancement is equivalent to our previous abstract description of the $OUC_{I,O}$ operator. The proof is given for a set $W \subseteq \mathcal{B}(I \cup O)$, in lieu of its being repeated for each of S and F.

- $OUC_{impl}(W) \subseteq OUC_{I,O}(W)$:

  Let $a \in OUC_{impl}(W)$. Then $a \in OUC_{k,impl}(W)$ for some minimal $k \in \omega$. In addition, there exists some $n \in \omega$ such that $a \in (\mathcal{T}^{(I \cup O)})^n$, so that there exist $x \in (\mathcal{T}^I)^n$ and $y \in (\mathcal{T}^O)^n$ such that $a = (x \cup y)$.

  By definition of $OUC_{k,impl}(W)$, there exists a chain $\{y_i\}_i \subseteq (\mathcal{T}^O)^n$ of length $(k+1)$ such that $y_0 \leq y_1 \leq \ldots \leq \ldots \leq y_k$ and $\forall i \in \{0, \ldots, k\}.(x \cup y_i) \in OUC_{i,impl}(W)$ and $y_k = y$. But then $(x \cup y_0) \in W$ and $y_0 \leq y$, so that $a = (x \cup y) \in OUC_{I,O}(W)$. ▮

- $OUC_{I,O}(W) \subseteq OUC_{impl}(W)$:

  Let $b \in OUC_{I,O}(W)$. Then there exist $n \in \omega, x \in (\mathcal{T}^I)^n$, and $y, y' \in (\mathcal{T}^O)^n$ such that $(x \cup y) \in W$ and $y \leq y'$ and $b = (x \cup y')$.

  But then there exists a chain $\{y_i\}_i \subseteq (\mathcal{T}^O)^n$ of length $m \leq (n+1)$ such that $y = y_0 \leq y_1 \leq \ldots \leq y_{(m-1)} \leq y_m = y'$ and such that each y_i differs from $y_{(i+1)}$ in precisely a single position. Thus $b = (x \cup y') \in OUC_{m,impl}(W) \subseteq OUC_{impl}(W)$. ▮

QED Lemma 5.8

Lemma 5.9 *If T is a sequential trace structure, then $OUC(T)$ is too.*

Proof:

The implementation described above makes it clear that all of $OUC_{impl}(S)$, $OUC_{impl}(F)$, and $OUC_{impl}(P)$ are regular, and that $OUC_{impl}(S) \cup OUC_{impl}(F) = OUC_{impl}(P)$. By Lemma 5.8, therefore, $OUC_{I,O}(S), OUC_{I,O}(F)$, and $OUC_{I,O}(P)$ are all regular sets. In addition, the prefix-closure of S and P is clearly preserved by the operator. We also note that if P is non-empty, then $OUC_{I,O}(P)$ is surely so as well.

We prove that $OUC_{I,O}(P)$ is receptive, and that $OUC_{I,O}(F)$ and $OUC_{I,O}(P)$ are input downward closed:

- $OUC_{I,O}(P)$ is receptive:

  Let $w' \in OUC_{I,O}(P)$. Then there exists some $w \in P$ such that $w' \in OUC_{I,O}(\{w\})$. Pick some such w.

  By definition of the $OUC_{I,O}$ operator,

  $$ext_1(w, P) \subseteq ext_1(w, OUC_{I,O}(P)) \subseteq ext_1(w', OUC_{I,O}(P))$$

  Therefore we can simply set $C_{w'}$ to be C_w (from T). Therefore $OUC_{I,O}(P)$ is receptive. ▮

- $OUC_{I,O}(F)$ and $OUC_{I,O}(P)$ are input-downward-closed:

  We prove that application of the $OUC_{I,O}$ operator to any input-downward-closed set *preserves* that set's input-downward-closure property.

Let $W \subseteq \mathcal{B}(I \cup O)$ be input-downward-closed. Let $n \in \omega$. Let $x, x' \in (\mathcal{T}^I)^n$ such that $x' \leq x$. Let $y \in (\mathcal{T}^O)^n$ such that $(x \cup y) \in OUC_{I,O}(W)$.

By definition of the OUC operator, there exists $y' \leq y$ such that $(x \cup y') \in W$. But then by input-downward-closure of W, $(x' \cup y') \in W$, and therefore $(x' \cup y) \in OUC_{I,O}(W)$. ∎

Theorem 5.10 *Let T be a sequential trace structure. Then $T \sim OUC(T)$.*

Proof: This follows by the proof presented for Theorem 4.14, with the types of x and y changed appropriately. ∎

We say that a sequential trace structure $T = (I, O, S, F)$ is *canonicalized* if $af(T) \subseteq F$, $F = F \cdot \mathcal{B}(A)$, $S \cap F = \emptyset$, and F and P are output-upward-closed. We will prove later that a canonicalized sequential trace structure is unique in its conformance equivalence class. For now, we prove only the fairly obvious statement that the procedures outlined above, if applied in such an order that no operator undoes the desired effects of any other operator previously applied, do indeed result in a canonicalized trace structure. In addition, we prove that application of these operators to an already canonicalized trace structure T has no effect.

Theorem 5.11 *Let T be a sequential trace structure. Then $fe(afm(OUC(T)))$ is a canonicalized sequential trace structure that is conformance equivalent to T.*

Proof:

Let T be a sequential trace structure. Let $T_1 = OUC(T), T_2 = afm(T_1)$, and $T_3 = fe(T_2)$.

We must prove that T_3 is a sequential trace structure, that $T_3 \sim T$, and that T_3 is canonicalized.

By Lemma 5.9, T_1 is a sequential trace structure. Hence by Lemma 5.2, T_2 is too. In addition, by Lemma 5.3, $F_2 = F_2 \cdot \mathcal{B}(A)$. Thus, by Lemma 5.6, T_3 is a sequential trace structure.

We utilize the transitivity of $\sim$ to prove that $T_3 \sim T$. By Theorem 5.10, $T \sim T_1$. By Theorem 5.4, $T_1 \sim T_2$. In addition, by Lemma 5.3, $F_2 = F_2 \cdot \mathcal{B}(A)$. Therefore by Theorem 5.7, $T_2 \sim T_3$. Thus by transitivity of the relation $\sim$, we have proved that $T \sim T_3$.

In order to prove that T_3 is canonicalized, we prove that each of the relevant criteria holds:

- F_3 and P_3 are output-upwards-closed:

  By Lemma 5.8, T_1 is output-upwards-closed. In order to prove that T_2 is also output-upwards-closed, it suffices to prove that $af(T_1)$ is output-upward-closed. We prove that $af(T_1)$ is output-upward-closed because F_1 and P_1 are.

  Let $y, y' \in (\mathcal{T}^O)^n$ such that $y \leq y'$, and $x \in (\mathcal{T}^I)^n$ such that $(x \cup y) \in P_1$. Then by output-upward-closure of P_1 and F_1, respectively, $ext((x \cup y), P_1) \subseteq ext((x \cup y'), P_1)$ and $ext((x \cup y), F_1) \subseteq ext((x \cup y'), F_1)$. Thus $(T_1)_{(x \cup y)} \sqsubseteq (T_1)_{(x \cup y')}$. But then by the sequential analog of Lemma 4.7, $(T_1)_{(x \cup y)} \preceq (T_1)_{(x \cup y')}$.

  Let $(x \cup y) \in af(T_1)$. Because for every legal environment E whose input set is T_1's output set and whose output set is T_1's input set, $E \cap (T_1)_{(x \cup y)}$ is *not* failure-free, the same must hold of

$E \cap (T_1)_{(x \cup y')}$. Therefore $(x \cup y') \in af(T_1)$ as well. Therefore $af(T_1)$ is output-upward-closed, and so $F_2 = af(T_1) \cdot \mathcal{B}(A)$ and $P_2 = F_2 \cup P_1$ are each output-upward-closed.

Finally, F_3 and P_3 are output-upward-closed because by definition of failure exclusion, $F_3 = F_2$ and $P_3 = P_2$. ▮

- $F_3 = F_3 \cdot \mathcal{B}(A)$:

  By Lemma 5.3, $F_2 = F_2 \cdot \mathcal{B}(A)$. Because $F_2 = F_3$, therefore $F_3 = F_3 \cdot \mathcal{B}(A)$. ▮

- $(S_3 \cap F_3) = \emptyset$: This follows directly from the definition of failure exclusion. ▮

- $af(T_3) \subseteq F_3$:

  By Lemma 5.3, $af(T_2) \subseteq F_2$. Failure exclusion affects neither the F-set nor the P-set of the trace structure being operated on, in this case T_2. Therefore $af(T_3) = af(T_2) \subseteq F_2 = F_3$. ▮

QED Theorem 5.11

In addition to being canonicalized and conformance equivalent to T, this derived trace structure is identical to T *if it was already canonicalized.*

Lemma 5.12 *If T is a canonicalized sequential trace structure, then $fe(afm(OUC(T))) = T$.*

Proof: Let T be a canonicalized sequential trace structure. Then F and P are output-upward-closed (that is, $F = OUC_{I,O}(F)$ and $P = OUC_{I,O}(P)$). Therefore $T = OUC(T)$.

Similarly, because T is canonicalized, $F = F \cdot \mathcal{B}(A)$ and $af(T) \subseteq F$. Therefore by Lemma 5.1 $F = af(T) \cdot \mathcal{B}(A)$, and so $T = afm(T)$.

And finally, because T is canonicalized, its S and F-sets are disjoint. Therefore $fe(T) = T$.

Because of these three facts, $fe(afm(OUC(T))) = fe(afm(T)) = fe(T) = T$. ▮

5.4.4 Deciding conformance

As in the combinational theory, we define the mirror of a canonicalized trace structure to be the result of swapping its inputs and outputs, taking the complements of its F and P-sets, and then swapping them. Formally, for $T = (I, O, F, P)$ a canonicalized sequential trace structure, $mir(T) = (O, I, \mathcal{B}(A) - P, \mathcal{B}(A) - F)$. We define $canon(T) = fe(afm(OUC(T)))$ and $T^{MaxEnv} = mir(canon(T))$. Those canonicalized trace structures for which T^{MaxEnv} is not a sequential trace structure are precisely the failure-forcing trace structures. For all other trace structures T, T^{MaxEnv} is the $\sqsubseteq$-maximal safe environment for T. Every conformance equivalence class contains a unique canonicalized element, which we call its *canonical* element. The class of canonical trace structures, with suitably redefined algebraic operators, forms a circuit algebra. Finally, the theorems that justify deciding conformance via canonicalization and mirroring hold for sequential trace structures as well.

In order to prove these results, we extend the preorder $\sqsubseteq$ to structures $(I, O, \emptyset, \emptyset)$ as we did in the combinational case. Sequential trace structure analogs of Theorems and Lemmas 4.18 through 4.32

are proved using the identical proofs to those presented for the combinational case in Section 4.2. The proof of the sequential analog of Lemma 4.17 closely follows that of Lemma 4.17, but it is not syntactically identical. Therefore we present it below for the skeptical (or thorough) reader.

As for combinational relation structures, the cumulative effect of all these results is to provide us with the outline of a decision procedure for determining, for given trace structures T and T' of the same (I, O)-type, whether or not $T \preceq T'$. Actual application of this procedure requires that we be able to effectively determine the autofailures $af(T')$ of the trace structure T'. An effective algorithm for determining $af(T')$ is presented in Chapter 6. The theorems also clarify that conformance respects our assumption that the appearance of a $\perp$ value on an output wire in a synchronous circuit specification indicates that any of the values 0, 1 or $\perp$ may appear in its stead (under the same circumstances) in the allowed implementations.

The decision procedure for checking conformance that is outlined by the above results is the following. In order to check whether or not $T \preceq T'$, we first determine whether or not T' is failure-forcing. (We do this by determining its autofailures: if the empty sequence ε is in $af(T')$, then T' is failure-forcing). If T' is failure-forcing, then $T \preceq T'$ by the sequential analog of Theorem 4.24. Otherwise, we compute $(T')^{MaxEnv}$. We then check for emptiness of the F-set of $T \cap (T')^{MaxEnv}$: if it is empty then $T \preceq T'$, and otherwise not.

We now present the proof of the sequential analog of Lemma 4.17. In the following subsection, we present examples that illustrate the conformance relation for sequential trace structures, and its use in hierarchical formal verification of synchronous circuits.

Lemma 5.13 *(The sequential analog of Lemma 4.17):*

Let $T = (I, O, S, F)$ be a canonicalized sequential trace structure such that $S \neq \emptyset$. Then $mir(T)$ is also a canonicalized sequential trace structure.

Proof: Let T be a canonicalized sequential trace structure. Let $T' = mir(T)$. We must prove that T' is a sequential trace structure and that it is canonicalized.

Because S, F and P are all regular, so are $S' = S, F' = \mathcal{B}(I \cup O) - P$, and $P' = \mathcal{B}(I \cup O) - F$. Also, S' is prefix-closed because S is. Because $S \neq \emptyset$ and $S \cap F = \emptyset$, it must be the case that $F \neq \mathcal{B}(A)$, and so $P' = \overline{F} \neq \emptyset$. P' is prefix-closed because $P' = \overline{F}$ and $F = F \cdot \mathcal{B}(A)$.

In order to prove that $P' = \overline{F}$ obeys the receptiveness constraint, we address two cases. In the first, $w \in \overline{P} \subseteq \overline{F} = P'$. Because P is prefix-closed, $\overline{P} = \overline{P} \cdot \mathcal{B}(A)$. Therefore in this case there exists $C_w \subseteq ext_1(w, \overline{F}) = \mathcal{B}(A)$ which is total in $\mathcal{T}^O$ and has the upward-chains property (in (O, I)).

As a preliminary to addressing the second case, that of $w \in S \subseteq \overline{F} = P'$, we note that for every sequential trace structure $T_0 = (I, O, F_0, P_0)$ and legal environment $E = (O, I, F_E, P_E)$,

$$\begin{aligned}(T_0 \cap E) \text{ is failure-free} \quad &\Longleftrightarrow \quad (P_0 \cap F_E) = \emptyset \wedge (F_0 \cap P_E) = \emptyset \\ &\Longleftrightarrow \quad F_E \subseteq \overline{P_0} \wedge P_E \subseteq \overline{F_0}\end{aligned}$$

Let $w \in S \subseteq \overline{F} = P'$. Then the above holds for $T_0 = T_w = (I, O, ext(w, F), ext(w, P))$. Because

$\overline{ext(w,F)} = ext(w,\overline{F})$ and $\overline{ext(w,P)} = ext(w,\overline{P})$, the equations above state that

$$T_w \cap E \text{ is failure-free if and only if } F_E \subseteq ext(w,\overline{P}) \textit{ and } P_E \subseteq ext(w,\overline{F})$$

We must prove that there exists $C \subseteq ext_1(w,\overline{F})$ that is total (in $\mathcal{T}^O$) and that has the upward-chains property (in (O,I)). The proof is by contradiction. Say that there exists no such $C \subseteq ext_1(w,\overline{F})$. Then clearly no $X \subseteq ext_1(w,\overline{F})$ can contain such a C either. Thus, by this containment-upward-closure property of the receptiveness constraint, there exists no legal environment E such that $T_w \cap E$ is failure-free. This is because $T_w \cap E$ is failure-free only if $P_E \subseteq ext(w,\overline{F})$. But if $P_E \subseteq ext(w,\overline{F})$ then it must be the case that $ext_1(\varepsilon, P_E) \subseteq ext_1(w,\overline{F})$. Thus there can exist no $\mathcal{T}^O$-total $C \subseteq ext_1(\varepsilon, P_E)$ having the upward-chains property in (O,I). Therefore P_E does not obey the receptiveness constraint, and so E is not a legal environment after all. By this argument, there exists no legal environment E such that $T_w \cap E$ is failure-free, and hence T_w is failure-forcing. But then $w \in af(T)$, and so, since T is canonicalized by assumption, $w \in F$. But by assumption, $w \in \overline{F}$. Therefore there must exist appropriate $C \subseteq ext_1(w, \overline{F} = P')$.

Finally, the proof that $P' = \overline{F}$ and $F' = \overline{P}$ are input-downward-closed (in (O,I)) follows from the fact that F and P are output-upwards-closed (in (I,O)) :

Let $n \in \omega$. Let $y, y' \in (\mathcal{T}^O)^n$ and $x \in (\mathcal{T}^I)^n$ such that $(y \cup x) \in \overline{F}$ and $y' \leq y$. We must prove that $(y' \cup x) \in \overline{F}$. By assumption, T is canonicalized. Therefore F is output-upwards-closed. Hence, because $(y \cup x) \notin F$, it must be the case that $(y' \cup x) \notin F$ as well. But then $(y' \cup x) \in \overline{F}$. The same argument may be repeated with P in place of F.

This concludes our proof that $T' = mir(T)$ is a sequential trace structure.

We proceed to prove that $T' = mir(T)$ is canonicalized:

- F' and P' are output-upwards-closed:

  $F' = \overline{P}$ is output-upwards-closed because P is input-downward-closed, and $P' = \overline{F}$ is output-upwards-closed because F is input-downward-closed.

  Let $n \in \omega$. Let $x, x' \in (\mathcal{T}^I)^n$ and $y \in (\mathcal{T}^O)^n$ such that $(y \cup x) \in \overline{P}$ and $x \leq x'$. We must prove that $(y \cup x') \in \overline{P}$. By assumption, P is input-downward-closed. Therefore, because $(y \cup x) \notin P$, it must be the case that $(y \cup x') \notin P$ as well. But then $(y \cup x') \in \overline{P} = F'$. The same argument may be repeated with F in place of P. ∎

- $F' = F' \cdot \mathcal{B}(A)$:

  P is prefix-closed. Therefore $\overline{P} = \overline{P} \cdot \mathcal{B}(A)$. In other words, $F' = F' \cdot \mathcal{B}(A)$. ∎

- $S' \cap F' = \emptyset$:

  $S' = S$ and $F' = \overline{P} = \overline{(S \cup F)} \subseteq \overline{S} = \overline{S'}$. Thus $w \in S' \Longrightarrow w \notin F'$ and $w \in F' \Longrightarrow w \notin S'$.

- $af(T') \subseteq F'$:

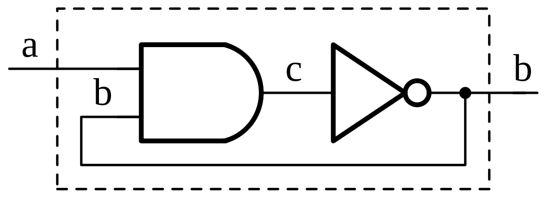

Figure 5.18: A circuit given as a specification

Let $w \in af(T')$. Either $w \in \overline{P}$ or $w \in P$. If $w \in \overline{P} = F'$ then we're done. We focus on the case in which $w \in P$.

If $w \in af(T')$ then $(T')_w$ is failure-forcing. Because $w \in P$, we know that T_w is a legal environment for $(T')_w$. But we also know that $T_w \cap (T')_w$ is failure-free, because its F-set is $((ext(w,P) \cap ext(w,\overline{P})) \cup (ext(w,F) \cap ext(w,\overline{F})))$, which is empty because $ext(w,\overline{W}) = \overline{ext(w,W)}$ for $W \in \{F,P\}$. Therefore $(T')_w$ cannot be failure-forcing. Hence $w \notin P$, and this case cannot occur. ∎

QED Lemma 5.13

5.4.5 Examples

In this section we present some examples which illustrate the conformance relation and its use in hierarchical verification of synchronous circuits.

Example 5.7 illustrates the meaning of a $\perp$ value on an output wire in a specification. We present a specification that is in fact a composite circuit model. It is not output-upward closed, although its component models are. When considered as a specification, it allows arbitrary values in place of a $\perp$ value on an output.

Example 5.8 illustrates some of the consequences of our accurate modeling of X-effects. We discuss the possibility of expanding the F-set of a specification in order to make a statement about the environments in which its valid implementations are expected to function correctly.

In Example 5.9 the specification is a sequential trace structure representation of a more complex circuit. In this case, formal verification allows us to check that an optimized version of the original circuit is indeed a legitimate substitute for the original. Finally, Example 5.10 illustrates hierarchical verification of synchronous circuits.

Example 5.7 *Consider the circuit of Figure 5.18. Although it is a combinational circuit, we are interested in considering it as a specification, and in determining whether or not this specification has any correct sequential implementations.*

We may represent this circuit by the sequential trace structure

$$T = del(\{c\})(T_{and-abc} \parallel T_{inv-cb})$$

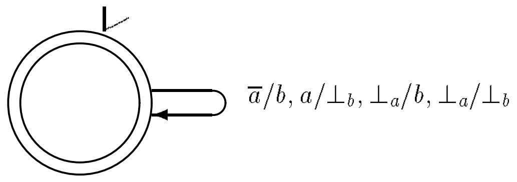

Figure 5.19: Automaton describing the S and P-sets of the circuit of Figure 5.18

where $T_{and-abc}$ and T_{inv-ab} are as defined in Example 5.6, and

$$T_{inv-cb} = ren([c \mapsto a, b \mapsto b])(T_{inv-ab})$$

Given our previous definitions of sequential trace structure models for an and-gate and an inverter, cited above, we derive the S and P-sets of T to be the language described by the automaton of Figure 5.19.

Consider this trace structure as a specification. Then certainly it conforms to itself. Thus the circuit depicted is a valid implementation of the trace structure T considered as a specification.

It is also the case that an inverter legitimately implements this specification:

$$T_{inv-ab} \preceq T$$

Similarly, the constant trace structure which outputs only the value 1 is a valid implementation:

$$T_1 = (\{a\}, \{b\}, (X_a/b)^*, \emptyset) \preceq T$$

These facts are not obvious from the automaton representation of the S-set S of T, but follow from output-upward closure. That is, $T = (\{a\}, \{b\}, S, \emptyset)$ is conformance equivalent to

$$T' = (\{a\}, \{b\}, OUC_{\{a\},\{b\}}(S), \emptyset),$$

and so any sequential trace structure that conforms to T' conforms also to T.

In order to clarify that output-upward closure extends beyond the combinational domain, we point out that the sequential trace structure

$$T_2 = (\{a\}, \{b\}, S_2, \emptyset)$$

— whose S-set S_2 is described by the automaton of Figure 5.20 — also conforms to T.

We now return to our SR-flipflop specification from the previous section. We illustrate that we can indeed implement this specification using only combinational parts and D-flipflops. We also

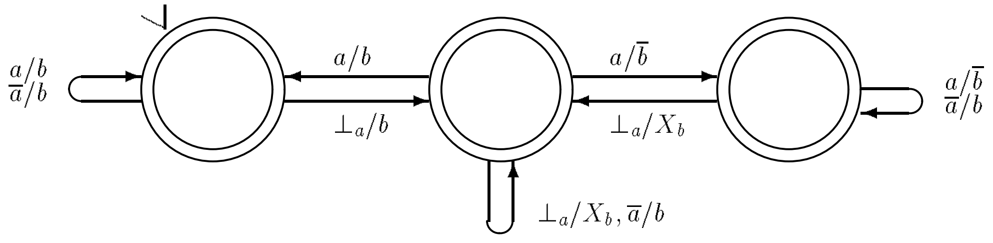

Figure 5.20: Automaton describing the S-set S_2 of T_2 of Example 5.7

show that the X-effects exhibited by some of our candidate implementation circuit models reflect real oscillatory behavior, which we may choose to allow or disallow by carefully selecting the appropriate specification to express our requirements.

Example 5.8 *As our second verification example, we note that according to sequential trace theory, the circuit C_9 of Example 5.6 correctly implements the SR-flipflop specification of Example 5.4. That is, $T_9 \preceq T_{SRff}$.*

Our models T_6, T_7 and T_8 of circuits C_6, C_7 and C_8, respectively – also presented in Example 5.6 – do not conform to this specification. This is because of X-effects: all three of these models allow (contain in their P-set) the initial behavior $X_s\overline{r}/q$ followed by $\perp_s\overline{r}/\perp_q$ or by $\perp_s\overline{r}/\overline{q}$; none of these sequences is in the P-set of this specification.

However, if we were to construct a specification of an SR-flipflop that explicitly disallowed any oscillating behavior on its inputs – that is, a specification that assumes the environment will never provide it with ill-formed input values – we would find that these three models do constitute acceptable implementations. In those environments that provide only stable Boolean input values, the models behave exactly like the specification. The SR-flipflop specification whose S and F-sets are provided by the combined automaton of Figure 5.21 meets these criteria – and indeed, all three of T_6, T_7 and T_8 conform to it. Of course, T_9 conforms to it as well.

Thus if we wish to model the possibility of oscillation, the original specification T_{SRff} provides more conservative verification results. However, if we wish to ignore oscillation, we may do so within the parameters of sequential trace theory. In general, the more extensive the F-set of a specification, the less restrictive it is.

Our method of formal verification allows us to determine whether or not a circuit which appears to be an optimized version of another circuit is indeed a legitimate substitute for it. This is illustrated by the following example.

Example 5.9 *Consider the circuit C_{11} of Figure 5.22, Formal verification in our model correctly concludes that the optimized circuit C'_{11} of Figure 5.23 is a valid implementation of the original circuit considered as a specification.*

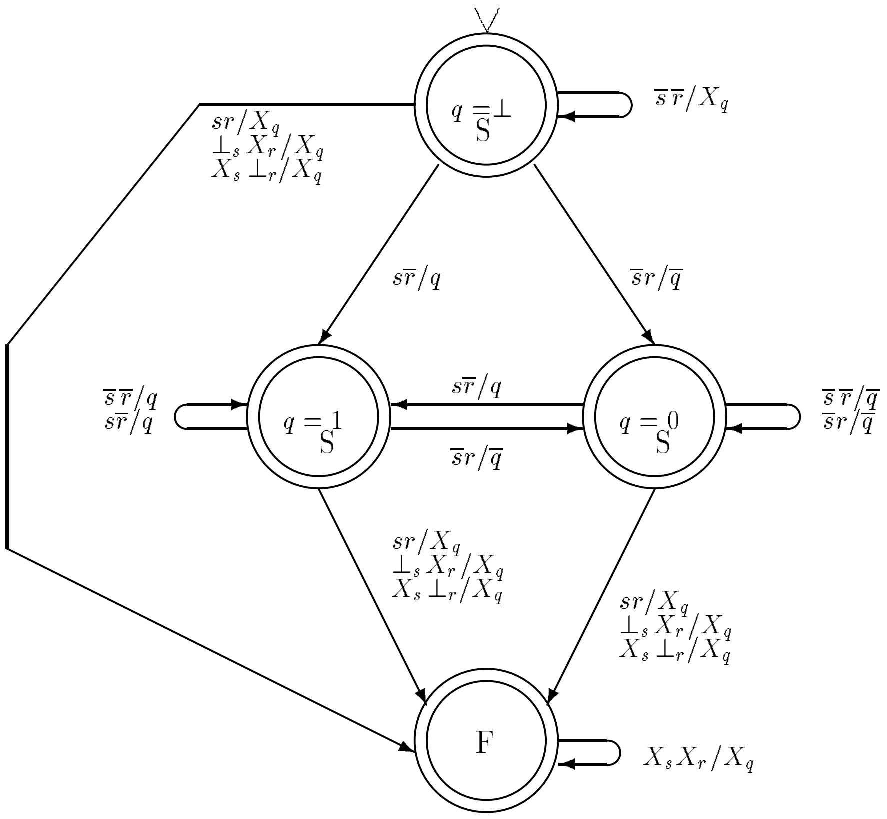

Figure 5.21: F-intensive behavioral specification of an SR-flipflop

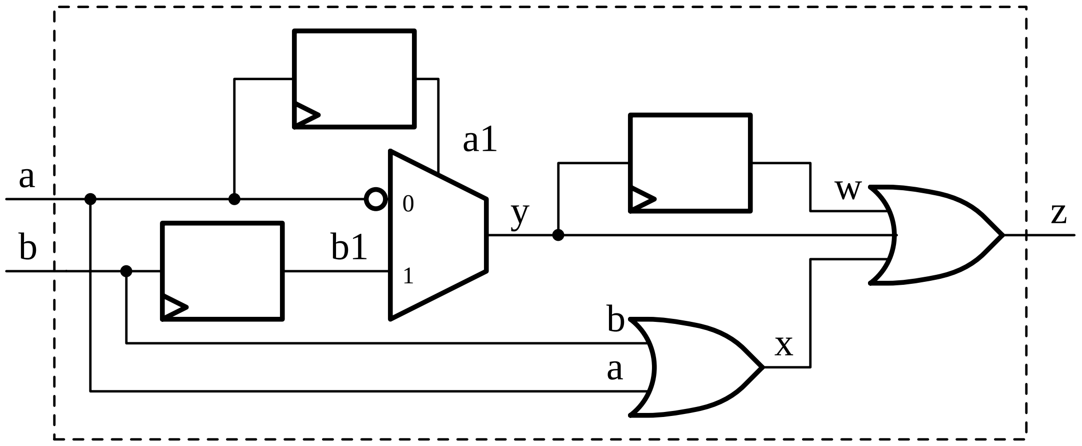

Figure 5.22: The original circuit C_{11} (Figure 4.6 of [55])

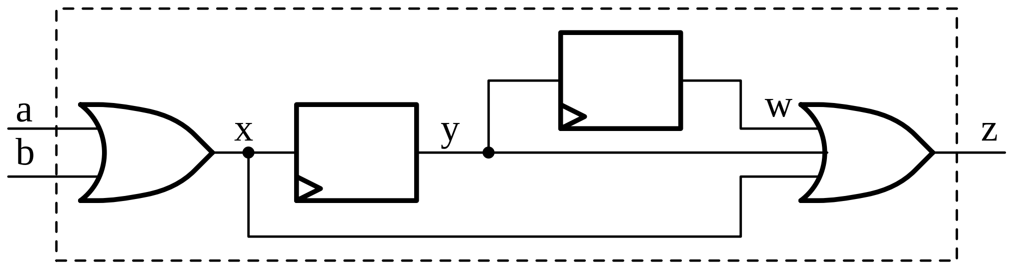

Figure 5.23: C'_{11}, an optimized version of C_{11}

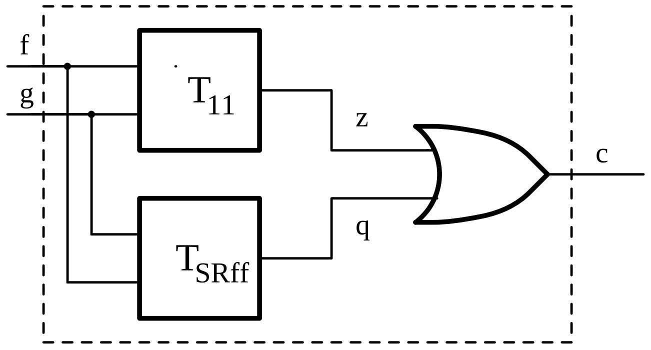

Figure 5.24: Composite specification for Example 5.10

In the following example we illustrate how *hierarchical* verification allows us to replace circuit parts for their specifications without repeating the verification process for the full new circuit.

Example 5.10 *Consider the specification*

$$T = del(\{q, z\})(ren([a \mapsto f, b \mapsto g])(T_{11}) \parallel ren([s \mapsto f, r \mapsto g])(T_{SRff}) \parallel ren([e \mapsto q, b \mapsto z])(T_{or-ebc}))$$

depicted in Figure 5.24, where T_{11} is our sequential trace structure representation of circuit C_{11} in Figure 5.22, T_{SRff} is the SR-flipflop specification of Figure 5.4 (page 127), and T_{or-ebc} is our sequential trace structure representation of an or-gate, with input wires labeled e and b and output wire labeled c, whose P-set appears in Figure 5.6 (page 131). According to Example 5.9, our sequential trace structure representation of the circuit of Figure 5.23, which we will call T'_{11}, conforms to T_{11}. Thus by mononicity of the renaming operator with respect to conformance, $ren([a \mapsto f, b \mapsto g])(T'_{11}) \preceq ren([a \mapsto f, b \mapsto g])(T_{11})$. According to Example 5.8, $T_9 \preceq T_{SRff}$. Let $T'_9 = ren([s \mapsto f, r \mapsto g])(T_9)$ and let $T'_{SRff} = ren([s \mapsto f, r \mapsto g])(T_{SRff})$. By monotonicity of the renaming operator with respect to conformance, $T'_9 \preceq T'_{SRff}$. Therefore by monotonicity of renaming, hiding and composition with respect to conformance, we may conclude that the circuit illustrated in Figure 5.25, whose trace

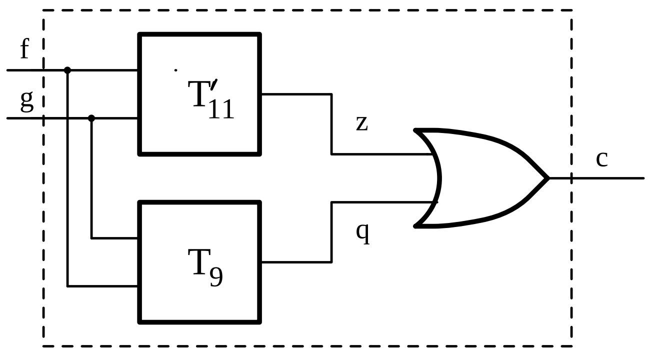

Figure 5.25: Composite implementation for Example 5.10

structure representation is

$$del(\{q,z\})[ren([a \mapsto f, b \mapsto g])(T'_{11}) \parallel ren([s \mapsto f, r \mapsto g])(T_9) \parallel ren([e \mapsto q, b \mapsto z])(T_{or-ebc})]$$

conforms to the specification T.

5.5 Substitution for synchronous circuit models

In this section we address the problem of correctness with respect to an environment for synchronous circuits and environments. We are able to derive a closed-form expression that specifies all and only the allowed substitute circuitry for a subcircuit in a given circuit, as in the combinational case. We first present our formal results and then illustrate their use via a series of examples.

5.5.1 Conformance with respect to an environment

The main substitution results for sequential trace structures are the following theorems. Theorem 5.14 can be proved using almost the identical collection of lemmas and proofs that were used to prove Theorem 4.33 (as presented in Sections 4.3.2 and 4.3.3). Theorem 5.15 follows directly from Theorem 5.14, by setting $T_0[\alpha] = E[\alpha]$ and $T = E[T'']$.

Theorem 5.14 *(The sequential analog of Theorem 4.33):*

If $T = (I, O, S, F)$ and $T' = (I_\alpha, O_\alpha, S', F')$ are sequential trace structures, and $T_0[\alpha]$ is an expression context of type (I, O), then

$$T_0[T'] \preceq T \text{ iff } T' \preceq f(T_0[\alpha], T)$$

Theorem 5.15 *(The sequential analog of Theorem 4.34):*

If $E[\alpha]$ is an expression context and $T'' = (I_\alpha, O_\alpha, S'', F'')$ is a sequential trace structure, then for all sequential trace structures $T' = (I_\alpha, O_\alpha, S', F')$,

$$T' \preceq_E T'' \Longleftrightarrow T' \preceq f(E[\alpha], E[T''])$$

In order to transform the proof of Theorem 4.33 into a proof of Theorem 5.14, we redefine the auxiliary function f. The third part of its definition is now:

$$f(T_3 \parallel T_2[\alpha], T) = \begin{cases} (I_\alpha, O_\alpha, \mathcal{B}(A_\alpha), \mathcal{B}(A_\alpha)) & \text{if } T \text{ is failure forcing (FFor)} \\ (I_\alpha, O_\alpha, \emptyset, \emptyset) & \text{if } del(A - A_2)(T^{MaxEnv} \parallel T_3) \text{ is FFor} \\ f(T_2[\alpha], (del(A - A_2)(T^{MaxEnv} \parallel T_3))^{MaxEnv}) & \text{otherwise} \end{cases}$$

When $(A \cap D) = \emptyset$, we define $del_O^{-1}((I, O, S, F)) = (I, O \cup D, del^{-1}(D)(S), del^{-1}(D)(F))$, as in the combinational case, and we extend the $\preceq$ relation to structures of the form $(I, O, \emptyset, \emptyset)$ for purposes of explanation.

Using these new definitions, we may use the proof of Theorem 4.33 from Section 4.3.2 to prove Theorem 5.14. The supporting lemmas on which it depends, which appear in Section 4.3.3, have sequential analogs as well. With one exception, the proofs are identical.

We now present the proof of the sequential analog of the lone exception, Lemma 4.38. In the following subsection, we present examples that illustrate the use of these theorems for correct substitution and rectification.

Lemma 5.16 *(The sequential analog of Lemma 4.38):*
Let $T = (I, O, F, P)$ be a sequential trace structure, and let $(D \cap (I \cup O)) = \emptyset$. Then

$$del_O^{-1}(D)(canon(T)) = canon(del_O^{-1}(D)(T))$$

Proof:
Let $T = (I, O, F, P)$. Let $w \in P$ and $w' \in del^{-1}(D)(w)$.

By the sequential analog of Lemma 4.36, which says that T is failure forcing if and only if $del_O^{-1}(D)(T)$ is failure forcing, and by Lemma 5.17 (below), T_w is failure-forcing if and only if $(del_O^{-1}(D)(T))_{w'}$ is failure forcing. This implies that $w \in af(T)$ if and only if $w' \in af(del_O^{-1}(D)(T))$. Therefore

$$w \in af(T) \Longrightarrow del_O^{-1}(D)(w) \subseteq af(del_O^{-1}(D)(T))$$

and

$$(del_O^{-1}(D)(w) \cap af(del_O^{-1}(D)(T))) \neq \emptyset \Longrightarrow w \in af(T)$$

Therefore $del_O^{-1}(D)(af(T)) = af(del_O^{-1}(D)(T))$. We will use this result in the remainder of the proof.

$$\begin{aligned}
del_O^{-1}(D)(canon(T)) &= del_O^{-1}(D)(canon((I,O,F,P))) \\
&= (I, O \cup D, del^{-1}(D)[af(OUC(T)) \cdot \mathcal{B}(A)], \\
&\qquad del^{-1}(D)[OUC_{I,O}(P) \cup af(OUC(T)) \cdot \mathcal{B}(A)]) \\
&= (I, O \cup D, del^{-1}(D)(af(OUC(T)) \cdot \mathcal{B}(A \cup D), \\
&\qquad del^{-1}(D)(OUC_{I,O}(P)) \cup del^{-1}(D)(af(OUC(T)) \cdot \mathcal{B}(A \cup D)) \\
&= (I, O \cup D, af(del_O^{-1}(D)(OUC(T))) \cdot \mathcal{B}(A \cup D), \\
&\qquad del^{-1}(D)(OUC_{I,O}(P)) \cup af(del_O^{-1}(D)(OUC(T))) \cdot \mathcal{B}(A \cup D)) \\
&\qquad \text{– by the proof above} \\
&= (I, \\
&\quad O \cup D, \\
&\quad af(OUC(del_O^{-1}(D)(T))) \cdot \mathcal{B}(A \cup D), \\
&\quad OUC_{I,(O \cup D)}(del^{-1}(D)(P)) \cup af(OUC(del_O^{-1}(D)(T))) \cdot \mathcal{B}(A \cup D)) \\
&\qquad \text{– by the sequential analog of Lemma 4.37} \\
&= canon(del_O^{-1}(D)(T))
\end{aligned}$$

Note that the S sets of both $del_O^{-1}(D)(canon(T))$ and $canon(del_O^{-1}(D)(T))$ are equal to their common P set minus their common F set, and are therefore equal to each other. Thus there was no need to mention failure-exclusion explicitly in the calculation above. ∎

The above proof depended on the following lemma, which does not have a combinational analog:

Lemma 5.17 *Let $T = (I, O, S, F)$ be a sequential trace structure. Let $(D \cap (I \cup O)) = \emptyset$. Let $w \in P$ and $w' \in del^{-1}(D)(w)$. Then $(del_O^{-1}(D)(T))_{w'} = del_O^{-1}(D)(T_w)$.*

Proof:

Let $T = (I, O, S, F)$. Let $w \in P$ and $w' \in del^{-1}(D)(w)$. Then

$$(del_O^{-1}(D)(T))_{w'} = (I, O \cup D, ext(w', del^{-1}(D)(S)), ext(w', del^{-1}(D)(F)))$$

and

$$del_O^{-1}(D)(T_w) = (I, O \cup D, del^{-1}(D)(ext(w, S)), del^{-1}(D)(ext(w, F)))$$

In order to prove that these two sequential trace structures are identical, we prove that for all sets $Y \subseteq \mathcal{B}(I \cup O)$ and sequences $w \in \mathcal{B}(I \cup O)$ and $w' \in del^{-1}(D)(w)$, it is the case that

$$ext(w', del^{-1}(D)(Y)) = del^{-1}(D)(ext(w, Y))$$

Fundamentally, this fact follows from the fact that del^{-1} and the $\cup$ operator on sequences commute. Application of this result to $Y = S$ and $Y = F$ proves the lemma.

- $(\Longrightarrow)$:

Let $z' \in ext(w', del^{-1}(D)(Y)$. Then there exist $n, m \in \omega$ such that $w' \in (\mathcal{T}^{(I \cup O \cup D)})^n$ and $z' \in (\mathcal{T}^{(I \cup O \cup D)})^m$. Furthermore, there exist $w_1 \in (\mathcal{T}^D)^n$, $z_0 \in (\mathcal{T}^{(I \cup O)})^m$ and $z_1 \in (\mathcal{T}^D)^m$ such that $w' = w \cup w_1$ and $z' = z_0 \cup z_1$.

By definition of the *ext* operator, $(w' \cdot z') \in del^{-1}(D)(Y)$. Thus $(w \cup z_0) \in Y$. Therefore $z_0 \in ext(w, Y)$. And finally, therefore $z' = (z_0 \cup z_1) \in del^{-1}(D)(ext(w, Y))$.

Therefore $ext(w', del^{-1}(D)(Y)) \subseteq del^{-1}(D)(ext(w, Y))$. ▮

- $(\Longleftarrow)$:

  Let $z \in del^{-1}(D)(ext(w, Y))$. Then there exist $n \in \omega$, $z_0 \in (\mathcal{T}^{(I \cup O)})^n$ and $z_1 \in (\mathcal{T}^D)^n$ such that $z = (z_0 \cup z_1)$.

  By definition of the del^{-1} operator, $z_0 \in ext(w, Y)$. Therefore, $(w \cdot z_0) \in Y$. It follows that $(w' \cdot z) \in del^{-1}(D)(Y)$, and so $z \in ext(w', del^{-1}(D)(Y))$.

  Therefore $del^{-1}(D)(ext(w, Y)) \subseteq ext(w', del^{-1}(D)(Y))$. ▮

QED Lemma 5.17

5.5.2 Examples

Theorems 5.14 and 5.15 may be used to characterize the full set of allowed substitutions for a subcircuit such that the input-output behavior of the full synchronous circuit is maintained as is (resynthesis), and to characterize the full set of allowed substitutions for a subcircuit such that the behavior of the full synchronous circuit meets a predetermined specification (synthesis and rectification). Our results generalize known solutions to these problems, as discussed in Section 1.5.3. In this section, we provide examples of these applications.

In the first two examples of this section, we walk through the derivation of the most general characterization of allowed substitute circuitry for a subcircuit of a given circuit, and describe some of the optimized circuits allowed by this derived specification. In the first example, the subcircuit's inputs are not constrained by the circuit of which it is a part; as a result, the derived specification has an empty F-set. In the second example, we illustrate how the F-set of the specification incorporates constraints placed by the surrounding circuitry on the values input to the subcircuit.

In our third example, we reconstruct the motivating example of [122] in our model, in order to illustrate how sequential trace structures may be used to express multiple Boolean relations. Finally, we provide a simple rectification example.

Example 5.11 (Example 38 of **[55])**

Consider the circuit of Figure 5.26. We would like to know what we may substitute for the inverter without modifying the full circuit's input-output behavior. In other words, what may we place in the blank box of Figure 5.27 so that the entire new circuit model conforms to our model of the circuit in Figure 5.26?

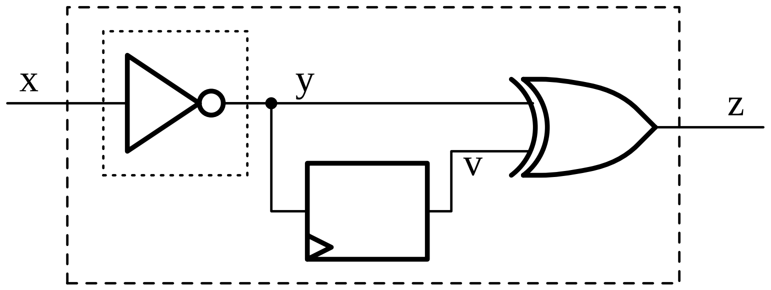

Figure 5.26: Original circuit as specification (Example 5.11)

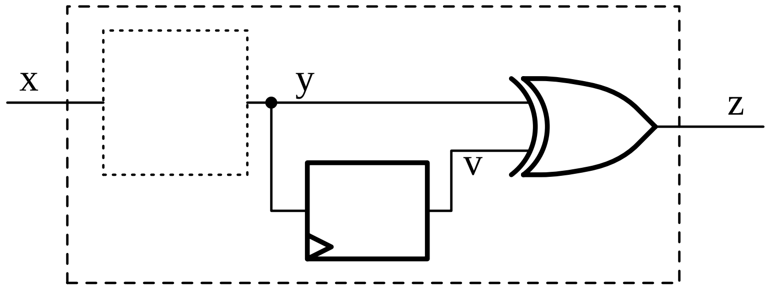

Figure 5.27: Subcircuit substitution

Recall that in Example 5.5 we derived a sequential trace structure representation of the circuit of Figure 5.26. Its P-set appears in Figure 5.9 on page 133. Call that trace structure T. Example 5.5 also contains trace structure representations of the components of T : the ones that concern us are L1 and XOR1.

Assume we maintain all sequential trace structures in canonical form at all times, so that $T^{MaxEnv} = mir(T)$. *Then according to Theorem 5.15, the most general specification for what may be placed in the blank box of Figure 5.27 is the sequential trace structure defined by the following circuit algebra expression:*

$$\begin{aligned} & f(del(\{y,v\})(\alpha_{\{x\},\{y\}} \parallel (L1 \parallel XOR1)), T) \\ &= f((\alpha_{\{x\},\{y\}} \parallel (L1 \parallel XOR1)), del_O^{-1}(\{y,v\})(T)) \\ &= f(\alpha_{\{x\},\{y\}}, mir(del(\{v,z\})(L1 \parallel XOR1 \parallel mir(del_O^{-1}(\{y,v\})(T))))) \\ &= mir(del(\{v,z\})(L1 \parallel XOR1 \parallel mir(del_O^{-1}(\{y,v\})(T)))) \\ &= mir(del(\{v,z\})(L1 \parallel XOR1 \parallel del^{-1}(\{y,v\})(mir(T)))) \end{aligned}$$

The final step in this equation follows from the sequential analog of Lemma 4.39.

We reduce this expression to

$$T' = (\{x\}, \{y\}, S', \emptyset)$$

where S' is the language accepted by the automaton of Figure 5.28.

A sequential trace structure conforms to T' if and only if it models legitimate substitute circuitry for the original inverter in our circuit. The allowed substitute subcircuits include an inverter (as in

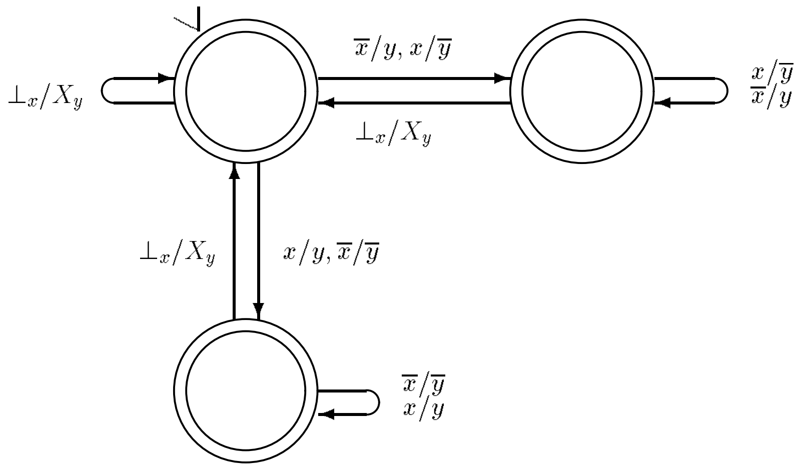
$\overline{x}/y, x/\overline{y}$
$\perp_x/X_y$
$\perp_x/X_y$
$x/\overline{y}$
$\overline{x}/y$
$\perp_x/X_y$
$x/y, \overline{x}/\overline{y}$
$\overline{x}/\overline{y}$
x/y

Figure 5.28: S-set of the most general specification for a replacement subcircuit (Example 5.11)

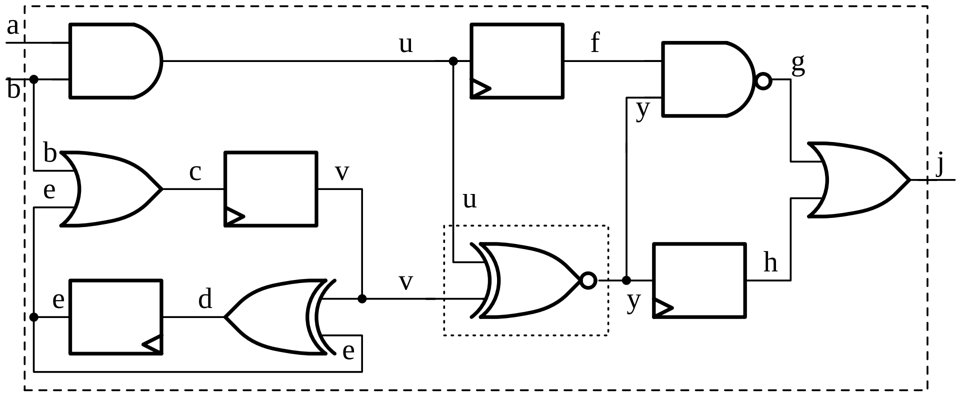

Figure 5.29: Circuit to be optimized (Example 5.12)

the original) and a non-inverting buffer.

In the preceding example, the most general characterization of all allowed subcircuits has an empty F-set. This is because the input to the subcircuit is an input to the full circuit, which is itself failure-free. In the following example, we briefly discuss a circuit and subcircuit for which this is not the case.

Example 5.12 (Examples 5 and 7-11 of [57])

Consider the circuit of Figure 5.29. We assume that the node labeled v *is initialized to the value* 1 *and that the node labeled* e *is initialized to the value* 0.

We wish to determine the most general specification which fully characterizes the allowed substitutions for the highlighted nxor-gate, such that the input-output behavior of the full circuit remains *unchanged.*

We solve for this most general characterization of the allowed substitutions according to Theorem 5.15, and derive a trace structure T' whose P-set P' has a minimal automaton representation of 26 states. Among these states is a reachable F-final state. P' admits an nxor-gate, of course. An and-gate may be substituted for the nxor-gate, as well. In addition, P' allows all correct clocked circuits that may be substituted here without modifying the input-output behavior of the full circuit.

Clearly an automaton of this size is too cumbersome to depict graphically. However, we note the following pattern: during the first and third clock cycles of the behavior described by the automaton, any input-output combination $w \in \mathcal{T}^{(I \cup O)}$ which assigns to v one of the values 0 or $\perp$ labels an edge leading to the automaton's F-final state. Thus any allowed substitute circuitry may assume that in its first and third clock cycle of operation, its input wire labeled v holds value 1 : in response to any other value on this wire at the relevant clock tick, it may exhibit arbitrary behavior – during the current and all subsequent clock cycles. This degree of freedom in the allowed implementations for the subcircuit follows directly from the fact that the values on the input wires of the designated subcircuit are constrained by the remainder of the full circuit.

As is clear from the size of the preceding example, we have automated these computations; there is no way to keep track of a trace set of the complexity of P' otherwise. We will introduce our algorithms and describe our software in Chapter 6.

In the following example, we reconstruct the motivating example of Sentovich et al's [122], in order to illustrate how our sequential trace structures provide the expressiveness of their MBRs. In Section 1.5.3 we discussed the relative expressiveness of MBRs and nondeterministic Mealy machines. We explained that MBRs are less expressive than these machines: nondeterministic Mealy machines may be used to keep track of the solutions still available within an MBR, given the input-output value combinations that we have committed to so far, but multiple distinct minimal nondeterministic Mealy machines may represent the same MBR. The following example attempts to make this more concrete.

Example 5.13 (Example 1.1 of [122])

Consider the automata $M1$ and $M2$ of Figure 5.30. They represent the behavior of the sequential trace structures $T_1 = (\{a\}, \{b\}, S_1, \emptyset)$ and $T_2 = (\{b\}, \{c\}, S_2, \emptyset)$, respectively. Their states are marked with the names of the states in the automata of Example 1.1 of [122] to which they correspond. In that example, MBRs were used to represent the degrees of freedom available for the correct implementation of combinational circuitry at each state of the automaton T_1. Using sequential trace structures, we identify all the degrees of freedom for implementing the subcircuit T_1 given that the design of T_2 is stable.

Define T to be their composition:

$$T = del(\{b\})(T_1 \parallel T_2)$$

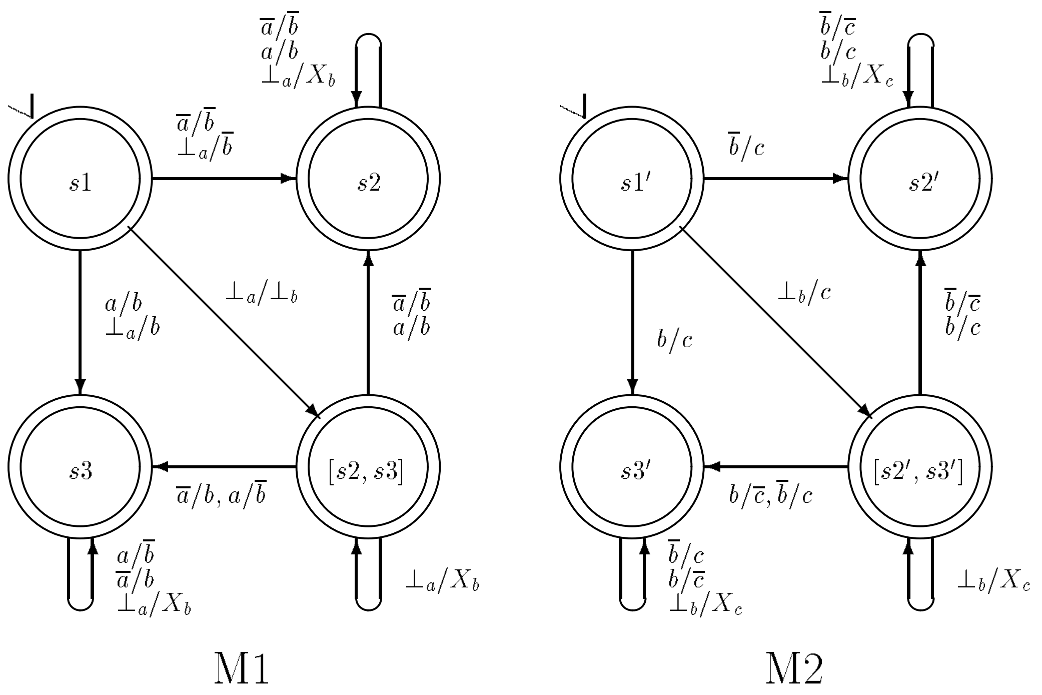

Figure 5.30: Specifications of two components

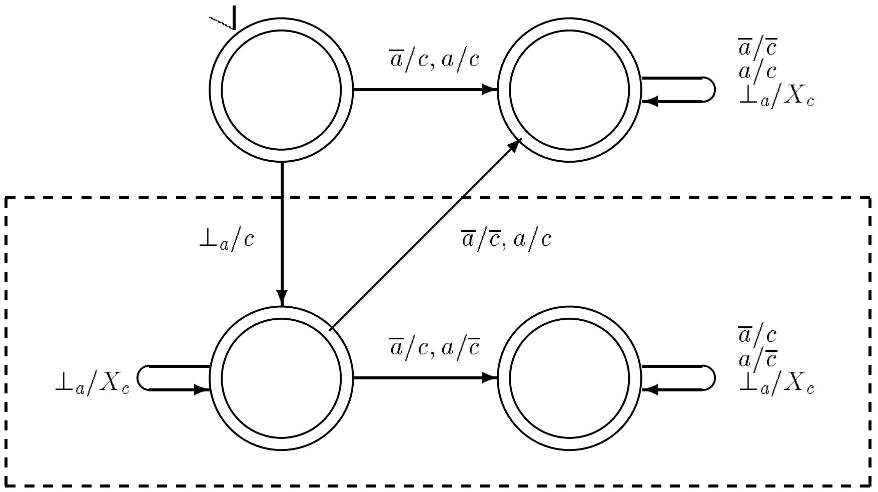
$\overline{a}/c, a/c$
$\overline{a}/\overline{c}$
a/c
$\perp_a/X_c$
$\perp_a/c$
$\overline{a}/\overline{c}, a/c$
$\overline{a}/c, a/\overline{c}$
$\overline{a}/c$
$a/\overline{c}$

Figure 5.31: Automaton representation of the behavior of $T = del(\{b\})(T_1 \parallel T_2)$

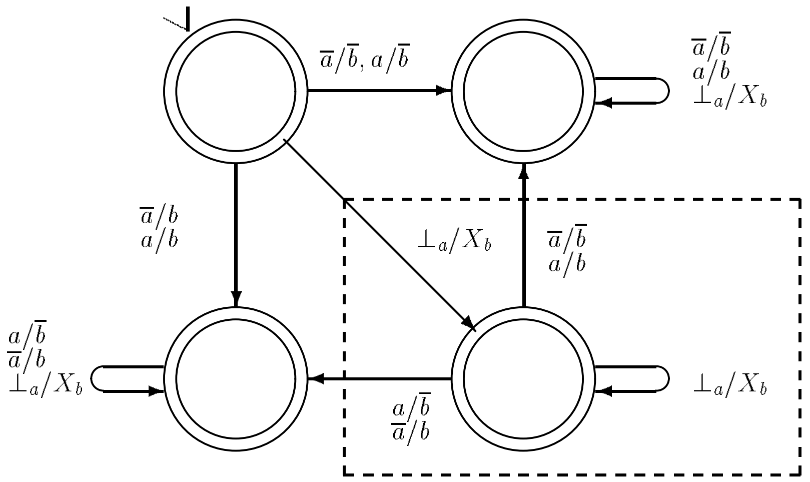

Figure 5.32: Behavioral specification of the allowed substitutions for T_1

A minimal automaton representation of the P-set of T appears in Figure 5.31. Note the highlighted area of this automaton, which has no counterpart in the Boolean example that we are reconstructing.

We seek to determine the most general specification T_1'' for T_1' such that

$$del(\{b\})(T_1' \parallel T_2) \preceq T$$

Applying Theorem 5.15, we determine that

$$T_1'' = mir(del(\{c\})(T_2 \parallel del^{-1}(\{b\})(mir(T))))$$

Figure 5.32 depicts the minimal automaton that accepts the P-set P_1'' of T_1''. Note that the effect of our additional wire value is transparent: as soon as a stable Boolean input value appears, every implementation T_1' allowed by this specification must stabilize to one of the allowed Boolean solutions. P_1'' allows all the solutions found by Sentovich et al. [122].

We conclude with a simple rectification example.

Example 5.14 Redesign example

Figure 5.33 depicts a circuit C_1, surrounded by rectification circuitry. C_1 is represented by a sequential trace structure $T_1 = (\{y\}, \{z\}, S_1, \emptyset)$, which may be derived by appropriate composition and hiding of the obvious component trace structures. We seek to rectify this circuit to implement the new specification $T_{Spec} = (\{x\}, \{w\}, S_{Spec}, \emptyset)$. An automaton that accepts S_{Spec} appears in Figure 5.34.

We will take two distinct approaches to this rectification problem. In the first approach, we seek to rectify the circuit by adding hardware to the original circuit C_1 as indicated in Figure 5.33. In the second, we identify a candidate component for replacement within C_1, and determine the most

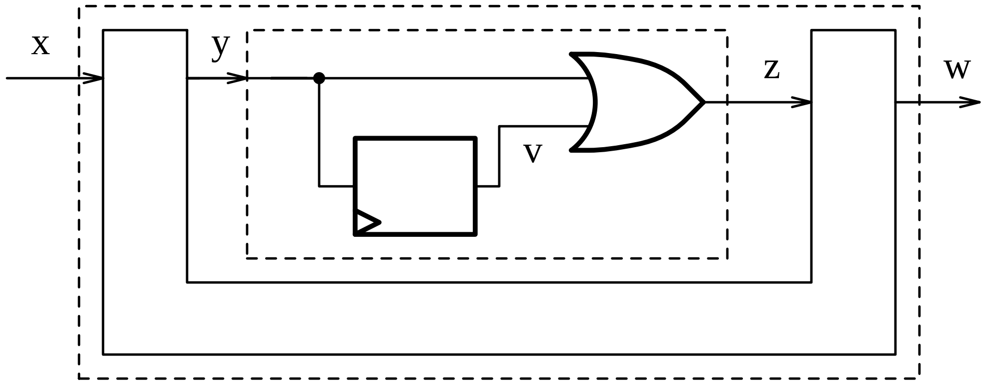

Figure 5.33: Rectification by addition of hardware

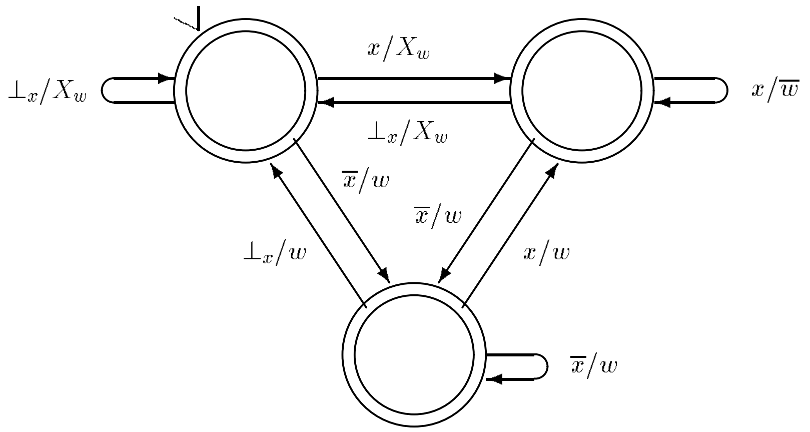

Figure 5.34: Behavioral specification S_{Spec} for the rectified circuit

general specification for the appropriate replacement circuitry.

In our first approach to the problem, we seek to determine $T_2 = (\{x, z\}, \{y, w\}, S_2, \emptyset)$ *such that*

$$del(\{y, z\})(T_1 \parallel T_2) \preceq T_{Spec}$$

Obviously, there are many possible such T_2. *We apply Theorem 5.14 to derive the most general specification* T_2' *of* T_2 *such that* $del(\{y, z\})(T_1 \parallel T_2) \preceq T_{Spec}$: *it is* $T_2' = (\{x, z\}, \{y, w\}, S_2', \emptyset)$, *whose S-set* S_2' *has a minimal automaton representation with ten states.*

Both T_{Spec} *itself and a minimal solution* T_2 *conform to* T_2'. *The minimal solution is*

$$T_2 = (T_{inv-xy} \parallel T_{buf-zw})$$

which inverts the input to T_1 *and passes its output value through unchanged.*

Another approach to modifying C_1 *to meet the new specification is to identify a subcircuit for*

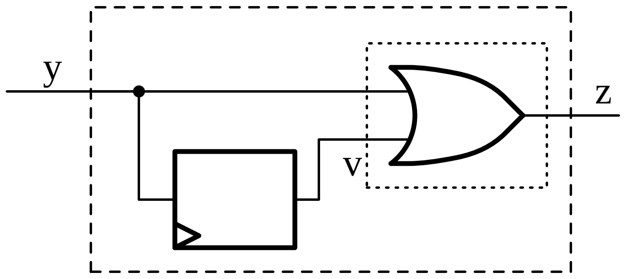

Figure 5.35: Rectification by modification of existing hardware

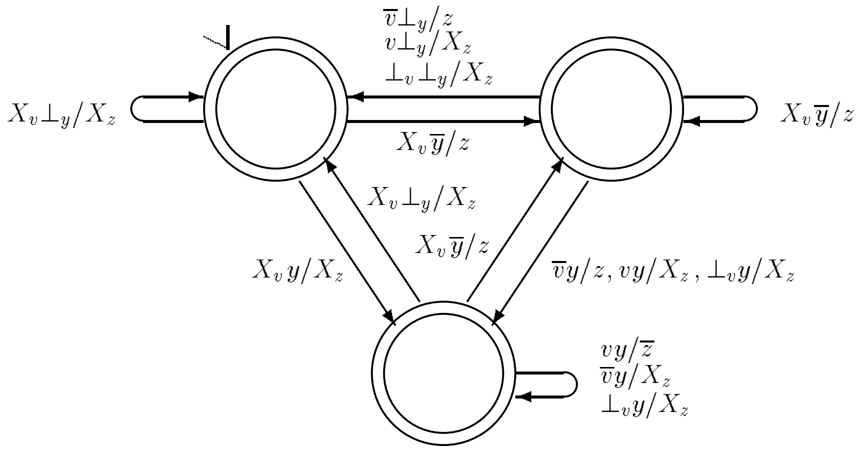

Figure 5.36: Behavioral specification S_3' of the allowed subcircuits T_3

possible modification. In this case, we identify the or-gate of C_1 as a candidate for modification (see Figure 5.35). Now we seek the most general specification T_3' for T_3 such that

$$del(\{v\})(T_3 \parallel T_{Dff-yv}) \preceq ren([x \mapsto y, w \mapsto z])(T_{Spec})$$

Let $T_{Spec-yz} = ren([x \mapsto y, w \mapsto z])(T_{Spec})$. Applying Theorem 5.14, we derive

$$T_3' = (\{y, v\}, \{z\}, S_3', \emptyset)$$

where a minimal automaton representation of S_3' appears in Figure 5.36.

Despite the fact that the latch component of C_1 constrains the relation between the values on the nodes y and v in consecutive clock cycles, this specification is failure-free. We might expect that a clock cycle in which $y = 1$ cannot be followed by a clock cycle in which $v \neq 1$. We would expect this constraint to manifest as a non-empty F-set for T_3'. However, it does not: the F-set of T_3' is empty.

The reason is that X-effects cancel any certainty about the value of y on the previous clock cycle, as follows. Let T_3 be any sequential trace structure that conforms to T_3'. Consider any circuit C_3 that is reasonably modeled by T_3. If the input wire y of C_3 appears to hold the value 1, it may be the case that y stabilized to 1 during this clock cycle, or it may be the case that y actually has value $\perp$. If y really holds the value 1, then v will stabilize to the value 1 in the following clock cycle. However, if the y node actually holds the value $\perp$ (that is, it did not stabilize to any Boolean value), then the latch component modeled by T_{Dff-yv} may output an arbitrary value on the node v in the following clock cycle. In other words, the latch too may perceive the $\perp$ value on node y to be any of $\perp$, 0, or 1, and may react accordingly in the following clock cycle. The model T_3 cannot distinguish the case in which y holds 1 from the case in which y holds $\perp$, and so no assumption may be made about the value of the node v in the following clock cycle. The same argument holds for the case in which y appears to hold the value 0. Therefore, the constraints imposed by the latch do not manifest as failure traces in T_3'.

Recall that in sequential trace theory, X-effects accurately *reflect the meaning of $\perp$. In this case, X-effects manifest in the fact that the latch and C_3 need not perceive this $\perp$ value in the same way.*

Chapter 6

Verification Algorithms

6.1 Introduction

6.1.1 Introduction

In Chapters 4 and 5, we presented theorems that provide the theoretical underpinnings of a decision procedure for conformance. However, converting these theorems and consequent algorithm outline into an actual decision procedure requires that we know of an effective procedure for each of the canonicalization steps. In particular, we require a decision procedure for determining the autofailures of a sequential trace structure. We also require an effective procedure for determining whether a combinational relation structure is an autofailure. It turns out that the latter is a special case of the former, and that this procedure involves determining whether a related T' is indeed a sequential trace structure (or combinational relation structure).

In this chapter, we present our algorithms for determining whether or not a structure is a sequential trace structure (or a combinational relation structure), and for performing the canonicalization of both sequential trace structures and combinational relation structures. We also address the computational complexity of canonicalization and of deciding conformance, and discuss our software implementation of these procedures.

6.1.2 Representations

In discussing our algorithms, we assume the structure (I, O, S, F) is represented by a triple consisting of the set I, the set O, and a finite-state machine M that represents *both* S and F. M is a *combined automaton,* as introduced in Section 5.2.3: it has two final sets, one for S and one for F. Employing the first of these two final sets produces $\mathcal{L}(M_1) = S$, and employing the second produces $\mathcal{L}(M_2) = F$. Of course, employing their union produces $\mathcal{L}(M_3) = P$. The two sets of final states need not be disjoint, as S and F themselves may overlap.

Let $A = I \cup O$. Then $M = \langle \Sigma, Q, \{q_0\}, \langle F_1, F_2 \rangle, \delta \rangle$, where $\Sigma = \mathcal{T}^A$, $F_1 \subseteq Q$ is the set of S-final states, and $F_2 \subseteq Q$ is the set of F-final states. We assume all $q \in Q$ are reachable from q_0, and that $(F_1 \cup F_2) = Q$. Because $\Sigma = \mathcal{T}^A$, M may be a deterministic finite-state automaton and yet not be a deterministic Mealy machine (see Section 2.2.3).

A word is also in order concerning our non-standard use of a single automaton for the combined representation of S and F. The determinization process creates new states which represent subsets of the original Q. In determinizing M, we require that any composite state introduced by this process be considered an F-final state if any of its component states are F-final, and that it be considered an S-final state if any of its component states are S-final. (Recall that S and F need not be disjoint). Thus, determinizing M preserves $\mathcal{L}(M_1) = S$ and $\mathcal{L}(M_2) = F$.

6.1.3 New notation

In this chapter we will utilize some new notation that is intended to aid in the clarity of the presentation.

We use the new terms I-downward-closed and O-downward-closed, in order to disambiguate between "input-downward-closed" in a context in which I is the input-set and O is the output-set, and in a context in which O is considered to be the input-set and I to be the output-set. Similarly, we may employ the new terms O-upward-closed and I-upward-closed in place of the ambiguous term "output-upward-closed." Note that I-upward-closed has the same meaning as "OUC in (O, I)," I-downward-closed has the same meaning as "IDC in (I, O)," etc. By extension, we define the terms (I, O)-receptive and (O, I)-receptive to mean the same as "receptive in (I, O)" and "receptive in (O, I)," respectively.

In addition, we may use the notation $\exists z' \geq z$ as shorthand for $\exists z' \in \langle\text{the appropriate set}\rangle . z \leq z'$. In our case, the appropriate set will always be one of $\mathcal{T}^I$ or $\mathcal{T}^O$, and will be obvious from the context.

6.2 Is T a sequential trace structure?

6.2.1 Introduction

Given a structure of the *form* of a trace structure, checking whether or not it is a sequential trace structure consists of checking for the regularity of S and F, the prefix-closure of S and P, the input-downward-closure of F and P, the non-emptiness of P, and the receptiveness of P.

We assume that the languages S, F and P are given to us in the form of an automaton or automata, so that they are regular by definition. Such a presentation also makes trivial the checks for prefix-closure of S and P and the non-emptiness of P.

We focus here on our algorithms for checking of a regular set that it is input-downward-closed, and for checking receptiveness.

6.2.2 Checking input-downward-closure

By analogy to the procedure presented in Section 5.4.3 for implementing the abstract OUC operator by a concrete operator OUC_{impl}, we define an IDC_{impl} operator that implements the abstract IDC operator. The IDC operator adds to its operand set all and only those traces necessary to make it input-downward-closed.

We check for input-downward-closure of a regular set W, represented by an automaton, by testing whether the languages W and $IDC_{impl}(W)$ are equal. This involves only known algorithms over finite-state automata.

6.2.3 Is P receptive?

6.2.3.1 Introduction

Let $T = (I, O, P)$. Let M be a *deterministic* finite-state automaton such that $\mathcal{L}(M) = P$. The relation between this representation and that of Section 6.1.2 is clear: we need merely combine the two final sets F_1 and F_2 into a single final set $F_3 = (F_1 \cup F_2)$.

In order to check that for each $w \in P$, there exists $C \subseteq ext_I(w, P)$ with the required properties, we will check of each state of M that there exists C contained in its set of out-edges that has the required properties. Note that in checking this property, we may ignore the target-states of these edges.

The transition function of M is δ, which is the union of all out-edge sets. For each $q \in Q$, let δ_q be the out-edges of q. Formally, $\delta_q = \{\langle q_1, e, q_2\rangle \mid q_1 = q\}$. Then $\delta = \bigcup_{q\in Q} \delta_q$. We define $lab(\delta_q)$ to be the *labels* on the edges in δ_q. Formally, $lab(\langle q, e, q'\rangle) = e$, and the natural extension of this function to sets yields $lab(\delta_q)$ equal to the set of edge-labels on outedges of q. Our algorithm checks, for each δ_q, that $lab(\delta_q)$ contains a subset C which is total in $\mathcal{T}^I$ and which has the upward-chains property.

Note that for each $q \in Q$, the edge-labels of δ_q are precisely $ext_I(w, P)$ for every $w \in P$ such that $q_0 \overset{w}{\Rightarrow} q$. Formally, for M any deterministic finite-state automaton that accepts P,

$$\forall q \in Q . \forall w, w' \in P . [[q_0 \overset{M,w}{\Longrightarrow} q \text{ and } q_0 \overset{M,w'}{\Longrightarrow} q] \Longrightarrow [ext_I(w, P) = ext_I(w', P)]]$$

and $\forall q \in Q . \forall w \in P . [[q_0 \overset{M,w}{\Longrightarrow} q] \Longrightarrow lab(\delta_q) = ext_I(w, P)]$. Because P is prefix-closed, and all states $q \in Q$ are reachable, P is receptive if and only if *every* $lab(\delta_q)$ has the required properties.

Formally, we verify of each δ_q that there exists $C \subseteq lab(\delta_q)$ such that C is total in $\mathcal{T}^I$ and such that

$$\forall x, x' \in \mathcal{T}^I . \forall y \in \mathcal{T}^O . [[x \leq x' \wedge (x \cup y) \in C] \Longrightarrow \exists y' \in \mathcal{T}^O . y \leq y' \wedge (x' \cup y') \in C]$$

If there exists $q \in Q$ for which there exists no such $C \subseteq lab(\delta_q)$, then P is not receptive. If there exists such C for every $q \in Q$, then P is receptive.

6.2.3.2 Searching for $C \subseteq lab(\delta_q)$

In order to prove or disprove the existence of a subset C with the required properties, we note the following fact.

Lemma 6.1 *Let $I \cap O = \emptyset$ and $Y \subseteq \mathcal{T}^{(I \cup O)}$. Let $C_1 \subseteq Y$ and $C_2 \subseteq Y$ both be total in $\mathcal{T}^I$ and both have the upward-chains property. Then $C = (C_1 \cup C_2)$ is also contained in Y, is total in $\mathcal{T}^I$ and has the upward-chains property.*

Proof:

The first two claims concerning C are obvious. We prove that C has the upward-chains property.

Let $x, x' \in \mathcal{T}^I$ such that $x \leq x'$. Let $y \in \mathcal{T}^O$ such that $(x \cup y) \in C$. We must prove that there exists $y' \in \mathcal{T}^O$ such that $y \leq y'$ and $(x' \cup y') \in C$.

We know that either $(x \cup y) \in C_1$ or $(x \cup y) \in C_2$ (or both). Say $(x \cup y) \in C_1$. Then by the upward-chains property of C_1, there exists $y' \geq y$ such that $(x' \cup y') \in C_1 \subseteq C$. If $(x \cup y) \notin C_1$ then $(x \cup y) \in C_2$. In this case, the upward-chains property of C_2 allows us to conclude that there exists $y' \geq y$ such that $(x' \cup y') \in C_2 \subseteq C$. Thus in either case, there exists $y' \in \mathcal{T}^O$ such that $y \leq y'$ and $(x' \cup y') \in C$. ∎

Thus it suffices to find the largest qualifying $C \subseteq ext_I(w, P)$, or prove it does not exist. We attempt to find such maximal C by pruning from δ_q only those elements that violate the upward-chains property. We then check whether or not the remaining subset of $lab(\delta_q)$ is total in $\mathcal{T}^I$. If it is, we have found the largest qualifying C, and hence there exists an appropriate C. If not, there exists no qualifying C, and this δ_q violates the receptiveness property.

In order to facilitate our search for the maximal $C \subseteq lab(\delta_q)$, we enter the elements of δ_q into a table in which each of the $3^{|I|}$ buckets is labeled with an element of $\mathcal{T}^I$. The elements in the bucket labeled $x \in \mathcal{T}^I$ are those out-edges of q whose label has input-value combination part x. More formally, for each $x \in \mathcal{T}^I$, the elements of the bucket labeled x are all of the form $\langle q, (x \cup y), q' \rangle$ for some $y \in \mathcal{T}^O$ and $q' \in Q$.

Consider the Hasse diagram of the partial order $\leq$ over $\mathcal{T}^I$. It may be considered to be divided into rows, labeled from 0 to $n = |I|$, where row m consists of all the elements of $\mathcal{T}^I$ containing precisely m non-$\perp$ values. Row 0 contains the unique element $\perp^I$, and row n consists of the 2^n elements of $\mathcal{T}^I$ that are Boolean vectors. In general, row m contains $2^m \cdot C(m, n)$ elements, where $C(m, n)$ is the number of ways to choose m distinct elements out of a set of n distinct elements. All elements in the same row are incomparable. Each element of row m is immediately "above" (in the partial order $\leq$) m elements of row $(m - 1)$, for $m \in \{1, 2, \ldots, n\}$. Similarly, each element of row m is immediately "below" (in $\leq$) $2 \cdot (n - m)$ elements of row $(m + 1)$, for $m \in \{0, 1, \ldots, (n - 1)\}$.

In order to check for violations of the upward-chains property, it suffices to check of every $(x \cup y) \in C$ and $x' \geq x$ *such that x' is in the Hasse-diagram row directly above that of x* that there exists some $y' \geq y$ such that $(x' \cup y') \in C$. By transitivity of $\leq$, if this property does hold in all cases, then the upward-chains property holds of C. We would also like to check this property in a single pass over δ_q. This means that we require that elements be deleted from δ_q in descending order within the partial order $\mathcal{T}^I$. If we test for the property in an order that corresponds to checking by row of the Hasse diagram of $\mathcal{T}^I$, in descending row order, we can perform the entire pruning procedure in a single pass.

In order to check for the existence of appropriate $C \subseteq lab(\delta_q)$, therefore, we would like our table to represent the Hasse diagram of $\mathcal{T}^I$. We add to bucket x of the table, either explicitly or implicitly, pointers to all buckets labeled with $x' \in \mathcal{T}^I$ such that $x' \geq x$ and such that x' is in the row directly above x in the Hasse diagram of the partial order.

Once δ_q has been entered into this data structure, we proceed to prune it by deleting those elements that violate the upward-chains property. The search-and-prune algorithm is as follows.

```
For each row of the Hasse diagram of $\mathcal{T}^I$, from second-from-the-top row *down* to bottom row:
    For each bucket $x$ in this row:
        For each element $(x \cup y)$ in bucket $x$ :
            For each bucket $x' \geq x$ in the Hasse-diagram row directly above $x$ :
                For each element $(x' \cup y')$ of bucket $x'$ :
                    If $y' \geq y$,
                      then
                            goto Check-next-bucket-$\geq$-x;
                      else
                            do-nothing (go to next element in bucket $x'$);

                    endForLoop; %% for each $(x' \cup y')$

                %% $x'$ has no $y' \geq y$ possibility ( $\nexists (x' \cup y') \in C$ such that $y' \geq y$) :
                %%
                $delete(x \cup y)$;
                goto Check-next-element-of-bucket-$x$;

                %% We did find $y' \geq y$ for $x'$;
                %%      Now we check the next $x'$ :
                %%
                Check-next-bucket-$\geq$-x:

                endForLoop; %% for each $x'$

            %% Final decision as to whether we keep $(x \cup y)$
            %%        or not has been made and executed;
            %% Go on to next element in bucket $x$ :
            %%
            Check-next-element-of-bucket-x:

            endForLoop; %% for each $(x \cup y)$
        endForLoop; %% for each $x$
    endForLoop; %% for each Hasse-diagram row, in descending row order
```

Note that all the elements of $lab(\delta_q)$ whose input-value combination part is Boolean *(i.e.,* is in the top row of the $\mathcal{T}^I$ Hasse diagram) are retained as elements of C. Because none of them can cause a violation of the upward-chains property, they all participate in the maximal C that the algorithm derives from δ_q.

This algorithm clearly terminates, as it involves no backtracking. There is repetition, as the algorithm may traverse the entries in a bucket x' multiple times. However, each time it does so, it is dealing with a different $(x \cup y) \in lab(\delta_q)$ such that $x \leq x'$.

At the termination of this algorithm, the maximal $C \subseteq ext_1(w, P)$ having the upward-chains property, for every $w \in P$ such that $q_0 \overset{w}{\Rightarrow} q$, remains in the table. This is because we have deleted precisely those elements of $lab(\delta_q)$ which violate the upward-chains property (and $lab(\delta_q) = ext_1(w, P)$ for each $w \in P$ such that $q_0 \overset{w}{\Rightarrow} q$). If the remaining set is total in $\mathcal{T}^I$, (that is, if every bucket in the table contains at least one entry), then the remaining set is a qualifying $C_w \subseteq ext_1(w, P)$ for every $w \in P$ such that $q_0 \overset{w}{\Rightarrow} q$. Therefore none of those $w \in P$ violate the receptiveness condition of P.

6.2.3.3 Complexity of the receptiveness check

The complexity of entering each δ_q into its table, preparatory to checking for the existence of appropriate $C \subseteq lab(\delta_q)$, depends on our data structures and on the original representation of M and δ. In our software implementation of the procedure described above, we use a variant of BDDs [33], which we call *ternary decision diagrams* (TDDs), to represent outedge sets. Therefore each δ_q is immediately accessible (in constant time). However, partitioning the TDD representation of δ_q according to the input-value combination of its label part takes time that is proportional to the number of nodes in that TDD, which is $3^{|I \cup O|}$ in the worst case. (The base of 3 for TDDs, rather than the usual 2 for BDDs, is a result of the fact that TDDs have *three*-way branching downward from every node, one branch for each of the three possible wire values). Exhaustive analysis of some of the possible efficiency enhancements one might add to an explicit representation, in which each element of δ_q is represented as a distinct piece of data, yields the same worst case upper bound on the time required to enter each δ_q into its table: table entry is in $\mathcal{O}(3^{|I \cup O|})$.

The search-and-prune algorithm requires time that is asymptotically polynomial in $3^{|I \cup O|}$, for each $q \in Q$. Consider the search-and-prune algorithm presented on the previous page. Its nesting structure makes it clear that this algorithm terminates in time bounded above by

$$\sum_{0 \leq m \leq n} 2^m \cdot C(m, n) \cdot 3^{n'} \cdot 2(n - m) \cdot 3^{n'} d$$

where $n = |I|$, $n' = |O|$, and d is the time required to check whether or not $y \leq y'$. Per row m of the $\mathcal{T}^I$ Hasse diagram, we traverse all $2^m \cdot C(m, n)$ buckets x, each of which may contain as many as $3^{|O|}$ elements, comparing each of these elements to all the elements (of which there are also as many as $3^{|O|}$) in each of the $2(n-m)$ buckets x' which are $\geq x$ and are in the $\mathcal{T}^I$ Hasse diagram row directly above that of x. Of course, while it is possible that every bucket contains $3^{|O|}$ elements, it is not technically possible that for *all* relevant x, x', and y, the first element $(x' \cup y')$ in bucket x' such that $y' \geq y$ is *always* the final element in bucket x'. Therefore this is a loose upper bound.

In fact our software implementation of the receptiveness check is able to check in a single pass over the data structures in the two buckets (that is, in a single pass for each $\langle x, x' \rangle$ pair) precisely which y in the x-bucket have corresponding $y' \geq y$ in the x'-bucket. This is because we represent all the elements in the x-bucket by a single ternary decision diagram (TDD), for every $x \in \mathcal{T}^I$. However, the time required in the worst case to compare *every* element in the x-bucket with *every* element in the x'-bucket is asymptotically proportional to $3^{n'} \cdot 3^{n'}$. Therefore we were unable to improve the previous upper bound on the time required for the algorithm in the *worst* case. In general, use of BDDs does not improve worst case upper bounds and does not support improvement for average case analysis either. Therefore the savings gained by our use of TDDs cannot easily be analyzed.

From this expression we derive an upper bound on the asymptotic complexity of the algorithm: it is in

$$\mathcal{O}(3^{|I|} \cdot 9^{|O|} \cdot |I|^2 \cdot d) = \mathcal{O}(3^{|I|} \cdot 9^{|O|})$$

for each $q \in Q$.

Thus the full check for receptiveness of P, including entering each δ_q into the workspace data structure and checking whether or not there exists appropriate $C_w \subseteq lab(\delta_q)$, is asymptotically bounded above by $3^{|I|} \cdot 9^{|O|} \cdot |Q|$.

6.3 Canonicalization

6.3.1 Introduction

Canonicalization involves three operations: output-upward-closure of F and P, autofailure manifestation, and failure exclusion. We have already described our implementation of the OUC operator (see Section 5.4.3). Failure exclusion can be implemented in the obvious way using known operations on finite-state automata. Autofailure manifestation can likewise be easily implemented on any finite-state automaton representation of F and P, assuming we can identify $af(T)$.

The difficulty lies in identifying $af(T)$. By definition, T_w is failure-forcing if and only if w is an autofailure. In the following subsection, we present our algorithm for checking of a sequential trace structure T whether or not it is failure-forcing. As a side-effect, the algorithm identifies the autofailures of $OUC(T)$.

Each state of the deterministic combined automaton that accepts F and P represents a set of sequences w, all of which have the same extensions in F and P. Therefore the algorithm actually applies to automaton states: its application to the states of a deterministic combined automaton that accepts F and P identifies the start-states of those $(OUC(T))_w$ which are failure forcing. Each of these states represents a subset of $af(OUC(T))$, which is what we require in order to canonicalize T. Because $OUC(T)$ is failure-forcing if and only if T is, we may focus on an FSM representation for $OUC(T)$.

The naive method for computing $af(OUC(T))$ would be to reiterate the algorithm below for each state q of a combined automaton that accepts $OUC_{I,O}(F)$ and $OUC_{I,O}(P)$, thereby determining for every $w \in OUC_{I,O}(P)$ whether or not $(OUC(T))_w$ is failure forcing. However, in determining whether or not T is failure-forcing, this algorithm computes as a side-effect the autofailures of $OUC(T)$. We may take advantage of this fact to avoid the need for multiple iterations of the full algorithm. Our software implementation takes this approach, which is formally justified in Section 6.3.2.3, below.

Formally, we note that for M a deterministic combined automaton that accepts F and P, $\forall q \in Q.\forall w, w' \in P.[[q_0 \overset{w}{\Rightarrow} q \text{ and } q_0 \overset{w'}{\Rightarrow} q] \implies [ext(w, F) = ext(w', F) \textit{ and } ext(w, P) = ext(w', P)]]$. Therefore, for such w and w', T_w and $T_{w'}$ are identical. We extend our notation and define a *state* $q \in Q$ of a deterministic combined automaton M that accepts F and P to be an autofailure, and to be failure-forcing, precisely when all the elements of $\{T_w \mid q_0 \overset{w}{\Rightarrow} q\}$ are failure-forcing, where q_0 is the start state of M.

In the following subsection, we present our algorithm for checking whether a state $q \in Q$ is failure forcing.

6.3.2 Is T failure-forcing?

6.3.2.1 Introduction

A sequential trace structure $T = (I, O, F, P)$ is failure-forcing if and only if it has no failure-free composition with any of its legal environments. Formally, for all sequential trace structures $E = (O, I, F_E, P_E)$, $(E \cap T)$ is not failure-free.

Let E_1 and E_2 be legal environments of T that have identical P-sets but differ in that $F_{E_1} = \emptyset$ and $F_{E_2} \neq \emptyset$. If $(T \cap E_2)$ is failure-free then $(T \cap E_1)$ is as well. Therefore, in order to determine whether or not T is failure forcing, it suffices to check whether or not there exists a sequential trace structure $E = (O, I, F_E = \emptyset, P_E)$ such that $(E \cap T)$ is failure-free. Such an E has P-set P_E such that $(F \cap P_E) = \emptyset$. Hence we check whether or not there exists a sequential trace structure $E = (O, I, P_E)$ such that $P_E \subseteq \overline{F}$. We attempt to find the maximal subset of $\overline{F}$ that is prefix-closed, regular, input-downward-closed (in (O, I)) and (O, I)-receptive. If this set is empty, T has no such legal environment, and hence T is failure forcing. If this set is non-empty, it is the P-set of a legal environment E of T such that $(T \cap E)$ is failure free – and thus in this case T is not failure forcing.

6.3.2.2 The algorithm

In this section we present our algorithm for deciding whether or not a given sequential trace structure is failure-forcing. The algorithm derives the maximalsubset of $\overline{F}$ which is O-downward-closed, (O, I)-receptive, regular, and prefix-closed. In order to ensure that the subset of $\overline{F}$ which we derive is prefix-closed, we "extension-close" F before complementing it. In order to ensure that the derived subset is input-downward-closed (in (O, I)), we also output-upward-close F before complementing it. We check for receptiveness of the complemented set $\overline{OUC(F \cdot A^*)}$ using the algorithm of Section 6.2.3: per state q of the resulting automaton M', we check that for every $w \in \overline{OUC(F \cdot A^*)}$ such that $q_0 \overset{M',w}{\Longrightarrow} q$ there exists $C_w \subseteq ext_1(w, \overline{OUC(F \cdot A^*)})$ with the required properties. If we encounter a state for which this does not hold, we simply delete it from consideration, and check that the remainder of the automaton does not violate receptiveness. Such a procedure is permissible because, after all, we are only looking for an acceptable *subset* of $\overline{F}$. Finally, in order to guarantee that the automaton remaining after deletion of an unacceptable state q still defines an input-downward-closed language, we apply the IDC_{impl} operator to M' before deleting any of its states. Because M' defines a language that is input-downward-closed already, application of this operator does not affect the language it defines: $\mathcal{L}(M') = \mathcal{L}(IDC_{impl}(M'))$. The formal description of the algorithm follows.

```
Input:
        Let $T = (I, O, F, P)$.
        Let $\mathcal{M}$ be a finite-state automaton accepting $F$.

$\mathcal{M}_0 \longleftarrow \mathcal{M}$ modified so that $F \longleftarrow F \cdot A^*$;
$\mathcal{M}_1 \longleftarrow OUC_{impl}(\mathcal{M}_0)$;
$\mathcal{M}_2 \longleftarrow$ the result of determinizing $\mathcal{M}_1$;
$\mathcal{M}_3 \longleftarrow$ the complement of $\mathcal{M}_2$ (swap final and non-final states);
                                        %% ALSO swap what we consider to be the input and output sets!!

$\mathcal{M}_4 \longleftarrow IDC_{impl}(\mathcal{M}_3)$;              %% where the input set is $O$ and the output set is $I$

$Q \longleftarrow$ the set of states of $\mathcal{M}_4$;
$\delta \longleftarrow$ the transition relation of $\mathcal{M}_4$;
Loop
        $Q' \longleftarrow Q$;          %% set up a second copy of the current $Q$ for later comparison
        For each state $q \in Q$ :
                $\delta_q \longleftarrow$ the outedges of $q$ in $\delta$
                If there does not exist $C_w \subseteq lab(\delta_q)$ such that
                                                $C_w$ is total in $\mathcal{T}^O$ and
                                                $C_w$ has the upward-chains property in $(O, I)$
                Then
                        %% Strip $q$ and all its in- and out-edges out of our copy of $\mathcal{M}_4$ :
                        %%
                        $Q \longleftarrow Q - \{q\}$
                        $\delta \longleftarrow \delta - (\delta_q \cup \{\langle q', e, q\rangle \in \delta \mid q' \in Q, e \in \mathcal{T}^{(I \cup O)}\})$
Until $Q = Q'$      %% Repeat until the remaining automaton is receptive or $Q' = \emptyset$
```

6.3.2.3 The correctness of the algorithm

If the remaining receptive automaton defines a non-empty language (if reachable $Q \neq \emptyset$), it defines the P-set of a sequential trace structure (whose F-set is empty) which can be composed failure-free with T. However, if reachable $Q = \emptyset$ at termination of the algorithm, any such structure has empty P-set – in other words, there does not exist such a sequential trace structure.

In order to prove this, it is necessary to prove that the algorithm does indeed derive the maximal subset of $\overline{F}$ which is O-downward-closed, (O, I)-receptive, regular, and prefix-closed.

The following facts are crucial to the correctness of the algorithm.

Lemma 6.2 *Let $I \cap O = \emptyset$. Let $W \subseteq \mathcal{B}(I \cup O)$ be an O-upward-closed set. Then $\overline{W}$ is an O-downward-closed set.*

Proof: Let $n \in \omega, x \in (\mathcal{B}^I)^n, y \in (\mathcal{B}^O)^n$ such that $(x \cup y) \in \overline{W}$. Let $y' \leq y$. We must prove that $(x \cup y') \in \overline{W}$.

Say not. Then $(x \cup y') \in W$. Thus by the O-upward-closure of W, it must be the case that $(x \cup y) \in W$ as well. But by assumption, $(x \cup y) \in \overline{W}$. Therefore it cannot be the case that $(x \cup y') \in W$, and so $(x \cup y') \in \overline{W}$. ▮

Lemma 6.3 *Let $I \cap O = \emptyset$. Let $W \subseteq \mathcal{B}(I \cup O)$. Then $\overline{OUC(W)}$ is the maximal O-downward-closed subset of $\overline{W}$ (where the OUC operator is defined in (I, O)).*

Proof: Let $Y = \overline{OUC(W)}$. Let Z be the maximal O-downward-closed subset of $\overline{W}$. Z is well-defined, because if two sets A and B are O-downward-closed, then $(A \cup B)$ is also O-downward-closed.

- $Y \subseteq Z$: By Lemma 6.2, Y is O-downward-closed. By definition of complement, Y is a subset of $\overline{W}$. Therefore $Y \subseteq Z$. ▮
- $Z \subseteq Y$: Say not. Then there exist $a \in \mathcal{B}(I)$ and $b \in \mathcal{B}(O)$ such that $(a \cup b) \in Z$ but $(a \cup b) \notin Y$. By definition of Y, this means that $(a \cup b) \in OUC(W)$. Thus there exists $b' \leq b$ such that $(a \cup b') \in W$. However, by definition of Z, $(a \cup b')$ must be in Z as well. Therefore $Z \not\subseteq \overline{W}$, which is a contradiction. Hence it must be the case that there exists no such $(a \cup b)$, and so $Z \subseteq Y$. ▮

Lemma 6.4 *Let $I \cap O = \emptyset$. If $W \subseteq \mathcal{B}(I \cup O)$ is input-downward-closed (I-downward-closed) then $W = IDC_{impl}(W)$ (where the IDC_{impl} operator is defined in (I, O)).*

Proof: By analogy to the proof in Section 5.4.3 that $OUC_{impl}(W) = OUC(W)$. (Lemma 5.8). ▮

Lemma 6.5 $OUC(W) \cdot A^* = OUC(W \cdot A^*)$

Proof: Obvious.

Lemma 6.6 *Let $I \cap O = \emptyset$. If the deterministic finite automaton M defines a language $L \subseteq \mathcal{B}(I \cup O)$, then the finite automaton resulting from the application of the IDC_{impl} operator to the M representation of L has an input-downward-closure property that is impervious to deletion of states.*

Proof: Let M be a deterministic finite-state automaton such that $\mathcal{L}(M) = L$. Let M' be the result of applying the IDC_{impl} operator to M.

Assume $M' = \langle \Sigma = \mathcal{T}^{(I \cup O)}, Q, \{q_0\}, Q_F = Q, \delta' = \bigcup_{q \in Q} \delta_q \rangle$. Let $q' \in (Q - \{q_0\})$. (If $q' = q_0$, and we remove q' from Q, then the resulting automaton trivially defines an input-downward-closed language: the empty language). Let $Q' = (Q - \{q'\})$. Let

$$M'' = \langle \Sigma, Q', \{q_0\}, Q'_F = Q', \delta'' = \delta' - (\delta_q \cup \{\langle q', e, q \rangle \in \delta' \mid q' \in Q, e \in \mathcal{T}^{(I \cup O)}\}), \rangle$$

Let $L'' = \mathcal{L}(M'')$. We must prove that L'' is input-downward-closed.

Let $w \in L''$. Let $n \in \omega$. Let $x \in (\mathcal{T}^I)^n$ and $y \in (\mathcal{T}^O)^n$ such that $w = (x \cup y)$. Let $x' \leq x$. We must prove that $(x' \cup y) \in L''$.

Because $w \in L''$, we know there exists a path $q_0, q_1, \ldots, q_n \subseteq Q'$ of length n such that

$$q_0 \overset{w_1}{\Rightarrow} q_1 \overset{w_2}{\Rightarrow} q_2 \overset{w_3}{\Rightarrow} \cdots \overset{w_n}{\Rightarrow} q_n$$

where $w_1 \cdot w_2 \cdot \ldots w_n = w$. In other words, for every $i \in \{1, 2, \ldots, n\}$, $\langle q_{(i-1)}, w_i = (x_i \cup y_i), q_i \rangle \in \delta''$. Therefore by definition of IDC_{impl}, for every $i \in \{1, 2, \ldots, n\}$, $\langle q_{(i-1)}, (x'_i \cup y_i), q_i \rangle \in \delta''$. Therefore $(x' \cup y) \in L''$. ∎

In order to prove that the algorithm is correct, we must prove that the algorithm does indeed derive the maximal subset of $\overline{F}$ which is O-downward-closed, (O, I)-receptive, regular, and prefix-closed.

We first define M_4^i by induction on $i \in \omega$: $M_4^0 = \mathcal{M}_4$ and $M_4^{(i+1)}$ is what remains of $\mathcal{M}_4$ after the i'th iteration through the inner loop of the algorithm. Q_4^k is the set of states of M_4^k.

Lemma 6.7 *The above algorithm terminates.*

Proof: For all $i \in \omega$, $Q_4^{(i+1)} \subseteq Q_4^i$. Therefore there exists minimal $k \in \omega$ such that one of the following holds: $Q_4^k = \emptyset$ or $Q_4^{(k+1)} = Q_4^k$. In both cases, the loop termination condition holds after the $(k+1)$'st iteration of the loop. ∎

Lemma 6.8 *If the above algorithm terminates after k iterations of the loop, then $\mathcal{L}(M_4^k)$ is O-downward-closed, (O, I)-receptive, prefix-closed and regular.*

Proof:

Clearly $\mathcal{L}(M_4^k)$ is regular, because it is expressed as a finite-state machine. We proceed to prove the remaining points.

- $\mathcal{L}(M_4^k)$ is O-downward-closed:

  By Lemma 5.8, $\mathcal{L}(\mathcal{M}_1)$ is O-upward-closed. By definition of FSM determinization, $\mathcal{L}(\mathcal{M}_2) = \mathcal{L}(\mathcal{M}_1)$, and therefore $\mathcal{L}(\mathcal{M}_2)$ is O-upward-closed. By Lemma 6.2, $\mathcal{L}(\mathcal{M}_3)$ is O-downward-closed. By Lemma 6.4, $\mathcal{L}(\mathcal{M}_4) = \mathcal{L}(\mathcal{M}_3)$. And finally, by definition of $\mathcal{M}_4$ and by Lemma 6.6, $\mathcal{L}(M_4^i)$ is O-downward-closed for every $i \leq k$. Therefore $\mathcal{L}(M_4^k)$ is O-downward-closed. ∎

- $\mathcal{L}(M_4^k)$ is (O, I)-receptive:

  By the loop termination condition, for every state q of M_4^k there exists $C \subseteq lab(\delta_q)$ that is total in $\mathcal{T}^O$ and that has the upward-chains property in (O, I), where δ_q are the outedges of q that remain in M_4^k. By the argument for the correctness of the algorithm for checking receptiveness (see Section 6.2.3), $\mathcal{L}(M_4^k)$ is therefore (O, I)-receptive. ∎

- $\mathcal{L}(M_4^k)$ is prefix-closed:

  By definition, $\mathcal{L}(\mathcal{M}_0) = \mathcal{L}(\mathcal{M}) \cdot A^*$. By Lemma 5.8, $\mathcal{L}(\mathcal{M}_1) = OUC(\mathcal{L}(\mathcal{M}_0))$. By Lemma 6.5, $OUC(\mathcal{L}(\mathcal{M}_0)) = OUC(\mathcal{L}(\mathcal{M})) \cdot A^*$. Therefore $\mathcal{L}(\mathcal{M}_1) = OUC(\mathcal{L}(\mathcal{M})) \cdot A^*$. As above, $\mathcal{L}(\mathcal{M}_2) = \mathcal{L}(\mathcal{M}_1)$. Clearly, therefore, $\mathcal{L}(\mathcal{M}_3)$ is prefix-closed. Because deletion of states and their in- and out-edges preserves prefix-closure, $\mathcal{L}(M_4^i)$ is prefix-closed for every $i \leq k$. Therefore $\mathcal{L}(M_4^k)$ is O-downward-closed. ∎

QED Lemma 6.8

Theorem 6.9 *If the above algorithm terminates after k iterations of the loop, then $\mathcal{L}(M_4^k)$ is the maximal subset of $\overline{F}$ that is O-downward-closed, (O, I)-receptive, prefix-closed and regular.*

Proof: By Lemma 6.8, $\mathcal{L}(M_4^k)$ is O-downward,closed, (O, I)-receptive, prefix-closed and regular. We must prove that if $L' \subseteq \overline{F}$ then either $L' \subseteq \mathcal{L}(M_4^k)$ or at least one of these four properties does not hold of L'.

Let $L' \subseteq \overline{F}$. We claim that one of the following must hold: $L' \subseteq \mathcal{L}(\mathcal{M}_3)$ or L' is not prefix-closed or L' is not O-downward-closed. As shown in the proof of Lemma 6.8, by Lemmas 5.8 and 6.5, $\mathcal{L}(\mathcal{M}_1) = OUC(\mathcal{L}(\mathcal{M})) \cdot A^*$. By definition, $\mathcal{L}(\mathcal{M}) = F$. Therefore $\mathcal{L}(\mathcal{M}_1) = OUC(F) \cdot A^*$. Thus $\mathcal{L}(\mathcal{M}_3) = \overline{OUC(F) \cdot A^*}$. Clearly, every prefix-closed subset of $\overline{OUC(F)}$ must be a subset of $\overline{OUC(F) \cdot A^*}$. And by Lemma 6.3, every O-downward-closed subset of $\overline{F}$ must be a subset of $\overline{OUC(F)}$. Therefore, every subset of $\overline{F}$ that is both prefix-closed and O-downward-closed must be a subset of $\mathcal{L}(\mathcal{M}_3)$.

Therefore, either $L' \subseteq \mathcal{L}(\mathcal{M}_3)$, or at least one of the four properties does not hold of L'.

Assume $L' \subseteq \mathcal{L}(\mathcal{M}_3)$. We claim that either $L' \subseteq \mathcal{L}(M_4^k)$ or L' is not (O, I)-receptive. If $\mathcal{M}_4$ is a deterministic finite-state machine, then this follows by the argument for the correctness of the algorithm for checking receptiveness (see Section 6.2.3). However, if $\mathcal{M}_4$ is a nondeterministic finite-state machine, the arguments given there do not obviously hold. This is because, for Q_4 the set of states of $\mathcal{M}_4$, there may exist two distinct $q_1, q_2 \in Q_4$ such that $q_0 \overset{\mathcal{M}_4,w}{\Longrightarrow} q_1$ and $q_0 \overset{\mathcal{M}_4,w}{\Longrightarrow} q_2$. In such a case, even if there exists appropriate $C_w \subseteq ext_I(w, \overline{OUC(F \cdot A^*)})$, it could hypothetically be the case that for all such appropriate C_w, $C_w \not\subseteq lab(\delta_{q_1})$ and $C_w \not\subseteq lab(\delta_{q_2})$. In that case the algorithm would erroneously remove q_1 and q_2 from M_4^k, thereby removing w from $\mathcal{L}(M_4^j)$ for some $j \leq k$.

In order to see that even when $\mathcal{M}_4$ is nondeterministic, it is still the case that every $w \notin \mathcal{L}(M_4^k)$ violates the receptiveness property in (O, I) of $\mathcal{L}(\mathcal{M}_4) = \mathcal{L}(\mathcal{M}_3)$, we must prove that in fact such a case cannot occur.

Referring back to the algorithm, we note that $\mathcal{M}_3 = \langle \Sigma, \delta, q_0, Q_3, Q_F \subseteq Q_3 \rangle$ is a deterministic automaton. Let $w \in \mathcal{L}(\mathcal{M}_3)$. Then there exists a unique $q' \in Q_3$ such that $q_0 \overset{\mathcal{M}_3,w}{\Longrightarrow} q'$. By definition, $lab(\delta_{q'}) = ext_I(w, \overline{OUC(F \cdot A^*)})$. While application of the IDC_{impl} operator to $\mathcal{M}_3$ increases δ to $\delta' \supseteq \delta$, it does not affect the set of edge-labels on the out-edges of any $q \in Q_3$. In other words, $lab(\delta'_q) = lab(\delta_q)$ for every $q \in Q_3 = Q_4$. Therefore, $lab(\delta'_{q'}) = ext_I(w, \overline{OUC(F \cdot A^*)})$ as well, and so if there exists $C_w \subseteq ext_I(w, \overline{OUC(F \cdot A^*)})$ with the appropriate properties, this C_w is a subset of $lab(\delta'_{q'})$. Therefore the hypothetical case outlined in the preceding paragraph cannot occur.

Thus if $L' \subseteq \mathcal{L}(\mathcal{M}_3)$, either $L' \subseteq \mathcal{L}(M_4^k)$, or L' is not (O, I)-receptive. Therefore either $L' \subseteq \mathcal{L}(M_4^k)$, or at least one of the four properties does not hold of L'. ∎

This concludes the proof that our algorithm derives the maximal subset having the required properties, which is a necessary and sufficient condition for its correctness.

Note that the states in $Q_4 - Q_4^k$ correspond precisely to the autofailure states of $OUC(T)$ in $\mathcal{M}_2$. This fact is necessary and sufficient to prove the correctness of our software implementation of the canonicalization process. In our software implementation, we maintain in each automaton a single sink state that is neither S-final nor F-final. This facilitates quick complementation of the language accepted by the automaton. $\mathcal{M}_2$ accepts F. We complement $\mathcal{M}_2$ to derive $\mathcal{M}_3$ by swapping the designation of each state as F-final or not. Thus each state of $\mathcal{M}_3$ corresponds to a unique state of $\mathcal{M}_2$. Q_4, the states of $\mathcal{M}_4$, are exactly the states of $\mathcal{M}_3$. As clarified in the proof above, the algorithm removes from $Q_4 = Q_3$ all and only those states that violate the receptiveness of $\mathcal{L}(\mathcal{M}_3)$. But these states correspond precisely to those autofailure states of $\mathcal{M}_2$ which were not already marked as F-final states. Therefore we may implement the canonicalization process by applying the above algorithm to $\mathcal{M}$ such that $\mathcal{L}(\mathcal{M}) = F$, and considering all the states of $Q_4 - Q_4^k$ as autofailures of $OUC(T)$. This is in fact how we derive $af(OUC(T))$ for purposes of autofailure manifestation.

6.3.2.4 The complexity of the algorithm

The worst-case time required for the algorithm to terminate is in $\mathcal{O}(4^{|Q|} \cdot 3^{|I|} \cdot 9^{|O|})$, where $|Q|$ is the cardinality of the state-set of the original automaton representation $\mathcal{M}$ of F. The computation is as follows.

Computation of $\mathcal{M}_0$ from $\mathcal{M}$ requires the replacement of δ_q for each $q \in Q_F$ by $3^{|I \cup O|}$ self-loops. The time required for this is in $\mathcal{O}(3^{|I \cup O|} \cdot |Q_F|)$ if the edge representation is explicit. Our software implementation of this algorithm requires only constant time to replace δ_q by a representation of $3^{|I \cup O|}$ self-loops, for each $q \in Q_F$. However, the program requires that the set of reachable states of $\mathcal{M}_0$ be made explicit. Hence we must traverse all of the remaining automaton in order to determine which states of $\mathcal{M}$ are now unreachable – a process which is in $\mathcal{O}(3^{|I \cup O|} \cdot |Q - Q_F|)$. Thus we again derive that the computation of $\mathcal{M}_0$ from $\mathcal{M}$ is in $\mathcal{O}(3^{|I \cup O|} \cdot |Q|)$.

Computation of $\mathcal{M}_1$ may require the addition of $3^{|I \cup O|}$ edges to each δ_q in the worst case. Hence the time this step requires is in $\mathcal{O}(3^{|I \cup O|} \cdot |Q|)$. In our software implementation, the TDD representation of δ_q must be traversed and expanded where required by the output-upward-closure operator. This process is in $\mathcal{O}(3^{|I \cup O|})$ because in the worst case the size of the TDD is $3^{|I \cup O|}$.

Computation of $\mathcal{M}_2$ requires determinization of a finite-state automaton having $|Q|$ states: this operation is exponential in $|Q|$. Let Q' be the state-set of $\mathcal{M}_2$. Then $|Q'| \leq 2^{|Q|}$. This causes the appearance of the exponential $2^{|Q|}$ in subsequent terms of the complexity bound.

Complementing $\mathcal{M}_2$ takes time proportional to the size of Q', and computing $\mathcal{M}_4$ from $\mathcal{M}_3$ takes time in $\mathcal{O}(3^{|I \cup O|} \cdot |Q'|)$, by the same reasoning used in calculating the time required to compute $\mathcal{M}_1$ from $\mathcal{M}_0$.

The nested-loop construction and explicit loop termination test of the algorithm make it clear that the receptiveness-violation check may be executed as many as

$$\sum_{1 \leq n \leq |Q'|} n = |Q'| \cdot (|Q'| + 1)/2 \in \mathcal{O}(|Q'|^2)$$

times in the worst case. By the results of Section 6.2.3, the per-state receptiveness-violation check takes time in $\mathcal{O}(3^{|I|} \cdot 9^{|O|})$ in the worst case.

Therefore the asymptotic upper bound on the worst-case time required by this algorithm to decide whether or not T is failure-forcing is

$$\mathcal{O}(2 \cdot |Q| \cdot 3^{|I \cup O|} + 2^{|Q|} + 2^{|Q|} \cdot 3^{|I \cup O|} + (2^{|Q|})^2 \cdot 3^{|I|} \cdot 9^{|O|})$$

or more succinctly,

$$\mathcal{O}(4^{|Q|} \cdot 3^{|I|} \cdot 9^{|O|})$$

6.4 Verification in Practice (Deciding Conformance)

We have implemented verification procedures for sequential trace structures using these procedures. More precisely, we utilize the results of the previous chapter in determining conformance, and we use the algorithms we have just described for implementing the details of that procedure. In order to determine whether or not $T \preceq T'$, for given sequential trace structures T and T' of the same (I, O)-type, we first apply the algorithm described above in order to determine the autofailures of $OUC(T')$. Note that in general $OUC(af(T')) \neq af(OUC(T'))$, but that T' is failure-forcing if and only if $OUC(T')$ is failure forcing. This is because although in general $OUC(T'_w) \neq (OUC(T'))_w$, the two trace structures are identical when $w = \varepsilon$.

If $OUC(T')$ is failure-forcing, then $T \preceq T'$. Otherwise, we apply autofailure manifestation and failure exclusion to $OUC(T')$, to derive $canon(T')$. We then employ our implementation of the algebraic composition operation and of the mirror operation to compute $T \parallel mir(canon(T'))$. We may determine whether or not this sequential trace structure is failure-free via a single pass over the set of states of its automaton representation.

We have implemented the algebraic operators, the mirror and canonicalization operations, and the conformance decision procedure for sequential trace structures. We can also use this software to check whether a given sequential trace structure represents correct substitute subcircuitry for a given location in a given circuit. We compute the the closed-form expression suggested by Theorem 5.14 as the specification. We may then determine, for any candidate replacement component, whether its sequential trace structure representation conforms to this specification.

Our software handles the formal verification of combinational relation structures via the synchronous theory: a combinational circuit is modeled by a sequential trace structure that repeats its combinational behavior during each clock cycle. This sequential trace structure T is failure-forcing if and only if the corresponding combinational relation structure T' is a combinational autofailure.

The computational complexity of our conformance decision procedure is dominated by the cost of determining the autofailures of $OUC(T')$. In our software implementation, composition of two trace structures T_1 and T_2 takes time in $\mathcal{O}(|Q_1| \cdot |Q_2| \cdot 3^{|I \cup O|})$, and computing the mirror of a trace structure T takes time linear in $|Q|$. However, computing the autofailures of $OUC(T')$ takes time in $\mathcal{O}(4^{|Q'|} \cdot 3^{|I|} \cdot 9^{|O|})$, as shown above. We expect the dominant factor to be the exponential, so that the asymptotic complexity of our decision procedure is polynomial in $3^{|I \cup O|}$.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we have developed a mathematical model of synchronous sequential circuits that supports both automated formal hierarchical verification and substitution. We have extended the hierarchical verification framework of asynchronous trace theory [61] to apply to both combinational circuits and to clocked sequential circuits. Our models allow nondeterministic specifications and may provide both a behavioral and a structural view of a circuit. In order to make these extensions, we have addressed the question of zero-delay cycles in a behavioral circuit model. Our solution to the zero-delay cycle problem avoids a particular class of false positives that may occur in formal verification as the result of disappearing behavior.

For substitution, we have utilized the structural view of a circuit in order to derive a formal description of the full design space available for the correct implementation of a subcircuit.

In addition to developing a theoretical framework to support behavioral and structural comparison of synchronous circuit models at various levels of detail, we have developed and implemented automatic decision procedures for both formal verification and substitution using these models.

In Chapter 2, we presented mathematical preliminaries and notation, and introduced the circuit algebra framework. In Chapter 3, we presented our formal model of combinational circuits and their requirements specifications, which incorporates our solution to the zero-delay cycle problem. These behavioral circuit models, called *combinational relation structures,* provide a *relational* model of circuit behavior. We showed that these models and the structural operations on them form a circuit algebra [61], which guarantees that our framework correctly identifies when two circuits must exhibit the same behavior.

In Chapter 4, we extended the formal hierarchical verification framework of asynchronous trace theory to combinational relation structures. One combinational relation structure correctly implements another if it may be safely substituted for it in all contexts; we call this relation *conformance.*

The outline of a decision procedure for conformance was derived using *mirrors* and results of circuit algebra. These results are an extension of those in asynchronous trace theory. However, the description of when two combinational relation structures are indistinguishable via verification is new, as it is based on the *instantaneous* interaction of a circuit with its environment. We presented a canonicalization process based on this description that provides a decision procedure for conformance. We also showed how the structural view of a circuit may be inverted to expose a subcircuit and the flexibility available for its correct implementation within a larger circuit. We derived a closed-form expression that formally describes the entire design space available for its correct implementation.

In Chapter 5, we developed the theory of *sequential trace structures,* which are our model of synchronous circuits and their requirements specifications. In order to correctly model the behavior of a synchronous circuit during a single clock cycle, these behavioral models are based on a modification of traditional Mealy machines: a behavior is a *trace,* which is a sequence of combinational behaviors. Sequential trace structures form a circuit algebra, and conformance between two sequential trace structures is defined as in the asynchronous and the combinational theories. Our description of when two sequential trace structures are indistinguishable via verification is based on the instantaneous interaction of a synchronous circuit with its (synchronous) environment within each clock cycle. This description was converted into an effective procedure for deciding conformance. And finally, we duplicated the substitution results derived for combinational models in the previous chapter, to apply to our synchronous circuit models.

In Chapter 6, we presented some details of the algorithms we employ in our software implementation of the verification procedures developed in the previous two chapters. We discussed the computational complexity of these algorithms, and proved their correctness.

7.2 Future Work

Synthesis and optimization: We have solved the problem of characterizing the space of allowed replacement components for a subcircuit in a given circuit. However, this space is very large, and so we would like to develop search techniques for finding optimal implementations. Both exact and heuristic search methods would be of interest.

More specifically, although we can derive the most general specification for allowed replacement components for a subcircuit in a given circuit, and we can determine of any candidate replacement component whether or not it is an acceptable substitution according to the desired behavior of the full circuit, we have not investigated the question of *synthesizing* a reasonable replacement component from this specification. This would involve synthesis of synchronous circuits from nondeterministic specifications, a problem that has been investigated in [54, 138] for example. It remains to be seen whether either of these approaches can be applied to sequential trace structures.

Liveness properties: Sequential trace structures can only express safety properties. If one were interested in a refinement process from very high-level specifications, one might conceivably want to express liveness properties. For example, a system specification provided early in the design process may be sufficienctly imprecise that it does not contain timeout information. In this case it could be desirable to be able to express unbounded liveness.

In order to express general liveness properties, sequential trace structures would have to be modified by dropping the prefix-closure constraint on P and by allowing infinite-length traces as behaviors. This expressiveness can easily be achieved using ω-automata. However, the current verification procedures would no longer be directly applicable. In particular, the definitions of canonicality and the canonicalization process require some thought.

Efficiency of formal verification: Currently, the asymptotic complexity of the verification procedure is dominated by the cost of determining whether a given structure of the *form* of a sequential trace structure is indeed a sequential trace structure (or may be made into one). The cost arises from the need to determine whether or not the upward-chains property holds or can be made to hold of this structure. Thus the precise definition of receptiveness directly impacts the complexity of the verification process.

One might consider modifying the precise definition of receptiveness in an attempt to obtain a more efficient decision procedure for formal verification. We have already moved somewhat in this direction: the original receptiveness definition that was proposed (for the combinational case) was the requirement that P contain a monotonic function (in (I, O)). Pratt has proved that the problem of determining for a given P whether or not it contains a monotonic function is NP-complete in $3^{|I \cup O|}$ [117]. Our best algorithm for determining whether P contains a monotonic function, in which P is represented as a set of Boolean vectors, takes time doubly exponential in $|\, I \cup O \,|$. Thus one would expect to move in the direction of even weaker approximations of full monotonicity in investigating this question.

Bibliography

[1] B. Alpern and F. Schneider. Verifying temporal properties without using temporal logic. Technical Report TR-85-723, Cornell University, 1985.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[3] B. Alpern and F. B. Schneider. Recognizing safety and liveness properties. *Distributed Computing*, 2(3):117–126, December 1987.

[4] A. Appel. Simulating digital circuits with one bit per wire. *IEEE Transactions on CAD*, 7(9):987–993, September 1988.

[5] P. Ashar, S. Devadas, and A. R. Newton. *Sequential Logic Synthesis.* Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1992.

[6] F. Baader and J. H. Siekmann. Unification theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming. Volume 2: Deduction Methodologies*, pages 41–125. Oxford University Press, 1993.

[7] A. Bailey, G. A. McCaskill, and G. J. Milne. An exercise in the automatic verification of asynchronous designs. *Formal Methods in System Design*, 4(3):213–242, May 1994.

[8] J. E. Barnes. *A Mathematical Theory of Synchronous Communication.* PhD thesis, Oxford University Computing Laboratory, 1993. Technical Monograph PRG-112.

[9] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Multilevel logic minimization using implicit don't cares. *IEEE Transactions on CAD*, 7(6):723–740, June 1988.

[10] J. F. Beetem. Hierarchical topological sorting of apparent loops via partitioning. *IEEE Transactions on CAD*, 11(5):607–619, May 1992.

[11] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[12] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: The SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, September 1991.

[13] A. Benveniste, P. Le Guernic, Y. Sorel, and M. Sorine. A denotational theory of synchronous reactive systems. *Information and Computation*, 99(2):192–230, August 1992.

[14] G. Berry, P. Couronne, and G. Gonthier. Synchronous programming of reactive systems: An introduction to ESTEREL. Technical Report (Rapport de Recherche) 647, INRIA, March 1987.

[15] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, November 1992.

[16] J. P. Billon. Perfect normal forms for discrete functions. Technical Report 87019, Bull Research Center, Louveciennes, France, June 1987.

[17] S. Bose and A. L. Fisher. Automatic verification of synchronous circuits using symbolic simulation and temporal logic. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification: VLSI Design Methods, II. Volume 2 of Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Houthalen, Belgium, 1989*, pages 151–158. North-Holland, 1990.

[18] F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1303, September 1991.

[19] R. K. Brayton. Algorithms for multi-level logic synthesis and optimization. In G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti, editors, *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, pages 197–248. Martinus Nijhoff Publishers, 1987. NATO ASI Series E, No. 136.

[20] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.

[21] R. K. Brayton and E. M. Sentovich. Network hierarchies and node minimization. *IEICE Transactions on Information and Systems*, E78-D(3):199–208, March 1995.

[22] R. K. Brayton and F. Somenzi. Boolean relations and the incomplete specification of logic networks. In G. Musgrave and U. Lauther, editors, *Proceedings of the International Conference on VLSI '89, Munich, August 1989*, pages 231–240. North-Holland, 1989.

[23] M. A. Breuer. A note on three-valued logic simulation. *IEEE Transactions on Computers*, C-21(4):399–402, April 1972.

[24] A. Bronstein. *MLP: String-Functional Semantics and Boyer-Moore Mechanization for the Formal Verification of Synchronous Circuits.* PhD thesis, Stanford University, Stanford, CA, December 1989. Tech Report STAN-CS-89-1293.

[25] A. Bronstein and C. L. Talcott. Formal verification of pipelines based on string-functional semantics. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification: VLSI Design Methods, II. Volume 2 of Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Houthalen, Belgium, 1989*, pages 349–366. North-Holland, 1990.

[26] A. Bronstein and C. L. Talcott. Formal verification of synchronous circuits based on string-functional semantics: The 7 Paillet circuits in Boyer-Moore. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems: International Workshop, Grenoble, France, June 1989*, pages 317–323. Springer-Verlag, 1990. LNCS 407.

[27] M. C. Browne and E. M. Clarke. SML: A high-level language for the design and verification of finite-state machines. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs, Proceedings of the International Working Conference, Grenoble, France, September 1986*, pages 269–292. North-Holland, 1987.

[28] M. C. Browne, E. M. Clarke, and D. L. Dill. Automatic circuit verification using temporal logic: Two new examples. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, pages 113–124. North-Holland, 1986.

[29] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, December 1986.

[30] R. E. Bryant. Race detection in MOS circuits by ternary simulation. In F. Anceau and E. J. Aas, editors, *Proceedings of the International Conference on VLSI '83*, pages 85–95. North-Holland, August 1983. Trondheim, Norway.

[31] R. E. Bryant. A switch-level model and simulator for MOS digital systems. *IEEE Transactions on Computers*, C-33(2):160–177, February 1984.

[32] R. E. Bryant. Can a simulator verify a circuit? In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, pages 125–136. North-Holland, 1986.

[33] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[34] R. E. Bryant. Algorithmic aspects of symbolic switch network analysis. *IEEE Transactions on CAD*, CAD-6(4):618–633, July 1987.

[35] R. E. Bryant. Boolean analysis of MOS circuits. *IEEE Transactions on CAD*, CAD-6(4):634–649, July 1987.

[36] R. E. Bryant. Formal verification of memory circuits by switch-level simulation. *IEEE Transactions on CAD*, 10(1):94–102, January 1991.

[37] R. E. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the ACM*, 38(2):299–328, April 1991.

[38] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A compiled simulator for MOS circuits. In *Proceedings of the 24th Design Automation Conference (DAC'87)*, pages 9–16, 1987.

[39] J. A. Brzozowski and C.-J. H. Seger. Advances in asynchronous circuit theory, Part II: Bounded inertial delay models, MOS circuits, design techniques. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 43:199–263, February 1991.

[40] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on CAD/ICAS*, 13(4):401–424, April 1994.

[41] J. R. Burch, D. Dill, E. Wolf, and G. De Micheli. Modeling hierarchical combinational circuits. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'93)*, pages 612–617, November 1993.

[42] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94(1):125–140, March 1992.

[43] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on the Principles of Programming Languages (POPL'87)*, pages 178–188, January 1987.

[44] E. Cerny and M. A. Marin. An approach to unified methodology of combinational switching circuits. *IEEE Transactions on Computers*, C-26(8):745–756, August 1977.

[45] A. C. L. Chiang, I. S. Reed, and A. V. Banes. Path sensitization, partial Boolean difference, and automated fault diagnosis. *IEEE Transactions on Computers*, C-21(2):189–195, February 1972.

[46] E. M. Clarke, I. A. Browne, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. Arnold, editor, *Proceedings of the 15th Colloquium on Trees in Algebra and Programming (CAAP'90)*, pages 103–116. Springer-Verlag, May 1990. LNCS 431.

[47] E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. *Information Processing Letters*, 46(6):301–308, July 1993.

[48] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[49] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS'89)*, pages 353–362, 1989.

[50] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Proceedings of the IEEE*, 79(9):1283–1292, September 1991.

[51] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using Boolean functional vectors. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification: VLSI Design Methods, II. Volume 2 of Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Houthalen, Belgium, 1989*, pages 179–196. North-Holland, 1990.

[52] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems: International Workshop, Grenoble, France, June 1989*, pages 365–373. Springer-Verlag, 1990. LNCS 407.

[53] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In *Proceedings of the Workshop on Computer-Aided Verification (CAV'90)*. Springer-Verlag, 1990. Also appeared as Volume 3 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS.

[54] M. Damiani. Nondeterministic finite-state machines and sequential Don't Cares. In *Proceedings, the European Design and Test Conference: EDAC, the European Conference on Design Automation; ETC, European Test Conference; and EUROASIC, the European event in ASIC design*, March 1994. Paris, France.

[55] M. Damiani. *Synthesis and Optimization of Synchronous Logic Circuits.* PhD thesis, Stanford University, Stanford, CA, June 1994. Tech Report STAN-CSL-TR-94-626.

[56] M. Damiani and G. De Micheli. Recurrence equations and the optimization of synchronous logic circuits. In *Proceedings of the 29th Design Automation Conference (DAC'92)*, pages 556–561, 1992.

[57] M. Damiani and G. De Micheli. *Don't Care* set specifications in combinational and synchronous logic circuits. *IEEE Transactions on CAD/ICAS*, 12(3):365–388, March 1993.

[58] S. Devadas, A. Ghosh, and K. Keutzer. *Logic Synthesis.* McGraw-Hill Series on Computer Engineering. McGraw-Hill, 1994.

[59] S. Devadas and K. Keutzer. An automata-theoretic approach to behavioral equivalence. *Integration, the VLSI Journal*, 12(2):109–129, December 1991.

[60] S. Devadas, H.-K. T. Ma, and A. R. Newton. On the verification of sequential machines at differing levels of abstraction. *IEEE Transactions on CAD*, 7(6):713–722, June 1988.

[61] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1989. An ACM Distinguished Dissertation 1988.

[62] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9(2):90–99, March 1965.

[63] M. Fujita, Y. Tamiya, Y. Kukimoto, and K. C. Chen. Application of Boolean unification to combinational logic synthesis. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD'91)*, pages 510–513, 1991. Subsequently expanded into [84].

[64] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1335, September 1991.

[65] A. Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2/3):151–238, October 1992.

[66] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1993.

[67] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous date flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1319, September 1991.

[68] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Euromicro'95*, September 1995. Como, Italy.

[69] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[70] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems: Proceedings of the NATO Advanced Study Institute, October 1984*, NATO ASI Series. Series F, Computer and System Sciences Vol. 13, pages 477–498. Springer-Verlag, 1985.

[71] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of Statecharts (extended abstract). In *Proceedings of the Second IEEE Symposium on Logic in Computer Science (LICS'87)*, pages 54–64, 1987.

[72] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985. Prentice-Hall International series in computer science.

[73] R. Hojati, R. Mueller-Thuns, and R. K. Brayton. Improving language containment using fairness graphs. In D. L. Dill, editor, *Computer-Aided Verification (CAV'94)*, pages 391–403. Springer-Verlag, 1994. Stanford, CA, LNCS 818.

[74] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[75] C. Huizing and W. P. de Roever. Introduction to design choices in the semantics of Statecharts. *Information Processing Letters*, 37(4):205–213, February 1991.

[76] C. Huizing, R. Gerth, and W. P. de Roever. Modelling Statecharts behaviour in a fully abstract way. In M. Dauchet and M. Nivat, editors, *Proceedings of the 13th Colloquium on Trees in Algebra and Programming (CAAP'88)*, pages 271–294. Springer-Verlag, March 1988. LNCS 299.

[77] J. S. Jephson, R. P. McQuarrie, and R. E. Vogelsberg. A three-value computer design verification system. *IBM Systems Journal*, 8(3):178–188, 1969.

[78] G. Jones. Getting your wires crossed. In R. Heldal, C. Kehler Holst, and P. Wadler, editors, *Functional Programming, Glasgow, 1991: Proceedings of the 1991 Glasgow Workshop, Portree, UK, August 1991*, Workshops in Computing Series, pages 191–206. Springer-Verlag, 1992.

[79] G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, pages 13–70. North-Holland, 1990.

[80] G. Jones and M. Sheeran. Deriving bit-serial circuits in Ruby. In A. Halaas and P. B. Denyer, editors, *Proceedings of the International Conference on VLSI '91*, IFIP Transactions A, Volume A-1, pages 71–80, August 1991. Edinburgh, UK.

[81] W. H. Kautz. The necessity of closed circuit loops in minimal combinational circuits. *IEEE Transactions on Computers*, C-19:162–164, February 1970.

[82] J. Kim and M. M. Newborn. The simplification of sequential machines with input restrictions. *IEEE Transactions on Computers*, C-21(12):1440–1443, December 1972.

[83] K. Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.

[84] Y. Kukimoto and M. Fujita. Applications of Boolean unification to combinational logic synthesis. *IEICE Transactions on Information and Systems*, E75-A(10):1212–1219, October 1992. This is an expanded version of [63].

[85] Y. Kukimoto and M. Fujita. Rectification method for lookup-table type FPGAs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'92)*, pages 54–61, November 1992.

[86] R. Kumar and T. Kropf, editors. *Proceedings of the Second Conference on Theorem Provers in Circuit Design (TPCD'94), Bad Herrenalb, Germany, September 1994*. Springer-Verlag, 1995. LNCS 901.

[87] R. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[88] R. P. Kurshan. Analysis of discrete event coordination. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, pages 414–453. Springer-Verlag, 1990. REX Workshop 1989 Proceedings. LNCS 430.

[89] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[90] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[91] B. Lin, G. de Jong, and T. Kolks. Modeling and optimization of hierarchical synchronous circuits. In *European Design and Test Conference*, March 1995. Paris, France.

[92] S. Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on CAD/ICAS*, 13(7):950–956, July 1994.

[93] S. Malik, E. M. Sentovich, R. K. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimizing sequential networks with combinational techniques. *IEEE Transactions on CAD*, 10(1):74–84, January 1991.

[94] F. Maraninchi. Argonaute: Graphical description, semantics and verification of reactive systems by using a process algebra. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems: International Workshop, Grenoble, France, June 1989*, pages 38–53. Springer-Verlag, 1990. LNCS 407.

[95] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In W. R. Cleaveland, editor, *Third International Conference on Concurrency Theory (CONCUR'92)*, pages 550–564. Springer-Verlag, August 1992. Stonybrook, NY, LNCS 630.

[96] P. N. Marinos. Derivation of minimal complete sets of test-input sequences using Boolean differences. *IEEE Transactions on Computers*, C-20(1):25–32, January 1971.

[97] U. Martin and T. Nipkow. Boolean unification – the story so far. *Journal of Symbolic Computation*, 7(3-4):275–293, March-April 1989.

[98] M. C. McFarland. Formal verification of sequential hardware: A tutorial. *IEEE Transactions on CAD/ICAS*, 12(5):633–654, May 1993.

[99] K. L. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem.* PhD thesis, Carnegie-Mellon University, 1992. Tech Report CMU-CS-92-131.

[100] K. L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–251, 1991. Tokyo, Japan.

[101] G. H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34:1045–1079, September 1955.

[102] G. De Micheli. Synchronous logic synthesis: Algorithms for cycle-time minimization. *IEEE Transactions on CAD*, 10(1):63–73, January 1991.

[103] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Series on Electronics and VLSI Circuits. McGraw-Hill, 1994.

[104] G. J. Milne. CIRCAL and the representation of communication, concurrency, and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, April 1985.

[105] G. J. Milne. Verifiably correct VLSI design. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, pages 1–22. North-Holland, 1986.

[106] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980. LNCS 92.

[107] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[108] R. Milner. *Communications and Concurrency*. Prentice-Hall International, 1989. Prentice-Hall International series in computer science.

[109] F. Moller. The definition of CIRCAL. In L. J. M. Claesen, editor, *Formal VLSI Specification and Synthesis: VLSI Design Methods, I. Volume 1 of Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Houthalen, Belgium, 1989*, pages 281–290. North-Holland, 1990.

[110] E. F. Moore. Gedanken-experiments on seuential machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956. Annals of Mathematics Studies no. 34.

[111] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The transduction method — design of logic networks based on permissible functions. *IEEE Transactions on Computers*, 38(10):1404–1423, October 1989.

[112] D. Pilaud and N. Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Sytems: Proceedings of a Symposium, Warwick, UK*, pages 99–110. Springer-Verlag, September 1988. LNCS 331.

[113] C. Pixley, V. Singhal, A. Aziz, and R. K. Brayton. Multi-level synthesis for safe replaceability. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'94)*, pages 442–449, November 1994.

[114] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, pages 510–584. Springer-Verlag, 1986. LNCS 224.

[115] A. Pnueli. How vital is liveness? Verifying timing properties of reactive and hybrid systems. Extended abstract. In W. R. Cleaveland, editor, *Third International Conference on Concurrency Theory (CONCUR'92)*, pages 162–175. Springer-Verlag, August 1992. Stonybrook, NY, LNCS 630.

[116] A. Pnueli and M. Shalev. What is in a step: On the semantics of Statecharts. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software: International Conference, Sendai, Japan, September 1991 (TACS'91)*, pages 244–264. Springer-Verlag, 1991. LNCS 526.

[117] V. R. Pratt. Private communication, January 1994.

[118] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, pages 337–351, April 1982. LNCS 137.

[119] J. Rho, G. Hachtel, and F. Somenzi. Don't Care sequences and the optimization of interacting finite-state machines. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'91)*, pages 418–421, November 1991.

[120] H. Savoj. *Don't Cares in Multi-Level Network Optimization.* PhD thesis, University of California, Berkeley, 1992.

[121] F. F. Sellers, Jr., M. Y. Hsiao, and L. W. Bearnson. Analyzing errors with the Boolean difference. *IEEE Transactions on Computers*, C-17(7):676–683, July 1968.

[122] E. M. Sentovich, V. Singhal, and R. K. Brayton. Multiple Boolean relations. In *International Workshop on Logic Synthesis, Workshop Notes*, pages 7b1–7b14, May 1993.

[123] M. Sheeran. Describing and reasoning about circuits using relations. In K. McEvoy and J. V. Tucker, editors, *Theoretical Foundations of VLSI Design: Proceedings of the Workshop on VLSI Design, Leeds, 1986*, Cambridge Tracts in Theoretical Computer Science, pages 263–298. Cambridge University Press, 1990.

[124] T. R. Shiple, H. Touati, and G. Berry. Causality analysis of circuits. Unpublished manuscript, May 1995.

[125] V. Singhal and C. Pixley. The verification problem for safe replaceability. In D. L. Dill, editor, *Computer-Aided Verification (CAV'94)*, pages 311–323. Springer-Verlag, 1994. Stanford, CA, LNCS 818.

[126] A. P. Sistla. On verifying that a concurrent program satisfies a nondeterministic specification. *Information Processing Letters*, 32:17–23, July 1989.

[127] A. P. Sistla. Proving correctness with respect to nondeterministic safety specifications. *Information Processing Letters*, 39:45–49, July 1991.

[128] V. Stavridou. *Formal Methods in Circuit Design.* Cambridge Tracts in Theoretical Computer Science 37. Cambridge University Press, 1993.

[129] V. Stavridou. Formal methods and VLSI engineering practice. *The Computer Journal*, 37(2):96–113, 1994.

[130] L. Stok. False loops through resource sharing. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'92)*, pages 345–348, November 1992.

[131] Synopsys. *HDL Compiler for Verilog: Reference Manual for Version 3.0*, December 1992.

[132] H.-Y. Wang and R. K. Brayton. Input Don't Care sequences in FSM networks. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'93)*, pages 321–328, November 1993.

[133] H.-Y. Wang and R. K. Brayton. Permissible observability relations in FSM networks. In *Proceedings of the 31st Design Automation Conference (DAC'94)*, pages 677–683, 1994.

[134] Y. Watanabe. *Logic Optimization of Interacting Components in Synchronous Digital Systems*. PhD thesis, University of California, Berkeley, 1994.

[135] Y. Watanabe and R. K. Brayton. Incremental synthesis for engineering changes. In *International Conference on Computer Design (ICCD'91)*, pages 40–43, 1991.

[136] Y. Watanabe and R. K. Brayton. The maximum set of permissible behaviors for FSM networks. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'93)*, pages 316–320, November 1993. The full version of this paper appears as [137] and as Chapter 4 of [134].

[137] Y. Watanabe and R. K. Brayton. The maximum set of permissible behaviors for FSM networks. Technical Memorandum UCB/ERL M93/61, Electronics Research Laboratory, UC Berkeley, August 1993.

[138] Y. Watanabe and R. K. Brayton. State minimization of pseudo non-deterministic FSM's. In *Proceedings, the European Design and Test Conference: EDAC, the European Conference on Design Automation; ETC, European Test Conference; and EUROASIC, the European event in ASIC design*, March 1994. Paris, France.