Chapter 1

Introduction

1.1 Motivation

With the continuous advancement of process and fabrication technologies, transistor feature sizes on VLSI circuits shrink and the complexity and degree of integration of such circuits exponentially increase, as predicted by the Moore's law. However, without CAD tools that help designers in all different aspects of designing such gigantic circuits, the utilization of the Moore's law would not have been possible.

In particular, the advent of sub-micron technologies have confronted VLSI designers with new challenges, some of which might demand whole new design methodologies. One such challenge in sub-micron design is to circumvent the limitations introduced by interconnect delay that rapidly becomes the dominant delay factor as feature sizes shrink and switching delays scale down. These parasitic limitations make distribution of signals across a chip and dealing with signal skew a serious problem, restricting the maximum performance achievable by (any) particular design style. Addressing the ever increasing demand for lower power consumption, especially for

portable applications, is among other important challenges in the design of highly integrated circuits. These challenges are of particular significance and magnitude in synchronous design styles where circuit activities are coordinated by a globally distributed periodic signal(s) called “clock”.

Synchronous design styles have been the dominant approach since mid 60’s, due to their relative ease and robustness. In such styles, the use of clock signals has introduced a level of abstraction in the time domain that hides many details about the temporal relations among circuit signals. This has greatly simplified timing analysis of such circuits, often reducing it to merely critical path analysis for the design of the clock signal. This simplification is possible because the only timing concern in a synchronous circuit is that the circuit has to be stable by the end of a clock cycle. As a result, the performance of a synchronous circuit is also a function of the worst case delay.

Recent years have witnessed extensive research on asynchronous design techniques and methodologies in an attempt to overcome, among others, the above mentioned challenges of sub-micron design. Instead of using a global clock, asynchronous circuits [77, 15, 39] use local handshaking to coordinate circuit activities and implement sequencing. Moreover, in an asynchronous design, computation starts as soon as new data is available, and once it is completed, the results can be immediately communicated via local handshaking. This more flexible and general method of operation makes asynchronous circuits highly concurrent systems.

Asynchronous circuits have the potential of outperforming their synchronous counterparts due to advantages such as elimination of clock skew problem, lower power consumption, low noise and low emission, average case instead of worst case performance, heterogeneous timing, easing of global timing issues, better potential for technology migration, automatic adaptation to fabrication and environmental variations, higher modularity, robust mutual exclusion and external input handling [83, 36]. In addition, emerging more aggressive asynchronous design techniques, that frequently use timing information to combat the full handshake overhead in area and delay by removing redundant handshakes and associated logic [73], are further improving the performance, power, area, and even testability of asynchronous designs. As a result, such advanced asynchronous design techniques are being more frequently used in stand alone designs, in interfacing synchronous circuits in different clock domains, or in heterogeneous circuits that have both synchronous and asynchronous components.

On the negative side, the lack of global synchronization and the high degree of concurrency in asynchronous circuits make their design, analysis, and verification a more serious challenge, if not an art. Without the level of abstraction that a clock signal provides in a synchronous circuit, variations in the speeds of components that are operating concurrently can no longer be ignored. In asynchronous circuit design, a great deal of attention has to be paid to the dynamic state of the circuit, avoiding “hazards” [77, 78]. Hazards are spurious signal transitions that can interfere with the correct operation of the circuit and even render its digital model invalid by taking the

circuit into a “metastable” state where one or more internal variables of the circuit take on a value *in between* the 0 and 1 designated values, possibly fluctuating in that range for an indefinite amount of time [78]. As shown in [78], a common cause of all types of hazards is the possibility for a gate to simultaneously receive contradictory signals on different inputs.

In verifying asynchronous circuits, the proper behavior of the circuit has to be assured for all possible execution paths, each corresponding to a different set of (varying) component delays, and along each such path hazard conditions (as mentioned above) have to be checked for. The nondeterminism resulting from unknown or varying component delays can lead to large number of execution paths and reachable states that can be exponential in the number of circuit components. In contrast, synchronous circuits not only have deterministic execution paths, but also have state space sizes that are *only* (at worst case) exponential in the number of state holding components (e.g., latches or flip-flops). Thus, verification of asynchronous circuits inherently suffers exponentially *more* from the so called “state explosion problem”. As a result, while symbolic model checkers--with their ability to alleviate the state explosion problem--have been successfully used in verification of large synchronous circuits, they have been far less successful in verification of asynchronous circuits of comparable sizes.

With the increased interest in asynchronous circuit design as a solution to overcome some of the bottlenecks of synchronous design in the sub-micron era, and because of the high inherent complexity of asynchronous system verification, research and development on specialized methodologies and CAD tools for the automation of

asynchronous design verification are attracting much interest. As a contribution to such efforts, this thesis presents an enhanced methodology and framework for efficient verification of a fundamental class of asynchronous circuits, speed-independent circuits, which can easily be extended/adapted to the verification of other types of asynchronous circuits such as *delay insensitive* circuits, *quasi-delay insensitive*, and also circuits with *relative timing assumptions*.

1.2 Speed-Independent Circuit Verification

Speed-independent circuits are a class of asynchronous circuits that assume the unbounded gate delay model for their components along with negligible wire delays; thus every fork in the circuit is assumed to be an *isochronic fork*, causing only negligible skew. Assuming such a delay model, an speed-independent circuit works properly for all possible ordering of events associated with all possible (and varying) relative delays of components. Seemingly restricted, speed-independence is a fundamental model based on which a broader range of asynchronous designs can be readily modeled, such as delay-insensitive designs [29, 30, 41, 17, 16, 50, 51, 56, 76], quasi-delay insensitive designs [12, 42, 23, 50], and even circuits with relative timing assumptions [73, 74, 26, 60]. For example, (quasi-) delay insensitivity of a circuit can be verified by checking speed-independence of the circuit having additional buffers (delay elements) inserted on the non-isochronic forks and input ports of the circuit [16].

The verification problems that are addressed in this thesis are checking hazard-freedom, and *conformance* of a circuit *implementation* to the circuit's *specification*. By conformance, a circuit implementation can be safely substituted for its specification with no danger in generating outputs that are not specified. The problem of checking conformance easily translates to that of checking *failure-freedom* of a *closed* circuit that is obtained by composing the circuit implementation with the *mirror* of the circuit specification. Mirroring a circuit specification yields a new circuit component, called an *environment module,* which together with the circuit implementation create a closed circuit. Failures are defined as any input signal transition at a circuit component that can disable an (previously enabled) output transition of that component. Failures described as such are thus reminiscent of *semi-modularity* failures in the circuit behavior [55, 57]. This notion of failure also covers *chokes*, where a choke is any (output) signal transition generated by the circuit implementation that is not specified in the circuit specification. Since chokes cannot thus be handled by the environment module of the circuit, they can be thought of as totally disabling the environment module, like a failure. (More formal definitions of these concepts are presented in [27].).

Interleaving semantics, also appearing in the literature as the GSW (Generalized Single Winner) race model [15], is commonly used to model the inherent concurrency in asynchronous circuit behavior. In this model of concurrency, when more than one circuit component is enabled (unstable), only one of them can change at any time. Yet, in an speed-independent circuit, concurrently enabled components always have equal chances to be the next component to change.

Theoretically, the failure-freedom of a closed circuit can be checked by performing reachability analysis over the state space of the circuit which is modeled using interleaving semantics. In practice, however, the size of the state space that can be exponential in the number of circuit components (signals), may quickly grow out of the reach of any practicable reachability analysis tool. Even symbolic reachability analysis techniques that implicitly (rather than explicitly) represent and handle (sets of) states and state transitions may soon reach their limits, even in verifying moderately sized circuits.

Research on verification of speed-independent circuits has thus focused on investigation and exploration of abstraction techniques to tackle the state space explosion problem associated with full reachability analysis. There exists a rich body of research and literature on various abstraction techniques to reduce the complexity of verification--of various properties and systems. Over-under approximations [86], assume guarantee paradigms [2], partial order techniques [1, 32, 33, 62, 63, 81, 82, 35, 37], homomorphic reductions [35, 47], divide and conquer paradigms and hierarchical approaches [47] are some of the better known general approaches that can, or have been, applied to speed-independent circuit verification in one or another way. However, there exist only a few theoretical frameworks that are specifically designed and tailored to address the verification of this fundamental class of asynchronous circuits, and yet fewer have attempted to combine various abstraction techniques for this problem. An overview of the previous work on verification of speed-independent circuits is presented next.

1.3 Related Work

The verification of speed-independent circuits has been given significant attention in the literature. Dill proposed a trace theoretic framework in which he formulated the notion of trace conformance of speed-independent circuits [27]. Trace conformance is a safety property of speed-independent circuits checking whether the circuit can generate outputs that are unexpected by its specification. Ebergen and Gingras introduced the notion of completeness with respect to a specification which is stronger than trace conformance in that it requires the circuit to be able to exhibit all the behaviors defined by the specification [31]. Gopalakrishnan *et al.* proposed a similar notion of strong conformance [34].

It is important to note that both Dill's work [27] and Ebergen and Gingras's work [31] *support* hierarchical verification of speed-independent circuits. Specifically, if a block of a circuit has been successfully verified against a specification, the block can be modeled by its specification rather than by its implementation when verifying the whole circuit. This feature is very useful since specifications can typically have more compact representations in a computer than the behavior of their corresponding implementations. Such hierarchical approaches, however, are not effective when a circuit is originally flat; i.e.; its circuit-blocks do not have specifications.

Numerous techniques have been proposed to speed up the verification process of a flat circuit. McMillan proposed a partial order approach based on a technique called *Petri-net unfolding* [53]. While very successful on some scalable examples, the worst-

case complexity is in fact no smaller than that of standard reachability analysis algorithms. Yoneda and Yoshikawa [88] proposed an extended version of a different type of partial order approach in which only a subset of interleavings of signals are needed to be explored [1, 32, 33, 62, 63, 81, 82, 35, 37]. While effective for some circuits, the run-time for other circuits was not impressive because of the high computational overhead associated with determining which interleavings to explore. Burch *et al.* proposed BDD-based techniques to implicitly analyze the circuit's state space [18]. While successful on some examples, the techniques do not improve the worst-case complexity of the algorithm. Lastly, Roig *et al.* introduced a modified symbolic breadth-first search algorithm which resulted in significant run-time improvements for some circuits, but again, the worst-case complexity of their algorithm stays the same [64].

To reduce the verification complexity, Beerel *et al.* proposed a two-phase approach in which first functional correctness (i.e., *complex-gate equivalence*) of the circuit was verified and then behavioral properties (i.e., *hazard-freedom*) were checked [7, 8]. The key to their technique is that the behavior of some of the circuit signals could be safely approximated, exponentially reducing the time and space complexity of the verification problem for many examples. Later, Roig et. al proposed a hierarchical approach which also had the advantage of approximating the behavior of some of the circuit signals [65]. Since our proposed technique is most directly related to these latter two works, we describe them in more detail.

The first step in both approaches by Beerel *et al.* and Roig *et al.* is to create a complex-gate circuit which effectively induces hierarchy by hiding the signals internal to

the complex-gates. The state space of the remaining external signals is then analyzed using standard reachability analysis techniques. In Beerel *et al.*'s technique, an analysis of this state space is used to deduce hazard-freedom of the internal hidden signals. In Roig *et al.*'s technique, projections of this state space are used as the environment of the complex-gates to verify the hidden signals. The key disadvantage of both techniques, however, is that the set of external signals needs to include all memory element outputs (i.e., memory elements cannot be hidden). Since most asynchronous circuits are dominated by memory elements, the number of external signals can still be large and their state space can be too large to analyze. This research started as an attempt to remove the above mentioned limitation on the set of external signals.

1.4 Thesis Contributions

Existing specialized frameworks have been less than successful in either fully characterizing and/or utilizing some of the unique properties of speed-independence. As an example, the specialized verification frameworks of [8, 64] use the behavior of an abstract circuit--obtained by collapsing the original circuit into a complex-gate circuit--as an abstraction of the circuit behavior which is then used to verify or deduce the failure freedom of each complex-gate. However, since they use a functional (or structural circuit) abstraction to find a behavioral abstraction--rather than a behavioral abstraction that is based on speed-independence properties--their approach, while the most coherent, has fundamental shortcomings that have been addressed by this thesis.

We have proposed a theoretical framework for verification of speed-independent circuits that incorporates a combination of different abstraction and reduction techniques to achieve efficiency. This framework is a generalization of that of [65]. We introduce the notion of a *safe abstraction* of the behavior of a set of *external* circuit variables (signals) as a behavior that is never an over-approximation of the actual behavior of external variables, and that is guaranteed to exactly resemble that behavior if the circuit is failure-free. We define the notion of *partitioning* the circuit into *circuit blocks* using the set of external signals, the notion of a *safe specification* for a circuit block that is derived from a safe abstraction, the notion of an *environment module* of a circuit block that is derived from a safe specification, and finally the notion of a *sub-circuit* as the composition of a circuit block with its environment module. We then prove the following important theorem about the relationship between failure-freedom of a circuit and failure-freedom of its sub-circuits that are derived from a safe abstraction: *a circuit is failure-free iff all of its sub-circuits are failure-free*. By this theorem (which is also the basis of the hierarchical verification framework of [65]), given a safe abstraction, the problem of verifying a circuit reduces to the problem of verifying its sub-circuits, with the verification results always being exact. Since the sub-circuits are smaller that the original circuit, and the complexity of verification is exponential in circuit size, this *divide and conquer* approach which can be recursively applied in a *hierarchical* fashion can significantly speed up the verification procedure. However, the success of this approach would heavily depend on the existence of efficient techniques for finding safe abstractions.

For efficient derivation of safe abstractions, we have proposed a novel partial order reduction approach. This approach, which substitutes the functional abstraction phase of [65], partially explores the state space of the circuit (avoiding the state space explosion problem) and constructs a sub-automaton of its behavior automaton. If the constructed sub-automaton is *projectable* onto the set of external variables, the behavior of its *projection* is shown to be a safe abstraction. We have proposed procedures that (concurrently) perform the partial order analysis, projectability check, and projection of the partial order sub-automaton onto the automaton of a safe abstraction.

We have devised our partial order technique based on some important properties of speed-independent circuits. Intuitively, in an speed-independent circuit, no output signal transition of a circuit component is ever disabled by an--*independent*--input signal transition. Here, two signals are called independent if they cannot disable each other, and a unique state is reached for different orderings of their transitions. Thus, in a speed-independent circuit, no output transitions are lost if the independent inputs are allowed to *settle* (stabilize). Based on this observation, assuming all dependent signals of a circuit are included in the set of external signals, our partial order technique always settles all independent internal variables of the circuit by any arbitrary order before exploring all orderings of transitions of external variables. The explored (external) behavior is proven to be exact if the circuit is speed-independent, and otherwise it might be an under-approximation. By this, our framework for hierarchical verification of speed-independence is also an *assume-guarantee* paradigm; assuming speed-inde-

pendence, the partial order has to explore the exact behavior of the external variables; this is guaranteed when the sub-circuits are all found to be failure-free.

The proposed approach for induced hierarchical verification of speed-independent circuits has been implemented in a CAD tool called SPHINX. SPHINX utilizes symbolic techniques using binary decision diagrams (BDDs) for efficient representation of states, state transitions, and the results of reachability analysis. It also uses an object oriented paradigm for representation and treatment of a circuit and its sub-circuits at different levels of hierarchy. SPHINX has been especially very successful in verifying speed-independent circuits that are particularly dominated by memory elements, e.g., FIFO controller circuits. This is due to its unique ability in *hiding* memory element outputs, a feature which was not supported by preceding frameworks that used functional/structural abstractions (e.g., complex-gate verification [65]).

This thesis is a presentation of my proposed theoretical framework for induced hierarchical verification of speed-independent circuits, its relationship to previous work, SPHINX--the developed CAD tool, and some experimental results. It also proposes some directions for future research, such as extending the current framework to the domain of circuits with relative timing assumptions.

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 introduces the models that we use to represent circuits and (their) behaviors. This includes our finite-state-automata based

model of a circuit as a collection of circuit modules, the notions of behavior and behavior projections together with behavior automata, sub-automata, and sub-automata projections, and finally the notion of a safe abstraction.

Our theoretical framework for induced hierarchical verification of speed-independent circuits is introduced in Chapter 3. The notions of partitioning a circuit into a set of circuit blocks using a set of external variables, a safe specification for a circuit block driven from a safe abstraction, an environment module of a circuit block driven from a safe specification, and finally the notion of a sub-circuit as the composition of a circuit block and its environment module are introduced in this chapter, all in relation to the notion of *inducing* hierarchy in a flat circuit. The consequential relationship between the failure-freedom of a circuit and that of its sub-circuits, which is the foundation of our hierarchical verification framework, is presented and proven at the end of this chapter.

In Chapter 4, we discuss some of the issues related to our hierarchical verification framework. The chapter includes a comparison of the approach with that of complex-gate verification in terms of their selection of external variables, the issue of selecting sets of external variables that can successfully induce hierarchy in verification of a circuit, and finally the concept of sequential hierarchical verification as a way of improving the performance of hierarchical verification.

Chapter 5 introduces our efficient technique for finding safe abstractions. We prove that our proposed partial order technique explores a partial behavior of the circuit that under certain conditions (projectability of its automaton) can be used to derive a safe

abstraction. We present an algorithm that concurrently performs the partial order analysis, checks the projectability of its automaton, and--if it is projectable--constructs a safe abstraction.

Chapter 6 presents a brief comparison of our verification approach with a number of other general reduction techniques and verification methodologies and tools. In particular, a more thorough comparison of our framework and that of complex-gate verification is presented in this chapter.

Chapter 7 presents a short overview of the status of our CAD tool, SPHINX, and our experimental results.

Finally Chapter 8 proposes some directions for related future research. It presents some ideas on how to extend the current framework to the domain of asynchronous circuits with relative timing assumptions. The chapter is closed by an open conjecture on the issue of using multiple safe abstractions for hierarchical verification.

Chapter 2

Models of Circuits and Behaviors

In this chapter we introduce the models that we use to represent circuits and (their) behaviors. This includes our finite-state-automata based model of a *circuit* as a collection of *circuit modules*, the notions of *behavior* and *behavior projections* together with *behavior automaton*, *sub-automaton*, and *sub-automaton projections*. The notion of a *safe abstraction* as a key component of our hierarchical verification framework is introduced at the end of this chapter.

2.1 Circuit Modules

In this section, we introduce our model for asynchronous components which we call a "circuit module". Circuit modules are the building blocks of asynchronous circuits and systems. The generic model of a component presented in this section is general enough to model different types of gates (e.g., combinational or sequential, deterministic or nondeterministic), and different types of specifications (e.g., Petri-nets, STGs, etc.).

Definition 2.1 [Circuit module] A *circuit module* is a tuple $M^i = \langle X^i, Z^i, Y^i, FA^i \rangle$, where

- $X^i = \{x_1^i, \ldots, x_{m^i}^i\}$ is the set of binary *module input variables*;
- $Z^i = \{z_1^i, \ldots, z_{p^i}^i\}$ is the set of binary *module output variables*;
- $Y^i = \{y_1^i, \ldots, y_{n^i}^i\}$ is the set of binary *module internal state variables;*
- $FA^i = \langle A^i, V^i, Q^i, \lambda^i, TR^i, \mu^i, q_0^i \rangle$ is a nondeterministic finite state automaton called the *module automaton*, where
  - $A^i = X^i \cup Z^i$ is the *input alphabet* of the automaton;
  - $V^i = X^i \cup Z^i \cup Y^i$ is the set of *module variables*, as well as *automaton variables;*
  - Q^i is the *state set* of the automaton;
  - $\lambda^i : Q^i \rightarrow L(V^i)$ is the *state labeling function* of the automaton. Here, $L(V^i)$ is the set of all surjective functions $l : V^i \rightarrow \{0, 1\}$;
  - $TR^i \subseteq Q^i \times (A^i \cup \varepsilon) \times Q^i$ is the *state transition relation* of the automaton. Here, ε is an additional symbol which identifies empty input transitions of the automaton;
  - $\mu^i : Q^i \times (A^i \cup \varepsilon) \rightarrow \{\mathrm{F}, \mathrm{S}\}$ is the *transition labeling function* of the automaton;
  - $q_0^i \in Q^i$ is the initial state of the automaton. ■

The components of a circuit module M^i are further explained below.

X^i, Y^i, and Z^i are pair-wise disjoint sets. $V^i = X^i \cup Z^i \cup Y^i$ is the set of *module variables* as indicated above. $X^i \cup Z^i$, the set of *module I/O variables*, is identical to the input alphabet A^i of FA^i. A symbol $a \in X^i$ ($a \in Z^i$) of the alphabet A^i corresponds to transitions on the associated input (output) of the circuit module.

We shall assume that an encoding scheme (by the internal state variables Y^i) is given for the internal states of the circuit module. Note that here, "internal state of the module" refers to what is required *beyond* the I/O state of the module to fully capture the module's state.

$\lambda^i : Q^i \rightarrow L(V^i)$ is an injective function assigning to each state of the automaton a unique function which in turn assigns binary values to every $v \in V^i$. As a result, each state $q \in Q^i$ is an interpretation of the module variables V^i; i.e, it assumes for every variable $v \in V^i$ a value in its binary range $\{0, 1\}$. Thus, states of the automaton correspond to total states (input/output/internal state) of the circuit module.

$TR^i \subseteq Q^i \times (A^i \cup \varepsilon) \times Q^i$ is associated with $\delta^i : Q^i \times (A^i \cup \varepsilon) \rightarrow 2^{Q^i}$, the *state transition function* of the automaton[1]. In general, any individual I/O signal transition of a circuit module is accompanied by some internal state change of the module; that is, some $Y \subseteq Y^i$ may change simultaneously and instantaneously together with an I/O signal (e.g., $a \in A^i$) transition. On the other hand, the circuit module can have internal state changes even in the absence of any I/O signal transitions.

1. Each element of the set 2^{Q_i} is one of the $2^{|Q_i|}$ subsets of the finite set Q_i.

Let $a \in A^i \cup \varepsilon$ be any symbol corresponding to a transition on the associated I/O signal (or an empty I/O transition in the case of $a = \varepsilon$), and $q, q' \in Q^i$ be any pair of automaton states. Then $(q, a, q') \in TR^i$ iff all of the following hold:

- $\lambda^i(q)(a) \neq \lambda^i(q')(a)$, and for all other I/O variables $b \in A^i$, $b \neq a$, $\lambda^i(q)(b) = \lambda^i(q')(b)$;
- there exists $Y \subseteq Y^i$ such that for all $v \in Y$, $\lambda^i(q)(v) \neq \lambda^i(q')(v)$, and for all $w \in Y^i - Y$, $\lambda^i(q)(w) = \lambda^i(q')(w)$;
- the total state of the circuit module can change according to $q' \in \delta^i(q, a)$ and through a transition of signal a. In other words, if the circuit module is at state q then a transition of signal a can take the circuit module to state q' by causing a simultaneous change in all internal state variables $v \in Y \subseteq Y^i$ of the module.

In the presence of the above conditions and if $a \in Z^i$, then we say that the output signal a is *enabled* at q. Any internal state variable $v \in Y$ is also said to be enabled at q.

Let $q \in Q^i$ be any state of the automaton. Then $(q, \varepsilon, q) \in TR^i$ is always a state transition of the automaton. In other words, every automaton state has a self-loop for ε. Such self loops represent the behavior of the module when it is *idle*; i.e., no event occurs at the module[2].

An important property of any circuit is its *receptiveness*. This property is related to the inability of a circuit to control the arrival of transitions on its inputs, and the fact

2. This notion of idle self loops is later used in composing modules' automata into a circuit automaton.

that unwanted input transitions are always possible [27, 40]. As such, any proper model for a circuit has to account for the receptiveness of all circuit components.

In our model, the receptiveness of any circuit module with respect to input signal transitions is modeled as follows: for any state $q \in Q^i$ and any *input* signal $a \in X^i$, there always exists a (some) state $q' \in Q^i$ and a corresponding state transition $(q, a, q') \in TR^i$. We say that FA^i is *complete* over X^i. In contrast, for any state $q \in Q^i$ and any *output* signal $a \in Z^i$, $q' \in Q^i$ and $(q, a, q') \in TR^i$ exist iff output signal a is *enabled* at q, but usually at each total state of the circuit module only a subset of the module's outputs are enabled to change.

Note that this model does not allow (and/or handle) simultaneous I/O signal changes; instead, all possible interleavings of simultaneously enabled I/O signal transitions are assumed to be included in the automaton of the circuit module. This convention is in accordance with interleaving semantics for circuit behavior, which we have adopted for our analysis of speed-independence. Interleaving semantics, also appearing in the literature as the GSW (Generalized Single Winner) race model [15], is commonly used to model the inherent concurrency of asynchronous circuit behavior. In this model of concurrency, when more than one circuit component is enabled (unstable), only one of them can change at any time; however, the order in which the components change cannot be predicted. For the particular case of speed-independent circuits, concurrently unstable components always have equal chance in being the next component to change.

The state transition labeling function $\mu^i : Q^i \times (A^i \cup \varepsilon) \rightarrow \{F, S\}$ labels the edges of the underlying transition diagram of the automaton (induced by TR^i). Let $q, q' \in Q^i$, $a \in A^i \cup \varepsilon$, and $(q, a, q') \in TR^i$. Then for any output signal $a \in Z^i \cup \varepsilon$ we always have $\mu^i(q, a) = S$; i.e., any state transition through an output signal transition is always considered a *success* transition. For any input signal $a \in X^i$, $\mu^i(q, a) = F$ iff the transition of a at q is an *illegal* input signal transition. If $\mu^i(q, a) = F$ then any automaton state transition $(q, a, q'') \in TR^i$ is called a *failure* transition. An illegal input transition is one which is either not expected by the circuit module (e.g., an *input choke* to an specification module), and/or one which is known to cause a circuit malfunction (e.g., a hazardous output). In particular, we shall call any input signal transition which disables a previously enabled output signal (or an internal state variable $y \in Y^i$), and thus violates *semi-modularity* [55, 57], an illegal input transition. More precisely, assume that $q, q' \in Q^i$, $x \in X^i$, $(q, x, q') \in TR^i$, and there exists an output signal $z \in Z^i$ (or an internal state variable $y \in Y^i$) which is enabled at state q but not so in state q'; then $\mu^i(q, x) = F$, marking all possible state transitions from q by the symbol x as failure transitions. In our model, illegal input transitions (e.g., chokes) do not change the internal state of a circuit module; that is, if $\mu^i(q, x) = F$, and $(q, x, q') \in TR^i$, then $\lambda^i(q)$ and $\lambda^i(q')$ differ only at the value they assign to the variable x. This convention is only to simplify the choice of a state that is entered by an illegal input transition.

A circuit module is *non-deterministic* if the firing of any output signal can ever disable another output (e.g., arbitration in an arbiter module). In such a case, the decision

of which output to fire is called a *choice*. A module which is not non-deterministic is said to be *deterministic*.

Note that state transitions caused by output signal changes are excluded from the set of failure transitions. This makes all *output choices* legal; that is, any output signal change disabling another output signal change represents a non-failure state transition in the module automaton.

$q_0^i \in Q^i$ corresponds to the initial total state of the circuit module (within a circuit).

2.2 Examples of Circuit Modules

The definition of a circuit module presented in the previous section is very general. In this section we show how elementary gates (combinational and sequential, deterministic and nondeterministic), and also higher level specifications (e.g., Petri-nets, STGs) can be modelled as circuit modules. It is to be noted that there may be many circuit module representations for a single gate/specification type. Such representations may differ in terms of the internal state encoding of the module, or the behavior manifested beyond the occurrence of a failure; however, they should all agree on the failure-free portion of their associated automata languages (the set of all I/O sequences corresponding to the failure-free runs of the associated automata), precisely capturing the I/O behavior of the physical module prior to failure occurrences.

2.2.1 Combinational Gates

A combinational gate is a deterministic circuit module $M^i = \langle X^i, Z^i, Y^i, FA^i \rangle$, such that:

- $Y^i = \varnothing$, and $V^i = A^i$;
- $FA^i = \langle A^i, A^{i}, Q^i, \lambda^i, TR^i, \mu^i, q_0^i \rangle$ is a *deterministic* finite state automaton such that:
  - $TR^i \subseteq Q^i \times (A^i \cup \varepsilon) \times Q^i$ is constructed based on the gate's functionality. Let F_j^i, $1 \le j \le p^i$, be the boolean function describing the jth output of the gate based on gate inputs; i.e., $z_j^i = F_j^i(x_1^i, \ldots, x_{m^i}^i)$. Then for any $q, q' \in Q^i$, $(q, z_j^i, q') \in TR^i$ iff $F_j^i(x_1^i, \ldots, x_{m^i}^i)\big|_{q'} = z_j^i\big|_{q'}$ and $z_j^i\big|_{q'} \neq z_j^i\big|_{q}$. Here, $.\big|_{q'}$ denotes that the function arguments are evaluated by $\lambda^i(q')$; thus the latter condition translates to $F_j^i(\lambda^i(q')(x_1^i), \ldots, \lambda^i(q')(x_{m^i}^i)) = \lambda^i(q')(z_j^i) \neq \lambda^i(q)(z_j^i)$.

Example 2.1 Figure 2.1.a depicts a NOR gate. The module description of the NOR gate is $M = \langle \{a, b\}, \{c\}, \varnothing, FA \rangle$ where the state diagram of FA, the module automaton, is depicted in Figure 2.1.b. The initial state is entered by an arrow; i.e., $\lambda(q_0) = 000$. ■

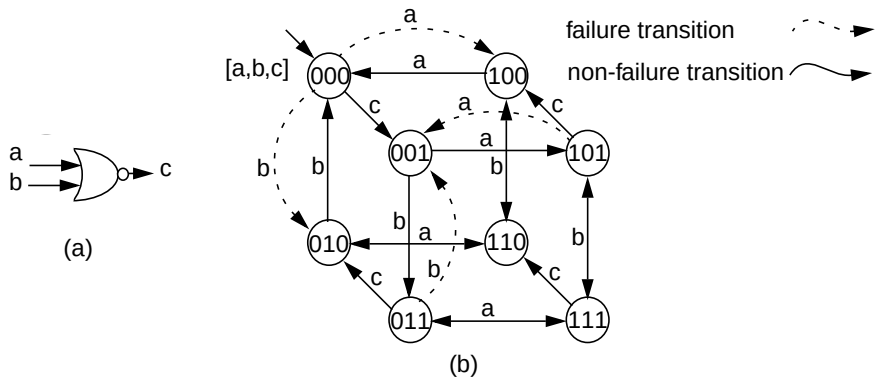


Fig. 2.1 Module description of a NOR gate.
(a) A NOR gate. (b) The module automaton of the NOR gate.

2.2.2 Sequential Gates

For most elementary sequential gates, the I/O state of the gate completely captures the state of the gate, without requiring any extra internal state variables. Examples of such gates are Flip-Flops, C-elements, and Mutual-Exclusion elements (ME). A sequential gate with no internal state variables is modeled as a circuit module $M^i = \langle X^i, Z^i, Y^i, FA^i \rangle$ such that:

- $Y^i = \varnothing$, and $V^i = A^i$;
- $FA^i = \langle A^i, A^i, Q^i, \lambda^i, TR^i, \mu^i, q_0^i \rangle$ is a *deterministic* finite state automaton such that:
  - $TR^i \subseteq Q^i \times (A^i \cup \varepsilon) \times Q^i$ is constructed based on the gate's functionality. Let F_j^i, $1 \le j \le p^i$, be the boolean function describing the next value of the jth output of the gate (denoted by z'^i_j) based on the present values of gate

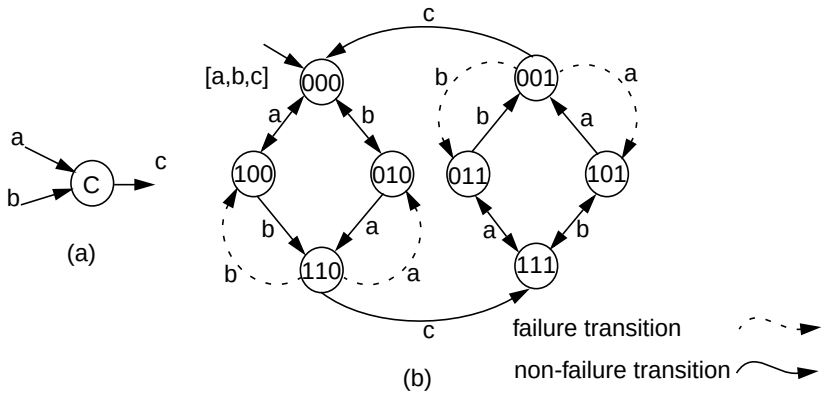


Fig. 2.2 Module description of a C-element gate.
(a) A C-element gate. (b) The module automaton of the C-element gate.

inputs and outputs; i.e., $z'^{i}_{j} = F^{i}_{j}(x^{i}_{1}, \ldots, x^{i}_{m^i}, z^{i}_{1}, \ldots, z^{i}_{p^i})$. Then for any $q, q' \in Q^i$, $(q, z^{i}_{j}, q') \in TR^i$ iff $F^{i}_{j}(x^{i}_{1}, \ldots, x^{i}_{m^i}, z^{i}_{1}, \ldots, z^{i}_{p^i})\Big|_{q} = z^{i}_{j}\Big|_{q'}$ and $z^{i}_{j}\Big|_{q'} \neq z^{i}_{j}\Big|_{q}$.

Example 2.2 Figure 2.2.a depicts a C-element gate. The module description of the C-element gate is $M = \langle \{a, b\}, \{c\}, \varnothing, FA \rangle$ where the state diagram of FA, the module automaton, is depicted in Figure 2.2.b. The initial state is entered by an arrow; i.e., $\lambda(q_0) = 000$. ■

Example 2.3 Figure 2.3.a depicts a Mutual-Exclusion (ME) element as described in [27]. The module description of the ME is $M = \langle \{r1, r2\}, \{a1, a2\}, \varnothing, FA \rangle$ where the state diagram of FA, the module automaton, is depicted in Figure 2.3.b. The initial state is entered by an arrow; i.e., $\lambda(q_0) = 0000$. Only non-failure state transitions are shown in Figure 2.3.b. It is to be noted that in an ME element if any input signal has a second transition before the outputs have changed, that would cause a failure state

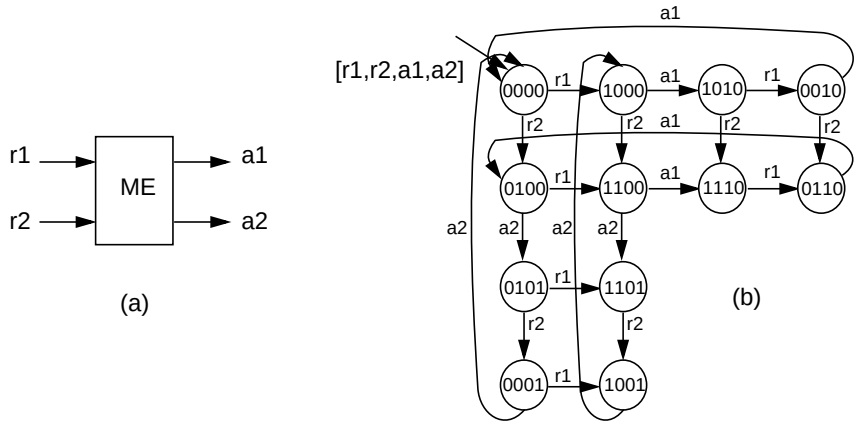


Fig. 2.3 Module description of a Mutual-Exclusion element.
(a) A Mutual-Exclusion element. (b) The module automaton of the ME element.

transition. Thus in Figure 2.3.b the reverse of any state transitions that is associated with an input signal change is a failure state transition (not shown for clarity). ■

Note that although the above Mutual-Exclusion element is a nondeterministic gate, its module automaton is deterministic. As a matter of fact, the module automaton of any circuit module that does not have internal state variables, is always deterministic. On the other hand, the existence of internal state variables can introduce nondeterminism in the module automaton iff there can be (at least) two states, with different internal states, reachable from a single state by the same I/O signal transition.

Example 2.4 Figure 2.4.a depicts a fair arbiter element as described in [48]. The module receives two independent requests to access a single resource, with signals $r1$ and $r2$, and grants access with signals $a1$ and $a2$, respectively (the latter two signals are mutually exclusive). The module description of the arbiter is

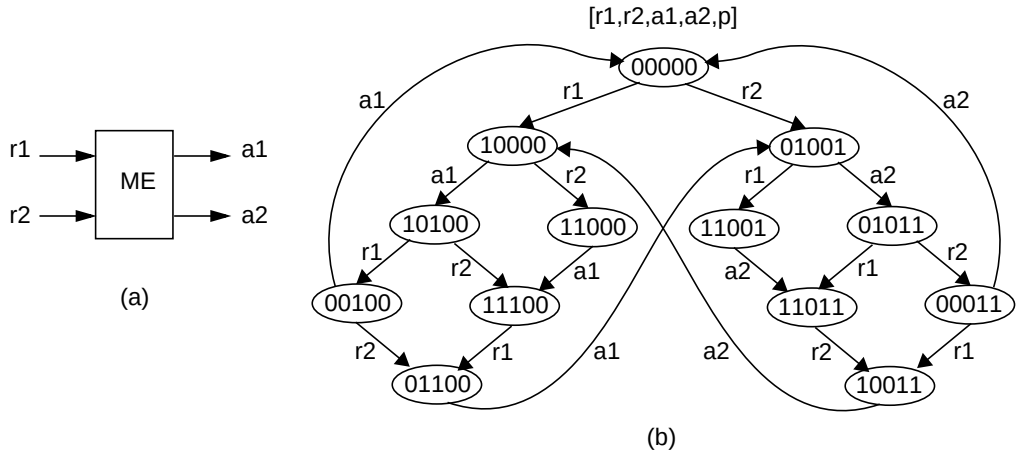


Fig. 2.4 Module description of a fair arbiter element.
(a) A fair arbiter element. (b) The module automaton of the fair arbiter.

$M = \langle \{r1, r2\}, \{a1, a2\}, \{p\}, FA \rangle$ where the state diagram of FA, the module automaton, is depicted in Figure 2.4.b. The initial state is entered by an arrow; i.e., $\lambda(q_0) = 00000$. Only non-failure state transitions are shown in Figure 2.4.b. It is to be noted that in an arbiter element if any input signal has a second transition before the outputs have changed, that would cause a failure state transition. Thus in Figure 2.4.b the reverse of any state transitions that is associated with an input signal change is a failure state transition (not shown for clarity). ■

Note that the above arbiter element is a nondeterministic gate with an internal variable p. However, its module automaton is deterministic. It is called fair because if it receives a request at one input, say $r1$, while it has already received a request at $r2$, it processes the request by $r2$ first, but once it is done with that, it processes the $r1$ request before it can react to a new request from $r2$. Unlike the ME element, the fair arbiter module is capable of distinguishing the order in which two, possibly concur-

rent, requests arrive at its inputs by means of the internal variable, p. As a result, from the initial state, 00000, the two sequence of signal transitions $r1, r2$ and $r2, r1$ lead to different states 11000 and 11001, respectively.

2.2.3 Specifications

We believe that any asynchronous specification with interleaving semantics can be modeled as a circuit module, once some encoding of the internal state of the module is adopted and the failure conditions are all identified.

Signal Transition Graphs (STGs [22, 68]) that are frequently used for specification of asynchronous circuit behavior are Petri-nets in which the Petri-net transitions are interpreted as circuit signal transitions (a complete introduction to Petri-nets can be found at [58]). The state of a Petri-net is completely captured by its *marking*; i.e., the distribution of *tokens* in Petri-net *places*. That is, the token-holding places together with the number of tokens in such places completely specify the internal state of a Petri-net specification. In a *safe* Petri-net (a Petri-net whose places have a capacity of only one token) the Petri-net marking is completely characterized by the token-holding places. Thus, a straight forward way of encoding the internal state of a safe Petri-net specifications would be to assign one internal state variable to each place of the Petri-net. Now, markings of the Petri-net will correspond to binary evaluations of the state variables; that is, for a given marking, a place holds a token iff the value of its associated internal state variable is 1. For simplicity, we consider only *safe* STG spec-

ifications although unsafe STGs can similarly be modeled using multiple variables to represent unsafe places. More efficient encoding schemes for Petri-net markings are proposed in [61].

The *pre-set* of a Petri-net place p, indicated by $\bullet p$, is defined as the set of signal transitions such that the firing of any of such transitions will put a token in that place. Similarly, the *post-set* of a place indicated by $p \bullet$ is the set of signal transitions such that for any of them to fire, a token has to be removed from that place. As an example, in Figure 2.6 we have $\bullet p_0 = \{ua_1\text{-}, ua_2\text{-}\}$ and $p_0 \bullet = \{ua_1+, ua_2+\}$. An *implicit* place p of a Petri-net (or STG) is one which exists between two consecutive signal transitions t and t' of an STG (Petri-net) such that $\bullet p = \{t\}$ and $p \bullet = \{t'\}$. As an example, place p_1 is an implicit place in the Petri-net of Figure 2.6, while p_0 is an *explicit* place. Implicit places of an STG are usually not drawn, as suggested by Figure 2.5.b which illustrates the same STG as that of Figure 2.6.

Example 2.5 The STG specification of a DME ring of length two is illustrated in Figure 2.5.b. This specification is an example of a safe STG. Thus, as already mentioned, the internal state of the specification can be easily encoded by defining one internal state variable per Petri-net place. $M = \langle \{ur_1, ur_2\}, \{ua_1, ua_2\}, \{p_0\}, FA \rangle$ would then define the circuit module of the specification, where $FA = \langle \{ur_1, ua_1, ur_2, ua_2\}, \{p_0, ur_1, ua_1, ur_2, ua_2\}, Q, \lambda, TR, \mu, q_0 \rangle$ is depicted in Figure 2.5.c, and $\lambda(q_0) = 10000$. Any state transition by an *output* signal change that is missing from Figure 2.5.c corresponds to a failure of a circuit implementation,

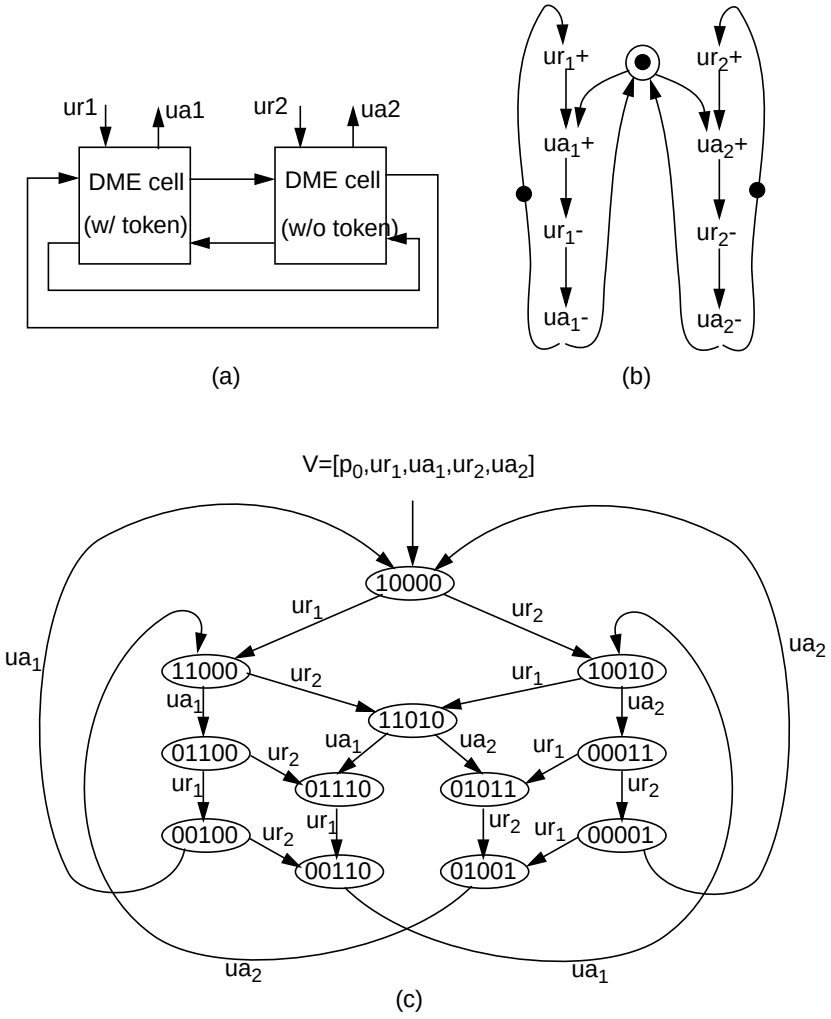


Fig. 2.5 Module description of a DME ring of length two.
(a) The block diagram of a DME ring of length two. (b) The STG specification of the circuit.
(c) The module automaton of the STG specification.

because a circuit implementation of this specification should not generate such output transitions. On the other hand, input transitions that are missing from the automaton correspond to transitions that are never applied to a circuit implementation of this specification; i.e., the specification restricts possible transitions at the inputs of a circuit implementation. ■

The set of internal variables of the module automaton of Figure 2.5.c includes a single variable associated with explicit place p_0; that is, we have defined no state variables associated with the implicit places of this STG specification. This is because the I/O state of this STG happens to uniquely determine the marking of its implicit places, eliminating the need to include the implicit places in the representation of the state of the specification.

Constructing the module automaton from a given Petri-net specification can be a complicated process requiring full traversal of the Petri-net. However, the module automaton can be fully expressed by a collection of transition relations: each such relation would represent the possible (eligible) transitions of an associated output (input) signal of the specification in terms of some portion of the internal state of the specification represented by a subset of Petri-net places. Figure 2.6 depicts the Petri-net specification of Figure 2.5.b with all of its implicit places. The transition relation of signal ua_1 can then be defined as $TR_{ua_1} = \{(011000, ua_1, 100100), (100010, ua_1, 010001)\}$, where the states of this transition relation are evaluations of the following ordered set of variables $[ua_1, p_0, p_1, p_2, p_3, p_4]$.

Representing the automaton of a specification by a collection of transition relations as described above would require one internal state variable associated with each implicit place of the specification. However, such variables can usually be projected away in later phases of hierarchical verification, as will be discussed later in this thesis.

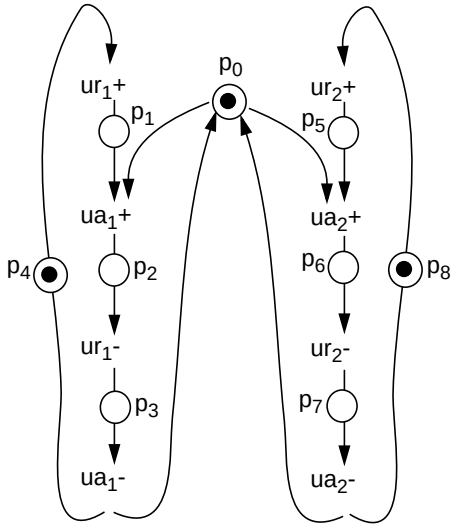


Fig. 2.6 Petri-net specification of A DME ring with all implicit places shown.

2.2.4 Environment Modules: Mirror of Specifications

Checking the *conformance* of a circuit to its specification is a common verification problem. Our notion of conformance follows that of [27]; that is, *safe substitution*. By this, a *circuit implementation* conforms to a *circuit specification* iff the former can be safely substituted for the latter in any context; i.e., the circuit implementation would not generate any output (transition) not specified in the circuit specification. This problem can be solved by checking the failure-freedom of a *closed* circuit composed of the original (open) circuit and the *mirror* of the specification [27]. Conformance checking will be discussed in detail in upcoming sections. In this section, we only define the notions of mirrored specifications and environment modules.

The mirror of a specification is obtained by simply switching the role of the input and output signals of the specification and identifying failure transitions accordingly.

The mirrored specification then comprises an *environment module* for the original circuit; one which interacts with the circuit by providing inputs to the circuit and accepting the circuits outputs. The composition of the original (open) circuit with this derived environment module creates a *closed* circuit. It has been shown that failure-freedom of this closed circuit guarantees the conformance of the original circuit to its specification [27].

Example 2.6 The circuit module for the mirror of the STG specification of Example 2.5 is defined as $M = \langle \{ua_1, ua_2\}, \{ur_1, ur_2\}, \{p_0\}, FA \rangle$, where $FA = \langle \{ur_1, ua_1, ur_2, ua_2\}, \{p_0, ur_1, ua_1, ur_2, ua_2\}, Q, \lambda, TR, \mu, q_0 \rangle$ is depicted in Figure 2.5.c, $\lambda(q_0) = 10000$. Again, any state transition by an input signal change that is missing from Figure 2.5.c corresponds to a failure state transition. Such transitions correspond to unexpected output transitions of a circuit implementation. ■

2.3 Circuit Model

In this section, we introduce our circuit model which we conveniently call a “circuit”. Circuits are composed of circuit modules. We shall only consider *closed* or *autonomous* circuits, with the notion of closed-ness being implicit in our definition of a circuit.

Definition 2.2 [Circuit] A *circuit* is a tuple $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$, where

- $M^C = \{M^1, \ldots, M^{n^C}\}$, $n^C \geq 1$, is a set of circuit modules, where M^i is defined

in Definition 2.1 for $1 \leq i \leq n^C$;

- $A^C = \bigcup_{1 \leq i \leq n^C} Z^i$ is the set of *circuit signals*;
- $V^C = \bigcup_{1 \leq i \leq n^C} Y^i \cup Z^i$ is the set of *circuit variables*;
- $G^C = \langle N^C, K^C \rangle$ is a connected directed graph, the *circuit graph*, where
  - $N^C = \{N^1, \ldots, N^{n^C}\}$, is the set of *circuit nodes*, where circuit node N^i is representative of circuit module M^i in the circuit graph;
  - $K^C \subseteq \bigcup_{1 \leq i \leq n^C} Z^i \times \bigcup_{1 \leq j \leq n^C,\, j \neq i} X^j$ is the set of *circuit edges*, such that for any input signal x_l^i of any circuit module M^i, $1 \leq i \leq n^C$ and $1 \leq l \leq m^i$, there exists exactly one output signal z_k^j of a circuit module M^j, $1 \leq j \leq n^C$, $j \neq i$, and $1 \leq k \leq p^j$, such that $(z_k^j, x_l^i) \in K^C$. In other words, (a) each input signal is connected to (and thus *driven by*) exactly one output signal, and (b) no input of a module is ever connected to an output of that same module;
- $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ is a nondeterministic finite state automaton called the *circuit automaton*, where
  - A^C is the *input alphabet* of the automaton;
  - V^C is the set of *automaton variables* which coincides with the set of circuit variables;
  - Q^C is the *state set* of the automaton;
  - $\lambda^C : Q^C \to L(V^C)$ is the *state labeling function* of the automaton. Here, $L(V^C)$ is the set of all surjective functions $l : V^C \to \{0, 1\}$;
  - $TR^C \subseteq Q^C \times (A^C \cup \varepsilon) \times Q^C$ is the *state transition relation* of the automaton;

- $\mu^C : Q^C \times (A^C \cup \varepsilon) \rightarrow \{\mathrm{F}, \mathrm{S}\}$ is the *transition labeling function* of the automaton;
- $q_0^C \in Q^C$ is the initial state of the automaton. ■

From the definition of a circuit graph it follows that, (a) there is no circuit edges (connections) between output signals of circuit modules, and (b) input signals of any circuit module are connected to output signals of other circuit modules. The first constraint above prohibits wired outputs; the second constraint disallows uncontrolled circuit module inputs, excluding from the set of circuits any non-autonomous collections of circuit modules. Note that dangling circuit module outputs that are not connected to any circuit module inputs are allowable. By this definition, a circuit has to be *closed*.

The second constraint on circuit edges, mentioned above, prohibits connections between inputs and outputs of any given module. This directly follows from the definition of a circuit module in which the set of input and output signal variables must be disjoint. One may wonder about real circuits in which there might be connections between inputs and outputs of a circuit component. As an example, consider a circuit with a 2-input AND gate component whose output drives one of its inputs. In such a case, the generic model for 2-input AND circuit modules cannot be used to model this particular AND gate; instead, a new circuit module, with one less input signal, needs to be devised to model this particular AND gate.

The sole presence of a circuit edge $(z_k^j, x_l^i) \in K$ between any pair of circuit modules M^j and M^i effectively synchronizes the transitions of signal z_k^j in M^j with that

of signal x_l^i in M^i. Thus any transition of the output signal z_k^j of M^j, is instantaneously seen as a transition on the input signal x_l^i of M^i. On the other hand, signal transitions of a circuit module are, in general, accompanied by instantaneous internal state changes. Thus, any transition on z_k^j will cause instantaneous changes in the internal states of M^i, M^j, and any other circuit module M^h for which $(z_k^j, x_m^h) \in K$.

This direct correspondence between any input signal x_l^i of any module M^i of a (closed) circuit, $1 \le i \le n^C$, and the output signal z_k^j of some other circuit module M^j, makes the set $\bigcup_{1 \le i \le n^C} X^i$ (all input signals of all modules of a circuit) an entity which carries only redundant information about the circuit. That is why (a) input signals of the circuit modules M^i, $1 \le i \le n^C$, appear *only* in the circuit graph description of circuit C, (b) the set of circuit signals A^C consists of only *output* signals of component modules (and not both input and output signals of modules), and (c) as a result, the set of circuit variables V^C consists of all circuit module *output* signals and internal state variables, but no module *input* signals.

As a collection of circuit modules connected to each other in the manner described above, the behavior of a circuit is determined by the coordinated behaviors of individual circuit modules. The coordination of individual module behaviors is itself a result of synchronized state transitions of the modules' automata. The circuit automaton FA^C is thus a *composition* of individual module automata $FA^1, \ldots, FA^{n^C}$ as described below.

The first step in composing the individual module automata $FA^1, \ldots, FA^{n^C}$ into the compound automaton FA^C is *variable substitution*. Let $(z_k^j, x_l^i) \in K$ be any circuit

edge indicating that input signal x_l^i is driven by output signal z_k^j. Variable substitution will then replace all occurrences of variable x_l^i in the model description of module M^i with the variable z_k^j. Variable substitution is thus simply a renaming operation.

In the rest of our description of the compound automaton FA^C, it is assumed that variable substitution is already performed.

A^C, the input alphabet of FA^C, coincides with the set of circuit signals. As previously mentioned, any circuit signal $a \in A^C$ is the output of exactly one circuit module and the input of zero or more *other* circuit modules. On the other hand, as a symbol of the alphabet of FA^C, any $a \in A^C$ corresponds to transitions on the associated circuit signal.

$\lambda^C : Q^C \rightarrow L(V^C)$ is an injective function assigning to each state of the automaton FA^C a unique function which in turn assigns binary values to every $v \in V^C$. As a result, each state $q \in Q^C$ is an interpretation of the circuit variables V^C; that is, it assigns to every variable $v \in V^C$ a value in its binary range $\{0, 1\}$. Let $q \in Q^C$ be any state of FA^C and $\lambda^C(q)$ be the label of that state. Moreover, let M^i be any module of the circuit C, and $\lambda^C(q)|V^i$ be the *restriction* of the function $\lambda^C(q)$ to the set of variables of M^i, V^i (note that variable substitution has already replaced each input variable $x \in X^i$ of M^i with some variable $v \in V^C$, and thus $V^i \subseteq V^C$; hence, $\lambda^C(q)|V^i$ is a well-defined function, $\lambda^C(q)|V^i : V^i \rightarrow \{0,1\}$). Then, there always exists a state $q^i \in Q^i$ such that $\lambda^C(q)|V^i = \lambda^i(q^i)$, and q^i is called the *local state* of FA^i associated with state q of FA^C.

Considering the state transition relation $TR^C \subseteq Q^C \times (A^C \cup \varepsilon) \times Q^C$, let $a \in A^C \cup \varepsilon$ be any symbol corresponding to a transition on the associated circuit signal (or an empty signal transition in the case of $a = \varepsilon$), $q, q' \in Q^C$ be any pair of automaton states such that $(q, a, q') \in TR^C$, and M^i be any module of the circuit C. Furthermore, let $q^i, q'^i \in Q^i$ be the local states of FA^i associated with states q, q' of FA^C, respectively, and let $a|V^i$ be defined as follows: $a|V^i = a$ if $a \in A^i$, and $a|V^i = \varepsilon$, otherwise. Finally, let $(q, a, q')|V^i$, the *restriction* of state transition (q, a, q') to V^i, be defined as $(q, a, q')|V^i = (q^i, a|V^i, q'^i)$. Then, $(q, a, q')|V^i$ is always a state transition of TR^i (i.e., $(q^i, a|V^i, q'^i) \in TR^i$) which is called the *local state transition* of FA^i associated with state transition (q, a, q') of FA^C.

The state transition labeling function $\mu^C : Q^C \times (A^C \cup \varepsilon) \rightarrow \{\mathrm{F}, \mathrm{S}\}$ labels the edges of the underlying transition diagram of FA^C (induced by TR^C). Let $a \in A^C \cup \varepsilon$ be any symbol corresponding to a transition on the associated circuit signal (or an empty signal transition in the case of $a = \varepsilon$), $q, q' \in Q^C$ be any pair of automaton states such that $(q, a, q') \in TR^C$. Let $\mu^C(q, a) = \mathrm{F}$ indicate that all state transitions by a from state q are failure state transitions while $\mu^C(q, a) = \mathrm{S}$ indicate that none of such state transitions are failure transitions. Then $\mu^C(q, a) = \mathrm{F}$ iff there exists a module M^i such that at its associated local state $q^i \in Q^i$ (where $\lambda^C(q)|V^i = \lambda^i(q^i)$) we have $\mu^i(q^i, a|V^i) = \mathrm{F}$; that is, all state transitions from state q and by symbol a are labeled as failure transitions iff there exists a module M^i such that from its local state, the symbol a causes failure state transitions. Note that by construction, we always have $\mu^C(q, \varepsilon) = \mathrm{S}$. We say that a circuit C is not *failure-free* iff

there exist $q \in Q^C$ and $a \in A^C$, such that $\mu^C(q, a) = \mathrm{F}$; otherwise, the circuit is failure-free.

A circuit is *non-deterministic* if it has a non-deterministic module which can exhibit a choice within the circuit.

$q_0^C \in Q^C$ is the initial state of automaton FA^C. Let M^i be any circuit module. Then we have $\lambda^i(q_0^i) = \lambda^C(q_0^C)\big|V^i$.

So far, we have described how states and state transitions of the compound automaton FA^C are constrained by states and state transitions of the component automata $FA^1, \ldots, FA^{n^C}$. An inductive description of the state space of FA^C based on those of $FA^1, \ldots, FA^{n^C}$ is given below.

Definition 2.3 [State space of a circuit automaton] Let $C = \langle M^C, A^C, V^C, G^C, FA^C\rangle$ be a circuit. The *state space* of $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C\rangle$ is inductively defined as follows:

(i) $\lambda^C(q_0^C)$, the label of the initial state $q_0^C \in Q^C$ of FA^C is uniquely selected such that for all M^i, $1 \le i \le n^C$, $\lambda^C(q_0^C)$ is an extension of $\lambda^i(q_0^i)$ to V^C.

(ii) Let $q \in Q^C$ be a state of FA^C. Moreover, let (a) the state label of q be $\lambda^C(q)$, (b) $a \in A^C$ be any symbol of the alphabet (associated with signal a of the circuit); by the circuit graph constraints, there must exist a unique module M^i, $1 \le i \le n^C$, such that $a \in Z^i$, (c) $\boldsymbol{M} \subseteq M^C$ be the set of all circuit modules of C such that $M^j \in \boldsymbol{M}$, $1 \le j \le n^C$, iff $a \in X^j$; that is, $\boldsymbol{M}$ is exactly the set of all modules which are driven by the signal a, (d) $q^i \in Q^i$ be the local state of M^i at q; i.e., $\lambda^C(q)|V^i = \lambda^i(q^i)$. Sim-

ilarly, let for any $M^j \in \boldsymbol{M}$, $q^j \in Q^j$ be the local state of M^j at q, (e) a be enabled at q^i and $(q^i, a, q'^i) \in TR^i$ be any of the possible state transitions in FA^i by symbol a. Similarly, let for any $M^j \in \boldsymbol{M}$, $(q^j, a, q'^j) \in TR^j$ be any of the possible state transitions in FA^j by symbol a. Then q' is a state of FA^C (i.e., $q' \in Q^C$) and (q, a, q') is a state transition of FA^C (i.e., $(q, a, q') \in TR^C$) if $\lambda^C(q')$, the label of q', satisfies the following constraints: (a) for M^i and $q'^i \in Q^i$ described above, $\lambda^C(q')|V^i = \lambda^i(q'^i)$, (b) for all $M^j \in \boldsymbol{M}$ and $q'^j \in Q^j$ described above, $\lambda^C(q')|V^j = \lambda^j(q'^j)$, and (c) for all $M^k \in M^C - \boldsymbol{M}$, $1 \le k \le n^C$, $\lambda^C(q')|V^k = \lambda^k(q^k)$. ■

The base part of the inductive definition above describes the initial state of FA^C. Notice that the initial state, and consequently FA^C itself, are well-defined iff the circuit modules are *initial-state-compatible*. That is, let (a) $a \in A^C$ be any circuit signal, (b) $\boldsymbol{M} \subseteq M^C$ be the set of all circuit modules such that $M^j \in \boldsymbol{M}$, $1 \le j \le n^C$, iff $a \in A^j$, (c) $q_0^j \in Q^j$ be the initial state of circuit module $M^j \in \boldsymbol{M}$. Then $\lambda^j(q_0^j)(a)$ has a unique value over all $M^j \in \boldsymbol{M}$.

The inductive step of the inductive definition above describes how Q^C (the set of states of FA^C), λ^C (the state labeling function of FA^C), and TR^C (the state transition function of FA^C) are inductively defined. μ^C (the transition labeling function of FA^C) is defined based on its description that was given earlier.

The automaton FA^C defined as above describes the behavior of the circuit as an *interleaved* behavior. In other words, no two circuit signals change simultaneously, although they may concurrently be enabled to change; instead, all possible interleavings of enabled signals are represented in FA^C. It is also noted that any signal change

will cause a simultaneous and instantaneous change in the internal state of any circuit module which has that signal as an I/O. Such module internal state changes are dictated by the automaton of the corresponding module.

Definition 2.4 [Changed variables of a transition] Let C be a circuit, $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be its automaton, and $(q, a, q') \in TR^C$ be any state transition of FA^C. Let $V \subseteq V^C$ be the set of all and only those circuit variables that change by state transition (q, a, q'); i.e., for all $v \in V$, $\lambda^C(q)(v) \neq \lambda^C(q')(v)$, and for all $w \in V^C - V$, $\lambda^C(q)(w) = \lambda^C(q')(w)$. Then we define $\boldsymbol{Changed}(q, a, q') = V$. Note that if $a \neq \varepsilon$, then $a \in \boldsymbol{Changed}(q, a, q')$. ■

The following recursive procedure for *full reachability analysis* of FA^C is directly derived from the inductive definition of FA^C.

Procedure 2.1 [Full reachability analysis of a circuit automaton]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit. The state space of $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ can be fully *constructed* and *explored* as follows:

(i) The initial state $q_0^C \in Q^C$ of FA^C is *constructed* such that for all M^i, $1 \leq i \leq n^C$, $\lambda^C(q_0^C)$ is an extension of $\lambda^i(q_0^i)$ to V^C.

(ii) Let $q \in Q^C$ be a previously *constructed* state of FA^C which has not been *explored* yet. Then by exploring q, we find all possible state transitions from q, and all states reachable from q through such state transitions. The state exploration at q is per-

formed as follows: for all states q' for which the inductive definition of FA^C would define (q, a, q') as a state transition of FA^C, $q' \in Q^C$ and $(q, a, q') \in TR^C$ are added to the constructed state space of FA^C.

The inductive construction of FA^C is completed when all previously constructed states of FA^C have already been explored; i.e., when the constructed state space reaches a *fixed point*. ■

Failure-freedom of a circuit can be exactly checked during full reachability analysis of the circuit automaton. As FA^C is being constructed, newly explored state transitions of FA^C are checked for failures. If any failure state transition is ever found, then the circuit is known to have a failure and there is no need to continue the construction of FA^C. Otherwise, the construction of FA^C is continued to completion, and the circuit would be declared as failure-free.

Since the size of the state space of a circuit C can be as big as $O(2^{|V^C|})$, checking failure freedom of a circuit through full reachability analysis would often suffer from the *state space explosion* problem; i.e., it can be very costly for large circuits, out of the capacity of even state-of-the-art computers. This is where techniques which enable us to check for failure-freedom without fully exploring the state space, and yet provide exact results become of great importance.

2.4 More on Circuit Automaton and Behavior

In this section, we present the notion of the behavior of a circuit C in terms of the *runs* of its automaton FA^C. We also introduce a set of operations on behaviors and automata which are used in following sections. It is to be noted that we use the kind of automaton which was introduced in the previous section to model any abstract behavior and not just that of a circuit. Thus in what follows, FA^C will characterize any behavior, and not necessarily that of a circuit, unless otherwise specified.

2.4.1 Automaton Behavior and Circuit Behavior

In this section, we define the notion of a *trace*, as a sequence of automaton states, based on which we then define *automaton behavior* and *circuit behavior.* We also define the notion of an automaton *string,* as a sequence of automaton symbols associated with an automaton trace. Finally, we define two functions, $\boldsymbol{Red}(.)$ and $\boldsymbol{FF}(.)$, over traces and behaviors. Function $\boldsymbol{Red}(.)$ keeps only the prefix of a trace (the subset of a behavior) which is necessary for the purpose of checking failure-freedom. Note that checking failure-freedom--the most important property of a circuit and the first to be verified--is completed as soon as any (single) failure is detected. This suggests that the behavior of a circuit beyond any failure point is of no significance; i.e., only those traces of a behavior whose prefixes are failure-free are of any interest for the purpose of verification. Function $\boldsymbol{FF}(.)$ returns the (longest) failure-free prefix of a trace, or the failure-free portion of a behavior.

Definition 2.5 [Trace] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton. A *run* (or *trace*) of the automaton FA^C is a sequence of states $t = q_0 q_1 \ldots q_n$ such that (i) $q_i \in Q^C$ for all $0 \leq i \leq n$, (ii) for any consecutive pair of states q_i, q_{i+1} in the sequence, there exists $a_i \in A^C \cup \varepsilon$ such that $(q_i, a_i, q_{i+1}) \in TR^C$. $\boldsymbol{Len}(t) = n$ is the *length* of such a run[3]. An *initialized run* of the automaton FA^C is a run $t = q_0 q_1 \ldots q_n$ which starts at the initial state of FA^C; that is $q_0 = q_0^C$. ■

Definition 2.6 [Automaton behavior] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton. The *automaton behavior*, denoted B^C, is defined to be the set of all initialized runs of FA^C. Such a set is *prefix-closed*; that is, if $q_0 q_1 \ldots q_n \in B^C$, then $q_0 q_1 \ldots q_i \in B^C$, for $0 \leq i \leq n$. ■

Definition 2.7 [Failure freedom] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton and B^C be its behavior. We say that FA^C (B^C) is not *failure-free* iff there exist $q \in Q^C$ and $a \in A^C$, such that $\mu^C(q, a) = \mathrm{F}$; otherwise, FA^C (B^C) is failure-free. ■

Definition 2.8 [Circuit behavior] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit. The *circuit behavior* is then defined to be the automaton behavior of FA^C, and is thus denoted by B^C. A circuit C is failure-free iff B^C is failure-free. ■

---

3. Note that we define the length of a run as the number of its state transitions, and not the number of its states.

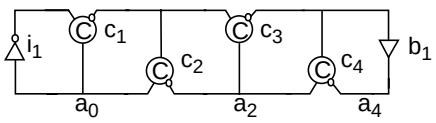


Fig. 2.7 A four-state FIFO controller in an abstract environment.

Example 2.7 Figure 2.7 depicts a four-stage FIFO controller. Two possible traces of the circuit behavior are

$$t_1 = \mathbf{000000}, \mathbf{100000}, \mathbf{110000}, \mathbf{111000}, 011000, 011100,$$

$$t_2 = \mathbf{000000}, \mathbf{100000}, \mathbf{110000}, \mathbf{111000}, 111100, 011100.$$

Here, a state is an evaluation of $[r_0, a_0, a_1, a_2, a_3, a_4]$. The two traces start with a common sequence of state transitions which is shown in bold face. Then, they express two different orderings of transitions of the two signals r_0 and a_2. ■

Definition 2.9 [Sub-behavior] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton and B^C be its behavior. We then call any prefix-closed set of initialized traces $B \subseteq B^C$ a *sub-behavior* of B^C. ■

At this point, we are ready to define the function $\boldsymbol{Red}(.)$ and its operation on traces and behaviors. This function removes from a trace the suffix of it past the first occurrence of a failure.

Definition 2.10 [Reduced trace, prime trace, reduced behavior]

Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, B^C be its behavior, and $t \in B^C$ be any automaton trace with $\boldsymbol{Len}(t) = n$. We then define

$\boldsymbol{Red}(t) = q_0 q_1 \ldots q_m$, $m \leq n$, to be the longest prefix of t such that $\mu^C(q_i, a_i) = S$ for all $0 \leq i < m-1$, where $(q_i, a_i, q_{i+1}) \in TR^C$. In other words, $\boldsymbol{Red}(t)$ is the longest prefix of t with the property that only the last state transition of $\boldsymbol{Red}(t)$ is possibly a failure transition. Trace t is called a *prime trace* iff $\boldsymbol{Red}(t) = t$. We also define $\boldsymbol{Red}(B^C) \subseteq B^C$ as the sub-behavior of B^C consisting of all and only the prime initialized traces of B^C. In other words, $t \in \boldsymbol{Red}(B^C)$ iff $t \in B^C$ and $\boldsymbol{Red}(t) = t$. ■

In the following, we define the function $\boldsymbol{FF}(.)$ and its operation on traces and behaviors. This function returns the longest prefix of a trace which is failure-free.

Definition 2.11 [Failure-free trace and failure-free sub-behavior]

Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, B^C be its behavior, and $t \in B^C$ be any automaton trace with $\boldsymbol{Len}(t) = n$. We then define $\boldsymbol{FF}(t) = q_0 q_1 \ldots q_m$, $m \leq n$, to be the longest prefix of t such that $\mu^C(q_i, a_i) = S$ for all $0 \leq i \leq m-1$, $(q_i, a_i, q_{i+1}) \in TR^C$. In other words, $\boldsymbol{FF}(t)$ is the longest prefix of t which is failure-free. Trace t is then called a *failure-free trace* iff $\boldsymbol{FF}(t) = t$. We also define $\boldsymbol{FF}(B^C) \subseteq B^C$ as the sub-behavior of B^C consisting of all and only the failure-free traces of B^C. In other words, $t \in \boldsymbol{FF}(B^C)$ iff $t \in B^C$ and $\boldsymbol{FF}(t) = t$. ■

Definition 2.12 [String of a trace] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, $t = q_0 q_1 \ldots q_n$ be any trace of the automaton, and $a_\varepsilon^t = a_0 a_1 \ldots a_{n-1}$, $a_i \in A^C \cup \varepsilon$ and $0 \leq i \leq n-1$, be the sequence of symbols (signal transitions) corresponding to trace t; i.e., $(q_i, a_i, q_{i+1}) \in TR^C$. Then the sequence of symbols

obtained from a_{ε}^{t} by removing all ε symbols is called the *string* associated with trace t, and is denoted by a^{t}. ■

Note that for $n = 0$ we define $a_{\varepsilon}^{t} = \varepsilon$. Also, if a_{ε}^{t} is a sequence of ε symbols only, then we define $a^{t} = \varepsilon$.

2.4.2 Projections of Behaviors

In this section, we define a function $\boldsymbol{Proj}(.)(.)$ and describe its operation on states, traces, and behaviors. Note that states are the building blocks of traces and therefore behaviors. On the other hand, each state is identified by an associated set of variables and the unique values assigned to them. Let $\boldsymbol{Proj}(V)(QTB)$ be any instant of the application of function $\boldsymbol{Proj}(.)(.)$ to an object of type state, trace, or behavior. The second argument, QTB, is a state (Q), a trace (T), or a behavior (B), and the first argument, V, is a subset of the variables associated with object QTB. The function maps object QTB to an object of the same type; i.e., it maps states to states, traces to traces, and behaviors to behaviors. The states of the resultant object are associated with the variables in V and their values, and while information regarding the variables in V are preserved, any information regarding the other variables of QTB are lost in the resultant object $\boldsymbol{Proj}(V)(QTB)$.

The function $\boldsymbol{Proj}(.)(.)$ can similarly be applied to strings. Let $\boldsymbol{Proj}(A)(S)$ be any instant of the application of function $\boldsymbol{Proj}(.)(.)$ to an object of type string. The second argument, S, is a string, and the first argument, A, is a subset of the automaton

alphabet A^C. The function maps object S to an object of the same type; i.e., it maps a string to another string, by simply removing any symbol which does not belong to the set A.

Definition 2.13 [W_V-transition] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, $V \subseteq V^C$, and $(q, a, q') \in TR^C$. Then if $V \cap \textbf{\textit{Changed}}(q, a, q') = W$, we say that (q, a, q') is a W_V-transition. ■

Definition 2.14 [V-compatibility, state projection]

Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, and $V \subseteq V^C$. Let $P_V^C \subseteq Q^C \times Q^C$ be a relation such that for any pair of states $q_i, q_j \in Q^C$, $(q_i, q_j) \in P_V^C$ iff $\lambda^C(q_i)|V = \lambda^C(q_j)|V$; that is, the labels of the two states q_i, q_j agree on the values that they assign to the variables in V. We say that any pair of states related by relation P_V^C are *V-compatible*. It is easy to see that P_V^C is an equivalence relation over the set of states Q^C and partitions that set into $p_V^C \geq 1$ equivalence classes, such that each class is associated with a unique function over the set of variables V. We represent the equivalence class of any state $q_i \in Q^C$ with respect to P_V^C with $[q_i]_V$. We are now ready to define (i) a new set of states Q_V^C, $|Q_V^C| = p_V^C$, (ii) a corresponding state labeling function $\lambda_V^C : Q_V^C \rightarrow L(V)$ for Q_V^C, and (iii) a mapping $\textbf{\textit{Proj}}(V) : Q^C \rightarrow Q_V^C$, as follows: for any $q_i \in Q^C$, $\textbf{\textit{Proj}}(V)$ maps all states $q \in [q_i]_V$ to a unique state $q_V \in Q_V^C$ such that $\lambda_V^C(q_V) = \lambda^C(q)|V = \lambda^C(q_i)|V$. Finally, if $\textbf{\textit{Proj}}(V)(q) = q_V$, then we say that q_V is the *projection of q onto V*. ■

So far, we have defined the function $\boldsymbol{Proj}(V) : Q^C \rightarrow Q_V^C$. At this point, we extend our definition of projection over a set of variables $\boldsymbol{Proj}(V)(.)$ to the domain of traces (runs) and behaviors.

Definition 2.15 [Trace projection] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton and B^C be its behavior. Let $V \subseteq V^C$ and $A = V \cap A^C$ (i.e., A consists of all and only those variables of V which belong to A^C). Let $t \in B^C$ be a trace of B^C. Then the *projection of t onto V*, denoted by $t_V = \boldsymbol{Proj}(V)(t)$, will be a sequence of states of Q_V^C, and is inductively defined as follows:

- if $t = q_0^C$, then $t_V = \boldsymbol{Proj}(V)(q_0^C)$;
- if $t = q_0^C q_1 \ldots q_i q_{i+1} = t' q_{i+1}$ and $\boldsymbol{Proj}(V)(t') = t'_V$, then

$$t_V = \begin{cases} t'_V \boldsymbol{Proj}(V)(q_{i+1}), & \boldsymbol{Proj}(V)(q_i) \neq \boldsymbol{Proj}(V)(q_{i+1}) \\ t'_V, & \text{otherwise} \end{cases}$$

In other words, each maximal subsequence of V-compatible states in t is mapped to a single state in t_V which is the projection of just any of the states of the subsequence. ■

Definition 2.16 [Behavior projection] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton and B^C be its behavior. Then for any $V \subseteq V^C$, the *projection of B^C onto V*, denoted by $\boldsymbol{Proj}(V)(B^C)$, is the set of traces such that $t_V \in \boldsymbol{Proj}(V)(B^C)$ iff there exists $t \in B^C$ such that $t_V = \boldsymbol{Proj}(V)(t)$. ■

Definition 2.17 [Exact abstraction of a behavior over a set of variables]

Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, and B^C be its behavior. Let $V \subseteq V^C$, and B be any set of traces over the set of variables V. We say that B is an *exact abstraction of* B^C *over* V iff $B = \boldsymbol{Proj}(V)(B^C)$. (Note that by this definition, $\boldsymbol{Proj}(V)(B^C)$ is itself an exact abstraction of B^C over V!). ■

Definition 2.18 [String projection] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, $V \subseteq A^C$, and a^t be a string of FA^C. Then the *projection of* a^t *onto* V, denoted by $\boldsymbol{Proj}(V)(a^t)$ is the string obtained from a^t by removing any symbol that does not belong to V. ■

Example 2.8 Let FA^C be an automaton with $V^C = \{a, b, c, d, e\}$, and let $B^C = (00001\ 10001\ 10101\ 00101\ 00100\ 00000\ 01000\ 01010\ 00010\ 00011)^*$.

Here, any state of $q \in Q^C$ is labeled with an evaluation of the ordered set $[a, b, c, d, e]$, and $q_0^C = 00001$. We have used regular expressions to simplify the description of the behavior. In addition, we imply that all prefixes of the above regular expression are also in B^C.

Projecting the behavior B^C onto the set $V^C - e = \{a, b, c, d\}$ would yield:

$\boldsymbol{Proj}(V^C - e)(B^C) = (0000\ 1000\ 1010\ 0010\ 0000\ 0100\ 0101\ 0001)^*$. Note that in this projected behavior, two different transitions are possible from state 0000; i.e., $(0000, 1000)$ and $(0000, 0100)$. However, this two transitions occur *only* in an alternate fashion in the projected behavior. This situation has occurred since two *semanti-*

cally different states of B^C (i.e., 00001 and 00000) are projected onto a single state of $\boldsymbol{Proj}(V^C - e)(B^C)$, 0000.

Now, consider projecting B^C onto the set $V^C - a = \{b, c, d, e\}$:

$\boldsymbol{Proj}(V^C - a)(B) = (0001\ 0101\ 0100\ 0000\ 1000\ 1010\ 0010\ 0011)^*$. This time, no two different states of B^C are projected onto a single state of $\boldsymbol{Proj}(V^C - a)(B^C)$. ■

We close this section by two lemmas which describe some useful properties of projections. The lemmas are trivial implications of the definitions of $\boldsymbol{Proj}(.)(.)$, traces and strings, and thus their proofs are omitted.

Lemma 2.1 [Successive projection] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, and $V \subseteq W \subseteq V^C$. Then for any projectable automaton entity e we have $\boldsymbol{Proj}(V)(\boldsymbol{Proj}(W)(e)) = \boldsymbol{Proj}(V)(e)$. ■

Lemma 2.2 [Strings and projections] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, B^C be its behavior and $V \subseteq V^C$. Let $t \in B^C$, a^t be the string associated with t, and $t_V = \boldsymbol{Proj}(V)(t)$. Then $a^{t_V} = \boldsymbol{Proj}(V)(a^t)$. In other words, the string associated with the projection of a trace t is the same as the projection of the string associated with t. ■

2.4.3 Sub-automaton and Projection of an Automaton

In this section, we first define the notion of a *sub-automaton* of an automaton. Then we define the notion of *collapsing* an automaton onto a set of automaton variables fol-

lowed by the notion of an *automaton projection* as any *collapsed automaton* whose behavior is an exact abstraction of the behavior of the original automaton. Finally, we present a set of sufficient conditions for a collapsed automaton to be an automaton projection.

Definition 2.19 [Sub-automaton] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton. We then define a *sub-automaton* of FA^C to be any automaton $\tilde{FA}^C = \langle A^C, V^C, \tilde{Q}^C, \tilde{\lambda}^C, \tilde{TR}^C, \tilde{\mu}^C, q_0^C \rangle$ such that (1) $\tilde{Q}^C \subseteq Q^C$, $q_0^C \in \tilde{Q}^C$, and for all $q \in \tilde{Q}^C$, $\tilde{\lambda}^C(q) = \lambda^C(q)$, (2) $\tilde{TR}^C \subseteq TR^C$, $\tilde{TR}^C \subseteq \tilde{Q}^C \times (A^C \cup \varepsilon) \times \tilde{Q}^C$, and for all $q, q' \in \tilde{Q}^C$ and $a \in A^C \cup \varepsilon$, if $(q, a, q') \in \tilde{TR}^C$ then $\tilde{\mu}^C(q, a) = S$, and (3) the underlying state transition graph of $\tilde{FA}^C$ is a connected subgraph of FA^C. ■

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit. The automaton FA^C, as we have defined, describes the whole state space of the circuit C. A sub-automaton $\tilde{FA}^C$, in contrast, describes the state space of the circuit only partially. A sub-automaton $\tilde{FA}^C$ can thus be constructed by partially exploring the state space of the circuit, instead of full exploration. The inductive construction of FA^C of a circuit C was previously described. In constructing a sub-automaton of FA^C, it suffices to explore only a subset of state transitions from any state which is under exploration, and repeat this procedure from any reached state which is not previously explored, until no such state (reached and unexplored) is left.

Note that while automaton FA^C may have failure transitions, we define any sub-automaton $\tilde{FA}^C$ to be failure-free. A more natural choice for the transition labeling

function of $\tilde{FA}^C$ would seem to be one which carries the labels of the transitions from FA^C to $\tilde{FA}^C$; i.e., one such that for all $q, q' \in \tilde{Q}^C$ and $a \in A^C \cup \varepsilon$, if $(q, a, q') \in \tilde{TR}^C$ then $\tilde{\mu}^C(q, a) = \mu^C(q, a)$. However, as will become clear in the coming sections, the choice of the transition labeling function is not critical or relevant to our analysis, and in fact our simplistic choice is indeed sufficient for the correctness of our framework.

Definition 2.20 [Collapsed automaton]

Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, $V \subseteq V^C$, and $A = V \cap A^C$. The *collapsed automaton of* FA^C *onto* V, denoted by $FA_V^C = \langle A, V, Q_V^C, \lambda_V^C, TR_V^C, \mu_V^C, q_{0V}^C \rangle$ is then defined as follows:

- Q_V^C is the codomain of $\boldsymbol{Proj}(V) : Q^C \to Q_V^C$. Thus $q_V \in Q_V^C$ iff there exists a $q \in Q^C$ such that $q_V = \boldsymbol{Proj}(V)(q)$; in particular, we have $q_{0V}^C = \boldsymbol{Proj}(V)(q_0^C)$.
- $\lambda_V^C : Q_V^C \to L(V)$ is such that for any pair of states q and q_V related by $q_V = \boldsymbol{Proj}(V)(q)$ we have $\lambda_V^C(q_V) = \lambda^C(q)|V$.
- $TR_V^C \subseteq Q_V^C \times (A \cup \varepsilon) \times Q_V^C$ is such that for any $a \in (A \cup \varepsilon)$ and $q_{Vi}, q_{Vj} \in Q_V^C$, $(q_{Vi}, a, q_{Vj}) \in TR_V^C$ iff there exists a pair of states $q_i, q_j \in Q^C$ such that $q_{Vi} = \boldsymbol{Proj}(V)(q_i)$, $q_{Vj} = \boldsymbol{Proj}(V)(q_j)$, and $(q_i, a, q_j) \in TR^C$.
- $\mu_V^C : Q_V^C \times (A \cup \varepsilon) \to \{\mathrm{F}, \mathrm{S}\}$ is such that for all $a \in (A \cup \varepsilon)$ and $q_{Vi}, q_{Vj} \in Q_V^C$ such that $(q_{Vi}, a, q_{Vj}) \in TR_V^C$, $\mu_V^C(q_{Vi}, a) = S$. That is, FA_V^C is defined to be failure-free. ■

Once again, we notice that while automaton FA^C may have failure transitions, we define its collapsed automaton FA_V^C to be failure-free. A more natural choice for the transition labeling function of FA_V^C would seem to be one with the following description:

- $\mu_V^C : Q_V^C \times (A \cup \varepsilon) \rightarrow \{\mathrm{F}, \mathrm{S}\}$ is such that $\mu_V^C(q_{Vi}, a) = \mathrm{F}$ iff there exists $q_i \in Q^C$, such that $q_{Vi} = \boldsymbol{Proj}(V)(q_i)$, and either (i) $\mu^C(q_i, a) = \mathrm{F}$, or (ii) there exists $b \in A^C - V$ such that $\mu^C(q_i, b) = \mathrm{F}$, *and* there exists a sequence of $A^C - V$ signal transitions from q_i, starting with a transition by b and leading to a state at which a is enabled.

However, similar to the case of a sub-automaton, we have made the simplistic choice of letting FA_V^C be failure-free since that would suffice for our analysis.

The definition of a collapsed automaton implies that it can be obtained from the original automaton by the following steps:

(i) take the underlying state diagram of the original automaton and relabel each state by restricting its labeling function to V;

(ii) merge any set of relabeled states that have a common (restricted) label into a single state with that common label. The resulting diagram will thus have states with *unique* labels. (Note that unique state labeling is a requirement of the kind of automaton that we have been using.). The resulting state diagram would represent the collapsed automaton.

Definition 2.21 [Automaton projection]

Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, $V \subseteq V^C$, and $A = V \cap A^C$. We say FA^C is *projectable onto* V and call the collapsed automaton FA_V^C an *automaton projection* iff B_V^C is an exact abstraction of B^C over V; i.e., $B_V^C = \boldsymbol{Proj}(V)(B^C)$. ■

As will be seen in the coming sections of this thesis, in our hierarchical verification approach we frequently need to simplify (reduce) and abstract the model of a behavior. Such abstractions are obtained by *hiding* some subset of the variables of the original behavior. To prevent false negative/positive verification results, the abstract model has to precisely capture the behavior of the non-hidden variables in the original model. In other words, the behavior of the abstract model should be equivalent to the projection of the behavior of the original model onto the same set of variables; i.e., the former should be an *exact abstraction* of the latter.

Although the projection of an automaton behavior is itself an exact abstraction, to obtain it we need to first obtain the behavior of the automaton by full reachability analysis of its underlying state diagram, and then find the projection of each trace of that behavior. In contrast, to collapse an automaton we simply need to appropriately examine the automaton's underlying state diagram, without the need to perform full reachability analysis, and if the collapsed automaton is an automaton projection, its behavior is indeed an exact abstraction. That considered, along with the fact that we already have chosen automaton over trace sets in modeling circuits and their behavior

(due to the more efficient and compact representation of automaton), we would prefer automaton projections over projections of automaton behavior to derive exact abstractions for specifications.

Note that a collapsed automaton is not always necessarily an automaton projection, and thus may not precisely represent the behavior of the non-hidden variables. Consider the two steps of the outlined procedure for collapsing an automaton. It is possible for the resulting collapsed automaton to represent a behavior that is not an exact abstraction, since the second step of collapsing can map semantically different states of the original automaton onto a single state, creating spurious state sequences (and strings) that are not present in the projection of the automaton behavior. Figure 2.8 illustrates this condition through a simple example.

Figure 2.8.a depicts the states diagram of an automaton, with a single state variable v_1. The automaton is to be collapsed onto its alphabet, $A^C = \{a_1, a_2, a_3, a_4\}$. Note that this automaton represents an alternate behavior in which a sequence of transitions on signals a_1 and a_3 alternates with a sequence of transitions on signals a_2 and a_4. Relabeling the states of the automaton results in two states with similar labels (See Figure 2.8.b). Note that the initialized state sequences of state diagram 2.8.b represent the projection of the (alternate) behavior of the original automaton. Merging the two states of diagram 2.8.b into a single state creates the automaton of Figure 2.8.c in which any interleaving of the two above-mentioned sequence of signal transitions are possible. The behavior of the automaton of Figure 2.8.c is a superset of the projection

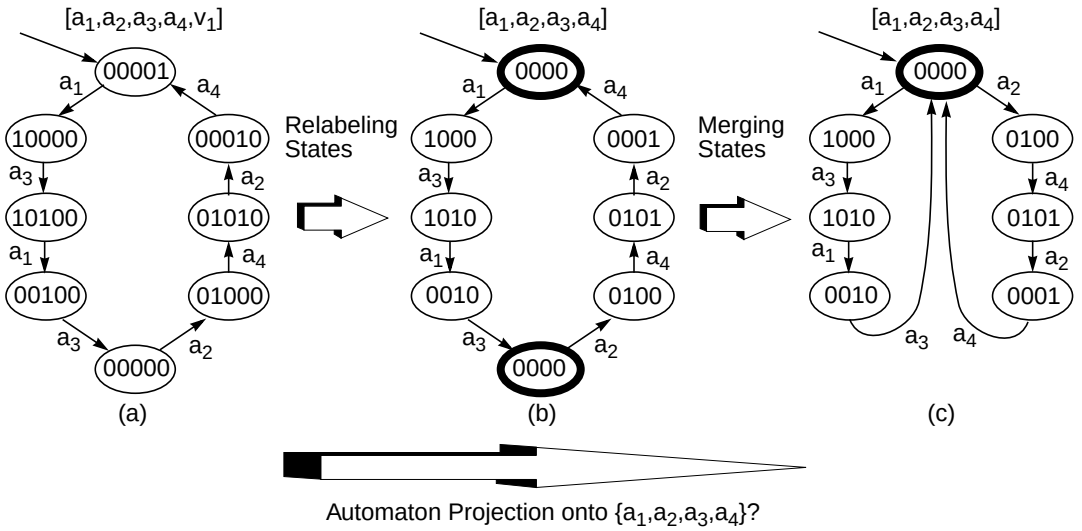


Fig. 2.8 When an automaton projection does not exist!
(a) State diagram of an automaton FA^C with A^C={a_1,a_2,a_3,a_4}, V^C={a_1,a_2,a_3,a_4,v_1}. (b) State diagram with relabeled states, hiding variable v_1. Initialized state sequences of this state diagram represent the projection of the behavior of the original automaton. (c) State diagram with the common label states merged allows state sequences such as 0010,0000,1000 which were not possible in state diagram (b).

of the behavior of the original automaton, and thus is not an exact abstraction of that behavior. In this case, we say that the original automaton *is not projectable* onto the indicated set of variables, or the collapsed automaton is not an automaton projection.

Situations such as the above example force us to impose and practice conditions on the projectability of an automaton to guarantee that the behavior of the collapsed automaton is indeed an exact abstraction of the original automaton's behavior.

Consider automaton $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ and any set $V \subseteq V^C$. We will call V the set of *external* variables, and $V^C - V$ the set of *hidden* variables. We know that the V-compatibility relation $P_V^C \subseteq Q^C \times Q^C$ partitions the set of automaton states into V-compatible equivalence classes. Thus any state $q \in Q^C$

belongs to a V-compatibility class $[q]_V$. The following theorem specifies the necessary and sufficient conditions for projectability of an automaton onto a set of external variables.

Theorem 2.3 [Necessary and sufficient conditions for projectability of an automaton] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, $V \subseteq V^C$ be a set of *external* variables, and $A = V \cap A^C$. The collapsed automaton FA_V^C is then an automaton projection *iff* for any pair of states $q_i, q'_i \in Q^C$ such that $q'_i \notin [q_i]_V$ and $(q_i, a, q'_i) \in TR^C$ is a W_V-transition (i.e., $V \cap \textbf{\textit{Changed}}(q_i, a, q'_i) = W \neq \emptyset$), and for any pair of states $q'_j, q_j \in Q^C$ such that $q'_j \notin [q_i]_V$, $q_j \in [q_i]_V$, and $(q'_j, b, q_j) \in TR^C$, there exists a pair of states $q_l, q'_l \in Q^C$, $q_l \in [q_i]_V$, $q'_l \notin [q_i]_V$, such that $(q_l, c, q'_l) \in TR^C$ is a W_V-transition and there exists a (possibly empty) sequence of V-compatible states from q_j to q_l. ■

The above theorem states that for FA^C to be projectable onto V, for any (external) W_V-transition $(q_i, a, q'_i) \in TR^C$ in FA^C it must be true that if $q_j \in Q^C$ is any state that is V-compatible with q_i and is either an initial state or reachable by an external transition, then there exists a (possibly empty) sequence of V-compatible transitions from q_j to a state q_l, such that a W_V-transition is possible from q_l.

It is straight forward to verify that the following is a reformulations of the above necessary and sufficient conditions for projectability of an automaton.

Conditions 2.22 [Necessary and sufficient conditions for projectability of an automaton] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, $V \subseteq V^C$ be a set of *external* variables, and $A = V \cap A^C$. The collapsed automaton FA_V^C is then an automaton projection *iff* the following conditions hold:

- Let $q_j \in Q^C$ be any initial state of Q^C, or any state to which there exists an external transition $(q'_j, b, q_j) \in TR^C$ from some state $q'_j \in Q^C$ such that $q'_j \notin [q_j]_V$. Let $Q_j \subseteq Q^C$ be the set of all states such that $q_k \in Q_j$ iff

  (i) q_k is reachable from q_j through a (possibly ε) sequence of V-compatible states, and

  (ii) there exists $(q_k, c, q_m) \in TR^C$, $q_m \notin [q_j]_V$; i.e., an external transition from q_k to a state that is not V-compatible with q_k.

  Then let $W_j = \{\boldsymbol{Proj}(V)(q_k, c, q_m) \,|\, (q_k, c, q_m) \in TR^C, q_k \in Q_j, q_m \notin [q_j]_V\}$ be the projection of all external state transitions from the states in Q_j.

- Let $q_l \in Q^C$ be any other initial state of Q^C, or any other state to which there exists an external transition $(q'_l, d, q_l) \in TR^C$ from some state $q'_l \in Q^C$ such that $q'_l \notin [q_j]_V$ and $q_l \in [q_j]_V$; i.e., q_j and q_l are V-compatible. Define Q_l and W_l similar to Q_j and W_j above.

- Then we must have $W_j = W_l$.

If the above conditions hold, then we have $Q_V^C = \{\boldsymbol{Proj}(V)(q_j)\}$ and $TR_V^C = \{W_j\}$, for all states q_j as described above. ■

2.5 Safe Abstractions and Observational Sufficiency

In this section, we first define our notion of a *safe abstraction* as an under approximation of the behavior of a subset of circuit variables which is guaranteed to be exact if the circuit is failure-free. We also define the notion of an *observationally sufficient* set of circuit variables whose behavior can be *safely* captured by a safe abstraction. Finally, we present a corollary suggesting that if the automaton of a circuit is projectable onto a set of circuit variables, then the behavior of the projected automaton is a safe abstraction of the circuit behavior over the same set of circuit variables.

Definition 2.23 [Safe abstraction] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit and B^C be its behavior. Then a behavior B^V over a set of variables $V \subseteq V^C$ is called a *safe abstraction* of B^C over V iff (a) B^V is the behavior of some automaton $FA^V = \langle A, V, Q^V, \lambda^V, TR^V, \mu^V, q_0^V \rangle$, $A = V \cap A^C$, (b) $B^V \subseteq \boldsymbol{Proj}(V)(B^C)$, and (c) $B^V = \boldsymbol{Proj}(V)(B^C)$ if the circuit is failure-free. ■

By definition, a safe abstraction of B^C over V is an automaton behavior which is an under-approximation of the behavior of the circuit variables V and yet it is guaranteed to be exact if the circuit is failure-free[4].

---

4. By the above definition, if any behavior B^V is a safe abstraction of a circuit behavior B^C over a set of circuit variables $V \subseteq V^C$, then B^V must be the behavior of an automaton $FA^V = \langle A, V, Q^V, \lambda^V, TR^V, \mu^V, q_0^V \rangle$, $A = V \cap A^C$. Thus, throughout this thesis, wherever we talk about a safe abstraction, the existence of such a corresponding automaton is automatically assumed.

Definition 2.24 [Observational sufficiency] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit and B^C be its behavior. Then a set $V \subseteq V^C$ is called *observationally sufficient* for B^C iff there exists an automaton $FA^V = \langle A, V, Q^V, \lambda^V, TR^V, \mu^V, q_0^V \rangle$, $A = V \cap A^C$, such that B^V is a *safe abstraction* of B^C over V. ■

By definition, the behavior of any set of observationally sufficient circuit variables is *safely* captured by the corresponding safe abstraction. Here, the word 'safely' is used to emphasize that safe abstractions never over-approximate the behavior of the corresponding variables. We will refer to an observationally sufficient set of variables as an OSV set.

Corollary 2.4 [Automata projections and safe abstractions]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit and B^C be its behavior. Let $V \subseteq V^C$, $A = V \cap A^C$, and $\tilde{FA}^C = \langle A^C, V^C, \tilde{Q}^C, \tilde{\lambda}^C, \tilde{TR}^C, \tilde{\mu}^C, q_0^C \rangle$ be a sub-automaton of FA^C such that $\boldsymbol{Proj}(V)(\tilde{B}^C) = \boldsymbol{Proj}(V)(B^C)$, and $\tilde{FA}^C$ is projectable onto V. Then $\tilde{B}_V^C$ is a safe abstraction of B^C over V. ■

This Corollary directly follows from $\tilde{B}_V^C$ being an exact abstraction of B^C over V.

Corollary 2.5 [Automata projections and safe abstractions]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit and B^C be its behavior. Moreover, let $V \subseteq V^C$, $A = V \cap A^C$, be such that FA_V^C is an automaton projection, and let B_V^C be the behavior of FA_V^C. Then B_V^C is a safe abstraction of B^C over V. ■

This Corollary directly follows from B_V^C being an exact abstraction of B^C over V.

2.6 Formal Proofs

In this section, we present our proofs of Theorems 2.3 and Corollary 2.5 by first introducing a lemma that is used in the proofs.

Lemma 2.6 [Over approximation by collapsed automata]

Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, $V \subseteq V^C$, and $A = V \cap A^C$. Let FA_V^C be the collapsed automaton of FA^C onto V, and B_V^C be its behavior. Then $B_V^C \supseteq \boldsymbol{Proj}(V)(B^C)$. ■

Proof (Sketch) We prove this Lemma by way of contradiction. Suppose $B_V^C \supseteq \boldsymbol{Proj}(V)(B^C)$ is not true. Then, there must exist a trace of shortest length $t_V \in \boldsymbol{Proj}(V)(B^C)$ such that $t_V \notin B_V^C$. Let $t_V = \boldsymbol{Proj}(V)(t)$, where $t = q_0 \ldots q_1 \ldots q_{n-1} \ldots q_n \in B^C$ is any trace whose projection onto V yields t_V and whose last transition is by a variable in V. Here, any indicated pair of states q_j, q_{j+1} of trace t, $0 \leq j \leq n-1$, are V-incompatible states that are separated by a sequence of states that are V-compatible with q_j. Let state $q \in Q^C$ be the state immediately preceding state q_n on trace t. Thus, there exists $a \in A \cup \varepsilon$ such that $(q, a, q_n) \in TR^C$. But then by construction of FA_V^C it follows that $(\boldsymbol{Proj}(V)(q), \boldsymbol{Proj}(V)(a), \boldsymbol{Proj}(V)(q_n)) \in TR_V^C$. However, since q and q_{n-1} are V-compatible, we will have $(\boldsymbol{Proj}(V)(q_{n-1}), \boldsymbol{Proj}(V)(a), \boldsymbol{Proj}(V)(q_n)) \in TR_V^C$. Now, since t_V is the shortest trace of interest, we must have $t'_V = \boldsymbol{Proj}(V)(t') \in B_V^C$, where $t' = q_0 \ldots q_1 \ldots q_{n-1} \in B^C$ is a prefix of trace t. Now, on one hand we have

$t'_V = \boldsymbol{Proj}(V)(t') = \boldsymbol{Proj}(V)(q_0 q_1 \ldots q_{n-1}) \in B_V^C$, and on the other hand we have $(\boldsymbol{Proj}(V)(q_{n-1}), \boldsymbol{Proj}(V)(a), \boldsymbol{Proj}(V)(q_n)) \in TR_V^C$. It then follows that $\boldsymbol{Proj}(V)(q_0 q_1 \ldots q_{n-1} q_n) \in B_V^C$, which in turn implies that $t_V = \boldsymbol{Proj}(V)(t) \in B_V^C$. Since the latter result yields a contradiction, $B_V^C \supseteq \boldsymbol{Proj}(V)(B^C)$ is indeed true. ■

Theorem 2.3: Necessary and sufficient conditions for projectability of an automaton] Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be any automaton, $V \subseteq V^C$ be a set of *external* variables, and $A = V \cap A^C$. The collapsed automaton FA_V^C is then an automaton projection *iff* for any pair of states $q_i, q'_i \in Q^C$ such that $q'_i \notin [q_i]_V$ and $(q_i, a, q'_i) \in TR^C$ is a W_V-transition (i.e., $V \cap \textbf{\textit{Changed}}(q_i, a, q'_i) = W \neq \emptyset$), and for any pair of states $q'_j, q_j \in Q^C$ such that $q'_j \notin [q_i]_V$, $q_j \in [q_i]_V$, and $(q'_j, b, q_j) \in TR^C$, there exists a pair of states $q_l, q'_l \in Q^C$, $q_l \in [q_i]_V$, $q'_l \notin [q_i]_V$, such that $(q_l, c, q'_l) \in TR^C$ is a W_V-transition and there exists a (possibly empty) sequence of V-compatible states from q_j to q_l. ■

Proof (Sketch) FA_V^C is an automaton projection iff $B_V^C = \textbf{\textit{Proj}}(V)(B^C)$. We already know from Lemma 2.6 that $B_V^C \supseteq \textbf{\textit{Proj}}(V)(B^C)$. Thus $B_V^C = \textbf{\textit{Proj}}(V)(B^C)$ would hold iff for all $t_V \in B_V^C$, $t_V \in \textbf{\textit{Proj}}(V)(B^C)$. We will show that the latter condition holds iff the indicated condition of this theorem holds.

First we show that if the condition of this theorem holds, then for all $t_V \in B_V^C$, $t_V \in \textbf{\textit{Proj}}(V)(B^C)$. To prove this by contradiction, suppose that the conditions hold but there exists a shortest trace $t_V = q_0 q_1 \ldots q_{n-1} q_n \in B_V^C$ such that $t_V \notin \textbf{\textit{Proj}}(V)(B^C)$. Considering trace t_V, we must have $(q_{n-1}, a', q_n) \in TR_V^C$, and thus by construction of FA_V^C there must exist two states $q_i, q'_i \in Q^C$ such that $q'_i \notin [q_i]_V$ and $(q_i, a, q'_i) \in TR^C$ is a W_V-transition, and $(q_{n-1}, a', q_n) = (\textbf{\textit{Proj}}(V)(q_i), \textbf{\textit{Proj}}(V)(a), \textbf{\textit{Proj}}(V)(q'_i))$. Since t_V is the shortest such trace, for its immediate prefix we have $t'_V = q_0 q_1 \ldots q_{n-1} \in \textbf{\textit{Proj}}(V)(B^C)$.

Thus, there must exist a trace $t' = q'_0 \ldots q'_1 \ldots q'_{n-1} \in B^C$ such that $t'_V = \textbf{\textit{Proj}}(V)(t')$. Here, any indicated pair of states q'_m, q'_{m+1} of trace t', $0 \leq m < n-1$, are V-incompatible states that are separated by a sequence of states that are V-compatible with q'_m. Thus, $t'_V = \textbf{\textit{Proj}}(V)(q'_0 q'_1 \ldots q'_{n-1})$. Let q'_j be the state immediately preceding state q'_{n-1} on trace t' with $(q'_j, b, q'_{n-1}) \in TR^C$. As noted previously, q'_{n-1} and q'_j are V-incompatible states. Moreover, since $\textbf{\textit{Proj}}(V)(q'_{n-1}) = \textbf{\textit{Proj}}(V)(q_i)$, we have $q'_{n-1} \in [q_i]_V$ and $q'_j \notin [q_i]_V$. Now (with $q_j = q'_{n-1}$) it follows from the conditions of the theorem that there exists a pair of states $q_l, q'_l \in Q^C$, $q_l \in [q_i]_V$, $q'_l \notin [q_i]_V$, such that $(q_l, c, q'_l) \in TR^C$ is a W_V-transition, with a (possibly empty) sequence of V-compatible states from q'_{n-1} to q_l. This is equivalent to saying that there exists a trace $t'' = q'_0 \ldots q'_1 \ldots q'_{n-1} \ldots q_l q'_l \in B^C$, and thus $\textbf{\textit{Proj}}(V)(t'') \in \textbf{\textit{Proj}}(V)(B^C)$. However, note that since $q_l, q'_{n-1} \in [q_i]_V$, and $(q_l, c, q'_l) \in TR^C$ is a W_V-transition, then $q'_l \in [q'_i]_V = [q_n]_V$, and thus $\textbf{\textit{Proj}}(V)(t'') = \textbf{\textit{Proj}}(V)(q'_0 q'_1 \ldots q'_{n-1} q'_l) = q_0 q_1 \ldots q_{n-1} q_n$. But the latter implies that $t_V = q_0 q_1 \ldots q_{n-1} q_n \in \textbf{\textit{Proj}}(V)(B^C)$, which is a contradiction.

Next, we show that if for all $t_V \in B_V^C$, $t_V \in \textbf{\textit{Proj}}(V)(B^C)$, then the condition of the theorem holds. To prove this by contradiction, suppose for all $t_V \in B_V^C$, $t_V \in \textbf{\textit{Proj}}(V)(B^C)$, but there exist a pair of states $q_i, q'_i \in Q^C$ such that $q'_i \notin [q_i]_V$ and $(q_i, a, q'_i) \in TR^C$ is a W_V-transition (i.e., $V \cap \textbf{\textit{Changed}}(q_i, a, q'_i) = W \neq \varnothing$), together with a pair of states $q'_j, q_j \in Q^C$ such that $q'_j \notin [q_i]_V$, $q_j \in [q_i]_V$, and $(q'_j, b, q_j) \in TR^C$, but there *does not* exist any pair of states $q_l, q'_l \in Q^C$, $q_l \in [q_i]_V$,

$q'_l \notin [q_i]_V$, such that $(q_l, c, q'_l) \in TR^C$ is a W_V-transition with a (possibly empty) sequence of V-compatible states from q_j to q_l. Note that since $(q_i, a, q'_i) \in TR^C$ is a W_V-transition, by construction of FA^C_V we have $(\textbf{\textit{Proj}}(V)(q_i), \textbf{\textit{Proj}}(V)(a), \textbf{\textit{Proj}}(V)(q'_i)) \in TR^C_V$. Now let $t' = q'_0 \ldots q'_j q_j \ldots q_l q'_l \in B^C$ be any trace such that $q_l \in [q_i]_V$, $q'_l \notin [q_i]_V$, $(q_l, c, q'_l) \in TR^C$, and q_l is reached from q_j through a sequence of V-compatible states, and let $t'_V = \textbf{\textit{Proj}}(V)(t')$. The last state transition of t'_V would be $(\textbf{\textit{Proj}}(V)(q_l), \textbf{\textit{Proj}}(V)(c), \textbf{\textit{Proj}}(V)(q'_l)) \in TR^C_V$, but that cannot ever be equal to $(\textbf{\textit{Proj}}(V)(q_i), \textbf{\textit{Proj}}(V)(a), \textbf{\textit{Proj}}(V)(q'_i)) \in TR^C_V$, because $(q_l, c, q'_l) \in TR^C$ is *not* a W_V-transition (note that $q_l \in [q_i]_V$). Thus, the last state of t'_V cannot be $\textbf{\textit{Proj}}(V)(q'_i)$, while because of $(\textbf{\textit{Proj}}(V)(q_i), \textbf{\textit{Proj}}(V)(a), \textbf{\textit{Proj}}(V)(q'_i)) \in TR^C_V$, there exists a trace $t''_V \in B^C_V$ whose prefix is same as that of t'_V but its last state is $\textbf{\textit{Proj}}(V)(q'_i)$. It then follows that for such $t''_V \in B^C_V$, $t''_V \notin \textbf{\textit{Proj}}(V)(B^C)$ which is a contradiction. ■

Corollary 2.4: [Automata projections and safe abstractions]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit and B^C be its behavior. Let $V \subseteq V^C$, $A = V \cap A^C$, and $\tilde{FA}^C = \langle A^C, V^C, \tilde{Q}^C, \tilde{\lambda}^C, \tilde{TR}^C, \tilde{\mu}^C, q_0^C \rangle$ be a sub-automaton of FA^C such that $\textbf{\textit{Proj}}(V)(\tilde{B}^C) = \textbf{\textit{Proj}}(V)(B^C)$, and $\tilde{FA}^C$ is projectable onto V. Then $\tilde{B}^C_V$ is a safe abstraction of B^C over V. ■

Proof (Sketch) Since $\tilde{FA}^C_V$ is an automaton projection, we know that $\tilde{B}^C_V = \textbf{\textit{Proj}}(V)(\tilde{B}^C)$. Now, since $\textbf{\textit{Proj}}(V)(\tilde{B}^C) = \textbf{\textit{Proj}}(V)(B^C)$, we will have

$\tilde{B}_V^C = \textbf{\textit{Proj}}(V)(B^C)$; i.e., $\tilde{B}_V^C$ is an exact abstraction of B^C over V. But then $\tilde{B}_V^C$ would also be a safe abstraction of B^C over V. ■

Corollary 2.5: [Automata projections and safe abstractions]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit and B^C be its behavior. Moreover, let $V \subseteq V^C$, $A = V \cap A^C$, be such that FA_V^C is an automaton projection, and let B_V^C be the behavior of FA_V^C. Then B_V^C is a safe abstraction of B^C over V. ■

Proof (Sketch) The proof of this corollary directly follows from Corollary 2.4, by letting $\tilde{FA}^C = FA^C$. ■

Chapter 3

Induced Hierarchical Verification of SI, Theoretical Framework

In the previous chapter, we introduced the notion of a safe abstraction as a behavior over a subset of circuit variables which may under-approximate the actual behavior of those variables, but is guaranteed to be exact if the circuit is failure-free. For a circuit that has a safe abstraction, we introduce in this chapter the notion of *sub-circuits* of the circuit. Such sub-circuits are derived from the safe abstraction and the *circuit blocks*, where circuit blocks are themselves the result of partitioning the circuit using the observationally sufficient variables of the safe abstraction. We prove in a main theorem of this chapter that for circuits which have a safe abstraction, failure-freedom of the circuit can be determined based on the failure-freedom of its *sub-circuits*. This important result is the basis of our framework for induced hierarchical verification of speed-independence, as will be seen in this chapter.

3.1 Partitioning a Circuit into Circuit-Blocks

In this section, we describe the notion of partitioning a circuit into circuit blocks using a selected set of circuit signals.

Definition 3.1 [Circuit block] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit and $E^C \subseteq A^C$ be a non-empty subset of circuit signals which we call *external* signals. We call $H^C = A^C - E^C$ as the set of hidden signals of the circuit. Let $R_E^C \subseteq M^C \times M^C$ be a relation such that for any two circuit modules $M^i, M^j \in M^C$, $(M^i, M^j) \in R_E^C$ iff $A^i \cap A^j \cap H^C \neq \varnothing$. In other words, M^i and M^j are related by R_E^C iff there exists a circuit signal $a \in H^C$ which is a common I/O signal of the two modules. Note that R_E^C is a reflexive and symmetric relation. Let R_E^{*C} be the transitive closure of the relation R_E^C; that is, $R_E^{*C} \supseteq R_E^C$ and for any modules $M^i, M^j, M^k \in M^C$, if $(M^i, M^j) \in R_E^{*C}$ and $(M^j, M^k) \in R_E^{*C}$ then $(M^i, M^k) \in R_E^{*C}$. Since R_E^{*C} is a reflexive, symmetric, and transitive relation, it is an equivalence relation over the set of circuit modules, and partitions that set into $r_E^C \geq 1$ equivalence classes, $M_{E,1}^C, \ldots, M_{E,r_E^C}^C$, each called a *circuit block*. ■

Let $M_{E,i}^C$, $1 \leq i \leq r_E^C$, be any circuit block. We define

- $X_{E,i}^C = \{a \in E^C | \exists\, M^j \in M_{E,i}^C, a \in X^j\}$ as the set of *external inputs* of circuit block $M_{E,i}^C$;
- $Z_{E,i}^C = \{a \in E^C | \exists\, M^j \in M_{E,i}^C, a \in Z^j\}$ as the set of *external outputs* of circuit block $M_{E,i}^C$;
- $Y_{E,i}^C = \{y \in V^C | \exists\, M^j \in M_{E,i}^C, y \in Y^j\}$ as the set of *state variables* of circuit

block $M^C_{E,i}$;

- $H^C_{E,i} = \{a \in H^C | \exists\, M^j \in M^C_{E,i}, a \in A^j\}$ as the set of *hidden (internal) signals* of circuit block $M^C_{E,i}$;
- $A^C_{E,i} = X^C_{E,i} \cup Z^C_{E,i} \cup H^C_{E,i}$ as the set of *signals* of circuit block $M^C_{E,i}$;
- $V^C_{E,i} = A^C_{E,i} \cup Y^C_{E,i}$ as the set of *variables* of circuit block $M^C_{E,i}$;

By definition, the equivalence relation R^{*C}_E which partitions the circuit into circuit blocks is such that for any pair of circuit modules M^i and M^j belonging to two different circuit blocks (i.e., $(M^i, M^j) \notin R^{*C}_E$), if M^j *feeds* M^i through a common I/O signal (i.e., $(z^j_k, x^i_l) \in K^C$) then the common I/O signal must be an external signal (i.e., $z^j_k \in E^C$). In other words, circuit modules which belong to different circuit blocks would never feed each other through internal signals. This further emphasizes the fact that the circuit modules of any circuit block can communicate with the rest of the circuit only through external signal transitions.

It is to be noted that in general, the set of external signals of a circuit block is a subset of the external signals of the circuit; i.e., $X^C_{E,i} \cup Z^C_{E,i} \subseteq E^C$. Thus some signals in E^C may be neither an input nor an output of a circuit block.

Example 3.1 Figure 3.1 shows three different partitions of a four-stage FIFO controller. For Figure 3.1(a), $E_1 = \{r_0, a_0\}$, for Figure 3.1(b), $E_2 = \{a_1, a_2\}$, and for Figure 3.1(c), $E_3 = \{r_0, a_0, a_3, a_4\}$. E_2, for example, partitions the circuit into two blocks, a left block $M^C_{E_2,1}$ and a right block $M^C_{E_2,2}$. Thus we have

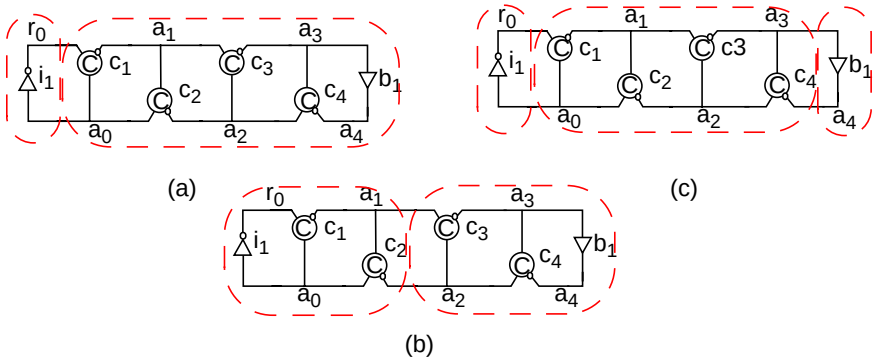


Fig. 3.1 Three different partitions of the four-stage FIFO controller.

$\{i_1, c_1, c_2\} \in M^C_{E_2,1}$, $X^C_{E_2,1} = \{a_2\}$ and $Z^C_{E_2,1} = \{a_1\}$, $Y^C_{E_2,1} = \varnothing$, $H^C_{E_2,1} = \{r_0, a_0\}$, and $A^C_{E_2,1} = \{r_0, a_0, a_1, a_2\}$. Note that a circuit block may not have any hidden signals, as is the case with the left circuit block induced by E_1. ■

3.2 Safe Abstractions and Sub-circuits of a Circuit

In this section, we define our notion of *sub-circuits* of a circuit. This notion is defined only in association with a safe abstraction for the behavior of a circuit over a selected set of its signals. A sub-circuit of such a circuit is the closed circuit composed of a circuit block and its abstract *environment module,* where an environment module of a circuit block is the *mirror* of a *safe specification* of the circuit block, and a safe specification is in turn obtained from the safe abstraction of the circuit. We show in the next section how the failure-freedom of a circuit is related to the failure-freedom of its sub-circuits.

3.3 Environment Module of a Circuit Block

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit and B^C be its behavior. Let $W^C \subseteq V^C$, $E^C = A^C \cap W^C$, and $FA^{W^C} = \langle E^C, W^C, Q^{W^C}, \lambda^{W^C}, TR^{W^C}, \mu^{W^C}, q_0^{W^C} \rangle$ be an automaton whose behavior B^{W^C} is a safe abstraction of B^C over W^C (thus W^C is observationally sufficient for B^C). We call W^C the set of *external variables*, E^C the corresponding set of *external signals*, and $W^C - E^C$ the set of *external state variables*. Let $M^C_{E,1}, \ldots, M^C_{E,r^C_E}$ be the set of circuit blocks of C as it is partitioned by the set of signals E^C. The safe abstraction B^{W^C} which is an approximation of the behavior of the circuit variables W^C, *specifies* for each circuit block $M^C_{E,i}$ how its I/O signals interact with each other and with (possibly) other external circuit variables.

Definition 3.2 [Safe specifications, and safe specification sets]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any circuit for which there exists a behavior B^{W^C} that is a safe abstraction of B^C over some $W^C \subseteq V^C$, and $E^C = A^C \cap W^C$. Let $M^C_{E,i}$ be any circuit block of C induced by E^C. Let $\hat{V}^C_{W,i} \subseteq W^C$ be any set of circuit variables satisfying the following conditions:

- $X^C_{E,i} \subseteq \hat{V}^C_{W,i}$; i.e., $\hat{V}^C_{W,i}$ includes all external inputs of circuit block $M^C_{E,i}$;
- $Z^C_{E,i} \subseteq \hat{V}^C_{W,i}$; i.e., $\hat{V}^C_{W,i}$ includes all external outputs of circuit block $M^C_{E,i}$;
- $FA^{W^C}_{\hat{V}^C_{W,i}}$, the collapsed automaton of FA^{W^C} onto $\hat{V}^C_{W,i}$ is a projection automaton.

We then define a new automaton $\hat{FA}^C_{W,i}$ by applying the following modifications to $FA^{W^C}_{\hat{V}^C_{W,i}}$:

- for any $a \in Z^C_{E,i}$ and $q \in Q^{W^C}_{\hat{V}^C_{W,i}}$, if there exists no $q' \in Q^{W^C}_{\hat{V}^C_{W,i}}$ such that $(q, a, q') \in TR^{W^C}_{\hat{V}^C_{W,i}}$, then (a) add state transition (q, a, q') to $\hat{TR}^C_{W,i}$, where state q' (that needs to be added to $\hat{Q}^C_{W,i}$) is such that $\hat{\lambda}^C_{W,i}(q)(a) = \lambda^{W^C}_{\hat{V}^C_{W,i}}(q)(a) \neq \hat{\lambda}^C_{W,i}(q')(a)$ and for all other $b \in \hat{V}^C_{W,i}, b \neq a$, $\hat{\lambda}^C_{W,i}(q)(b) = \lambda^{W^C}_{\hat{V}^C_{W,i}}(q)(b) = \hat{\lambda}^C_{W,i}(q')(b)$, and (b) let $\hat{\mu}^C_{W,i}(q, a) = \mathrm{F}$.

Then we say that $\hat{B}^C_{W,i}$, the behavior of automaton $\hat{FA}^C_{W,i}$, is a *safe specification* for circuit block $M^C_{E,i}$, derived from the safe abstraction B^{W^C}. We call $X^C_{E,i}$ as the set of inputs of the safe specification, and $Z^C_{E,i}$ as the set of outputs of the safe specification. We also call the (non-empty) set of all possible safe specifications of $M^C_{E,i}$ as the *safe specification set* of $M^C_{E,i}$ and denote it by $\boldsymbol{B}^C_{W,i}$. ■

Note that while automaton $FA^{W^C}_{\hat{V}^C_{W,i}}$ is failure-free, $\hat{FA}^C_{W,i}$ is not, and contains newly introduced failure state transitions. Specifically, the failure state transitions introduced into the safe specification of $M^C_{E,i}$ imply that for $M^C_{E,i}$ to be failure-free, it should not produce any output transitions that are not originally present in safe abstraction B^{W^C}. Note that since FA^{W^C} and hence $FA^{W^C}_{\hat{V}^C_{W,i}}$ do not originally include those failure transitions, and the behaviors of those automata beyond such unexpected failure transitions are not specified, we had to furnish $\hat{FA}^C_{W,i}$ with that information. In doing so, we simply specify the state entered immediately after a failure transition to be one which differs from the preceding state only in the value of the changed output signal. These modifications introduce new traces into the behavior of $\hat{FA}^C_{W,i}$, compared to that of $FA^{W^C}_{\hat{V}^C_{W,i}}$. However, all the newly introduced traces are failure traces,

and their failure-free prefixes are already in the behavior of $FA^{W^C}_{\hat{V}^C_{W,i}}$ (see Lemma 3.8 in the formal proof section of this chapter).

The definition of a safe specification of a circuit block implies that a circuit block can potentially have many safe specifications, as long as the alphabet of their automata satisfies the indicated conditions. It also implies that B^{W^C} is itself a safe specification for any circuit block it induces. Although B^{W^C} can always be used as a safe specification for any circuit block, reducing it to other smaller safe specifications by means of projecting its automaton will often speed up the overall hierarchical verification process.

Example 3.2 Figure 3.2.a depicts a four-stage FIFO controller that is partitioned into two circuit blocks by the set of external signals $E = \{a_1, a_2\}$ (Figure 3.2.b). Figure 3.2.c depicts the state diagram of a safe abstraction of the circuit behavior over E. As indicated in Figure 3.2.d, the safe abstraction is used to derive safe specifications for each of the two circuit blocks. While the signals of the safe abstraction do not have any input/output attribute, an explicit distinction is made between the input and output signals of each of the two safe specifications. In this example, the graph of the automaton of each safe specification contains that of the safe abstraction (here, the safe abstraction, and not a projection of it, is used to derive the safe specifications). However, each safe specification also has additional transitions identifying unspecified output transitions; i.e., any output transition that is not present in the safe abstraction.

■

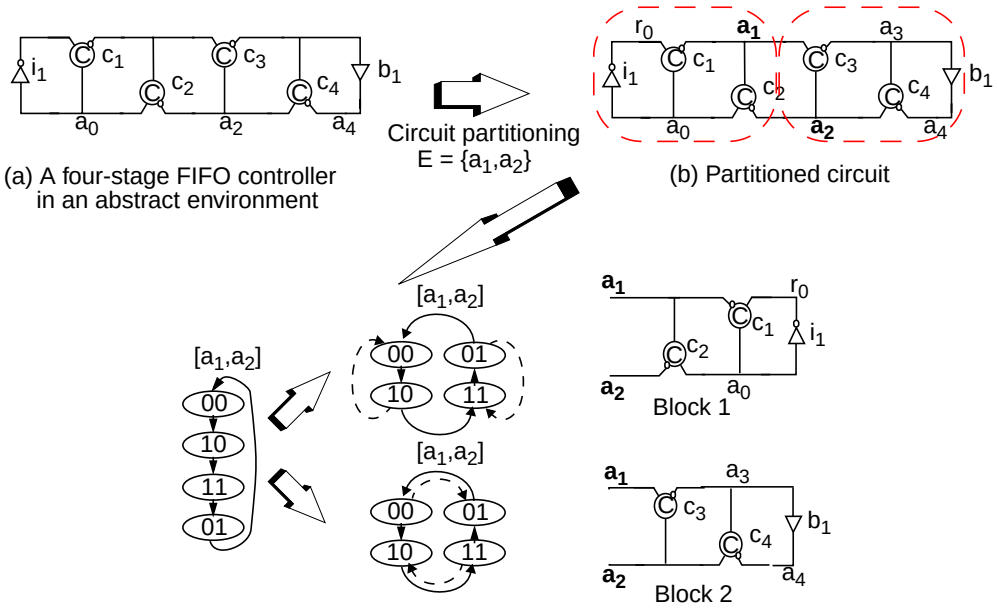


Fig. 3.2 Deriving safe specifications for circuit blocks from a safe abstraction.
Illegal output transitions of the safe specifications are illustrated with dotted arrows.

Definition 3.3 [Environment module] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any circuit for which there exists a behavior B^{W^C} that is a safe abstraction of B^C over some $W^C \subseteq V^C$, and $E^C = A^C \cap W^C$. Let $M^C_{E,i}$ be any circuit block of C induced by E^C, and let $\hat{B}^C_{W,i} \in \boldsymbol{B}^C_{W,i}$ be any safe specification for $M^C_{E,i}$. Finally, let $\hat{A}^C_{W,i} \subseteq \hat{V}^C_{W,i}$ be the subset of $\hat{V}^C_{W,i}$ consisting of circuit signals only and no state variables. Then the *environment module* $\hat{M}^C_{W,i} = \langle \hat{X}^C_{W,i}, \hat{Z}^C_{W,i}, \hat{Y}^C_{W,i}, \hat{FA}^C_{W,i} \rangle$ of $M^C_{E,i}$ corresponding to $\hat{B}^C_{W,i}$ is defined as follows:

- $\hat{X}^C_{W,i} = Z^C_{E,i}$;
- $\hat{Z}^C_{W,i} = \hat{A}^C_{W,i} - \hat{X}^C_{W,i}$;

• $\hat{Y}^{C}_{W,i} = \hat{V}^{C}_{W,i} - \hat{A}^{C}_{W,i}$.

$\hat{M}^{C}_{W,i}$ is in fact a virtual circuit module abstracting the environment of circuit block $M^{C}_{E,i}$. ■

It is easy to verify that (a) the input signals of the environment module are exactly the external output signals of the circuit block, (b) the output signals of the environment module include all the external input signals of the circuit block, and possibly some additional signals from E^{C}, and (c) the state variables of the environment module are a subset of the circuit's external state variables.

Since an environment module of a circuit block is defined based on a safe specification of the circuit block, a circuit block $M^{C}_{E,i}$ may have many possible environment modules each corresponding to a different element of $\boldsymbol{B}^{C}_{W,i}$. The safe specifications of a circuit block (and thus the corresponding environment modules) may differ in terms of the size of their representation (e.g., automaton size) which is generally a monotonically increasing function of the number of automaton variables. In our framework, although the safe specifications of a circuit block are all equivalent in terms of their utility for hierarchical verification, we prefer the ones with smaller representations over others.

The environment module $\hat{M}^{C}_{W,i}$ of $M^{C}_{E,i}$ defined above is in fact the *mirror* of the safe specification $\hat{B}^{C}_{W,i}$ derived from the safe abstraction B^{W^C} [27]. As indicated in the definition of $\hat{M}^{C}_{W,i}$, its set of input signals is exactly the set of output signals of the safe specification; its set of output signals includes the set of input signals of the safe

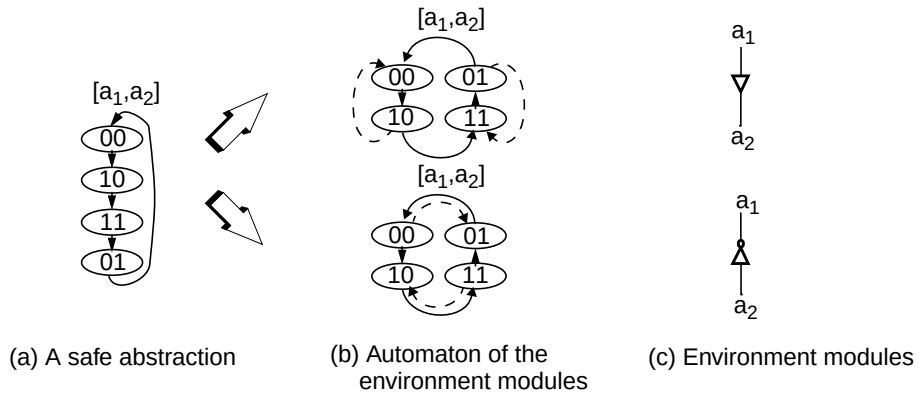


Fig. 3.3 Deriving safe specifications for circuit blocks from a safe abstraction.
Illegal input transitions of the environment modules are illustrated with dotted arrows.

specification; its set of internal state variables consists of all the state variables of the safe specification, and its automaton is the same as the automaton of the safe specification. Thus not only the role of inputs and outputs have changed from the safe specification to the environment module, but also failure state transitions of the safe specification that corresponded to unexpected *output* transitions of the circuit block are mapped to illegal *input* transitions (input chokes) of the environment module. These changes exactly characterize a mirroring procedure.

We need to emphasize that environment modules of circuit blocks of a circuit are defined only given a safe abstraction of circuit behavior over the (observationally sufficient) set of external variables W^C.

Example 3.3 An example of deriving environment modules for circuit blocks of a partitioned circuit from their safe specifications is shown in Figure 3.3. Each environment module is simply the mirror of the corresponding safe specification of

Figure 3.2. Thus, unexpected output transitions of each safe specification are translated to illegal input transitions at the corresponding environment module. In this example, the automaton of the environment module of the above circuit block turns out to be isomorphic to the automaton of a buffer, and that of the bottom circuit block turns out to be isomorphic to the automaton of an inverter. ■

3.4 Subcircuits

In this section, we show how a circuit block together with its environment module create a *sub-circuit* of the original circuit.

Definition 3.4 [Sub-circuit] Let C be a circuit and $W^C \subseteq V^C$, $E^C = A^C \cap W^C$, and $FA^{W^C} = \langle E^C, W^C, Q^{W^C}, \lambda^{W^C}, TR^{W^C}, \mu^{W^C}, q_0^{W^C} \rangle$ be an automaton whose behavior B^{W^C} is a safe abstraction of B^C over W^C (thus W^C is observationally sufficient for B^C). We then call W^C and A^C as the set of *external* variables, and the set of external signals of C, respectively. Let $M_{E,1}^C$, …, $M_{E,r_E^C}^C$ be the set of circuit blocks of C. For any circuit block $M_{E,i}^C$, $\left|M_{E,i}^C\right| = n_{E,i}^C$, and any environment module $\hat{M}_{W,i}^C$ of it, we can devise a *sub-circuit* $C_{W,i}^C = C' = \langle M^{C'}, A^{C'}, V^{C'}, G^{C'}, FA^{C'} \rangle$ as follows:

- $M^{C'} = M_{E,i}^C \cup \hat{M}_{W,i}^C$;
- $A^{C'} = \hat{A}_{W,i}^C \cup A_{E,i}^C$;
- $V^{C'} = A^{C'} \cup \hat{V}_{W,i}^C \cup Y_{E,i}^C$;
- $G^{C'} = \langle N^{C'}, K^{C'} \rangle$ is such that

- $N^{C'} = \{N^1, \ldots, N^{n^C_{E,i}+1}\}$ and N^j is representative of circuit module $M^j \in M^{C'}$;
- $K^{C'} = \{(z^j_k, x^h_l) \in K^C \,\big|\, M^j, M^h \in M^C_{E,i}\} \cup \{(\hat{z}^i_k, x^h_l) \,\big|\, M^h \in M^C_{E,i}, \hat{z}^i_k = x^h_l\} \cup \{(z^j_k, \hat{x}^i_l) \,\big|\, M^j \in M^C_{E,i}, z^j_k = \hat{x}^i_l\}$,

  where signals of $\hat{M}^C_{W,i}$ are identified by a $\hat{}$;
- $FA^{C'} = \langle A^{C'}, V^{C'}, Q^{C'}, \lambda^{C'}, TR^{C'}, \mu^{C'}, q_0^{C'}\rangle$ is the composition of the automata $FA^1, \ldots, FA^{n^C_{E,i}}, \hat{FA}^C_{W,i}$. ■

Thus, informally speaking, sub-circuit $C^C_{W,i}$ is devised by cutting circuit block $M^C_{E,i}$ out of C and connecting it to environment module $\hat{M}^C_{W,i}$ accordingly. We note that (a) since the circuit modules of circuit block $M^C_{E,i}$ also belong to circuit C, they are all initial-state-compatible, and (b) since $\hat{B}^C_{W,i}$ --driven from B^{W^C} by way of projecting its automaton--is a safe specification, the initial state of $\hat{FA}^C_{W,i}$ is compatible with the initial state of C, and therefore with that of all circuit modules in $M^C_{E,i}$. The initial-state-compatibility of all circuit modules of $C^C_{W,i}$ guarantee that the circuit automaton $FA^{C'}$ is well-defined.

Example 3.4 Figure 3.4.a depicts the four-stage FIFO controller of Figure 3.2 that is partitioned into two circuit blocks by the set of external signals $E = \{a_1, a_2\}$. As mentioned in Example 3.3, the environment module of the left circuit block has the automaton of a buffer, while that of the right circuit block has the automaton of an inverter (remember that in deriving those specifications from the safe abstraction, no

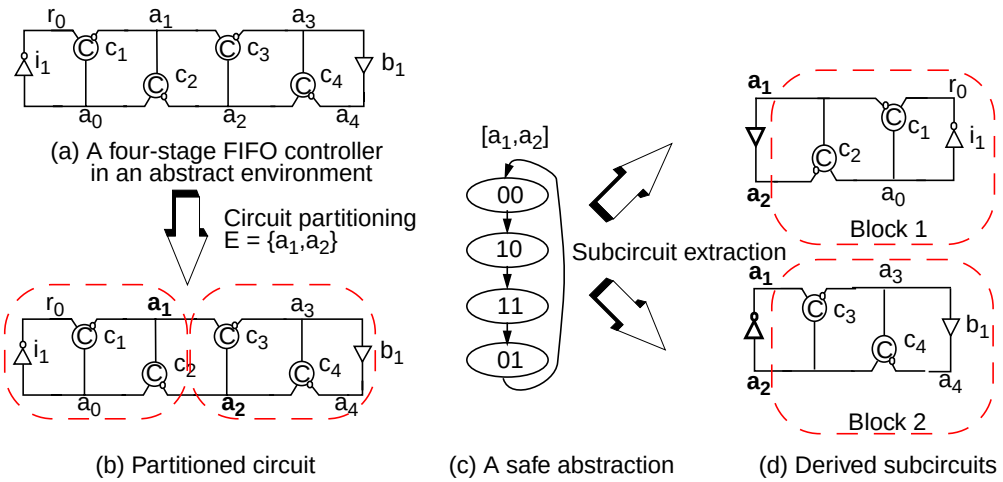


Fig. 3.4 A four-stage FIFO controller and its sub-circuits.

projection was performed). The combination of each circuit block and its environment module has defined a sub-circuit as shown in Figure 3.4.d. ■

We have shown how given a safe abstraction B^{W^C} for the behavior of a circuit C, over a set of observationally sufficient variables W^C, the sub-circuits of the circuit can be constructed. In our hierarchical verification framework, the original circuit is said to be at the 1st level of hierarchy, while its sub-circuits are said to be at the 2nd level of hierarchy. Given a safe abstraction of the behavior of circuit $C^C_{W,i}$ over a corresponding set of signals $W^{C^C_{W,i}}$, the subcircuits of $C^C_{W,i}$ can be similarly constructed. The jth sub-circuit of $C^C_{W,i}$ is thus denoted by $C^{C^C_{W,i}}_{W,j}$. This procedure can be repeated up to any finite level of hierarchy at which the size of a sub-circuit is small enough for the purpose of flat verification. The relationship between the verification of a circuit and that of its sub-circuits is the topic of the following section.

3.5 Circuit Failure-freedom and Sub-circuits' Failure-freedom

In this section, we present a key result which is the basis of our framework for hierarchical verification of speed-independent circuits and systems. We show how the problem of verifying failure-freedom of a circuit can be recursively broken into a collection of smaller problems of verifying the failure-freedom of the sub-circuits of the circuit. Since verification of failure-freedom has computational complexity that is worst-case exponential in the number of circuit variables, such hierarchical approaches which are basically divide and conquer techniques can significantly speed up the verification process.

The two theorems of this section collectively suggest that if there exists a safe abstraction of the behavior of a circuit over a set of external variables, then the circuit is failure-free iff all of its corresponding sub-circuits are failure-free. For the purpose of clarity, we first present each theorem, its implications, and some intuition behind its proof. We then present a more comprehensive sketch of the proofs of the two theorems for the interested reader.

Theorem 3.1 [Circuit versus sub-circuit failure-freedom, I]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any circuit for which there exists a behavior B^{W^C} that is a safe abstraction of B^C over some $W^C \subseteq V^C$, and $E^C = A^C \cap W^C$. Then, if any sub-circuit $C^C_{W,i}$ is not failure-free, then C is not failure-free. ■

The above theorem states that a negative verification result for any sub-circuit of a circuit is always indicative of the failure of the circuit itself. Thus, verifying the fail-

ure-freedom of a circuit by way of verifying its sub-circuits can never generate a false negative result.

The intuition behind the proof of this theorem is as follows. A sub-circuit failure is an illegal input signal transition either at some ordinary circuit module of the corresponding circuit block (e.g. a hazard), or at the environment module (i.e., an input choke to the environment module, or equivalently, an output transition unexpected by the safe specification of the corresponding circuit block). However, (a) any failure at an ordinary circuit module of the sub-circuit is guaranteed to be identically present in the original circuit; this is true since a sub-circuit is actually a circuit block which is operated in an abstract environment that is never an over-approximation of the actual environment of the circuit block, and (b) any input choke to the environment module of the sub-circuit indicates that the safe abstraction, from which the environment module is derived, is an under-approximation; however, by definition of a safe abstraction, this can be true only if the original circuit was not failure-free. Thus, any sub-circuit failure is always indicative of some circuit failure.

Theorem 3.2 [Circuit versus sub-circuit failure-freedom, II]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any circuit for which there exists a behavior B^{W^C} that is a safe abstraction of B^C over some $W^C \subseteq V^C$, and $E^C = A^C \cap W^C$. If all sub-circuits $C^C_{W,1}, \ldots, C^C_{W, r^C_E}$ are failure-free, then C, itself, is failure-free. ■

The above theorem states that positive verification results for all sub-circuits is always indicative of the failure-freedom of the circuit itself. Thus, verifying the fail-

ure-freedom of a circuit by way of verifying its sub-circuits can never generate a false positive result.

The intuition behind the proof of this theorem is as follows. A circuit failure is an illegal signal transition at the input of some circuit module (a driven module), generated by another circuit module (a driving module). This failing signal is either an external signal or an internal signal of the circuit. If the failing signal is external, then either its failing transition is captured in the safe abstraction or it is not. If a failing external signal transition is captured in the safe abstraction, then an identical failure must have manifested itself in the sub-circuit containing the driven module. If a failing external signal transition is *not* captured in the safe abstraction, then the under-approximated behavior of the driving module would have manifested itself as a choke to the environment module of the sub-circuit containing the driving module. Thus, any circuit failure on an external signal is guaranteed to be captured as a failure in some sub-circuit. On the other hand, if the failing signal is an internal circuit signal, then an identical failure would have manifested itself in the sub-circuit containing the driven (and the driving) circuit module, if the specification of the corresponding circuit block is exact; thus, any circuit failure on an internal signal is also guaranteed to be captured as a failure in some sub-circuit. Hence, if all sub-circuits are verified as failure-free, then the circuit must have been failure-free itself.

Before we present our proofs of Theorems 3.1 and 3.2, we would like to further signify the dual role of external variables in our verification framework; i.e., (a) being

the set of variables whose behavior is approximated by a safe abstraction, and (b) containing the set of external signals that partition the circuit into circuit blocks.

As indicated by the two theorems of this section, for any circuit which has a safe abstraction over a set of external circuit variables, there exists a particular relationship between the failure-freedom of the circuit and that of its induced sub-circuits. We are specifically interested in this particular relationship because it is the foundation of our hierarchical verification framework. Here, we would like to show that the relationship of our interest do not generally exist if the set of circuit blocks were *arbitrary.*

We define an *arbitrary circuit block* as any subset of circuit modules. We also define an *arbitrary set of circuit blocks* to be any set of arbitrary circuit blocks. The *input* and *output* signals of the circuit blocks of an arbitrary set are defined as follows: any signal that is driven by a circuit module in one circuit block and drives a circuit module in another circuit block is an output of the first circuit block and an input of the second circuit block.

Consider Figure 3.5 which depicts two *overlapping* arbitrary circuit blocks CB_1 and CB_2. Assume that signal a is driven by the common portion of CB_1 and CB_2, and drives modules in each of CB_1 and CB_2. Our definition of *input* signals, given above, would not label a as an input signal of either of CB_1 or CB_2, since a is actually driven from within both circuit blocks. On the other hand, for a to be labeled as an *output* signal of either of the two circuit blocks, a has to be an input *signal* of a third circuit block; in such a case, a would be an output of both CB_1 and CB_2.

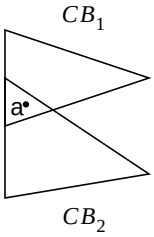


Fig. 3.5 Two overlapping arbitrary circuit blocks

Next, we describe a set of necessary conditions that an arbitrary set of circuit blocks has to satisfy before their failure-freedom can have any significant relationship to that of the circuit.

(i) An arbitrary set of circuit blocks must be a covering set for the circuit modules; i.e., each circuit module must belong to at least one arbitrary circuit block. This constraint is to guarantee that verification of a circuit by means of verifying its sub-circuits is inclusive and there is no circuit module which is not verified within any sub-circuit.

(ii) Input signals of any arbitrary circuit block must all be external. This constraint is to guarantee that the environment module of the circuit block which is obtained from the safe abstraction--and thus lacks direct information regarding the behavior of internal signals--can appropriately drive all inputs of the circuit block. Since input signals of a circuit block are output signals of other circuit blocks, this constraint also implies that output signals of any arbitrary circuit block must all be external.

(iii) If two circuit blocks overlap, then any signal which is driven by a module common to the two circuit blocks has to be external. This constraint is to avoid a particular problem that is illustrated in Figure 3.6. Figure 3.6 depicts two overlapping arbitrary

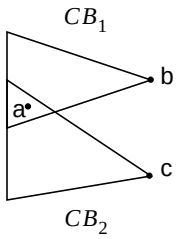


Fig. 3.6 Overlapping arbitrary blocks with a non-external common signal.

circuit blocks CB_1 and CB_2. Assume that b and c are (external) output signals of CB_1 and CB_2, respectively, and a is an internal signal driven by a circuit module in the common portion of the two circuit blocks. Moreover, assume that the only sequence of transitions that can possibly occur on the signals a, b, and c in the original circuit is $c+, a+, b+, a-, b-, c-$, such that $c+$ is required for $a+$, $a+$ is required for $b+$, and $b+$ is required for $a-$. Assume that the lower level circuit is failure-free and that there exists a safe abstraction over the set of its external signals (note that b and c belong to the set of external signals, but not a). Such a safe abstraction would have the sequence of transitions $c+, b+, b-, c-$. This new sequence lacks any information about the relative order of transitions on signal a with respect to those of signals b and c. As an example, this sequence suggests that $\hat{CB}_2$ (the environment module of CB_2) can produce a $b+$ transition right after a $c+$ transition is produced by CB_2. Thus, in the sub-circuit which is the composition of CB_2 and $\hat{CB}_2$, a $c+$ would enable not only $a+$ (through CB_2) but also $b+$ (through $\hat{CB}_2$); however if $b+$ occurs before $a+$, it would enable $a-$ (through CB_2), which is equivalent to disabling $a+$

which was already enabled by $c+$. In other words, in the sub-circuit associated with CB_2, signal a can become enabled and then disabled without having a chance to fire. This situation will be detected as a failure in that sub-circuit, while the original circuit was in fact failure-free. In such a case, taking the sub-circuit failure as an indication of a circuit failure would generate nothing but a false negative verification result.

In general, overlapping pairs of arbitrary circuit blocks that have non-external common signals do not always satisfy the particular conditions of the example of Figure 3.6 which led to false negative verification results. However, by disallowing non-external common signals all together, the possibility of generating such false negative verification results is removed.

It can easily be seen that any set of circuit blocks created by *partitioning* a circuit by a set of external signals happens to satisfy our indicated set of necessary conditions. As proved next, such circuit blocks, together with the safe abstraction over the set of external variables, define sub-circuits whose failure properties do in fact relate to that of the circuit in the ways suggested by Theorems 3.1 and 3.2.

3.6 Formal Proofs

We present our proofs of Theorems 3.1 and 3.2 by first introducing some lemmas and corollaries which are used in the proofs.

Lemma 3.3 [Projection of safe specifications] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any circuit for which there exists a behavior B^{W^C} that is a safe abstraction of B^C over

some $W^C \subseteq V^C$, and $E^C = A^C \cap W^C$. Let $M^C_{E,i}$ be any circuit block of C induced by E^C, and $\hat{B}^C_{W,i}$ be a safe specification of $M^C_{E,i}$. Then $\boldsymbol{Proj}(\hat{V}^C_{W,i})(B^{W^C}) \subseteq \hat{B}^C_{W,i}$ and $\boldsymbol{FF}(\hat{B}^C_{W,i}) = \boldsymbol{Proj}(\hat{V}^C_{W,i})(B^{W^C})$. ■

Proof (Sketch) $\hat{FA}^C_{W,i}$--the automaton of $\hat{B}^C_{W,i}$--is obtained from $FA^{W^C}_{\hat{V}^C_{W,i}}$--the projection of automaton FA^{W^C} onto $\hat{V}^C_{W,i}$--by solely introducing new failure state transitions, which in turn introduce new failure traces into behavior $\hat{B}^C_{W,i}$. Thus, we have $B^{W^C}_{\hat{V}^C_{W,i}} \subseteq \hat{B}^C_{W,i}$ and $\boldsymbol{FF}(\hat{B}^C_{W,i}) = B^{W^C}_{\hat{V}^C_{W,i}}$. On the other hand, by definition of an automaton projection we know that $B^{W^C}_{\hat{V}^C_{W,i}} = \boldsymbol{Proj}(\hat{V}^C_{W,i})(B^{W^C})$. It then follows that $\boldsymbol{FF}(\hat{B}^C_{W,i}) = \boldsymbol{Proj}(\hat{V}^C_{W,i})(B^{W^C})$. ■

Lemma 3.4 [Under approximation of the I/O behavior of a circuit block] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any circuit for which there exists a behavior B^{W^C} that is a safe abstraction of B^C over some $W^C \subseteq V^C$, and $E^C = A^C \cap W^C$. Let $M^C_{E,i}$ be a circuit block of C, and $\hat{M}^C_{W,i}$ be its environment module. Then $\boldsymbol{Proj}(\hat{A}^C_{W,i})(B^{W^C}) \subseteq \boldsymbol{Proj}(\hat{A}^C_{W,i})(B^C)$. ■

Proof (Sketch) Since B^{W^C} is a safe abstraction of B^C over $W^C \subseteq V^C$, we have

$$B^{W^C} \subseteq \boldsymbol{Proj}(W^C)(B^C), \tag{1}$$

and by applying function $\boldsymbol{Proj}(.)(.)$ to both sides of relation (1) we have

$$\boldsymbol{Proj}(\hat{A}^C_{W,i})(B^{W^C}) \subseteq \boldsymbol{Proj}(\hat{A}^C_{W,i})(\boldsymbol{Proj}(W^C)(B^C)). \tag{2}$$

However, by Lemma 2.1 we have

$$\boldsymbol{Proj}(\hat{A}^C_{W,i})(\boldsymbol{Proj}(W^C)(B^C)) = \boldsymbol{Proj}(\hat{A}^C_{W,i})(B^C). \tag{3}$$

From (2) and (3) we conclude that

$$\boldsymbol{Proj}(\hat{A}^{C}_{W,i})(B^{W^C}) \subseteq \boldsymbol{Proj}(\hat{A}^{C}_{W,i})(B^{C})\,. \blacksquare \qquad (4)$$

Lemma 3.5 [Properties of traces captured in a safe specification] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any circuit for which there exists a behavior B^{W^C} that is a safe abstraction of B^C over some $W^C \subseteq V^C$, and $E^C = A^C \cap W^C$. Let $C' = C^{C}_{W,i}$ be any sub-circuit of C, and $t \in B^C$ be any trace for which there exists $t^{\hat{V}^{C}_{W,i}} \in \hat{B}^{C}_{W,i}$ such that $\boldsymbol{Proj}(\hat{V}^{C}_{W,i})(t) = t^{\hat{V}^{C}_{W,i}}$. Then $\boldsymbol{Proj}(V^{C'})(t) \in B^{C'}$. Moreover, if $t' \in B^{C'}$ is any trace such that $\boldsymbol{Proj}(\hat{V}^{C}_{W,i})(t') = t^{\hat{V}^{C}_{W,i}}$, then $t' \in \boldsymbol{Proj}(V^{C'})(B^C)$.

Informally speaking, Lemma 3.5 states that if a circuit trace t is successfully abstracted within the safe specification of circuit block $M^{C}_{E,i}$ (i.e. by trace $t^{\hat{V}^{C}_{W,i}}$), then not only (the projection of) trace t will be (locally) present in sub-circuit $C' = C^{C}_{W,i}$, but also any other trace t' of sub-circuit C' that adheres to trace $t^{\hat{V}^{C}_{W,i}}$ --and thus to t-- will be (globally) present in circuit C. ■

Proof (Sketch) We know that the I/O signals of circuit block $M^{C}_{E,i}$ (corresponding to sub-circuit $C^{C}_{W,i}$) via which $M^{C}_{E,i}$ interacts with its actual environment (i.e., the rest of the circuit) are all external signals; that is,

$$X^{C}_{E,i} \cup Z^{C}_{E,i} \subseteq E^C\,, \qquad (5)$$

where $E^C \subseteq W^C$ is the set of external circuit signals. We also know that $\hat{A}^{C}_{W,i}$, the I/O signals of environment module $\hat{M}^{C}_{W,i}$ via which $\hat{M}^{C}_{W,i}$ interacts with circuit block $M^{C}_{E,i}$, are all external signals, and in particular

$$(X^C_{E,i} \cup Z^C_{E,i}) \subseteq \hat{A}^C_{W,i} \subseteq E^C \subseteq W^C. \quad (6)$$

Since $\hat{A}^C_{W,i} \subseteq \hat{V}^C_{W,i} \subseteq W^C$, from (6) have

$$(X^C_{E,i} \cup Z^C_{E,i}) \subseteq \hat{A}^C_{W,i} \subseteq \hat{V}_{W,i} \subseteq W^C. \quad (7)$$

From (7) and $\boldsymbol{Proj}(\hat{V}^C_{W,i})(t) = t^{\hat{V}^C_{W,i}}$, and by using Lemma 2.1 we have

$$\boldsymbol{Proj}(X^C_{E,i} \cup Z^C_{E,i})(t) = \boldsymbol{Proj}(X^C_{E,i} \cup Z^C_{E,i})(t^{\hat{V}^C_{W,i}}) \quad (8)$$

Let a^t and a^{tv} be the strings associated with traces $t \in B^C$ and $t^{\hat{V}^C_{W,i}} \in \hat{B}^C_{W,i}$, respectively. Then from (8) we have

$$\boldsymbol{Proj}(X^C_{E,i} \cup Z^C_{E,i})(a^t) = \boldsymbol{Proj}(X^C_{E,i} \cup Z^C_{E,i})(a^{tv}). \quad (9)$$

In equation (9), $\boldsymbol{Proj}(X^C_{E,i} \cup Z^C_{E,i})(a^t)$ corresponds to a sequence of transitions on the I/O signals of circuit block $M^C_{E,i}$ when it is operating within its actual environment, circuit C; equivalently, $\boldsymbol{Proj}(X^C_{E,i} \cup Z^C_{E,i})(a^t)$ is a sequence of I/O signal transitions of the actual environment of circuit block $M^C_{E,i}$. On the other hand $\boldsymbol{Proj}(X^C_{E,i} \cup Z^C_{E,i})(a^{tv})$ corresponds to a sequence of I/O signal transitions of the safe specification of circuit block $M^C_{E,i}$ -- $\hat{B}^C_{W,i}$; equivalently, $\boldsymbol{Proj}(X^C_{E,i} \cup Z^C_{E,i})(a^{tv})$ is a sequence of I/O signal transitions of the abstract environment of circuit block $M^C_{E,i}$ -- $\hat{M}^C_{W,i}$.

By a similar argument, if $t' \in B^{C'}$ is such that $\boldsymbol{Proj}(\hat{V}^C_{W,i})(t') = t^{\hat{V}^C_{W,i}}$ then we have

$$\boldsymbol{Proj}(X^C_{E,i} \cup Z^C_{E,i})(t') = \boldsymbol{Proj}(X^C_{E,i} \cup Z^C_{E,i})(t^{\hat{V}^C_{W,i}}), \quad (10)$$

and if we let $a^{t'}$ be the string associated with trace $t' \in B^{C'}$ then from (10) we have

$$\boldsymbol{Proj}(X^C_{E,i} \cup Z^C_{E,i})(a^{t'}) = \boldsymbol{Proj}(X^C_{E,i} \cup Z^C_{E,i})(a^{tv}). \quad (11)$$

In equation (11), $\boldsymbol{Proj}(X^{C}_{E,i} \cup Z^{C}_{E,i})(a^{t'})$ corresponds to a sequence of transitions on the I/O signals of circuit block $M^{C}_{E,i}$ when it is operating within its abstract environment, $\hat{M}^{C}_{W,i}$; equivalently, $\boldsymbol{Proj}(X^{C}_{E,i} \cup Z^{C}_{E,i})(a^{t'})$ is a sequence of I/O signal transitions of the abstract environment of circuit block $M^{C}_{E,i}$, as also confirmed by the right side of equation (11).

From equations (9) and (11) we conclude that

$$\boldsymbol{Proj}(X^{C}_{E,i} \cup Z^{C}_{E,i})(a^{t'}) \;=\; \boldsymbol{Proj}(X^{C}_{E,i} \cup Z^{C}_{E,i})(a^{t}). \qquad (12)$$

Now, consider circuit block $M^{C}_{E,i}$ interacting via its--all external--I/O signals with (a) its actual environment, and (b) its abstract environment $\hat{M}^{C}_{W,i}$ whose automaton behavior is $\hat{B}^{C}_{W,i}$. Equations (12) suggest that circuit block $M^{C}_{E,i}$ can experience, the same sequence of I/O signal transitions within both environments.

Naturally then, the original environment of $M^{C}_{E,i}$ is not distinguishable from the abstract environment of $M^{C}_{E,i}$ when their sequence of interactions with $M^{C}_{E,i}$ adhere to t (as well as t'). On the other hand, the behavior of any circuit block (i.e., its set of all possible traces) is inherently unique per any unique sequence of I/O interactions. Intuitively, it then follows that

- (i) the same sequence of transitions of $V^{C'}$ (the collection of variables of $M^{C}_{E,i}$ and $\hat{M}^{C}_{W,i}$) along trace t of the original circuit must also be observable in C' (the sub-circuit composed of $M^{C}_{E,i}$ and $\hat{M}^{C}_{W,i}$). In other words, we must have $\boldsymbol{Proj}(V^{C'})(t) \in B^{C'}$;
- (ii) the same sequence of transitions of $V^{C'}$ along trace t' of sub-circuit C' must

also be observable in C. In other words, we must have $t' \in \boldsymbol{Proj}(V^{C'})(B^C)$.

Claim (i) above (and similarly, claim (ii)) can be proven by an induction on the length of trace t (and that of t') and the enabling conditions of circuit modules of $M^C_{E,i}$. The inductive proofs of the two claims are very similar. However, for the sake of completeness, we present both of them in what follows.

- (i) Consider the original circuit C and its sub-circuit C'. Let $t \in B^C$ be any circuit trace. The circuit modules of $M^C_{E,i}$ have a unique initial state in both C and C'; that is, the two circuits are initial-state-compatible. Now, since the initial state of any circuit uniquely defines the trace of that circuit which has a length of one, for the base case of $\boldsymbol{Len}(t) = 1$ we have $\boldsymbol{Proj}(V^{C'})(t) \in B^{C'}$. Now assume that $\boldsymbol{Proj}(V^{C'})(t_n) \in B^{C'}$ holds for any trace $t_n \in B^C$ of length n for which $\boldsymbol{Proj}(\hat{V}^C_{W,i})(t_n) = t_n^{\hat{V}^C_{W,i}} \in \hat{B}^C_{W,i}$. (Note that trace $t_n^{\hat{V}^C_{W,i}}$, corresponding to trace t_n, is not necessarily of length n, and the subscript is only to emphasize the correspondence.). We show that any trace $t_{n+1} \in B^C$ of length $n+1$, such that t_n is the prefix of t_{n+1} and $\boldsymbol{Proj}(\hat{V}^C_{W,i})(t_{n+1}) = t_{n+1}^{\hat{V}^C_{W,i}} \in \hat{B}^C_{W,i}$ will satisfy the condition $\boldsymbol{Proj}(V^{C'})(t_{n+1}) \in B^{C'}$. To see this, if the last state transition of t_{n+1} involves no variables of C', then obviously $\boldsymbol{Proj}(V^{C'})(t_{n+1}) = \boldsymbol{Proj}(V^{C'})(t_n) \in B^{C'}$. Otherwise, any variable of C' involved in the last state transition of t_{n+1} is either driven by a module in $M^C_{E,i}$ or by one outside $M^C_{E,i}$. First consider the case in which $M^C_{E,i}$ drives a changing variable of the last state transition of t_{n+1}: since the modules of $M^C_{E,i}$ have experienced the same set of signal transitions in both C and C', up to the last

state transition of t_{n+1}, their state prior to the last transition is the same in both circuits, and thus at that point any signal of $M^C_{E,i}$ which is enabled in C is also enabled in C'. Secondly, consider the case in which a changing variable of C' in the last state transition of t_{n+1} is driven by a module outside $M^C_{E,i}$: since $\hat{B}^C_{W,i}$ is the automaton behavior of $\hat{M}^C_{W,i}$ and $\boldsymbol{Proj}(\hat{V}^C_{W,i})(t_{n+1}) = t^{\hat{V}^C_{W,i}}_{n+1} \in \hat{B}^C_{W,i}$, the transitions of any such variable (who has to be an external variable) along trace t_{n+1} are preserved in the automaton of $\hat{M}^C_{W,i}$; that is, any such variable changes are also enabled in C'. Thus in both cases we observe that any change of variables of $V^{C'}$ that occurs at the last state transition of t_{n+1} in the original circuit, is also enabled at the last state of $\boldsymbol{Proj}(V^{C'})(t_n)$ in sub-circuit C'. It then follows that $\boldsymbol{Proj}(V^{C'})(t_{n+1}) \in B^{C'}$.

• (ii) Consider the original circuit C and its sub-circuit C', and let $t' \in B^{C'}$ be any sub-circuit trace. Since C and C' are initial-state-compatible, for the base case of $\boldsymbol{Len}(t') = 1$ we have $t' \in \boldsymbol{Proj}(V^{C'})(B^C)$. Now assume that $t'_n \in \boldsymbol{Proj}(V^{C'})(B^C)$ holds for any trace $t'_n \in B^{C'}$ of length n for which $\boldsymbol{Proj}(\hat{V}^C_{W,i})(t'_n) = t^{\hat{V}^C_{W,i}}_n$, where $t^{\hat{V}^C_{W,i}}_n \in \hat{B}^C_{W,i}$. (Note that trace $t^{\hat{V}^C_{W,i}}_n$, corresponding to trace t'_n, is not necessarily of length n, and the subscript is only to emphasize the correspondence.).We show that any trace $t'_{n+1} \in B^{C'}$ of length $n+1$, such that t'_n is the prefix of t'_{n+1}, $\boldsymbol{Proj}(\hat{V}^C_{W,i})(t'_{n+1}) = t^{\hat{V}^C_{W,i}}_{n+1}$, and $t^{\hat{V}^C_{W,i}}_{n+1} \in \hat{B}^C_{W,i}$, will satisfy the condition $t'_{n+1} \in \boldsymbol{Proj}(V^{C'})(B^C)$. To see this, any variable of C' involved in the last state transition of t'_{n+1} is either driven by a module in $M^C_{E,i}$ or by $\hat{M}^C_{W,i}$. First consider the case in which $M^C_{E,i}$ drives a

changing variable of the last state transition of t'_{n+1}: since the modules of $M^C_{E,i}$ have experienced the same set of signal transitions in both C and C', up to the last state transition of t'_{n+1}, their state prior to the last transition is the same in both circuits, and thus at that point any signal of $M^C_{E,i}$ which is enabled in C' is also enabled in C. Secondly, consider the case in which a changing variable of C' in the last state transition of t'_{n+1} is driven by $\hat{M}^C_{W,i}$ and is thus a variable in $\hat{V}^C_{W,i}$: from $t^{\hat{V}^C_{W,i}}_{n+1} \in \textbf{\textit{Proj}}(\hat{V}^C_{W,i})(B^C)$ and $\textbf{\textit{Proj}}(\hat{V}^C_{W,i})(t'_{n+1}) = t^{\hat{V}^C_{W,i}}_{n+1}$ we know that $\textbf{\textit{Proj}}(\hat{V}^C_{W,i})(t'_{n+1}) \in \textbf{\textit{Proj}}(\hat{V}^C_{W,i})(B^C)$, suggesting that any $\hat{V}^C_{W,i}$ changes in C' and along t'_{n+1} are also enabled in C. Thus in both cases we observe that any change of variables of $V^{C'}$ that occurs at the last state transition of t'_{n+1} in sub-circuit C', is also enabled (possibly after a sequence of non-$V^{C'}$ signal transitions) in C. It then follows that $t'_{n+1} \in \textbf{\textit{Proj}}(V^{C'})(B^C)$. ■

Corollary 3.6 [Properties of traces captured in a safe abstraction] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any circuit for which there exists a behavior B^{W^C} that is a safe abstraction of B^C over some $W^C \subseteq V^C$. Let $t \in B^C$ be any trace for which there exists $t^{W^C} \in B^{W^C}$ such that $\textbf{\textit{Proj}}(W^C)(t) = t^{W^C}$, and let $C' = C^C_{W,i}$ be any sub-circuit of C. Then $\textbf{\textit{Proj}}(V^{C'})(t) \in B^{C'}$. Moreover, if $t' \in B^{C'}$ is any trace such that $\textbf{\textit{Proj}}(\hat{V}^C_{W,i})(t') = \textbf{\textit{Proj}}(\hat{V}^C_{W,i})(t^{W^C})$, then $t' \in \textbf{\textit{Proj}}(V^{C'})(B^C)$.

Informally speaking, Corollary 3.6 suggests that if a circuit trace t is successfully abstracted by a safe abstraction (i.e. by trace t^{W^C}), then not only (the projection of) trace t will be locally present in any sub-circuit of the circuit, but also any trace t' of

any sub-circuit C' that adheres to trace t^{W^C}--and thus to t--will be (globally) present in circuit C. ■

Proof (Sketch) Since $\boldsymbol{Proj}(W^C)(t) = t^{W^C}$ and $\hat{V}^C_{W,i} \subseteq W^C$, by Lemma 2.1 we have

$$\boldsymbol{Proj}(\hat{V}^C_{W,i})(t) = \boldsymbol{Proj}(\hat{V}^C_{W,i})(t^{W^C}). \tag{13}$$

Since $t^{W^C} \in B^{W^C}$, we have

$$\boldsymbol{Proj}(\hat{V}^C_{W,i})(t^{W^C}) \in \boldsymbol{Proj}(\hat{V}^C_{W,i})(B^{W^C}). \tag{14}$$

From (13) and (14) we have

$$\boldsymbol{Proj}(\hat{V}^C_{W,i})(t) \in \boldsymbol{Proj}(\hat{V}^C_{W,i})(B^{W^C}). \tag{15}$$

From Lemma 3.3 we have

$$\boldsymbol{Proj}(\hat{V}^C_{W,i})(B^{W^C}) \subseteq \hat{B}^C_{W,i}. \tag{16}$$

From (15) and (16) we have

$$\boldsymbol{Proj}(\hat{V}^C_{W,i})(t) = t^{\hat{V}^C_{W,i}} \in \hat{B}^C_{W,i}. \tag{17}$$

It then follows from (17) and Lemma 3.5 that $\boldsymbol{Proj}(V^{C'})(t) \in B^{C'}$.

On the other hand, from $\boldsymbol{Proj}(\hat{V}^C_{W,i})(t') = \boldsymbol{Proj}(\hat{V}^C_{W,i})(t^{W^C})$ and (13) we have

$$\boldsymbol{Proj}(\hat{V}^C_{W,i})(t') = \boldsymbol{Proj}(\hat{V}^C_{W,i})(t), \tag{18}$$

and then from (17) and (18) we have

$$\boldsymbol{Proj}(\hat{V}^C_{W,i})(t') = t^{\hat{V}^C_{W,i}}. \tag{19}$$

It then follows from (19) and Lemma 3.5 that $t' \in \boldsymbol{Proj}(V^{C'})(B^C)$. ■

Note that condition $\boldsymbol{Proj}(\hat{V}^{C}_{W,i})(t') = \boldsymbol{Proj}(\hat{V}^{C}_{W,i})(t^{W^C})$ of Corollary 3.6 is equivalent to $\boldsymbol{Proj}(\hat{V}^{C}_{W,i})(t') \in \boldsymbol{Proj}(\hat{V}^{C}_{W,i})(B^{W^C})$.

Corollary 3.7 [Circuit and sub-circuit behaviors]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any circuit for which there exists a behavior $B^{W^C} = \boldsymbol{Proj}(W^C)(B^C)$ that is a safe abstraction of B^C over some $W^C \subseteq V^C$, and let $C' = C^{C}_{W,i}$ be any sub-circuit of C. Then $\boldsymbol{Proj}(V^{C'})(B^C) \subseteq B^{C'}$. ■

Informally speaking, Corollary 3.7 suggests that if a safe abstraction of the behavior of a circuit is exact, then the projection of the circuit behavior will be locally present in any sub-circuit of the circuit. That is, there is no circuit trace not exhibited by each sub-circuit.

Proof (Sketch) For the special case of $B^{W^C} = \boldsymbol{Proj}(W^C)(B^C)$, for any $t \in B^C$ there exists a $t^{W^C} \in B^{W^C}$ such that $\boldsymbol{Proj}(W^C)(t) = t^{W^C}$, and thus by Corollary 3.6 we have $\boldsymbol{Proj}(V^{C'})(t) \in B^{C'}$. It then follows that $\boldsymbol{Proj}(V^{C'})(B^C) \subseteq B^{C'}$. ■

Lemma 3.8 [Under approximation of reduced sub-circuit behaviors] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any circuit for which there exists a behavior $B^{W^C} \subseteq \boldsymbol{Proj}(W^C)(B^C)$ that is a safe abstraction of B^C over some $W^C \subseteq V^C$, $E^C = A^C \cap W^C$, and let $C' = C^{C}_{W,i}$ be any sub-circuit of C. Then $\boldsymbol{Red}(B^{C'}) \subseteq \boldsymbol{Proj}(V^{C'})(B^C)$. ■

Informally speaking, Lemma 3.8 suggests that the behavior of any sub-circuit of a circuit with a safe abstraction, when reduced, is completely present in the projection of

the circuit behavior. That is, there is no prime trace of the sub-circuit not exhibited by the circuit. Note that a prime trace, if not failure-free itself, has an immediate prefix that is failure-free.

Proof (Sketch) By Lemma 3.4, we know that $\boldsymbol{Proj}(\hat{A}^{C}_{W,i})(B^{W^C}) \subseteq \boldsymbol{Proj}(\hat{A}^{C}_{W,i})(B^{C})$; thus, the possible interactions of circuit-block $M^{C}_{E,i}$ with the rest of the circuit can only be under-approximated by environment module $\hat{M}^{C}_{W,i}$. To see this, note that environment module $\hat{M}^{C}_{W,i}$ is directly derived from (a projection of) B^{W^C} by solely labeling unexpected signal transitions at the *inputs* of $\hat{M}^{C}_{W,i}$ as failure transitions; thus, the behavior of the *output* signals of $\hat{M}^{C}_{W,i}$ --who serve as the input signals of circuit block $M^{C}_{E,i}$ --exactly adhere to B^{W^C}. Now, within such an under-approximated abstract environment $\hat{M}^{C}_{W,i}$, the (reduced or prime) behavior of circuit block $M^{C}_{E,i}$ can only be an under-approximation of the behavior of $M^{C}_{E,i}$ within its real environment; i.e., $\boldsymbol{Red}(B^{C'}) \subseteq \boldsymbol{Proj}(V^{C'})(B^{C})$. This relation is stated over $\boldsymbol{Red}(B^{C'})$, and not $B^{C'}$. The reason is that if $B^{C'}$ contains an input choke to $\hat{M}^{C}_{W,i}$, since the reaction of $\hat{M}^{C}_{W,i}$ to that choke is not originally specified by the safe abstraction, any behavior beyond that failure point can be a spurious behavior (introduced by our arbitrary choice of the destination state of a failure transition), not necessarily present in the original circuit. However, the fact that the above relation holds for $\boldsymbol{Red}(B^{C'})$ and not $B^{C'}$ does not make it any less attractive. This is true since only the first fault along any trace is significant to us; i.e., we only care about prime traces and behaviors.

More formally, consider any prime trace $t' \in B^{C'}$ and let $t'_{\hat{V}^C_{W,i}} = \textbf{\textit{Proj}}(\hat{V}^C_{W,i})(t')$. There are two cases; either $t'_{\hat{V}^C_{W,i}} \in \textbf{\textit{Proj}}(\hat{V}^C_{W,i})(B^{W^C})$ or $t'_{\hat{V}^C_{W,i}} \notin \textbf{\textit{Proj}}(\hat{V}^C_{W,i})(B^{W^C})$. In case of $t'_{\hat{V}^C_{W,i}} \in \textbf{\textit{Proj}}(\hat{V}^C_{W,i})(B^{W^C})$, Corollary 3.6 immediately suggests that $t' \in \textbf{\textit{Proj}}(V^{C'})(B^C)$. So, consider the case of $t'_{\hat{V}^C_{W,i}} \notin \textbf{\textit{Proj}}(\hat{V}^C_{W,i})(B^{W^C})$, where $t'_{\hat{V}^C_{W,i}}$ must be a failure trace of C' ending with an input choke to environment module $\hat{M}^C_{W,i}$. Let $t'' \in B^{C'}$ be the immediate prefix of t'. Note that since t' is a prime failure trace, its prefix t'' will be failure-free, and we will have $t''_{\hat{V}^C_{W,i}} = \textbf{\textit{Proj}}(\hat{V}^C_{W,i})(t'') \in \textbf{\textit{Proj}}(\hat{V}^C_{W,i})(B^{W^C})$. But then by Corollary 3.6 we will have $t'' \in \textbf{\textit{Proj}}(V^{C'})(B^C)$. This suggests that the modules of $M^C_{E,i}$ can experience the same sequence of signal transitions of t'' in both C and C', reaching a common local state in $M^C_{E,i}$ at the end of t''. But then, any signal of $M^C_{E,i}$ which is enabled in C' at the end of t'' is also enabled in C; that is, the last (failure) state *transition* of C' along t' is also enabled in C, although the *reached* states may not be compatible. In other words, $t' \in \textbf{\textit{Proj}}(V^{C'})(B^C)$ for the case of $t'_{\hat{V}^C_{W,i}} \notin \textbf{\textit{Proj}}(\hat{V}^C_{W,i})(B^{W^C})$ (again, note that the last signal transition of t' is present in $\textbf{\textit{Proj}}(V^{C'})(B^C)$, but probably not the last state of t'). Thus we have shown that $t' \in \textbf{\textit{Proj}}(V^{C'})(B^C)$ holds for any prime trace $t' \in B^{C'}$, which is equivalent to saying $\textbf{\textit{Red}}(B^{C'}) \subseteq \textbf{\textit{Proj}}(V^{C'})(B^C)$. ■

At this point we are ready to present the proof of the main two theorems of this section, Theorem 3.1 and Theorem 3.2.

Theorem 3.1. [Circuit versus sub-circuit failure-freedom, I]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any circuit for which there exists a behavior B^{W^C} that is a safe abstraction of B^C over some $W^C \subseteq V^C$, and $E^C = A^C \cap W^C$. Then, if any sub-circuit $C' = C^C_{W,i}$ is not failure-free, then C is not failure-free.

Proof (Sketch) Let $t' \in B^{C'}$ be any (shortest) failure trace of C' which is prime; i.e., $t' \in \boldsymbol{Red}(B^{C'})$. By Lemma 3.8 we have $\boldsymbol{Red}(B^{C'}) \subseteq \boldsymbol{Proj}(V^{C'})(B^C)$, which together with $t' \in \boldsymbol{Red}(B^{C'})$ suggest that $t' \in \boldsymbol{Proj}(V^{C'})(B^C)$. Thus, there must exist a trace $t \in B^C$ such that $t' = \boldsymbol{Proj}(V^{C'})(t)$; that is, the variables of sub-circuit C' can observe the same sequence of transitions (that of t') in both C and C'. Now, consider the last state transition of trace $t' \in B^{C'}$ which is by assumption a failure transition. The failing circuit module of C' (experiencing an illegal input signal transition) is either an ordinary module of $C' = C^C_{W,i}$ (and thus a module of C), or it is environment module $\hat{M}^C_{W,i}$. If the failing module of C' is an ordinary module, then the failure is obviously a failure of C as well, since the failing module can experience exactly the same sequence of events in both C and C'. On the other hand, if $\hat{M}^C_{W,i}$ is the failing module of C' (i.e., the transition of an external signal is causing an input choke to $\hat{M}^C_{W,i}$), then the actual output behavior of circuit block $M^C_{E,i}$ must have been under estimated by safe specification $\hat{B}^C_{W,i}$; but this can happen only if the behavior of the external variables W^C was under-approximated by safe abstraction B^{W^C}. (Remember that safe specification $\hat{B}^C_{W,i}$ which defines the expected input transitions of $\hat{M}^C_{W,i}$ is obtained via a projection of safe abstraction B^{W^C}.). However, by definition of a safe abstraction, B^{W^C} is obliged to exactly resemble the behavior of W^C if circuit C is

failure-free; in other words, if B^{W^C} is not exact, then C is not failure-free. Hence, input chokes to $\hat{M}^C_{W,i}$ are always indicative of circuit C failure. This completes our proof that any failure in any sub-circuit C' is always an indication of failure of circuit C. ■

Theorem 3.2. [Circuit versus sub-circuit failure-freedom, II]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any circuit for which there exists a behavior B^{W^C} that is a safe abstraction of B^C over some $W^C \subseteq V^C$, and $E^C = A^C \cap W^C$. If all sub-circuits $C^C_{W,1}, \ldots, C^C_{W, r^C_E}$ are failure-free, then C is, itself, failure-free. ■

Proof (Sketch): We prove the failure-freedom of C by way of contradiction. Suppose C is not failure-free. Under this assumption, and by the definition of a safe abstraction, we must have $B^{W^C} \subseteq \boldsymbol{Proj}(W^C)(B^C)$; that is, either $B^{W^C} \subset \boldsymbol{Proj}(W^C)(B^C)$ or $B^{W^C} = \boldsymbol{Proj}(W^C)(B^C)$.

If $B^{W^C} \subset \boldsymbol{Proj}(W^C)(B^C)$ is the case (i.e., B^{W^C} under-approximates $\boldsymbol{Proj}(W^C)(B^C)$), then there must be a shortest trace $t = q_0 \ldots q_1 \ldots q_n \ldots r q_{n+1} \in B^C$ such that $t_{W^C} = \boldsymbol{Proj}(W^C)(t) = \boldsymbol{Proj}(W^C)(q_0 \ldots q_1 \ldots q_n \ldots r q_{n+1}) \notin B^{W^C}$. Here, all and only those states of trace t which are entered with some external variable change are labeled as q_j, $0 \leq j \leq n+1$. Thus any pair of states q_j and q_{j+1} are separated by maximal non-observable sub-traces of t. (A non-observable sub-trace is one which does not contain any external variable (W^C) changes.). State r is the next to last state of trace t, and there must be an external signal $a \in E^C \subseteq W^C$ which is involved in the transition from state r to state q_{n+1}. There must then exist a unique circuit block $M^C_{E,i}$ such that signal a is an external output signal of $M^C_{E,i}$; that is $a \in Z^C_{E,i}$. Now consider $t' = q_0 \ldots q_1 \ldots q_n \ldots r$, the immediate prefix of t, and its projection $t'_{W^C} = \boldsymbol{Proj}(W^C)(t') = \boldsymbol{Proj}(W^C)(q_0 \ldots q_1 \ldots q_n)$. (Note that the projection of the last maximal non-observable sub-trace of t' is the same as $\boldsymbol{Proj}(W^C)(q_n)$.).

Since t is the shortest trace of interest, we must have $t'_{W^C} \in B^{W^C}$; but then by Lemma 3.5 we must have $\boldsymbol{Proj}(V^{C'})(t') \in B^{C'}$, where $C' = C^C_{W,i}$. Hence, the circuit modules of block $M^C_{E,i}$ can experience the same sequence of transitions (that of $\boldsymbol{Proj}(V^{C'})(t'_{W^C})$) in both C and $C' = C^C_{W,i}$. But this suggests that external output signal a of $M^C_{E,i}$ is enabled at state $\boldsymbol{Proj}(V^{C'})(q_n) = \boldsymbol{Proj}(V^{C'})(r)$. On the other hand $\hat{B}^C_{W,i}$, the safe specification of $M^C_{E,i}$, specifies any transition of signal a at state $\boldsymbol{Proj}(V^{C'})(r)$ as a failure transition; this is true because $t_{W^C} \notin B^{W^C}$ implies that $\boldsymbol{Proj}(V^{C'})(t_{W^C}) \notin \boldsymbol{Proj}(V^{C'})(B^{W^C})$. Now, since on one hand a is enabled at state $\boldsymbol{Proj}(V^{C'})(r)$ of C', and on the other hand it is not expected to be enabled by safe specification $\hat{B}^C_{W,i}$, any transition of a will cause an input choke to environment module $\hat{M}^C_{W,i}$, suggesting that C' is not failure-free. But all sub-circuits of C are failure-free by the conditions of Theorem 3.2. Thus, the assumption of C not being failure-free leads to a contradiction in the case of $B^{W^C} \subset \boldsymbol{Proj}(W^C)(B^C)$.

Next, under the assumption of C not being failure-free, consider the case of $B^{W^C} = \boldsymbol{Proj}(W^C)(B^C)$. Then, there must exist a shortest (prime) failing trace $t = q_0 \ldots q_1 \ldots q_n \ldots r'r \in B^C$, an internal signal $a \in H^C$, a unique circuit module $M^j \in M^C$ of circuit C, such that signal $a \in X^j$ has a transition from state r' to r which is illegal. There must also exist a unique circuit block $M^C_{E,i}$ such that $M^j \in M^C_{E,i}$. By Corollary 3.7, $B^{W^C} = \boldsymbol{Proj}(W^C)(B^C)$ implies that $\boldsymbol{Proj}(V^{C'})(t) \in B^{C'}$. This implies that the above failure at circuit element M^j will also be present in $C^C_{W,i}$, suggesting that $C^C_{W,i}$ is not failure-free. Thus, the assumption

of C not being failure-free leads to a contradiction in the case of $B^{W^C} = \boldsymbol{Proj}(W^C)(B^C)$.

We have just shown that the assumption of C not being failure-free always would imply the presence of some failure in some sub-circuit which is in contradiction with the conditions of Theorem 3.2. Thus circuit C must be failure-free if all of its sub-circuits are failure-free. ■

Chapter 4

Induced Hierarchical Verification of Speed-Independence, Issues

In this section, we first compare our proposed framework for hierarchical verification of speed-independent circuits with that of complex-gate verification, in terms of how the two frameworks choose the set of external variables over which safe abstractions are found. Next, we discuss the issue of choosing sets of external variables that are observationally sufficient (OSV sets), and how the choice can affect the performance of hierarchical verification. Finally, we introduce the concept of sequential hierarchical verification (SHV) as a heuristic that can improve the performance of hierarchical verification through better informed decisions; on the selection of external variables, and/or on the order in which sub-circuits are verified.

4.1 Circuit Blocks Versus Complex-Gates

Our proposed framework for induced hierarchical verification of speed-independent circuits is a generalization of a previous technique for two-level hierarchical verifica-

tion of speed-independent circuits, called complex-gate verification [64, 65]. Both frameworks try to find a safe abstraction of the circuit behavior over a set of external variables which is then used to induce hierarchy in verification of the circuit. In this subsection, we compare the two frameworks in terms of their constraints for selection of sets of external variables, and how such constraints affect the requirements and performance of the two frameworks.

In complex-gate verification, the set of external circuit variables over which a safe abstraction is found is taken as a superset of all output signals of sequential circuit modules. Then, for any module with external outputs, the module and the combinational cone of logic driving it are collapsed into a complex-gate and complete reachability analysis is performed on the collapsed circuit to find the behavior of its set of external signals. Such sets of external signals partition a circuit into circuit blocks each of which containing one or more complex-gates. Once a safe abstraction is found, each circuit block can then be checked for conformance to its specification which is derived from the safe abstraction. Note that since the complex-gate circuit has less signals, its full reachability analysis is less expensive than that of the flat circuit.

First of all, note the limitation of this technique in not being able to hide outputs of sequential modules. This limitation is not present in our more general verification framework. Being able to hide more signals, our framework can potentially outperform this technique when deriving safe abstractions.

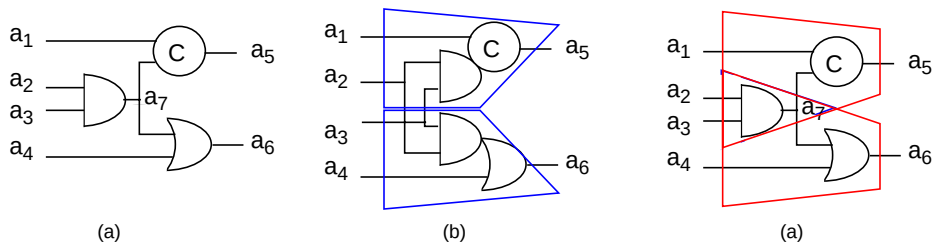


Fig. 4.1 A portion of a circuit with a multiple fan-out signal a_7.
(b) Complex-gate circuit. (c) Equivalent overlapped circuit blocks.

A second limitation of this approach is concerned with the verification of individual complex-gates. It very often is the case that complex-gates of a circuit overlap (See Figure 4.1). Overlapping arbitrary circuit blocks and their associated problems were discussed in a previous section. We noted that any signal in the common portion of two overlapped arbitrary circuit blocks has to be external. However, in complex-gate verification approach, all signals of the common portion of two complex-gates are hidden, since they are internal signals of each of the two complex-gates. This suggests that a complex-gate with overlapped logic cannot be verified individually. There are two ways to solve this problem (See Figure 4.2). The first solution is to add to the set of external signals, any signal which would have otherwise forked into two different (single output) complex-gates. This solution will increase the number of external signals, and thus add to the complexity of deriving safe abstractions. Another solution is to combine overlapping complex-gates into multiple output complex-gates in such a way that no two multiple output complex-gates overlap. (Note that such multiple-output complex-gates are in fact same as the circuit blocks induced by partitioning the circuit by the set of external signals.). This solution can potentially result large circuit blocks

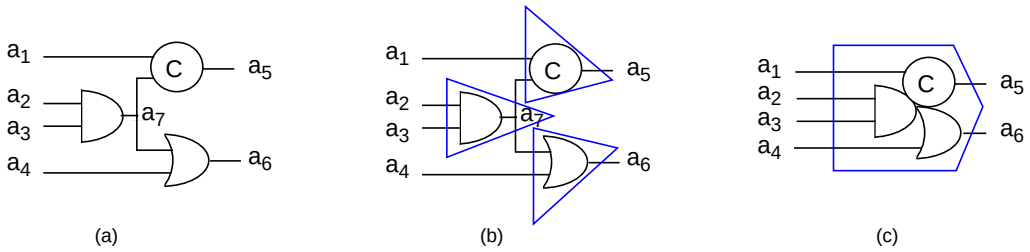


Fig. 4.2 Two solutions to the problem of overlapping complex-gates.
(b) Including signal a in the set of external signals, and thus increasing the number of circuit blocks,
(c) Combining the two single-output complex-gate into a larger two-output complex-gate, reducing the number of circuit blocks.

whose verification would be more expensive than smaller ones. Such large blocks may need to be further partitioned into smaller blocks by choosing the signals that fork into multiple complex-gates as the external signals of the next level of hierarchy. This solution can be less expensive than the first one. However, both solutions reveal that to correctly verify the circuit in this framework, not all outputs of combinational modules can always be effectively hidden. This limitation, together with not being able to ever hide the outputs of sequential modules, highlights the advantage of our more general framework.

4.2 Selection of OSV Sets for Hierarchical Verification

One of the most controversial issues with our hierarchical verification technique is the problem of choosing the set of external variables. While this problem, in its most general form, can be an interesting subject for future research, some ad hoc and inherent solutions are already available for it.

Very often, observationally sufficient sets of circuit variables--over which safe abstractions exist--have high correlations with handshake signals of the circuit. Since full handshake protocols are an essential part of any speed-independent design, especially at higher levels of design hierarchy, coming up with OSV sets is not a hard problem, and designers can easily make an initial guess for an OSV set. If the observational sufficiency of such a set can not be proven (e.g., an attempt to find a safe abstraction over that set fails), it is often easy to figure out which signals/variables were involved in violating the safety of the abstract behavior. Such signals/variables can then be added to the set of external variables, and this procedure can be repeated until a safe abstraction, and thus an OSV set, is found. This approach usually works very well, unless the initial guess is not a good one.

It is to be noted, that failure in finding a safe abstraction over an OSV set would cause a failure in recognizing its observational sufficiency. We can ensure that a set is OSV only when we are successful in finding a safe abstraction (i.e., when the underlying sub-automaton is projectable); otherwise, we had better choose another set of external signals and see if we can find a safe abstraction over them. On the other hand, increasing the size of a set of external signals is not always a guarantee that it will eventually become OSV, and stay OSV from that point on. In general, a set of variables which is an unrecognized OSV set can easily loose the property by inclusion of a new variable(s), or it may retain the property but not be recognized as an OSV set again.

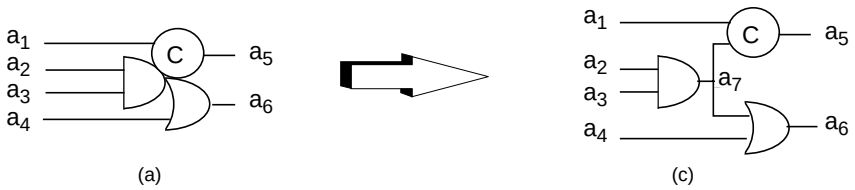


Fig. 4.3 An example of technology mapping

As the circuit is broken up into smaller and smaller circuit blocks, it becomes harder to choose sets of external signals, since not much handshaking may be present inside small pieces of the circuit. As we discussed in the section about complex-gate verification, the output signals of combinational gates can be hidden in many cases. Exceptions can include cases where the output of a sequential gate *have to be* absent from an external set of handshake signals.

To solve the problem of which sequential module outputs to hide, the circuit designers can once again come to help. An example of this case is in technology mapping of SI circuits using sequential decomposition [21, 25, 46]. Sequential decomposition substitutes a multi fan-in gate with a functionally and behaviorally equivalent cone of logic that is composed of gates with smaller fan-ins (See Figure 4.3). Since only the output of the new cone is expected to behave exactly as that of the original gate, and in that case, the behavior of the newly introduced signals connecting the set of modules is insignificant, they can all be hidden, even if they are outputs of sequential gates (See Figure 4.4). This is very similar to the case of complex-gate circuits. First, remember that for any circuit, the set of signals of the corresponding complex-gate circuit is always an OSV set. Secondly, note that when a module is decomposed,

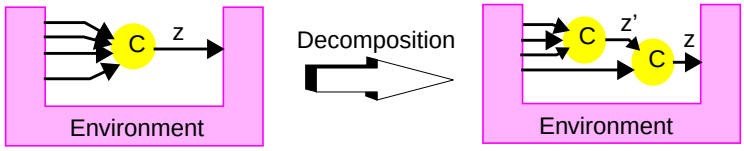


Fig. 4.4 An example of sequential decomposition in technology mapping.
The newly introduced signal Z' can be hidden during hierarchical verification.

the resulting modules can be thought of as collapsing into the original module, as if the original module is a *pseudo* complex-gate. It then follows that the new signals introduced by sequential decomposition can all be hidden.

4.3 Sequential Hierarchical Verification, SHV

In this section, we present some general directives which can potentially speed up hierarchical verification. We will also discuss the issues involved with such procedures.

As was mentioned in the previous section, OSV sets are very often a collection of handshake signals of the circuit. For circuits that are composed of a large number of communicating circuit blocks, the number of handshake signals can be very large. (This can also be true at lower levels of the design hierarchy.). However, since the cost of finding a safe abstractions is exponential in the size of the selected set of external variables, we are much more interested in smaller sets. Smaller OSV sets may not represent all the circuit blocks of a particular level of the design hierarchy; i.e., a smaller OSV set usually represents larger circuit blocks, and a larger circuit block may encom-

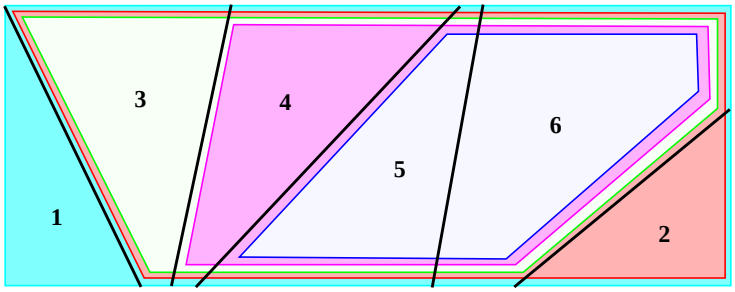


Fig. 4.5 An abstract illustration of Sequential Hierarchical Verification.
SHV can be directed by the knowledge of possible failure locations. Blocks of the circuit at the highest level are ordered by the possibility of failure existence. At each level of hierarchy, the larger circuit block is further partitioned into two circuit blocks, a small one and a large one, such that the location of the next highly possible failure falls in the smaller block.

pass a couple of circuit blocks associated with a larger OSV set. Smaller OSV sets very often include the handshake variables among subsets of communicating circuit blocks, where the circuit blocks within each subset have direct mutual communications. Thus, smaller OSV sets tend to partition the circuit into circuit blocks, such that each circuit block is collection of adjacent circuit blocks associated with a larger OSV set. Now, given a small OSV set, its circuit blocks can be further partitioned into smaller ones. This suggests that smaller OSV sets increase the depth of hierarchical verification, but speed up the derivation of safe abstractions. There is a trade off between the speed up of deriving safe abstractions and the increase in the depth of hierarchy. However, since the former has an exponential cost and the latter has a sub-exponential cost, smaller OSV sets are better preferred.

Now, consider the case in which the size of the circuit blocks associated with a small OSV are not balanced; i.e., some of the circuit blocks are small and can be verified in fewer levels of hierarchy. If such OSV sets exist for a given circuit and the

designer is most concerned about design errors located in the smaller circuit blocks, the verification procedure can be sped up by first verifying those small circuit blocks, and proceeding to other circuit blocks only once the small ones are found to be failure-free. In this paradigm, the larger blocks are broken into smaller ones in a similar fashion; that is, OSV sets are chosen in such a way that culprit design errors are most probably located in smaller circuit blocks. We call this verification paradigm *sequential hierarchical verification*, or SHV (See Figure 4.5). Note that although this technique can potentially speed up finding failures, a final decision on failure-freedom of any block of the circuit cannot me made until all blocks are verified as failure-free.

Knowledge of the possible location of design errors is not the only motivation for SHV. Another motivation for SHV can be the relative ease of finding safe abstractions. For example, consider a circuit which is to be verified against a specification. The circuit can be thought of as a collection of cones of logic, each driven by the inputs of the circuit and driving one output of the circuit. Now, if there exists a (reasonably) small cone of logic and a small OSV set containing the I/O signals of that cone, then that cone can be verified quickly, and the rest of the circuit can be verified in a similar fashion, sequentially (See Figure 4.6). Here, the SHV paradigm is directed towards speeding up the verification, without necessarily having the knowledge of the possible location of design errors.

Finally, it is to be noted that the performance of any SHV procedure is very dependent on the choice of appropriate OSV sets (and their existence), and ordering of veri-

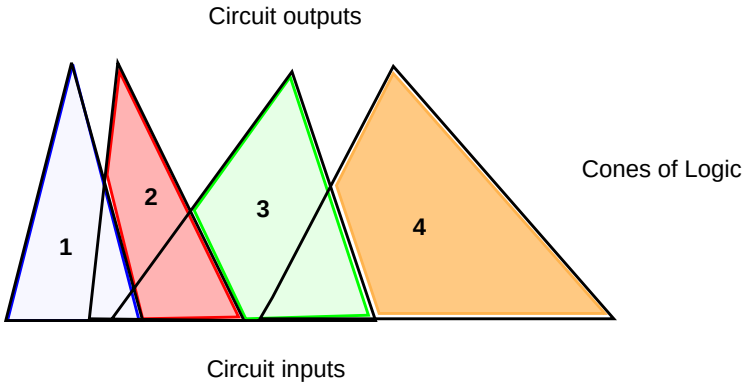


Fig. 4.6 An abstract illustration of Sequential Hierarchical Verification.
SHV can be directed by the ease of deriving safe abstractions for cones of logic. Cones of logic are verified in the specified order. At each step, the next cone constitutes a single small circuit block, while the remaining cones constitute a large circuit block(s).

fication of the circuit blocks at each level of hierarchy. While designers (as well as their intuition) should be able to guide such SHV approaches in many cases, devising heuristics for SHV verification can be an interesting subject for future research.

Chapter 5

Finding Safe Abstractions

Our hierarchical verification framework was presented in a previous chapter, along with a proof of its correctness. In this framework, a safe abstraction of the behavior of a circuit over a set of external variables is used to verify sub-circuits of the circuit that are induced by the safe abstraction--in a recursive and hierarchical fashion. This hierarchical approach, assuming that there are efficient techniques to derive safe abstractions, can speed up the verification process.

Safe abstractions and efficient techniques to actually find them are the subject of this chapter. We use a partial order technique to find safe abstractions. This partial order technique constructs a subtle sub-automaton of the circuit automaton by partially exploring the state space of the circuit in a delicate fashion. The circuit sub-automaton is constructed with the goal of preserving *all* external variable transitions and maintaining as little number of interleavings of internal variable transitions as possible. By construction, if the sub-automaton is projectable onto the set of external variables, then the behavior of its projection is guaranteed to be a safe abstraction of the circuit behavior. Since partial order techniques are reduction techniques that mitigate the state

explosion problem, by using them in deriving safe abstractions we have achieved our goal of efficient hierarchical verification.

This chapter is organized as follows. In Section 5.1 the general concepts and terminology associated with partial order reductions are introduced. In Section 5.2 we show how a particular class of partial order reduction techniques can be utilized for our specific problem of finding safe abstractions. This technique is capable of constructing a sub-automaton of circuit automaton that preserves the behavior of external variables of failure-free circuits. We know from the previous chapter that if such sub-automaton is also projectable onto the set of external variables, its projection would be a safe abstraction. Based on the requirements of this particular partial order technique, we then derive a set of constraints for the set of external circuit variables. Finally, we present a first partial order reduction algorithm and proof its correctness in generating reduced state spaces that can be used for finding safe abstractions. In Section 5.3, we present an enhanced partial order algorithm as a complete solution for finding safe abstractions. This algorithm is also furnished with an embedded procedure for on-the-fly projection of the constructed sub-automaton. The correctness of the enhanced algorithm is proven, and the chapter is closed by presenting an optimized version of the algorithm which can further improve the performance of partial order reduction for finding safe abstractions.

5.1 Some Background

Formal verification paradigms that are based on state space exploration can often greatly benefit from partial order reduction techniques that help attack the state space explosion problem [1, 62, 63, 32, 33, 81, 82]. In asynchronous systems, which are highly concurrent systems, one source of state space explosion is the exponential ($n!$) number of possible interleavings of n concurrent events. If the concurrent events are *independent*, then all such interleavings are equivalent since they all lead to the same state. Now, if the property of the system to be verified does not depend on the ordering of such concurrent (independent) events, it would suffice to explore just one representative interleaving of them from the set of all possible interleavings. Consequently, during state space exploration, at each state it suffices to explore an *ample* set of enabled transitions, rather than all of them. This can usually lead to significant reduction in the size of the explored state space, especially for highly concurrent asynchronous systems. In our framework, we use partial order reduction in finding a safe abstraction of the behavior of a set of *external* circuit variables. As we will see, our partial order reduction, assuming that the *external* variables are *independent* of the *internal* variables, explores in a failure-free circuit only one interleaving of *independent internal transitions*, while exploring all possible external transitions (and thus their interleavings). The explored sub-automaton of the circuit automaton will thus preserve the exact behavior of the external variables of a failure-free circuit, and thus, if it is also projectable onto the set of external variables, its projection would be a safe

abstraction. The details of this approach are the subject of the following sections of this chapter.

In the following subsections, we review the portion of the general framework for partial order reductions [1, 62, 63, 32, 33, 81, 82] that is relevant to our work. Instead of presenting the associated concepts in their original (general) form, we have occasionally tailored some of them into our own framework, only to ease the presentation.

5.1.1 Partial Order Reductions

Peled [62] gives a very concise and yet complete overview of partial order reduction techniques for the analysis of concurrent systems that are modeled with interleaved semantics. In his overview, the general concepts in partial order reductions are presented first, followed by different sets of conditions that must be met for valid reductions in formalisms that include among others LTL (Linear Time Logic), CTL (Computational Tree Logic), and process algebra.

In our framework, a two step procedure is proposed for finding safe abstractions. The first step involves finding a sub-behavior of a circuit that would preserve the behavior of external variables of a failure-free circuit. This problem is shown to be equivalent to the problem of generating a reduced state space of the (failure-free) circuit such that for each trace in the full state space, there is a *stuttering equivalent trace* in the reduced one. Partial order reductions for LTL (Linear Temporal Logic) are claimed to precisely generate what we are looking for; a reduced state space that is

equivalent to the full one *up to stuttering*. Thus, even without going over LTL logics, we have been able to prove the correctness of our partial order technique for finding safe abstractions by showing that it satisfies all the necessary conditions (for LTL), and that it is thus valid by construction.

Our following overview of partial order reduction techniques is accordingly restricted to the domain of reductions for LTL [62]. However, since we directly focus on conditions for stuttering equivalence (and not general LTL properties) we will skip an overview of LTL logics.

We will introduce the relevant concepts, and give specific examples that will gradually form the connection between the general reduction technique (for stuttering equivalence), and our quest for finding safe abstractions.

Definition 5.1 [Finite transition system] [62] A finite transition system is a triple $FTS = \langle FA, AP, L\rangle$, where $FA = \langle A, V, Q, \lambda, TR, \mu, q_0\rangle$ is a finite state automaton, AP is a finite set of *propositions*, and $L{:}Q \rightarrow 2^{AP}$ is an *assignment function*. For any sequence of states $t = q_0 q_1 q_2 \ldots$, we define the corresponding *propositions sequence as* $\boldsymbol{Prop}(t) = L(q_0)L(q_1)L(q_2)\ldots$. ■

Example 5.1 Let $C = \langle M^C, A^C, V^C, G^C, FA^C\rangle$ be a circuit and $W^C \subseteq V^C$ be a set of external variables. We can then define transition system $FTS^C = \langle FA^C, AP^C, L^C\rangle$ as follows: $AP^C = \{\boldsymbol{Proj}(W^C)(q) | q \in Q^C\}$, and $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. Thus, the transition system simply assigns to each state of the circuit automaton FA^C, the projection of that state onto the set of external variables, W^C. ■

As mentioned in [62], partial order reduction is based on several observations about the nature of concurrent computations and specification formalisms. The first observation is that concurrently executed transitions are often *commutative*. This is usually formalized in the definition of *independence*.

In the following, we have tailored the general notion of independence [1, 62, 63, 32, 33, 81, 82] to our own framework, so that it appropriately accounts for the particular way that we label the states of a transition system.

Definition 5.2 [Independent variables] [62]

Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be a circuit automaton. A pair of distinct variables $v, w \in V^C$ are *independent*, written $v \sim w$ if for all states $q \in Q^C$, if $v, w \in \boldsymbol{Enabled}(q)$, then for all transitions $(q, a, q') \in TR^C$ that change v but not w, w is enabled in q', and for all transitions $(q, b, q'') \in TR^C$ that change w but not v, v is enabled in q'', and there exists a unique state $q''' \in Q^C$ such that all a-transitions (there has to exist at least one) from state q'' that change v, and all b-transitions (there has to exist at least one) from state q' that change w lead to q'''; i.e., any two strings a, b and b, a from state q that change v and w (in different orders) always lead to a single state q''' (here, $a, b \in A^C \cup \varepsilon$). ■

Intuitively, two variables are independent if no transition that changes only one of them can disable the other one, and any order of execution of two signal transitions, each changing one of the variables, leads to the same global state. The independence relation on the variables of a circuit automaton is irreflexive and symmetric. It is also

notable that by the above definition of independence, two variables that can change *simultaneously* by a single state transition, are not necessarily dependent.

Let q be any state, and v and w be any two enabled independent variables at q. Then if $(q, a, q') \in TR^C$ is any transition changing v but not w, and $(q, b, q''), (q', b, q''') \in TR^C$ are any pair of transitions changing w but not v, then we must have $(q'', a, q''') \in TR^C$ is also changing v. If a specification is only interested in the first and last states, q and q''', then we do not need to explore the transitions of both v and w from q. Otherwise, one must consider the possibility that the value of propositions might be different at the intermediate states q' and q'', and even be different from those at q or q''', and if so, the transitions of both variables v and w might need to be explored from state q for a valid partial order reduction.

Example 5.2 Let M^i be any module of a circuit C such that $Y^i \neq \varnothing$; i.e., the module has internal variables. Assume that all (local) states of module M^i are reachable within a given circuit. Let $v \in A^i$ and $w \in V^i - A^i$ be any pair of module variables for which there exists a transition $(q, v, q') \in TR^C$ that changes both variables $(\lambda^C(q)|w \neq \lambda^C(q')|w)$, then v and w are simply changing simultaneously at q. On the other hand, if w is enabled in q but disabled in q' without being changed $(\lambda^C(q)|w = \lambda^C(q')|w)$, then v and w are dependent. Similarly, if v and w are enabled in q and there exists a transition $(q, u, q') \in TR^C$, $u \in A^i \cup \varepsilon$, that changes w and disables v, then again v and w are dependent. ■

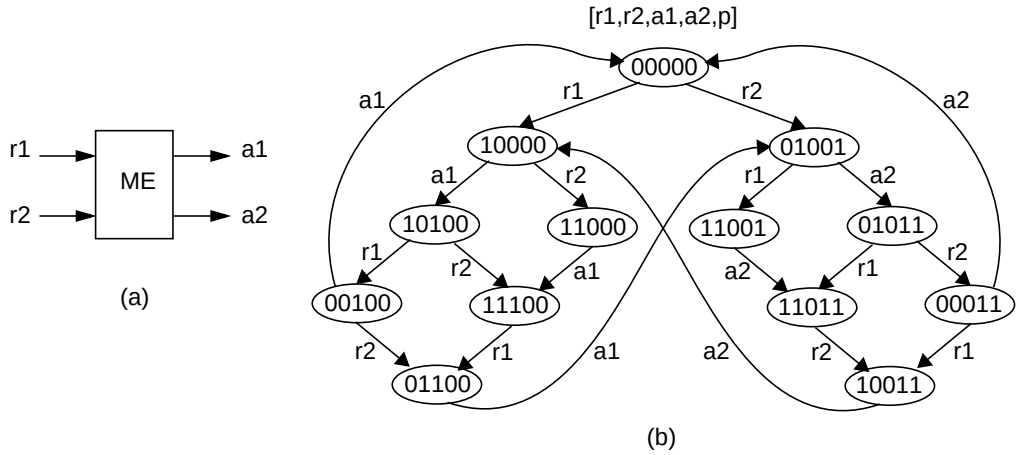


Fig. 5.1 Module description of a fair arbiter element.
(a) A fair arbiter element. (b) The module automaton of the fair arbiter.

Example 5.3 Figure 5.1 shows the module automaton of a fair arbiter M^i. From the module automaton, it can be seen that if M^i is a module in a circuit C, and there exists $q \in Q^C$ such that $\boldsymbol{Proj}(V^i)(q) = 00000$, $r1, r2 \in \boldsymbol{enabled}(q)$, and $r1$ and $r2$ cannot disable each other at q, then different states can be reached from q depending on which signal $r1$ or $r2$ makes its transition first. Thus, signals $r1$ and $r2$ are dependent in circuit C. Note that variable p is also enabled at state q, however, transition of signal $r1$ will disable it. Thus, variables p and $r1$ are dependent in C. Finally, p can simultaneously change with all other three signals $r2$, $a1$ and $a2$, without being dependent with any of them. ■

Example 5.4 Figure 5.2 shows the module automaton of a mutual exclusion (ME) module M^i. From the module automaton, it can be seen that if M^i is a module of a circuit C, and there exists $q \in Q^C$ such that $\boldsymbol{Proj}(V^i)(q) = 0000$ and

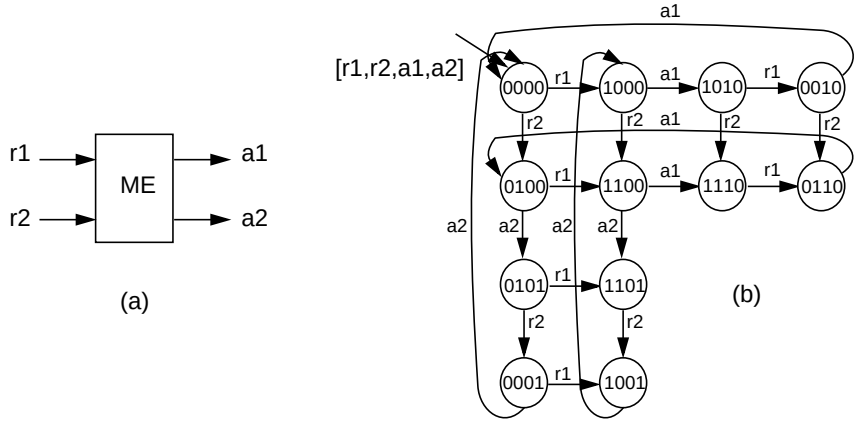


Fig. 5.2 Module description of a Mutual-Exclusion element.
(a) A Mutual-Exclusion element. (b) The module automaton of the ME element.

$r1, r2 \in \boldsymbol{enabled}(q)$, then regardless of the order in which signals $r1$ and $r2$ make their transitions, a unique state can be reached if no other variable changes along the two transitions and $r1$ and $r2$ do not disable each other at q. Thus, the two signals can (possibly) be independent. However, if one of them can disable the other one (e.g., if the circuit has a failure), then the two will be dependent. Now, assume there exists $q \in Q^C$ such that $\boldsymbol{Proj}(V^i)(q) = 1100$. Then both $a1$ and $a2$ are enabled at q, but transition of either of them disables the other one. $a1$ and $a2$ are thus dependent, however, this output choice is not considered a failure. ■

A taxonomy of all possible dependencies between circuit variables is summarized in the following.

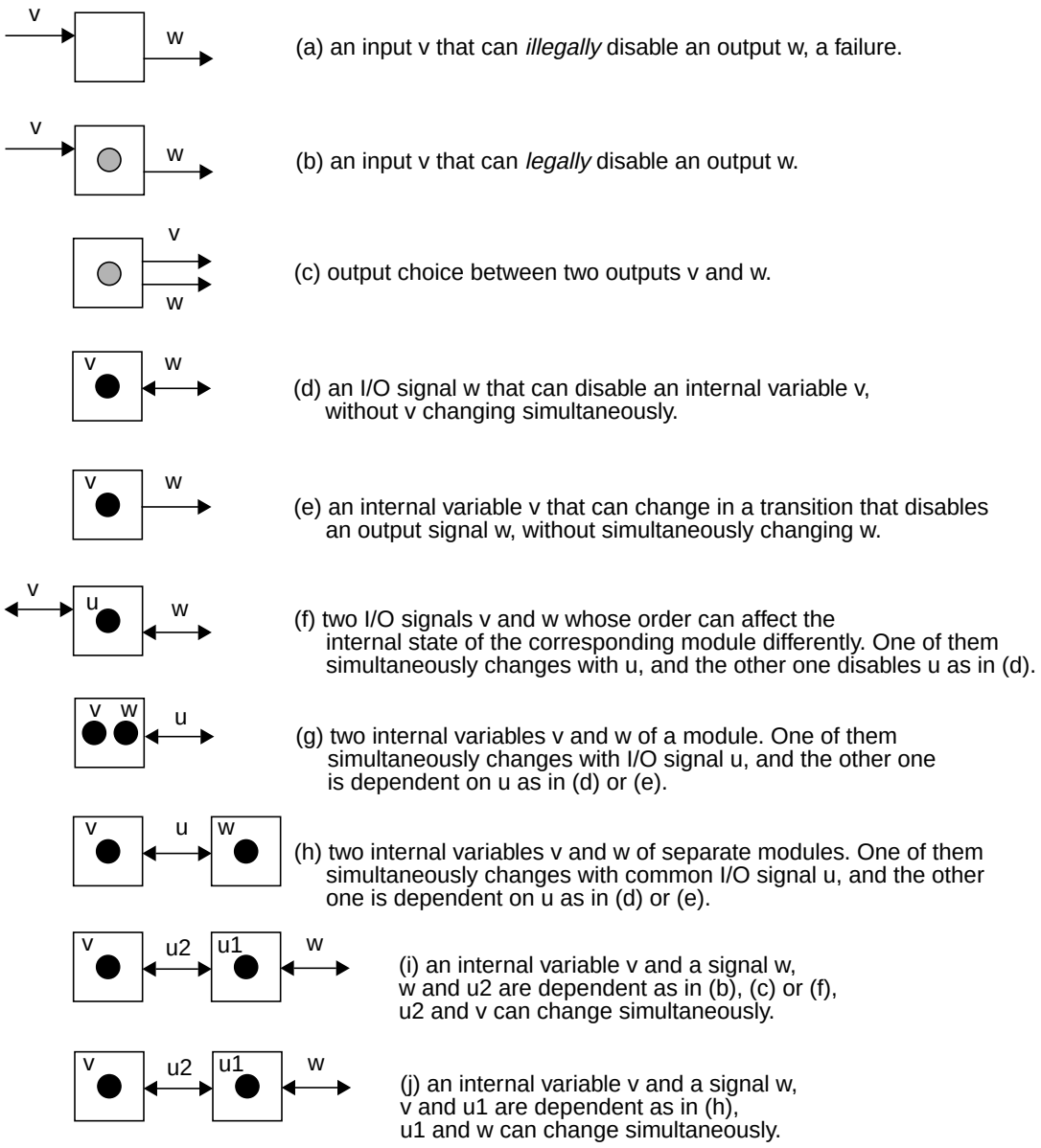


Fig. 5.3 Classification of dependency between any two circuit variables v and w.

Observation 5.1 [Classification of dependencies between circuit variables]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit. A classification of all kinds of dependencies between circuit variables is depicted in Figure 5.3. In the first two cases ((a) and (b)), dependency is due to an input being able to disable an output of a module. The incurred non-determinism can be associated with either a *legal* (acceptable) behavior (case (b)), or an undesirable failure (case (a)). Output choice (case (c)) is another form of legal non-determinism where an output can disable another output; it thus creates dependency between the two outputs (see Example 5.4). If any I/O signal of a module can disable an internal state variable (case (d)), or conversely, if the internal state variable can change in a transition that disables the I/O signal (case (e)), then the I/O signal and the internal state variable are dependent. Case (f) is different from case (b) or (c) in that the two signals v and w do not necessarily disable each other; rather, the module might reach different local states by different interleavings of the two variables. As indicated in Figure 5.3.f, dependence of an internal variable u with one I/O signal v, and its simultaneous transition with another I/O signal w has made the two I/O signals dependent. The four last cases are similar to case (f) in that if any two variables v and u are dependent, then any third variable w that can simultaneously change with v (u) is also dependent on u (v). There might be other dependency types that are missed in Figure 5.3, but the important result of this classification is that any kind of *legal* dependency between two circuit variables is the result of dependencies of types (b), (c), (d), or (e) between (possibly other) pairs of variables that are extended to other variables by means of simultaneity of transitions. ■

Definition 5.3 [Simultaneity, prime, and failure-free dependency conditions] Based on Observation 5.1, we define the set of *prime dependency conditions* as the set containing conditions (b), (c), (d), and (e) of Figure 5.3. We define the *simultaneity condition* to exist between any two circuit variables that can ever change simultaneously. We call the union of prime dependency conditions and the simultaneity condition as *failure-free dependency conditions*. ■

A second observation about concurrent systems with interleaved semantics is that often the transitions of only a few variables can change the truth values of the propositional variables, and thus be visible.

Definition 5.4 [Invisible variables] [62] Let $FTS = \langle FA, AP, L \rangle$ be a finite transition system. A variable $v \in V^C$ is *invisible* if for all transitions $(q, a, q') \in TR$ that change variable v, we have $L(q) = L(q')$. ■

Example 5.5 Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit, $W^C \subseteq V^C$ be a set of external circuit variables, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system as described in Example 5.1 (i.e., $L^C(q) = \boldsymbol{Proj}(W^C)(q)$). Then all variables of W^C are visible. If in addition, W^C is such that no pair of variables $v \in W^C$ and $w \in V^C - W^C$ can change simultaneously, then any $w \in V^C - W^C$ would be an invisible variable. ■

Definition 5.5 [Stuttering equivalence] [62]

Let $FTS = \langle FA, AP, L \rangle$ be a finite transition system. The stutter removal operator $\boldsymbol{Stutt}(.)$ applied to a propositions sequence ρ results in a sequence $\boldsymbol{Stutt}(\rho)$ where each consecutive repetition of labeling is replaced by a single occurrence. Two proposition sequences σ and ρ are *equivalent up to stuttering* if $\boldsymbol{Stutt}(\sigma) = \boldsymbol{Stutt}(\rho)$. Two sequence of states t and t' are *stutter equivalent* if $\boldsymbol{Stutt}(\boldsymbol{Prop}(t)) = \boldsymbol{Stutt}(\boldsymbol{Prop}(t'))$. ■

Example 5.6 Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit, $W^C \subseteq V^C$ be a set of external circuit variables, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system as described in Example 5.1 (i.e., $L^C(q) = \boldsymbol{Proj}(W^C)(q)$). Then for any trace $t = q_0 q_1 q_2 \ldots$ we have $\boldsymbol{Proj}(W^C)(t) = \boldsymbol{Stutt}(\boldsymbol{Prop}(t))$. ■

The next notion that is defined in [62] is that of a *persistent function*. In partial order state exploration, the subset of enabled variables whose transitions are selected to be explored from a state q should be independent, not only of all the remaining enabled variables in state q, but also of any variable that can become enabled in a state reachable from q by transitions of variables not in the selected set.

Definition 5.6 [Persistent functions and sets] [62]

Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be a circuit automaton. A function $\Delta: Q^C \rightarrow V^C$ is *persistent* if for every state $q \in Q^C$ the following holds: for all variables $v \in \Delta(q)$, (a) v is enabled in q ($v \in \boldsymbol{Enabled}(q)$), and (b) for any sequence of

state transitions t from q that changes variables in $V^C/\Delta(q)$ only, v is independent of all variables that can ever change (or become enabled) along t. $\Delta(q)$ is then called a *persistent set* of variables at q. ■

As we will see in the following subsection, for partial order reduction we require a selected set of enabled transitions that are explored from a given state to be persistent. Note that by definition of a persistent set, a set that includes all enabled variables of a state q would be persistent at q. In general, we can possibly have more than one persistent set of variables at each state q. The choice of the persistent set can however affect not only the structure of the explored state space, but also, the validity of partial order reduction.

The last definition of this section is that of a TMSCC.

Definition 5.7 [Terminal Maximal Strongly Connected Component, TMSCC]

Let $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ be an automaton (e.g., a sub-automaton of a circuit automaton FA^C). A subset $\hat{Q}^C \subseteq Q^C$ is a *strongly connected component of* FA^C iff within FA^C, all states in $\hat{Q}^C$ are reachable from all states in $\hat{Q}^C$. A strongly connected component in FA^C is *maximal* if it is not properly included in any other strongly connected component, and it is *terminal* if there is no outgoing transitions from it; i.e., there is no state not in $\hat{Q}^C$ that is reachable from a state in $\hat{Q}^C$. ■

By the above definition, a strongly connected component that is terminal is also maximal, and thus a TMSCC.

Definition 5.8 [Internal TMSCC] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit, and $W^C \subseteq V^C$ be a set of external circuit variables. A W^C-compatible subset $\hat{Q}^C \subseteq Q^C$ is called an *internal TMSCC* iff there exists a state $q \in \hat{Q}^C$ such that for any state q' that is reachable from q by any sequence of W^C-compatible states, we have $q' \in \hat{Q}^C$ and there exists a sequence of W^C-compatible states from q' back to q. Note that by the above definition, $\hat{Q}^C \subseteq Q^C$ is an internal TMSCC iff the above condition holds for all states $q \in \hat{Q}^C$. Moreover, this definition implies that $\hat{Q}^C$ is closed, in the sense that no sequence of W^C-compatible states from any state $q \in \hat{Q}^C$ can leave $\hat{Q}^C$. ■

5.1.2 Partial Order Reduction for Stuttering Equivalence

In partial order exploration of the state space of a system (e.g., a circuit), the transitions of only a subset of enabled variables at any state q are explored. By carefully choosing this subset, the properties of interest can be checked over the reduced state space instead of the full state space, without incurring any false positive or negative results. Under such conditions, the properly selected subset of variables at any state q is usually called an *ample* set, and denoted by $\boldsymbol{Ample}(q) \subseteq \boldsymbol{Enabled}(q)$.

We are particularly interested in a partial order reduction that would generate a reduced state space such that for each trace of the full state space, there exists a stuttering equivalent trace in the reduced one. Assuming that depth first search (DFS) is used for state space exploration, there exists a set of conditions for selection of ample sets

that guarantee stuttering equivalence between the full and reduced state spaces [62]. Note that during DFS, reaching a state that is already on the search stack implies closing a cycle.

Conditions 5.9 [Ample sets for stuttering equivalence] [62]

Let $FTS = \langle FA, AP, L \rangle$ be a finite transition system. To generate a sub-automaton $\tilde{FA}$ (using DFS) that is stuttering equivalent to FA, it is sufficient for ample sets of variables at each state $q \in Q$ to satisfy the following conditions.

C1: $\boldsymbol{Ample}(q)$ is a persistent set.

C2: If $\boldsymbol{Ample}(q) \neq \boldsymbol{Enabled}(q)$ (i.e.; q is not *fully expanded)*, then all variables in $\boldsymbol{Ample}(q)$ are invisible.

C3: For every TMSCC in $\tilde{FA}$, there exists at least one fully expanded state [81]. ■

To better understand condition **C1**, consider any subtrace t in FA that starts from state q. Two possible situations can happen [62]:

Case 1. Let v be the first variable from $\boldsymbol{Ample}(q)$ that changes along t. Then condition **C1** guarantees that v is independent of all the variables that change before it on t. Thus by applying the definition of independence repeatedly, all the transitions on t prior to the transition by v can be commuted with the transition by v. The result would be a trace t' starting from q whose first transition changes a variable v in $\boldsymbol{Ample}(q)$.

Case 2. If no transition by a variable in $\boldsymbol{Ample}(q)$ occurs on t, then by condition **C1**, any variable $v \in \boldsymbol{Ample}(q)$ is independent of all variables that change along t. Thus by definition of independence, one can form subtrace(s) t' starting from q by first firing any transition(s) changing variable v, and then consecutively firing the transitions of t.

The above two cases suggest that for any sequence of transitions t from a state q of FA, there exists a sequence t' that starts by the transitions of a variable from $\boldsymbol{Ample}(q)$.

Condition **C2** is to make the two subtraces t and t' in both of the above cases stuttering equivalent. First consider the case that $\boldsymbol{Ample}(q) \neq \boldsymbol{Enabled}(q)$. Then none of the variables in $\boldsymbol{Ample}(q)$ are visible. Now, moving a transition, that does not change any visible variable, to the beginning of trace t (**Case 1**), or inserting such a transition at the beginning of trace t (**Case 2**), would not change the propositional sequence of t, and as a result, t and t' will be stutter equivalent. In this case, **C1** and **C2** together suggest that t' would contain all the properties of t; confirming that it is sufficient to explore from q only transitions of $\boldsymbol{Ample}(q)$. Next, consider the case that $\boldsymbol{Ample}(q) = \boldsymbol{Enabled}(q)$; then we already explore all enabled transitions from state q.

When $\boldsymbol{Ample}(q) \neq \boldsymbol{Enabled}(q)$, the transitions of any variable $w \in \boldsymbol{Enabled}(q) / \boldsymbol{Ample}(q)$ are deferred (note that w stays enabled in any state that is reached from q by a transition of a variable from $\boldsymbol{Ample}(q)$). Condition **C3** is to prevent a situation in which $\boldsymbol{Ample}(q) \neq \boldsymbol{Enabled}(q)$ and the transition of a variable

$w \in \boldsymbol{Enabled}(q) / \boldsymbol{Ample}(q)$ can be deferred forever along a closed cycle of states containing state q. Note that if a variable that is enabled everywhere in a TMSCC does not appear in the selected persistent sets of any of those states, then there could exist a trace in the full state space that is not represented in the reduced state space by any stuttering equivalent trace, which can lead to incorrect verification results. By enforcing at least one state of each TMSCC of the reduced state space to be fully expanded, this latter situation would be avoided.

An ample set that satisfies the above conditions insists on exploring *all* enabled transitions from a state, *or* exploring the transitions of a persistent and invisible set of variables only, such that at least one state of each TMSCC in the reduced state space is fully expanded.

5.2 A First Partial Order Technique to Find Safe Abstractions

In this section, we present our first partial order reduction technique to find safe abstractions. This partial order technique constructs a sub-automaton of the circuit automaton such that, if it is projectable onto the set of external variables, its projection would be a safe abstraction of the circuit behavior.

We first show how the first step in finding a safe abstraction can be formalized as a search for a reduced state space that is stuttering equivalent with the full state space of the circuit. This would assert that the partial order reduction of section 5.1.2, that generates stutter equivalent reduced state spaces, can be used in finding safe abstractions.

Then we propose a set of conditions on external variables, and a strategy for selective search (when using DFS state space exploration) that satisfy all the ample set conditions for stutter equivalence (Conditions 5.9, Section 5.1.2). By construction, the resultant partial order reduction would automatically be a valid one, and thus can be used towards finding a safe abstraction.

5.2.1 Feasibility

In this subsection we show why and how partial order reductions can be used to derive safe abstractions.

Theorem 5.2 [Behavior projections and stutter equivalence] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit, $W^C \subseteq V^C$ be a set of external circuit variables, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. If $\tilde{FA}^C$ is any sub-automaton of FA^C that is stuttering equivalent with FA^C (with respect to FTS^C), then we have $\boldsymbol{Proj}(W^C)(\tilde{B}^C) = \boldsymbol{Proj}(W^C)(B^C)$. ■

Proof The above proposition is an immediate result of the following facts: because of stuttering equivalence of $\tilde{FA}^C$ and FA^C, for any trace $t \in B^C$ there always exists a trace $\tilde{t} \in \tilde{B}^C$ such that $\boldsymbol{Stutt}(\boldsymbol{Prop}(t)) = \boldsymbol{Stutt}(\boldsymbol{Prop}(\tilde{t}))$, and since $\boldsymbol{Stutt}(\boldsymbol{Prop}(t)) = \boldsymbol{Proj}(W^C)(t)$, we have $\boldsymbol{Proj}(W^C)(t) = \boldsymbol{Proj}(W^C)(\tilde{t})$. The latter result directly implies that $\boldsymbol{Proj}(W^C)(\tilde{B}^C) = \boldsymbol{Proj}(W^C)(B^C)$.

Corollary 5.3 [Safe abstractions and stutter equivalence] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit, $W^C \subseteq V^C$ be a set of external circuit variables, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. If partial order reduction for stuttering equivalence (Section 5.1.2) is used to construct a sub-automaton $\tilde{FA}^C$ of FA^C, and $\tilde{FA}^C$ is also projectable onto W^C, then $\tilde{B}^C_{W^C}$ is a safe abstraction of B^C over W^C. ■

Proof The above corollary is a direct implication of Proposition 5.2 and Corollary 2.4. It implies that partial order reduction for stuttering equivalence has indeed the *potential* of finding safe abstractions. For this purpose, we need to devise an strategy for selection of ample sets that satisfy Conditions 5.9 of Section 5.1.2.

Before we present our strategy for selection of ample sets, we present our general procedure to construct a sub-automaton of a circuit automaton using any ample set strategy.

Procedure 5.1 [Construction of circuit sub-automaton by partial order reduction] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any circuit, $W^C \subseteq V^C$ be a set of external circuit variables, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. Given any strategy for selection of ample sets for stuttering equivalent partial order reduction, a corresponding sub-automaton $\tilde{FA}^C = \langle A^C, V^C, \tilde{Q}^C, \tilde{\lambda}^C, \tilde{TR}^C, \tilde{\mu}^C, q_0^C \rangle$ of $FA^C = \langle A^C, V^C, Q^C, \lambda^C, TR^C, \mu^C, q_0^C \rangle$ is constructed using the following steps:

(i) let $\tilde{Q}^C = q_0^C$, and $\tilde{TR}^C = \varnothing$;

```
Construct_subautomaton(q, a, q'){
    Q̃^C = Q̃^C ∪ q';
    λ̃^C(q') = λ^C(q');
    T̃R^C = T̃R^C ∪ (q, a, q');
    μ̃^C(q, a) = S;
}
```

Fig. 5.4 Constructing partial order sub-automaton.
States and transitions are added to sub-automaton $\tilde{FA}^C$ as FA^C is being partially explored.

(ii) for any state transition $(q, a, q') \in TR^C$ that is explored from a state $q \in \tilde{Q}^C$, (i.e., $(q, a, q') \in \textbf{\textit{Ample}}(q)$) let $\tilde{Q}^C = \tilde{Q}^C \cup q'$, $\tilde{\lambda}^C(q') = \lambda^C(q')$, $\tilde{TR}^C = \tilde{TR}^C \cup (q, a, q')$, and $\tilde{\mu}^C(q, a) = S$. ■

We need to emphasize that usually $\textbf{\textit{Ample}}(q)$ is computed on the fly as a function of the partially constructed sub-automaton $\tilde{FA}^C$.

The above procedure is independent of any ample set strategy, or any search strategy for that matter (DFS or BFS); it simply specifies how to construct the sub-automaton as the state space of the circuit automaton is partially explored. Algorithm `Construct_subautomaton` of Figure 5.4, implements step (ii) of the above procedure.

In the following subsections, we first derive a set of criteria for external variables, based on conditions for ample sets in partial order reduction for stuttering equivalence (Conditions 5.9 of Section 5.1.2). These conditions, in turn, have a number of implications about the independence of circuit variables and persistency of sets of them. Finally, assuming that external variables satisfy our specified conditions, we present

our first strategy for selective search that satisfy the ample set conditions for stutter equivalence (Conditions 5.9 of Section 5.1.2).

5.2.2 Conditions on the Set of External Variables

In this section, we first introduce some new propositions and definitions. Then, based on the conditions for ample sets (Conditions 5.9 of Section 5.1.2), we derive a set of conditions for the set of external variables. The implications of these conditions are studied in the next subsection.

Proposition 5.4 [Visibility of external variables] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit, $W^C \subseteq V^C$ be a set of external circuit variables, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. Then all variables in W^C are visible. Moreover, any variable in $V^C - W^C$ that is not simultaneous with any variable in W^C is invisible. ■

The above proposition directly follows from the definitions of visibility and simultaneity. It implies that to satisfy our ample set conditions (Conditions 5.9 of Section 5.1.2), if a state q is not fully expanded, then we must have $\boldsymbol{Ample}(q) \cap W^C = \varnothing$.

As seen in Example 5.5, if W^C is such that no pair of variables $v \in W^C$ and $w \in V^C - W^C$ are simultaneous, then any varaible $w \in V^C - W^C$ would be an invisible variable, and thus can be included in $\boldsymbol{Ample}(q)$ of any state q that is not fully expanded. If in addition, W^C contains any variable v that is dependent on any other variable, then any variable $w \in V^C - W^C$ would be independent of all other circuit

variables. Under these latter conditions, consider any state q, and any variable $v \in V^C - W^C$ that is enabled at state q; then $\{v\}$ is always a persistent set. The reason is that there is no sequence of state transitions from q that, without changing v, can lead to a state at which a variable w that depends on v can become enabled. The reason: no variable w that depends on v exists.

Based on the above observations, we have devised a set of conditions on the set of external variables that would then lead to a trivial strategy for selection of ample sets.

Definition 5.10 [Closure under failure-free dependence]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit, $W^C \subseteq V^C$ be its set of external variables, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \textbf{\textit{Proj}}(W^C)(q)$. Assume that W^C includes the subset of circuit variables $V_P^C \subseteq V^C$ that are *prime dependent*. That is, for all signals $v \in V^C$, if there exists any variable w such that v and w are dependent under the prime dependency conditions, then we must have $v, w \in W^C$. Assume that W^C is also *closed* under the simultaneity dependency condition in the following sense: for all variables $v \in W^C$, any variable $w \in V^C$ that can ever change simultaneously with v must also be included in W^C (i.e., $w \in W^C$). Then, we call such a set of external signals *closed under failure-free dependence*. ■

Theorem 5.5 [Persistency and invisibility by closure under failure-free dependence] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a circuit, $W^C \subseteq V^C$ be a set of external variables that is closed under failure-free dependence, and

$FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \textbf{\textit{Proj}}(W^C)(q)$. Then for any state q and any enabled internal signal $v \in (\textbf{\textit{Enabled}}(q) \cap A^C) - W^C$, a persistent and invisible set at q is $\boldsymbol{P}(q) = \{v\} \cup \{w | w \in \textbf{\textit{Enabled}}(q), w \text{ can simultaneously change with } v\}$. ■

Proof (Sketch) First, we show that $\boldsymbol{P}(q)$ is invisible. If there exists a variable $w \in \boldsymbol{P}(q)$ that is visible, then it must be capable of simultaneously changing with a variable $u \in W^C$. But then, since W^C is closed under failure-free dependence, we must have $w \in W^C$, and by the same token, we must have $v \in W^C$ which is a contradiction. As a result, we must have $\boldsymbol{P}(q) \subseteq V^C - W^C$ is an invisible set. Next, we show that $\boldsymbol{P}(q)$ is persistent. If it is not, then there must exist a variable $u \notin \boldsymbol{P}(q)$ that is dependent on a variable $w \in \boldsymbol{P}(q)$, and u can become enabled through a sequence of transitions not involving $\boldsymbol{P}(q)$. But, since W^C is assumed to be closed under failure-free dependence, no such pairs of variables, u and w can ever be dependent, or otherwise $u, w \in W^C$ which is a contradiction. Thus $\boldsymbol{P}(q)$ is persistent. ■

Note that by closure under failure-free dependence, a set of external variables might include independent variables as well. Also, under those conditions, the set of internal variables $V^C - W^C$ can include pairs of simultaneous variables, if they are not dependent or simultaneous to any external variables. Another interesting results of this condition is that if any I/O signal of any module is made external, then all internal variables of the module that are simultaneous with it must also be made external, together with any other I/O of the module that is, recursively, simultaneous with them.

In the following subsection, we will present our procedure for construction of stuttering equivalent circuit sub-automaton (for a failure-free circuit) and the corresponding strategy for selection of ample sets.

5.2.3 A First Partial Order Reduction

We are now ready to introduce our first algorithm for partial order exploration of the state space of a failure-free partitioned circuit; a selective search that satisfies the ample set conditions for stuttering equivalent partial order reduction.

Algorithm 5.2 [`DFS_1`, a first algorithm for partial order reduction]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a failure-free circuit, $W^C \subseteq V^C$ be a set of external circuit variables that is closed under failure-free dependence, $E^C = A^C \cap W^C$, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. Algorithm `DFS_1` of Figure 5.5 is a DFS algorithm that constructs a sub-automaton $\tilde{FA}^C = \langle A^C, V^C, \tilde{Q}^C, \tilde{\lambda}^C, \tilde{TR}^C, \tilde{\mu}^C, q_0^C \rangle$ of FA^C that is stuttering equivalent with FA^C; i.e., its ample set strategy satisfies Conditions 5.9 of Section 5.1.2. ■

Partial order reduction starts by calling procedure `Partial_Order` of Figure 5.6, that would call `DFS_1` for (each of) the initial state(s) of the circuit.

Before we prove that Algorithm `DFS_1` indeed constructs a stuttering equivalent sub-automaton of the circuit automaton of a failure-free circuit, we explain how the

```
1  DFS_1(i){ /* DFS on circuit block i */
2     Pop(q);
3     if q ∈ Q̃^C or Enabled(q) = ∅ then
4        return;
5     /* try to explore a single internal transition of block i
6        to a state that is not on the search stack */
7     for each v ∈ (Enabled(q) − W^C) ∩ V^C_{E,i} {
8        /* v is an enabled internal signal of block i */
9        if (q, v, q') ∈ TR^C and q' ∉ Stack then {
10          Construct_subautomaton(q, v, q');
11          Push(q');
12          DFS_1(i);
13          return;
14       }
15    }
16    /* if all internal transitions of block i lead to states on
17       the search stack, move on to the next block i + 1 and try
18       to explore an internal transition of that block */
19    if i ≠ r^C_E then { /* not the last block */
20       Push(q);
21       DFS_1(i + 1);
22       return;
23    }
24    /* if this was the last block, then fully expand state q */
25    else {
26       /* */
27       /* explore all transitions from state q */
28       for each v ∈ Enabled(q) {
29          /* v is any enabled signal */
30          for each (q, v, q') ∈ TR^C {
31             Construct_subautomaton(q, v, q');
32             /* continue the DFS search from each
33                un-explored state q' */
34             if q' ∉ Q̃^C then {
35                Push(q');
36                DFS_1(1);
37             }
38          }
39       }
40    }
41 }
```

Fig. 5.5 Algorithm DFS_1.
An algorithm for generation of a stutter equivalent reduced state space of an SI circuit.

```
1  Partial_Order(){
2     Q̃^C = T̃R^C = ∅;
3     for each initial state q_0 {
4        Push(q_0);
5        DFS_1(1);
6
7  }
```

Fig. 5.6 Partial order reduction using Algorithm `DFS_1`.

algorithm works. The circuit blocks are numbered from 1 to r_E^C. Each recursive call of the algorithm receives as an argument the number of a circuit block which is to be searched for an ample (internal) transition. The search stack is initialized with an initial state, and `DFS_1` is called with the first circuit block. Then, `DFS_1` repeatedly does the following: assuming that the current state q (popped from the stack) is not previously explored, if there exists any transition $(q, v, q') \in TR^C$ by an internal signal of current block i such that state q' is not on the search stack, that transition is explored and the search is continued from state q' and within circuit block i; otherwise, unless this is the last block, the search is continued from state q and within the next circuit block $i + 1$; if on the other hand, this is the last circuit block, then q is fully explored and the search is continued from each of the reached states and from within circuit block 1. As any DFS search, previously explored states that are already in the reduced state space, or states not having any outgoing transitions, are not further processed once they are popped from the stack. For reduced state spaces of possibly smaller sizes, one can enforce exploration of internal transitions to previously explored states, assuming that they are not on the search stack.

A more intuitive analysis of the behavior of Algorithm `DFS_1` is as follows. The goal of the algorithm is to direct the circuit into a *non-transient* state where the internal variables of the circuit are either stabilized or involved in a non-transient oscillation. To do this, the algorithm successively directs each circuit block $M^C_{E,i}$ into a *local* non-transient state where the internal variables of the block are either stabilized or involved in a non-transient oscillation. In directing a circuit block $M^C_{E,i}$ from a state q^i_0 to its non-transient state, at any intermediate state q if there exists any (arbitrary) internal transition to any state q' that is not on the DFS stack (and hence does not close a cycle) then our partial order explores only that transition by letting $\boldsymbol{Ample}(q)$ to be a singleton set containing the corresponding variable. Thus, the goal is to explore, from q^i_0 a single interleaving of internal signal transitions leading to a state at which all internal signals of the block are stabilized (or more generally, have made all of their transitions); however, in the presence of internal oscillations, an arbitrary internal signal transition might lead to a state on the DFS path, and close a cycle (oscillation). In such a case, a valid partial order should avoid a case in which a variable that is enabled everywhere along a cycle is never included in any ample set. This is required for the satisfaction of condition **C3** of Section 5.1.2 for ample sets. That is why, in stabilizing the internal signals of a circuit block, Algorithm `DFS_1` tries to avoid closing cycles as much as possible. Eventually, a state q is reached at which either no internal signal of circuit block $M^C_{E,i}$ is enabled, or all transitions of such signals lead to states that are on the DFS stack. It is easy to see that any such state q would be a local non-transient state of block $M^C_{E,i}$. At this point, Algorithm `DFS_1` starts directing circuit block

$M^{C}_{E,\,i+1}$ to its local non-transient states, starting from state $q_0^{i+1} = q$. Once all circuit blocks are successively directed to their local non-transient states, and (as can be proven) the whole circuit is in a global non-transient state, all enabled transitions from such a state are explored, and the DFS search is continued from each state q that can be reached by such transitions, such that q was not previously explored.

Proof [Algorithm 5.2, `DFS_1`, generates a stuttering equivalent sub-automaton of a failure-free circuit]

(Sketch) We need to show that the selectively explored sets of transitions in algorithm `DFS_1` satisfy the ample set conditions of Section 5.1.2.

C1: We note that from each state q that is visited by `DFS_1`, either a single internal transition (lines 7-15) or all enabled transitions are explored (lines 28-39). However, since the set of external variables is closed under failure-free dependence, both of the above situations characterize a persistent set, and thus persistency condition **C1** for an ample set is satisfied.

C2: Since external transitions are explored only from states that are fully explored, visibility condition **C2** is also satisfied by the selective search of `DFS_1`.

C3: We note that unless a state is fully explored, `DFS_1` does not explore any of its enabled transitions to states that are on the search stack. On the other hand, `DFS_1`, as an ordinary DFS algorithm (that does not re-explore states), can close a cycle in the searched space *only* by exploring transitions to states on the search stack [24]. As a result, *all* cycles in the reduced state space that is explored by `DFS_1` have a state that

is fully explored. Since any TMSCC consists of states with cycle(s) between any pair of them, if all cycles of the reduced state space have a state that is fully explored, then all TMSCCs of the reduced state space (if there exists any) will also have a state that is fully explored. Thus, the selective search of `DFS_1` also satisfies condition **C3** for ample sets.

Since all the three conditions are met, `DFS_1` indeed generates a stuttering equivalent reduced state space for a failure-free circuit (i.e., if the internal variables are indeed failure-free independent of all other variables). ■

Algorithm `DFS_1` simply generates a reduced state space that is stuttering equivalent to the full state space of a failure-free circuit. To find a safe abstraction of the behavior of a circuit, whether it is failure-free or not, the sub-automaton that is constructed by Algorithm `DFS_1` (automaton of the partially explored state space) has to be projected onto the set of external variables. We will end this section without presenting any algorithm for projection of the sub-automaton constructed by `DFS_1`. The reason is that such an algorithm would not be a simple one, and since in practice we will not use `DFS_1` to find safe abstractions, our efforts for devising or presenting such an algorithm would be wasted. In the following section, we present an enhanced algorithm for partial order reduction that is a close representative of what we use in practice. The enhanced algorithm has automatically provided a way for simple on-the-fly projection of the constructed sub-automaton that would be discussed in the next section.

5.3 An Enhanced Partial Order Reduction

In this section, we present an enhanced algorithm for stuttering equivalent partial order reduction (for a failure-free circuit) that has an embedded procedure to check the projectability of the partial order sub-automaton and compute its projection, on-the-fly. The new algorithm is thus capable of directly finding a safe abstraction. This enhanced algorithm, instead of the authentic DFS used in algorithm `DFS_1`, uses what we call *parallel* DFS. Parallel DFS can be regarded as a special kind of breadth first search (BFS), which can in turn be implemented using symbolic techniques and BDDs. In this section, we first present the new algorithm, and then prove its correctness in finding a safe abstraction in the following way. We prove that the reduced state space (sub-automaton) generated by the algorithm is stuttering equivalent to the full state space, if the circuit is failure-free. This is proven by showing that ample set conditions are satisfied by the algorithm's selective search. We also prove that if the circuit is failure-free, then the embedded procedure for on-the-fly projection finds an automaton projection of the constructed sub-automaton *iff* it is projectable, and otherwise it aborts the algorithm. In the same regard, we also prove that if the circuit is not failure-free and the on-the-fly projection procedure does not abort, then for all the traces of its generated automaton, there exist a stuttering equivalent trace in the reduced state space. These properties are proven based on properties of failure-free independence. Together, these results would imply the correctness of the overall approach in finding a safe abstraction.

```
1  Safe_abstraction()
2     for each initial state q0 {
3        Push(q0, q0);
4        DFS_2(q0, 1);
5     }
6  }
```

Fig. 5.7 Finding a safe abstraction using Algorithm `DFS_2`.

In Section 5.3.3, a further optimized version of the new algorithm is presented that can further speed up and reduce the size of the explored state space.

5.3.1 A Complete Solution to Finding a Safe Abstraction

In this section we present a new partial order algorithm incorporating independent DFS searches that can be performed in parallel.

Algorithm 5.3 [`DFS_2`, an enhanced algorithm for finding safe abstractions]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a failure-free circuit, $W^C \subseteq V^C$ be a set of external circuit variables that is closed under failure-free dependence, $E^C = A^C \cap W^C$, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. Algorithm `DFS_2` (Figure 5.8) is a *parallel* DFS algorithm that constructs a sub-automaton $\tilde{FA}^C = \langle A^C, V^C, \tilde{Q}^C, \tilde{\lambda}^C, \tilde{TR}^C, \tilde{\mu}^C, q_0^C \rangle$ of FA^C that is stuttering equivalent with FA^C; i.e., its ample set strategy satisfies Conditions 5.9 of Section 5.1.2. Moreover, its embedded procedure `Construct_projection` (Figure 5.10) finds an automaton projection of the constructed sub-automaton *iff* it is project-

```
1  DFS_2( p, i ) /* DFS on the stack of state p and circuit block i */
2     Pop ( q, p ); /* pop a state q from the stack of state p */
3     if Enabled(q) = ∅ then
4        return;
5     /* try to explore a single internal transition of block i
6        to a state that is not on the stack of p */
7     for each v ∈ (Enabled(q) – W^C) ∩ V^C_{E,i} {
8        /* v is an enabled internal signal of block i */
9        if (q, v, q') ∈ TR^C and q' ∉ Stack(p) then {
10          Construct_subautomaton( q, v, q' );
11          Push( q', p );
12          DFS_2( p, i );
13          return;
14       }
15    }
16    /* if all internal transitions of block i lead to states on
17       the search stack of p, move on to the next block i + 1 and
18       try to explore an internal transition of that block */
19    if i ≠ r^C_E then { /* not the last block */
20       Push( q, p );
21       DFS_2( p, i + 1 );
22       return;
23    }
24    /* the end of the DFS path from state p is reached */
25    else {
26       Construct_projection( q );
27       Explore_internal_trans( p, q );
28       for each v ∈ Enabled(q) ∩ W^C {/* explore external trans */
29          /* v is an enabled external signal */
30          for each (q, v, q') ∈ TR^C {
31             Construct_subautomaton( q, v, q' );
32             /* initiate a new DFS search from each
33                un-explored state q' */
34             if q' ∉ Q̃^C then {
35                Push( q', q' );
36                DFS_2( q', 1 );
37             }
38          }
39       }
40    }
41 }
```

Fig. 5.8 Algorithm DFS_2.
An enhanced algorithm for generation of a stutter equivalent reduced state space of an SI circuit. It fully expands the terminal states of each independent DFS search.

```
Construct_projection (q){
   Temp = ∅;
   for each (q, v, q') ∈ TR^C s.t v ∈ Enabled(q) ∩ W^C
      Temp = Temp ∪ Proj(W^C)(q, v, q');
   if Proj(W^C)(q) ∈ Q̃_V^C then
      if Temp ≠ {(r, a, s) ∈ T̃R_V^C | r = Proj(W^C)(q)} then
         exit ("Not a safe abstraction");
      else
         return;
   else {
      Q̃_V^C = Q̃_V^C ∪ Proj(W^C)(q);
      T̃R_V^C = T̃R_V^C ∪ Temp;
   }
}
```

Fig. 5.10 On-the-fly projection and projectability check of the sub-automaton.

able, and otherwise it aborts the algorithm. Finally, if procedure `Construct_projection` does not abort the algorithm, then the behavior of its output automaton is always a safe abstraction of the circuit behavior, even when the circuit is not failure-free. ■

To find a safe abstraction, procedure `Safe_abstraction` of Figure 5.7 is called, which would call `DFS_2` for (each of) the initial state(s) of the circuit. For on-the-fly projection and projectability check of the constructed sub-automaton, `DFS_2` calls procedure `Construct_projection` of Figure 5.10.

Before we prove the above mentioned properties of Algorithm `DFS_2`, we explain how the algorithm works. Algorithm `DFS_2`, instead of a single stack, utilizes multiple DFS stacks that are initiated either from the initial state(s) or from states that are entered after an external variable transition. These stacks are identified by the (label of

```
1  Explore_internal_trans( p, q ) {
2     for each v ∈ Enabled(q) − W^C and (q, v, q') ∈ TR^C {
3        Construct_subautomaton( q, v, q' );
4        /* if M^C_{E,i} is the circuit block that drives signal v
5           then explore the same sequence of signal transitions
6           that was previously explored in block M^C_{E,i} along
7           the DFS path of the stack of p */
8        if v ∈ H^C_{E,i} then {
9           if s ∈ Stack(p) s.t. Proj(H^C_{E,i})(s) = Proj(H^C_{E,i})(q') then {
10             repeat {
11                /* s' is on top of s on the stack of p */
12                s' = Top(s, p);
13                if (s, w, s') ∈ ~TR^C and w ∈ H^C_{E,i} then {
14                   if (q', w, q'') ∈ TR^C then {
15                      Construct_subautomaton( q', w, q'' );
16                      q' = q'';
17                      s = s';
18                   }
19                else
20                   /* at this point we should have q' = q */
21                   break; /* quit the repeat loop */
22             } until 0;
23          }
24       }
25    }
26 }
```

Fig. 5.9 Algorithm `Explore_internal_trans`.
An algorithm for exploration of internal transitions from the terminal states of independent DFS paths of Algorithm `DFS_2`.

the) state from which they were initiated. The recursive function, `DFS_2` has thus two parameters: the first parameter is the stack identifier, and the second one is the number of the circuit block from which the DFS search has to be continued (similar to the case of Algorithm `DFS_1`). Each DFS stack is associated with an independent DFS search of the circuit state space that is started from the *initial state* of the stack (the state at which the stack was initiated and which identifies the stack), and ends at a state from

which all enabled transitions are explored, called the *terminal state* of that DFS. Each independent DFS search goes through all the r_E^C circuit blocks in order, and explores in each block a *maximal* cycle-free sequence of internal signal transitions of the block, by never exploring a transition to a state that is on its own stack. Eventually, each independent DFS search reaches a *terminal* state in the last circuit block from which either no internal transition is possible, or the transition of any internal signal would close a cycle of signal transitions in the *local* state of the circuit block that drives that signal. At the terminal state of each DFS path, procedure `Explore_internal_trans` of Figure 5.9 explores all enabled internal transitions, and from each of the reached states it finds a sequence of internal transitions back to the same terminal state. All enabled external transitions of the terminal state are also explored and a new DFS search is initiated from each new state that is reached. Thus each DFS search explores a cycle-free sequence (path) of states from the initial state of its stack to a terminal state at which all enabled transitions are explored.

The DFS searches of Algorithm `DFS_2` are independent in the sense that they are free to re-explore states that were previously explored (added to the reduced state space) by preceding DFS searches. This is different from the authentic DFS search of Algorithm `DFS_1` which avoids re-exploring states that are already in the reduced state space (compare lines 3 of the two algorithms). This redundancy of `DFS_2` is only to force each independent DFS path to *complete* exploration of a maximal cycle-free sequence of internal transitions before it is terminated, even if parts of this path (sequence) overlap with paths that were explored previously. The same thing is also

true about the selective search of procedure `Explore_internal_trans`; i.e., it is free to re-explore states of the reduced state space.

Comparing the two algorithms `DFS_1` and `DFS_2`, line by line, one can observe that they are quite similar, with the following differences (a) `DFS_2` uses local stacks that are initiated at the initial state(s) of the circuit or after each external signal transition to a new state, while `DFS_1` uses a global stack that is initiated just once, (b) `DFS_2` might re-explore states, while `DFS_1` avoids that (compare lines 3 of the two algorithms), (c) after exploring maximal sequences of internal transitions (i.e., at the terminal states of independent DFS paths), `DFS_2` uses a directed independent DFS search to explore internal transitions (compare lines 27 and 28 of the two algorithms), in the sense that the explored paths are led back to the terminal state, and (d) `DFS_2` also carries a procedure for on-the-fly projection of the reduced state space (line 26).

One major consequence of the first three above mentioned differences between algorithms `DFS_2` and `DFS_1`, in terms of the structure of the reduced state spaces that they create, is the possibility of existence of extra cycles in the state space generated by `DFS_2` that are not fully expanded at the state that closes the cycle. These cycles can be the result of exploring an internal transition to a previously explored state that is not on the current local stack of Algorithm `DFS_2`, but is on the global path from the lastly explored initial state of the circuit to the initial state of the current stack. Such cycles reside on the single stack of Algorithm `DFS_1`, without the closing state being fully explored, although the cycle does have a fully expanded state. An example of this case is illustrated in Figure 5.11 where the DFS path from state q_3 to q_4 does not close any

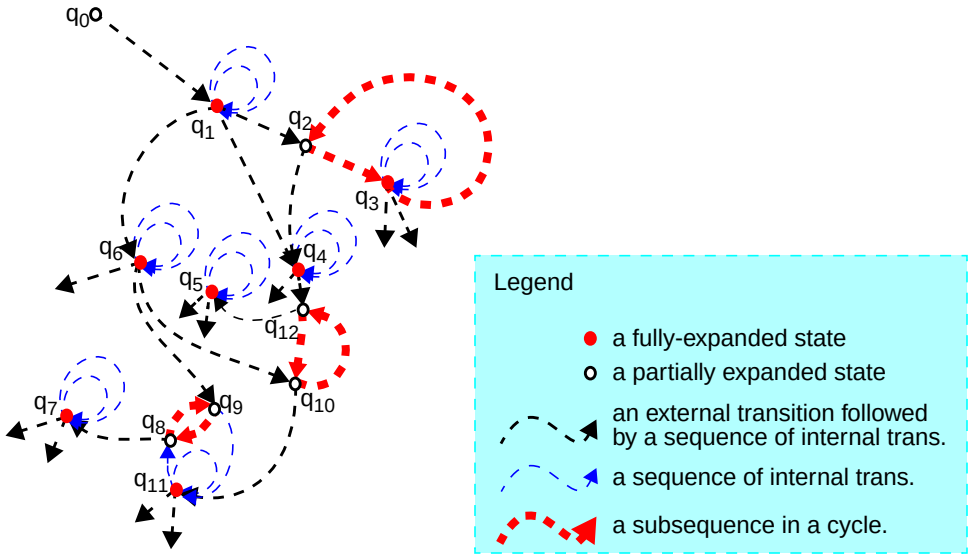


Fig. 5.11 Algorithm `DFS_2` can create additional cycles.
The reduced state space generated by `DFS_2` can have cycles that are not possible by Algorithm `DFS_1`.

cycle on itself; however, it closes a cycle on the global stack that starts from initial state q_0 and passes through q_1, q_2, q_3, q_2, etc. The cycle is closed at state q_2 which is an ancestor of state q_3, the initial state of the local stack. Although the state at which the cycle is closed (q_2) is not fully expanded, the cycle does have a state that is fully expanded, i.e., q_3. Algorithm `DFS_2` can also create cycles of internal transitions that are not on a global DFS path, but are created by independently explored paths (DFS paths or paths of internal transitions explored by `Explore_internal_trans`) that happen to cross each other more than once, in certain ways. Such cycles might have no state that is fully expanded. Two example of this case are illustrated in Figure 5.11. The cycle containing states q_{10} and q_{12} is created by two independent DFS paths starting from states q_4 and q_6, respectively. The cycle containing state q_8 and q_9 is created by

a DFS path starting from state q_6 crossing a cycle of internal transitions from state q_{11} to itself. In both of these examples, no state on the cycles is fully expanded.

In the rest of this section, we will prove that Algorithm `DFS_2` indeed generates stuttering equivalent reduced state spaces for failure-free circuit by showing that it satisfies all the ample set conditions of Section 5.1.2. We will show that none of the additional cycles that can be introduced in the reduced state space generated by Algorithm `DFS_2` are TMSCCs, and thus the fact that they might not have a fully expanded state is harmless. Moreover, we show that the persistency condition for the selected set of transitions from each state explored by Algorithm `DFS_2` is satisfied. We also prove the ability of the embedded procedure `Construct_projection` in finding a safe abstraction, if one exists.

5.3.2 Proof of Correctness

To prove the correctness of Algorithm `DFS_2` for generation of a stuttering equivalent partial order sub-automaton of circuit automaton, we need to show that it satisfies the three ample set conditions of Section 5.1.2.

Procedure `Explore_internal_trans` that is illustrated in Figure 5.9 performs a selected search that is different in style from the one within the body of the Algorithm.

Before we explain how procedure `Explore_internal_trans` works, we make the following observations about any path from the initial state p to the terminal state q of a local DFS. At line 27 of Algorithm `DFS_2`, where terminal state q of the stack of

state p is going to be fully expanded, the stack of state p contains a (cycle-free) sequence of states $t = q_{n_0}^0 q_1^1 q_2^1 \ldots q_{n_1}^1 q_1^2 q_2^2 \ldots q_{n_2}^2 \ldots\ldots q_1^m q_2^m \ldots q_{n_m}^m$ from state p to state q, where we have $m = r_E^C$, $q_{n_0}^0 = p$, and $q_{n_m}^m = q$. This sequence consists of r_E^C possibly empty subsequences of states $t_i = q_{n_{i-1}}^{i-1} q_1^i q_2^i \ldots q_{n_i}^i$, $1 \le i \le r_E^C$. Each t_i is a sequence of $n_i + 1$ unique states (because t, and hence t_i, are cycle-free paths of states), and only internal signals of circuit block $M_{E,i}^C$ change along t_i. Moreover, the last state $q_{n_i}^i$ of any subsequence t_i has the property that the transition of any internal signal of block $M_{E,i}^C$ from state $q_{n_i}^i$ would lead to a state of t_i. Finally, since no internal signal of block $M_{E,i}^C$ changes along t after state $q_{n_i}^i$ is explored, for all $1 \le i \le r_E^C$ we have $\boldsymbol{Proj}(H_{E,i}^C)(q) = \boldsymbol{Proj}(H_{E,i}^C)(q_{n_i}^i)$, and for any transition (q, v, q'), if $v \in H_{E,i}^C$ then there exists a state $s \in t_i$ such that $\boldsymbol{Proj}(H_{E,i}^C)(s) = \boldsymbol{Proj}(H_{E,i}^C)(q')$. These properties of trace t and its subtraces t_i are the result of the particular way that Algorithm `DFS_2` explores internal transitions of the circuit before reaching a terminal state of the stack of state p. Intuitively, along trace t, at any state $q_{n_i}^i$ and beyond (e.g., at terminal state q), each circuit block $M_{E,i}^C$ is in a maximal *local* cycle of states, and no matter what transitions happen outside of block $M_{E,i}^C$, and as long as no external transitions occur, any transition by an enabled internal signal of block $M_{E,i}^C$ will take the circuit to a state that was *locally* visited before (in block $M_{E,i}^C$), along t_i.

At this point we are ready to explain how procedure `Explore_internal_trans` works. For each internal transition (q, v, q') from a terminal state q to a state q', by an internal signal v of a circuit block $M_{E,i}^C$ ($v \in H_{E,i}^C$), `Explore_internal_trans` explores transition (q, v, q') followed by a sequence of transitions from q' back to

state q. This is achieved by exploring the same sequence of *signal* transitions that were previously explored to state $q^i_{n_i}$ from a state s along subsequence $t_i = q^{i-1}_{n_{i-1}} q^i_1 q^i_2 \ldots q^i_{n_i}$, where s has the property that $\boldsymbol{Proj}(H^C_{E,i})(s) = \boldsymbol{Proj}(H^C_{E,i})(q')$. Existence of such a state s and $\boldsymbol{Proj}(H^C_{E,i})(q^i_{n_i}) = \boldsymbol{Proj}(H^C_{E,i})(q)$ are guaranteed by properties of trace t that we had just discussed.

The above observation about the terminal states of DFS paths of Algorithm `DFS_2` can be summarized in the following lemma.

Lemma 5.6 [Internal transitions from terminal states of DFS paths] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a failure-free circuit, $W^C \subseteq V^C$ be a set of external circuit variables that is closed under failure-free dependence, $E^C = A^C \cap W^C$, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. Let Algorithm `DFS_2` be used for partial exploration of the state space of C. At line 27 of Algorithm `DFS_2`, where state q would be the terminal state of the DFS path started from state p, state q has the following property: for any state q' that is reachable from state q by the transition of an enabled internal signal $v \in \boldsymbol{Enabled}(q) - W^C$ (i.e., $(q, v, q') \in TR^C$), there exists a sequence of internal transitions from state q' back to state q. ■

Each of the sequences of states that are explored by procedure `Explore_internal_trans` from terminal state q, close a cycle at that state. However,

all such cycles (even if associated with a TMSCC in the reduced state space) have a common state, q, that is fully expanded.

Having explained the operation of procedure `Explore_internal_trans`, we are now ready to prove that the reduced state space explored by Algorithm `DFS_2` is stuttering equivalent with the full state space, if the circuit is failure free (i.e., the internal signals are actually failure-free independent).

Theorem 5.7 [Algorithm 5.3, `DFS_2`, generates a stuttering equivalent sub-automaton] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a failure-free circuit, $W^C \subseteq V^C$ be a set of external circuit variables that is closed under failure-free dependence, $E^C = A^C \cap W^C$, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. Then Algorithm `DFS_2` constructs a sub-automaton $\tilde{FA}^C = \langle A^C, V^C, \tilde{Q}^C, \tilde{\lambda}^C, \tilde{TR}^C, \tilde{\mu}^C, q_0^C \rangle$ of FA^C that is stuttering equivalent with FA^C. ■

Proof (Sketch) To prove the correctness of Algorithm `DFS_2` in generating a reduced state space that is stuttering equivalent with the full state space of the partitioned circuit, we show that it satisfies all three conditions for selection of ample sets.

C1: This condition is satisfied by the selective search of Algorithm `DFS_2` for the following reasons. At each state q of the reduced state space that is generated by Algorithm `DFS_2`, the set of enabled variables whose transitions are explored, denoted by $\boldsymbol{Ample}(q) \subseteq \boldsymbol{Enabled}(q)$ has (only) one of the following forms:

(a) $\boldsymbol{Ample}(q) = \{v\}$, where $v \in \boldsymbol{Enabled}(q) - W^C$. This condition happens when state q is explored just once, or when it is explored multiple times (along different paths), but each time the same internal transition from it is explored.

(b) $\boldsymbol{Ample}(q) \subseteq \boldsymbol{Enabled}(q) - W^C$. This condition happens when state q is explored multiple times (along different paths), but not the same internal transitions are explored each time.

(c) $\boldsymbol{Ample}(q) = \boldsymbol{Enabled}(q)$. This condition happens when state q is fully expanded at the terminal state of at least one DFS path. Note that it is possible for a state to be fully expanded by one DFS search, while other independent DFS searches, or `Explore_internal_trans`, might have explored only single transitions from that state.

The set $\boldsymbol{Ample}(q)$ is thus a persistent set in each of the above situations. The reason is that any non-empty subset of internal transitions is always a persistent set (since W^C is assumed to be closed under failure-free dependence), and $\boldsymbol{Enabled}(q)$ is also always a persistent set.

C2: This condition is satisfied by the selective search of Algorithm `DFS_2` because the only place that any visible (i.e., external) transition is explored by that algorithm is when a state is fully expanded.

C3: To prove that this condition is satisfied by Algorithm `DFS_2`, we need to show that any TMSCC in the reduced state space generated by that algorithm has a state that is fully expanded. A TMSCC in the reduced state space $\tilde{FA}^C$ is a subset $\hat{Q}^C \subseteq \tilde{Q}^C$ such

that (a) each state $q \in \hat{Q}^C$ can reach any other state $q' \in \hat{Q}^C$ through a sequence of states in $\hat{Q}^C$, *and* (b) there exists *no* transition from any state $q \in \hat{Q}^C$ to any state $q' \notin \hat{Q}^C$. By condition (a) above, for each pair of states $q, q' \in \hat{Q}^C$, there exists a cycle of states within $\hat{Q}^C$ that contains the two states. The above two conditions also imply that if any state $q \in \tilde{Q}^C$ belongs to a TMSCC, then any state that is reachable from state q also belongs to the same TMSCC. To prove condition **C3**, it is sufficient to prove for *every* state q of the reduced state space that if q belongs to a TMSCC, then that TMSCC has a state that is fully expanded (note that by definition of a TMSCC, each state can belong to at most *one* TMSCC). It is thus sufficient to show that from *every* state of the reduced state space, there exists a path to a state that is fully expanded. But this is exactly what is enforced by Algorithm `DFS_2`; i.e., any independent DFS search is stretched to a state that is fully expanded, and the states that are explored by procedure `Explore_internal_trans` also have paths to the originating state that is also fully expanded. Thus, Algorithm `DFS_2` indeed satisfies condition **C3**. ■

To prove that procedure `Construct_projection` that is embedded in `DFS_2` can find a safe abstraction, we need to first present some lemmas.

Lemma 5.8 [The terminal state of any DFS path belongs to an internal TMSCC] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a failure-free circuit, $W^C \subseteq V^C$ be a set of external circuit variables that is closed under failure-free dependence, $E^C = A^C \cap W^C$, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with

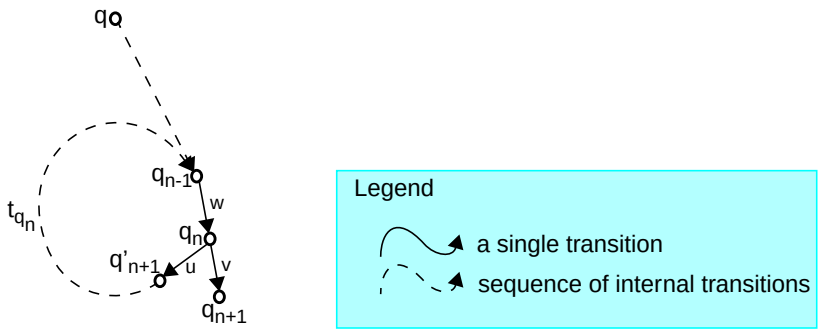


Fig. 5.12 Illustration of the inductive case of Lemma 5.8.

$L^C(q) = \textbf{\textit{Proj}}(W^C)(q)$. Also let Algorithm `DFS_2` be used for partial exploration of the state space of C. Then the terminal state q of any DFS path, that is started from any state p, belongs to an internal TMSCC of FA^C. That is--by definition of an internal TMSCC--from any state q' that is reachable from q through a sequence of internal transitions, there exists a sequence of internal transitions from q' back to q, and all such states q' belong to the internal TMSCC. ■

Proof (Sketch) We prove this lemma by first showing that for all $n \geq 1$, and all sequences of internal transitions $t_n = qq_1q_2\ldots q_{n-1}q_n$ from any terminal state q, there exists a sequence of internal transitions from q_n to q_{n-1}. Since this is a recursive property, it also implies that any state q_i on t_n, $1 < i \leq n$, can reach its preceding state, q_{i-1}, through a sequence of internal transitions (note that $q_0 = q$). Thus, it also implies that any q_i on t_n, and in particular q_n, can reach q through some sequence of internal transitions.

We prove the above property using an induction on the length n of the sequence of internal transitions from a terminal state q to any other state q_n.

Basis step: If q_1 is any state reached from terminal state q by a sequence of internal transitions of length one; i.e., $(q, v, q_1) \in TR^C$ and $v \in \textbf{\textit{Enabled}}(q) - W^C$, then by Lemma 5.6 there exists a sequence of internal transitions from q_1 back to q.

Inductive hypothesis: let q_n be any state that is reachable from terminal state q by any sequence of internal transitions $t_n = qq_1q_2\ldots q_{n-1}q_n$ of length n, and assume that there exists a sequence of internal transitions from q_n to q_{n-1}.

Inductive step: for any state q_{n+1} that is reachable from q by any sequence of internal transitions $t_{n+1} = qq_1q_2\ldots q_{n-1}q_nq_{n+1}$ of length $n+1$, there exists a sequence of internal transitions from q_{n+1} to q_n.

To prove this, for any state q_n described in the inductive hypothesis, and any state q_{n+1} that is reachable from q_n by an internal transition (i.e., $(q_n, v, q_{n+1}) \in TR^C$, $v \in \textbf{\textit{Enabled}}(q_n) - W^C$, and q_{n+1} is reachable form q by a sequence t_{n+1} of length $n+1$), we show that there exists a sequence of internal transitions from q_{n+1} to q_n.

As illustrated in Figure 5.12, let $t_{q_n} = q_nq'_{n+1}\ldots q_{n-1}$ be the presumed sequence of internal transitions from q_n to q_{n-1}, $(q_n, u, q'_{n+1}) \in TR^C$, $u \in \textbf{\textit{Enabled}}(q_n) - W^C$, $(q_{n-1}, w, q_n) \in TR^C$, and $w \in \textbf{\textit{Enabled}}(q_{n-1}) - W^C$. Next consider the following two possible cases:

(a) v makes a transition along trace t_{q_n}. Then as illustrated by the commutative diagram of Figure 5.13, and because of the presumed independence of all internal transitions, there must exist a sequence t'_{q_n} from q_n to q_{n-1} whose first transition is by

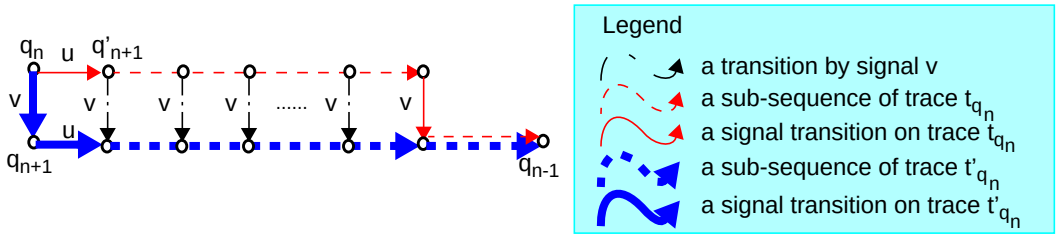


Fig. 5.13 Illustration of case (a) in the proof of Lemma 5.8.
If signal v makes a transition along sequence t_{q_n}, then there exists a sequence of internal transitions from q_{n+1} to q_{n-1}.

signal v to state q_{n+1}. But this means that the suffix of this sequence is from q_{n+1} to q_{n-1}, and hence there exists a sequence from q_{n+1} to q_n.

(b) v does not make a transition along trace t_{q_n}, and thus $v \in \textbf{\textit{Enabled}}(q_{n-1}) - W^C$, and $(q_{n-1}, v, q'_n) \in TR^C$. By the inductive hypothesis, there must exist a cycle of internal transitions $t_{q_{n-1}} = q_{n-1}q'_n \ldots q_{n-1}$ from q_{n-1}. But then, as illustrated by the commutative diagram of Figure 5.14, and because of the presumed independence of all internal transitions, there must exist a sequence t'_{q_n} from q_n to q_{n-1} whose first transition is by signal v to state q_{n+1}. But this again means that the suffix of sequence t'_{q_n} is from q_{n+1} to q_{n-1}, and hence there exists a sequence from q_{n+1} to q_n.

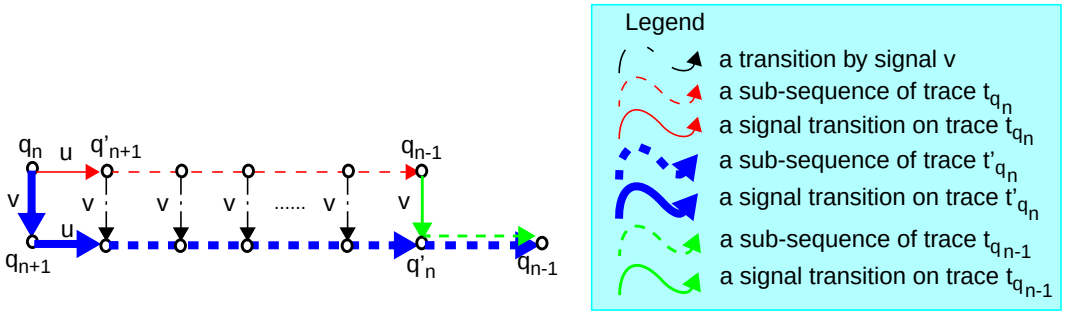


Fig. 5.14 Illustration of case (b) in the proof of Lemma 5.8.
If signal v does not make a transition along sequence t_{q_n}, then there exists a sequence of internal transitions from q_{n+1} to q_{n-1}.

Conditions (a) and (b) above indicate that there always exists a sequence of internal transitions from state q_{n+1} to q_n, and as a result a sequence back to q. ■

Lemma 5.9 [Convergence of sequences of internal transitions from a single state] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a failure-free circuit, $W^C \subseteq V^C$ be a set of external circuit variables that is closed under failure-free dependence, $E^C = A^C \cap W^C$, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. Also let Algorithm `DFS_2` be used for partial exploration of the state space of C. Let $p \in \tilde{Q}^C$ be any state (in particular, the initial state of any independent DFS path), and $q, q' \in \tilde{Q}^C$ be any pair of states (in particular, the terminal states of any two independent DFS paths) such that q and q' are reachable from p by sequences of internal transitions. Then, there must exist a state $p' \in Q^C$ that is reachable from both q and q' through sequences of internal transitions. ■

Proof (Sketch) This lemma directly follows from Keller's result [43] about independent signals and transitions. ■

Lemma 5.10 [Terminal states reachable from a single state belong to the same internal TMSCC] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a failure-free circuit, $W^C \subseteq V^C$ be a set of external circuit variables that is closed under failure-free dependence, $E^C = A^C \cap W^C$, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. Also let Algorithm `DFS_2` be used for partial exploration of the state space of C. Let $p \in \tilde{Q}^C$ be any state (in particular, the initial state of any independent DFS path), and $q, q' \in \tilde{Q}^C$ be the terminal states of any two

independent DFS paths such that q and q' are reachable from p by sequences of internal transitions. Then q and q' belong to the same internal TMSCC. ■

Proof (Sketch) By Lemma 5.9, there exists a state $p' \in Q^C$ that is reachable from both q and q' through sequences of internal transitions. By Lemma 5.8, terminal state q must belong to an internal TMSCC, $\hat{Q} \subseteq Q^C$, and similarly, terminal state q' must belong to an internal TMSCC, $\hat{Q}' \subseteq Q^C$. By the definition of an internal TMSCC, state p' that is reachable from both q and q' must belong to both of $\hat{Q}$ and $\hat{Q}'$. But since the internal TMSCC to which a state belongs is unique, we must have $\hat{Q} = \hat{Q}'$. ■

Lemma 5.11 [Internal transitions cannot disable external transitions] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a failure-free circuit, $W^C \subseteq V^C$ be a set of external circuit variables that is closed under failure-free dependence, $E^C = A^C \cap W^C$, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. Then no internal transition can ever disable any external variable. ■

Proof (Sketch) Assume that there exists an internal transition by a signal $v \in V^C - W^C$ that can disable an external variable $w \in W^C$. Then, either v and w are legally dependent, which would contradict the closure of W^C under failure-free dependence, or the internal transition is a failure, which would contradict the failure-freedom of the circuit. ■

Lemma 5.12 [Uniqueness of the set of enabled external variables in an internal TMSCC] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a failure-free circuit, $W^C \subseteq V^C$ be a set of external circuit variables that is closed under failure-free dependence, $E^C = A^C \cap W^C$, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. Let $\hat{Q} \subseteq Q^C$ be an internal TMSCC. Then for all $q \in \hat{Q}$, the set of enabled external variables, $\boldsymbol{Enabled}(q) \cap W^C$, is unique. ■

Proof (Sketch) Assume there exists a pair of states $q, q' \in \hat{Q}$ in the internal TMSCC, such that $\boldsymbol{Enabled}(q) \cap W^C \neq \boldsymbol{Enabled}(q') \cap W^C$. Then there must exist an external variable $v \in W^C$ that is enabled in q but is not enabled in q'. Since $\hat{Q}$ is an internal TMSCC, there must exist a sequence of internal transitions from q to q'. Now, along any such sequence, external variable v must have become disabled, without being fired. This suggests that external variable v is not independent of all internal signals, contradicting our assumption about closure of W^C under failure-free dependence. ■

Theorem 5.13 [`DFS_2` and finding a safe abstraction for a circuit] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a failure-free circuit, $W^C \subseteq V^C$ be a set of external circuit variables that is closed under failure-free dependence, $E^C = A^C \cap W^C$, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. Let Algorithm `DFS_2` be used for construction of a stuttering equivalent sub-automaton, $\tilde{FA}^C$. Then, embedded procedure `Construct_projection` (Figure 5.10) constructs $\tilde{FA}^C_{W^C}$ *iff* $\tilde{FA}^C$ is projectable onto W^C, and otherwise it

aborts the algorithm. Moreover, if procedure `Construct_projection` does not abort the algorithm, then the behavior of its output automaton is always a safe abstraction of the circuit behavior, even when the circuit is not failure-free. ■

The first part of the above theorem implies that for a failure-free circuit, if `DFS_2` constructs a sub-automaton $\tilde{FA}^C$ that is projectable onto W^C, then `Construct_projection` constructs nothing but $\tilde{FA}^C_{W^C}$, and $\tilde{B}^C_{W^C}$ is a safe (exact) abstraction of B^C over W^C; otherwise, `Construct_projection` simply aborts. The second part of the theorem implies that the algorithm's output--if it does not abort--is *always* a safe abstraction.

Proof (Sketch) By Conditions 2.22 for projectability of an automaton, $\tilde{FA}^C$ is projectable onto W^C *iff* the following conditions hold:

- Let $q_j \in \tilde{Q}^C$ be any initial state of $\tilde{Q}^C$, or any state to which there exists an external transition $(q'_j, b, q_j) \in \tilde{TR}^C$ from some state $q'_j \in \tilde{Q}^C$ such that $q'_j \notin [q_j]_{W^C}$. Let $Q_j \subseteq \tilde{Q}^C$ be the set of all states such that $q_k \in Q_j$ iff

  (i) q_k is reachable from q_j through a (possibly ε) sequence of W^C-compatible states in $\tilde{Q}^C$, and

  (ii) there exists $(q_k, c, q_m) \in \tilde{TR}^C, q_m \notin [q_j]_{W^C}$; i.e., an external transition from q_k to a state that is not W^C-compatible with q_k.

  Then let

  $W_j = \{\boldsymbol{Proj}(W^C)(q_k, c, q_m) \,|\, (q_k, c, q_m) \in \tilde{TR}^C, q_k \in Q_j, q_m \notin [q_j]_{W^C}\}$ be the projection of all external state transitions from the states in Q_j.

- Let $q_l \in \tilde{Q}^C$ be any other initial state of $\tilde{Q}^C$, or any other state to which there exists an external transition $(q'_l, d, q_l) \in \tilde{TR}^C$ from some state $q'_l \in \tilde{Q}^C$ such that $q'_l \notin [q_j]_{W^C}$ and $q_l \in [q_j]_{W^C}$; i.e., q_j and q_l are W^C-compatible. Define Q_l and W_l similar to Q_j and W_j above.
- Then we must have $W_j = W_l$.

If the above conditions hold, then we have $Q_V^C = \{\textbf{\textit{Proj}}(V)(q_j)\}$ and $TR_V^C = \{W_j\}$, for all states q_j as described above.

Because of the specific way that `DFS_2` constructs $\tilde{FA}^C$, any state q_j or q_l in the above conditions must be the initial state of some independent DFS path, and any state q'_j, q'_l, or q_k must be a terminal state that is fully expanded. As a result, $\tilde{FA}^C$ is projectable onto W^C iff the following conditions hold:

- Let $q_j \in \tilde{Q}^C$ be the initial state of any independent DFS path of Algorithm `DFS_2`. Let $Q_j \subseteq \tilde{Q}^C$ be the set of *all* terminal (fully expanded) states that are reachable from q_j through sequences of internal transitions. Let $W_j = \{\textbf{\textit{Proj}}(W^C)(q_k, c, q_m) \,|\, (q_k, c, q_m) \in \tilde{TR}^C, q_k \in Q_j, q_m \notin [q_j]_{W^C}\}$ be the projection of all external state transitions from the terminal states in Q_j.
- Let $q_l \in \tilde{Q}^C$ be the initial state of any other independent DFS path of Algorithm `DFS_2`, such that $q_l \in [q_j]_{W^C}$; i.e., q_j and q_l are W^C-compatible. Define Q_l and W_l similar to Q_j and W_j above.
- Then we must have $W_j = W_l$.

Now, consider procedure `Construct_projection` of Figure 5.10. This procedure is called once for each independent DFS. That is, if $q_j \in \tilde{Q}^C$ is the initial state of a DFS path, then `Construct_projection` is called at terminal state $q'_j \in \tilde{Q}^C$ of this DFS path. For each such terminal state q'_j (and thus each initial state q_j), `Construct_projection` computes a set $W'_j = \{\textbf{\textit{Proj}}(W^C)(q'_j, c, q_m) \big| (q'_j, c, q_m) \in \tilde{TR}^C, q_m \notin [q'_j]_{W^C}\}$ (note that since the initial and terminal states of any DFS path are W^C-compatible, we have $[q'_j]_{W^C} = [q_j]_{W^C}$). Note that since $q'_j \in Q_j$, we have $W'_j \subseteq W_j$. Next, `Construct_projection` checks the validity of $W'_j = W'_l$ for all previously processed DFS paths whose terminal states, q'_l (and thus initial states q_l), are W^C-compatible with terminal state q'_j (and thus initial state q_j). If the above check fails, then the algorithm is aborted. Otherwise, if no such terminal state q'_l (and thus initial state q_l) was processed before, all states and transitions in W'_j are added to $\tilde{FA}^C_{W^C}$. In essence, procedure `Construct_projection` analyses only a sub-behavior $B'^C \subseteq \tilde{B}^C$ and tries to find an automaton $FA'^C_{W^C}$ such that $B'^C_{W^C} = \textbf{\textit{Proj}}(W^C)(B'^C)$.

Now, for failure-free circuits we always have $W'_j = W_j$ (by Lemma 5.12), and as a result $\textbf{\textit{Proj}}(W^C)(B'^C) = \textbf{\textit{Proj}}(W^C)(\tilde{B}^C)$. Thus, if the circuit is failure-free, then `Construct_projection` would correctly check the projectability of $\tilde{FA}^C$, and correctly compute $\tilde{FA}^C_{W^C}$ or abort (if $\tilde{FA}^C$ is not projectable), without ever computing Q_j. (Note that if $\tilde{FA}^C_{W^C}$ exists, then we have $FA'^C_{W^C} = \tilde{FA}^C_{W^C}$.). As a result, if the circuit is failure-free, then `Construct_projection` constructs $\tilde{FA}^C_{W^C}$ *iff* $\tilde{FA}^C$ is project-

able onto W^C, and otherwise it aborts, suggesting that it could not find a safe abstraction.

For a circuit that is not failure-free, the result of Lemma 5.12 will not hold, and $W'_j = W_j$ would not generally be true. In such a case, either `Construct_projection` finds an automaton $FA'^C_{W^C}$ such that $B'^C_{W^C} = \textbf{\textit{Proj}}(W^C)(B'^C)$, or it aborts. If it succeeds, then as a result of $B'^C \subseteq \tilde{B}^C$, $\tilde{B}^C \subseteq B^C$, and $B'^C_{W^C} = \textbf{\textit{Proj}}(W^C)(B'^C)$, we have $B'^C_{W^C} \subseteq \textbf{\textit{Proj}}(W^C)(B^C)$; i.e., $B'^C_{W^C}$ is a safe abstraction of B^C over W^C. ■

If `Construct_projection` aborts, it simply means that we couldn't find a safe abstraction. Note that when a circuit is failure-free, the projectability of $\tilde{FA}^C$ is independent of the DFS paths and, in particular, their terminal states. In contrast, when the circuit is not failure-free, the projectability of $\tilde{FA}^C$ can vary by how the DFS paths are explored (the order in which internal signals fire along those paths). Thus, for a circuit that is not failure-free, failure of `Construct_projection` to find a safe abstraction--even if one could have been found--is not considered a short-coming of the algorithm.

Observation 5.14 [Algorithm 5.3, `DFS_2`, and the UEE conditions for finding a safe abstraction] Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be any given circuit, $W^C \subseteq V^C$ be a set of external circuit variables that is closed under failure-free dependence, $E^C = A^C \cap W^C$, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \textbf{\textit{Proj}}(W^C)(q)$. Let Algorithm `DFS_2` be used for construction of a stuttering equivalent sub-automaton, $\tilde{FA}^C$. Then `Construct_projection`

successfully finds a safe abstraction iff the set of external transitions from terminal states of independent DFS paths that are W^C-compatible is unique. In other words, it finds a safe abstraction iff W^C-compatible terminal states have the *Unique External Excitation* property, or UEE. ■

Proof [Algorithm 5.3, `DFS_2`, an enhanced algorithm for finding safe abstractions]

The proof of correctness of `DFS_2` in finding a safe abstraction immediately follows from Theorems 5.7 and 5.13. ■

Before closing this subsection, we need to emphasize that if `DFS_2` fails to find a safe abstraction, (the UEE conditions are not satisfied), then W^C is *potentially* not observationally sufficient, and another set of external variables (that has to be closed under failure-free dependence) has to be chosen for hierarchical verification.

Example 5.7 Figure 5.15 shows a four-stage FIFO controller that is partitioned with three different set of external signals. Figure 5.15.c is an example of a set of external signals over which a safe abstraction cannot be found. Intuitively, the middle circuit block in Figure 5.15.c can hold different number of tokens in the same external state. On the other hand, the output behavior of that circuit block depends on the number of tokens in it. As a result, the behavior of the corresponding set of external signals is not projectable, and hence a safe abstraction over it does not exist. Figure 5.15.a shows an example of a set of external signals over which a safe abstraction does exist; however, the right sub-circuit created by the safe abstraction is exactly the same as the original

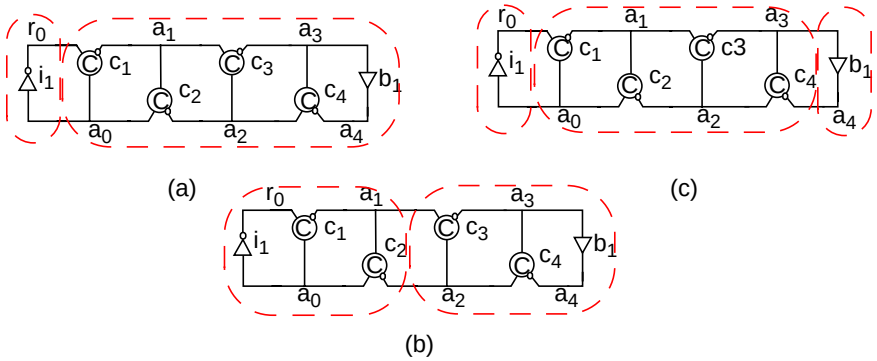


Fig. 5.15 Three different partitions of a four-stage FIFO controller.

flat circuit. As a result, the particular partition of Figure 5.15.a does not create any real hierarchy in the circuit. Finally, Figure 5.15.b shows an example of a set of external signals that not only a safe abstraction over it does exist, but also it can successfully induce hierarchy in verification of the circuit. ■

5.3.3 Further Optimizations

In this subsection, we present a further optimized version of our enhanced algorithm for finding safe abstractions. This version of the algorithm is called `DFS_3`, and is depicted in Figure 5.16. It is called by `Safe_abstraction`, in the same way that `DFS_2` is called. And it is exactly like `DFS_2`, except that it does not call `Explore_internal_trans`; i.e., it does not explore the transitions of enabled internal signals from the terminal states of the DFS paths. As was shown in the previous subsection, in a failure-free circuit, the terminal state of any DFS path belongs to an internal TMSCC, and any sequence of internal transitions from the terminal state always

```
1  DFS_3( p, i ) /* DFS on the stack of state p and circuit block i */
2     Pop ( q, p ); /* pop a state q from the stack of state p */
3     if Enabled(q) = ∅ then
4        return;
5     /* try to explore a single internal transition of block i
6        to a state that is not on the stack of p */
7     for each v ∈ (Enabled(q) – W^C) ∩ V^C_{E,i} {
8        /* v is an enabled internal signal of block i */
9        if (q, v, q') ∈ TR^C and q' ∉ Stack(p) then {
10          Construct_subautomaton( q, v, q' );
11          Push( q', p );
12          DFS_3( p, i );
13          return;
14       }
15    }
16    /* if all internal transitions of block i lead to states on
17       the search stack of p, move on to the next block i + 1 and
18       try to explore an internal transition of that block */
19    if i ≠ r^C_E then { /* not the last block */
20       Push( q, p );
21       DFS_3( p, i + 1 );
22       return;
23    }
24    /* the end of the DFS path from state p is reached */
25    else {
26       Construct_projection( q );
27       /* explore the external transitions from state q */
28       for each v ∈ Enabled(q) ∩ W^C {
29          /* v is an enabled external signal */
30          for each (q, v, q') ∈ TR^C {
31             Construct_subautomaton( q, v, q' );
32             /* initiate a new DFS search from each
33                un-explored state q' */
34             if q' ∉ Q̃^C then {
35                Push( q', q' );
36                DFS_3( q', 1 );
37             }
38          }
39       }
40    }
41 }
```

Fig. 5.16 Algorithm DFS_3.
An optimized version of Algorithm DFS_2, for finding a safe abstractions.

leads to other states of the same internal TMSCC. On the other hand, all states of any internal TMSCC were shown to have a unique set of enabled external variables. Thus, exploring enabled internal transitions from the terminal states of DFS paths will not create any new information about the behavior of the external variables. In fact, the particular selective search of procedure `Explore_internal_trans` that explores from any terminal state a path of internal transitions back to the same terminal state, was intentionally devised so to emphasize the redundancy of exploring internal transitions from terminal states.

Algorithm 5.4 [`DFS_3`, a further optimized algorithm to find safe abstractions]

Let $C = \langle M^C, A^C, V^C, G^C, FA^C \rangle$ be a failure-free circuit, $W^C \subseteq V^C$ be a set of external circuit variables that is closed under failure-free dependence, $E^C = A^C \cap W^C$, and $FTS^C = \langle FA^C, AP^C, L^C \rangle$ be a finite transition system with $L^C(q) = \boldsymbol{Proj}(W^C)(q)$. Algorithm `DFS_3` (Figure 5.16) is an optimized version of algorithm `DFS_2` that constructs a sub-automaton $\tilde{FA}^C = \langle A^C, V^C, \tilde{Q}^C, \tilde{\lambda}^C, \tilde{TR}^C, \tilde{\mu}^C, q_0^C \rangle$ of FA^C that is stuttering equivalent with FA^C. Moreover, its embedded procedure `Construct_projection` (Figure 5.10) finds an automaton projection of the constructed sub-automaton *iff* it is projectable, and otherwise it aborts the algorithm. Finally, if procedure `Construct_projection` does not abort the algorithm, then the behavior of its output automaton is always a safe abstraction of the circuit behavior, even when the circuit is not failure-free. ■

Proof [Algorithm 5.4, `DFS_3`, a further optimized algorithm for finding safe abstractions]

(Sketch) The correctness of Algorithm `DFS_3` directly follows from the correctness of Algorithm `DFS_2`, and the fact that in a failure-free circuit, any state that is reachable from the terminal state of any DFS path belongs to the same internal TMSCC, and thus has the same set of enabled external variables. As a result, exploration of enabled internal transitions from the terminal states of DFS paths in Algorithm `DFS_2` is a redundant computation that can be removed. ■

The above theorem states that for partial order reduction of a failure-free circuit, the selected set of enabled variables whose transitions are explored at terminal states of DFS paths need to include only the external variables, without violating the visibility condition (condition **C2**) of ample sets.

Example 5.8 Figure 5.17 depicts a FIFO controller of length eight partitioned in the middle into two circuit blocks ($E = \{a_3, a_4\}$). A safe abstraction of the circuit behavior over E is found using our partial order procedure. Since the sub-automaton that is constructed by our procedure is very big, any of its sequences of internal signal transitions, starting immediately after an external signal transition, is collapsed into and depicted as a single state transition to the corresponding terminal state. The constructed sub-automaton satisfies UEE, and thus the collapsed automaton is indeed a safe abstraction. ■

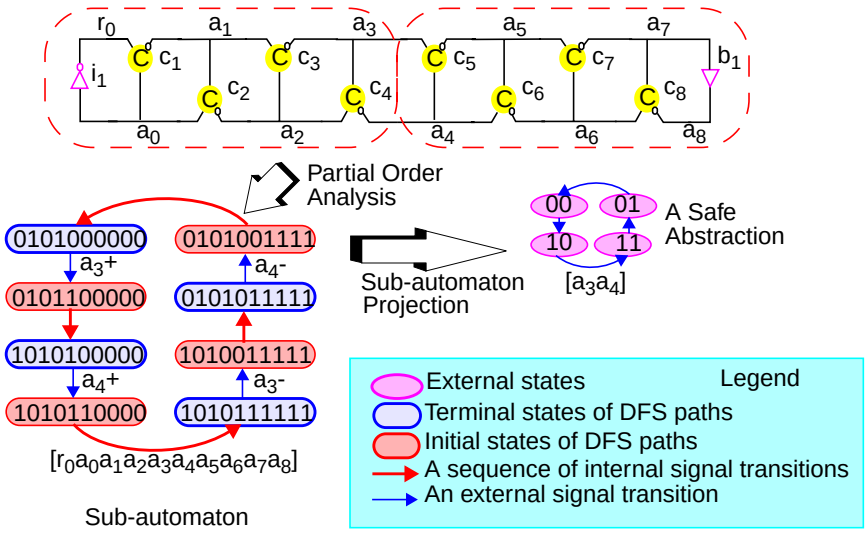


Fig. 5.17 Finding a safe abstraction for the behavior of a FIFO controller.

Before we close this chapter, we would like to make two final notes about the selection of external variables. As previously suggested, `construct_projection` may fail to construct the automaton of a safe abstraction because of failure transitions that go undiagnosed during construction of the partial order sub-automaton. This may cause a seemingly unnecessary search for a safe abstraction over *other* sets of external variables that might repeatedly fail because of the inherent failure of a circuit. To avoid such a condition, we can check for failure transitions during construction of the partial order sub-automaton. In this case, as soon as a failure is detected, the verification procedure can be quit. Although checking failures during partial order reduction incur some additional cost, this approach will remove the need for unnecessary subsequent search for safe abstractions.

The second note is regarding an extension of this framework in which a partial order sub-automaton that is not projectable (e.g., does not satisfy UEE) can still be used to find safe abstractions assuming that appropriate state encoding is used to distinguish between externally-compatible terminal states of the sub-automaton that do not have the same enabled external transitions. Such encoding schemes may introduce new state variables into the system and require additional analysis of the relabelled sub-automaton, adding to both space and time complexity of the algorithm. However, this approach removes the inherent complexity of our current framework in trying different set of external variables to find a safe abstraction.

Chapter 6

In Comparison

In this chapter, we first present an overall view of our framework for hierarchical verification of speed-independent circuits. In Section 6.2, we show how our framework is in fact based on an assume-guarantee paradigm. In Section 6.3, we present a comparison of our framework with that of complex-gate verification and show how we have succeeded in generalizing and extending that framework. Finally in Section 6.4, we show how our efforts compare to other verification efforts in terms of the reduction and/or abstraction techniques that are used.

6.1 The Flow of Our Approach Illustrated by an Example

In this section, we simply review the steps involved in one level of recursion of our hierarchal verification approach; i.e., the steps taken starting from a circuit to the derivation of its sub-circuits. Since these steps have already been discussed and each illustrated by an example, we illustrate the whole flow in a single example illustrated in Figure 6.1. The Figure is assumed to be self explanatory.

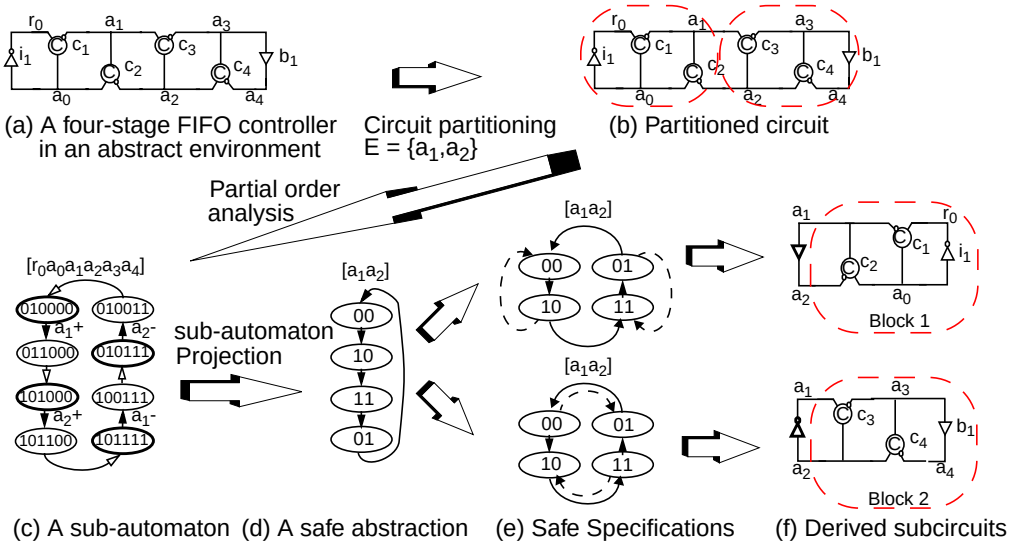


Fig. 6.1 One level of hierarchical verification for a FIFO controller.

6.2 Induced Hierarchical Verification, an Assume Guarantee Paradigm

In this section, we briefly show how our hierarchical verification technique for SI circuits can be viewed as an assume guarantee paradigm.

Our technique can be viewed as one which starts by assuming that a given circuit is failure-free (or SI), and then tries to guarantee that assumption by proving that the induced sub-circuits are failure-free. With the assumption of failure-freedom for the circuit, any safe abstraction (i.e., one that is generated by our partial order technique) would *exactly* resemble the behavior of the selected set of external variables. However, if any induced sub-circuit is found to have a failure, it would be an indication that the

initial assumption could not be guaranteed, and thus was not a true assumption. It is to be reminded that a failure in a sub-circuit is either at an internal module or at the environment module (i.e., a choke) of the corresponding circuit block; in the first case, the failure would be a failure of the same module inside the original circuit as well, contradicting our assumption of failure-freedom, and in the second case, the choke would reveal that the safe abstraction was not exact, indirectly contradicting our assumption that the circuit was failure-free.

6.3 Relation to Complex-Gate Verification

In this section, we briefly revisit a previous hierarchical verification technique--complex-gate verification. We show how complex-gate verification is inherently based on the same principles and observations about speed-independent circuits that were presented in the previous chapter. Moreover, we will show how our hierarchical verification framework have succeeded in generalizing upon complex-gate verification.

Our hierarchical verification framework initially started out as an attempt to extend and generalize complex-gate verification. Complex-gate verification is characterized by two phases; a *functional verification* phase, followed by a *behavioral verification* phase. In the functional verification phase, the circuit is collapsed and abstracted into a network of complex-gates that is checked for *functional correctness* (e.g., conformance to specification) by full exploration of its state space. Once functional correctness is established, the explored behavior of the complex-gate circuit is used to derive an

abstract environment for each induced circuit block. In the behavioral verification phase, failure-freedom of each circuit block in its abstract environment is checked. A circuit is said to be failure-free if it is both functionally and behaviorally correct [64, 65].

The functional phase in complex-gate verification is the counterpart of deriving safe abstractions in our framework. However, while we use *behavioral abstraction* (i.e., partial order reduction) to derive a safe abstraction of a circuit's behavior, the functional verification phase--as suggested by its name--uses *functional abstraction* to find an abstract behavior of the complex-gate circuit. The functional verification phase conceives of a circuit block that is collapsed into a complex-gate as a functional black box, focusing on the functionality of the circuit block rather than its behavior. The same complex-gate can also be conceived as a circuit block whose internal modules have zero delays and whose outputs are all *lazy* signals; i.e., the outputs fire only after all internal signals of the complex-gate have stabilized. It is this alternative view of a complex-gate, focusing on the behavior of the corresponding circuit block rather than its functionality, that has lead us to our partial order technique for behavioral abstraction. The above discussion also implies, although not immediately apparent, that in the special case where the set of external circuit variables includes all complex-gate outputs, functional abstraction and our behavioral abstraction generate the same results.

The generality of our framework as compared to complex-gate verification arises from the fact that an observationally sufficient set of external variables that partition a circuit into circuit blocks does not need to include all outputs of memory elements,

and/or cut all cycles in the circuit, while the set of external variables partitioning the circuit into complex-gates does. As a result, since no memory element output is ever internal to a complex-gate, the functional verification phase effectively assumes laziness for *all* such signals. In contrast, in our framework we may be able to *hide* some memory element outputs or cycles while deriving safe abstractions.

Since the functional abstraction is equivalent to some sort of partial order reduction with *static* choice of ample sets, it is easy to comprehend that the set of external signals in complex-gate verification, similar to our framework, has to always satisfy the closure under failure-free dependence conditions. The absence of *hidden* memory element outputs in complex-gate verification, however, has implications that have facilitated the derivation of safe abstractions in that framework. In our framework, the sub-automaton that is constructed by our partial order analysis has to satisfy a certain condition before it can be used to derive a safe abstraction; i.e., it has to be projectable onto the set of external variables. In complex-gate verification, the absence of *hidden* memory element outputs or cycles in complex-gates makes their output excitations depend only on their inputs/outputs. That is why complex-gates are treated as functional blocks in the functional verification phase. Now, had we used our equivalent partial order analysis instead, we would have noticed that regardless of the order of transitions on the non-lazy (hidden) circuit variables, the final excitations of lazy variables would depend on the value of (all) lazy variables only, and that the hidden non-lazy variables would never oscillate and would always *stabilize* at unique values. In other words, for a complex-gate circuit, our partial order analysis would always con-

struct a sub-automaton that has no cycle of hidden state transitions and is always projectable onto the set of complex-gate input/outputs (because it satisfies UEE). Moreover, since the projection of such a sub-automaton would always be a safe abstraction, the set of complex-gate input/outputs would *always* be identified as observationally sufficient. It is this feature that has facilitated the derivation of safe abstractions in complex-gate verification by removing the need to check any additional conditions (i.e., projectability).

It also becomes clear why complex-gate verification cannot support circuit blocks with internal memory modules, or combinational cycles. Complex-gate verification is closely concerned with the *functional* aspect of a complex-gate rather than its *behavioral* aspect; i.e., it is based on the fact that if the output excitation of a circuit block can be expressed as a *function* of its input/output signals only, then the behavior of the corresponding complex-gate circuit is indeed a safe abstraction. Now, if a circuit block has internal memory modules or cycles, its output excitation is generally not a *function* of its input/output signals only--it also depends on the current state of the internal variables of the block. Now, if the output excitation of such a circuit block is approximated by a function over only the input/output signals of the block, such that the function is possibly an under-approximation or an over-approximation of the actual excitation of the block within its environment, then there would be no trivial relationship between the outcomes of functional/behavioral phases of complex-gate verification and the actual failure-freedom of the original circuit. This is because the framework of complex-gate verification assumes that the *exact* functionality of the circuit blocks (com-

plex-gates) in terms of their input/output signals is given, or easily computable. One might argue that complete analyses of the behavior of circuit blocks within their *actual* environments can always be used to compute their *exact* functionality, but that would be contrary to the goal of induced hierarchical verification--verifying circuit blocks in *abstract* environments that are derived from safe abstractions which are found *without* complete behavioral analyses.

Our comparison of the two frameworks and their relationship can be summarized as follows.

In our attempt to generalize upon complex-gate verification, we first identified two inherent and implicit properties of complex-gates:

(a) complex-gates, as functional blocks, internally *stabilize* before having any output transitions,

(b) complex-gates have *unique internal stabilizations*, and thus when internally stabilized, they also have *unique output excitations* as functions of their (external) input/output signals only.

Having identified the above properties, we next tried to exploit the same properties into our own framework:

(a) we utilized and implemented the notion of *stabilization* into our behavioral abstraction by having our partial order technique explore only those traces of a circuit on which the internal variables of circuit blocks always *stabilize* (or reach a terminal oscillatory state) before external I/O transitions occur,

(b) we explicitly enforced the notion of *unique output excitations* in internally-stable (or terminal oscillatory) states by always checking the projectability of the sub-automaton that is constructed by our partial order analysis. This has guaranteed the correctness of our approach in the more general case that we have circuit blocks with internal memory elements or cycles.

Our proposed behavioral abstraction has brought us the advantage of being able to exercise hiding memory element outputs or cycles. However, this is achieved with the additional cost of checking, among others, the projectability of the constructed sub-automaton onto the set of external variables, a condition that is automatically satisfied in complex-gate verification.

6.4 Comparison with other Reduction Techniques

Partial order, abstraction, and hierarchical verification techniques have been extensively used in different tools to reduce the complexity of verification [14, 35, 37]. This section discusses the relationship between the traditional usage of such techniques and our proposed induced hierarchical verification approach.

In VIS [14], abstraction is referred to using non-determinism to abstract the behavior of some circuit signals. Specifically, the signals are treated as primary inputs whose behavior is totally unconstrained. This is probably too conservative for our application because such non-determinism would introduce unreachable states which may exhibit hazards, leading to false negatives. In contrast, we refer to an approximation of the

behavior of a subset of circuit signals as an abstraction. Moreover, unlike that of VIS, our abstraction never overestimates the behavior of the signals.

We have already discussed in detail the relation between our framework and partial order reduction techniques [1, 62, 63, 32, 33, 81, 82]. The tricky part of our use of partial order reduction techniques is that we do not know, *a priori*, whether a circuit is failure-free or not; yet we make the implicit *assumption* that it is failure-free and use partial order reduction to find the behavior of the external variables. Consequently, unlike the typical use of partial orders, our technique may actually under-approximate the behavior of the external variables. A key feature of our technique is that if there is any such under-approximation, it is always detected in the form of a failure in some sub-circuit with the conclusion that the circuit is not failure-free.

The presumed independence of signals in a speed-independent (failure-free) circuit allows us to take advantage of different techniques to speed up the partial order analysis. For example, a form of symbolic trajectory analysis can be used for internal stabilization of the circuit (at fixed external states). This symbolic trajectory analysis has two benefits. First, since the hidden signals are independent and the ordering of input transitions for hidden circuit elements with no state variables is immaterial, for such circuit elements we can use non-interleaving semantics in which enabled input signals are allowed to fire simultaneously (e.g., [85]). This reduces the number of iterations to stabilize the internal state of the circuit. Secondly, since the behavior of hidden variables is analyzed only locally when stabilizing the corresponding circuit

block, the hidden signals appear only locally and temporarily in BDD computations; i.e., the hidden variables need not be global BDD variables.

We believe that our technique is similar to homomorphic reductions as used in COSPAN [5, 35, 47]. In COSPAN, such homomorphisms simplify the language containment test between a model and a task by removing irrelevant aspects of the model. We conjecture that our safe abstraction can be viewed as a result of a homomorphic transformation which simplifies the model of the environment for each sub-circuit. In our framework, the homomorphic system is automatically generated (using our partial order technique) once a set of external signals is given, and the validity of the homomorphism is checked by checking a sufficient condition for observational sufficiency (projectability of the constructed sub-automaton). Moreover, we believe that this homomorphic reduction is both necessary and sufficient for verifying the non-reduced problem, and consequently does not lead to any false negatives, as can potentially happen with homomorphic reductions in COSPAN.

Our approach is also similar to the more general assume-guarantee paradigm used in *reactive modules* [2]. In that paradigm, a composition of reactive modules is verified through verification of each module in an abstract environment followed by the verification of the composition of abstract environments. We believe that our safe abstraction is to some degree analogous to an abstract environment with some differences. The most obvious difference may be that our methodology does not need a separate step of verifying the composition of abstract environments.

In comparison with other work on verification of speed-independent circuits, we should also note that Weih and Greenstreet developed a verification framework for speed-independent circuits with similar characteristics as ours but for a somewhat different purpose [85]. Specifically, rather than verifying speed-independence of a circuit, their goal is to verify *local formulas* for circuits that have already been verified to be hazard-free (i.e., semi-modular). In other words, in a preprocessing stage, they must rely on traditional techniques to verify the speed-independence of the design. Nevertheless, their ideas are similar to ours in that to achieve their goal, they argue that only one interleaving needs to be analyzed. Finally, Kishinesky *et al*.'s work on analysis and identification of a class of speed-independent circuits, called distributive circuits [45], is based on derivation of an event specification of the circuit behavior in an STG-like notation that also avoids the state space explosion problem. Their derivation of such an specification is based on notions of dependency and concurrency similar to our framework.

Chapter 7

SPHINX

We have developed a CAD tool named SPHINX which implements our proposed induced hierarchical verification framework for speed-independent circuits.

There are three types of input files to the program. One input file describes the structure of the circuit as an interconnection of elementary, macro, or specification modules, along with additional information about the initial value of circuit signals and suggested ordering for BDD variables. The second type of input file is used for the description of macro modules that are a collection of elementary circuit modules (e.g., gates). The third type of input file describes the specification modules as Petri-Net or STG specifications. The user has to interactively specify external variables of the circuit (and those of sub-circuits at different levels of hierarchy), and variables that need to be projected away to derive safe specifications for circuit blocks. The current version of the tool does not perform any analysis to identify legal dependencies between circuit variables, and the user is expected to choose the set of external variables in such a way that they are closed under failure-free dependence. While adding a feature to automatically identify legal dependencies is quite straight forward, such a feature

seems to be not much of use, since in many speed-independent circuits, the only types of legal dependencies are between the outputs of non-deterministic modules (output choice) which can easily be identified by the user.

The tool automatically encodes the automaton representation of modules. For each circuit and its specified set of external signals, the program finds a safe abstraction if one exists, using symbolic partial order analysis, and automatically partitions the circuit into circuit blocks. Next, for each subsequent sub-circuit, the components of the sub-circuit are assigned (overloaded by) new descriptions relative to the context of that sub-circuit, and the sub-circuit is recursively analyzed. At the lowest level of the hierarchy, symbolic reachability analysis is used to verify failure-freedom of the flat sub-circuit. For comparison purposes, the tool can also perform symbolic reachability analysis and verify hazard-freedom on the original flat circuit. The tool can also utilize the extra level of abstraction of complex-gates. That is, for further speed up, the partial order analysis can be alternatively performed on the complex-gate abstraction of the partitioned circuit, where the combinational cones of logic within the circuit blocks are collapsed into complex-gates. The program can generate a state diagram description of any partially or fully explored state space that can be interpreted and viewed by another program, PARG (by Tomas Rokicki).

Symbolic techniques (using the CUDD package of VIS [14]) are used to handle sets of states and any operations on them, including the partial order exploration of the

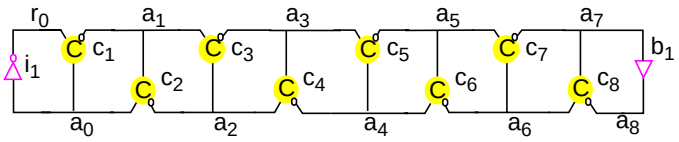


Fig. 7.1 A FIFO controller of length = 8.

state space, any full reachability analysis of the state space of a sub-circuit, checking the projectability of automata, and automata projections.

The executable files of SPHINX, together with descriptions of tool capabilities, guidelines, sample circuits, and runtimes are accessible at http://jungfrau.usc.edu/SPHINX/sphinx.html.

Table 1 shows some runtime results of the tool for two sets of examples, FIFO controller (Figure 7.1) and DME-ring circuits of different lengths (Figures 7.2 and 7.3), on a Sun SPARCstation 5 with 32 MBytes of memory. As a measure of the amount of memory required, we use the maximum number of BDD nodes *in use* before any instance of garbage collection. The table shows that our hierarchical approach yields significant runtime speed ups compared to flat verification, especially for the FIFO controller which is an example of a circuit dominated by memory elements that can be successfully hidden in our verification framework. In fact, the speed up grows exponentially with the length of the FIFO. This can be explained by the fact that, in the FIFO circuits, the depth of hierarchy logarithmically increases with the size of the circuit, while the maximum number of external gates always stays constant at four.

Table 1: SPHINX Run-Time Results

Circuit	Depth of Hierarchy	Max # of External Gates	CPU-Time (ms)	Peak # of BDD Nodes	CPU-Time Ratio	BDD Size Ratio	Projection Depth
FIFO 4	0	6	120	1,005	1.0	1.0	-
FIFO 4	**1**	**4**	**110**	**582**	**1.1**	**1.7**	**-**
FIFO 8	0	10	1,010	4,964	1.0	1.0	-
FIFO 8	**2**	**4**	**420**	**1,645**	**2.4**	**3.0**	**-**
FIFO 16	0	18	64,510	585,740	1.0	1.0	-
FIFO 16	**3**	**4**	**1,030**	**4,193**	**62.7**	**140**	**-**
FIFO 32	0	34	1.5e+7	5.5e+5	1.0	1.0	-
FIFO 32	**4**	**4**	**3,780**	**11,530**	**~4000**	**~47.0**	**-**
FIFO 64	0	66	>180h	>35Mbyte	1.0	1.0	-
FIFO 64	**5**	**4**	**12,220**	**28,116**	**N/A**	**N/A**	**-**
DME-ring 2	0	32	6,490	16,584	1.0	1.0	-
DME-ring 2	1	21	5,310	17,768	1.2	0.9	-
DME-ring 2	**1**	**19**	**4,630**	**25,194**	**1.4**	**0.7**	**1**
DME-ring 2	2	15	7,550	15,880	0.9	1.0	1
DME-ring 2	2	12	8,150	101,353	0.8	0.2	2
DME-ring 3	0	48	95,320	501,919	1.0	1.0	-
DME-ring 3	1	26	24,300	28,094	3.9	17.9	-
DME-ring 3	**1**	**21**	**22,730**	**31,734**	**4.2**	**15.8**	**1**
DME-ring 3	2	17	29,600	26,702	3.2	18.8	1
DME-ring 3	2	15	37,510	26,702	2.5	18.8	2
DME-ring 4	0	64	617,470	3,086,251	1.0	1.0	-
DME-ring 4	1	31	94,390	46,632	6.5	66.2	-
DME-ring 4	**1**	**26**	**79,750**	**48,466**	**7.7**	**63.7**	**1**
DME-ring 4	2	22	115,280	48,466	5.4	63.7	1

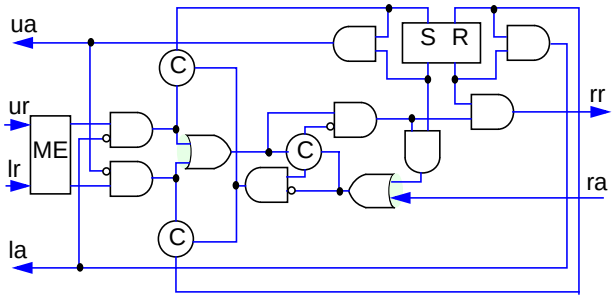


Fig. 7.2 A DME cell.

On the other hand, for the DME-ring example where the non-deterministic outputs of ME modules cannot be hidden, the depth of hierarchy remains constant while the number of initial sub-circuits and their set of external signals grow linearly with the size of the circuit. Limited projection of the safe abstraction has proven to be the best option for the verification of the DME-ring example. This is due to the high cost of checking projectability of the safe abstractions and computing their projections.

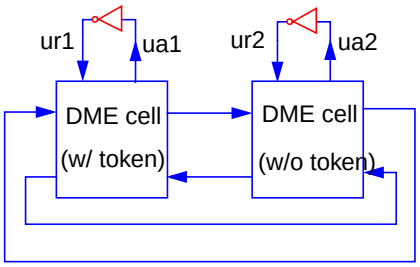


Fig. 7.3 A DME ring of length = 2.

Chapter 8

Directions for Future Research

We presented a new approach for induced hierarchical verification of speed-independent circuits that improves upon previous approaches on some circuits. The approach generalizes previous efforts for the verification of speed-independent circuits [7, 8, 27, 53, 64, 65, 88] and is believed to have interesting relationships with current efforts in the analysis of synchronous circuits that have combinational loops [49, 20, 71, 72].

Our CAD tool SPHINX is already available on the World-Wide-Web [91]. Our experiments with the tool have focused on example circuits for which the tool would promise advantage over available tools such as Versify [64, 65], because our technique is a generalization of those techniques, and reduces to them for other circuits. However, there is still room to further improve, optimize, and even test the tool on more circuits. Some features of the program that can be improved are its interface, and error trace generation.

The following is a list of possible future directions for this research:

- a formal study of extension of the framework to the verification of asynchronous circuits with relative timing assumptions and in particular self-timed circuits; i.e., hierarchical verification and hierarchical extraction of relative-timing constraints/assumptions for such circuits, integration of the results of this research with techniques for (flat) verification of relative-timed circuits [44], etc.
- a formal study of extensions of this work to the verification of delay-insensitive circuits, quasi-delay insensitive circuits, and verification of liveness properties.
- exploring applications of the framework to the analysis of synchronous circuits that have combinational loops.
- research on techniques/heuristics for automatic selection of observationally sufficient sets of external signals.

In the following sections, we first present some of our preliminary ideas and directions for extending the scope of our current framework to the domain of relative timed circuit verification. We then present a discussion on the issues involved in using multiple safe abstractions for hierarchical verification, and will close this chapter with an open conjecture on the correctness of a potential solution for this problem.

8.1 Hierarchical Verification of Relative-Timed Circuits

Speed-independent circuits (systems), as we know, are circuits which should work correctly regardless of--absolute or relative--component delays. Equivalently, a SI circuit

should work correctly for all possible ordering of events (e.g., signal transitions). In verifying speed-independent circuits (i.e., *untimed systems*), modules and specifications are modeled as if they have unbounded delays. Moreover, time is not *quantitatively* modeled; rather, it is inherently modeled in the evolution of the system through event occurrences.

In this section, we briefly discuss the problem of verifying another class of asynchronous circuits; circuits which are not speed-independent *per se*, yet their failures are avoided by restricting the possible ordering of events through a set of *timing constraints/assumptions*.

Timing constraints can be provided in the form of bounded delays for circuit modules and specifications. Verification of systems with such timing constraints (metric timing) requires explicit representation of time. There are techniques and tools for the verification of such systems which use either *discrete* or *continuous* time models. Continuous time models can provide accurate verification results. Discrete time models (e.g., [18, 13]), on the other hand, are often not as accurate, and may have partial failure coverage [79, 80, 89, 90]. Discrete time models use timer variables to model the passage of time, while dense time models use notions such as *unit-cubes* (regions) [3], or *convex geometric regions* (or zones) [28, 11, 38]. Both techniques suffer from the additional cost associated with explicit modeling of time, which aggravates the already serious problem of state explosion. Partial order techniques have been used to avoid the explosion of timed states (or regions) [87, 70, 84]. [66, 67, 59, 9, 10] use *partially ordered sets* (POSETs) of events to reduce the number of regions per untimed states.

Timing constraints can also be provided in the form of *relative timing assumptions/ constraints* (RTA/RTC). Often, the environment behavior is assumed to be restricted by relative timing assumptions (RTA), while the circuit behavior is constrained by relative timing constrains (RTC). An RTA/RTC imposes restrictions on the possible orderings of some set of related events. As an example, an RTC may indicate that if a signal transition will causally enable two other transitions, one of them always will occur before the other one. Such constraints restrict the reachable state space of a (closed) circuit, and if chosen correctly, can prevent a circuit from reaching its (untimed) failure states. Then, a physical implementation of the circuit will operate safely within its environment, if both the implementation and the environment meet their relative timing constraints/assumptions.

Aggressive asynchronous circuit design using relative timing is becoming the state of the art in asynchronous design. The RAPPID architecture is an example of such efforts [69]. Design and verification of RT circuits has been addressed by [73, 74, 26, 60].

Verification of circuits with relative timing information does not require explicit modeling of time. All is needed is to impose timing assumptions/constraints when exploring the untimed state space of the circuit, pruning any part of the untimed state space which can be entered only by violation of such constraints/assumptions. Thus flat verification of circuits with relative timing information seems to be a trivial problem.

Our hierarchical verification framework for SI circuits can easily be generalized to handle RT circuits. For this, we simply need to exercise RTA/RTCs during any partial or full state space exploration of the circuit, whether it is for finding safe abstractions, or for flat verification of a sub-circuit. As a result, safe abstractions of RT circuits would contain only those interleaving of events on external signals which adhere to RTA/RTCs.

A safe abstractions of an RT circuit may need to carry further information about relative ordering of events. For example, consider a timing constraint which involves three signals a, b, and c, with a eventually enabling b and c in some particular order. Moreover, assume that at some level of hierarchy, a becomes a hidden signal of one circuit block, and b and c become hidden signals of another circuit block. If RT information is not passed appropriately across levels of hierarchy, correct verification of the circuit block containing b and c may not be possible. This is because information regarding the correct ordering of events on b and c might have been lost in the safe abstraction, due to all signals a, b, and c being hidden signals. This example shows the importance of preservation of RT constraints across levels of hierarchy and also across circuit blocks of any level of hierarchy.

Any RTA or RTC constraint can be modeled as an additional sequential circuit module whose inputs are the signals involved in the constraint. The state of such a module accordingly evolves by events on its input signals. We call any such module which represents a constraint a *constraint module*. The firings of any circuit signal a of an RT circuit is controlled by the set of all constraints which constrain signal a.

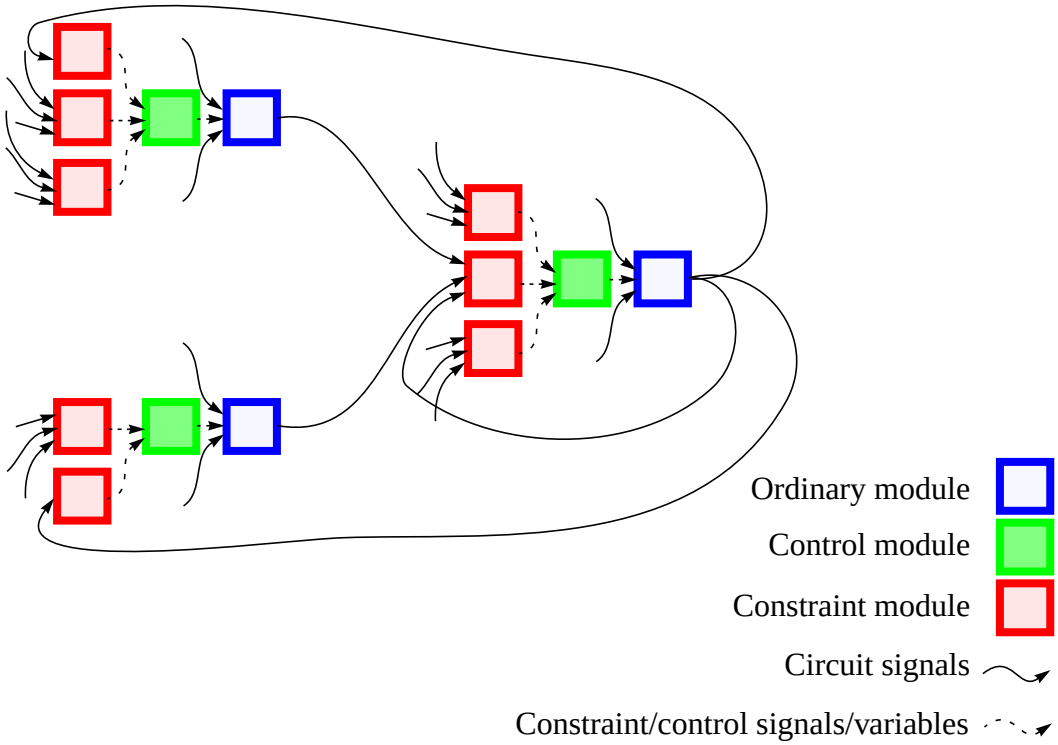


Fig. 8.1 Modeling an RT circuit as an SI circuit with additional circuitry.
The additional circuitry are to enforce the RT constraints and assumptions.

This can be modeled by an additional (control) input signal at the module that drives signal a. This additional control input signal would allow the driving module to fire signal a only if all RTA/RTCs containing that event are satisfied. Such a control signal can in turn be produced by some logic that monitors the states of all constraint modules which have signal a in their input, and generates a '1' output only if all the constraints are met. We call such a module a *control module*. (See Figure 8.1).

Modelling the effect of RTA/RTS by introducing constraint modules, new control signals at ordinary circuit modules, and the logic driving such control signals, enables us to use the same framework for hierarchical verification of RT circuits that we had

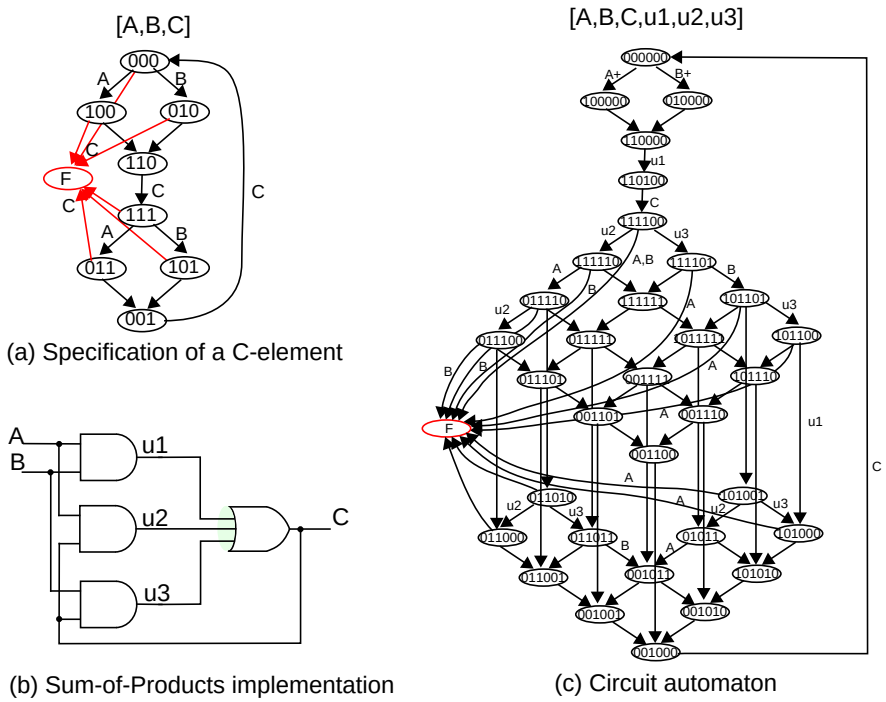


Fig. 8.2 A C-element.
(a) Specification, (b) Sum-of-Product implementation, (c) circuit automaton

proposed for verification of speed-independent circuits. In other words, an RT circuit modeled as described above, can be verified for failure-freedom as an ordinary SI circuit. The advantage of modeling RT circuits as SI circuits with additional circuitry is that safe abstractions of such circuits will automatically contain and carry all required RT information necessary for correct verification.

Example 8.1 Figure 8.2 shows the specification of a C-element, together with a Sum-of-Product implementation of it, and the circuit automaton of the implementation. As suggested by the circuit automaton, this SoP implementation of a C-element is not speed-independent, or failure-free. Note that to simplify the illustration, the failure

transitions of the circuit are all directed to a failure state labeled with F. The main cause of failures in this implementation is the possibility for the inputs of the circuit to change before the internal signals of it have stabilized. For example, consider the scenario in which inputs A and B both become ‘1’, causing u1 and then output C become ‘1’. Now, if before signal u2 (u3) gets a chance to rise to ‘1’, input A (B) falls to ‘0’, then u2 (u3) becomes disabled, causing a failure. This failure can cause output C to fall to ‘0’, while it is expected to remain at ‘1’.

The SoP implementation of C-element will be failure-free if two relative timing constraints are satisfied by the environment of the C-element. These constraints limit the response time of the environment to changes at the output of the C-element, such that the circuit’s inputs are not changed before its internal signals are stabilized. These two RTCs are C+ u2+ < C+ A- and C+ u3+ < C+ B-. They suggest that the sequence of transitions consisting of rising of C followed by rising of u2 (u3), has to happen before the sequence of transitions consisting of rising of C followed by falling of A (B).

For this particular circuit, RTC condition C+ u2+ < C+ A- (similarly C+ u3+ < C+ B-) can be modelled by a module with the automaton of Figure 8.3. The inputs of such a module are signals C and u2, and its output is signal A. Failure transitions (by unexpected inputs) are omitted for the sake of clarity. This model is developed using knowledge about sequences of signal transitions that are possible by the circuit. For example, it is not possible for signal C of this circuit to fall before both signals A and u2 fall. If such knowledge is not available, the RTC can be modelled with more general

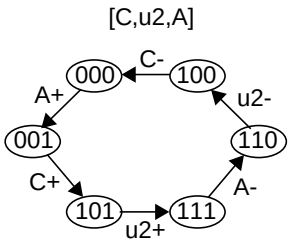
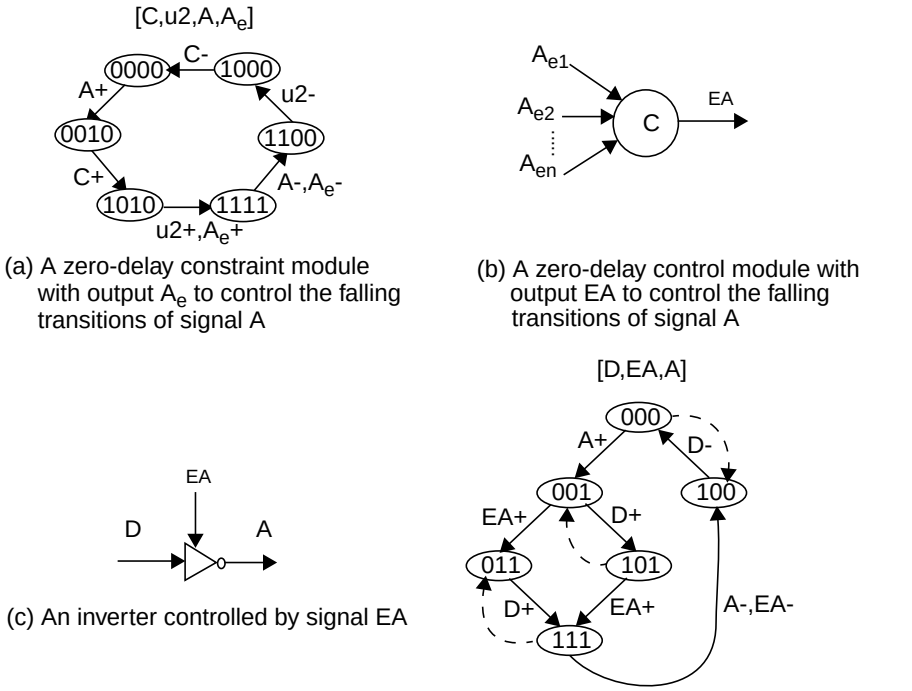


Fig. 8.3 Modelling an RTC.
Modelling RTC C+ u2+ < C+ A- by an automaton using knowledge about the behavior of a C-element.

and complicated Petri-Nets. Such general models may allow the signals involved in the RTC to reset right after they have made the transitions specified in the RTC; however, after a reset, they may not allow the signals to make any further transitions before all transitions specified in the RTC have occurred. Such general models may also allow each signal to reset any time after it has made its transition, as late as right before its next expected transition in the RTC.

The above modules cannot be directly composed with the ordinary modules of a circuit to impose the corresponding RTC constraints. The reason is that the outputs of these modules (e.g., signal A) are in fact driven by other modules (e.g., the environment), and by definition of a circuit, no two modules can drive the same signal. This problem can be resolved by making all signals involved in an RTC constraint as input signals of its constraint module, and include an output signal that simultaneously becomes ‘1’ with the second transition of the RTC, and ‘0’ with the last transition of the RTC. Such a module acts like a *zero-delay* module whose output fires right after it becomes enabled, with no delay. The automaton of such a constraint module for RTC



(a) A zero-delay constraint module with output A_e to control the falling transitions of signal A

(b) A zero-delay control module with output EA to control the falling transitions of signal A

(c) An inverter controlled by signal EA

(d) The module automaton of the controlled inverter

Fig. 8.4 Modeling the effect of multiple RTCs on an inverter.
RTC C+ u2+ < C+ A- and other RTCs that constrain transitions of signal A, the output of an inverter.

C+ u2+ < C+ A- is depicted in Figure 8.4.a. The set of inputs of the new constraint module is {C,u2,A}, and its output set is $\{A_e\}$. Signal A_e can then be used to enable the falling transition of signal A.

As an example, assume that there is an inverter in the circuit that drives signal A (Figure 8.4.c). Moreover, assume that there are multiple RTC constraint modules for the falling transition of A, and each of them have a zero-delay output, A_{ei}. The control module of the inverter gate can be modelled by a zero-delay speed-independent C-element that collects the A_{ei} signals and generates a '1' at its output, EA, immediately fol-

lowing the instance that all A_{ei} signals become one (Figure 8.4.b). Note that the falling of A will simultaneously reset all A_{ei} signals and signal EA. The model of the inverter gate that drives signal A also needs to be modified, such that it supports an additional input that is driven by signal EA (Figure 8.4.c and 8.4.d). This controlled model of an inverter will monitor its control input EA and have a falling transition at the output A only if EA is a '1', otherwise, the output transition is *postponed* (Figure 8.4.d). Note that transition of A after EA does not need to be a zero-delay transition. It is notable that the model of the controlled inverter that is illustrated in Figure 8.4.d is a simplified model, based on knowledge about the possible behaviors; e.g., it has taken advantage of the knowledge that EA can become '1' only after A rises. If such knowledge were not captured in the model, its automaton would be more complicated. ■

The above example illustrates some of the issues that arise when using our framework for hierarchical verification of RT circuits. The most important issue is that modelling the effect of RTC/RTAs by additional circuitry requires the introduction of the notion of zero-delay modules into the framework. Our present framework already supports the notion of internal state variables of modules that can simultaneously change with the I/O signals of their modules. Extending the framework to handle concurrent transitions of I/O signals is believed to affect only the semantics of the behavior of a circuit, and not the correctness of the framework. However, a proof of concept and feasibility of this approach requires further research. Efficient implementation of the effect of RTA/RTCs through additional circuitry and modified ordinary modules, choosing OSV sets over the newly introduced variables, and correct handling of pro-

jections of such variables seem to be other important issues that need to be further investigated and researched.

We close this section by pointing to another problem which is closely related to verification of RT circuits; i.e., automatic extraction of RTA/RTCs for such circuits. Assuming that automatic extraction frameworks are available for flat RT circuits, it might be possible to combine our proposed hierarchical verification framework with such frameworks for hierarchical extraction of RTA/RTCs. Studying the involved issues and problems is another interesting area for future research.

8.2 Hierarchical Verification using Multiple Safe Abstractions

In this subsection, we discuss a variation of our hierarchical verification framework which seems to be an attractive alternative approach. This variation aims at verifying the conformance of circuit blocks of a circuit to safe specifications that are not derived from the same safe abstractions; i.e., it uses multiple safe abstractions for derivation of sub-circuits. A particular problem with this approach is illustrated through an example. Then we propose a slight modification in our framework which *might* legitimize using multiple safe abstractions for hierarchical verification. However, the correctness of the new approach, or the existence of any correct approach for hierarchical verification using multiple safe abstractions remains an open problem.

In our framework, we partition a given circuit into a set of circuit blocks, find a safe abstraction of the behavior of the external signals that partition the circuit, and

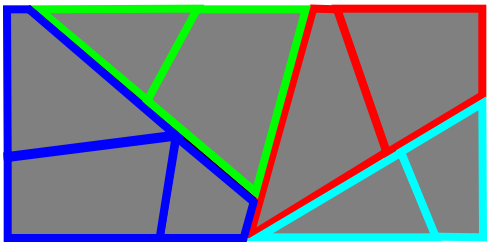


Fig. 8.5 An abstract view of a circuit with a covering set of super-blocks.
Each super-block is partitioned into a set of circuit blocks. Not all circuit blocks are verified against the same safe abstraction. Only circuit blocks of the same super-block (having the same color) are verified using the same safe abstractions. The super-blocks can overlap.

verify each circuit block against a safe specification that is obtained from the safe abstraction. An alternative approach which may come to mind is to:

(i) select a set of *circuit super-blocks* that is a *covering* set for the circuit modules, and can possibly overlap. Each super-block is partitioned into a set of circuit blocks,

(ii) for each super-block, find a safe abstraction over a set of external circuit variables that is a superset of the I/O variables of the circuit blocks in that super-block,

(iii) verify each circuit block of a super-block against a safe specification that is derived from the safe abstraction that is found for that super-block (See Figure 8.5).

This alternative approach is appealing since a safe abstraction that is used to verify the circuit blocks of a super-block has potentially a smaller set of external signals compared to a single safe abstraction that is used to verify all circuit blocks of a circuit. Since the complexity of finding (each) safe abstractions is exponential in the number of (the corresponding) external variables, the overall cost of finding multiple safe abstractions can be less than that of finding a single safe abstraction for all the circuit

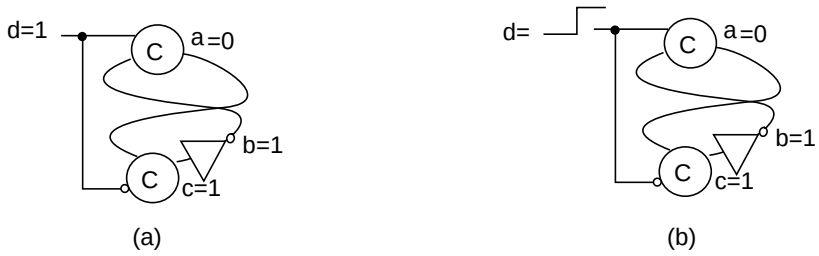


Fig. 8.6 Incorrect verification using multiple safe abstractions.
This circuit is not correctly verifiable if multiple safe abstractions are used to verify its circuit blocks. (a) a problematic state of the circuit, (b) a transition leading to the problematic state.

blocks. Note that by allowing the super-blocks to overlap in the proposed scenario, a single module can appear in multiple circuit blocks (of multiple super-blocks), and be verified multiple times.

The problem with the above approach arises from the very fact that not all circuit blocks are verified against the same safe abstraction. It is not hard to imagine a case in which the safe abstractions that are used to verify the circuit blocks of different super-blocks are all under approximated abstractions such that the internal failures of their corresponding circuit blocks do not get a chance to manifest themselves. In contrast, when a single under approximated safe abstraction is used to verify all circuit blocks, the sources of under approximation which are failure(s) in some of the circuit blocks will all be found during the verification of those failing circuit blocks.

Figure 8.6.a illustrates an example of a circuit which is not always verifiable if multiple safe abstractions are used to verify its circuit blocks. The indicated (initial) state of the circuit ($[abcd] = 0111$) is selected very carefully; all three signals a, b,

and c are simultaneously enabled in that state, but the firing of any one of them will disable another one of them, causing a failure. For example, a can rise and disable c; b can fall and disable a; and c can fall and disable b. All of the following are true of this circuit:

(i) $0111, 1111, 1011$ is a trace of the circuit which yields an under-approximated safe abstraction $11, 01$ over the I/O signals of the inverter ($[bc]$). The inverter is failure-free in the environment specified by this safe abstraction;

(ii) $0111, 0101, 1101$ is a trace of the circuit which yields an under-approximated safe abstraction $011, 111$ over the I/O signals of the top C-element ($[abd]$). The top C-element is failure-free in the environment specified by this safe abstraction;

(iii) $0111, 0011, 0001$ is a trace of the circuit which yields an under-approximated safe abstraction $011, 001$ over the I/O signals of the bottom C-element ($[acd]$). The bottom C-element is failure-free in the environment specified by this safe abstraction.

Thus, using different safe abstractions to verify the circuit blocks (modules) of this circuit may result a false positive verification result for the circuit.

On the other hand, and as an example of using a single safe abstraction to verify all circuit blocks, if the two C-elements were verified using the safe abstraction used for the inverter (i.e., $0111, 1111, 1011$), then a failure on the top C-element would have been detected, which would correctly imply the failure of the circuit.

The example of Figure 8.6.a was able to illustrate the potential problem of using multiple safe abstractions because it had multiple failures which could mask each

other in different safe abstractions. This particular condition, although seemingly artificial, can occur in practice, as illustrated in Figure 8.6.b. In Figure 8.6.b, there exists a race between the transitions of the two signals d and b, such that b falling first will cause no subsequent failures, but d rising first will enable multiple simultaneous failures; i.e., d rising will enable two more transitions (on a and c), with all of the enabled transitions leading to failures. This situation is reminiscent of a violation of fundamental mode constraints, since the input signal d is changing before the circuit has stabilized.

Multiple simultaneous failures are not always enabled as described above; they can become enabled as a result of a single failure as well. In such a case, it might be possible to locate the actual source of the failures (the single failure initiating the other failures) in the sub-circuit containing the failing module.

At this point, we present one possible solution to the problem of using multiple safe abstractions for hierarchical verification. We have not been able to prove or disprove the correctness of this solution yet, and thus it remains as an open problem for future research. This possible solution is based on a slight modification of the definition and derivation of safe abstractions. Our original definitions suggest that a safe abstraction of the behavior of a circuit is the projection of a sub-automaton of the circuit automaton that is assumed to be *failure-free*. This implies that during the construction of the sub-automaton out of the circuit description, there is no need to pay attention to failure transitions that might have been explored, since any such failures can be detected later when verifying the circuit blocks using a single safe abstraction.

Not checking for failures during derivation of safe abstractions is also motivated by the fact that it reduces the cost of finding safe abstractions.

The suggested modification in the definition and derivation of safe abstractions is as follows: during construction of each sub-automaton from which a safe abstraction is derived, failure transitions are *all* checked for; if any failure transition is detected then the circuit obviously has a failure, otherwise, the sub-automaton is *truly* failure-free. The new solution would then identify a circuit as failure-free iff for any super-block of the circuit the sub-automaton used to derive its safe abstraction is truly failure-free *and* all sub-circuits of the super-block are failure-free.

The above modification in the derivation of safe abstractions guarantees that if a failure transition that is located outside a super-block is explored in the safe abstraction of that super-block, the failure is not masked out during verification of the circuit blocks of the super-block. However, it is still possible for all failures that are located outside the super-block to be missed from the safe abstraction. Such a condition can result an under-approximated safe abstraction which can in turn hide internal failures of the super-block.

The only situation in which the suggested solution can fail to correctly verify a circuit is when the circuit has a failure, yet all sub-automata of safe abstractions and all sub-circuits of the super-blocks are failure-free. This can happen only if during verification of sub-circuits, failures originating from within the circuit blocks are never activated. But that can *possibly* happen only if all (failure-free) safe abstractions which are used for verifying failing circuit blocks are under-approximations that can hide all the

failures of those blocks. Note that if a safe abstraction is exact, the internal failures of the circuit blocks of its super-block are always guaranteed to be found. A prove or disprove of the suggested solution has to show whether or not the combination of the above conditions is ever possible.

Even if the correctness of the suggested solution can be proven, the approach can be more expensive than our original approach (using a single safe abstraction) since it has to investigate all transitions of all sub-automata for possible failures.

Bibliography

[1] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Proc. of CAV-97*, Vol. 1254 of LNCS, pp. 340-351. Springer, 1997.

[2] R. Alur, T. A. Henzinger. Reactive Modules. In *Proc. of LICS*-96, pp. 207-218.

[3] R. Alur. *Techniques for Automatic Verification of Real-Time Systems*. Ph.D. Thesis, Stanford University, August 1991.

[4] R. Alur and D. Dill. A theory of timed automata. Theoretical Computer Science, Vol. 126, No. 2, pp. 183-235, 1994.

[5] R. Alur and R. P. Kurshan. Timing analysis in COSPAN. In *Hybrid Systems III*. Springer-Verlag, 1996.

[6] P. A. Beerel. *CAD Tools for the Synthesis, Verification, and Testability of Robust Asynchronous Circuits*. Ph.D. Thesis, Stanford University, August 1994.

[7] P. A. Beerel, J. R. Burch, and T. H.-Y. Meng. Efficient verification of determinate speed-independent circuits. In *Proc. of ICCAD-93*, pp. 261-267. IEEE Computer Society Press, 1993.

[8] P. A. Beerel, J. R. Burch, and T. H. Meng. Checking combinational equivalence of speed-independent circuits. In *Formal Methods in System Design*, Vol. 13, pp. 37-85, Kluwer Academic Publishers, Boston, 1998.

[9] W. Belluomini and C. J. Myers. Verification of timed systems using POSETS. In *Proc. of International Conference on Computer Aided Verification*, 1997.

[10] W. Belluomini and C. J. Myers. Efficient timing analysis algorithms for timed state space exploration. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, April 1997.

[11] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, Vol. 17, No. 3, March 1991.

[12] K. van Berkel, F. Huberts, and Ad Peeters. Stretching quasi delay insensitivity by means of extended isochronic forks. In *Asynchronous Design Methodologies*, pp. 99-106. IEEE Computer Society Press, May 1995

[13] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Proc. International Conference on Computer Aided Verification*, 1997.

[14] R. K. Brayton et al. VIS: A System for verification and synthesis. In *Proc. of CAV-96*, pp. 428-432. Springer, 1996.

[15] J. A. Brzozowski, C.-J. H. Seger. *Asynchronous Circuits*. Springer-Verlag, New York, 1995.

[16] J. A. Brzozowski and H. Zhang. Delay-insensitivity and semi-modularity. Technical Report CS-97-11, Dept. of Comp. Science, Univ. of Waterloo, March 1997.

[17] J. A. Brzozowski and J. C. Ebergen. On the delay-sensitivity of gate networks. IEEE Transactions on Computers, vol. 41, pp. 1349-1360, November 1992.

[18] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 4, April 1994, pp. 401-424.

[19] J. R. Burch. Modeling timing assumptions in trace theory. In *ICCD*, 1989.

[20] J. R. Burch, D. Dill, E. Wolf, and G. De Micheli. Modeling hierarchical combinational circuits. In *Proc. of ICCAD-93*, pp. 612-618. IEEE Computer Society Press, 1993.

[21] S. M. Burns. General conditions for the decomposition of state holding elements. In *Proc. of Async-96*, pp. 48-57, 1996.

[22] T.-A. Chu. Synthesis of self-timed control circuits from graphs: An example. In *Proc. of IEEE International Conference on Computer Design*, pp. 565-571, October 1986.

[23] J. N. Cook. *Production rule verification for quasi-delay-insensitive circuits*. Master's thesis, California Institute of Technology, June 1993.

[24] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[25] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, E. Pastor, A. Yakovlev. Decomposition and technology mapping of speed-independent circuits using Boolean relations. In *Proc. of Intl. Conf. on CAD-97*, pp. 220-227, 1997.

[26] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Taubin, and A. Yakovlev. Lazy transition systems: application to timing optimization of asynchronous circuits. In Proc. International Conf. Computer-Aided Design (ICCAD), pp. 324-331, November 1998.

[27] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. In *ACM Distinguished Dissertations Series*, The MIT Press, 1988.

[28] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proc. of the Workshop on Automatic Verification Methods for Finite-State Systems*, 1989.

[29] Jo C. Ebergen and Ad M.G. Peeters. Modulo-N Counters: Design and Analysis of Delay-Insensitive Circuits. In *Proc. Int. workshop on designing correct circuits*, pp. 27-46, Elsevier 1992.

[30] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*. Dissertation, Eindhoven University of Technology, Dept. of Computing Science. October 1987.

[31] J. C. Ebergen and S. Gingras. A verifier for network decompositions of command-based specifications. In *Proc. of HICCS*, 1993.

[32] P. Godefroid. *Partial-Order Methods for Verification of Concurrent Systems*. Springer, 1996.

[33] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, Vol. 110, pp.305-326, May 1994.

[34] G. Gopalakrishnan. A correctness criterion for asynchronous circuit validation and optimization. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 11, Nov. 1994.

[35] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In *Proc. of CAV-96*, Vol. 1102 of LNCS, pp. 423-427. Springer, 1996.

[36] S. Hauck. Asynchronous Design Methodologies: An Overview. In *Proceedings of the IEEE*. Vol. 83, No. 1, pp. 69-93, January 1995.

[37] G. H. Holzmann and D. Peled. The state of SPIN. In *Proc. of CAV-96*, Vol. 1102 of LNCS, pp. 385-389. Springer, 1996.

[38] H. Hulgaard. *Timing Analysis and Verification of Timed Asynchronous Circuits*. Ph.D. thesis, University of Washington, 1995.

[39] M. B. Josephs, S. M. Nowick, and C. H. (kees) Van Berkel. Modeling and Design of Asynchronous Circuits. In *Proceedings of the IEEE*. Vol. 87, No. 2, pp. 234-242, February 1999.

[40] M. B. Josephs. Receptive Process Theory. In *Acta Informatica*. Vol. 29, pp. 17-31, 1992.

[41] M. B. Josephs and J. T. Udding. An Overview of DI Algebra. *Proceedings of 26th Annu. Hawaii Int. Conf. System Sciences*, 1993, Vol. 1, pp. 329-338.

[42] H. Kagotani and T. Nanya. A synthesis method of quasi-delay-insensitive processors based on dependency graph. In *Asia-Pacific Conference on Hardware Description Languages (APCHDL)*, pp. 177-184, October 1994.

[43] R. M. Keller. A fundamental theorem of asynchronous parallel computation. *Lecture Notes in Computer Science*, Vol. 24, pp. 103-112, 1975.

[44] H. Kim and P. A. Beerel. Relative Timing Based Verification of Timed Circuits and Systems. In *Proc. of International Workshop on Logic Synthesis*, IWLS'99, June 1999.

[45] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. Analysis and identification of speed-independent circuits on an event model. To appear in *Formal Methods in System Design*.

[46] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Technology mapping for speed-independent circuits: decomposition and resynthesis. In *Proc. of Async-97*, 1997.

[47] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes - The Automata-Theoretic Approach*. Princeton Univ. Press, 1994.

[48] L. Lavagno. *Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from Signal Transition Graphs*. Ph.D. Dissertation, Univ. California, Berkeley, CA, 1992.

[49] S. Malik. Analysis of cyclic combinational circuits. In *Proc. of ICCAD-93*, pp. 618-625. IEEE Computer Society Press, 1993.

[50] R. Manohar and A.J. Martin. Quasi-delay-insensitive circuits are Turing-complete. Invited paper, Second International Symposium on *Advanced Research in Asynchronous Circuits and Systems*, March 1996.

[51] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. Distributed Computing, vol. 1, pp. 226-234, 1986.

[52] A. Mazurkiewitcz. Basic notions of trace theory. In *Workshop on Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, Vol. 354, of Lecture Notes in Computer Science, pp. 285-363. Springer, 1988.

[53] K. L. McMillan. A technique of state space search based on unfolding. In *Formal Methods in System Design*, Vol. 6, pp. 45-65, Kluwer Academic Publishers, Boston, 1995.

[54] K. L. McMillan. *Symbolic Model Checking*. New York, Kluwer Academic Publishers, 1993.

[55] R. E. Miller. *Switching Theory. Vol. II: Sequential Circuits and Machines*. John Wiley and Sons, 1965.

[56] C. E. Molnar, T. P. Fang, and F. U. Rosenberger. Synthesis of Delay-Insensitive Modules. *Proceedings of the 1985 Chapel Hill Conference on VLSI*, H. Fuchs, ed., Computer Science Press, Rockville, Maryland, pp. 67-86, 1986.

[57] D. E. Muller and W. S. Bartky. A Theory of Asynchronous Circuits. In *The annals of the Computation Laboratory of Harvard University. Vol. XXIX: Proceedings of an International Symposium on the Theory of Switching, Part I*. pp. 204-243, Harvard University Press., 1959.

[58] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, Vol. 77, No. 4, pp. 541-574, Apr. 1989.

[59] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. Ph.D. Thesis, Stanford University, 1995.

[60] R. Negulescu and A. Peeters. Verification of Speed-Dependences in Single-Rail Handshake Circuits. In *Proc. of the 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1998.

[61] E. Pastor, J. Cortadella M. A. Pena. Structural Methods to Improve the Symbolic Analysis of Petri Nets. In *Proc. 20th International Conference on Application and Theory of Petri Nets*, June 1999.

[62] D. Peled. Ten Years of Partial Order Reduction. In *Proc. of 10th International Conference on Computer Aided Verification*, pp. 17-28, Springer, 1998.

[63] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Formal Methods in System Design*, Vol. 8, pp. 39-64, Kluwer Academic Publishers, Boston, 1996.

[64] O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri Nets. In *16th Intl. Conf. on Theory and Application of Petri-Nets*, Torino, Italy, June 1996.

[65] O. Roig. *Formal Verification and Testing of Asynchronous Circuits*. Ph.D. Thesis, Univ. of Politecnica de Catalunya, Barcelona, 1997.

[66] T. G. Rokicki. *Representing and Modeling Circuits*. Ph.D. Thesis, Stanford University, 1993.

[67] T. G. Rokicki and C. J. Myers. Automatic verification of timed circuits. In *Proc. of International Conference on Computer Aided Verification*, pp. 468-480, Springer-Verlag, 1994.

[68] L. Ya. Rosenblum and A. V. Yakovlev. Signal graphs: From self-timed to timed ones. In *International Workshop on Timed Petri Nets*, pp. 199-206, July 1985.

[69] S. Rotem, K. Stevens, R. Ginosar, P. Beerel, C. Myers, K. Yun, R. Kol, C. Dike, M. Roncken, and B. Agapiev. RAPPID: An Asynchronous Instruction Length Decoder. In *Proc. of the 5th International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE, April 1999.

[70] A. Semenov and A. Yakovlev. Verification of asynchronous circuits using time Petri-net unfolding. In *Proc. of ACM/IEEE Design Automation Conference*, 1996.

[71] T. R. Shiple, G, Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proc. of ED&TC-96*, pp. 328-333, March 1996.

[72] T. R. Shiple, V. Singhal, R.K. Brayton, and A.L. Sangiovnni-Vincentelli. Analysis of combinational cycles in sequential circuits. In *Proc. of ISCAS-96*, pp. 592-595, May 1996.

[73] K. Stevens, S. Rotem, M. Burns, J. Cortadella, R. Ginosar, M. Kishinevsky, and M. Roncken. CAD directions for high performance asynchronous circuits. In *Proc. ACM/IEEE Design Automation Conference*, pp. 116-121, 1999.

[74] K. Stevens, R. Ginosar, S. Rotem. Relative Timing. In *Proc. of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE, 1999.

[75] S. Tasiran and R. K. Brayton. STARI: A case study in compositional and hierarchical timing verification. In *Proc. International Conference on Computer Aided Verification*, 1997.

[76] J. T. Udding. A Formal Model for Defining and Classifying Delay-Insensitive Circuits and Systems. *Distributed Computing*, Vol. 1, No. 4, pp. 197-204, 1986.

[77] S. H. Unger. *Asynchronous Sequential Switching Circuits*. John Wiley and Sons, 1969.

[78] S. H. Unger. Hazards, Critical Races, and Metastability. In *IEEE Transactions on Computers*, Vol. 44, No. 6, pp. 754-768, June 1995.

[79] V. Vakilotojar and P. A. Beerel. RTL verification of timed asynchronous and heterogeneous systems using symbolic model checking. In *INTEGRATION, The VLSI Journal*, December 1997.

[80] V. Vakilotojar and P. A. Beerel. RTL verification of timed asynchronous and heterogeneous systems using symbolic model checking. In *Proc. of ASPDAC-97*, January 1997.

[81] A. Valmari. Stubborn sets for reduced state space generation. In *Proc. 2nd Workshop on Computer Aided Verification*, pp. 491-515, 1990.

[82] A. Valmari. On-the-fly verification with stubborn sets. In *Proc. of CAV-93*, Vol. 697 of LNCS, pp. 397-408. Springer-Verlag, 1993.

[83] C. H. (Kees) Van Berkel, M. B. Josephs, and S. M. Nowick. Scanning the Technology. In *Proceedings of the IEEE*. Vol. 87, No. 2, pp. 223-233, February 1999.

[84] E. Verlind, G. de Jong, and B. Lin. Efficient partial enumeration for timing analysis of asynchronous systems. In *Proc. of ACM/IEEE Design Automation Conference*, 1996.

[85] D. T. Weih and M. R. Greenstreet. Verification of speed-independent data-path circuits. In *IEE Proceedings-Computers and Digital Techniques*, Vol. 143, No. 5, pp. 295-300, Sept. 1996.

[86] H. Wong-Toi and D. L. Dill. Verification of real-time systems by successive over and under approximation, In *Proc. of The International Conference on Computer-Aided Verification*, July 1995.

[87] T. Yoneda, A. Shibayama, B. Schlingloff, and E. M. Clarck. Efficient verification of parallel real time systems. In Costas Courcoubetis, editor, *Computer Aided Verification*, pp. 321-323. Springer-Verlag, 1993.

[88] T. Yoneda and T. Yoshikawa. Using partial orders for trace theoretic verification of asynchronous circuits. In *Proc. of ASYNC-96*, pp. 152-163, March 1996.

[89] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. Dooply, and J. Arceo. The design and verification of a low-control-overhead asynchronous differential equation solver. In *Proc. of ASYNC-97*, April 1997.

[90] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. Dooply, and J. Arceo. The design and verification of a low-control-overhead asynchronous differential equation solver. In *IEEE Transactions on VLSI*, Dec. 1998.

[91] The current version of SPHINX is accessible at http://jungfrau.usc.edu/SPHINX/sphinx.html.