Hardware Equivalence and Property Verification

Invited chapter for "Boolean Methods and Models in Mathematics, Computer Science and Engineering", pp. 599-674,
Yves Crama and Peter L. Hammer (eds.)
Encyclopedia of Mathematics and its Applications 134,
Cambridge University Press, 2010

Jie-Hong Roland Jiang[⋆] and Tiziano Villa[⋆⋆]

1 Introduction to Formal Verification

1.1 The Problem of Verification

Synthesis and verification are two basic steps in designing a digital electronic system, which may involve both hardware and software components. Synthesis aims to produce an implementation that satisfies the specification while minimizing some cost objectives, such as circuit area, code size, timing, power consumption, etc. Verification deals with the certification that the synthesized component is correct.

In system design, hardware synthesis and verification are more developed than the software counterparts, and will be our focus. The reason of this asymmetric development is three-fold: First, hardware design automation is better driven by industrial needs; after all, hardware costs are more tangible. Second, the correctness and time-to-market criteria of hardware design are in general more stringent. As a result, hardware design requires rigorous design methodology and high automation. Third, hardware synthesis and verification admit simpler formulation and are better studied.

There are various types of hardware verification, according to design stages, methodologies, and objectives. By design stages, verification can be deployed in high-level design from specification, called *design verification*, during synthesis transformation, called *implementation verification*, or after circuit manufacturing, called *manufacture verification*.

Manufacture verification is also known as *testing*. There is a whole research and engineering community devoted to it. In hardware testing, we would like to know if some defects appear in a manufactured circuit by testing the conformance between it and its intended design. In this case, the circuit — usually a sequential circuit, modeled abstractly as a finite-state machine (FSM) — is treated as a black box. The temporal behavior of the FSM can only be observed through

[⋆] Dept. of Electrical Engineering/Graduate Institute of Electronics Engineering, National Taiwan University, Taipei 10617, Taiwan. Email:`jhjiang@cc.ee.ntu.edu.tw`.

[⋆⋆] Dipartimento d'Informatica, Universita' di Verona, Strada Le Grazie, 15, 37134 Verona, Italy. Email: `tiziano.villa@univr.it`.

output responses to input stimuli. In contrast, in design and implementation verification, a state transition system is seen as a white box. Its transition relation is known beforehand, and thus not only the output sequences but also the state traces are known for given input sequences.

By methodologies, verification can be *informal* or *formal.* Informal verification shows the presence, but not the absence, of design errors; formal verification shows both the presence and absence. Verifying the correctness of the design itself can be performed in different contexts. A common design practice at different stages is *simulation*, i.e., stimulating the circuit with input sequences to check if the actual output sequences correspond to the expected ones [Gos93,Pil98,ZKP00,Mic03,Lam05] The confidence of the outcome depends on the view of the circuit (e.g., digital blocks, logic gates, or transistors) and the size of the simulated input space. For instance, it is reasonable to simulate an electronic circuit using a numerical simulator like SPICE, to characterize the behavior of a circuit at the transistor level corresponding to a few logic gates. For register-transfer level (RTL) and gate-level circuits, there are logic simulators, but it is usually impossible to exercise the complete input space. Therefore simulation is commonly conceived as an informal approach to verification.

Since the 1980s, theoretical and practical breakthroughs made possible *formal verification*, the automatic process of checking formally whether a design satisfies some requirements. ”Formally” means that we establish that the circuit satisfies the requirements by a mathematical analysis of the system, exploring the full set of behaviors, which is equivalent to an exhaustive simulation. A variety of formalisms has been introduced and studied to support a rigorous analysis, and they are based on automata and logical calculus. Automata may be regular or ω-regular, according to whether we care about computed sequences of finite or infinite lengths.

A very important class of formal methods goes under the name of *model checking* [CGP99], where a finite-state system is represented by a labeled state transition graph, the so-called Kripke structure, where labels of a state are the values of atomic propositions in that state (for example the values of the registers). Properties about the system are expressed as formulae in temporal logic, of which the state transition system is to be a “model”. So model checking consists of traversing the graph of the transition system and of verifying that it satisfies the formula representing the property, i.e., the system is a model of the property. We do not discuss here infinite-state systems (software programs); there are however vibrant research efforts about verification of such systems, too.

By objectives, from the point-of-view of the design flow, formal verification may be divided into *property checking* (is what I designed what I wanted?) and *equivalence checking* (is what I produced at a certain design stage, the same as what I had at the previous design stage?).

The objective of equivalence checking is to ensure that, in a design developed through phases that refine previous stages, the refinement process preserves the original behavior. So equivalence checking is a sanity check to guarantee that we are maintaining the design integrity through the synthesis and optimization phases. In theory these transformations should be correct by construction; however the same tools that perform them should be formally verified, and the process would be endless. So it is a good practice to check conformance of input and output of each synthesis phase.

In property verification, given a transition system, we test whether the informal description of the specification has been captured correctly by a formal description with a hardware description language (an HDL), by checking whether the system satisfies some property. Properties capture functionality, timing and temporal relations, safety (is it possible for a bad event to happen?), liveness (does an expected good event eventually take place?) and fairness (does every request eventually receive service?).

Safety properties are the most common to express. For instance, do the traffic light controllers of a crossroad make sure that intersecting road lines have mutually exclusive rights of the way? As an example of liveness, in communication systems with lossy channels, we make certain types of fairness assumptions about the loss of messages, i.e., if a message is transmitted infinitely many times, then it will be received infinitely often. In such systems, in general, we can only prove liveness properties. Fairness assumptions are often imposed upon liveness properties, but are not needed for safety properties. In fact, the extent of the role played by liveness and fairness properties is debated. It was suggested that liveness property checking can be converted into safety property checking [SB04].

In this chapter we focus on safety properties because they are the most commonly used in hardware verification problems. Two important hardware verification problems, sequential equivalence checking and resetability verification, fall into this category. They assert that something bad never happens during the evolution of the system and their violation can be detected by analyzing finite executions of the system. They can be posed as checking that some property p holds in the reachable states R of system S starting from the initial state(s). This checking justifies the crucial role played by *reachability analysis* in the verification of safety properties.

Notice that hardware designs may be specified hierarchically; this is also consistent with how a human designer operates. In order to formally verify a design, it must first be converted into a simpler "verifiable" format. The design is specified as a set of interacting systems; each has a finite number of configurations, called states. States and transition between states constitute FSMs. The entire system is an FSM, which can be obtained by composing the FSMs associated with each component. Hence the first step in verification consists of obtaining a complete FSM description of the system. Given a present state (or current configuration), the next state (or successive configuration) of an FSM can be written as a function of its present state and inputs (transition function or transition relation).

1.2 Model Checking and Temporal Logics

In this subsection we briefly survey model checking of properties expressed in temporal logics. A temporal logic is used to express the ordering of events in time by means of operators that specify properties such as "property *p* will eventually hold". There are various versions of temporal logics, such as linear temporal logic (LTL), computation tree logic (CTL), and other variants. Different temporal logics can have different expressive powers. For instance, LTL and CTL are incomparable. That is, there are temporal formulae in LTL not expressible in CTL, and vice versa. Below we focus on CTL to exemplify temporal logics.

Computation trees are derived from state transition graphs. The graph structure is unwound into an infinite tree rooted at the initial state. Figure 1 shows an example of unwinding a graph into a tree. Paths in this tree represent all possible computations of the system being modeled. Formulae in CTL refer to the computation tree derived from the model. CTL is classified as a branching time logic (in contrast to a linear time logic, such as LTL) because it has operators that describe the branching structure of this tree.

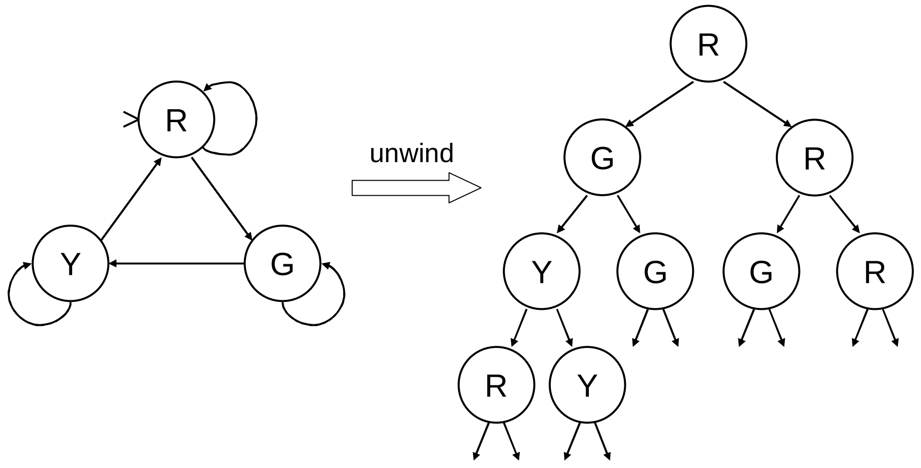


Fig. 1. Unwinding of state transition graph.

Formulae in CTL are built from atomic propositions (where each proposition corresponds to a variable in the model), standard boolean connectives of propositional logic (e.g., AND, OR, XOR, NOT), and temporal operators. Each temporal operator consists of two parts[1]: a path quantifier **A** or **E** followed by a temporal modality **F**, **G**, **X**, or **U**. All temporal operators are interpreted relative to an implicit "current state". There are in general many execution paths (sequences of state transitions) of the system starting at the current state. The path quantifier indicates whether the modality defines a property that should be true of all those possible paths (denoted by universal path quantifier **A**) or whether the property needs only hold on some path (denoted by existential path

[1] A formula that contains any temporal modality of **F**, **G**, **X**, and **U** without an associated path quantifier **A** or **E** is *not* a legal CTL formula.

quantifier **E**). The temporal modalities describe the ordering of events in time along an execution path and have the following intuitive meaning:

1. **F** ϕ (reads "ϕ holds sometime in the future") is true on a path if there exists a state in the path where formula ϕ is true.
2. **G** ϕ (reads "ϕ holds globally") is true on a path if ϕ is true at every state in the path.
3. **X** ϕ (reads "ϕ holds in the next state") is true on a path if ϕ is true in the state reached immediately after the current state in the path.
4. ϕ **U** ψ (reads "ϕ holds until ψ holds", called "strong until"[2]) is true on a path if ψ is true in some state in the path, and ϕ holds in all preceding states.

The state of a system consists of the values stored in all registers. Each formula of the logic is either true or false in a given state; its truth is evaluated from the truth of its subformulae in a recursive fashion, until one reaches atomic propositions that are either true or false in a given state. A formula is satisfied by a system if it is true for all the initial states of the system. If the property does not hold, the model checker will produce a counterexample, that is an execution path that witnesses the failure. An efficient algorithm for automatic model checking used also in VIS has been described by Clarke et al. [CGMZ95]. The following table shows examples of evaluations of formulae on the computation tree of Figure 1:

formula	T/F
EG (RED)	true
E (RED **U** GREEN)	true
AF (GREEN)	false

1.3 Tools and Representations

Tools for automatic formal verification have been based on various theoretical approaches, of which model checking and language containment have been the most useful in hardware verification. In temporal logic model checking, the properties to be checked are expressed as formulae in a temporal logic, and the system is expressed as a finite-state system. In particular, CTL model checking is a formalism pioneered by Clarke and Emerson [CES86] to verify whether a finite-state system satisfies properties expressed as formulae in CTL, a branching-time temporal logic. SMV [McM93], a system developed at Carnegie Mellon University, belongs to this class of tools. Symbolic trajectory evaluation [HS97] uses a restricted form of linear temporal logic to express properties as bounded-length sequences of circuit states. Verification is done by using an extension of symbolic simulation.

[2] "Weak until" is when ϕ holds forever, i.e., ψ is not required to hold at some state in the future.

Certain properties are not expressible in CTL, but they can be expressed as ω-automata. The second approach, language containment, requires the description of the system and properties as ω-automata, and verifies correctness by checking that the language of the system is contained in the language of the property. Note that certain types of CTL properties involving existential quantification are not expressible by ω-automata. COSPAN [Kur94], a system developed at Bell Labs, offers language containment checking.

VIS [BHSV$^+$96a,BHSV$^+$96b], a system developed jointly at the University of California at Berkeley and the University of Colorado at Boulder emphasizes model checking, but it also offers to the user a limited form of language containment (language emptiness) checking.

The computation engines inside these tools represent and operate with discrete functions, which represent sets of elements rather than individual elements. Such techniques are often referred to as *implicit/symbolic computation*, in contrast to *explicit/enumerative computation*.

Implicit representations allow to operate on sets of states, instead of individual states. A typical example is the computation of the set of reachable states from the initial states. Using binary decision diagrams (BDDs) as an underlying data structure, there are cases when Boolean formulae and operations on them can be performed efficiently even on instances of very huge size, where explicit enumeration would fail. The reason is that the size of BDD representations is not linear in the size of the represented sets. A paper summarized this state of affair with the catchy title: "Symbolic model checking: 10^{20} states and beyond" [BCM$^+$92]. The other side of the coin is that this is not guaranteed to happen, because there may be no such good representation with BDDs, or if it exists, we may be unable to find it (the size of BDDs depends on the order of the support variables). Issues of BDD ordering, partition of the system representation, and new types of decision diagrams have been explored to represent compactly important sets of Boolean and discrete functions.

In the last years, the role played by BDD computations has been restricted due to their lack of robustness; at the same time improvements of SAT solvers (which check the satisfiability of formulae with conjunctions of clauses) made them the preferred computational engines and allowed a more judicious balance in the usage of different function representations. In the rest of the chapter, we will clarify the rationale behind these technical developments.

We can summarize the prevailing data representation and typical design size in successive generation of tools as follows:

1. explicit representation, 10^4 states, 1980s;
2. implicit BDD-based representation, 10^{30} states, 1990s;
3. implicit SAT-based representation, 10^{100} states, 2000s.

In this survey we cannot provide a comprehensive coverage of hardware verification. We will focus on equivalence checking, which is a special case of safety property checking. Even in the restricted domain, we have to leave out some interesting topics and recent advances. An in-depth discussion on aspects of sequential equivalence checking will be given, because it is the most widely encountered case, still a subject of active theoretical and experimental investigation.

In Section 2 we introduce basic terminology and technical background. In Section 3 we discuss the computational complexity of hardware equivalence checking. In Section 4 we introduce combinational equivalence checking. In Section 5 we describe the basics of sequential equivalence checking, which is further expanded in Section 6 introducing bounded and unbounded model checking. Section 7 discusses how to bridge the gap between combinational and sequential equivalence checking. Section 8 covers resetability verification and other variants of equivalence checking. Section 9 concludes mentioning advanced topics and active research areas.

2 Preliminaries

In the sequel, as a notational convention, a vector (or an ordered set) $\boldsymbol{v} = (v_1, \dots, v_n)$ is specified in a bold-faced letter. The cardinality of $\boldsymbol{v}$ is denoted as $|\boldsymbol{v}|$, and the (unordered) set $\{v_1, \dots, v_n\}$ is denoted as $\{\boldsymbol{v}\}$.

Characteristic Functions Progress since the 1980s in expanding the capability of formal verification can be attributed to the introduction of the data structure called Reduced Ordered Binary Decision Diagram (ROBDD) [Bry86], which turned out to be effective in representing and manipulating Boolean functions. ROBDDs allowed to reduce computations on sets to Boolean computations on their *characteristic functions*. A characteristic function, in our discussion, is a (total) function $\chi_A : U \to \mathbb{B}$, where U is a finite set and $\mathbb{B} = \{\mathbf{false}, \mathbf{true}\}$ or $\{0, 1\}$, such that $\chi_A(s) = 1$ ﬀ $s \in A$. (In some applications, it can be extended to multiple-valued output, e.g. $\{\mathbf{true}, \mathbf{false}, \mathbf{undefined}\}$.) When represented in computers, a characteristic function is often a Boolean formula where multiple-valued symbols are encoded and expressed in a binary form. It serves as a predicate indicating some membership problem. That is, the function χ_A answers a query, whether an element $e \in U$ is in $A \subseteq U$. Essentially, any finite set A can be represented with a characteristic function χ_A such that an element $e \in A$ if and only if $\chi_A(e) = \mathbf{true}$. Thus set operations (e.g., intersection $\cap$, union $\cup$, and complement) are in effect Boolean operations (e.g., conjunction $\wedge$, disjunction $\vee$, and negation $\neg$, respectively) over characteristic functions. ($\mathbf{false}$ and $\mathbf{true}$ can be seen as characteristic functions for the empty set $\emptyset$ and universal set U, respectively.) In the sequel, we shall not distinguish a set (respectively a set operation) and its corresponding characteristic function (respectively its corresponding Boolean operation).

By dealing with characteristic functions, we are able to manipulate sets of elements simultaneously rather than manipulate elements individually. For instance, the intersection of two sets A and B can be done by performing $\chi_A \wedge \chi_B$ rather than examine, for each element $e \in A$, whether e is in B as well. Such approaches capable of manipulating sets of elements simultaneously are known as (*implicit*) *symbolic algorithms* in contrast to the traditional (*explicit*) *enumerative algorithms.* Although BDDs and symbolic algorithms were once almost synonyms, more recently other data structures were developed as alternatives to BDDs. Notably, Boolean reasoning engines using SAT (satisfiability solving over conjunctive normal forms) and AIGs (And-Inverter Graphs), for instance, are gaining their popularity in hardware verification.

State Transition Systems We are concerned with the type of state transition system called *finite state machine* (FSM).

Definition 1 (Finite State Machine). *An FSM $\mathcal{M}$ is a tuple $(Q, Q^0, \Sigma, \Omega, \boldsymbol{\delta}, \boldsymbol{\lambda})$, where Q is a finite set of states, $Q^0 \subseteq Q$ is the set of initial states, Σ and Ω are the input and output alphabets, respectively, and $\boldsymbol{\delta} : \Sigma \times Q \to Q$ (respectively $\boldsymbol{\lambda} : \Sigma \times Q \to \Omega$) is the transition function (respectively output function).*

(In our discussion, we focus on Mealy-type FSMs, whose output valuations depend on both input and state variables. The results can be applied to Moore-type FSMs as well, whose output valuations depend only on state variables.) In the sequel, we shall assume that $\boldsymbol{\delta}$ and $\boldsymbol{\lambda}$ are total functions. That is, the FSMs to be discussed are deterministic and completely specified.

In modern hardware designs, state, input and output symbols are encoded by binary representations; transition and output functions are encoded by Boolean functional vectors. Let $\boldsymbol{s} = (s_1, \ldots, s_n)$ and $\boldsymbol{s}' = (s'_1, \ldots, s'_n)$ be the vectors of (binary-valued) current- and next-state variables, respectively; let $\boldsymbol{x} = (x_1, \ldots, x_k)$ and $\boldsymbol{y} = (y_1, \ldots, y_l)$ be the vectors of (binary-valued) input and output variables, respectively. That is, vector $\boldsymbol{s}$ encodes the states Q, $\boldsymbol{x}$ encodes Σ, and $\boldsymbol{y}$ encodes Ω. With notation $[\![\boldsymbol{v}]\!]$ representing the set of all possible valuations (or interpretations) on variables $\boldsymbol{v}$, we have $Q = [\![\boldsymbol{s}]\!]$, $\Sigma = [\![\boldsymbol{x}]\!]$ and $\Omega = [\![\boldsymbol{y}]\!]$. Also, let $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_n)$ and $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_l)$. Then $\delta_i : [\![\boldsymbol{x}]\!] \times [\![\boldsymbol{s}]\!] \to [\![s'_i]\!]$ for $i = 1, \ldots, n$ and $\lambda_j : [\![\boldsymbol{x}]\!] \times [\![\boldsymbol{s}]\!] \to [\![y_j]\!]$ for $j = 1, \ldots, l$. Given two states q_0 and q_1 of an FSM with $q_1 = \boldsymbol{\delta}(\sigma, q_0)$ for some $\sigma \in \Sigma$, q_1 is called the *successor* of q_0 under σ, denoted as $Succ_\sigma(q_0)$, and q_0 is called the *predecessor* of q_1 under σ, denoted as $Pred_\sigma(q_1)$.

An FSM (in the six-tuple form) can be alternatively represented with a graph.

Definition 2 (State Transition Graph). *Given an FSM $\mathcal{M} = (Q, Q^0, \Sigma, \Omega, \boldsymbol{\delta}, \boldsymbol{\lambda})$, it can be represented with a* state transition graph *(STG) $G = (V, E)$, where any state $q \in Q$ is modeled as a vertex $q \in V$ and any transition $q_t = \boldsymbol{\delta}(\sigma, q_s)$ is modeled as a directed edge $(q_s, q_t) \in E$ labelled 'σ/ω' for $\omega = \boldsymbol{\lambda}(\sigma, q)$. Also, for initial states, their corresponding vertices are identified.*

Example 1. Figure 2 shows the schematic diagrams of two FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ and their respective corresponding STGs G_1 and G_2.

Since the behavior of an FSM is described with transition functions, its state transitions are deterministic. That is, given any input assignment and any current state of the FSM, there is a unique next-state. A more general description

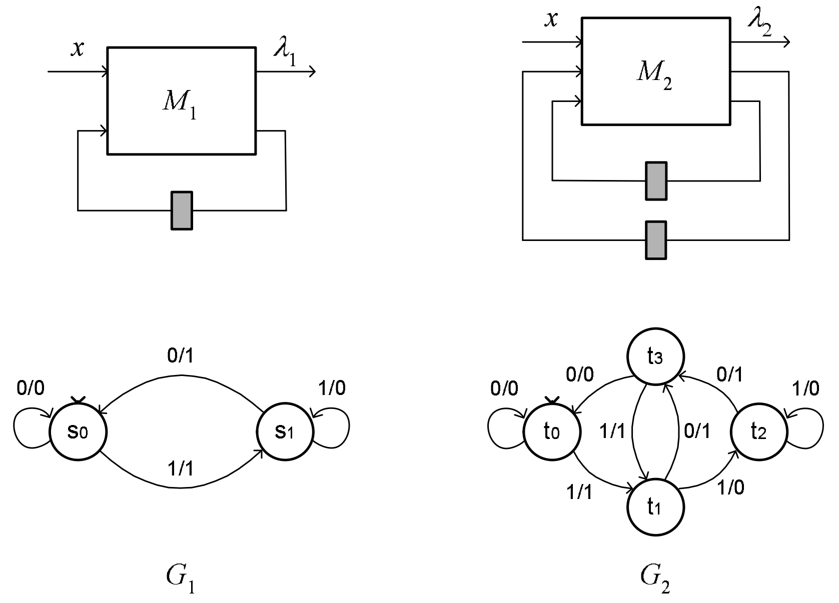


Fig. 2. Two FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ (in schematic diagrams) with their corresponding STGs G_1 and G_2 are shown in (a) and (b), respectively. In a schematic circuit diagram, detailed functions are omitted and only connections are shown, and state-holding elements are denoted with shaded boxes. An initial state in an STG is identified with an arrowhead, e.g., state s_0 in G_1.

is to use *transition relations* instead of *transition functions*. In this case, nondeterministic transitions can be handled. Essentially, a transition function can be converted to a transition relation. We may consider the Cartesian product $\Sigma \times Q$ as the new state space. By treating input variables as part of the state variables, we can write $T((\boldsymbol{x}, \boldsymbol{s}), (\boldsymbol{x}', \boldsymbol{s}')) = \bigwedge_i (s'_i \equiv \delta_i(\boldsymbol{x}, \boldsymbol{s}))$, where the equality sign "$\equiv$" means exclusive-nor, often denoted as $\overline{\oplus}$, in Boolean operation. Since input variables $\boldsymbol{x}'$ of the next time frame are unconstrained, we write $T(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}')$ in place of $T((\boldsymbol{x}, \boldsymbol{s}), (\boldsymbol{x}', \boldsymbol{s}'))$ for short. Consequently, FSMs can be described with transition relations in addition to transition functions. In the sequel, we may alternatively use the more general transition relation notation, and describe an FSM with the tuple $(Q, Q^0, \Sigma, \Omega, T, \boldsymbol{\lambda})$.

To study the behavior between two FSMs subject to the same input stimuli, we define a product machine as follows.

Definition 3 (Product FSM). *Given two FSMs* $\mathcal{M}_1 = (Q_1, Q_1^0, \Sigma, \Omega, \boldsymbol{\delta}_1, \boldsymbol{\lambda}_1)$ *and* $\mathcal{M}_2 = (Q_2, Q_2^0, \Sigma, \Omega, \boldsymbol{\delta}_2, \boldsymbol{\lambda}_2)$*, the* product FSM *of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *is* $\mathcal{M}_\times = (Q_1 \times Q_2, Q_1^0 \times Q_2^0, \Sigma, \mathbb{B}, \boldsymbol{\delta}_\times, \boldsymbol{\lambda}_\times)$ *with*

$$\boldsymbol{\delta}_\times(\boldsymbol{x}, (\boldsymbol{s}_1, \boldsymbol{s}_2)) = (\boldsymbol{\delta}_1(\boldsymbol{x}, \boldsymbol{s}_1), \boldsymbol{\delta}_2(\boldsymbol{x}, \boldsymbol{s}_2)) \text{ and}$$

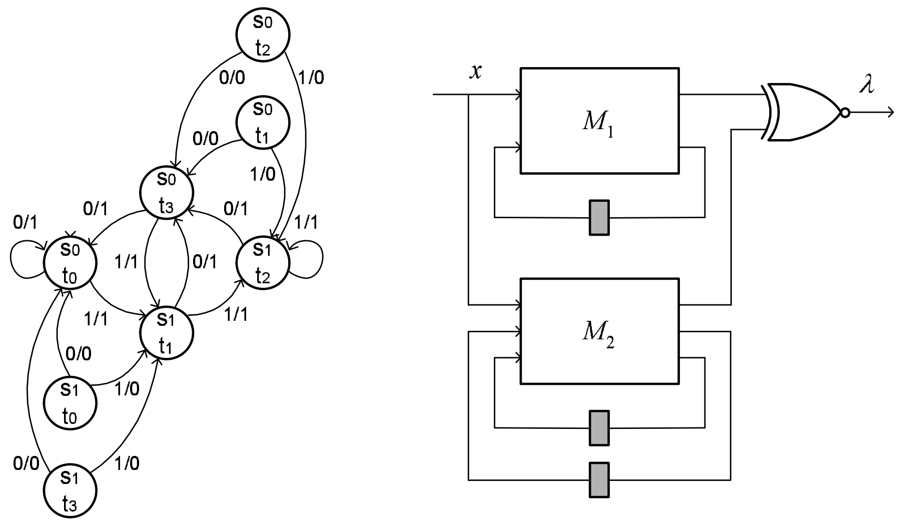


Fig. 3. The product FSM of $\mathcal{M}_1$ and $\mathcal{M}_2$ of Figure 2, and its corresponding STG.

$$\boldsymbol{\lambda}_\times(\boldsymbol{x},(\boldsymbol{s}_1,\boldsymbol{s}_2)) = \bigwedge_i(\lambda_{1i}(\boldsymbol{x},\boldsymbol{s}_1) \equiv \lambda_{2i}(\boldsymbol{x},\boldsymbol{s}_2)),$$

where $\boldsymbol{x}$ is the input variable vector, and $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ are the state variable vectors of $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively.

We call the STG $G_\times$ of $\mathcal{M}_\times$ as the *product STG* of the STGs of $\mathcal{M}_1$ and $\mathcal{M}_2$. Notice that by building the product FSM the state space may increase substantially.

A product FSM $\mathcal{M}_\times$ is such that its two constituent FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ are executed synchronously with the same input stimuli while their outputs are compared. The output of $\mathcal{M}_\times$ equals 1 if the outputs of $\mathcal{M}_1$ and $\mathcal{M}_2$ are the same. Otherwise, 0 is produced. Recall and observe that the miter structure in combinational equivalence checking can be considered as a special case of a product FSM (with a single state).

Example 2. Figure 3 shows the product FSM $\mathcal{M}_\times$ of the FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ in Figure 2, and the corresponding STG of $\mathcal{M}_\times$.

To model the disjoint union of two FSMs, we define

Definition 4 (Multiplexed FSM). *Given two FSMs $\mathcal{M}_1 = (Q_1, Q_1^0, \Sigma, \Omega, \boldsymbol{\delta}_1, \boldsymbol{\lambda}_1)$ and $\mathcal{M}_2 = (Q_2, Q_2^0, \Sigma, \Omega, \boldsymbol{\delta}_2, \boldsymbol{\lambda}_2)$, the* multiplexed FSM *of $\mathcal{M}_1$ and $\mathcal{M}_2$ is $\mathcal{M}_\uplus = (Q_1 \uplus Q_2, Q_1^0 \uplus Q_2^0, \Sigma, \Omega, \boldsymbol{\delta}_\uplus, \boldsymbol{\lambda}_\uplus)$ with*

$$\boldsymbol{\delta}_\uplus(\boldsymbol{x},(\boldsymbol{s}_\alpha,\alpha)) = \begin{cases} \boldsymbol{\delta}_1(\boldsymbol{x},\boldsymbol{s}_1) \text{ if } \alpha = 1 \\ \boldsymbol{\delta}_2(\boldsymbol{x},\boldsymbol{s}_2) \text{ if } \alpha = 2 \end{cases}, \text{ and}$$

$$\boldsymbol{\lambda}_\uplus(\boldsymbol{x},(\boldsymbol{s}_\alpha,\alpha)) = \begin{cases} \boldsymbol{\lambda}_1(\boldsymbol{x},\boldsymbol{s}_1) \text{ if } \alpha = 1 \\ \boldsymbol{\lambda}_2(\boldsymbol{x},\boldsymbol{s}_2) \text{ if } \alpha = 2 \end{cases}$$

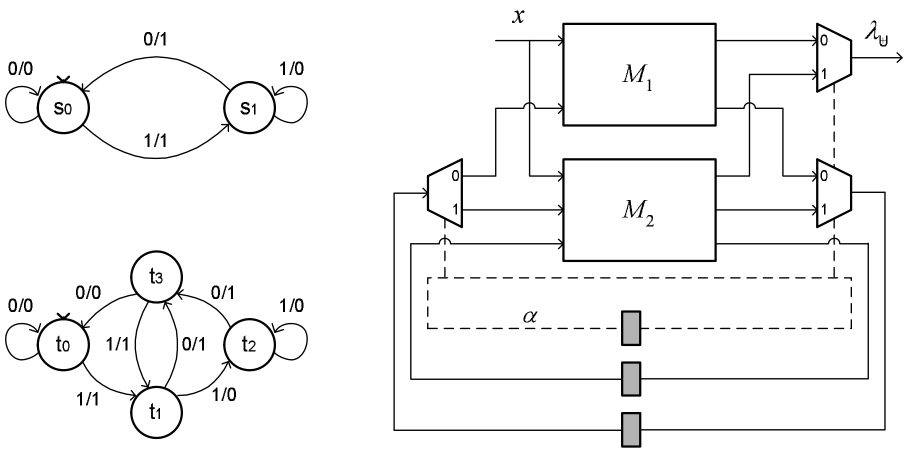


Fig. 4. The multiplexed FSM of $\mathcal{M}_1$ and $\mathcal{M}_2$ of Figure 2, and its corresponding STG.

where symbol $\uplus$ *denotes the* disjoint union *operator,* $\alpha = \{1, 2\}$ *acts as a machine indicator,* $\boldsymbol{x}$ *is the input variable vector, and* $\boldsymbol{s}_1$ *and* $\boldsymbol{s}_2$ *are the state variable vectors of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, respectively.*

We call the STG $G_\uplus$ of $\mathcal{M}_\uplus$ as the *disjoint union STG* of the STGs of $\mathcal{M}_1$ and $\mathcal{M}_2$. Notice that the state space of a multiplexed FSM is the disjoint union of those of its two constituent FSMs and it is usually much smaller than the state space of the product FSM.

Suppose that $\mathcal{M}_1$ and $\mathcal{M}_2$ have m_1 and M_2 registers respectively and that $m_2 \geq m_1$. To minimize the state variables of the multiplexed machine, we pair arbitrarily every next state variable of $\mathcal{M}_1$ with one of $\mathcal{M}_2$. The pair is then multiplexed before being fed to a register, whose output is then demultiplexed to recover the current state variables for $\mathcal{M}_1$ and $\mathcal{M}_2$. In addition one self-looped auxiliary state variable is added, α, which controls all multiplexers, as indicated by the dotted lines in Figure 4; the value α stays constant as its initial value.

A multiplexed FSM includes its two constituent FSMs and acts as $\mathcal{M}_1$ or $\mathcal{M}_2$ depending on the machine indicator α. Although a multiplexed FSM is not as intuitive as a product FSM, its usefulness in equivalence verification will be demonstrated later in Section 5.

Example 3. Figure 4 shows the multiplexed FSM $\mathcal{M}_\uplus$ of $\mathcal{M}_1$ and $\mathcal{M}_2$ of Figure 2, and the corresponding STG of $\mathcal{M}_\uplus$. Note that the pairing of transition functions between $\mathcal{M}_1$ and $\mathcal{M}_2$ for the inputs of a multiplexor can be arbitrary, whereas that of output functions needs to be the corresponding pairs. When $\alpha = 0$ (respectively $\alpha = 1$), $\mathcal{M}_\uplus$ behaves like $\mathcal{M}_1$ (respectively $\mathcal{M}_2$) of Figure 2. In the STG of $\mathcal{M}_\uplus$, states with $\alpha = 0$ are $\{s_0, s_1\}$, and those with $\alpha = 1$ are $\{t_0, \ldots, t_3\}$.

Image and Pre-image Computation There are two important operations that are key ingredients to formal hardware verification. The *image* of state set $C \subseteq \Sigma \times Q$, which depends on both input variables $\boldsymbol{x}$ and current-state variables $\boldsymbol{s}$, with respect to transition relation T is computed as

$$Img(C, T) = [\exists \boldsymbol{x}, \boldsymbol{s}.(T(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}') \wedge C(\boldsymbol{x}, \boldsymbol{s}))]_{\boldsymbol{s}' \leftarrow \boldsymbol{s}},$$

where the above subscript denotes the substitution of $\boldsymbol{s}$ in place of $\boldsymbol{s}'$ in order that the resulting image depends on current-state variables rather than next-state variables. (Rigorously speaking, the above $C(\boldsymbol{x}, \boldsymbol{s})$ means $\chi_C(\boldsymbol{x}, \boldsymbol{s})$.) It characterizes the set of states that can be reached under the inputs and current states constrained by C. On the other hand, the *pre-image* of $C^\dagger$ over next-state variables with respect to transition relation T can be computed as

$$PreImg(C^\dagger, T) = \exists \boldsymbol{x}, \boldsymbol{s}'.(T(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}') \wedge C^\dagger(\boldsymbol{s}')).$$

It characterizes the set of states that can reach $C^\dagger$ in one step under some input assignments to variables $\boldsymbol{x}$.

Given initial states I and an input sequence $\boldsymbol{\sigma} = \sigma_1, \ldots, \sigma_n$ for $\sigma_i \in \Sigma$ the image computation can be applied to obtain the destination states D with respect to a transition relation T as follows.

$$\begin{aligned} D_0 &= I(\boldsymbol{s}) \\ &\vdots \\ D_n &= Img(\sigma_n(\boldsymbol{x}) \wedge D_{n-1}(\boldsymbol{s}), T(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}')) \end{aligned}$$

Then the set of states D_n equals D. We denote the destination states D of initial states I under input sequence $\boldsymbol{\sigma}$ with respect to transition relation T as $D = Img_{\boldsymbol{\sigma}}(I, T)$.

Definition 5. *A state set $C \subseteq Q$ is called* closed *under a transition relation T if $\{C \cup Img(C, T)\} \subseteq C$. Otherwise, C is* open.

That is, a closed state set C cannot transition out of C under any inputs.

Let $\boldsymbol{\sigma}_1$ and $\boldsymbol{\sigma}_2$ be two input sequences. We denote their concatenation as $\boldsymbol{\sigma}_1 \circ \boldsymbol{\sigma}_2$, meaning that $\boldsymbol{\sigma}_1$ is applied first and then followed by $\boldsymbol{\sigma}_2$.

3 Computational Complexity of Equivalence Checking

3.1 Complexity of Combinational Equivalence Checking

Combinational equivalence checking (CEC) is basically tautology checking of Boolean expressions.

CEC of Boolean Circuits
INSTANCE: Two combinational Boolean circuits C_1 and C_2.
QUESTION: Are the output values of C_1 and C_2 the same for each input vector? That is, are C_1 and C_2 equivalent?

Theorem 1. *CEC is coNP-complete.*

The proof is by reduction from validity, i.e., given an instance of validity checking of formula ϕ, we build polynomially an instance of CEC as follows: C_1 is the natural circuit representation of the formula ϕ and C_2 is any circuit representing the function constant 1. Then C_1 is equivalent to C_2 if and only if ϕ is valid. If a formula ϕ is not valid, then it can be disqualified succinctly by providing a truth assignment that does not satisfy ϕ. On the other hand, a valid ϕ has no such disqualification. VALIDITY is also called TAUTOLOGY. Checking the VALIDITY of a Boolean expression in disjunctive normal form (DNF) is coNP-complete. Any language $\mathcal{L}$ in coNP is reducible to VALIDITY. A string x is in $\mathcal{L}$ if and only if the formula φ_x in DNF by the reduction from $\mathcal{L}$ to VALIDITY is valid. Indeed, if $\mathcal{L}$ is in coNP, then its complement $\overline{\mathcal{L}}$ is in NP. Hence there is a reduction from $\overline{\mathcal{L}}$ to SAT. A string x is in $\overline{\mathcal{L}}$ if and only if the formula φ_x in conjunctive normal form (CNF) by the reduction is satisfiable.

3.2 Complexity of Sequential Equivalence Checking

The complexity measure of sequential equivalence checking (SEC) depends on how state-transition systems are represented. When they are represented explicitly by state-transition graphs, the corresponding SEC is tractable in terms of the size of the graphs. When they are represented by sequential circuits (with logic gates and memory elements), the corresponding SEC is intractable in the size of the circuits.

SEC of Deterministic Automata (explicit graph representation)
instance: Two deterministic finite automata A_1 and A_2.
question: Are the output languages produced by A_1 and A_2 the same?

The problem is in P (solvable in polynomial time in the size of the input graph), because we can perform state minimization of A_1 and A_2. It is well-known that, given a finite automaton or finite transducer, state minimization yields a unique minimum automaton up to state renaming. So the SEC problem is reduced to verifying the isomorphism of the two state-minimized automata. Notice that graph isomorphism is not an easy problem; however comparing the graphs of the two minimized automata is easy because they are special graphs with a unique starting state and unique edge labelling for every state.

The SEC problem becomes harder when the state-transition system is not given explicitly via a graph, but implicitly via a circuit (or a program) that encodes the graph. So a sequential circuit can be seen as a succinct representation of a graph $G = (V, E)$ with states V and (labelled) transitions $E \subseteq V \times V$. An edge $(s, t) \in E$ with label a denotes that t is the next state of s under input a. In this way we are encoding an exponentially large graph into a small circuit, but the reachability problem on the succinct graph representation jumps in complexity from P to PSPACE.

SEC of Sequential Circuits (implicit graph representation)
instance: Two sequential Boolean circuits C_1 and C_2 with the same input and

output variables, and with respective initial states s_1 and s_2.
QUESTION: Are the output sequences of C_1 and C_2 the same for every sequence of input vectors?

The negation of the above problem can be stated as follows.

SNEC of Sequential Circuits (implicit graph representation)
INSTANCE: Two sequential Boolean circuits C_1 and C_2 with the same input and output variables, and with respective initial states s_1 and s_2.
QUESTION: Is there a sequence of input vectors such that C_1 and C_2 differ in at least one output value?

Theorem 2. *SNEC is PSPACE-complete.*

Proof. We show first that it is in PSPACE, because the following polynomial-space algorithm decides the language. For convenience, consider the product machine of C_1 and C_2 with initial state $s = (s_1, s_2)$. The algorithm first guesses the length l of a path from s providing a counter-example to equivalence, and then guesses its $l-2$ inner vertices and the final vertex t (vertices are states of the product machine).

1. Guess a number $l \in \{2, 3, \ldots, 2^n\}$, where n is the the number of storage elements in the product machine (i.e., the sum of the number of storage elements in C_1 and C_2), and so 2^n is an upper bound on the length of a counter-example to equivalence.
2. Set $k = 1$.
3. Repeat until k is equal to $l - 1$:
   (a) Guess $v_k \in \mathbb{B}^n$ (a state of the product machine) and an input vector i.
   (b) Simulate the product machine of C_1 and C_2 from state v_{k-1} under i:
   If v_k is not the next state of v_{k-1} (with $v_0 = s$) under i "reject",
   else if C_1 and C_2 differ at v_k in at least one output "accept" ($v_k = t$)
   else if $k = l - 1$ "reject".
   (c) Set $k = k + 1$.

The above simulation of the product machine takes polynomial space.

SNEC is PSPACE-hard. Let $\mathcal{L}$ be an arbitrary language in PSPACE and $M_{\mathcal{L}}$ be a Turing machine[3] deciding $\mathcal{L}$ using space no more than *poly*(n), polynomial in the size n of the input given to $M_{\mathcal{L}}$. We show that $\mathcal{L} \leq_p$ SNEC, namely, $\mathcal{L}$ is polynomial-time reducible to SNEC.

The idea of the reduction is to build in polynomial time a circuit that is an implicit representation of the configuration graph of $M_{\mathcal{L}}$. The vertices of the configuration graph are represented by the encoded vectors stored in the memory elements of the sequential circuit. The number of memory elements is a polynomial in n because each configuration can grow at most as a polynomial in n.

[3] We assume that $M_{\mathcal{L}}$ is a deterministic Turing machine, due to the theorem by Savitch [Sav70], stating that a non-deterministic Turing machine using $f(n)$ space can be simulated by a deterministic Turing machine using $f^2(n)$ space for any polynomial function f. So the simulation preserves polynomial space.

Given an input string w to $M_{\mathcal{L}}$ with $|w| = n$ (size of the input), the possible configurations of $M_{\mathcal{L}}$ when running on w are of length at most $poly(n)$. Each configuration can be represented as a sequence of $poly(n)$ cells where each cell contains either a symbol from the work alphabet of $M_{\mathcal{L}}$ or a symbol from the state set. Therefore each cell can be represented by a constant number c of bits (that does not depend on the input length) and the entire configuration can be represented by a sequence of $m = c \cdot poly(n)$ bits. In addition we define a vector of m bits that does not represent a configuration, but instead means "accepted" and another vector of m bits meaning "rejected".

Given w, let us see in detail how to build an instance of SNEC.

First we construct a sequential circuit C_1^w that emulates a run of $M_{\mathcal{L}}$ on w. This sequential circuit has m storage elements (to store the configurations of $M_{\mathcal{L}}$ running on w) and logic circuitry computing the next configuration yielded by the current configuration. The circuitry will implement with logic gates the transition relation of the Turing machine $M_{\mathcal{L}}$, in practice a look-up table will do. Two states correspond to the vectors for "accepted" and "rejected", respectively. The initial state s_1 of C_1^w corresponds to the initial configuration of $M_{\mathcal{L}}$ on w. Moreover, C_1^w outputs 1 in every state. We can build in polynomial time the sequential circuit C_1^w that works in polynomial space. Notice that this sequential circuit is autonomous, i.e., it has no inputs, except the clock.

Finally, to complete the construction of the instance of SNEC, we build C_2^w as a copy of C_1^w, which differs from C_1^w only in the fact that it outputs 0 in the accepted state of C_2^w. As before, the initial state s_2 of C_2^w corresponds to the initial configuration of $M_{\mathcal{L}}$ on w.

In summary, we established PSPACE-hardness, because

1. The overall reduction can be done in polynomial time.
2. $M_{\mathcal{L}}$ accepts w if and only if, starting respectively from s_1 and s_2, C_1 and C_2 reach a state where they differ in at least one output value.

■

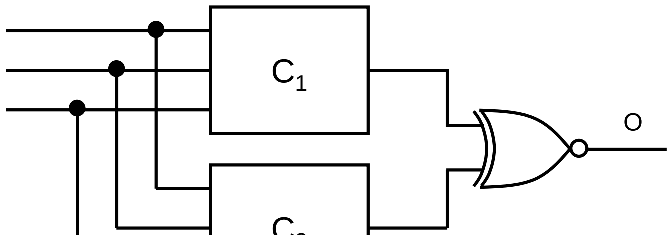
The fact that SNEC is PSPACE-complete implies by definition that its complement problem SEC is coPSPACE-complete. By a theorem of Immerman [Imm88] and Szelepscenyi [Sze88], it holds that NSPACE = coNSPACE; by a theorem of Savitch [Sav70] it holds that PSPACE = NSPACE. So it follows that PSPACE = coNSPACE = coPSPACE.

Corollary 1. *SEC is PSPACE-complete.*

It was proved recently in [JB06] that even a restricted case of sequential equivalence checking where the circuits differ only by retiming and resynthesis transformations it is still PSPACE-complete (by reduction from finite function generation [GJ79]).

4 Combinational Equivalence Checking

Given two combinational Boolean circuits C_1 and C_2, the problem is to check whether the corresponding outputs of the two circuits are the same for every



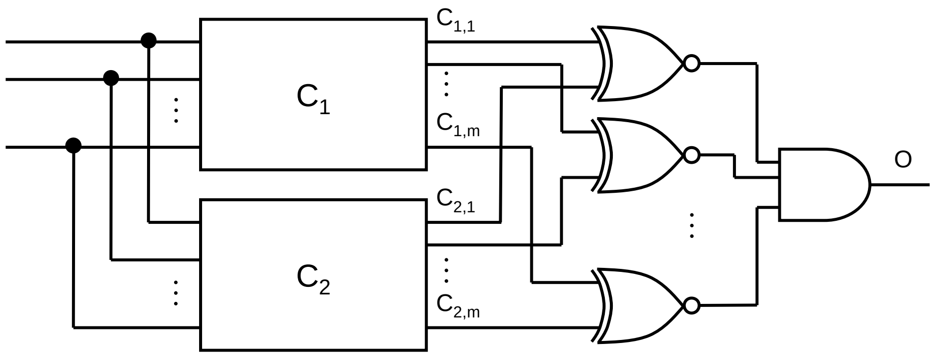


Fig. 6. Multi-output miter circuit from networks C_1 and C_2.

input vector $\boldsymbol{i} = (i_1, \ldots, i_n)$. For single-output circuits, their equivalence can be expressed as

$$\forall \boldsymbol{i}.(C_1(\boldsymbol{i}) \equiv C_2(\boldsymbol{i})). \tag{1}$$

For multi-output circuits with outputs $C_{1,j}$ and $C_{2,j}$, where $j = 1, \ldots, m$, their equivalence can be expressed as:

$$\forall \boldsymbol{i}. \bigwedge_{j=1}^{m} (C_{1,j}(\boldsymbol{i}) \equiv C_{2,j}(\boldsymbol{i})). \tag{2}$$

The tautology checking of Formulas (1) and (2) can be achieved through the so-called "miter" circuit construction [Bra93] as shown in Figures 5 and 6, respectively. A counterexample to equivalence exists if and only if there is an input vector that makes the output of a miter circuit false. Observe that the miter circuit of CEC is similar to the product machine of SEC.

The tautology checking can be alternatively negated for satisfiability checking. Technically speaking, it makes no difference between tautology checking and satisfiability checking. One can dually define the miter circuit construction by replacing the XNOR-gates with XOR-gates and by replacing the final AND-gate with an OR-gate. Then the previous statements are dualized in that circuit equivalence is the same as the miter circuit being satisfiable at the output.

We will review the foundations of the two basic approaches to combinational equivalence checking: functional techniques and structural techniques. Surveys of CEC techniques can be found in [JNFSV97] for early work, and in [KB02,FKS05] for more recent work.

4.1 Functional Combinational Equivalence Checking

In functional combinational equivalence checking, the functions realized by the two circuits are compared directly by representing them with canonical representations (they guarantee a unique form for a given function). Since Bryant's influential paper [Bry86], ROBDDs have been the most widely used canonical data structure for Boolean functions. For instance, given the miter construction in Figure 6, one builds the ROBDD for $O = \bigwedge_j (C_{1,j} \equiv C_{2,j})$. If the BDD represents constant 1, C_1 and C_2 are equivalent. Otherwise, C_1 and C_2 are not. Notice that the ROBDD representing the constant-1 function has a unique representation by the constant-1 terminal node.

A major problem is that ROBDDs may be unable to represent circuits with a large number of primary inputs, and in general their growth is not bounded *a priori* by tight bounds. To overcome this problem, structural properties of the multi-level representations of the circuits must be exploited in the equivalence check.

4.2 Structural Combinational Equivalence Checking

Structural techniques exploit the circuit gate-level representation to verify equivalence. Two common structural techniques are based on SAT solving and Automatic Test Pattern Generation (ATPG) [KMSM01].

Combinational Equivalence by SAT Given the miter of Figure 5, its output produces 0 under some input truth assignment if and only if the two circuits C_1 and C_2 represent different Boolean functions. A SAT solver can be used to search a counterexample to equivalence.

To apply SAT we must translate a circuit structure into a CNF, by introducing new variables for all circuit nodes and adding relational constraints in CNF to model the functionality of each gate (as originally proposed by Tseitin [Tse70]). The translation is linear in the number of gates if the fanin sizes of XOR-gates and XNOR-gates are upper bounded by a constant.

Example 4. Consider the circuit of Figure 7. By Tseitin's circuit-to-CNF conversion, in addition to the primary input variables a, b, and c, variables x, y,

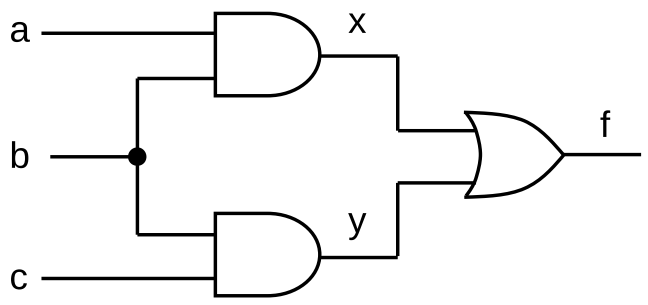


Fig. 7. A combinational circuit for circuit-to-CNF conversion.

and f are added to represent the logic values of the gate outputs. The CNFs of the upper AND-gate, lower AND-gate, and OR-gate are

$$(\overline{a} \vee \overline{b} \vee x)(a \vee \overline{x})(b \vee \overline{x}), \tag{3}$$

$$(\overline{b} \vee \overline{c} \vee y)(b \vee \overline{y})(c \vee \overline{y}), \text{ and} \tag{4}$$

$$(x \vee y \vee \overline{f})(\overline{x} \vee f)(\overline{y} \vee f), \text{ respectively.} \tag{5}$$

The CNF of the whole circuit is simply the conjunction of Formulas (3), (4), and (5), namely,

$$(\overline{a} \vee \overline{b} \vee x)(a \vee \overline{x})(b \vee \overline{x})(\overline{b} \vee \overline{c} \vee y)(b \vee \overline{y})(c \vee \overline{y})(x \vee y \vee \overline{f})(\overline{x} \vee f)(\overline{y} \vee f).$$

Even though SAT is NP-complete, much progress has been made into engineering methods that solve instances of practical interest. A modern SAT solver may deploy a mixture of *backtrack search*, *implication (Boolean constraint propagation)*, *conflict analysis*, *resolution*, *non-chronological backtracking (conflict-based learning)*, and *two-literal watching*. We refer to specialized literature to study the topic.

Combinational Equivalence by ATPG Automatic Test Pattern Generation (ATPG) has the goal to generate a test for every fault in a circuit. Under the stuck-at-fault model, for a test to exist there must be a set of input assignments such that the fault under test can be

1. activated, i.e., the faulty stuck-at-1 signal is set to 0 and stuck-at-0 signal to 1, and
2. observed at the primary outputs, i.e., the faulty effect at the fault location propagates to at least one primary output.

The connection between ATPG and equivalence checking is that, given the miter circuit of Figure 6, if the stuck-at-1 fault at O is not testable, then C_1 is equivalent to C_2. Otherwise the test patterns are counterexamples to the assertion of equivalence.

Plain SAT- and ATPG-based CEC tend to fail if the size of the miter circuit is large. A key idea to alleviate this problem is to exploit similarities between C_1 and C_2 for miter circuit reduction. The similarities are likely to exist when many internal nodes in C_1 and C_2 are equivalent. It is expected that the circuits to be compared share structural and functional similarities, because usually CEC is applied to verify the correctness of incremental synthesis steps. So, for each node n_1 in a given list of good candidate nodes in C_1, and for each node n_2 in a given list of good candidate nodes in C_2, one builds the miter circuit $z = (n_1 \equiv n_2)$. If z stuck-at-1 is not testable or SAT does not find a falsifying assignment, then n_1 and n_2 are equivalent. Otherwise, a counterexample is found and they are inequivalent. Identification of the internal equivalences is intrinsically a difficult problem. It is usually done by scanning the circuit from inputs to outputs, with a variety of methods from local analysis to comparison of local BDDs.

In identifying equivalent nodes, if n_1 and n_2 turn out to be equivalent, then there can be different strategies to simplify the miter circuit. One (safe) method is to merge n_1 and n_2 by replacing n_2 with n_1. Another more aggressive (but unsafe) method is to substitute a free variable for both n_1 and n_2 in the miter circuit. The latter may result in false negatives, that is, C_1 and C_2 may be declared inequivalent even if they are equivalent, because the newly added variables are unconstrained and allow truth assignments that are infeasible due to the constraints imposed by the sub-circuits of n_1 and n_2. To fix this problem, one has to partition carefully the circuit and in the worst-case enlarge the partition to include the primary inputs. Or, one can apply range-preserving parametric representation for miter circuit rewriting.

There is a close relationship between ATPG and SAT [KMSM01]. On the one hand, ATPG can be seen as SAT solving directly over circuits rather than CNF formulas. On the other hand, ATPG can be reformulated as standard SAT solving by expressing the constraints of fault-activation and observation as a CNF formula. ATPG and SAT have their own strengths and weaknesses. ATPG has better circuit information for Boolean reasoning; SAT has a simpler and more generic data structure for Boolean reasoning. In practice, efficient hybrid methods deploying features of both ATPG and SAT have been developed.

4.3 Trading Off Canonicity vs. Structure

Between the two extremes of functional equivalence checking with canonical representations, like ROBDDs, which do not scale well, and structural equivalence checking directly on the network, a middle ground has been found with "intermediate" representations that store efficiently the multi-level network by using structural hashing to detect common subnetworks.

One such data structure is AND-Inverter graphs (AIGs), which are Boolean networks composed of two-input AND-gates and inverters represented as flipped bits on the edges [KPKG02]. AIGs are a multi-level logic representation whose construction time and size are proportional linearly to the size of the circuit. AIGs are not canonical. That is, a Boolean function has many AIG representations. For example, function $F = abc$ can be represented as $((ab)c)$, $(a(bc))$,

$((ac)(bc))$, etc. However, when comparing circuits C_1 and C_2, structural graph hashing can be applied to their AIGs to identify structurally identical subgraphs in the miter structure. The AIG representation of the miter is constructed from the inputs to the outputs in topological order, as it is done with BDDs; before adding a new AND vertex, a hash-lookup checks for the existence of a structurally equivalent vertex, and, if it exists, uses it to realize the current AND [KvE01].

Since AIGs are not canonical, internal nodes of an AIG may have equivalent functionality. This increases the number of AIG nodes and makes reasoning on the graph structure inefficient and time consuming. An algorithm to detect and merge equivalent nodes (called functional reduction) during the process of AIG construction has been presented in [MCJB05]; this construction yields a functional representation called Functionally Reduced AIGs (FRAIGs). FRAIGs are "semi-canonical" because no two nodes in a FRAIG have the same function in terms of the primary inputs, but the same function may have different FRAIG structures. The construction uses simulation and SAT.

It is the small set of base functions (two-input AND-gates and inverters) that makes AIGs structure hashing efficient and makes them preferable in scalable Boolean function manipulation.

There are other noncanonical representations, such as Boolean Expression Diagrams (BEDs) and extended BDDs (XBDDs), that use richer sets of base functions. The BED data structure [HWA99] is obtained by extending the ROBDD representation with operator vertices, besides variable vertices; a variable vertex (inherited from standard BDDs) has as attributes a variable and two child vertices, whereas an operator vertex has as attributes a binary Boolean operator and two child vertices. Any Boolean circuit can be transformed into a BED by a translation linear in size. For instance, since there are combinational circuits implementing multiplication using only a quadratic number of gates, there are BEDs of this size representing them. Moreover, BEDs can recognize and share identical sub-expressions. The other side of the coin is that BEDs are not canonical. In contrast, XBDDs [PHS94] are obtained by adding structural variables which can be both universally and existentially quantified. Quantifications are described as annotations on pointers leading to nodes with structural variables.

5 Sequential Equivalence Checking

In the design process of a hardware system, a design may be transformed (manually by circuit designers or automatically by software tools) back and forth to explore an optimal implementation with respect to some design criteria. These transformations may introduce errors in the design. It is crucial to ensure that the final optimized design is indeed equivalent to the original one. Therefore, equivalence verification is one of the most important problems in ensuring the correctness of hardware designs.

When two circuits under comparison contain no state-holding elements (registers), the corresponding verification problem is called *combinational equivalence* *checking*; in contrast, when the circuits under comparison contain registers, the corresponding verification problem is called *sequential equivalence checking*. In fact, combinational equivalence checking can be seen as a special case of sequential equivalence checking since a combinational circuit can be modeled as a single-state finite state machine. We argued in Section 3 that the computational complexity of combinational equivalence checking is coNP-complete whereas that of sequential checking is PSPACE-complete. Even though combinational equivalence is likely to be *intractable* [GJ79] (not solvable in deterministic polynomial time) in theory, it is considered to be efficiently solvable in practice in most design instances (except for arithmetic circuits). As a matter of fact, the equivalence of combinational circuits of multi-million logic gates has been demonstrated, e.g., [KK97]. This apparent contradiction arises from the fact that, in real-world designs, two circuits under comparison mostly possess structural similarities to a large extent. This structural information provides hints assisting equivalence checking. Nevertheless, structural similarities in sequential equivalence checking may not be as helpful as in the combinational case. As a result, there is still no efficient solution to sequential equivalence checking even in practical applications. In the section we study approaches to general sequential equivalence checking; in the next two sections we will discuss some reduction techniques to make sequential equivalence checking more like a combinational one.

To begin with, we define

Definition 6 (k-step State Equivalence). *Two states q_1 and q_2 of an FSM $(Q, Q^0, \Sigma, \Omega, T, \boldsymbol{\lambda})$ are* k-step equivalent*, denoted as $q_1 \sim^k q_2$, if they satisfy the relation $\sim^k$ that is defined inductively as follows:*

$$\begin{aligned} q_1 \sim^0 q_2 :& \quad \forall \sigma \in \Sigma.(\boldsymbol{\lambda}(\sigma, q_1) \equiv \boldsymbol{\lambda}(\sigma, q_2)) \\ &\vdots \\ q_1 \sim^k q_2 :& \quad (q_1 \sim^{k-1} q_2) \wedge \forall \sigma \in \Sigma.(Img_\sigma(q_1, T) \sim^{k-1} Img_\sigma(q_2, T)) \end{aligned}$$

It can be easily checked that $\sim^k$ is reflexive, symmetric and transitive, and thus forms an equivalence relation. Moreover, when viewed as a set, $\sim^k$ is monotonically decreasing, i.e., $\sim^k \supseteq \sim^{k+1}$, for $k = 0, 1, \ldots$ because $(q_1, q_2) \in \sim^{k+1}$ implies $(q_1, q_2) \in \sim^k$. In particular, if $\sim^k = \sim^{k+1}$ for some k, then $\sim^{k+i} = \sim^{k+i+1}$ for $i = 0, 1, \ldots$. Such $\sim^k$, denoted as $\sim^*$, is known as the greatest fixed-point of the k-step state equivalence of an FSM.

The partition induced by $\sim^*$ is one instance of the so-called *stable partition*.

Definition 7 (Stable Partition). *A partition π is* stable *with respect to a transition relation T, if for any q_1, q_2 in the same block of π and $(q_1, q_1') \in T$, then there exists q_2', with $(q_2, q_2') \in T$, in the same block as q_1'.*

The above fixed-point computation is a process of *stabilization* for a given arbitrary initial partition (not necessarily the partition $\sim^0$ induced by the observational equivalence due to the outputs $\boldsymbol{\lambda}$). In fact, the fixed point derived in this way is the *coarsest stable partition* that refines the initial partition. (A partition

π_1 *refines* another π_2 if any two elements in a block of π_1 are also in a block of π_2.)

Relation $\sim^*$ is equivalent to the state equivalence defined below.

Definition 8 (State Equivalence). *Two states q_1 and q_2 of an FSM $\mathcal{M} = (Q, Q^0, \Sigma, \Omega, T, \boldsymbol{\lambda})$ are* equivalent*, denoted as $q_1 \sim q_2$, if $\boldsymbol{\lambda}(\sigma', Img_{\boldsymbol{\sigma}}(q_1, T)) = \boldsymbol{\lambda}(\sigma', Img_{\boldsymbol{\sigma}}(q_2, T))$ for every $\sigma' \in \Sigma$ and every finite (including empty) input sequence $\boldsymbol{\sigma} \in \Sigma^*$.*

That is, starting from either q_1 or q_2 with $q_1 \sim q_2$, an FSM behaves indistinguishably from its input-output behavior. It can be shown that $\sim$ and $\sim^*$ are identical. As a matter of fact, $\sim^*$ can be seen as an operational definition of $\sim$. It allows a finite computation of state equivalence. (As a side note, it is possible to define other notions of state equivalence, whose distinguishing capabilities may induce a preorder $\preceq$ on different equivalence definitions. For instance, equivalence relation $\sim_S$, with $q_1 \sim_S q_2$ if and only if $q_1 = q_2$, is the most distinguishing state equivalence relation, and thus is on the top of the preorder. On the other hand, equivalence relation $\sim_U$, with $q_1 \sim_U q_2$ for any states q_1, q_2, is the least distinguishing state equivalence relation, and thus is at the bottom of the preorder. In comparison, we see that $\sim_U \preceq \sim \preceq \sim_S$ among other possible definitions of state equivalences.)

The state equivalence of an FSM can be straightforwardly generalized to state equivalence among multiple FSMs by treating the disjoint state spaces as a single large state space. On the other hand, we may define FSM equivalence as follows.

Definition 9 (FSM Equivalence). *Two FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ are* equivalent *if starting from their respective initial states, their input-output behaviors are indistinguishable.*

Given two FSMs, the problem of sequential equivalence checking asks if these two machines are indistinguishable from their output sequences under any input sequence. Notice that, even if any state of one FSM has a corresponding equivalent state in the other, the two FSMs may be inequivalent unless they start from equivalent initial states.

Based on their underlying data structures, verification approaches can be divided into two classes: that of explicit enumerative algorithms and that of implicit symbolic algorithms. The former perform manipulations directly over state transition graphs (e.g., reachability analysis based on traversal on state transition graphs); the latter, on the other hand, perform manipulations over characteristic functions representing state sets, where characteristic functions are realized be some abstract data types, such as ROBDD, CNF, AIG, etc. (e.g., reachability analysis based on Boolean manipulations).

Before effective data structures and algorithms were available for Boolean manipulations in the late 1980s, early verification algorithms performed explicit enumeration over state transition graphs. Since the late 1980s, verification algorithms have relied on implicit manipulations over Boolean formulae and characteristic functions. Since STGs (in a graph representation) and FSMs (in a six-tuple representation) point to the same state transition system, verification can be done both explicitly over STGs or implicitly over FSMs.

Before delving into the algorithms, we introduce some FSM constructs (and thus their STG counterparts) that will be useful later on.

Definition 10 (Quotient FSM). *Given an FSM $\mathcal{M} = (Q, Q^0, \Sigma, \Omega, \boldsymbol{\delta}, \boldsymbol{\lambda})$, its* quotient FSM $\mathcal{M}_{/\sim} = (Q_{/\sim}, Q^0_{/\sim}, \Sigma, \Omega, \boldsymbol{\delta}_{/\sim}, \boldsymbol{\lambda}_{/\sim})$ *(with respect to relation $\sim$) is obtained by collapsing equivalent states $C = \{q_i \in Q \mid q_i \sim q$ for some $q \in Q\}$ into its arbitrary representative state, denoted as $q_C \in Q_{/\sim}$, of the equivalence class. In addition, for any $\sigma \in \Sigma$,*

$$
\begin{aligned}
q_C \in Q^0_{/\sim} &\Leftrightarrow C \cap Q^0 \neq \emptyset, \\
q_{C_2} = \boldsymbol{\delta}_{/\sim}(\sigma, q_{C_1}) &\Leftrightarrow q_2 = \boldsymbol{\delta}(\sigma, q_1) \text{ for some } q_1 \in C_1, q_2 \in C_2, \text{ and} \\
\omega = \boldsymbol{\lambda}_{/\sim}(\sigma, q_C) &\Leftrightarrow \omega = \boldsymbol{\lambda}(\sigma, q) \text{ for some } q \in C.
\end{aligned}
$$

(Recall the assumption that all initial states in an FSM have to be equivalent. So under this assumption there is a single initial state in any quotient FSM.) Also, we call the STG $G_{/\sim}$ of $\mathcal{M}_{/\sim}$ as the *quotient STG* of $\mathcal{M}$.

We can think of $\mathcal{M}_{/\sim}$ as a *reachability-preserving abstraction* of $\mathcal{M}$ with respect to the partition $\sim$ imposed by the observational equivalence of output functions. It is an *abstraction* in the sense that some detailed state information is hidden and the FSM is simplified. The abstraction is *reachability-preserving* in that, for any $\sigma \in \Sigma$, if $\exists\, q_1 \in C_1.\ \boldsymbol{\delta}(\sigma, q_1) \in C_2$, then $\forall\, q_1 \in C_1.\ \boldsymbol{\delta}(\sigma, q_1) \in C_2$. The significance of this kind of abstraction is that, for a given property to be verified which can be formulated as a reachability problem, there exists a reachable state violating the property in the original transition system if and only if there exists a reachable bad state in the abstracted counterpart. Hence the abstraction is both sound and complete for safety property checking. For instance, in $\mathcal{M}_{/\sim}$, $\boldsymbol{\lambda}$ induces an initial partition $\sim^0$ (which distinguishes states with different output observations, i.e., our concerned property) over the state space Q. We may study if some state in one block of $\sim^0$ is reachable from a state in another block. The answer is the same, no matter whether the analysis is conducted over $\mathcal{M}$ or $\mathcal{M}_{/\sim}$. This kind of abstraction is helpful in simplifying sequential equivalence checking and other general safety property checking.

5.1 Explicit Graph Algorithms

In discussing explicit graph algorithms, we are concerned with STGs (instead of FSMs in the six-tuple representation). We introduce three STG-based approaches for sequential equivalence checking. One relies on analyzing reachability on a product STG. Another relies on checking the isomorphism between two quotient STGs. The third one, similar to the previous one though, relies on building the quotient of a disjoint union STG.

Reachability Analysis in the Product of Two STGs In reachability analysis, one wants to assert that bad states of an STG are unreachable from its initial states. In the context of equivalence checking of two STGs G_1 and G_2, we are concerned with their product STG $G_\times$. A state of $G_\times$ is bad if any of its outgoing edges contains a 0, instead of a 1, in the output label.

Example 5. In Figure 3, states $\{(s_0, t_1), (s_0, t_2), (s_1, t_0), (s_1, t_3)\}$ are bad.

Proposition 1. *Two STGs are equivalent if and only if any output label on the edges in the reachable subspace of their product STG is 1.*

Reachability analysis on an STG with n vertices and m edges can be done, e.g. by a breadth-first traversal, in linear time complexity $O(m + n)$. Assume the input alphabet is of size k, i.e. $|\Sigma| = k$. Then $m = kn$ and the above complexity can be rewritten as $O(kn)$. Therefore, to traverse the product space of two STGs $G_1 = (V_1, E_1)$ with $|V_1| = n_1$ and $G_2 = (V_2, E_2)$ with $|V_2| = n_2$, the time complexity is of $O(kn_1n_2)$.

Isomorphism of the Quotients of Two STGs In addition to the previous approach based on reachability, the equivalence of two STGs can be alternatively formulated based on the partition induced by state equivalence. Essentially,

Proposition 2. *The quotients of equivalent STGs are canonical (i.e. unique up to an isomorphism).*

Proof. For two equivalent STGs G_1 and G_2, there exists a bijection between the equivalence classes of states and their initial states must be in corresponding equivalence classes. Otherwise, there exists an input sequence that can drive G_1 and G_2 into inequivalent states. Since states in an equivalent class are merged in computing a quotient, there exists a bijection between equivalent states of $G_{1/\sim}$ and $G_{2/\sim}$. That is, $G_{1/\sim}$ and $G_{2/\sim}$ are isomorphic. The proposition follows. ■

Therefore to check the equivalence of two FSMs, one can first canonicalize the corresponding STGs into their quotient (state-minimized) forms, and then check their graph isomorphism. Two FSMs are equivalent if and only if their quotient STGs are isomorphic (including the matching of initial states) in the reachable state subspace.

The computation of equivalent states is, in fact, already implicit in Definition 6. Figure 8 sketches a procedure computing equivalent states. The initial partition is induced by $\sim^0$. The partition is then iteratively refined. A block B_j in the current partition $\pi^{(i)}$ is split into several sub-blocks such that two states $q_1, q_2 \in B_j$ are in different sub-blocks if and only if $\exists \sigma \in \Sigma$ such that the successor of q_1 and that of q_2 under σ go to different blocks in $\pi^{(i)}$. The procedure terminates when no more refinements can be made. Any block in the final partition represents an equivalence class of states. Given a state equivalence partition, the corresponding quotient STG can then be constructed by merging equivalent states.

```
ComputeEquivalentStates
  input: an STG $G$
  output: the coarsest partition $\pi$ of equivalent states of $G$
  begin
  01  $i := 0$
  02  let $\pi^{(i)}$ be the partition of states induced by $\sim^0$
  03  repeat
  04    $i := i + 1$
  04    $\pi^{(i)} := \emptyset$
  05    for each block $B_j \in \pi^{(i-1)}$
  06      refine $B_j$ into a partition $\pi_{B_j} = \{C_k\}$ with
            $q_1, q_2 \in C_k \Leftrightarrow \forall \sigma \in \Sigma, \exists B_l \in \pi^{(i-1)}.Succ_\sigma(q_1), Succ_\sigma(q_2) \in B_l$
  07      $\pi^{(i)} := \pi^{(i)} \cup \pi_{B_j}$
  08  until $|\pi^{(i)}| = |\pi^{(i-1)}|$
  10  return $\pi^{(i)}$
  end
```

Fig. 8. An algorithm that computes the coarsest partition of the state equivalence of a given STG.

For an STG with n vertices and m edges, an analysis shows that the above algorithm for computing equivalent states may take up to n iterations. Because in each iteration, we may need to traverse m edges, the total time complexity is of $O(mn)$. An improved algorithm that constructs the coarsest stable partition in $O(m \log n)$ time can be found in [Hop71]. (The result was extended in [PT87] to relational coarsest partition and may deal with nondeterministic transitions beyond STGs.) Moreover, constructing the quotient STG for a given partition can be done in $O(m + n)$ time. Consequently, the total time complexity in constructing a quotient STG can be achieved in $O(m \log n)$ time. On the other hand, the isomorphism checking of two quotient STGs is of linear time complexity in the graph size. (The isomorphism checking here is much easier than general graph isomorphism checking because the quotient STGs are deterministic in their state transitions and the correspondence of their two initial states is known.) Consequently, equivalence checking of two STGs, $G_1 = (V_1, E_1)$ with $|V_1| = n_1, |E_1| = m_1$ and $G_2 = (V_2, E_2)$ with $|V_2| = n_2, |E_2| = m_2$, is of time complexity $O(m_1 \log n_1 + m_2 \log n_2)$. In terms of the alphabet size $|\Sigma| = k$, the complexity can be reexpressed as $O(k(n_1 \log n_1 + n_2 \log n_2))$.

State Equivalence in the Disjoint Union of Two STGs In the previous approach, the quotients of two STGs are derived individually and then compared. Here we show that the quotient computation can be computed once, and further that isomorphism checking is unnecessary.

Again, the same procedure of Figure 8 is used. However this time, rather than minimizing STGs G_1 and G_2 individually twice, we apply the procedure once on the disjoint union $G_\uplus$ of the two constituent STGs. In the end, the algorithm yields a partition that contains the state equivalence information not only within the individual STGs but also the equivalence information between them. Essentially,

Proposition 3. *Two STGs G_1 and G_2 are equivalent if and only if their initial states are in the same equivalence class of their disjoint union STG.*

Even though the computation is simplified, the time complexity is essentially the same, i.e., $O(k(n_1 \log n_1 + n_2 \log n_2))$, as the previous algorithm based on isomorphism checking of two quotient STGs.

In the above three enumerative algorithms, STGs are the underlying data structure representing state transition systems. It should be noted that the size of an STG (in terms of the number of vertices and edges) can be exponentially larger than an FSM in the six-tuple representation (in terms of the number of bits or Boolean circuit representing the six tuple). Therefore the early enumerative algorithms may not be effective due to the state explosion problem, namely that the number of states is exponentially larger than the number of state variables. In what follows, we introduce symbolic algorithms, which manipulate Boolean formulae instead of enumerating through transition graphs.

5.2 Implicit Symbolic Algorithms

The previous graph enumerative algorithms are not scalable to large instances. The reasons are twofold: firstly, representing STGs is memory-space consuming since every state and transition needs to be represented explicitly; secondly, enumeration over STGs is time consuming since no parallelism can be exploited, that is, transitions need to be enumerated separately. These shortcomings are overcome in the symbolic algorithms to be introduced.

In symbolic algorithms, state transition systems are represented with FSMs (in the six-tuple representation) or sequential circuits. Enumeration is done through Boolean manipulations. Therefore, sets of states and transitions can be manipulated simultaneously.

To design a symbolic algorithm, its underlying data structure must be effective in representing Boolean formulae and in supporting Boolean manipulations. ROBDD (or BDD for short) is one such data structure. In the sequel, unless otherwise stated, BDD is meant to be ROBDD. There are several unique properties of BDDs that make them particularly suitable for verification:

1. Most Boolean functions can be compactly represented by BDDs.
2. Boolean manipulations over BDDs are efficient (polynomial in BDD sizes).
3. BDD representation is canonical. Thus, checking equivalence of two BDDs (with the same variable ordering) takes constant time.

Despite these nice properties, BDD sizes are in general unpredictable and sensitive to different variable orderings. Due to this intrinsic feature, BDD-based algorithms suffer from non-robustness even though good heuristics exist for finding good variable orderings. Alltogether the BDD is still one of the most important data structures for model checking algorithms, and, in the following discussion, we shall assume them to be the data structure of choice.

```
Algorithm: ForwardStateTraversal
  input: initial states $I$ and a transition relation $T$
  output: reachable states $R$
  begin
  01  $i := 0$
  02  $R^{(0)} := I$
  03  repeat
  04  $i := i + 1$
  05  $R^{(i)} := R^{(i-1)} \cup Img(R^{(i-1)}, T)$
  06  until $R^{(i)} = R^{(i-1)}$
  07  return $R^{(i)}$
  end
```

Fig. 9. An algorithm that performs forward reachability analysis for given initial states and a transition relation.

```
Algorithm: BackwardStateTraversal
  input: final states $F$ and a transition relation $T$
  output: reachable states $R$
  begin
  01  $i := 0$
  02  $R^{(0)} := F$
  03  repeat
  04  $i := i + 1$
  05  $R^{(i)} := R^{(i-1)} \cup PreImg(R^{(i-1)}, T)$
  06  until $R^{(i)} = R^{(i-1)}$
  07  return $R^{(i)}$
  end
```

Fig. 10. An algorithm that performs backward reachability analysis for given target states and a transition relation.

Reachability Analysis of Product Machine As in the case of explicit graph algorithms, we may formulate the equivalence checking problem of two FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ as a reachability problem over the product FSM $\mathcal{M}_\times$. Similarly to Proposition 1, we have

Proposition 4. *Two FSMs $\mathcal{M}_1 = (Q_1, Q_1^0, \Sigma, \Omega, T_1, \boldsymbol{\lambda}_1)$ and $\mathcal{M}_2 = (Q_2, Q_2^0, \Sigma, \Omega, T_2, \boldsymbol{\lambda}_2)$ are equivalent if and only if no bad state $(q_1, q_2) \in Q_1 \times Q_2$ with $\boldsymbol{\lambda}_1(\sigma, q_1) \neq \boldsymbol{\lambda}_2(\sigma, q_2)$ for some $\sigma \in \Sigma$ is reachable from the initial states $Q_1^0 \times Q_2^0$ of the product FSM.*

Since we are concerned with emptiness of the set intersection of the reachable states and the bad states of the product FSM, reachability analysis over the product machine forms the computation core of sequential equivalence checking. Essentially, reachability analysis can be conducted in two directions, forward [CBM89] and backward [Fil91]. Figures 9 and 10 sketch the procedures for forward and backward reachability analysis, respectively.

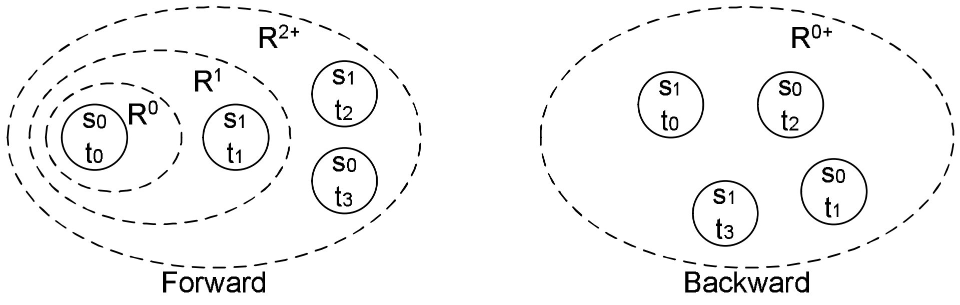


Fig. 11. The reached state sets of the product FSM of Figure 3 with forward and backward state traversals.

Example 6. Figure 11 shows the effective reached state sets of the product FSM of Figure 3 with forward and backward state traversals. The two constituent FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent because in forward traversal the reachable state sets are disjoint with the bad states $\{(s_0, t_1), (s_0, t_2), (s_1, t_0), (s_1, t_3)\}$ (whose transitions may produce '0' at the output), and similarly in backward traversal the states reachable from bad states are disjoint with the initial states. In this example, the forward traversal requires more iterations than the backward traversal. In the symbolic algorithms, the reached state sets are represented with characteristic functions.

The most sophisticated computation involved in a reachability analysis is the existential quantification in the image computation (in Step 5 of the algorithms in Figures 9 and 10). Extensive research efforts have been made in the 1990s to extend the capability of image computation. One of the most important and successful techniques is the so-called *quantification scheduling* [TSL$^+$90], which determines the order of quantification over a set of variables. The objective is to make quantifications as local as possible such that the intermediate Boolean formulae are kept small.

Unlike that of an explicit algorithm over STGs, the complexity measure of an implicit algorithm is harder to quantify because it heavily depends on BDD sizes, which are not well predictable. On the other hand, since the procedures of Figures 9 and 10 in effect perform breadth-first search, the number of iterations is upper bounded by the forward and backward diameters, respectively. (The forward diameter is the length of the longest shortest path starting from an initial state to any state; the backward diameter is the length of the longest shortest path starting from any state to a target state.) Notice that the forward and backward diameter may differ substantially. Hence it is sometimes beneficial to combine both forward and backward analysis in the same procedure.

State Partitioning of Multiplexed Machine An alternative to symbolic equivalence checking is through the state equivalence formulation [JB03].

A *decomposition chart* of a Boolean function $\phi : [\![\boldsymbol{v}_r]\!] \times [\![\boldsymbol{v}_c]\!] \to \mathbb{B}$ is a two dimensional truth table with rows indexed by $[\![\boldsymbol{v}_r]\!]$ and columns indexed by $[\![\boldsymbol{v}_c]\!]$. We call $\boldsymbol{v}_r$ (respectively $\boldsymbol{v}_c$) the free-set (respectively bound-set) variables. To see an interesting connection between a decomposition chart and a BDD of function ϕ, we play a trick on BDD variable ordering. Let the bound-set variables $\boldsymbol{v}_c$ be ordered above the free-set variables $\boldsymbol{v}_r$. Under this variable ordering criterion, we define the *cutset* of a BDD to be the set of BDD nodes not controlled by variables $\boldsymbol{v}_c$ with an incoming edge from some BDD node controlled by $\boldsymbol{v}_c$. Consequently, every valuation $c \in [\![\boldsymbol{v}_c]\!]$ of the bound-set variables corresponds to a path from the root node of the BDD to a node in the cutset. Moreover, the function of this node in the cutset is $\phi(\boldsymbol{v}_r, c)$, i.e., the function of column c (or called the *column pattern* of c) in the decomposition chart. Due to the BDD property that no two nodes in a BDD possess the same function (since the ordered BDD is reduced), we know that, for each column pattern in a decomposition chart, there is a unique corresponding BDD node in the cutset. Thus the number of distinct column patterns equals the size of the cutset.

Example 7. Consider a Boolean function $f : \mathbb{B}^4 \to \mathbb{B}$ over variables $\boldsymbol{v} = (v_1, v_2, v_3, v_4)$ with

$$f(\boldsymbol{v}) = v_2 v_4.$$

Let $\boldsymbol{v}_c = (v_1, v_2)$ and $\boldsymbol{v}_r = (v_3, v_4)$ be the bound-set and free-set variables, respectively. Figure 12 (a) and (b) show the decomposition chart and the BDD of f. The function of the BDD node controlled by v_4 corresponds to the column pattern of the first and third columns. The function of the zero-terminal node of the BDD corresponds to the column pattern of the second and fourth columns. Valuations $\{(0,0),(1,0)\}$ of (v_1, v_2) (the indices for the first and third columns, respectively) are in one equivalence class; valuations $\{(0,1),(1,1)\}$ of (v_1, v_2) (the indices for the second and fourth columns, respectively) are in the other equivalence class. Observe that valuations $\{(0,0),(1,0)\}$ (respectively $\{(0,1),(1,1)\}$) correspond to the else-branch (respectively then-branch) of the root of the BDD of (b). The decomposition chart and the BDD are equivalent representations.

To see how this is useful, suppose we are asked to compute the partition over $[\![\boldsymbol{v}_c]\!]$ induced by the observational output of ϕ for all $[\![\boldsymbol{v}_r]\!]$. That is, we need to characterize the equivalence classes among elements in $[\![\boldsymbol{v}_c]\!]$ such that two elements $c_1, c_2 \in [\![\boldsymbol{v}_c]\!]$ are in the same equivalence class if and only if $\phi(r, c_1) = \phi(r, c_2)$ for all $r \in [\![\boldsymbol{v}_r]\!]$. Equivalently, in the decomposition chart of ϕ, two columns c_1 and c_2 having the same pattern (i.e. $\phi(r, c_1) = \phi(r, c_2)$ for all row index $r \in [\![\boldsymbol{v}_r]\!]$) are in the same equivalence class. Characterizing such equivalence classes can be done implicitly by constructing a BDD with $\boldsymbol{v}_c$ ordered above $\boldsymbol{v}_r$. It is because the paths leading to a node in the cutset of the BDD correspond to an equivalence class. Hence the cutset and the paths leading to it encode all the information we need. Essentially, the size of the cutset equals the number of equivalence classes in the partition. Note that the BDD structure below the cutset is immaterial, and only the structure above (and including) the cutset is important. Therefore, we may possibly abstract the BDD

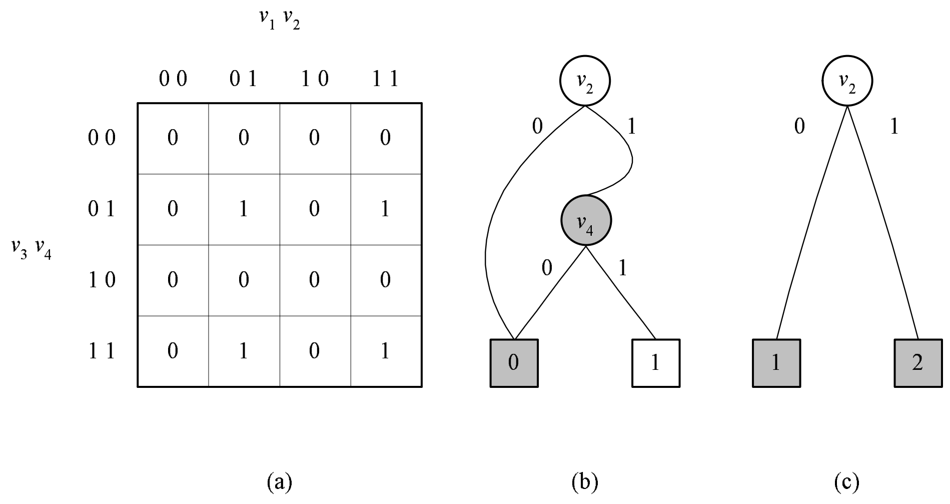


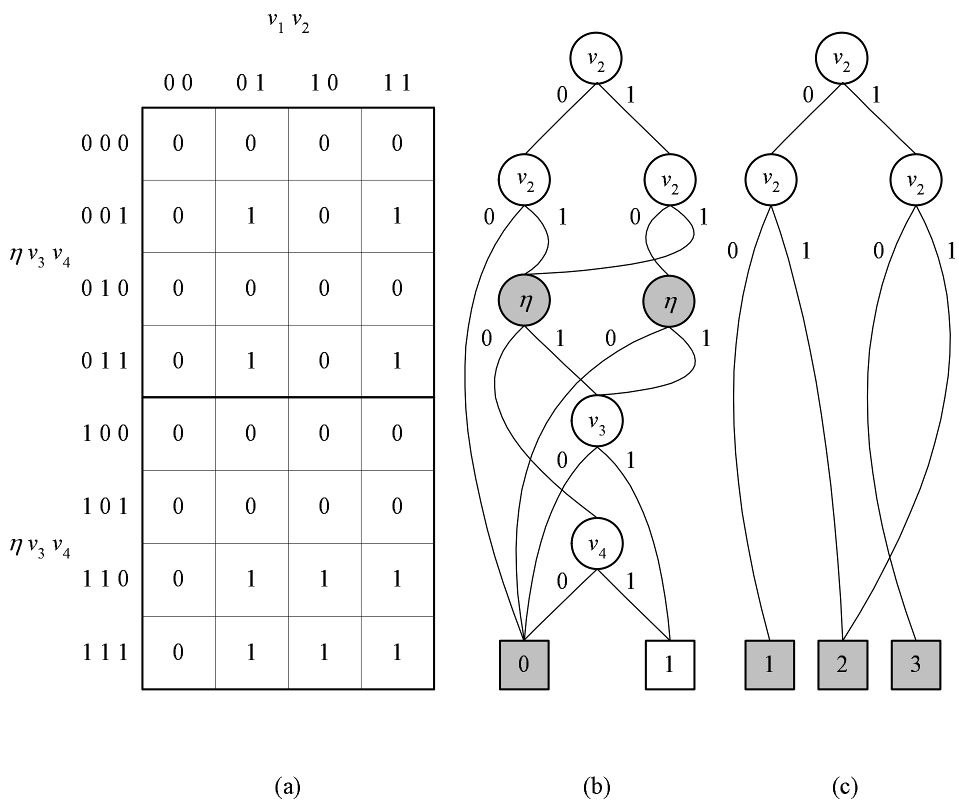
Fig. 12. (a) The decomposition chart of function $f(\boldsymbol{v}) = v_2v_4$ with bound-set and free-set variables $\{v_1, v_2\}$ and $\{v_3, v_4\}$, respectively. (b) The BDD of $f(\boldsymbol{v})$ with the bound-set variables ordered above the free-set variables. The nodes in the cutset are shaded. (c) An MTBDD abstraction.

with a multi-terminal binary decision diagram (MTBDD). In the abstraction, each node in the cutset of the BDD is replaced with a distinct terminal node in the MTBDD while the structure of the BDD above the cutset is preserved in the MTBDD. Thereby, the MTBDD can be seen as a function that maps an element to the index of its corresponding equivalence class. Note that even though the decomposition chart and the BDD are equivalent representations of equivalence classes, the latter can be much more succinct than the former.

Example 8. The BDD of Figure 12 (b) can be abstracted with the MTBDD of (c) when the primary concern is the equivalence classes rather than the characteristics of the equivalence classes. In the abstraction, the zero-terminal node of (b) is replaced with the one-terminal node in (c), and the cutset node controlled by v_4 of (b) is replaced with the two-terminal node in (c). Notice that the BDD structure above the cutset of (b) is preserved in the MTBDD (above the terminal nodes) of (c).

We may need also to compute the partition induced by a set of functions $\{\phi_1(\boldsymbol{v}_r, \boldsymbol{v}_c), \ldots, \phi_n(\boldsymbol{v}_r, \boldsymbol{v}_c)\}$ instead of just a single function. We may think that the partition is induced by the column patterns in the decomposition chart formed by stacking the decomposition charts of $\phi_1, \ldots, \phi_n$ altogether. To apply the implicit computation using BDDs, we may construct a single *hyper-function* $\Phi : [\![\eta]\!] \times [\![\boldsymbol{v}_r]\!] \times [\![\boldsymbol{v}_c]\!] \to \mathbb{B}$ with

$$\Phi(\eta, \boldsymbol{v}_r, \boldsymbol{v}_c) = \bigwedge_{i=1}^{n} ((\eta \equiv i) \wedge \phi_i(\boldsymbol{v}_r, \boldsymbol{v}_c)),$$



$\eta v_3 v_4$ \ $v_1 v_2$	0 0	0 1	1 0	1 1
0 0 0	0	0	0	0
0 0 1	0	1	0	1
0 1 0	0	0	0	0
0 1 1	0	1	0	1
1 0 0	0	0	0	0
1 0 1	0	0	0	0
1 1 0	0	1	1	1
1 1 1	0	1	1	1

Fig. 13. (a) The decomposition chart of function $h(\eta, \boldsymbol{v}) = \overline{\eta} v_2 v_4 \vee \eta v_3(v_1 \vee v_2)$ with bound-set and free-set variables $\{v_1, v_2\}$ and $\{\eta, v_3, v_4\}$, respectively. (b) The BDD of $h(\eta, \boldsymbol{v})$ with the bound-set variables ordered above the free-set variables. The nodes in the cutset are shaded. (c) An MTBDD abstraction.

where η is an n-valued variable (which can be encoded with a vector of Boolean variables in constructing the BDD of Φ). In the BDD of Φ, let variables η and $\boldsymbol{v}_r$ be the free set, and variables $\boldsymbol{v}_c$ be the bound set. Accordingly, the cutset of the BDD characterizes all the equivalence classes induces by $\phi_1, \ldots, \phi_n$.

Example 9. To compute the partition induced by $f(\boldsymbol{v}) = v_2 v_4$ and $g(\boldsymbol{v}) = v_3(v_1 \vee v_2)$ with respect to bound set variables $\{v_1, v_2\}$. We construct the hyper-function h of f and g as $h(\eta, \boldsymbol{v}) = \overline{\eta} f(\boldsymbol{v}) \vee \eta g(\boldsymbol{v}) = \overline{\eta} v_2 v_4 \vee \eta v_3(v_1 \vee v_2)$. Let $\{v_1, v_2\}$ and $\{\eta, v_3, v_4\}$ be the bound-set and free-set variables, respectively. Figure 13 shows the decomposition chart, BDD, and MTBDD of $h(\eta, \boldsymbol{v})$ in (a), (b), and (c), respectively.

To see how the BDD analogue of the decomposition chart is useful in equivalence verification, we first show that it can be exploited to characterize the state equivalence relation of an FSM. We take advantage of the BDD-based characterization of equivalence classes to compute equivalent states of an FSM $\mathcal{M} = ([\![\boldsymbol{s}]\!], Q^0, [\![\boldsymbol{x}]\!], \Omega, \boldsymbol{\delta}, \boldsymbol{\lambda})$. Essentially we need to compute the partition π^* from π^0, where π^i denotes the partition induced by $\sim^i$. To compute π^0 we build the BDD of a hyper-function for the output functions $\boldsymbol{\lambda}(\boldsymbol{x}, \boldsymbol{s})$ by setting $\boldsymbol{s}$ as the bound-set variables and all others as the free-set variables. Accordingly, the cutset C^0 of the BDD characterizes π^0. The BDD can then be simplified and abstracted into an MTBDD, which defines a function $\chi^0 : [\![\boldsymbol{s}]\!] \to \{1, \ldots, |C^0|\}$. For $q \in [\![\boldsymbol{s}]\!]$, $\chi^0(q)$ gives the equivalence class to which state q belongs.

By Definition 6, to compute π^{k+1} from π^k for $k \geq 0$, observe that two states $q_1, q_2 \in [\![\boldsymbol{s}]\!]$ are in the same equivalence class of π^{k+1} if and only if

1. $\chi^k(q_1) = \chi^k(q_2)$, and
2. $\chi^k(Succ_\sigma(q_1)) = \chi^k(Succ_\sigma(q_2))$ for all $\sigma \in [\![\boldsymbol{x}]\!]$.

Assuming χ^k of π^k has been derived, we need to compute the equivalence classes induced by the functions $\chi^k(\boldsymbol{s})$ and $\chi^k(\boldsymbol{\delta}(\boldsymbol{x}, \boldsymbol{s}))$. Hence, we need to build the BDD for the hyper-function of $\chi^k(\boldsymbol{s})$ and $\chi^k(\boldsymbol{\delta}(\boldsymbol{x}, \boldsymbol{s}))$ by setting state variables $\boldsymbol{s}$ as the only bound-set variables. (Note that the multiple-valued function χ^k can be rewritten in terms of a vector of Boolean functions, and thus can be represented with a BDD through a hyper-function construction. Also, $\chi^k(\boldsymbol{\delta}(\boldsymbol{x}, \boldsymbol{s}))$ can be obtained through the BDD composition operation by substituting the variable s_i' of $\chi^k(\boldsymbol{s}')$ with the corresponding transition function $\delta_i(\boldsymbol{x}, \boldsymbol{s})$.) Again, the cutset C^{k+1} of the BDD gives us the equivalence classes of π^{k+1}, from which we may obtain the MTBDD of χ^{k+1}. The iteration terminates when the number of equivalence classes does not increase, i.e., $|C^{k+1}| = |C^k|$. Upon termination, the MTBDD of χ^* characterizes the partition π^*.

For checking the equivalence of two FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ (as in the development of the corresponding explicit algorithm), the previous procedure can be extended to characterize the state equivalence of the multiplexed FSMs $\mathcal{M}_\uplus$ of $\mathcal{M}_1$ and $\mathcal{M}_2$. (Note that the auxiliary variable α is treated as a state variable and thus is considered as a bound-set variable.) Upon termination, we check if the initial states of $\mathcal{M}_1$ and $\mathcal{M}_2$ are in the same equivalence class. By Proposition 3, $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent if and only if their initial states are in the same equivalence class. A detailed exposition can be found in [JB03].

Example 10. Figure 14 shows the state space partitioning of the multiplexed FSM of Figure 4. The two constituent FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent because their initial states s_0 and t_0 are in the same equivalence class in the final partition π^1. In the symbolic algorithm, the equivalence classes of a partition are represented with a BDD.

In the computation of state equivalence, since every equivalence class is represented with a BDD node, the verification capability of the approach may be limited. In practice, the approach may handle designs with up to millions of equivalence classes. Hence, reducing the peak amount of equivalence classes may extend the verification capability and improve the efficiency of the algorithm. An effective way to do so is by divide-and-conquer and handling separately the

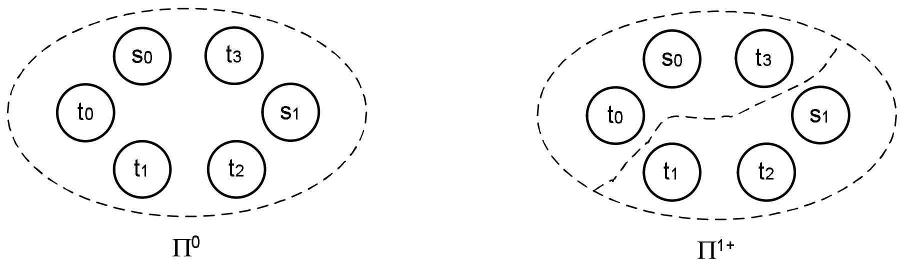


Fig. 14. The state space partitioning of the multiplexed FSM of Figure 4.

partition induced by every output function. In fact, two FSMs are equivalent if and only if their initial states are in one equivalence class for the partition induced by every output function. Actually, the number of equivalence classes induced by all output functions may be exponentially larger than the one induced by any single function. (This reduction can be considered as one instance of the so-called *cone-of-influence reduction* [CGP99].) Another way to reduce the number of equivalence classes is to restrict state partitioning to the subspace of reachable states. When the (exact or over-approximated) reachable state sets of $\mathcal{M}_1$ and $\mathcal{M}_2$ are known, the state equivalence can be characterized within these subspaces by the BDD *constrain* operator [CM90,JB03], which removes from consideration useless equivalence classes. (Note that, unlike verification in the disjoint union space, verification in the product space may not benefit much from reachability information of the individual FSMs. Moreover, reachability analysis on individual FSMs may be much easier than analysis on their product FSM.)

Whereas reachability analysis of a product FSM is usually unpredictable, verification in the disjoint union space may be more robust. After all, two equivalent FSMs must have the same number of equivalence classes in their respective reachable state spaces. On the other hand, one limitation of the above BDD-based characterization of equivalence classes is that it is applicable only to partitions induced by functions rather than by relations; in comparison, verification based on reachability analysis is more flexible and can handle nondeterministic transitions straightforwardly.

Connections between Reachability and Equivalence There is an interesting connection between reachable state pairs and equivalent state pairs.

Assume $\mathcal{M}_1 = (Q_1, Q_1^0, \Sigma_1, \Omega_1, \boldsymbol{\delta}_1, \boldsymbol{\lambda}_1)$ and $\mathcal{M}_2 = (Q_2, Q_2^0, \Sigma_2, \Omega_2, \boldsymbol{\delta}_2, \boldsymbol{\lambda}_2)$ are the FSMs to be verified. Let $\mathcal{M}_\times$ and $\mathcal{M}_\uplus$ be their product and multiplexed FSMs, respectively. In forward reachability analysis of $\mathcal{M}_\times$, any reached state set $R^{(i)} \subseteq Q_1 \times Q_2$ in Figure 9 can be seen as an equivalence relation between Q_1 and Q_2 that must be maintained for the two FSMs to be equivalent. Note that forward reachability analysis of $\mathcal{M}_\times$ characterizes equivalent state pairs only reachable from the initial states of $\mathcal{M}_\times$ (but not all equivalent state pairs in $Q_1 \times Q_2$).

In contrast, in backward reachability analysis of $\mathcal{M}_\times$, the complement of $R^{(i)} \subseteq Q_1 \times Q_2$ in Figure 10 is equal to $\sim^i \subseteq (Q_1 \uplus Q_2) \times (Q_1 \uplus Q_2)$ of $\mathcal{M}_\uplus$ except for ignoring the equivalence among individual FSMs, that is,

$$(Q_1 \times Q_2)\backslash R^{(i)} \ = \sim^i \backslash\{Q_1 \times Q_1 \cup Q_2 \times Q_2\}.$$

Example 11. Consider the reached state set with backward state traversal in Figure 11

$$R^* = (s0, t1), (s0, t2), (s1, t0), (s1, t3),$$

and the state space partition of the related multiplexed FSM in Figure 14

$$\sim^* = (s0, t0), (s0, t3), (s1, t1), (s1, t2), (t0, t3)(t1, t2),$$

it holds that $(Q_1 \times Q_2)\backslash R^\star \ = \sim^\star \backslash\{Q_1 \times Q_1 \cup Q_2 \times Q_2\}$.

Note that backward reachability analysis of $\mathcal{M}_\times$ characterizes all equivalent state pairs in $Q_1 \times Q_2$ whereas state partitioning of $\mathcal{M}_\uplus$ considers all equivalent state pairs in $Q_1 \times Q_2 \cup Q_1 \times Q_1 \cup Q_2 \times Q_2$. Consequently, the number of iterations needed in state partitioning of $\mathcal{M}_\uplus$ may be greater than that in backward reachability analysis of $\mathcal{M}_\times$. Nevertheless, when restricted to the reachable subspace of $\mathcal{M}_1$ and $\mathcal{M}_2$, state partitioning of $\mathcal{M}_\uplus$ terminates at the same iteration as backward reachability analysis of $\mathcal{M}_\times$ since the unreachable state space is not partitioned. In fact, backward reachability analysis can be seen as state space partitioning over the product space, in contrast to state space partitioning over the disjoint union space. (Notice that $R^{(i)}$ in forward reachability analysis does not follow k-step state equivalence so the number of iterations may not be comparable with backward reachability analysis.)

6 Safety Property Checking through Time-frame Expansion

In Section 5, we see that sequential equivalence checking can be formulated as reachability analysis, where we want to assert that the underlying product machine always outputs **true** under any state reachable from the initial state set. It is in fact an instance of safety property checking, where temporal properties of a state transition system can be checked through reachability analysis. In this section, we broaden our discussion a bit and speak of safety property checking. The discussion is in turn applicable to sequential equivalence checking.

The capacity of ROBDD-based verification algorithms is typically limited to designs with at most hundreds of registers. This limitation is due to the memory explosion problem because BDDs may not represent some characteristic functions efficiently or may grow too large due to quantifier elimination (e.g., in image computation), even though dynamic variable reordering [Rud93], quantification scheduling, and some other techniques are helpful.

In addition to BDDs, satisfiability (SAT) solving over Boolean formulae in the conjunctive normal form (CNF) is another core engine for Boolean reasoning. SAT solvers based on the the DPLL algorithm (Davis-Putnam-Logemann-Loveland) [DP60,DLL62] were recently engineered into a very effective technology, due to breakthroughs including conflict-based non-chronological backtracking [MSS99], two watched literals [Zha97], fast Boolean constraint propagation [MMZM01], etc. As a consequence, more and more verification problems are reformulated as SAT solving to exploit the capability of SAT solvers.

In SAT solving, we are asked if a CNF Boolean formula $\phi(\boldsymbol{x})$ is *satisfiable* (true under some valuation, or interpretation, of the Boolean variables $\boldsymbol{x}$, i.e., $\exists \boldsymbol{x}.\phi(\boldsymbol{x})$). Therefore, a SAT solver semantically performs existential quantification. On the other hand, a SAT solver can be exploited to test if a Boolean formula $\phi(\boldsymbol{x})$ is *valid* (true under every valuation of $\boldsymbol{x}$, i.e., $\forall \boldsymbol{x}.\phi(\boldsymbol{x})$) by testing the satisfiability of the complement $\neg\phi(\boldsymbol{x})$. That is, $\phi(\boldsymbol{x})$ is valid if and only if $\neg\phi(\boldsymbol{x})$ is unsatisfiable. However, the main problem of validity checking is whether the complement can be represented efficiently because a SAT solver expects $\neg\phi(\boldsymbol{x})$ to be in CNF. In general, complementing a CNF into another CNF may suffer from an exponential blow-up in the Boolean formula size. Fortunately, in hardware verification, the problem with the complement can sometimes be overcome. For instance, to test if the output of a circuit always produces 1 regardless of the valuations of the input variables, we may add an inverter at the output. By translating the new circuit into a CNF, we may apply SAT solving. The original circuit always produces 1 if and only if the CNF of the new circuit is unsatisfiable. Translating a circuit into a CNF can be done in polynomial time, and the size of the resultant CNF is polynomial in the size of the circuit [PG86]. (The polynomial conversion is possible due to the introduction of new local variables. Existentially quantifying out these variables results in an equivalent Boolean formula depending only on the original variables.) Nonetheless, image computation using SAT is less straightforward than using BDDs. In fact, a real hard task for SAT solving is to solve quantified Boolean formulae alternating existential and universal quantifications: solving quantified Boolean formulae (QBFs) is PSPACE-complete [SM73], whereas SAT [Coo71] is NP-complete. After all, SAT solvers are good at spotting *one* satisfying valuation instead of *all* satisfying valuations.

In contrast to BDD-based algorithms, SAT-based verification algorithms are less memory hungry. However, long run time may be the main bottleneck in SAT solving. In hardware verification, SAT may be preferable to BDDs in that a verification task is less likely to abort due to memory limitations, and it catches more design bugs as time flows, so it is more suitable in hardware debugging than BDDs. However, because SAT is not effective in dealing with quantified Boolean formulae (QBFs), SAT is not as applicable to general model checking as BDDs. To take full advantage of the strengths of SAT solving, the approach to model checking needs to be modified as discussed below.

6.1 Bounded Model Checking

To verify whether a state transition system respects a temporal safety property, it can be translated into a set of QBFs. (In fact, any PSPACE-complete problem can be converted into QBF solving.) For instance, the termination condition of a reachability analysis of Figure 9 or 10 can be expressed with a QBF. Essentially, $R^{(i+1)} \Rightarrow R^{(i)}$ (equivalent to the termination check $R^{(i+1)} = R^{(i)}$) is valid if and only if $R^{(i+1)} \wedge \neg R^{(i)}$, or equivalently $\mathit{Img}(R^{(i)}, T) \wedge \neg R^{(i)}$, is unsatisfiable. Recursively reexpressing $R^{(i)}$ with $R^{(i-1)} \vee \mathit{Img}(R^{(i-1)}, T)$ yields a QBF, where existential and universal quantifications alternate due to the complement of $R^{(i)}$ and the existential quantification in image computation. However, notice that SAT solvers are not good at solving QBFs.

Example 12. Given the initial state set I and the transition relation T, the termination check $R^{(2)} = R^{(1)}$ can be translated to the QBF

$$\forall \boldsymbol{s}^0, \boldsymbol{s}^1, \boldsymbol{s}^2, \exists \boldsymbol{t}^0.[(I(\boldsymbol{s}^0) \wedge T(\boldsymbol{s}^0, \boldsymbol{s}^1) \wedge T(\boldsymbol{s}^1, \boldsymbol{s}^2)) \Rightarrow (I(\boldsymbol{s}^2) \vee (I(\boldsymbol{t}^0) \wedge T(\boldsymbol{t}^0, \boldsymbol{s}^2)))],$$

where $T(\boldsymbol{s}, \boldsymbol{s}')$ is an abbreviation for $\exists \boldsymbol{x}.T(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{s}')$ (or, alternatively, it results from considering input variables $\boldsymbol{x}$ as part of the states variables). The QBF asserts that for all states $\boldsymbol{s}^0, \boldsymbol{s}^1, \boldsymbol{s}^2$, if $\boldsymbol{s}^0$ is an initial state, and there are transitions respectively from $\boldsymbol{s}^0$ to $\boldsymbol{s}^1$ and from $\boldsymbol{s}^1$ to $\boldsymbol{s}^2$, then either $\boldsymbol{s}^2$ is an initial state or $\boldsymbol{s}^2$ can be reached from an initial state $\boldsymbol{t}^0$. In other words, if a state is reached at the second iteration, it must be an initial state or have been reached at the first iteration. That is, no new state is reached at the second iteration.

To take full advantage of SAT solving, we had better restrict the formulation to quantifier-free Boolean formulae. One possible solution is to ignore the termination condition in reachability analysis, and to ask if the considered transition system satisfies the property under any execution of input sequences whose length is upper-bounded by k. By sacrificing "completeness," instead of QBF solving one will be reduced to solving quantifier-free Boolean formulae, as we will see shortly. This bounded-length model checking is known as *bounded model checking* (BMC) [BCCZ99]. Note that, for finite-state systems, BMC is indeed complete, however, under a rather useless upper bound on k, e.g., the number of states of the transition system. Although BMC is in theory complete for finite state systems, in almost all practical cases it runs out of computational resources far before the completeness bound is reached. Therefore, it is considered incomplete in practice.

BMC is very similar to sequential ATPG, developed earlier in the testing community, with some minor difference in their data structures. Let I, T, and P be the initial state set, transition relation, and temporal property, respectively. (In the context of sequential equivalence checking, the temporal property $P(\boldsymbol{s})$ to be verified is $\lambda_\times(\boldsymbol{s}) \equiv 1$.) BMC checks the satisfiability of the following Boolean formulae in order

$$\textsc{Bmc}^0 : I(\boldsymbol{s}^0) \wedge \neg P(\boldsymbol{s}^0),$$
$$\textsc{Bmc}^1 : I(\boldsymbol{s}^0) \wedge T(\boldsymbol{s}^0, \boldsymbol{s}^1) \wedge \neg P(\boldsymbol{s}^1),$$

$$\vdots$$

$$\textsc{Bmc}^k : I(\boldsymbol{s}^0) \wedge T(\boldsymbol{s}^0, \boldsymbol{s}^1) \wedge \cdots \wedge T(\boldsymbol{s}^{k-1}, \boldsymbol{s}^k) \wedge \neg P(\boldsymbol{s}^k),$$

where the state variables are annotated with time indices in superscript. (Here we treat primary input variables as part of the state variables.) The procedure either stops at a satisfiable Boolean formula $\textsc{Bmc}^i$ or proceeds otherwise with a new formula $\textsc{Bmc}^{i+1}$. Boolean formula $\textsc{Bmc}^i$ is satisfiable if the temporal property P is violated and a counterexample of length i is found. Consequently, BMC can be used to locate bugs under some length bound constrained mainly by computational resources.

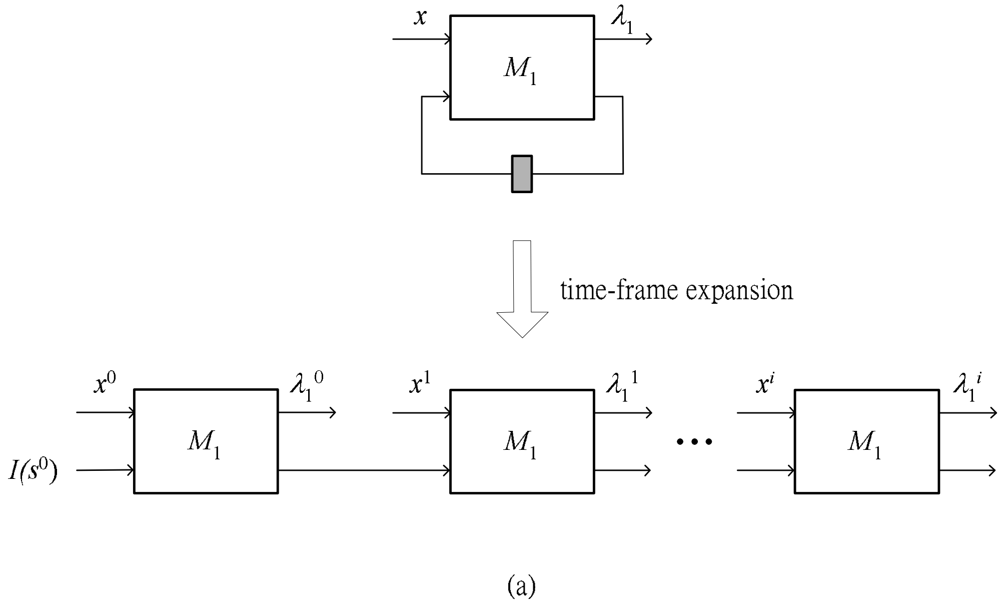
To "visualize" BMC with circuits, the feedback loop of a sequential circuit is eliminated and replaced with a series of concatenated replicas of the combinational core of the circuit. A sequential circuit is expanded into a combinational one with several time frames. Thereby, BMC converts a sequential verification problem into a combinational one (which can be solved using SAT straightforwardly) through the so-called *time-frame expansion*. In effect, the corresponding STG of the original circuit is unwound into a tree through the expansion. Thus states at level i of the tree correspond to states at the i-th time frame reached from the initial states; BMC tries to locate a state at level i that violates the property P under verification.

Example 13. Figure 15 (a) shows the time-frame expansion of the FSM $\mathcal{M}_1$ of Figure 2. When $\mathcal{M}_1$ is converted to its time-frame expansion, the corresponding STG is unwound as shown in Figure 15 (b). Since circuits can be efficiently converted to CNFs as mentioned earlier, BMC using SAT applies.

Figure 16 sketches the procedure of BMC. Let D be the target states that violate property P, i.e., the $\neg P$-states. The procedure checks if there is some target state reachable from the initial states I under a transition path of length i. Because the Boolean formulae of two consecutive iterations $\textsc{Bmc}^i$ and $\textsc{Bmc}^{i+1}$ are almost the same, intuitively many of the learned clauses induced by the conflict analysis in $\textsc{Bmc}^i$ are applicable for solving $\textsc{Bmc}^{i+1}$. Therefore, one important issue for BMC to be effective is how to efficiently reuse some of the learned clauses from the previous iterations and remove those that become invalid in the current iteration. It brings up the notion of *incremental SAT solving*. More details can be found in, e.g., [WKS01,ES03].

6.2 Unbounded Model Checking

The main weakness of BMC is the termination criterion: it is unknown how many time frames are needed to conclude for the absence of bugs. The length of time-frame expansion is mostly set by the limit of computational resources. Shortly after the introduction of BMC, several techniques, e.g. [SSS00,McM02,McM03], were developed to cope with the termination problem in order to guarantee automatic termination of SAT-based model checking whenever a proof is established,



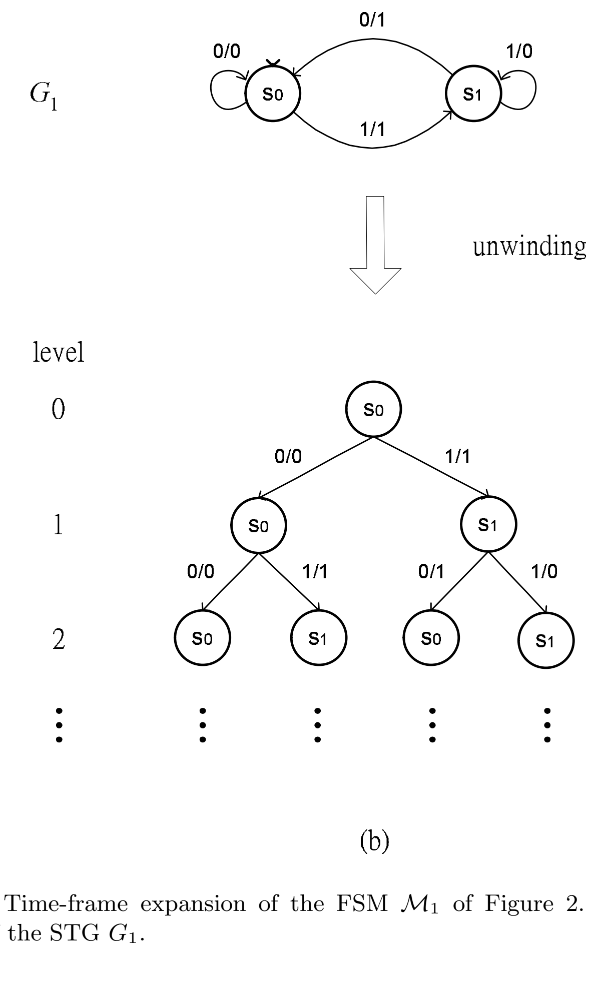


Fig. 15. (a) Time-frame expansion of the FSM $\mathcal{M}_1$ of Figure 2. (b) The effective unwinding of the STG G_1.

```
BoundedModelChecking
  input: transition relation $T$, initial states $I$, destination states $D$
  output: YES, if reachable; NO, otherwise
  begin
  01  $i := 0$
  02  repeat
  03    $\text{BMC}^i := I(\boldsymbol{s}^0) \wedge T(\boldsymbol{s}^0, \boldsymbol{s}^1) \wedge \cdots \wedge T(\boldsymbol{s}^{i-1}, \boldsymbol{s}^i) \wedge D(\boldsymbol{s}^i)$
  04    if $\text{BMC}^i$ is satisfiable
  05      return YES (with a satisfying valuation)
  06    $i := i + 1$
  07  until false
  end
```

Fig. 16. The procedure of bounded model checking.

just like in BDD-based model checking. Removing the boundedness limitation of BMC yields the so-called *unbounded model checking* (UMC).

Of the techniques reported in [SSS00,McM02,McM03], the first approach [SSS00] will be introduced from a different perspective in Section 7.3. The second method [McM02] performs quantifier elimination by the observation that universal quantification in CNF is trivial, and dually is so existential quantification in DNF (disjunctive normal form).

Example 14. The QBF $\forall b.[(a \vee b \vee \neg c)(a \vee \neg b \vee d)]$ equals the quantifier-free formula $(a \vee \neg c)(a \vee d)$ by simply removing the variables to be universally quantified from the original CNF expression.

By converting Boolean formulae back-and-forth between CNF and DNF, quantifications in model checking can be achieved. Not surprisingly, the conversion risks exponential blow-up in the Boolean formulae. Below we focus on the third technique, introduced by McMillan [McM03], a beautiful application of the famous Craig Interpolation Theorem [Cra57].

Theorem 3 (Craig 57). *Let A and B be two Boolean formulae with $A \wedge B$ unsatisfiable. Then there exists a Boolean formula A', called the* interpolant *for A and B, such that*

1. *$A \Rightarrow A'$,*
2. *$A' \wedge B$ is unsatisfiable, and*
3. *A' only refers to the common variables of A and B.*

Note that, while Craig's interpolation theorem holds for first-order logic, in our application we use it in the restricted case of propositional logic.

Krajíček [Kra97] and Pudlák [Pud97] showed that an interpolant can be generated from a *resolution refutation* [Rob65] of an unsatisfiable CNF with complexity linear in the proof. (Interested readers are referred to Pudlák [Pud97] for a detailed exposition; a different but equivalent construct can be found in [McM03].) In fact, modern DPLL-style SAT solvers (such as GRASP [MSS99], Chaff [MMZM01], BerkMin [GN02], etc.) can be exploited to generate resolution

```
UnboundedModelChecking
input: transition relation $T$, initial states $I$, destination states $D$
output: Yes, if reachable; No, otherwise
begin
01  if $I \wedge D$ is satisfiable
02      return Yes (with a satisfying valuation)
03  $i := 0$
04  repeat
05      $R := I(\boldsymbol{s}^{-1})$
06      $A := R(\boldsymbol{s}^{-1}) \wedge T(\boldsymbol{s}^{-1}, \boldsymbol{s}^{0})$
07      $B := T(\boldsymbol{s}^{0}, \boldsymbol{s}^{1}) \wedge \cdots \wedge T(\boldsymbol{s}^{i-1}, \boldsymbol{s}^{i}) \wedge (D(\boldsymbol{s}^{0}) \vee \cdots \vee D(\boldsymbol{s}^{i}))$
08      if $A \wedge B$ is satisfiable
09          return Yes (with a satisfying valuation)
10      repeat
11          generate interpolant $A'(\boldsymbol{s}^{0})$ for $A$ and $B$
12          if $A'(\boldsymbol{s}^{0}) \Rightarrow R(\boldsymbol{s}^{0})$
13              return No
14          $R := R(\boldsymbol{s}^{-1}) \vee A'(\boldsymbol{s}^{-1})$
15          $A := R(\boldsymbol{s}^{-1}) \wedge T(\boldsymbol{s}^{-1}, \boldsymbol{s}^{0})$
16      until $A \wedge B$ is satisfiable
17      $i := i + 1$
18  until false
end
```

Fig. 17. An algorithm that performs unbounded model checking based on interpolation.

refutations for unsatisfiable CNFs, and thus interpolants (with respect to prespecified clause sets A and B). In what follows, we suppose that SAT solvers capable of generating interpolants are available.

Figure 17 shows the algorithm of interpolation-based unbounded model checking, where all Boolean formulae are assumed to be in CNF. Given the transition relation T, it checks if the destination states D are reachable from the initial states I. Variable vectors in parentheses are instantiated with time indices in superscript like in BMC. Observe that the outer loop of the algorithm is the same essentially as the BMC steps of Figure 16, whereas the inner loop performs over-approximated reachability analysis.

To understand the over-approximated image computation in the inner loop, we sketch the main idea. Recall that reachability analysis looks for the fixed point $R^* = \vee_i R^{(i)}$, where $R^{(i)} = R^{(i-1)} \vee \mathit{Img}(R^{(i-1)}, T)$ and $R^{(0)} = I$, the initial state set. A destination state set D is reachable from I if and only if $R^* \wedge D \neq$ **false**. Replacing the exact image computation *Img* with an over-approximation image operator Img' (i.e., $\mathit{Img}(C, T) \Rightarrow \mathit{Img}'(C, T)$ for any state set C and transition relation T) results in an over-approximated reachability computation. To preserve the accuracy of such approximated reachability analysis, we define the *adequacy* of an image over-approximation as follows.

Definition 11 (Adequacy). *Given a state transition system with transition relation T, an over-approximation R' of the image of a state set C, i.e. $R' \supseteq$ $Img(C,T)$, is* adequate *with respect to the destination states D, when, if C cannot reach D, R' cannot reach D.*

So if an over-approximation image operator Img' is adequate with respect to D, then D is reachable from I if and only if $R'^{*} \wedge D \neq \mathbf{false}$, where $R'^{*} = \vee_i R'^{(i)}$ with $R'^{(i)} = R'^{(i-1)} \vee Img'(R'^{(i-1)}, T)$ and $R'^{(0)} = I$. The question is how to find an adequate Img' operator. As a step to our goal, we define first an image operator that is adequate only for k steps.

Definition 12 (k-adequacy). *Given a state transition system with transition relation T, an over-approximation R' of the image of a state set C, i.e. $R' \supseteq$ $Img(C,T)$, is* k-adequate *with respect to the destination states D when, if C cannot reach D, R' cannot reach D in k or fewer steps.*

We will argue that in the k-th iteration of the outer loop of the UMC algorithm in Figure 17, Step 11 is a k-adequate image operation. (Notice that, unlike exact image computation, no existential quantification is needed in the computation.) In addition, if k is such that all states reaching D are within distance k, k-adequacy becomes general adequacy, because the over-approximated reachable states computed from I under such a k-adequate image operator are disjoint from the states reaching D. The conclusion is that even under such an approximation, reachability analysis with respect to I and D returns a correct answer, within a finite number of steps.

Figure 18 illustrates the computation of a few steps of the inner loop of the interpolation-based UMC at the k-th iteration of the outer loop, where $R'(\boldsymbol{s}^{-1}) = I(\boldsymbol{s}^{-1}) \vee A'(\boldsymbol{s}^{-1})$, $R''(\boldsymbol{s}^{-1}) = R'(\boldsymbol{s}^{-1}) \vee A''(\boldsymbol{s}^{-1})$, and $T^k(\boldsymbol{s}^0, \boldsymbol{s}^1, \ldots, \boldsymbol{s}^k)$ is a shorthand for $T(\boldsymbol{s}^0, \boldsymbol{s}^1) \wedge T(\boldsymbol{s}^1, \boldsymbol{s}^2) \wedge \cdots \wedge T(\boldsymbol{s}^{k-1}, \boldsymbol{s}^k)$. By Theorem 3, for unsatisfiable $A \wedge B$ with $A = R(\boldsymbol{s}^{-1}) \wedge T(\boldsymbol{s}^{-1}, \boldsymbol{s}^0)$ and $B = T(\boldsymbol{s}^0, \boldsymbol{s}^1) \wedge T(\boldsymbol{s}^1, \boldsymbol{s}^2) \wedge \cdots \wedge T(\boldsymbol{s}^{k-1}, \boldsymbol{s}^k) \wedge (D(\boldsymbol{s}^0) \vee \cdots \vee D(\boldsymbol{s}^k))$, we know that A' depends only on variables $\boldsymbol{s}^0$ (which are common to A and B), that $A \Rightarrow A'$ implies A' is an over-approximation of $Img(R,T)$, and that the unsatisfiability of $A' \wedge B$ implies states A' cannot reach D within k steps. Consequently, A' is a k-adequate over-approximation of the image of R. (Note that Boolean formula B changes only outside the inner loop so k-adequacy is maintained for the image computations at the k-th iteration of the outer loop.) As an example, R' (respectively R'') of Figure 18 is essentially a k-adequate over-approximation of $R^{(1)}$ (respectively $R^{(2)}$) in the exact forward reachability analysis of Figure 9. The inner loop of Figure 17 iterates until the over-approximated reached states R can reach the destination states D in $k+1$ transitions, that is, $A \wedge B$ is satisfiable at Step 16. Since R is an over-approximated reached state set, the satisfying valuation of Step 16 may be a spurious counter-example. Thus, another iteration of the outer loop is needed to strengthen the k-adequacy of image computation by increasing i. It is worth mentioning that, if in the i-th iteration of the outer loop of Figure 17 the inner loop is executed j times, then for sure Step 9 will not be executed until the $(i+j)$-th iteration of the outer loop because it implies I cannot

$$\begin{tikzcd}
\underbrace{I(\boldsymbol{s}^{-1}) \wedge T(\boldsymbol{s}^{-1}, \boldsymbol{s}^{0})}_{A} & \wedge \arrow[d] & \underbrace{T^{k}(\boldsymbol{s}^{0}, \boldsymbol{s}^{1}, \ldots, \boldsymbol{s}^{k}) \wedge (D(\boldsymbol{s}^{0}) \vee \cdots \vee D(\boldsymbol{s}^{k}))}_{B} \\
 & A'(\boldsymbol{s}^{0}) \arrow[dl] & \\
\underbrace{R'(\boldsymbol{s}^{-1}) \wedge T(\boldsymbol{s}^{-1}, \boldsymbol{s}^{0})}_{A} & \wedge \arrow[d] & \underbrace{T^{k}(\boldsymbol{s}^{0}, \boldsymbol{s}^{1}, \ldots, \boldsymbol{s}^{k}) \wedge (D(\boldsymbol{s}^{0}) \vee \cdots \vee D(\boldsymbol{s}^{k}))}_{B} \\
 & A''(\boldsymbol{s}^{0}) \arrow[dl] & \\
\underbrace{R''(\boldsymbol{s}^{-1}) \wedge T(\boldsymbol{s}^{-1}, \boldsymbol{s}^{0})}_{A} & \wedge & \underbrace{T^{k}(\boldsymbol{s}^{0}, \boldsymbol{s}^{1}, \ldots, \boldsymbol{s}^{k}) \wedge (D(\boldsymbol{s}^{0}) \vee \cdots \vee D(\boldsymbol{s}^{k}))}_{B} \\
 & \vdots &
\end{tikzcd}$$

Fig. 18. The image over-approximations $A', A'', \ldots$ computed by the inner loop (Steps 10–16) at the k-th iteration of the outer loop of the UMC procedure of Figure 17.

reach D in $(i + j)$ transitions. Therefore, Step 17 can be placed alternatively in the inner loop between Steps 15 and 16 of the procedure.

We comment on Steps 7, 12, and 14 of Figure 17. For Step 7, translating the Boolean formula into CNF may be difficult due to the disjunctions. Nevertheless, the aforementioned polynomial-size translation from circuit to CNF [PG86] can be applied, provided that the circuit representing the characteristic function of D is available. Note that the disjunctions of $D(\boldsymbol{s}^{i})$ with $D(\boldsymbol{s}^{0}), \ldots, D(\boldsymbol{s}^{i-1})$ are necessary for k-adequacy to hold for the over-approximated image computation. (In BMC, these disjunctions are redundant in Boolean formula $\textsc{Bmc}^{i}$ because in checking $\textsc{Bmc}^{i}$ we assume $\textsc{Bmc}^{j}$ is unsatisfiable for any $j = 0, \ldots, i$.) For Step12, checking the validity of $A' \Rightarrow R$ is identical to checking the unsatisfiability of $A' \wedge \neg R$. Since the interpolant A' is in circuit representation when generated from a refutation [Pud97,McM03] and the circuit for the characteristic function of R can be built, the circuit representing $A' \wedge \neg R$ can be built as well and translated into CNF in polynomial time for a SAT solver to solve. For Step 14, similar to Step 7, the disjunction of A' and R can be translated into CNF by the circuit-to-CNF conversion from the circuits representing the characteristic functions of A' and R.

The UMC procedure of Figure 17 terminates at Step 2, 9 or 13. At Step 2, it returns YES because the initial states I and destination states D intersect. At Step 9, it returns YES because at least an initial state reaches the destination set in $i + 1$ transitions. At Step 13, it returns No because there is no newly reached state in the over-approximated image A' and thus the reached state set R is closed, and moreover it is disjoint from D.

The number i of iterations of the outer loop of Figure 17 is bounded from above by the backward diameter from destination states D (i.e. the length of the longest shortest path that leads to a state in D). The reason is that, when i is larger than or equal to this diameter, Boolean formula B is satisfiable under any state reaching D, that is, B contains all the states reaching D. As long as I cannot reach D, the computed interpolant in every inner-loop iteration must be disjoint from B and thus is adequate. As the over-approximated reached state set R grows monotonically, the computation must reach a fixed point and the algorithm terminates eventually at Step 13. The upper bound defined by the backward diameter is tight and is the same as the number of iterations needed in BDD-based backward reachability analysis. However, unlike BDD-based exact reachability analysis, the interpolation-based UMC may terminate in fewer iterations than this bound due to the over-approximated image computation.

In BMC, the proof of a satisfiable Bmc^i is useful in providing a trace that activates a design error. On the contrary, the proof of an unsatisfiable Bmc^i is rather useless and is a waste in some sense. In comparison, the interpolation-based UMC extracts useful information out of the refutation through the Craig Interpolation Theorem. On the other hand, when BDD-based and SAT-based model checking approaches are compared, it is observed that SAT-based methods only generate proofs relevant to the properties to be checked, and thus are often more effective than BDD-based methods, which may generate irrelevant proofs as well as relevant ones.

7 Bridging the Gap between Combinational and Sequential Equivalence Checking

The aforementioned verification methods of Sections 5 and 6 make no assumptions about structural similarities between circuits to be compared. They are applicable to equivalence verification of designs with completely different implementations. However, it is this generality that limits their capability and scalability to verify large designs. The generality is often unnecessary because in most cases the designs being compared possess structural similarities to some extent. Although the similarities may not be sufficient to demonstrate the equivalence between two designs, their identification may substantially simplify verification tasks. As an extreme example, when the relation between state encodings of the two designs to be compared is known, the equivalence verification becomes pure combinational checking, and the complexity reduces from a PSPACE-complete problem (sequential equivalence checking) to a coNP-complete problem (combinational equivalence checking). This dramatic complexity reduction motivates the exploitation of circuit similarities in equivalence verification.

To be precise, we define equivalent signals of sequential circuits as a metric for structural similarities, where the term “signal” means a wire in a circuit.

Definition 13 (Signal Equivalence). *Given a circuit implementation of an FSM, two signals in the circuit are* equivalent *(i.e. they are* corresponding signals*) if their values are all identical or all complementary to each other in any execution of the FSM.*

In fact, this notion of correspondence can be extended straightforwardly to signals between two different circuits. For instance, one may construct a product machine of the two circuits and seek equivalent signals inside it.

Given two sequential circuits, especially when one is optimized from the other through logic synthesis tools, their circuit structures, though may look different, may possess a large amount of equivalent signals. The identification of these equivalent signals may not be trivial. The labels (names) of signals may have been changed or lost during logic optimization, for instance, due to the creation of new signals, elimination of existing signals, etc. There is little hope to obtain corresponding signals simply by name matching. Moreover, even signals with the same name are not necessarily equivalent and they need to be proved as well. Identifying corresponding signals without relying on pure name matching has its practical importance. In this section, we introduce some effective techniques that identify circuit similarities for sequential circuits without relying on name matching.

7.1 Inductive Register Correspondence

Among signals whose correspondences are to be identified, signals at register outputs (state variables) are the most important to demonstrate the equivalence between two sequential circuits. We call the equivalence of signals at register outputs as the *register correspondence*. It can be envisaged that, if two sequential circuits differ only combinationally, there exists a register correspondence between these two circuits, that is, every state transition function of one circuit has an equivalent transition function of the other. If the register correspondence is established, then the underlying sequential equivalence checking problem is reduced to a combinational one.

By Definition 13, to compute equivalent signals requires the knowledge of the reachable state set of an FSM. As reachability analysis is hard, we turn to seek approximation approaches to the identification of equivalent signals. In fact, induction can be used as an effective way of detecting register correspondences [Fil92,vEJ95]. Although the characterization is incomplete, it captures an interesting class of register correspondences.

Definition 14 (Inductive Register Correspondence). *An* inductive register correspondence *of an FSM $\mathcal{M} = (\llbracket \boldsymbol{s} \rrbracket, I, \llbracket \boldsymbol{x} \rrbracket, \Omega, \boldsymbol{\delta}, \boldsymbol{\lambda})$ is an equivalence relation $\overset{\mathrm{rc}}{=} \subseteq \{\boldsymbol{s}\} \times \{\boldsymbol{s}\}$ over the state variables which satisfies, for all valuations of $\boldsymbol{x}, \boldsymbol{s}$,*

$$\begin{aligned} \textit{Base case}: \ & I(\boldsymbol{s}) \Rightarrow R_{\overset{\mathrm{rc}}{=}}(\boldsymbol{s}), \textit{and} && (6) \\ \textit{Inductive case}: \ & R_{\overset{\mathrm{rc}}{=}}(\boldsymbol{s}) \Rightarrow R_{\overset{\mathrm{rc}}{=}}(\boldsymbol{\delta}(\boldsymbol{x}, \boldsymbol{s})), && (7) \end{aligned}$$

where

$$R_{\overset{\text{rc}}{\equiv}}(\boldsymbol{s}) = \bigwedge_{(s_i,s_j)\in \overset{\text{rc}}{\equiv}} s_i \equiv s_j.$$

It can be checked that, for state variables s_i and s_j satisfying the above conditions, their valuations are always identical throughout the execution of the underlying FSM.

Notice that the equivalence relations $\overset{\text{rc}}{\equiv}$ satisfying 6 and 7 may not be unique. However, there exists a unique *maximum* inductive register correspondence. In fact, the equivalence relations satisfying Equations (6) and (7) form a bounded partially ordered set (or poset) $P = (Z, \subseteq)$, where Z is the set of equivalence relations satisfying Equations (6) and (7). Since relation $\subseteq$ is a partial order (satisfying reflexivity, antisymmetry, and transitivity) on Z, $P = (Z, \subseteq)$ is a poset. In particular, the empty set $\emptyset$ is a lower bound for Z; it corresponds to an equivalence relation with no equivalent state variables. On the other hand, there is an upper bound for Z. This is because, if $\overset{\text{rc}}{\equiv}_1$ and $\overset{\text{rc}}{\equiv}_2$ are two elements in Z, then $\overset{\text{rc}}{\equiv}_1 \cup \overset{\text{rc}}{\equiv}_2$ forms another valid inductive register correspondence and is in Z as well. It certifies that a (locally) *maximal* inductive register correspondence is also a (globally) *maximum* one.

The signal correspondence of Definition 14 can be slightly relaxed by further considering complementary state variables s_i and s_j satisfying Equations (6) and (7) with

$$R_{\overset{\text{rc}}{\equiv}}(\boldsymbol{s}) = \bigwedge_{(s_i,s_j)\in \overset{\text{rc}}{\equiv}} s_i \equiv \overline{s_j}.$$

Since the valuations of s_i and s_j are always complementary to each other in the execution of the underlying FSM, they can be seen as corresponding signals as well.

The procedure sketched in Figure 19 computes the maximum inductive register correspondence, where the function call *InitialValue*(s) obtains the initial value of state variable s. Here we assume that the initial value of any register is a fixed deterministic value. That is, there is a single initial state in the underlying FSM. (Note that, for a state variable with nondeterministic initial values, it may not have equivalent state variables by Definition 14.) For the sake of simplicity, the computed register correspondence does not include state variables with complementary values although this extension can be added easily. On the other hand, as mentioned earlier, the procedure can be used to compute the register correspondence between two sequential circuits because it can take the product machine of the two circuits as its input.

Example 15. Consider the FSM $\mathcal{M} = (\mathbb{B}^4, (0,0,0,0) \in \mathbb{B}^4, \mathbb{B}, \Omega, \boldsymbol{\delta}, \boldsymbol{\lambda})$, where $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_4)$ with

$$\begin{aligned}
\delta_1(x, \boldsymbol{s}) &= \overline{x}\, s_1 \vee x\, s_3, \\
\delta_2(x, \boldsymbol{s}) &= \overline{x}\, s_2 \vee x\, s_3 \vee (s_1 \oplus s_2), \\
\delta_3(x, \boldsymbol{s}) &= \overline{x}\, s_3 \vee x\, \overline{s_1}\, \overline{s_3}, \text{ and} \\
\delta_4(x, \boldsymbol{s}) &= (\overline{x}\, s_4 \vee x\, \overline{s_2}\, \overline{s_4})(\overline{s_2} \vee \overline{s_3}).
\end{aligned}$$

```
ComputeInductiveRegisterCorrespondence
  input: a sequential circuit $\mathcal{M}$ with transition functions $\boldsymbol{\delta}$ and
         input and state variables $\boldsymbol{x}$ and $\boldsymbol{s}$, respectively
  output: the inductive register correspondence of $\mathcal{M}$
  begin
  01  $i := 0$
  02  $\overset{\mathrm{rc}}{=}^{(i)} := \{(s_p, s_q) \mid \mathit{InitialValue}(s_p) = \mathit{InitialValue}(s_q)\}$
  03  repeat
  04    $i := i + 1$
  05    $R := \bigwedge_{(s_p,s_q) \in \overset{\mathrm{rc}}{=}^{(i-1)}} (s_p \equiv s_q)$
  06    $\overset{\mathrm{rc}}{=}^{(i)} := \{(s_p, s_q) \in \overset{\mathrm{rc}}{=}^{(i-1)} \mid \forall \boldsymbol{x}, \boldsymbol{s}.[R(\boldsymbol{s}) \Rightarrow (\delta_p(\boldsymbol{x}, \boldsymbol{s}) \equiv \delta_q(\boldsymbol{x}, \boldsymbol{s}))]\}$
  07  until $\overset{\mathrm{rc}}{=}^{(i)} = \overset{\mathrm{rc}}{=}^{(i-1)}$
  08  return $\overset{\mathrm{rc}}{=}^{(i)}$
  end
```

Fig. 19. An algorithm that characterizes the inductive register correspondence for a given sequential circuit.

The inductive register correspondence of $\mathcal{M}$ can be identified through the procedure of Figure 19. It can be checked that the register correspondences at different iterations are

$$\begin{aligned}
\overset{\mathrm{rc}}{=}^{(0)} &= \{(s_1, s_2), (s_1, s_3), (s_1, s_4), (s_2, s_3), (s_2, s_4), (s_3, s_4)\}, \\
\overset{\mathrm{rc}}{=}^{(1)} &= \{(s_1, s_2)\}, \text{ and} \\
\overset{\mathrm{rc}}{=}^{(2)} &= \{(s_1, s_2)\}.
\end{aligned}$$

Thus, the procedure terminates in two iterations and identifies s_1 and s_2 as corresponding state variables. Since s_1 and s_2 have the same valuations throughout the execution of $\mathcal{M}$, they are equivalent signals.

Theorem 4. *The procedure of Figure 19 terminates and returns the maximum inductive register correspondence.*

Proof. The termination is easily seen because $\overset{\mathrm{rc}}{=}^{(i)}$ is initially finite and decreases monotonically until $\overset{\mathrm{rc}}{=}^{(i)}$ equals $\overset{\mathrm{rc}}{=}^{(i-1)}$ with cardinality no less than zero.

The computed register correspondence upon termination follows Definition 14 because Steps 1 and 6 of the procedure in Figure 19 are satisfied. On the other hand, we show that the returned register correspondence is maximum. Let S be the set of state variables. Then $\mathcal{L} = (\mathcal{P}(S \times S), \subseteq, S \times S, \emptyset)$ forms a *complete lattice* since relation $\subseteq$ is a partial order on $\mathcal{P}(S \times S)$ (the power set of $S \times S$), and $S \times S$ and $\emptyset$ are the greatest and least elements, respectively, of $\mathcal{P}(S \times S)$. Also, observe that Step 6 in the procedure of Figure 19 defines a mapping $f : (S \times S) \to (S \times S)$, which is monotone, that is, for all $a, b \in \mathcal{P}(S \times S)$, $a \subseteq b$ implies $f(a) \subseteq f(b)$. Since the procedure iteratively removes inequivalent state-variable pairs through f, it computes a greatest fixed point by the Knaster-Tarski Theorem [Tar55]. ■

The procedure *ComputeInductiveRegisterCorrespondence* is effective and scalable to large designs, e.g, with tens of thousands of registers. The main reason of this practicality is that an equivalence relation among state variables, instead of states, are computed. As a result, the state explosion problem does not occur. However, it should be noted that not all register correspondences can be characterized inductively. In fact, the relation $\overset{\text{rc}}{\equiv}$ can be seen as an over-approximation of the reachable state set R of the underlying sequential circuit $\mathcal{M}$, i.e., $\overset{\text{rc}}{\equiv} \supseteq R$, because $\overset{\text{rc}}{\equiv}$ must be an invariant over R. The procedure in Figure 19 can be seen as a kind of reachability analysis as the inductive register correspondence $\overset{\text{rc}}{\equiv}^{(i)}$ at the ith iteration is gradually refined, i.e., $\overset{\text{rc}}{\equiv}^{(i)} \supseteq \overset{\text{rc}}{\equiv}^{(i+1)}$, and approximates R more and more accurately.

Example 16. Continue Example 15. State variables s_3 and s_4 are, in fact, equivalent signals by Definition 13. However, they cannot be detected by the inductive procedure of Figure 19. The reason is that the valuations of s_3 and s_4 are the same in the reachable state space, with characteristic function $R(\boldsymbol{s}) = \overline{s_1}\,\overline{s_2}\,\overline{s_3}\,\overline{s_4} \vee \overline{s_1}\,\overline{s_2}\,s_3\,s_4 \vee s_1\,s_2\,\overline{s_3}\,\overline{s_4}$, of $\mathcal{M}$. However, it is not true in the over-approximated state space, with characteristic function $R_{\overset{\text{rc}}{\equiv}}(\boldsymbol{s}) = s_1\,s_2 \vee \overline{s_1}\,\overline{s_2}$, because δ_3 and δ_4 are not equivalent under $R_{\overset{\text{rc}}{\equiv}}(\boldsymbol{s})$.

The algorithm of inductive register correspondence can be applied for sequential equivalence checking of two FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ by constructing their product machine $\mathcal{M}_\times$ and analyzing register correspondence on $\mathcal{M}_\times$. It checks if under the computed over-approximated reachable state set $R_{\overset{\text{rc}}{\equiv}}$ the output of $\mathcal{M}_\times$ always produces constant one, i.e., for all valuations of $\boldsymbol{x}, \boldsymbol{s}$

$$R_{\overset{\text{rc}}{\equiv}}(\boldsymbol{s}) \Rightarrow \lambda_\times(\boldsymbol{x}, \boldsymbol{s}) \equiv 1. \tag{8}$$

If the above formula is valid, then $\mathcal{M}_1$ and $\mathcal{M}_2$ are indeed equivalent. Otherwise, $\mathcal{M}_1$ and $\mathcal{M}_2$ may or may not be equivalent. It yields the so-called *false negative*, which means a negative answer may be problematic. Therefore, inductive register correspondence is incomplete for sequential equivalence checking. As a matter of fact, its proving power is very limited and only complete for special occasions, e.g., the FSMs under comparison are transformed combinationally.

Quasi-inductive Functional Dependency In inductive register correspondence $\overset{\text{rc}}{\equiv}$, every two state variables s_i and s_j are related with the identity function or the complement of the identity function. That is, $s_i \equiv s_j$ or $s_i \equiv \overline{s_j}$. In fact, more general *functional dependencies* [JB04] can be considered, where a state variable s_i can be expressed as a function θ over other state variables $s_1, \ldots, s_{i-1}, s_{s+1}, \ldots, s_n$, i.e., $s_i = \theta(s_1, \ldots, s_{i-1}, s_{s+1}, \ldots, s_n)$. Similar to inductive register correspondence, functional dependencies may be characterized in an inductive manner. However, the computation may risk non-termination unless some artificial conditions are further imposed. A detailed exposition on combinational and sequential functional dependencies can be found in [JB04]. Based on Craig interpolation and incremental SAT solving, recent work [LJHM07] further extends the scalability of the exploration of combinational functional dependencies, where designs with hundreds of thousands of gates are handled effectively.

7.2 Inductive Signal Correspondence

In register correspondence, we search an equivalence relation over signals at register outputs. In fact, there is no need to restrict ourselves to the correspondence of registers. Given a circuit implementation of an FSM, we may compute the correspondences for all signals (including intermediate signals) of the circuit [vE00]. At first glance it may seem that there is no advantage in computing correspondences among intermediate signals. However, it turns out that identifying such correspondences helps to tighten the induction hypothesis and thus to tighten the approximation of the reachable state set.

Similar to Definition 14, we have the following definition.

Definition 15 (Inductive Signal Correspondence). *An* inductive signal correspondence *of a circuit implementing FSM* $\mathcal{M} = ([\![\boldsymbol{s}]\!], I, [\![\boldsymbol{x}]\!], \Omega, \boldsymbol{\delta}, \boldsymbol{\lambda})$ *is an equivalence relation* $\overset{\text{sc}}{\equiv} \subseteq F \times F$ *over the set* F *of circuit signals which satisfies, for all valuations of* $\boldsymbol{x}, \boldsymbol{s}$*,*

$$\textit{Base case}: \; I(\boldsymbol{s}) \Rightarrow R_{\overset{\text{sc}}{\equiv}}(\boldsymbol{x}, \boldsymbol{s}), \textit{and} \tag{9}$$

$$\textit{Inductive case}: \; R_{\overset{\text{sc}}{\equiv}}(\boldsymbol{x}, \boldsymbol{s}) \Rightarrow R'_{\overset{\text{sc}}{\equiv}}(\boldsymbol{x}, \boldsymbol{s}), \tag{10}$$

where

$$R_{\overset{\text{sc}}{\equiv}}(\boldsymbol{x}, \boldsymbol{s}) = \bigwedge_{(f_i, f_j) \in \overset{\text{sc}}{\equiv}} f_i(\boldsymbol{x}, \boldsymbol{s}) \equiv f_j(\boldsymbol{x}, \boldsymbol{s}), \textit{and}$$

$$R'_{\overset{\text{sc}}{\equiv}}(\boldsymbol{x}, \boldsymbol{s}) = \bigwedge_{(f_i, f_j) \in \overset{\text{sc}}{\equiv}} \forall \boldsymbol{x}'.[f_i(\boldsymbol{x}', \boldsymbol{\delta}(\boldsymbol{x}, \boldsymbol{s})) \equiv f_j(\boldsymbol{x}', \boldsymbol{\delta}(\boldsymbol{x}, \boldsymbol{s}))].$$

In the above definition, the function of a signal $f_i \in F$ is denoted as $f_i(\boldsymbol{x}, \boldsymbol{s})$. Note that the functions of all signals are expressed in terms of primary-input and state variables. With almost the same procedure, we may compute corresponding signals as shown in Figure 20. Comparing $R_{\overset{\text{rc}}{\equiv}}$ and $R_{\overset{\text{sc}}{\equiv}}$, we see that $R_{\overset{\text{sc}}{\equiv}}(\boldsymbol{x}, \boldsymbol{s})$ imposes a strictly stronger precondition for the implication in Equation (10). With the stronger inductive hypothesis, it is possible to identify more register correspondence not detectable in $\overset{\text{rc}}{\equiv}$.

As a modern integrated-circuit design may contain millions of logic gates, considering the correspondences of all signals is formidable. In implementing the computation of inductive signal correspondence, it is desirable to first screen out obvious inequivalent signal pairs. For that purpose, simulation is often adopted and is shown to be fast and effective.

The above algorithms in computing register and signal correspondences assume that the initial values of registers are known and unique. However, when the initial values are not known or not unique *a priori*, the computation needs to be modified to take nondeterministic initialization into account. Different notions

```
ComputeInductiveSignalCorrespondence
  input: a sequential circuit (with signals $F$) implementing $\mathcal{M} = ([\![\boldsymbol{s}]\!], I, [\![\boldsymbol{x}]\!], \Omega, \boldsymbol{\delta}, \boldsymbol{\lambda})$
  output: the inductive register correspondence of $\mathcal{M}$
  begin
  01  $i := 0$
  02  $\overset{\text{sc}}{\equiv}^{(i)} := \{(f_p, f_q) \mid \forall \boldsymbol{x}, \boldsymbol{s}.[(f_p(\boldsymbol{x}, \boldsymbol{s}) \equiv f_q(\boldsymbol{x}, \boldsymbol{s})) \wedge I(\boldsymbol{s})]\}$
  03  repeat
  04      $i := i + 1$
  05      $R := \bigwedge_{(f_p, f_q) \in \overset{\text{sc}}{\equiv}^{(i-1)}} (f_p(\boldsymbol{x}, \boldsymbol{s}) \equiv f_q(\boldsymbol{x}, \boldsymbol{s}))$
  06      $R' := \bigwedge_{(f_p, f_q) \in \overset{\text{sc}}{\equiv}^{(i-1)}} (f_p(\boldsymbol{x}', \boldsymbol{\delta}(\boldsymbol{x}, \boldsymbol{s})) \equiv f_q(\boldsymbol{x}', \boldsymbol{\delta}(\boldsymbol{x}, \boldsymbol{s})))$
  07      $\overset{\text{sc}}{\equiv}^{(i)} := \{(f_p, f_q) \in \overset{\text{sc}}{\equiv}^{(i-1)} \mid \forall \boldsymbol{x}, \boldsymbol{x}', \boldsymbol{s}.[R(\boldsymbol{x}, \boldsymbol{s}) \Rightarrow R'(\boldsymbol{x}, \boldsymbol{x}', \boldsymbol{s})]\}$
  08  until $\overset{\text{sc}}{\equiv}^{(i)} = \overset{\text{sc}}{\equiv}^{(i-1)}$
  09  return $\overset{\text{sc}}{\equiv}^{(i)}$
  end
```

Fig. 20. An algorithm that characterizes the inductive signal correspondence for a given sequential circuit.

of conformance, e.g. [BS98], may be defined depending on the imposed initial conditions. (Initialization issues are to be discussed in Section 8.) Nevertheless, the essential computation based on induction is the same.

Similar to $\overset{\text{rc}}{\equiv}$, signal correspondence $\overset{\text{sc}}{\equiv}$ can be used for checking the equivalence of two sequential circuits $\mathcal{M}_1$ and $\mathcal{M}_2$. In effect, these two FSMs are equivalent if, for all valuations of $\boldsymbol{x}, \boldsymbol{s}$,

$$R_{\overset{\text{sc}}{\equiv}}(\boldsymbol{x}, \boldsymbol{s}) \Rightarrow \lambda_{\times}(\boldsymbol{x}, \boldsymbol{s}) \equiv 1. \tag{11}$$

Similar to the case of register correspondence, the above condition is not sufficient to show the inequivalence of two FSMs. That is, false negatives may occur as well. Nevertheless, the effectiveness in proving of signal correspondence is strictly stronger than that of register correspondence. In particular, it is complete for verifying circuits optimized by retiming [LS83,LS91]. By exploiting signal correspondence among different timeframes, inductive signal correspondence can further be made complete for equivalence checking of circuits optimized with retiming plus resynthesis plus retiming [JH07]. Although verification via signal correspondence is incomplete in general, signal correspondence can be used as a preprocessing step to simplify the circuits to be compared before reachability analysis is applied.

7.3 Inductive Safety Property Checking

Signal correspondence captures the equivalence relation between two signals. However, equivalence relation is not the only criterion to be exploited. For instance, among many other possibilities, the relation of implication, $\Rightarrow$, between two signals can be computed, as was suggested in [BC00]. It yields an even stronger induction hypothesis than equivalence relation, $\Leftrightarrow$, since equivalent signals $f_1 \Leftrightarrow f_2$ can be captured by $(f_1 \Rightarrow f_2) \wedge (f_2 \Rightarrow f_1)$ but $f_1 \Rightarrow f_2$ cannot be written in terms of $\Leftrightarrow$. A further generalization is to consider general temporal properties as the candidates to be proved by induction. In fact, $R_{\underline{\underline{\text{rc}}}}$ and $R_{\underline{\underline{\text{sc}}}}$ are just two special instances of temporal properties of interest. The inductive proof of a general safety property P can be formalized as follows.

$$\text{Base case}: \; I(\boldsymbol{s}) \Rightarrow P(\boldsymbol{s}), \text{ and} \tag{12}$$

$$\text{Inductive case}: \; P(\boldsymbol{s}) \wedge T(\boldsymbol{s}, \boldsymbol{s}') \Rightarrow P(\boldsymbol{s}'), \tag{13}$$

where initial state set I may be of cardinality greater than one, and transition relation T may contain non-deterministic transitions. (Recall that we may treat primary input variables as part of the state variables. Thereby, T can be expressed in terms of state variables only.) By Definition 5, note that the induction aims to certify that P characterizes a state set closed under the transition relation T. (It is easily seen that the reachable state set R of reachability analysis satisfies Equations (12) and (13).)

Observe that proving safety properties by the above induction is incomplete. When Equations (12) and (13) are satisfied, P holds at any state reachable from I under T. In contrast, when Equation (12) is violated, P is not true for some initial state, and a true counterexample is generated. When Equation (13) is violated instead, P may or may not hold for all reachable states, and a spurious counterexample, i.e., a false negative, may be generated. Examining the causes, we see that the induction fails (one of Equations (12) and (13) is violated) if and only if, in the entire state space, there exists a P-state (a state that satisfies P) that can transition to a $\overline{P}$-state (a state that satisfies $\overline{P}$) as shown in Figure 21 (a). In particular, a false negative can be generated if and only if, in the unreachable state space, there exists a transition of Figure 21 (a). Since the entire state space is an over-approximation of the reachable state space, inductive proofs are in general incomplete.

To make induction complete for property checking, k-step induction [SSS00] was introduced to strengthen the induction hypothesis by increasing the transition length k when a false negative occurs. Formally speaking, a k-step induction consists of

$$\text{Base case}: \; I(\boldsymbol{s}^0) \wedge T^k(\boldsymbol{s}^0, \ldots, \boldsymbol{s}^k) \Rightarrow P^k(\boldsymbol{s}^0, \ldots, \boldsymbol{s}^k), \text{ and} \tag{14}$$

$$\text{Inductive case}: \; P^k(\boldsymbol{s}^0, \ldots, \boldsymbol{s}^k) \wedge T^{k+1}(\boldsymbol{s}^0, \ldots, \boldsymbol{s}^{k+1}) \Rightarrow P(\boldsymbol{s}^{k+1}), \tag{15}$$

where a state-variable vector $\boldsymbol{s}$ is annotated with a superscript i indicating that $\boldsymbol{s}$ is instantiated for the ith time-frame, $T^i(\boldsymbol{s}^0, \ldots, \boldsymbol{s}^i)$ is the shorthand for $T(\boldsymbol{s}^0, \boldsymbol{s}^1) \wedge T(\boldsymbol{s}^1, \boldsymbol{s}^2) \wedge \cdots \wedge T(\boldsymbol{s}^{i-1}, \boldsymbol{s}^i)$, and $P^i(\boldsymbol{s}^0, \ldots, \boldsymbol{s}^i)$ is the shorthand for $P(\boldsymbol{s}^0) \wedge \cdots \wedge P(\boldsymbol{s}^i)$. Since for $k = 0$ Equations (14) and (15) reduce to Equations (12) and (13), respectively, the ordinary induction is a special case of the k-step induction.

It is not hard to see that a false negative caused by Figure 21 (a) is ruled out in a 2-step induction. Similarly, a false negative caused by Figure 21 (b) (which

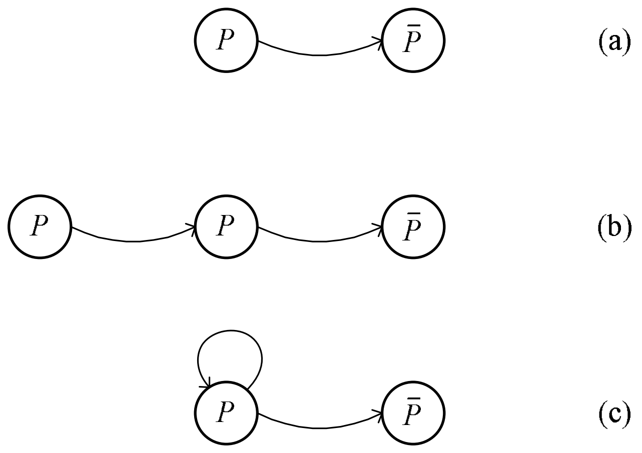


Fig. 21. (a) A transition condition that makes an 1-step induction fail. (b) A transition condition that makes a 2-step induction fail. (c) A transition condition that makes a general k-step induction fail.

may appear in a 2-step induction) is ruled out in a 3-step induction. Increasing the length of transition paths along which P is satisfied makes sure that the condition that fails the k-step induction does not appear in the $(k+i)$-step induction for $i = 1, 2, \ldots$. However, a false negative with an infinite path length is possible, e.g., due to the transition of Figure 21 (c); it cannot be ruled out by increasing k. Thus, the k-step induction risks non-termination.

Fortunately, this problem can be overcome by adding the following uniqueness criterion, in addition to Equations (14) and (15).

$$\bigwedge_{i \le j \le k} \boldsymbol{s}^i \not\equiv \boldsymbol{s}^j \quad \left(= \bigwedge_{i \le j \le k} \bigvee_{l} s^i_l \not\equiv s^j_l \right) \tag{16}$$

for the state variables. Effectively, in the k-step induction, only simple paths (on which a state never appears twice) are considered. With the uniqueness constraint, the termination of the k-step induction is guaranteed.

To summarize, the basic procedure for proving a safety property by the k-step induction (with the uniqueness criterion) proceeds as follows. We start from $k = 1$, i.e., the normal induction. In a k-step induction, three cases may happen. If Equations (14) and (15) are both satisfied, the property is proven to be true for the underlying transition system. If Equation (14) is violated, the property does not hold for the underlying state transition system. In these two cases, the induction terminates. On the other hand, if Equation (15) is violated, we increase k by 1, and repeat the induction. In fact, k is upper-bounded by the length of the longest simple path from an initial state in the underlying STG, which is not smaller and, in fact, can be exponentially larger than the forward diameter (the upper bound for forward reachability analysis). An example of such a worst-case is a complete graph.

The induction criteria (including Equations (14), (15), and the uniqueness criterion) can be formulated as SAT solving, especially, incremental SAT solving [ES03]. Therefore, in addition to the interpolation-based approach in Section 6.2, k-step induction is an alternative solution to unbounded model checking of safety properties.

7.4 Some Reduction Techniques

Re-encoding States for Similarity The aforementioned computation of register correspondence and that of functional dependency identify the functional connections among registers.

Consider the special case of two state-minimized equivalent FSMs. When their corresponding equivalent states are encoded in the same way, they are in fact combinationally equivalent. Once their register correspondence is established, combinational check suffices to show sequential equivalence. On the other hand, when the corresponding equivalent states are encoded differently, no register correspondence can be concluded in general. In this case, more powerful techniques, such as the identification of functional dependency, are needed.

When two FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ have a one-to-one equivalent state mapping, there exists some re-encoding able to produce identical next-state and output functions thus reducing again sequential to combinational equivalence. (The mapping needs not hold for the entire state sets of $\mathcal{M}_1$ and $\mathcal{M}_2$, and can hold merely for their exact or over-approximated reachable state sets.) How to find such re-encoding is a problem that has been addressed in the work by Quer et al. [QCC+00]. Even when complete re-encoding cannot be found, due to computational expensiveness or nonexistence, partial re-encoding may increase the similarity between the two FSMs and simplify their product machine. Notice that the number of states of a product machine may be larger by an exponential factor with respect to the number of states of the constituent machines, even though by experimental evidence it is usually larger by only a small constant factor. The technique was suggested to work also for sequential circuits with different number of registers, to ease sequential equivalence checking.

State minimization of a sequential circuit without explicit state space exploration has been studied in [LN91,Tam93,JB03], where equivalence classes are represented with BDDs.

Exposing Registers for Observability One can view the process to check the equivalence of FSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ through the product machine $\mathcal{M}_\times$ as constructing a relation on pairs of states $R \subseteq Q_1 \times Q_2$ such that $(q_1, q_2) \in R$ if they are simultaneously reachable, and they produce the same outputs under the same inputs, namely,

$$(q_1, q_2) \in R \Leftrightarrow ((q_1, q_2) \text{ reachable in } \mathcal{M}_\times \wedge \forall \sigma \in \Sigma.(\lambda_1(\sigma, q_1) \equiv \lambda_2(\sigma, q_2))).$$

It can be proved that the two FSMs are equivalent if and only if the relation R forms a *bi-simulation relation*, which for every $(q_1, q_2) \in R$ and every $\sigma \in \Sigma$ satisfies

1. $(q_1^0, q_2^0) \in R$ for $q_1^0 \in Q_1^0$ and $q_2^0 \in Q_2^0$,
2. $\delta_1(\sigma, q_1) = q_1'$ implies $\exists q_2' \in Q_2.(\delta_2(\sigma, q_2) = q_2' \wedge (q_1', q_2') \in R)$, and
3. $\delta_2(\sigma, q_2) = q_2'$ implies $\exists q_1' \in Q_1.(\delta_1(\sigma, q_1) = q_1' \wedge (q_1', q_2') \in R)$.

In other words, the two FSMs are equivalent if and only if firstly their initial states are equivalent, secondly for every state q_1' (q_2') to which q_1 (q_2) can go, there is a state q_2' (q_1') to which q_2 (q_1) can go under the same input, and thirdly q_1' and q_2' are equivalent. In order to obtain R, we need to find the set of simultaneously reachable state pairs by state traversal of $\mathcal{M}_\times$.

By exposing register outputs as observable pseudo primary outputs, state equivalence is simplified to 1-step state equivalence (recall Definition 6). The idea was exploited in [AGM01] to check the bi-simulation property of a state relation between $\mathcal{M}_1$ and $\mathcal{M}_2$ under 1-step state equivalence.

Consider constructing, instead of the relation R, another relation $R^\dagger \subseteq Q_1 \times Q_2$, called *1-equivalence relation* in [AGM01], with

$$(q_1, q_2) \in R^\dagger \Leftrightarrow (q_1 \text{ reachable in } \mathcal{M}_1 \wedge q_2 \text{ reachable in } \mathcal{M}_2 \wedge \forall \sigma \in \Sigma.(\lambda_1(\sigma, q_1) \equiv \lambda_2(\sigma, q_2))).$$

That is, two states $q_1 \in Q_1$ and $q_2 \in Q_2$ with $(q_1, q_2) \in R^\dagger$ if they are *individually* reachable in their constituent FSMs and produce the same outputs under the same inputs. Therefore $R^\dagger$ differs from R in that it holds between pairs of states *individually* reachable in the two FSMs. So to compute $R^\dagger$ we need not find out which states are *simultaneously* reachable by traversing the product machine, but we only need to know the set of reachable states of each individual FSM.

A question to ask is whether or not still the proposition holds that $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent if and only if the relation $R^\dagger$ is a bi-simulation relation. Since $R \subseteq R^\dagger$, it is immediate that, if $R^\dagger$ satisfies the bi-simulation property, then so does R. Hence the checking is sufficient to verify the equivalence of $\mathcal{M}_1$ and $\mathcal{M}_2$. However, since $R^\dagger$ is an over-approximation of R, there can be a false negative, i.e., a case when the check fails on a pair of states that are not simultaneously reachable.

To rule out such false negatives, in [AGM01] the *complete-1-distinguishability* (C-1-D) property, defined as any two states can be distinguished at the outputs with an input sequence of length 1, is imposed upon the states of $\mathcal{M}_1$ and $\mathcal{M}_2$. By restricting $\mathcal{M}_1$ to obey the C-1-D property, it can be proved that, if $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent, then $R^\dagger$ is a bi-simulation relation. Conversely, if $R^\dagger$ is not a bi-simulation relation, then for sure $\mathcal{M}_1$ and $\mathcal{M}_2$ are inequivalent. The C-1-D property avoids the false negatives because, when the property holds for $\mathcal{M}_1$, for each state q_2 in $\mathcal{M}_2$ there is exactly one state q_1 in $\mathcal{M}_1$ such that $(q_1, q_2) \in R^\dagger$, and also, if q_2 is reachable in $\mathcal{M}_2$, then (q_1, q_2) is reachable in $\mathcal{M}_\times$. So, when $\mathcal{M}_1$ has the C-1-D property, the set of pairs of equivalent separately reachable states is the same as the set of pairs of equivalent simultaneously reachable states.

The C-1-D property can be viewed as a condition under which backward reachability analysis attains convergence in one iteration, because backward traversal refines iteratively the partition obtained initially by 1-step state equivalence. (Connections between reachability analysis and state equivalence are detailed in Section 5.2.)

The above discussion can be straightforwardly generalized to k-equivalence relation and complete-k-distinguishability. To make sure that $\mathcal{M}_1$ is 1- or k-distinguishable, some registers have to be exposed as observable primary outputs. It imposes restrictions on circuit synthesis. When all registers are exposed, the circuit can only be synthesized combinationally keeping intact register boundaries. Therefore the method can be seen as intermediate between pure combinational synthesis and verification versus unrestricted sequential synthesis and verification.

8 A Hierarchy of Classes of Sequential Equivalence

In the previous exposition, we assumed that FSMs can start from their prespecified initial states, so that equivalence checking reduces to verifying the equivalence of the initial states. This initialization assumption however should be questioned because in reality a circuit may not begin with the desired initial state and this fact has to be enforced somehow. We are going to discuss next what happens when there are no known initial states. In fact, the way an FSM is initialized affects the definition of sequential equivalence. We first study the initialization problem of FSMs and then discuss some variants of sequential equivalence.

8.1 Resetability and Alignability

In hardware implementation of an FSM, it is not immediate that the implemented FSM can be prepared in its designated initial states. This is because, when a circuit is powered up, the latched values in the state-holding elements (registers) are uncontrollable and can be of value either 0 or 1 indefinitely. Therefore, the circuit can be in an arbitrary state. An initialization (or reset) mechanism is needed to drive the circuit into some designated initial states before the FSM can operate correctly. A simple approach to rectifying the problem of uncertain power-up states is to add reset circuitry for every register. (A register is of *explicit reset* if some reset circuitry is associated with it. Otherwise, a register is of *implicit reset.*) Once a circuit is powered up, a reset signal is activated to trigger the reset mechanism for a register to latch its right initial value. However, this approach may incur heavy area penalties especially for a design with an excessive number of registers because the reset signal needs to be connected to all the registers. Another solution (without incurring any hardware overhead) to the reset problem is to apply some input sequence enforcing registers to have desired values. On the other hand, a hybrid solution with *partial reset* is possible, where only a subset of the registers is selected for explicit reset.

It is noteworthy that, if we treat the additional reset circuitry as an integral part of a design and thus the reset signal is part of the primary inputs, explicit reset can be seen as a special case of implicit reset. Consequently, we may only need to focus on implicit reset as we shall do in the sequel. Because the input sequences for implicit reset may not always exist, we study the resetability of an FSM. Moreover, for a resetable FSM, we would like to obtain an input sequence that resets the FSM.

To reflect the situation that a circuit when powered up may not be in designated initial states, we modify the definition of an FSM and speak of a *hardware finite state machine* instead.

Definition 16 (Hardware Finite State Machine). *A* hardware finite state machine *(HFSM) is a tuple* $(Q, \Sigma, \Omega, \boldsymbol{\delta}, \boldsymbol{\lambda})$ *or* $(Q, \Sigma, \Omega, T, \boldsymbol{\lambda})$*, similar to the definition of an FSM except for the absence of an initial state set.*

We define a *hardware state transition graph* (HSTG) for an HFSM in the same way as an STG for an FSM except that initial states are not identified. An HFSM (HSTG) is more primitive than an FSM (STG) in the sense that there is no notion of initial states.

Given an HFSM and some target initial state, we may study if the HFSM can be brought to this state through some input sequence from any power-up states.

Definition 17 (Strict Resetability). *An HFSM* $\mathcal{H} = (Q, \Sigma, \Omega, T, \boldsymbol{\lambda})$ *is called* strictly resetable *if there exists an input sequence* $\boldsymbol{\sigma} \in \Sigma^*$ *(known as the* reset sequence *or* synchronization sequence*) and a state* $q_0 \in Q$ *(known as the* reset state*) such that* $Img_{\boldsymbol{\sigma}}(q, T) = q_0$*, for any* $q \in Q$*.*

From the above definition, an FSM $\mathcal{M}$ with a single initial state can be derived from an HFSM $\mathcal{H}$ if $\mathcal{H}$ is resetable with respect to the initial state of $\mathcal{M}$. Notice that a reset sequence is *universal* in the sense that it applies to arbitrary power-up states. By initializing an HFSM to a single reset state, Definition 17 may seem somewhat restricted. When in particular two states are equivalent, it does not matter if the HFSM is reset to anyone of them. It brings up a relaxed definition of resetability.

Definition 18 (Essential Resetability). *An HFSM is called* essentially resetable *if, under any power-up states, it can be initialized to a set of equivalent reset states* I *through a common reset sequence.*

In hardware design, FSMs are supposed to behave deterministically after reset. Therefore, unless the reset states I are all equivalent, the HFSM $\mathcal{H}$ may behave differently depending on which state in I it is reset to. It is the reason that we assume all initial states must be equivalent.

Strict resetability and essential resetability can be connected as follows.

Proposition 5. *An HFSM is essentially resetable if and only if its quotient (i.e., state-minimized) HFSM is strictly resetable.*

(A quotient HFSM is defined the same as a quotient FSM except for the ignorance of initial states.) The proof can be shown straightforwardly by the deferred

Lemma 1, and is omitted. Essential resetability is more adequate than strict resetability in the context of hardware design. In the sequel, when resetability is referred, we shall mean essential resetability unless otherwise said. Equivalently, we may assume an HFSM is in its quotient form and speak of strict resetability.

Algorithms for resetability analysis, as those for reachability analysis, are heavily influenced by the data structures used to realize the computations. Following the historical development, we begin with an enumerative approach to resetability analysis as well as reset sequence generation, and then we proceed with an algorithm that lends itself to a symbolic implementation.

Explicit Graph Algorithm for Resetability Analysis The reset problem can be resolved based on explicit graph enumeration [Koh78]. Essentially, the resetability of an HFSM can be easily understood through a tree construction.

Definition 19 (Synchronization Tree). *The* synchronization tree *of an HFSM $\mathcal{H} = (Q, \Sigma, \Omega, T, \boldsymbol{\lambda})$ is an infinite tree $G = (V, E)$, where a vertex $v \in V$ corresponds to a state set $Q_v \subseteq Q$ and an edge labelled with $\sigma \in \Sigma$ from vertex u to vertex v signifies $Q_v = Img_\sigma(Q_u, T)$. The unique root vertex $r \in V$ corresponds to the universal state set Q, i.e., $Q_r = Q$.*

A synchronization tree enumerates all possible input sequences and lists the corresponding possible states of the considered HFSM.

Example 17. Figure 22 shows an example, where T is the synchronization tree of the given HSTG G. Under an empty input sequence (at power-up), the HFSM may be in any state as indicated in the root node; under the input sequence '0', it is possible to be in any state of $\{q_0, q_1, q_3\}$ as indicated in the left successor node of the root.

Since we are concerned with finite state sets, labels on vertices will repeat eventually. A finite construction suffices for an exhaustive enumeration.

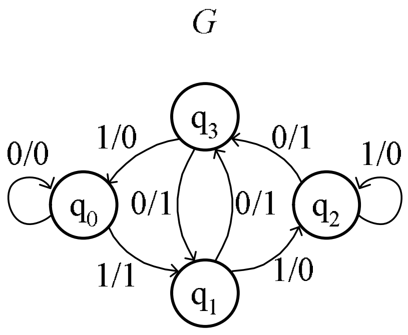
The relation between a synchronization tree and the resetability of an HFSM can be stated as follows.

Proposition 6. *An HFSM is resetable with respect to a state set $I \subseteq Q$ if and only if its corresponding synchronization tree has a path from the root vertex r to some vertex v with its corresponding state set $Q_v \subseteq \{q \in Q \mid \exists q_i \in I.q \sim q_i\}$. Also, the ordered labels on the edges along this path give the corresponding reset sequence.*

(Assume that all the states in I are equivalent.) In hardware design, short reset sequences are preferable to long ones.

Example 18. The HSTG of Figure 22 is resetable to any reset state because input sequences '1,1,0', '1,1,1', '1,1,0,0', and '1,1,0,1' are reset sequences for reset states q_3, q_2, q_1, and q_0, respectively.

Whereas the synchronization tree gives an exact characterization of resetability, there are some interesting necessary conditions for an HFSM to be resetable



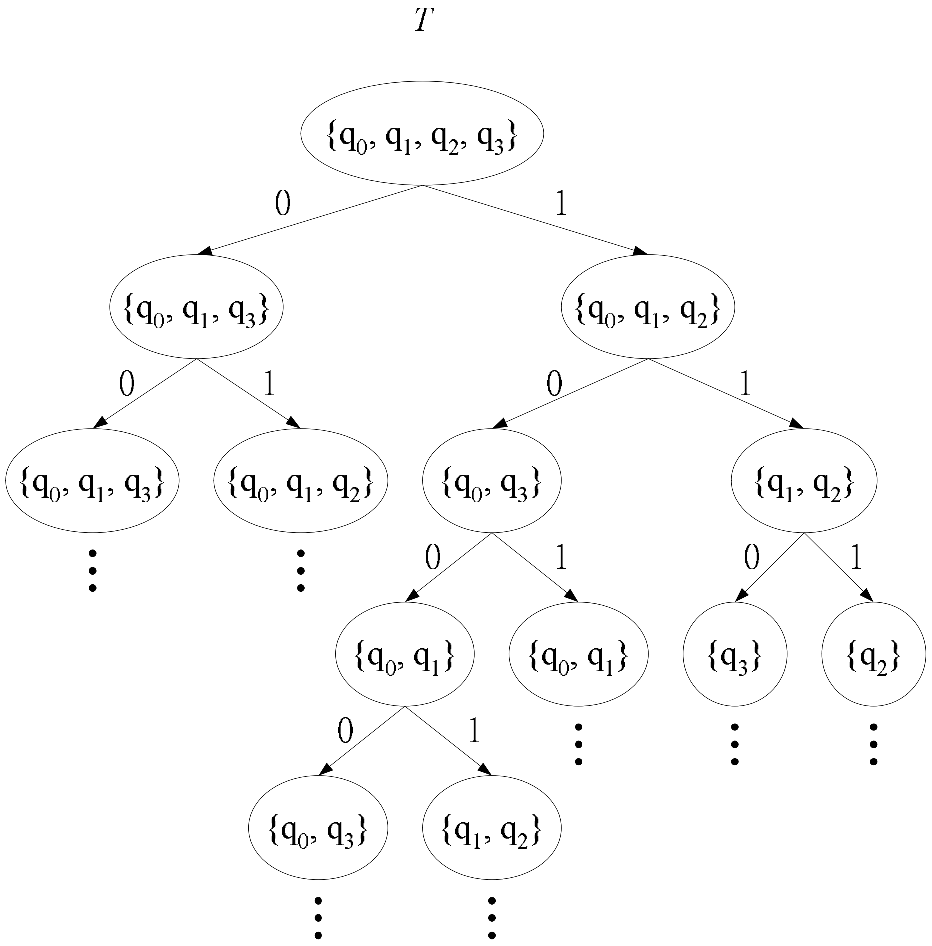


Fig. 22. A hardware state transition graph and its corresponding synchronization tree.

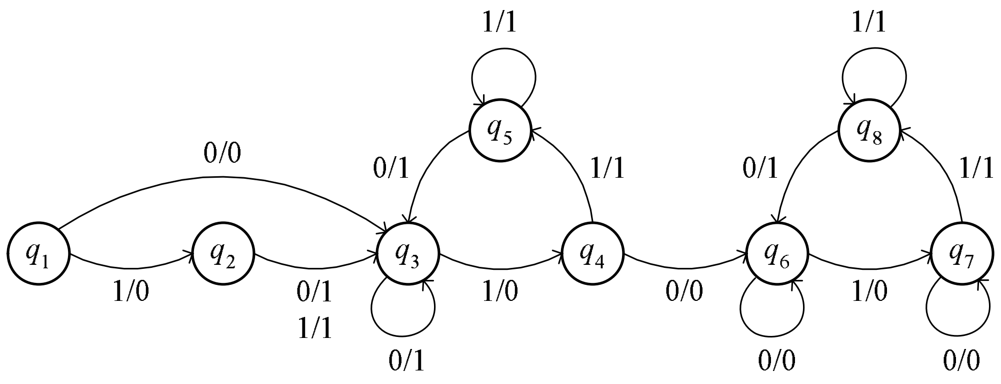


Fig. 23. A hardware state transition graph.

that may be used for fast screening of resetability. Whether an HFSM is resetable depends on the structure of its HSTG, in particular, the *strongly connected components* (SCCs).

Definition 20 (Strongly Connected Component). *A subgraph G of an STG or HSTG is a* strongly connected component *if every (ordered) vertex pair (u, v) of G is connected by some path from u to v. An SCC is called* closed *if its vertex set corresponds to a closed state set. Otherwise it is* open.

Note that there may exist an open SCC contained by a closed SCC, but not the converse, that is, a closed SCC cannot be contained by an open or closed SCC except for itself. If fact, any two distinct closed SCCs must be disjoint.

Example 19. In Figure 23, the open SCC induced by vertex q_8 is contained in the closed SCC induced by vertices q_6, q_7, q_8.

Proposition 7. *The reset states of an HFSM $\mathcal{H}$ must be inside a closed SCC, or equivalent to some state in a closed SCC of the HSTG of $\mathcal{H}$.*

Proof. For the sake of contradiction, assume the reset states of $\mathcal{H}$ are not in a closed SCC and inequivalent to any state in a closed SCC. Then, when powered up in a state in a closed SCC, $\mathcal{H}$ cannot be reset to the designated reset states. It contradicts with the assumption that $\mathcal{H}$ is resetable. ■

Proposition 8. *An HFSM $\mathcal{H}$ is resetable only if either its HSTG has a single closed SCC or all the state subgraphs induced by the closed SCCs are isomorphic after state minimization.*

Proof. Assume the HSTG of $\mathcal{H}$ has multiple inequivalent closed SCCs. Then $\mathcal{H}$ once powered up in some closed SCC cannot escape to another closed SCC. Hence $\mathcal{H}$ may not always be reset to some designated reset state. It contradicts with the assumption that $\mathcal{H}$ is resetable. ■

However, the converse is not true because states in a closed SCC may not be resetable at all.

Proposition 9. *For a resetable HFSM, any state reachable from a reset state must be a reset state.*

Proof. After a reset sequence $\boldsymbol{\sigma}$ is applied, an HFSM is in an equivalence class of states I (the reset states with respect to $\boldsymbol{\sigma}$). Let state $q^{\dagger}$ be reachable from some state in I under input sequence $\boldsymbol{\sigma}^{\dagger}$. Then, all such $q^{\dagger}$ must be equivalent since states in I are all equivalent, and thus form another set of equivalent reset states with respect to a new reset sequence $\boldsymbol{\sigma} \circ \boldsymbol{\sigma}^{\dagger}$. ■

This proposition can be justified with Example 18.

The resetability analysis discussed above relies on explicit enumeration over the synchronization tree. An alternative approach to resetability verification relies on reachability analysis and can be implemented symbolically with BDDs [PJH94].

Implicit Symbolic Algorithm for Resetability Analysis Resetability analysis can be reformulated in terms of the alignability of a pair of states.

Definition 21 (State Alignment). *Two states q_1 and q_2 of an HFSM $\mathcal{H} = (Q, \Sigma, \Omega, T, \boldsymbol{\lambda})$ are* alignable *if there exists an* aligning sequence $\boldsymbol{\sigma} \in \Sigma^*$ *of q_1 and q_2 such that $Img_{\boldsymbol{\sigma}}(q_1, T) \sim Img_{\boldsymbol{\sigma}}(q_2, T)$.*

Lemma 1. *For an HFSM $\mathcal{H} = (Q, \Sigma, \Omega, T, \boldsymbol{\lambda})$, if two states $q_1, q_2 \in Q$ are alignable under some input sequence $\boldsymbol{\sigma} \in \Sigma^*$, then states $q_3, q_4 \in Q$, with $q_3 \sim q_1$ and $q_4 \sim q_2$, are alignable as well under $\boldsymbol{\sigma}$.*

Proof. Since q_1 and q_2 are alignable under $\boldsymbol{\sigma}$, $Img_{\boldsymbol{\sigma}}(q_1, T) \sim Img_{\boldsymbol{\sigma}}(q_2, T)$. On the other hand, $q_3 \sim q_1$ implies $Img_{\boldsymbol{\sigma}}(q_3, T) \sim Img_{\boldsymbol{\sigma}}(q_1, T)$. Similarly, $Img_{\boldsymbol{\sigma}}(q_4, T) \sim Img_{\boldsymbol{\sigma}}(q_2, T)$. It follows that $Img_{\boldsymbol{\sigma}}(q_3, T) \sim Img_{\boldsymbol{\sigma}}(q_4, T)$. That is, q_3 and q_4 are alignable under $\boldsymbol{\sigma}$. ■

Consequently, we may speak of alignments over state equivalence classes rather than individual states. To compute reset sequences, it is more convenient to first reduce an HSTG to its quotient (state-minimized) HSTG such that no state equivalence needs to be checked later on.

The following theorem shows that a universal reset sequence can be built from concatenating local aligning sequences that align pairwise states.

Theorem 5. *An HFSM is resetable if and only if every of its state pairs is alignable under some finite input sequence.*

Proof. ($\Rightarrow$) By the definition of HFSM resetability, all state pairs are alignable (under a unique reset sequence).

($\Leftarrow$) For an HFSM with any pair of its states alignable, a universal sequence that aligns all state pairs can be constructed from local aligning sequences. Figures 24 and 25 show a procedure that determines if a given HFSM is resetable and returns a reset sequence if it exists. The correctness of the procedure is proved in Theorem 6. ■

```
Algorithm: ComputeResetSequence
  input: a state-minimized HFSM $\mathcal{H} = (Q, \Sigma, \Omega, T, \boldsymbol{\lambda})$
  output: a reset sequence
  begin
  01  $i := 0$
  02  $S^{(i)} := Q$
  03  repeat
  04     $i := i + 1$
  05     $\boldsymbol{\sigma}_i := ComputeAligningSequence(s_a, s_b, \mathcal{H})$ for some $s_a, s_b \in S^{(i-1)}$ and $s_a \neq s_b$
  06     if $\boldsymbol{\sigma}_i = \emptyset$
  07        return HFSM unresetable
  08     $S^{(i)} := Img_{\boldsymbol{\sigma}_i}(S^{(i-1)}, T)$
  09  until $|S^{(i)}| < 2$
  10  return $\boldsymbol{\sigma}_1 \circ \cdots \circ \boldsymbol{\sigma}_i$
  end
```

Fig. 24. An algorithm that computes a universal reset sequence for a given HFSM.

The procedure can be implemented with BDD-based computations [PJH94].

Theorem 6. *The procedure in Figures 24 and 25 terminates and returns a correct answer upon termination.*

Proof. The algorithm *ComputeResetSequence* in Figure 24 implicitly assumes $|Q| \geq 2$. Otherwise, the given HFSM $\mathcal{H}$ is readily in its reset state when powered up, and thus there is no need to compute the reset sequence.

In the iterative computation, two states not aligned yet are selected for alignment. If the two states are unalignable, they provide a certificate of non-resetability of the given HFSM and the program terminates. Otherwise, the set $S^{(i)}$ in Figure 24 decreases monotonically since at each iteration a state pair is aligned. Because $|S^{(i-1)}| - |S^{(i)}| \geq 1$ and $S^{(0)} = Q$, there exists some j such that $|S^{(j)}| < 2$ and thus the procedure terminates.

To see that the computed input sequence is indeed a reset sequence, consider first the algorithm *ComputeAligningSequence*. The procedure consists of two disjoint loops. The first loop performs forward reachability to check if state pair (q_a, q_b) in the product machine of $\mathcal{H} \times \mathcal{H}$ can reach some state $(q, q) \in Q \times Q$. (The product of two HFSMs is defined similarly to that of two FSMs except for ignoring the initial state set.) If the answer is negative, states q_a and q_b are not alignable and an empty input sequence is returned. Otherwise, the input sequence for (q_a, q_b) to reach (q, q) is computed by backward reachability analysis in the second loop. The computed aligning sequences are concatenated and returned by algorithm *ComputeResetSequence*. The overall input sequence drives the given HFSM $\mathcal{H}$ to a unique state (i.e. the corresponding reset state) upon termination. That unique state is the only state in the final image $S^{(i)} := Img_{\boldsymbol{\sigma}_i}(S^{(i-1)}, T)$ produced by the algorithm at the last cycle, when the condition $|S^{(i)}| < 2$ holds. ■

```
Algorithm: ComputeAligningSequence
  input: two distinct states $q_a, q_b$ and their underlying HFSM $\mathcal{H}$
  output: an input sequence that aligns the given state pair
  begin
  01  let $T_\times$ be the transition relation of the product machine $\mathcal{H} \times \mathcal{H}$
  02  $i := 0$
  03  $R^{(0)} := (q_a, q_b)$
  04  repeat
  05    $i := i + 1$
  06    $R^{(i)} := R^{(i-1)} \cup Img(R^{(i-1)}, T_\times)$
  07    $D := R^{(i)} \cap \{(q, q) \mid q \in Q\}$
  08  until $D \neq \emptyset$ or $R^{(i)} = R^{(i-1)}$
  09  if $D = \emptyset$
  10    return $\emptyset$
  11  $n := i$
  12  let $(q_d^{(i)}, q_d^{(i)}) \in D$
  13  repeat
  14    let $(q_d^{(i-1)}, q_d^{(i-1)}) \in \{R^{(i-1)} \cap PreImg((q_d^{(i)}, q_d^{(i)}), T_\times)\}$
  15    $\sigma_i := \arg_{\boldsymbol{x}}[T_\times(\boldsymbol{x}, (q_d^{(i-1)}, q_d^{(i-1)}), (q_d^{(i)}, q_d^{(i)}))]$
  16    $i := i - 1$
  17  until $i = 0$
  18  return $\sigma_1 \circ \cdots \circ \sigma_n$
  end
```

Fig. 25. An algorithm that computes an aligning sequence for a given state pair under a given transition relation.

If the resetability of an HFSM is the only primary concern but not the reset sequences, then we may test the resetability more effectively based on the following corollary.

Corollary 2. *An HFSM $\mathcal{H} = (Q, \Sigma, \Omega, T, \boldsymbol{\lambda})$ is strictly resetable if and only if any state of the product HFSM $\mathcal{H} \times \mathcal{H} = (Q \times Q, \Sigma, \Omega, T_\times, \lambda_\times)$ can reach some state in $S = \{(q, q) \mid q \in Q\}$, i.e., the backward reachable state set of S equals the universal set.*

Proof. The corollary follows from Theorem 5. ■

A similar argument holds for essential resetability by modifying S to be the set $\{(q_1, q_2) \mid q_1, q_2 \in Q, q_1 \sim q_2\}$.

8.2 A Landscape of Sequential Equivalences

Depending on the equivalence criteria of HFSMs in the initialization phase, we may have different notions of equivalence.

Alignment Equivalence Definition 21 and Theorem 5 can be generalized to the alignment of two different HFSMs.

Definition 22 (HFSM Alignment and Alignment Equivalence). *Two HFSMs $\mathcal{H}_1 = (Q_1, \Sigma, \Omega, T_1, \boldsymbol{\lambda}_1)$ and $\mathcal{H}_2 = (Q_2, \Sigma, \Omega, T_2, \boldsymbol{\lambda}_2)$ are* alignable *if $\forall q_1 \in Q_1, q_2 \in Q_2, \exists \boldsymbol{\sigma} \in \Sigma^*. Img_{\boldsymbol{\sigma}}(q_1, T_1) \sim Img_{\boldsymbol{\sigma}}(q_2, T_2)$. In this case, we say that $\mathcal{H}_1$ and $\mathcal{H}_2$ are* alignment equivalent*, denoted as $\mathcal{H}_1 \cong \mathcal{H}_2$.*

Similar to Theorem 5, we have

Theorem 7. *Two HFSMs $\mathcal{H}_1 = (Q_1, \Sigma, \Omega, T_1, \boldsymbol{\lambda}_1)$ and $\mathcal{H}_2 = (Q_2, \Sigma, \Omega, T_2, \boldsymbol{\lambda}_2)$ are alignable (or alignment equivalent) if and only if $\exists \boldsymbol{\sigma} \in \Sigma^*, \forall q_1 \in Q_1, q_2 \in Q_2. Img_{\boldsymbol{\sigma}}(q_1, T_1) \sim Img_{\boldsymbol{\sigma}}(q_2, T_2)$.*

Proof. ($\Leftarrow$) This direction follows from Definition 22.

($\Rightarrow$) A universal aligning sequence can be constructed in a way similar to the procedure *ComputeResetSequence* in Figure 24. ■

In other words, $\mathcal{H}_1 \cong \mathcal{H}_2$ if they share a common reset sequence and behave equivalently after reset. Alignment equivalence is more stringent than FSM equivalence since it requires that two FSMs share a common reset sequence in addition to indistinguishable input-output behavior after reset. However, note that, by the definition of alignment equivalence of two HFSMs, their input-output behaviors during the reset phase need not be identical.

Theorem 8. *Alignment equivalence $\cong$ is symmetric and transitive, but not necessarily reflexive in general.*

Proof. For $\mathcal{H}_1 \cong \mathcal{H}_2$, any state pair (q_1, q_2), with q_1 of $\mathcal{H}_1$ and q_2 of $\mathcal{H}_2$, is alignable and independent of the order of $\mathcal{H}_1$ and $\mathcal{H}_2$. Therefore, alignment equivalence is symmetric.

To prove the transitivity, assume $\mathcal{H}_1 \cong \mathcal{H}_2$ and $\mathcal{H}_1 \cong \mathcal{H}_3$. We show $\mathcal{H}_2 \cong \mathcal{H}_3$. There exists a reset sequence $\boldsymbol{\sigma}_{1,2}$ aligning $\mathcal{H}_1$ and $\mathcal{H}_2$ and a reset sequence $\boldsymbol{\sigma}_{1,3}$ aligning $\mathcal{H}_1$ and $\mathcal{H}_3$. Let $\boldsymbol{\sigma} = \boldsymbol{\sigma}_{1,2} \circ \boldsymbol{\sigma}_{1,3}$. We claim that $\boldsymbol{\sigma}$ is a reset sequence aligning any pair of $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$. To see it, consider first the alignment of $\mathcal{H}_1$ and $\mathcal{H}_2$ under $\boldsymbol{\sigma}$. After $\boldsymbol{\sigma}_{1,2}$ is applied, $\mathcal{H}_1$ and $\mathcal{H}_2$ are aligned and the current states of $\mathcal{H}_1$ and $\mathcal{H}_2$ are in a single equivalence class. Now applying another input sequence $\boldsymbol{\sigma}_{1,3}$ on $\mathcal{H}_1$ and $\mathcal{H}_2$ does not drive equivalent states to non-equivalent states. Therefore, the new current states of $\mathcal{H}_1$ and $\mathcal{H}_2$ after applying $\boldsymbol{\sigma}$ remain in a single equivalence class. That is, for any $q_1 \in Q_1$ and $q_2 \in Q_2$, let $q_1' = Img_{\boldsymbol{\sigma}}(q_1, T_1)$ and $q_2' = Img_{\boldsymbol{\sigma}}(q_2, T_2)$. Then $q_1' \sim q_2'$. On the other hand, $\boldsymbol{\sigma}$ must be an alignment sequence for $\mathcal{H}_1$ and $\mathcal{H}_3$ because, no matter what state pair $(q_1^\dagger, q_3^\dagger)$ is reached after applying $\boldsymbol{\sigma}_{1,2}$ to $\mathcal{H}_1$ and $\mathcal{H}_3$, it can always be aligned by $\boldsymbol{\sigma}_{1,3}$. Thus, for any $q_3 \in Q_3$, let $q_3' = Img_{\boldsymbol{\sigma}}(q_3, T_3)$. Then $q_1' \sim q_3'$, from which $q_2' \sim q_3'$, and thus $\mathcal{H}_2 \cong \mathcal{H}_3$.

Alignment equivalence is not necessarily reflexive because, if an HFSM is not essentially resetable at all, it cannot be alignment equivalent to itself. ■

Theorem 9. *An HFSM is alignment equivalent to itself if and only if it is essentially resetable.*

Proof. Let $\mathcal{H}^\dagger$ be another copy of an HFSM $\mathcal{H}$.

($\Rightarrow$) Suppose $\mathcal{H}$ is alignment equivalent to itself. Then any state pair $(q_1, q_2^\dagger)$ for q_1 of $\mathcal{H}$ and $q_2^\dagger$ of $\mathcal{H}^\dagger$ can be aligned by a universal reset sequence. Since every state of one HFSM, say $q_2^\dagger$ of $\mathcal{H}^\dagger$, has a corresponding equivalent state of the other, say q_2 of $\mathcal{H}$, then state pair (q_1, q_2) of $\mathcal{H}$ can be aligned in $\mathcal{H}$ with the same reset sequence by Lemma 1. Thus $\mathcal{H}$ must be essentially resetable.

($\Leftarrow$) If $\mathcal{H}$ is essentially resetable, then any state pair (q_1, q_2) of $\mathcal{H}$ can be aligned by some universal reset sequence $\boldsymbol{\sigma}$. Consider the disjoint union state space of $\mathcal{H}$ and $\mathcal{H}^\dagger$. Every state of one HFSM has a corresponding equivalent state of the other, say $q_2 \sim q_2^\dagger$ for state $q_2^\dagger$ of $\mathcal{H}^\dagger$. Again by Lemma 1, $(q_1, q_2^\dagger)$ has the same reset sequence $\boldsymbol{\sigma}$. Hence $\mathcal{H}$ and $\mathcal{H}^\dagger$ are alignment equivalent. ■

In other words, alignment equivalence $\cong$ is reflexive only for essentially resetable HFSMs. Therefore, alignment equivalence $\cong$ is an equivalence relation among essentially resetable HFSMs.

Corollary 3. *If $\mathcal{H}_1 \cong \mathcal{H}_2$, then $\mathcal{H}_1 \cong \mathcal{H}_1$.*

Proof. From Theorem 8, $\mathcal{H}_1 \cong \mathcal{H}_2$ implies $\mathcal{H}_2 \cong \mathcal{H}_1$ by symmetry. Further, $\mathcal{H}_1 \cong \mathcal{H}_1$ holds by transitivity. ■

For two HFSMs, suppose that any state in one HFSM has a corresponding equivalent state in the other. Even then, it is not necessarily the case that there exists an input sequence that aligns these two HFSMs. This is similar to the fact that the converse condition of Proposition 8 does not hold. However the next theorem shows that then unalignability implies that both of the two HFSMs are unresetable.

Theorem 10. *Given two HFSMs, assume any state in one machine has some corresponding equivalent state in the other. Then these two HFSMs are alignable if and only if one of them (equivalently, each of them) is essentially resetable.*

Proof. Let Q and $Q^\dagger$ be the state sets of HFSMs $\mathcal{H}$ and $\mathcal{H}^\dagger$, respectively. Assume any state of $\mathcal{H}$ has a corresponding equivalent state of $\mathcal{H}^\dagger$, and vice versa.

($\Rightarrow$) Assume $\mathcal{H} \cong \mathcal{H}^\dagger$. Then, by Lemma 1, any state pair $(q_1, q_2) \in Q \times Q$ of $\mathcal{H}$ must be alignable because there always exists $(q_1, q_2^\dagger) \in Q \times Q^\dagger$, with $q_2^\dagger \sim q_2$, alignable between $\mathcal{H}$ and $\mathcal{H}^\dagger$. Therefore, $\mathcal{H}$ is essentially resetable. Similarly, we know $\mathcal{H}^\dagger$ is essentially resetable as well.

($\Leftarrow$) Without loss of generality, assume $\mathcal{H}$ is resetable. Then, by Lemma 1, any state pair $(q_1, q_2^\dagger) \in Q \times Q^\dagger$ must be alignable between $\mathcal{H}$ and $\mathcal{H}^\dagger$ because there always exists $(q_1, q_2) \in Q \times Q$, with $q_2^\dagger \sim q_2$, alignable in $\mathcal{H}$. Therefore, $\mathcal{H}$ and $\mathcal{H}^\dagger$ are alignable. ■

Theorem 11. *Consider the closed SCCs of the HSTGs of two alignment equivalent HFSMs. Suppose that power-up states are in the SCCs. Then any reset sequence for one HFSM is also a reset sequence for the other.*

Proof. In the SCCs, any state in one HFSM must have an equivalent state in the other. Otherwise, these two HFSMs are not alignment equivalent. On the other hand, any of the HFSMs is alignment equivalent to itself by Corollary 3, and thus resetable by Theorem 9. From Lemma 1, these two HFSMs must share the same set of reset sequences in the SCCs. ■

To study the structure of a reset sequence, we define

Definition 23. *A state q of an HFSM is* dangling *if it has no predecessor states (i.e., no states can transition to q) or all of its predecessor states are dangling. Otherwise, it is* non-dangling*.*

Example 20. Consider Figure 23. States q_1 and q_2 are dangling, and all others are non-dangling.

Any reset sequence $\boldsymbol{\sigma}$ can be decomposed into three subsequences such that $\boldsymbol{\sigma} = \boldsymbol{\sigma}_1 \circ \boldsymbol{\sigma}_2 \circ \boldsymbol{\sigma}_3$ with

1. Subsequence $\boldsymbol{\sigma}_1$ drives an HFSM out of dangling paths. In this phase, as long as $|\boldsymbol{\sigma}_1|$ is no less than the length of the longest dangling path, any input sequence is valid. (A shorter $\boldsymbol{\sigma}_1$ may require detailed knowledge about the transitions of the dangling paths of an HSTG.)
2. Subsequence $\boldsymbol{\sigma}_2$ drives an HFSM out of open SCCs and into a closed SCC.
3. Subsequence $\boldsymbol{\sigma}_3$ enforces an HFSM entering a reset state (in a closed SCC).

Notice that subsequences $\boldsymbol{\sigma}_1$ and $\boldsymbol{\sigma}_2$ may be empty.

Example 21. Continuing Figure 23, observe that the HSTG has a reset sequence $\boldsymbol{\sigma} =$ '$\underline{1,1}, \underline{0,1,0}, \underline{0,1,1}$' with reset state q_8. As analyzed above, $\boldsymbol{\sigma}$ can be decomposed into $\boldsymbol{\sigma}_1 =$ '1, 1', $\boldsymbol{\sigma}_2 =$ '0, 1, 0', and $\boldsymbol{\sigma}_3 =$ '0, 1, 1'.

For two alignment equivalent HFSMs, any state in a closed SCC of one HFSM must have an equivalent state in a closed SCC of the other. Therefore, by Theorem 11, subsequence $\boldsymbol{\sigma}_3$ is a reset subsequence common to all alignment equivalent HFSMs. Two alignment equivalent HFSMs can always have a common suffix.

In certain classes of circuit transformations, e.g. see [JB06], only dangling paths of an HSTG can be changed. In that case, (non-)resetability is preserved.

Reset-Independent Equivalences Recall that FSM equivalence is an equivalence relation over FSMs, where initial states are pre-specified; alignment equivalence is an equivalence relation only over essentially resetable HFSMs. Here we study some other notions of equivalence over HFSMs that are independent of initial states or resetability.

Definition 24 (Strict Equivalence). *Two HFSMs are* strictly equivalent *if, for every state in one machine, there is an equivalent state in the other.*

This definition corresponds to the definition of "FSM equivalence" in [HS66,Koh78], where the "FSM" is meant to be our "HFSM." Here we name it differently to avoid confusion.

If two HFSMs are strictly equivalent, they are equivalent according to Definition 9 as long as the initial states are properly specified. On the other hand, FSM equivalence (of Definition 9) implies HFSM strict equivalence under restriction to the reachable state subspace. It can be checked that HFSM strict equivalence forms an equivalence relation over resetable and non-resetable HFSMs.

Another equivalence definition [SP94], more relaxed than HFSM strict equivalence, is based on the notion of safe replacement.

Definition 25 (Safe Replacement). *An HFSM $\mathcal{H}_1 = (Q_1, \Sigma, \Omega, T_1, \boldsymbol{\lambda}_1)$ is a* safe replacement *for $\mathcal{H}_2 = (Q_2, \Sigma, \Omega, T_2, \boldsymbol{\lambda}_2)$, denoted as $\mathcal{H}_1 \sqsubseteq \mathcal{H}_2$, if $\forall q_1 \in Q_1, \boldsymbol{\sigma} \in \Sigma^*, \exists q_2 \in Q_2$ such that $\boldsymbol{\lambda}_1(\sigma_1, q_1) = \boldsymbol{\lambda}_2(\sigma_1, q_2)$ and $\boldsymbol{\lambda}_1(\sigma_i, Img_{\boldsymbol{\sigma}_{i-1}}(q_1, T_1)) = \boldsymbol{\lambda}_2(\sigma_i, Img_{\boldsymbol{\sigma}_{i-1}}(q_2, T_2))$ for $i = 2, \ldots, |\boldsymbol{\sigma}|$, where $\boldsymbol{\sigma}_j = \sigma_1 \sigma_2 \ldots \sigma_j$ is a substring of $\boldsymbol{\sigma}$.*

By comparing the above definition with Definition 8 of equivalent states, one notices that the difference is in the quantification order $\forall \boldsymbol{\sigma} \exists q_2$ here vs. $\exists q_2 \forall \boldsymbol{\sigma}$ in Definition 8. In Definition 25, a state $q_1 \in Q_1$ needs not have an equivalent state $q_2 \in Q_2$, but for every input sequence there may be a different state of Q_2 that behaves like q_1 under that input sequence. However, if $\mathcal{H}_2$ is replaced with $\mathcal{H}_1$ for $\mathcal{H}_1 \sqsubseteq \mathcal{H}_2$, the underlying environment will not experience any new response because the input-output behavior of $\mathcal{H}_1$ when powered up can always be simulated by $\mathcal{H}_2$.

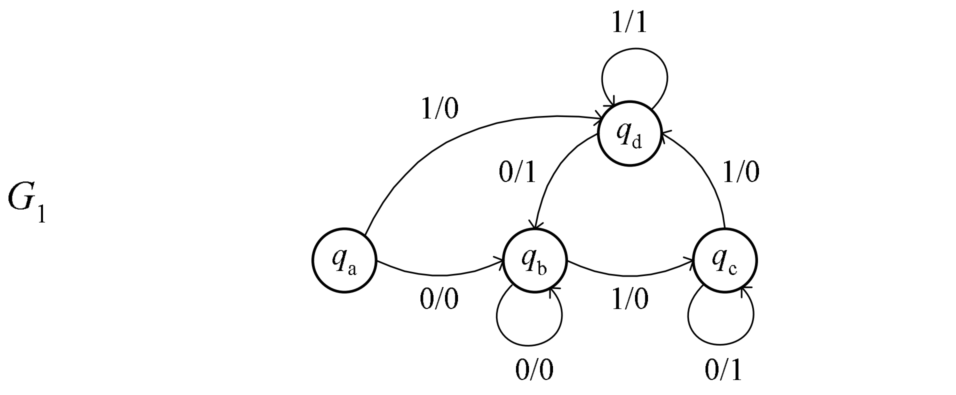
Example 22. Let G_1 and G_2 of Figure 26 be the HSTGs of HFSMs $\mathcal{H}_1$ and $\mathcal{H}_2$, respectively. Then $\mathcal{H}_1 \sqsubseteq \mathcal{H}_2$. This is because states q_b, q_c, and q_d of G_1 are equivalent to q_6, q_7, and q_8 of G_2, respectively. In addition, q_a of G_1 and q_4 of G_2 are equivalent under input 0 while q_a of G_1 and q_7 of G_2 are equivalent under input 1.

Definition 26 (Safe-Replacement Equivalence). *Two HFSMs $\mathcal{H}_1$ and $\mathcal{H}_2$ are* safe-replacement equivalent *if $\mathcal{H}_1 \sqsubseteq \mathcal{H}_2$ and $\mathcal{H}_2 \sqsubseteq \mathcal{H}_1$.*

Safe-replacement equivalence is reflexive, symmetric and transitive, and thus forms an equivalence relation. From the logical validity $\exists\forall \Rightarrow \forall\exists$, it follows that strict equivalence implies safe-replacement equivalence. Moreover, regardless of resetability, safe-replacement equivalence is the coarsest condition for the environment not being able to distinguish the replacement of one HFSM with another.

When restricted to *resetable* HFSMs, we can connect safe replacement and alignment equivalence as follows.

Theorem 12. *If HFSM $\mathcal{H}_1 = (Q_1, \Sigma, \Omega, T_1, \boldsymbol{\lambda}_1)$ is a safe replacement of a resetable HFSM $\mathcal{H}_2 = (Q_2, \Sigma, \Omega, T_2, \boldsymbol{\lambda}_2)$, then $\mathcal{H}_1$ and $\mathcal{H}_2$ are alignment equivalent.*



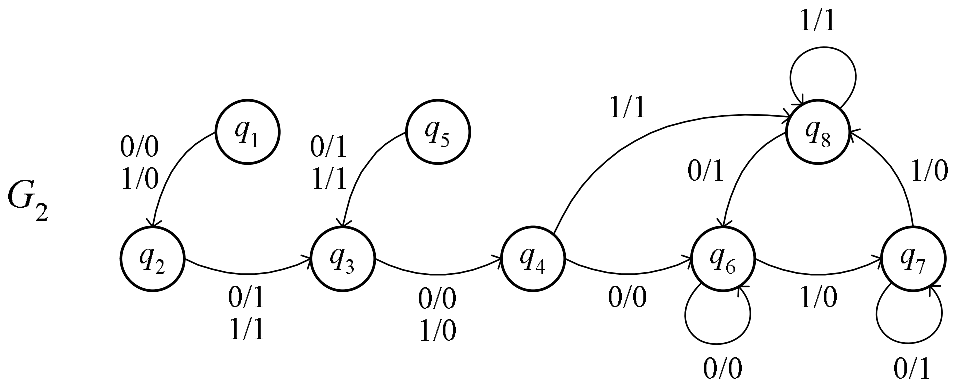


Fig. 26. Two HSTGs G_1 and G_2, where given any state in G_1 and any input sequence $\boldsymbol{\sigma}$, there exists a state in G_2 that generates the same output sequence under $\boldsymbol{\sigma}$.

Proof. We show that safe replacement preserves reset sequences, that is, any reset sequence of $\mathcal{H}_2$ is also a reset sequence of $\mathcal{H}_1$. In addition, $\mathcal{H}_1$ and $\mathcal{H}_2$ are aligned to equivalent states under the same reset sequence.

Assuming that $\boldsymbol{\sigma}$ is a reset sequence of $\mathcal{H}_2$, we must show $\forall q_1 \in Q_1, q_2 \in Q_2 . Img_{\boldsymbol{\sigma}}(q_1, T_1) \sim Img_{\boldsymbol{\sigma}}(q_2, T_2)$. Suppose by contradiction that there exists $q_1' \in Q_1$ such that $Img_{\boldsymbol{\sigma}}(q_1', T_1) \not\sim Img_{\boldsymbol{\sigma}}(q_2, T_2)$. Then there exist $\boldsymbol{\rho} \in \Sigma^*$ and $\sigma' \in \Sigma$ such that $\boldsymbol{\lambda}_1(\sigma', Img_{\boldsymbol{\sigma}\circ\boldsymbol{\rho}}(q_1', T_1)) \neq \boldsymbol{\lambda}_2(\sigma', Img_{\boldsymbol{\sigma}\circ\boldsymbol{\rho}}(q_2, T_2))$. But, since $\mathcal{H}_1$ is a safe replacement of $\mathcal{H}_2$, there is a state $q_2' \in Q_2$ such that $\boldsymbol{\lambda}_1(\sigma', Img_{\boldsymbol{\sigma}\circ\boldsymbol{\rho}}(q_1', T_1)) = \boldsymbol{\lambda}_2(\sigma', Img_{\boldsymbol{\sigma}\circ\boldsymbol{\rho}}(q_2', T_2))$. Because $\boldsymbol{\sigma}$ is a reset sequence of $\mathcal{H}_2$, so is $\boldsymbol{\sigma}\circ\boldsymbol{\rho}$ by Proposition 9. Consequently, for every $q_2 \in Q_2$, $\boldsymbol{\lambda}_2(\sigma', Img_{\boldsymbol{\sigma}\circ\boldsymbol{\rho}}(q_2, T_2)) = \boldsymbol{\lambda}_2(\sigma', Img_{\boldsymbol{\sigma}\circ\boldsymbol{\rho}}(q_2', T_2)) = \boldsymbol{\lambda}_1(\sigma', Img_{\boldsymbol{\sigma}\circ\boldsymbol{\rho}}(q_1', T_1))$. It contradicts with $\boldsymbol{\lambda}_1(\sigma', Img_{\boldsymbol{\sigma}\circ\boldsymbol{\rho}}(q_1', T_1)) \neq \boldsymbol{\lambda}_2(\sigma', Img_{\boldsymbol{\sigma}\circ\boldsymbol{\rho}}(q_2, T_2))$. Therefore, $\boldsymbol{\sigma}$ is a reset sequence of $\mathcal{H}_1$ as well, and $Img_{\boldsymbol{\sigma}}(q_1, T_1) \sim Img_{\boldsymbol{\sigma}}(q_2, T_2)$ holds for any $q_1 \in Q_1$ and $q_2 \in Q_2$. By Theorem 7, it follows that $\mathcal{H}_1$ and $\mathcal{H}_2$ are alignment equivalent. ■

Thus safe-replacement equivalence imposes a stronger condition than alignment equivalence over resetable HFSMs.

Consider two resetable HFSMs $\mathcal{H}_1$ and $\mathcal{H}_2$ that are safe-replacement equivalent. During reset, the output sequences of $\mathcal{H}_1$ and $\mathcal{H}_2$ may differ even under the same reset sequence. After reset, however, their output sequences will be the same under the same input sequence. Note that, even though the output sequences of the two HFSMs may differ during reset, the underlying environment still cannot tell if one HFSM is replaced with the other. (These inconsistent output sequences are purely due to the nondeterminism at power-up. Even for the same HFSM, the output sequence during reset at this time may differ from that at next time.)

Example 23. Consider the corresponding HFSM $\mathcal{H}$ of the HSTG G of Figure 22. Suppose two copies of $\mathcal{H}$ are powered up in different states, say q_0 and q_1. (These two copies are safe-replacement equivalent.) Under the reset sequence '1,1,1', they produce output sequences '1,0,0' and '0,0,0', respectively, and both are reset to state q_2.

Safe replacement is sometimes more stringent than necessary. In some cases, delayed replacements [SPAB95] are allowed, where the new HFSM replacing the old one can be clocked for several cycles before the original reset sequence is applied. Here we do not care about input-output behavior in the first n clock cycles.

Definition 27 (Delayed HFSM). *The* n-cycle delayed HFSM *of an HFSM* $\mathcal{H} = (Q, \Sigma, \Omega, T, \boldsymbol{\lambda})$, *denoted as* $\mathcal{H}^n$, *is the same as* $\mathcal{H}$ *except for restricting its state set to* $\{q' \mid \exists\, q \in Q, \boldsymbol{\sigma} \in \Sigma^n.\ q' = Img_{\boldsymbol{\sigma}}(q, T)\}$.

The purpose of delaying an HFSM for n clock cycles is to let the HFSM get rid of some dangling states before the original reset sequence is applied.

Definition 28 (Delay Replacement). *An HFSM* $\mathcal{H}_1 = (Q_1, \Sigma, \Omega, T_1, \boldsymbol{\lambda}_1)$ *is an* n-cycle delayed safe replacement *(or simply* n-cycle delay replacement*) for* $\mathcal{H}_2 = (Q_2, \Sigma, \Omega, T_2, \boldsymbol{\lambda}_2)$ *if* ${\mathcal{H}_1}^n \sqsubseteq \mathcal{H}_2$.

It is easily seen that ${\mathcal{H}_1}^n \sqsubseteq \mathcal{H}_2$ implies ${\mathcal{H}_1}^{n+1} \sqsubseteq \mathcal{H}_2$. Moreover, delay replacement is a relaxed definition of safe replacement since a safe replacement is also a 0-cycle delayed replacement. On the other hand, we may connect delay replacement and alignment equivalence over resetable HFSMs as follows.

Proposition 10. *If HFSM* $\mathcal{H}_1$ *is an n-cycle delay replacement of a resetable HFSM* $\mathcal{H}_2$, *then* $\mathcal{H}_1$ *and* $\mathcal{H}_2$ *are alignment equivalent.*

Proof. Suppose $\boldsymbol{\sigma}_2$ is a reset sequence of $\mathcal{H}_2$. Since $\mathcal{H}_1$ is an n-cycle delay replacement of $\mathcal{H}_2$, then, for any input sequence $\boldsymbol{\sigma}_1$ with $|\boldsymbol{\sigma}_1| = n$ and for any $q_1 \in Q_1$, $Img_{\boldsymbol{\sigma}_1}(q_1, T_1)$ is in the state set Q_1^n of $\mathcal{H}_1^n$; moreover, $\mathcal{H}_1^n$ is a safe replacement of $\mathcal{H}_2$, but safe replacement preserves reset sequences by Theorem 12, and so $\boldsymbol{\sigma} = \boldsymbol{\sigma}_1 \circ \boldsymbol{\sigma}_2$ is a reset sequence for $\mathcal{H}_1$ and $\forall q_1 \in Q_1, q_2 \in Q_2.\ Img_{\boldsymbol{\sigma}}(q_1, T_1) \sim Img_{\boldsymbol{\sigma}_2}(q_2, T_2)$. However, observe that $\boldsymbol{\sigma}_1 \circ \boldsymbol{\sigma}_2$ must also be a reset sequence for

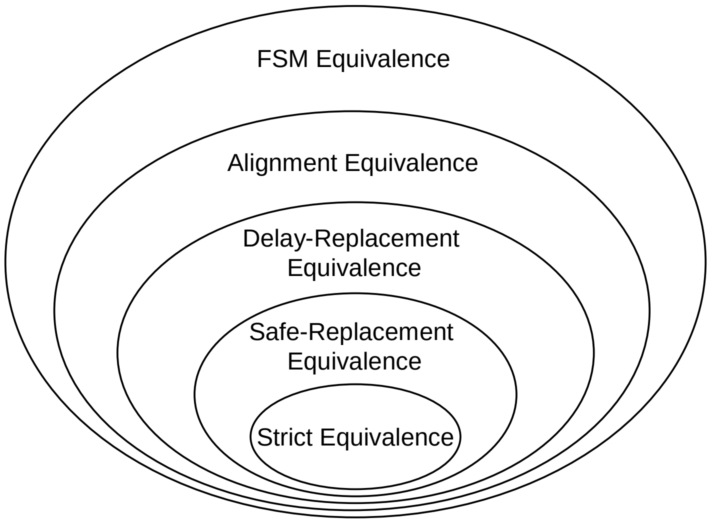


Fig. 27. The hierarchical structure formed by the inclusion relation among various notions of equivalence over resetable HFSMs.

$\mathcal{H}_2$ since the possible states of $\mathcal{H}_2$ after applying $\boldsymbol{\sigma}_1$ are a subset of Q_2 which can still be reset by $\boldsymbol{\sigma}_2$. Thus, $\forall q_1 \in Q_1, q_2 \in Q_2.\ \mathit{Img}_{\boldsymbol{\sigma}}(q_1, T_1) \sim \mathit{Img}_{\boldsymbol{\sigma}}(q_2, T_2)$, and $\mathcal{H}_1 \cong \mathcal{H}_2$. ■

Similar to the definition of safe-replacement equivalence, we may define delay-replacement equivalence as follows.

Definition 29 (Delay-Replacement Equivalence). *Two HFSMs $\mathcal{H}_1$ and $\mathcal{H}_2$ are* delay-replacement equivalent *if ${\mathcal{H}_1}^m \sqsubseteq \mathcal{H}_2$ and ${\mathcal{H}_2}^n \sqsubseteq \mathcal{H}_1$ for some positive integers m and n.*

Figure 27 summarizes the hierarchical structure among the aforementioned equivalence relations over *resetable* HFSMs, where FSM equivalence (of Definition 9) disregards any inconsistent behavior before initialization (in fact the two circuits under comparison need not be initialized in the same way) and cares only equivalence after initialization. The Euler diagram shows that, if two resetable HFSMs are equivalent under a contained relation, then they are equivalent under the containing relation. There are other notions of equivalence mainly developed in the testing community. A review can be found in [MS05].

9 Conclusions

This chapter provided the foundations of combinational and sequential hardware equivalence. Due to space reasons, interesting topics were not covered: e.g.,

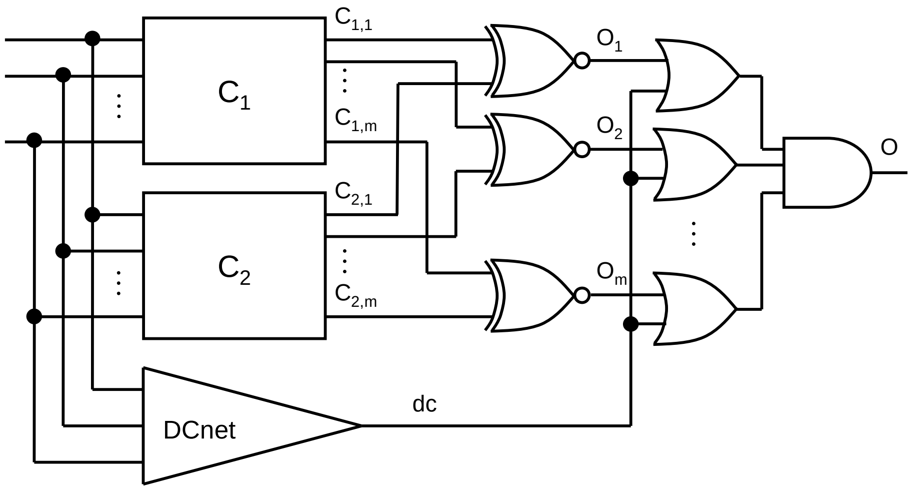


Fig. 28. Multi-output miter circuit from networks C_1 and C_2, augmented by a don't-care sequential network *DCnet*.

arithmetic circuit verification, property checking for RTL codes, verification under different levels of abstraction, don't cares, etc.

External sequential don't cares (and combinational don't cares as a special case) arise as incomplete specification that can be used for better optimization of the final circuit. They cannot be disregarded during verification to avoid false negatives, i.e., situations where a reported difference is actually due to a don't care sequence, and thus it cannot happen. Sequential don't cares have been discussed in [HCC+00], whose authors describe a software package Aquila, a sequential equivalence checker that is able to handle them. Aquila is based on an array of techniques centered around ATPG analysis. The way in which sequential don't cares are incorporated can be understood from the revised miter construction shown in Figure 28. It can be seen that the external don't-care set is represented by an additional sequential network with only one primary output *dc* (part of the specification). If an input sequence is a don't-care sequence, then the value of the output signal *dc* is 1, otherwise it is 0. For example, they mention the case of a sequential multiplier that produces one care output for every certain number of clock-cycles; e.g., the don't care network might produce the output sequence 11111110, which means that only in the eighth cycle, the output is a care output.

The basic fact is that the stuck-at-1 fault at the output O in Figure 28 is untestable if and only if signal $C_{1,k}$ is equal to signal $C_{2,k}$, $k = 1, \ldots, m$ for all care input sequences. Indeed, to enforce the output O to 0, at least one of the OR-gates should produce a 0, which requires $dc = 0$ (care sequence) and the XNOR gate to 0 (distinguishing sequence). So a sequential ATPG program can either prove the equivalence or find a distinguishing sequence for a pair of output signals.

In addition to external don't cares, there are internal don't cares that represent impossible value combinations at some state variables, corresponding to unreachable states. Provided one has a cheap way of finding them, they can be exploited to ease the task of verification by avoiding traversing the unreachable state space.

Other discussions of don't cares in combinational and sequential equivalence checking can be found in [SFVD05] and in [RRT04,RRTR03], where it is remarked that don't cares can be seen as extending equivalence checking into a problem of inclusion checking, meaning that the implementation is included in the specification if there is an assignment of the don't-care conditions that makes them equal.

An even more radical extension, under the name of *black box equivalence checking*, supposes that the specification is known, but only parts of the implementation are completed or known (black boxes being the unfinished or unknown parts), so that inequivalence is declared when the implementation differs from the specification for all possible replacements of the black boxes (see [GDDB00,JBM$^+$00] and [SB01]). Then Scholl and Becker [SB01,SB02] studied the related problem when boxes implement an incompletely specified function (complete specifications vs. incomplete implementations) and variants of it, arising in situations like design partitioning in incompletely specified blocks or use of incompletely specified intellectual property cores; their formulation is applicable also to the dual case when an incomplete specification is checked against a complete implementation. The solution is based on transforming the implementation into a circuit that models the incompleteness, but for lack of space we will not elaborate further on this interesting topic, whose generalization to model checking is still object of active research [NS04].

Notice that the effort documented in this chapter to solve hardware equivalence checking corresponds to verifying on the product machine a formula, namely,

$$\mathbf{AG}(O = 0)$$

in CTL. It is read as: for every path and at every node in the path, it is true that the output O is 0. In general the **AG** operator asserts that a formula is true for all possible evolutions in the future. As mentioned in Section 1.2, however, this is just one simple formula, out of the many that can be built in CTL or other temporal logics, using the full power of path quantifiers and temporal modalities. A discussion of general model checking requires telling another story.

References

[AGM01] P. Ashar, A. Gupta, and S. Malik. Using complete-1-distinguishability for FSM equivalence checking. *ACM Trans. Des. Autom. Electron. Syst.*, 6(4):569–590, 2001.

[BC00] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In *Proc. Formal Methods in Computer-Aided Design*, 2000.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[BCM$^+$92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[BHSV$^+$96a] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. VIS. In M. Srivas and A. Camilleri, editors, *Proc. of the Conf. on Formal Methods in Computer-Aided Design*, volume 1166 of *LNCS*, pages 248–256. Springer Verlag, November 1996.

[BHSV$^+$96b] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. In R. Alur and T. Henzinger, editors, *Proc. of the Conf. on Computer-Aided Verification*, volume 1102 of *LNCS*, pages 332–334. Springer Verlag, August 1996.

[Bra93] D. Brand. Verification of large synthesized designs. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 534–537, November 1993.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, pages 677–691, 1986.

[BS98] J. Burch and V. Singhal. Robust latch mapping for combinational equivalence checking. In *Proc. Int'l Conf. on Computer-Aided Design*, 1998.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proc. Int'l Workshop Automatic Verification Methods for Finite State Systems*, 1989.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CGMZ95] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *32nd Design Automation Conference (DAC 95)*, pages 427–432, San Francisco, CA, USA, 1995.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

[CM90] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proc. Int'l Conference on Computer-Aided Design*, pages 126–129, 1990.

[Coo71] S. Cook. The complexity of theorem-proving procedures. In *Proc. IEEE Symposium on the Foundations of Computer Science*, pages 151–158, 1971.

[Cra57] W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[DLL62] M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[ES03] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. In *Proc. Bounded Model Checking*, 2003.

[Fil91] T. Filkorn. A method for symbolic verification of synchronous circuits. In *Proc. Int'l Symp. Computer Hardware Description Languages and their Applications*, pages 249–259, 1991.

[Fil92] T. Filkorn. *Symbolische methoden für die verifikation endlicher zustandssysteme*. Ph.D. dissertation, Institut für Informatik der Technischen Universität München, 1992.

[FKS05] M. Fujita, S. Komatsu, and H. Saito. Formal verification techniques for digital systems. In H.B. Diab, S. Hassoun, and A.Y. Zomaya, editors, *Dependable Computing Systems*. J. Wiley, 2005.

[GDDB00] W. Guenther, N. Drechsler, R. Drechsler, and B. Becker. Verification of designs containing black boxes. In *EUROMICRO*, pages 100–105, 2000.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GN02] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proc. Design Automation and Test in Europe*, pages 142–149, 2002.

[Gos93] J. B. Gosling. *Simulation in the Design of Digital Electronic Systems*. Cambridge University Press, 1993.

[HCC$^+$00] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, C.-Y. Huang, and F. Brewer. Aquila: An equivalence checking system for large sequential designs. *IEEE Trans. Computers*, 49(5):443–464, 2000.

[Hop71] J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196, New York, 1971. Academic Press.

[HS66] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, 1966.

[HS97] S. Hazelhurst and C.-J. Seger. Symbolic trajectory evaluation. In T. Kropf, editor, *Formal Hardware Verification*, volume 1287 of *Lecture Notes in Computer Science*, pages 3–78. Springer, 1997.

[HWA99] H. Hulgaard, P. F. Williams, and H. R. Andersen. Equivalence checking of combinational circuits using boolean expression diagrams. *IEEE Transactions on Computer-Aided Design*, 18(7):903–917, 1999.

[Imm88] N. Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, 1988.

[JB03] J.-H. Jiang and R. Brayton. On the verification of sequential equivalence. *IEEE Trans. Computer-Aided Design*, 6:686–697, 2003.

[JB04] J.-H. Jiang and R. Brayton. Functional dependency for verification reduction. In *Proc. Int'l Conf. on Computer Aided Verification*, pages 268–280, 2004.

[JB06] J.-H. Jiang and R. Brayton. Retiming and resynthesis: A complexity perspective. *IEEE Trans. Computer-Aided Design*, 2006. to appear.

[JBM$^+$00] A. Jain, V. Boppana, R. Mukherjee, J. Jain, M. Fujita, and M. Hsiao. Testing, verification, and diagnosis in the presence of unknowns. In *VLSI Test Symposium*, pages 263–269, 2000.

[JH07] J.-H. R. Jiang and W.-L. Hung. Inductive equivalence checking under retiming and resynthesis. In *Proc. Int'l Conference on Computer-Aided Design*, pages 326–333, 2007.

[JNFSV97] J. Jain, A. Narayan, M. Fujita, and A. Sangiovanni-Vincentelli. A survey of techniques for formal verification of combinational circuits. In *The*

Proceedings of the International Conference on Computer Design, pages 445–454, 1997.

[KB02] W. Kunz and A. Biere. SAT and ATPG: Boolean engines for formal hardware verification. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 782–785, November 2002.

[KK97] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proc. Design Automation Conference*, pages 263–268, 1997.

[KMSM01] W. Kunz, J. Marques-Silva, and S. Malik. SAT and ATPG: Algorithms for boolean decision problems. In R. Brayton, S. Hassoun, and T. Sasao, editors, *Logic Synthesis and Verification*, pages 309–341. Kluwer, 2001.

[Koh78] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, 1978.

[KPKG02] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design*, 21(12):1377–1394, 2002.

[Kra97] J. Krajíček. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *Journal of Symbolic Logic*, 62(2):457–486, 1997.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[KvE01] A. Kuehlmann and C.A.J. van Eijk. Combinational and sequential equivalence checking. In R. Brayton, S. Hassoun, and T. Sasao, editors, *Logic Synthesis and Verification*, pages 343–372. Kluwer, 2001.

[Lam05] W. K. C. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches.* Prentice Hall Professional Technical Reference, 2005.

[LJHM07] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko. Scalable exploration of functional dependency by interpolation and incremental SAT solving. In *Proc. Int'l Conference on Computer-Aided Design*, pages 227–233, 2007.

[LN91] B. Lin and A. R. Newton. Implicit manipulation of equivalence classes using binary decision diagrams. In *ICCD*, pages 81–85. IEEE Computer Society, 1991.

[LS83] C. Leiserson and J. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, 1983.

[LS91] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[MCJB05] A. Mishchenko, S. Chatterjee, J.-H. R. Jiang, and R. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, Electronics Research Laboratory, University of California, Berkeley, March 2005.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[McM02] K. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proc. Computer Aided Verification*, pages 250–264, 2002.

[McM03] K. McMillan. Interpolation and SAT-based model checking. In *Proc. Computer Aided Verification*, pages 1–13, 2003.

[Mic03] A. Miczo. *Digital Logic Testing and Simulation.* J. Wiley, 2003.

[MMZM01] M. Moskewicz, C. Madigan, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference*, pages 530–535, 2001.

[MS05] M. Mneimneh and K. Sakallah. Principles of sequential-equivalence verification. *IEEE Design & Test of Computers*, pages 248–257, 2005.

[MSS99] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. on Computers*, 48(5):506–521, 1999.

[NS04] T. Nopper and C. Scholl. Approximate symbolic model checking for incomplete designs. In *FMCAD*, pages 290–305, 2004.

[PG86] D. Plaisted and S. Greenbaum. A structure preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[PHS94] B. Plessier, G. Hachtel, and F. Somenzi. Extended BDDs: trading off canonicity for structure in verification algorithms. *Form. Methods Syst. Des.*, 4(2):167–185, 1994.

[Pil98] L. Pillage. *Electronic Circuit and System Simulation Methods (SRE)*. McGraw-Hill, 1998.

[PJH94] C. Pixley, S. Jeong, and G. Hatchel. Exact calculation of synchronization sequences based on binary decision diagrams. *IEEE Trans. Computer-Aided Design*, 13:1024–1034, 1994.

[PT87] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[Pud97] P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62(2):981–998, 1997.

[QCC$^+$00] S. Quer, G. Cabodi, P. Camurati, L. Lavagno, E. Sentovich, and R. K. Brayton. Verification of similar FSMs by mixing incremental re-encoding, reachability analysis, and combinational checks. *Formal Methods in System Design*, 17(2):107–134, 2000.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[RRT04] S. Rahim, B. Rouzeyre, and L. Torres. A flip-flop matching engine to verify sequential optimizations. *Computing and Informatics*, 23(5-6):437–460, 2004.

[RRTR03] S. Rahim, B. Rouzeyre, L. Torres, and J. Rampon. Matching in the presence of don't cares and redundant sequential elements for sequential equivalence checking. In *HLDVT '03: Proceedings of the Eighth IEEE International Workshop on High-Level Design Validation and Test Workshop*, page 129, Washington, DC, USA, 2003. IEEE Computer Society.

[Rud93] R. Rudell. Dynamic variable ordering for binary decision diagrams. In *Proc. Int'l Conference on Computer-Aided Design*, pages 42–47, 1993.

[Sav70] W. J. Savitch. Relationships between nondeterministic and deterministic space complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[SB01] C. Scholl and B. Becker. Checking equivalence for partial implementations. In *DAC*, pages 238–243, 2001.

[SB02] C. Scholl and B. Becker. Checking equivalence for circuits containing incompletely specified boxes. In *ICCD*, pages 56–63, 2002.

[SB04] V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *Int'l Journal on Software Tools for Technology Transfer (STTT)*, 5(2-3), 2004. Springer.

[SFVD05] S. Safarpour, G. Fey, A. Veneris, and R. Drechsler. Utilizing don't care states in SAT-based bounded sequential problems. In *GLSVSLI '05: Proceedings of the 15th ACM Great Lakes symposium on VLSI*, pages 264–269, New York, NY, USA, 2005. ACM Press.

[SM73] L. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *Proc. ACM Symposium on the Theory of Computing*, pages 1–9, 1973.

[SP94] V. Singhal and C. Pixley. The verification problem for safe replaceability. In *Proc. Computer Aided Verification*, pages 311–323, 1994.

[SPAB95] V. Singhal, C. Pixley, A. Aziz, and R. Brayton. Exploiting power-up delay for sequential optimization. In *Proc. European Design Automation Conference*, pages 54–59, 1995.

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proc. Formal Methods in Computer-Aided Design*, 2000.

[Sze88] R. Szelepcsenyi. The method of forced enumeration for nondeterministic automata. *Acta Inf.*, 26(3):279–284, 1988.

[Tam93] T. Tamisier. Computing the observable equivalence relation of a finite state machine. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 184–187, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[Tar55] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tse70] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, pages 466–483, 1970.

[TSL$^+$90] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using BDDs. In *Proc. Int'l Conference on Computer-Aided Design*, pages 130–133, 1990.

[vE00] C. A. J. van Eijk. Sequential equivalence checking based on structural similarities. *IEEE Trans. Computer-Aided Design*, 19(7):814–819, 2000.

[vEJ95] C. A. J. van Eijk and J. A. G. Jess. Detection of equivalent state variables in finite state machine verification. In *Proc. Int'l Workshop on Logic Synthesis*, 1995.

[WKS01] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proc. Design Automation Conference*, pages 542–545, 2001.

[Zha97] H. Zhang. SATO: An efficient propositional prover. In *Proc. Int'l Conf. on Automated Deduction*, pages 272–275, 1997.

[ZKP00] B. P. Zeigler, T. G. Kim, and H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, 2000.