

Formal Hardware Verification Methods: A Survey

AARTI GUPTA

School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, Pennsylvania 15213.

Abstract. Growing advances in VLSI technology have led to an increased level of complexity in current hardware systems. Late detection of design errors typically results in higher costs due to the associated time delay as well as loss of production. Thus it is important that hardware designs be free of errors. Formal verification has become an increasingly important technique towards establishing the correctness of hardware designs. In this article we survey the research that has been done in this area, with an emphasis on more recent trends. We present a classification framework for the various methods, based on the forms of the specification, the implementation, and the proof method. This framework enables us to better highlight the relationships and interactions between seemingly different approaches.

Keywords: formal verification, hardware verification, specification, design correctness, proof methods

1. Introduction

Technological advances in the areas of design and fabrication have made hardware systems much larger today than before. As faster, physically smaller and higher functionality circuits are designed, in large part due to progress made in VLSI, their complexity continues to grow. Simulation has traditionally been used to check for correct operation of such systems, since it has long become impossible to reason about them informally. However, even this is now proving to be inadequate due to computational demands of the task involved. It is not practically feasible to simulate all possible input patterns to verify a hardware design. An alternative to post-design verification is the use of automated synthesis techniques supporting a correct-by-construction design style. Logic synthesis techniques have been fairly successful in automating the low-level (gate-level) logic design of hardware systems. However, more progress is needed to automate the design process at the higher levels in order to produce designs of the same quality as is achievable today by hand. Until such time as synthesis technology matures, high-level design of circuits will continue to be done manually, thus making post-design verification essential.

Typically, a much reduced subset of the exhaustive set of patterns is simulated after the design of a system. It is hoped that no bugs have been overlooked in this process. Unfortunately, this is not always the case in practice. Numerous instances exist of cases where errors have been discovered too late in the design cycle, sometimes even after the commercial production and marketing of a

product. Such late detection of errors is a very costly proposition—not only in terms of lost time to market a product, but also in terms of a potential loss of production in case of product recall. These are compelling reasons for verifying hardware to be correct, and completely correct, right at the design stage.

A comparatively recent alternative to simulation has been the use of formal verification for determining hardware correctness. Formal verification is, in some sense, like a mathematical proof. Just as correctness of a mathematically proven theorem holds regardless of the particular values that it is applied to, correctness of a formally verified hardware design holds regardless of its input values. Thus, consideration of *all cases* is implicit in a methodology for formal verification. In addition to being theoretically sound, these methods have been demonstrated to work reasonably well in practice too. Their success has attracted a fair deal of attention from both the research community and the industry, with exciting progress being made on many fronts.

1.1. What is formal hardware verification?

We consider a formal hardware verification problem to consist of *formally establishing that an implementation satisfies a specification*. The term *implementation* (*Imp*) refers to the hardware design that is to be verified. This entity can correspond to a design description at any level of the hardware abstraction hierarchy, not just the final physical layout (as is traditionally regarded in some areas). The term *specification* (*Spec*) refers to the property with respect to which correctness is to be determined. It can be expressed in a variety of ways—as a behavioral description, an abstracted structural description, a timing requirement, etc. For the purpose of this survey, both entities—the implementation and the specification—are regarded as given within the scope of any one problem, and it is required to formally prove the appropriate “satisfaction” relation.

In particular, we do not address directly the problem of *specification validation*, i.e. whether the specification means what it is intended to mean, whether it really expresses the property one desires to verify, whether it completely characterizes correct operation, etc. It can be indirectly cast in terms of the formal verification framework described above in the following sense—a specification for a particular verification problem can itself be made the object of scrutiny, by serving as an implementation for another verification problem at a conceptually higher level. The purpose of the latter problem is to test if the meaning of the original specification is as intended, the “intended” meaning thus serving as a specification at the higher level. Note that this does imply a leap of faith at some level where specifications are held to be infallible, a necessary characteristic of any mechanism for formal representation. Similarly, at the lowest end as well, we do not specifically address the problem of *model validation*, i.e. whether the model used to represent the implementation is consistent, valid, correct etc. It is obvious that the quality of verification can only be as good as quality of the models

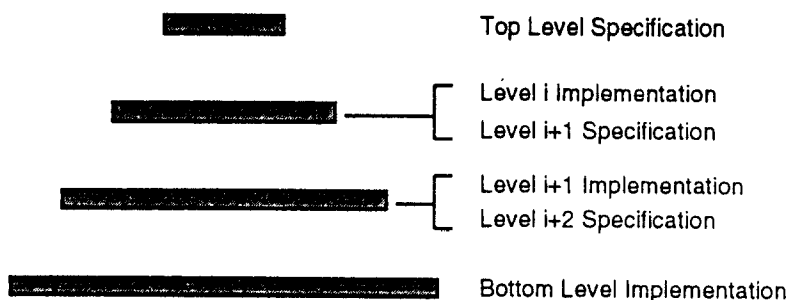


Figure 1. Hierarchical verification.

used. On the other hand, models are essentially abstracted representations, and should be kept simple for efficiency reasons. A compromise between quality and simplicity is therefore necessary in order to make models practically useful. We shall highlight examples of such compromises in our descriptions of the research work in this area.

An important feature of the above formulation is that it admits hierarchical verification corresponding to successive levels of the hardware abstraction hierarchy. Typically, the design of a hardware system is organized at different levels of abstraction, the topmost level representing the most abstract view of the system and the bottommost being the least abstract, usually consisting of actual layouts. Verification tasks can also be organized naturally at these same levels. An implementation description for a task at any given level serves also as a statement of the specification for a task at the next lower level, as shown in Figure 1. In this manner, top-level specifications can be successively implemented and verified at each level, thus leading to implementation of an overall verified system. Hierarchical organization not only makes this verification process natural, it also makes the task tractable. Dealing with the complexity of a complete system description of even modest size, by standards today, is out of bounds for most verification techniques. By breaking this large problem into smaller pieces that can be handled individually, the verification problem is made more manageable. It effectively increases the range of circuit sizes that can be handled in practice.

1.2. Expected contributions

Other survey articles have been written on the subject of formal hardware verification. A useful early reference is that presented by Camurati and Prinetto [1]. A recent survey-tutorial has been presented in book form by Yoeli [4], which includes several landmark papers published in this area. Subareas within this field

have also been the subject of other surveys—a classic survey on application of temporal logic to the specification and verification of reactive systems has been presented by Pnueli [3], and another for automatic verification of finite-state controllers has been presented by Grumberg and Clarke [2].

Formal hardware verification enjoys a special place within the research community today, as it has brought about a synthesis of the engineering methods on one hand and the theoretical, formal methods on the other. In our survey, we make an attempt to present a comprehensive picture of the various approaches that researchers with seemingly different biases have explored. We discuss important design issues relevant to a hardware verification methodology in general, and evaluate these for particular approaches. The emphasis is on the underlying theory rather than the implementation details of each method, focusing on how it relates to the basic formulation of a verification problem (in terms of a specification, an implementation, and their relationship; as described in the previous section). We also present a classification framework that highlights the similarities and differences between various approaches. Finally, with the body of research in this field already growing at an amazing pace, we hope that our survey will serve as a useful source of pointers into the vast literature available. For convenience, references in the bibliography section have been grouped subjectwise (along with related background references).

1.3. Organization of the article

In Section 2 of this article, we discuss design issues pertaining to a formal verification methodology. We highlight different dimensions of the basic verification problem in order to motivate criteria that we will use subsequently for our classification. This is followed by brief notes on the notation used thereafter.

Section 3 consists of summary descriptions of various approaches, organized under three subsections—logic, automata/language theory, and hybrid formalisms—according to the formalism used by each approach for representing the specifications.

The subsection on logic starts with a brief review of the logic terminology commonly used. We then describe approaches that represent specifications as statements in a particular kind of logic, with an associated set of syntactic and semantic rules. The logics we cover under this subsection are (in sequential order of presentation):

- 3.1.1 first-order predicate logic and propositional logic,
- 3.1.2 a specialized form of first-order logic called Boyer–Moore Logic,
- 3.1.3 higher-order logic,
- 3.1.4 temporal logic,
- 3.1.5 extended temporal logic,
- 3.1.6 mu-calculus, and

3.1.7 functional approaches and special calculi.

(Though the last category does not strictly belong under “logic”, it has been included because of syntactic similarities.)

The subsection on automata/language theory deals with approaches that represent specifications as automata, languages, machines, trace structures, etc. Verification proceeds typically in the form of checking for:

- 3.2.1 machine equivalence,
- 3.2.2 language containment, and
- 3.2.3 trace conformation.

The subsection on hybrid formalisms includes approaches that use the relationship between logics and automata/language theory to convert specifications expressed in the former to those in the latter. Specifically, we describe approaches that use the relationship between:

- 3.3.1 linear time temporal logic and finite-state automata, and
- 3.3.2 temporal logics (various kinds) and Büchi automata.

We end Section 3 by giving pointers to major results regarding the relative expressiveness of various formalisms discussed in this section.

In Section 4 we present a global view of the work described in the previous section. Details are suppressed and the important features highlighted by presenting a classification framework for research done in this area. The focus is on underlying similarities and differences between various approaches.

In Section 5 we point out some recent trends and future directions that are likely to gain in importance as more progress is made.

Finally, our conclusions are presented in Section 6.

2. Formal verification methodology: Design issues

2.1. Problem dimensions

Since the idea of using formal methods for verifying hardware was first introduced, researchers have explored numerous approaches to the problem. Before we describe these and assess their similarities and differences, it is instructive to consider various facets of the problem itself. A typical verification problem consists of formally establishing a relationship between an implementation and a specification. The fact that this reasoning has to be formal requires that some kind of formalism be used to express all three entities—implementation, specification, and the relationship between them. We consider each of these entities separately in this section and discuss relevant design issues.

2.1.1. Implementation. An implementation consists of a description of the actual hardware design that is to be verified. Several alternatives for a formal representation have been explored by researchers, e.g. a network of transistors/gates, finite-state automata/machines, descriptions in logic, etc. The exact choice usually depends on what relevant aspects of hardware one wishes to model. In any case, it is of great benefit to use a representation language that makes explicit the syntax and semantics of the particular hardware abstractions employed. One of the most important questions in this regard is this: What levels of the hardware abstraction hierarchy can the chosen representation model? In other words, can it represent hardware at the switch level, gate level, register-transfer level (at all levels)? Associated with each of these levels are physical/conceptual phenomena about which decisions must be made, whether or not the representation language should provide mechanisms to model them.

These are:

- circuit level—can it model analog behavior, hazard and metastable conditions, varying electrical parameters, different logic families?
- switch level—can it model signal strengths, threshold effects, bidirectionality, short circuits, propagation delays?
- gate level—can it model composition, hierarchy, parameterization, gate delays?
- register-transfer level—can it model nondeterminism, concurrency (if so, how does it handle communication/synchronization), infinite computations, composition, hierarchy?

Not surprisingly, these choices determine to a large extent the class of circuits for which a given approach is applicable. For example, an approach that uses a pure binary switch-level model may not be able to catch errors that result from analog effects like charge-sharing, threshold drops, etc. In general, this can compromise the validity of the verification results obtained. Seen from the application end, some of the interesting classes (not mutually exclusive) of circuits that one might wish to verify are:

- combinational/sequential
- synchronous/asynchronous
(asynchronous circuits may be delay-sensitive/delay-insensitive/speed-independent)
- finite-state automata/machines (with finite/infinite computations)
- pipelined hardware
- parameterized hardware (e.g. systolic architectures)

An implementation representation affects not only the class of applications for an approach, but also its performance limits. For example, a representation that does not support hierarchical descriptions would tend to run out of steam quickly on large modular hardware designs. With most approaches, the combinational

explosion in the size of a description, which results from a flat, non-hierarchical style, limits the scale of hardware that can be verified in practice.

2.1.2. Specification. A specification is a description of the intended/required behavior of a hardware design. Various formalisms have been used to represent specifications. The popular ones can be broadly classified as follows:

- logic—propositional logic, first-order predicate logic, higher-order logic, modal logic (e.g. temporal logic, extended temporal logic), mu-calculus, and
- automata/language theory—finite-state automata on finite words, finite-state automata on infinite words, trace structures, etc.

From the point of view of verification, a variety of correctness properties are often used:

- functional correctness properties—e.g. a given circuit should exhibit the functionality of an adder
- safety (invariant) and liveness properties—e.g. in a mutual exclusion system with two processes *A* and *B*,
 - safety property: simultaneous access will never be granted to both *A* and *B*
 - liveness property: if *A* wants to enter its critical section, it will eventually do so
- timing properties—e.g. access to a process will be granted within five seconds of placing a request with an arbiter

For each type of correctness property, it is often the case that some formalisms are more suitable for specification than others. For example, for specification of liveness properties, a logic that reasons explicitly about time (e.g. temporal logic) is more suitable than a logic that does not provide any special facilities for doing so (e.g. first-order predicate logic). A related issue regards the expressiveness of the formalism, i.e. what properties can a given formalism express? After all, if the desired property cannot even be represented notationally, it can certainly not be verified. For example, as will be described later, temporal logic *cannot* express the requirement that a given condition hold on every other state of a computation sequence.

Another design issue regarding the specification formalism is this: What kind of abstractions can a formalism express? Abstractions are used to suppress irrelevant detail in order to focus on objects of interest, and they form an essential part of any modeling paradigm. Within the specific context of hardware verification, we have already described a hierarchical methodology based on different levels of the hardware abstraction hierarchy. Each level of this hierarchy is related through appropriate abstractions to the next. By using abstraction as a form of specification, i.e. by using specifications to represent a valid abstract view

of the implementation, a natural way to decompose the overall task (of system verification) is made available. Thus, apart from the simplicity they afford, abstractions are necessary to cope with the complexity of problems in practice. Several kinds of abstraction mechanisms have been found useful for the purpose of specification [59, 61]. Some of these are as follows:

- *Structural abstraction* involves suppression of internal structural details and provides an externally observable view of the implementation.
- *Behavioral abstraction* involves partial specification of behavior leaving some cases undefined; it is useful for postponing development of finer specifications to the next level of detail.
- *Data abstraction* provides an abstract view of the data used by the implementation and is useful for specifying functional properties more naturally.
- *Temporal abstraction* relates the time scales of the implementation and the specification; it is useful for reasoning about time-dependent behaviors.

The practical usefulness of a specification formalism is enhanced by its ability to provide easy formulation of these abstraction mechanisms.

2.1.3. Relationship between implementation and specification. Formal verification involves furnishing a proof that an implementation “satisfies” a specification. This notion of satisfaction also has to be formalized, typically in the form of requiring that a certain formal relationship hold between the descriptions of the implementation and the specification. Various notions have been used by researchers, the semantics for each of these ensuring that the intended satisfaction relation is met. Some of the commonly encountered forms of proof methods used for establishing the formal relationship are as follows:

- *Theorem-proving.* The relationship between a specification and an implementation is regarded as a theorem in logic, to be proved within the context of a proof calculus, where the implementation provides axioms and assumptions that the proof can draw upon (described in Section 3.1).
- *Model checking.* The specification is in the form of a logic formula, the truth of which is determined with respect to a semantic model provided by an implementation (described in Section 3.1).
- *Equivalence checking.* The equivalence of a specification and an implementation is checked, e.g. equivalence of functions, equivalence of finite-state automata, etc.
- *Language containment.* The language representing an implementation is shown to be contained in the language representing a specification.

No matter what form of proof method is employed, several criteria need to be evaluated in order to make meaningful comparisons between various approaches. The important amongst these are:

- the nature of the relationship explored
 - $Imp \equiv Spec$ (implementation is equivalent to specification)
 - $Imp \Rightarrow Spec$ (implementation logically implies specification)
 - $Imp \models Spec$ (implementation provides a semantic model with respect to which specification is true)
- soundness of the proof method (every statement that is provable is logically true; see Section 3.1 for details), and completeness of the proof method (every statement that is logically true is provable; see Section 3.1 for details)
- the degree of automation—whether the proof generation process is automatic, semi-automatic, or uses reasoning by hand (a higher degree of automation directly improves the scalability of an approach)
- the computational complexity, in cases where an algorithm is available (a low computational complexity indicates better scalability)
- whether the proof methodology can handle
 - compositional proofs: proofs for a large module can be constructed syntactically from proofs of its component parts
 - hierarchical proofs: proofs for an entire system can be organized hierarchically at various levels of abstraction
 - induction proofs: proofs can reason about parameterized designs that are described inductively

2.2. Classifying criteria

It is fairly clear that there are multiple dimensions to a formal hardware verification method. With the wide spectrum of choices available in the design space, it is no wonder that there exist a variety of approaches pursued by different researchers. In order to understand these better, we would like to select a dimension that facilitates a good exposition of the other features also. The implementation representation, the specification representation, and the form of proof method are all good candidates for forming the basis of a classification. Of these, we feel that the *specification formalism* used to represent a specification provides a good discrimination criterion between different approaches. The implications of a particular choice for the specification formalism are reflected both in the implementation representation chosen and the form of proof method employed. (We are not in any way suggesting that this is the *first* choice made when designing a verification approach, only that it affects to a large extent the forms of the other two). In Section 3, we describe various approaches as they differ along this dimension, therefore providing a natural (linear) order to our presentation.

We also feel that any attempt at providing a classification would necessarily have to draw upon all three criteria mentioned above. We present a framework

that highlights these features in Section 4 of this paper. Various combinations along these axes are illustrative of the interactions and relationships that exist between different approaches.

2.3. Other practical issues

Apart from theoretical issues, e.g. the computational complexity and the soundness/completeness of an approach, we address (where appropriate) some practical issues that are important in typical applications:

- What level of representation (of both an implementation and a specification) gives a good compromise between expressiveness and efficiency?
- Is it possible to introduce inconsistencies in the proof system and thereby violate validity claims?
- At what level can/should a proof technique get automated?
- Does the approach admit executable specifications? (Executability allows a specification to be simulated in order to “verify” that it means what it is intended to mean.)
- What kind of help, if any, is available for diagnosis and rectification?
- What kind of help, if any, is available for design revision and modification?

2.4. Notation

We use the following notation through rest of this article:

- Boolean connectives: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\equiv$ represent negation, conjunction, disjunction, implication, and equivalence, respectively
- Quantifiers: $\exists$, and $\forall$ represent existential and universal quantification, respectively
- Set connectives: $\subset$, $\cup$, and $\cap$ represent subset relation, set union, and set intersection, respectively
- Formulas: denoted by ϕ , ψ
- States: denoted by s_0 , s_1 , $s_2 \dots$
Set of states: denoted by S
Sequence of states: denoted by σ

3. Description of hardware verification techniques

In this section we present summary descriptions of hardware verification methods explored by various researchers. As mentioned in the previous section, we have chosen the specification formalism as the discriminating criterion for our

presentation. Different formalisms are broadly categorized under logic, automata/language theory, and hybrid formalisms. In the first case, specifications are expressed as statements in a particular kind of logic. In the second case, a specification is represented in the form of a language (or equivalently an automaton). Finally, the third case uses a hybrid approach, where a specification initially expressed as a logic statement is converted to an equivalent language/automaton representation.

In the following subsections we describe approaches that use each of these formalisms, dividing each further by subareas as outlined earlier. Due to extensive literature available in the area of formal hardware verification, we cannot possibly do justice to the fine details of every approach. We therefore focus on a typical example approach for each subarea and discuss its formulation in terms of the basic verification problem (as described in Section 1.2). Where appropriate, we give details of the syntactic and semantic terminology used and supplement it with published examples. (For ease of reference, we have tried to follow closely the author's description and terminology.) We discuss complexity, soundness/completeness, and other design issues for the approach, and we then highlight its strengths and limitations. This is followed by brief comments on and pointers to related work in the area.

3.1. Logic

Logic is generally regarded as the study of principles of reasoning and is a subject of study in various disciplines such as philosophy, mathematics, and computer science. In this article, we consider *formal* logic, in that a *formal language* and a *formal system* are used to represent logical truths and logical methods of reasoning. We briefly review the terminology and notation that is commonly used in standard texts on logic [8].

A formal language is identified with a set of *well-formed-formulas* (hereafter referred to as formulas) specified syntactically by a set of symbols (called an alphabet) and a set of rules for the formation of formulas. A formal system consists of a formal language and a deductive apparatus, the latter consisting of a set of formulas called *axioms* and a set of transformation rules for formulas called *inference rules*. The underlying idea is to represent a proof in the formal system as a sequence of formulas, each of which is either an axiom or is derived by application of an inference rule to formulas appearing previously in the sequence, the last formula of the sequence typically referred to as the *theorem* being proved. Note that a proof satisfies purely syntactic requirements, with no regard to the meanings of the formulas.

The semantics of a formal language is specified through an *interpretation* that assigns meanings to its symbols and formulas. It typically consists of a (non-empty) domain of discourse and a mapping of the various symbols to appropriate entities with respect to the domain. (For example, if the alphabet

consists of constant/variable/function/predicate symbols, it maps constant and variable symbols to elements of the domain, function symbols to functions on the domain, and predicate symbols to relations on the domain.) Model theory deals with the theory of these interpretations. In a formal language representation of logic, formulas are typically assigned a value true/false, and a *model* of a formula typically refers to an interpretation that makes the formula true (denoted “ $\text{model} \models \text{formula}$ ”). Formulas that do not have variables (called closed formulas) are interpreted with respect to a structure (interpretation minus the mapping for variables). The set of true closed formulas with respect to a given structure is often referred to as the *theory* of that structure.

By appropriate representation of logical axioms and logical rules of inference, logical truths can be proved as theorems in a particular formal system. The metatheory of such systems deals with issues such as *soundness* and *completeness*. In general, soundness of a logic system means that all formulas that are provable within the proof system (theorems) are logically (semantically) true. Completeness deals with the other direction, i.e. it means that all logically true formulas are provable as theorems within the logic system. A logic system is almost worthless if it is not sound (false formulas can then be proved). On the other hand, logic systems can be useful in practice even when not complete, by yielding proofs of a useful subset of the set of logically true formulas.

Different syntactic and semantic rules associated with formal languages and systems lead to different kinds of logics. A variety of these have been studied by logicians, with a wide range in expressiveness of the language traded off against simplicity of the proof methods. We refer the interested reader to convenient handbooks for theoretical details [5, 6]. In the remainder of this section, we describe the application of some of these logics to the task of hardware verification. Typically, a specification is represented as a formula in the logic being used. The implementation is represented either as a formula or as a semantic model, as follows:

1. In the former case, verification takes the form of theorem-proving, i.e. the required relationship (logical equivalence/logical implication) between the formulas representing the implementation and the specification, is regarded as a theorem to be proved using the associated deductive apparatus. The abstractions used to model hardware typically provide axioms that such a proof can draw upon.
2. In the latter case, both theorem-proving and model checking can be used. For theorem-proving, the semantic model representing the implementation provides additional axioms that can be used to prove the truth of the specification formula. Model checking, on the other hand, deals directly with the semantic relationship and shows that the implementation is a model for the formula that represents the specification. (We compare theorem-proving and model checking techniques in Section 5.1.)

3.1.1. First-order predicate logic. First-order predicate logic is one of the most extensively studied logics and has found numerous applications, especially in the study of the foundations of mathematics [7, 9]. Its language alphabet consists of a signature (consisting of countable sets of symbols for constants, functions, and predicates), symbols for variables, and a set of standard Boolean connectives ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\equiv$) and quantifiers ($\exists$, $\forall$). There are two main syntactic categories—terms and formulas. Terms consist of constants, variables, and function applications to argument terms. Formulas consist of atomic formulas (predicates), Boolean combinations of component formulas, and quantified formulas (with quantification allowed on variables only). An interpretation for a first-order logic consists of a structure (a domain of discourse and appropriate mappings of the signature symbols) and an assignment for the variables (mapped to domain elements). Semantically, terms denote elements in the domain, and formulas are interpreted as true/false. Different first-order languages are obtained depending on the exact set of signature symbols used and their interpretations. Various proof systems have been studied for first-order logics and have been shown to be both sound and complete [6]. Propositional logic can be regarded as a restriction of first-order logic to a Boolean domain (“True”, “False”), thereby making the quantifiers, function, and predicate symbols unnecessary; however, quantification does facilitate concise expression. Tautology-checking, used to ascertain the truth of arbitrary propositional formulas, is sound and complete and is of NP-hard complexity [20].

Since hardware systems deal mostly with Boolean-valued signals, it was natural for early researchers to use propositional logic to model the behavior of digital devices. Due to the underlying assumption of zero-time delay of Boolean gates, this approach works well only for functional specification of combinational circuits, and is inadequate for reasoning about general hardware, e.g. sequential circuits [23]. The next natural choice was first-order predicate logic. Also, success in the area of software verification of Floyd–Hoare assertional methods [19, 21] (which typically use first-order predicate assertions) encouraged use of similar methods for verifying hardware. In this section, we describe approaches that have used first-order predicate logic, or some restricted subset of it, for expressing specifications. Most of these draw upon the Hoare-style verification techniques for software, some combining them with traditional simulation techniques for hardware. (Note: For ease of presentation, we have included propositional approaches also in this section.)

3.1.1.1. Applications of software verification techniques. Darringer gave examples of verifying microprogram implementations and digital logic [18] using the notion of a *simulation relation* borrowed from the area of program verification [24]. In this approach, program descriptions of both specification and implementation are interspersed with control points, which are then associated with parameterized assertions expressed in first-order predicate logic. A simulation relation specifies the correspondence between states and the associated control points. In order to

show the two program descriptions to be equivalent, it is proved that if they are started in corresponding states, then they will continue to stay in corresponding states. This proof of equivalence is carried out by a comparison of the two symbolic execution trees using a simplifier and a theorem-prover. Such a method works well for systems with a well-defined concept of corresponding states and where it is easy to provide a simulation relation. Also, for complete automation, additional theories may be required to simplify the symbolic expressions.

Another approach proposed by Shostak [26] is an adaptation of Floyd's assertional method for sequential program verification [19]. A circuit graph is used to represent a hardware circuit by associating a node with each circuit element, with directed edges representing connections between them. Dangling edges represent overall circuit inputs and outputs. The behavior of each circuit element is modeled by a *transfer predicate* that describes the relationship between its inputs and outputs as a function of time. This circuit graph is annotated with predicates much as a program graph is. A complete specification consists of input, output, and initial-condition specifications for the different kinds of edges. Correctness is taken to mean that if the circuit inputs satisfy the input specifications, if the initial-condition assertions hold, and if each circuit element operates according to its transfer predicate, then the circuit outputs are guaranteed to satisfy the output specification. The proof of correctness employs simultaneous induction over time and over structure of the circuit graph. As in Floyd's method, this proof hinges on *loop assertions (invariants)* that cut circuit cycles. Verification conditions are proved with the help of a mechanical theorem-prover called STP. Apart from dealing with circuits, this approach is applicable to arbitrary concurrent systems, e.g. processors in a distributed network.

A drawback of this approach is the detailed level of the specifications required. The assertions are tied in with the hardware representation (the circuit graph) and do not provide any abstracted view of the system. Also, the use of functions to represent input-output behavior leads to restricted modeling ability, e.g. this approach cannot easily model effects like bidirectional switch behavior. Finally, hand-tuning is needed to guide effectively the semi-automated theorem-prover through various steps of the proof.

3.1.1.2. Applications of hardware simulation techniques. Bryant has used a logic simulation methodology for formal verification of sequential circuits [14]. Both the circuit (implementation) and the specification are represented as finite-state machines. Their input-output equivalence is verified by observing the output of a simulator that models the behavior of the circuit machine.

Given a general sequential machine, the problem of "machine identification", i.e. establishing equivalence to a known machine, was studied by Moore [25]. He proved that the behavior of a sequential system cannot be characterized by simply observing its response to a set of stimuli, unless additional information about it, e.g. an upper bound on the number of states, is known. Bryant's methodology does not assume an upper bound on the number of circuit states,

but uses an enhanced simulator with three-valued logic modeling. The three-valued logic used consists of 1, 0, and a third state X that denotes an unknown or indeterminate value.

Bryant proves that a “black-box” simulation (i.e. one in which only the input-output behavior of a simulated circuit can be observed) can verify only *definite* systems. (A system is definite if for some constant k , the output of the system at any time depends only on the last k inputs.) In order to verify more general sequential systems, it is necessary to describe the transition behavior of the implementation and to relate its internal states to the specification. The states of the implementation automaton are encoded using Boolean variables, and next-state and output functions are described as Boolean functions over these variables. Floyd-Hoare-style circuit assertions (in the form of pre-conditions and post-conditions) are generated manually to cover the transition behavior of the specification automaton. These assertions are then verified for the circuit using the three-valued logic simulator. A circuit is said to satisfy an assertion if the post-conditions on state and output variables (expressed as Boolean formulas over these variables) are true for all transitions corresponding to states and inputs that satisfy the pre-conditions.

Since the circuit verification problem is NP-hard in general, several techniques are proposed by Bryant to make the above approach attractive in practice. One such technique is to utilize X to indicate a “don’t-care” condition. Since Bryant’s simulator is monotonic with respect to X , (i.e. if X ’s in an input pattern result in a 0 or a 1 on a circuit node, then the same result would occur if the X ’s were replaced by 0’s or 1’s) the effects of a number of Boolean input sequences can be simulated with a single ternary input sequence. An illustrative example of this technique, and the methodology outlined above, is provided by Bryant for the verification of an N -bit RAM by simulating just $O(N \log N)$ patterns [16]. The assertions are expressed in a restricted form of propositional logic and the circuit representation is derived by a switch-level simulator called COSMOS [17]. (COSMOS uses canonical Boolean function representations called Binary Decision Diagrams (BDDs) [12], and efficient algorithms for symbolic analysis of circuits [13, 15].)

Another technique suggested by Bryant is *symbolic simulation*. In symbolic simulation, the input patterns are allowed to contain Boolean variables in addition to the constants (0, 1, and X). Efficient symbolic manipulation techniques for Boolean functions allow multiple input patterns to be simulated in one step, potentially leading to much better results than can be obtained with conventional exhaustive simulation. An approach that uses this technique was presented by Bose and Fisher [11]. They describe a symbolic simulation method for verifying synchronous pipelined circuits based on Hoare-style verification [21]. To deal with the conceptual complexity associated with pipelined designs, they suggest the use of an *abstraction function*, originally introduced by Hoare to work with abstract data types¹ [22]. Given a state of the pipelined machine, the abstraction function maps it to an abstract unpipelined state. Behavioral specifications for

this abstract state space are given in terms of pre- and post-conditions, expressed in propositional logic. By choosing the same domain for the abstract states as the circuit value domain, they are able to automate both the evaluation of the abstraction function as well as verification of the behavioral assertions. Their technique is demonstrated on a CMOS implementation of a systolic stack. The actual circuit provides inputs to the “abstraction circuit”, and the assertions are verified at the abstract state level by a symbolic simulator (COSMOS with minor extensions). In the style of program verification, this involves introduction of and reasoning with an invariant.

Beatty, Bryant, and Seger have also used symbolic simulation for Hoare-style verification, but in a different manner from Bose and Fisher. Instead of using an abstraction function, their technique uses a representation function, mapping an abstract system state to an internal circuit state as modeled by COSMOS [10]. A complete specification consists of a high-level functionality specification, a description of the circuit’s interface to its environment, and a representation mapping. The functionality specification for the abstract system state is in the form of parameterized assertions consisting of pre- and post-conditions (restricted to conjunctions of simple predicates). The circuit itself consists of a transistor-level description. Verification of the mapped assertions (using the representation function) is accomplished at the switch level, again, by COSMOS. Examples of their technique include verification of a moving-data and a stationary-data stack.

3.1.1.3. Strengths and limitations. The simulation approaches described above derive their main strengths as well as weaknesses from COSMOS, the underlying simulator. COSMOS employs a powerful circuit model capable of handling low-level circuit effects like charge-sharing, ratioed transistor sizes, bidirectional pass transistors, and precharged logic (though it does not do very well on some tricky circuits that involve transient charge-sharing). The ternary logic (with don’t-care value X) used to model node values is very useful in handling multiple input patterns during logic simulation. The monotonicity requirement ensures that the results of such simulations are valid in case of actual 0/1 values in place of X ’s. Another major strength of the simulator lies in its efficient symbolic Boolean manipulation techniques, which can be used for symbolic simulations. By representing the node values as Boolean functions of symbolic Boolean variables, circuit behavior can be efficiently verified against expected behavior (specified in terms of functions of the same set of symbolic variables).

One of the major limitations of COSMOS, as of most simulators, is its inherent non-hierarchical nature. Essentially, the complete circuit needs to be analyzed in order to reason about a property, however localized that property may be. Also, no conclusion can be drawn regarding correctness of a circuit composed from verifiably correct components. A limited amount of modularity in the circuit description is allowed in terms of functional block models, but these can currently be used only for logic simulation, not for symbolic simulation.

3.1.2. Boyer–Moore computational logic. In this section, we focus on a restricted form of first-order logic called Boyer–Moore computational logic, which was developed for the explicit purpose of reasoning about computations. We start with a brief summary of the logic and its associated theorem-prover; a detailed description can be found in the presentation by Boyer and Moore [30]. An illustrative example of its use for hardware verification is then described in detail, followed by brief notes on other example applications.

Boyer–Moore logic is a quantifier-free² first-order logic with equality. Its syntax, which uses a prefix notation, resembles that of Lisp. Terms in this logic consist of variables and function applications. Constants are represented as functions with no arguments. Logical connectives, e.g. *not*, *and*, *or* are defined in terms of primitive logical constants *true*, *false*, and the primitive connective *if*. Equality is included in the form of a function *equal* and axioms that characterize it. A user is allowed to introduce inductively constructed object types (typically characterized by a set of axioms) by using the *Shell Principle*. According to this principle, all such type definitions should be accompanied by specification of a recognizer function, a constructor function, and an accessor function for that object type. A user is also allowed to introduce axioms that define new functions. In order to avoid inconsistencies due to their addition, these axioms have to satisfy the *Principle of Definition*. This principle ensures that all new functions are defined either non-recursively in terms of pre-defined functions, or in case of recursive definitions, a well-founded ordering exists on some measure of the arguments that decreases with each recursive call. The rules of inference used for reasoning in this logic are the standard ones used for propositional logic and the principle of induction³, which relies on the same well-founded ordering used by the definition axioms.

The Boyer–Moore theorem-prover provides an automated facility for generating proofs in the logic described above. The basic theory used by the system (logic plus theorem-prover) includes shells that axiomatize natural numbers, negative integers, lists and character strings. Commands are also provided for adding shells, defining new functions and proving theorems. However, the process of proof generation is not fully automatic, in that the theorem-prover may need assistance from the user for setting up intermediate lemmas and helpful definitions. The strong mathematical foundation and heuristics that have been built into the system have made it an effective tool that has been used in a number of application areas [29].

3.1.2.1. Example approach with Boyer–Moore logic. Hunt demonstrated the use of Boyer–Moore logic and the theorem-prover for verification of FM8501, a microprogrammed 16-bit microprocessor similar in complexity to a PDP-11 [34, 35]. The specification presents a programmer's view of FM8501 in the form of an interpretation function at the macro-instruction level. The implementation consists of its description as a hardware interpreter that operates at the micro-instruction level. Recursive function definitions within the logic are used to represent the

various combinational and sequential hardware units. (These definitions can be automatically expanded to obtain gate graphs consisting of standard Boolean gates). Verification is performed by proving a theorem that states the equivalence of the two descriptions, under appropriate assumptions of initial conditions.

Hunt's original work on the FM8501 microprocessor was extended to FM8502, a 32-bit microprocessor with a richer instruction set [36]. This work was part of a larger effort on systems verification [28], which included (in addition) verification of a code-generator for a simple high-level language [38], an assembler and linking loader [37], and a simple operating system kernel [27]. Each component in this "short stack" was formally specified as an abstract finite-state machine, and verified by showing its correspondence with a lower-level machine.

3.1.2.2. *Strengths and limitations.* Hunt's work highlights several features that are important for any effort that attempts to verify circuits at the scale of microprocessors. On the positive side, Hunt demonstrated the effectiveness of using a mechanized (though not fully automated) theorem-proving facility. A large number of theorems can arise while verifying a circuit as extensive and complex as a microprocessor. Trying to prove all of these by hand is highly unattractive if not actually impossible. Automation, to whatever degree possible, is desirable for all such efforts. Hunt also made good use of the recursion and induction principles allowed by Boyer-Moore logic, to reason about hardware functions with arbitrary-sized arguments. This helps in reuse of the same theorems for different implementations. For example, the operation of an n -bit ALU was verified, allowing a later instantiation for a specific size. Also, correctness was proved with respect to different data types—Boolean vectors, natural numbers, and integers—thus allowing effective data abstractions. In the case of the ALU, for example, this bridged the semantic gap between hardware bits and arithmetic numbers. Another important detail was the inclusion of hardware and software resets in the respective descriptions.

On the negative side, the handling of time is unsatisfactory in Hunt's approach. This is partly because of the limited expressibility of the Boyer-Moore logic. Since it is of first-order, functions of time cannot be represented directly in the logic. Sequencing, both at the micro- and the macro-levels, is implemented by using self-recursion with an explicit clock parameter. The first-order restriction also makes it difficult to model asynchronous inputs that are typically part of a microprocessor-memory interface. Hunt has used the concept of an "oracle" parameter that represents both the ticking of time as well as the asynchronous input values in each cycle. Handling of interrupts has not been attempted at all. Another limitation of this work is that low-level circuit aspects have not been considered. The main reason for this is that logic predicates, when used directly to correspond to circuit elements, can model only the binary (True/False) behavior of circuits. To effectively combine both high- and low-level circuit effects, either better circuit models within Boyer-Moore logic, or an interface with other

appropriate systems is necessary. A possible candidate for the latter is a switch-level simulator, with such hybrid approaches likely to become more common in the future. In summary, equivalence testing of micro- and macro-instruction levels is an important step towards verifying a microprocessor; however, it is still far from the goal of complete verification.

3.1.2.3. Related work. Another interesting approach that uses Boyer–Moore logic is that of German and Wang to prove functional properties of parameterized hardware designs [33]. Parameterization provides a compact way of representing a family of designs and has been used increasingly with hardware description languages. Typically, control parameters (that range over numbers) represent variation across size, and component parameters (that range over hardware circuits) represent variation across functionality. German and Wang have demonstrated verification of designs that use both kinds of parameters, based on the language Zeus. At present they deal with only synchronous circuits and operate mostly at the gate level. The use of induction is a powerful technique for verifying parameterized designs. It allows a whole family of circuits to be verified with a single proof, and is particularly useful in cases where it is impractical to verify large-sized circuits, since the proof size is independent of the circuit size. In this respect, Boyer–Moore logic, with its built-in heuristics for induction and recursion, is very well suited for verifying parameterized hardware.

Both approaches described above use terms within Boyer–Moore logic to directly represent hardware. Another technique is to use some other theory to characterize circuits, and then implement the theory within this logic. Such an approach has been adopted by Bronstein and Talcott [31] to mechanize a theory with a *functional semantics* for synchronous circuits. In essence, a synchronous circuit is identified with a monotonic, length-preserving function on *finite strings*, these being the complete histories of inputs and outputs of the circuit. (This is similar to a combinational circuit being identified with a function on Boolean values.) For example, a register with input string X , output string Y and an initial value a is represented as $Y = a.Past(X)$, where ‘.’ denotes concatenation and the string function $Past(X)$ represents all but the last character of X . By taking this functional view of each circuit element, a circuit can be characterized as a potentially recursive system of equations. An implementation of this theory in Boyer–Moore logic has been used to mechanically verify a sequence of synchronous circuits of increasing complexity, devised by Paillet [32]. Functional theories have the advantages of inherent compositionality and mathematical elegance, which is in direct contrast to the machine semantics traditionally employed to model synchronous circuits such as finite-state machines (FSMs). FSMs are not easily amenable to composition, and typically lead to an explosion in the number of states of systems based on them. However, functional approaches for characterizing circuits suffer from some disadvantages too. In general, it is difficult to represent bidirectional circuit elements (e.g. pass transistors), low-level circuit behavior, nondeterminism, and concurrency. As for

the particular approach used by Bronstein and Talcott, they admit a technical drawback in their formulation—that of using a domain of strings of equal length. This makes it tedious to reason about those circuits where the implementation operates at a time-scale different from that of the given specification.

3.1.3. Higher-order logic. In the previous sections we have described methods that use some kind of first-order logic for specification. The “first-order” part refers to the fact that only the domain variables are allowed to be quantified in such logics. If quantification is allowed over subsets of these variables (i.e. over predicates), one gets a “second-order” (predicate) logic. Continuing in this manner one obtains “higher-order” logics, which allow quantification over arbitrary predicates. The ability to quantify over predicate symbols leads to a greater power of expressiveness in higher-order logics than in the first-order case. For example, one can state the principle of mathematical induction on natural numbers by the following higher-order formula:

$$\forall P.((P\ 0) \wedge (\forall n.(P\ n) \Rightarrow (P\ (n + 1)))) \Rightarrow \forall n.(P\ n)$$

The above formula asserts that for all properties P , if P is true for 0 and if it being true for n implies that it is also true for $(n + 1)$, then it is true for all n . It is not possible to express this in first-order logic, since quantification over predicates (property P in this example) is not allowed. Another significant difference is that higher-order logics admit higher-order predicates and functions, i.e. arguments and results of these predicates and functions can themselves be predicates or functions. This imparts a first-class status to functions, and allows them to be manipulated just like ordinary values, leading to a more mathematically elegant formalism. It is these advantages of increased expressiveness and elegance that have attracted some researchers to explore higher-order logics as a means for specifying hardware.

However, higher-order logic systems suffer from some disadvantages too. The increased expressiveness carries with it a price tag of increased complexity of analysis. One disadvantage is the incompleteness of a sound proof system for most higher-order logics⁴, e.g. incompleteness of standard second-order predicate logic [6]. This makes logical reasoning more difficult than in the first-order case, and one has to rely on ingenious inference rules and heuristics. Also, inconsistencies can easily arise in higher-order systems if the semantics are not carefully defined [7]. A semantic model using a type hierarchy of domain elements (instead of a flat domain as in the first-order case) is effective against some kinds of inconsistencies, e.g. a famous one called Russell’s Paradox [7]. These disadvantages notwithstanding, hardware verification efforts that use higher-order logics have become increasingly popular in the past few years. The important consideration in most cases is to use some controlled form of logic and inferencing in order to minimize the risk of inconsistencies, while reaping the benefits of a powerful representation mechanism. In the remainder of this section we first

describe the HOL system in detail, which exemplifies concepts used by most higher-order logic approaches, and then briefly describe the other approaches.

3.1.3.1. Description of HOL. The HOL system was developed by the Hardware Verification Group at University of Cambridge, England. This system is based on a version of higher-order logic developed by Gordon for the purpose of hardware specification and verification. Both the logic and the theorem-proving system are collectively referred to as HOL, the former being “HOL logic” and the latter “HOL system”.

The HOL logic is derived from Church’s Simple Type Theory with the addition of polymorphism in types, and the Axiom of Choice built in via Hilbert’s ϵ -operator [47]. Syntactically, HOL uses the standard predicate logic notation with the same symbols for negation, conjunction, disjunction, implication, quantification, etc. There are four kinds of terms—constants, variables, function applications, and lambda-terms that denote functional abstractions. A strict type discipline is followed in order to avoid inconsistencies like Russell’s Paradox, and an automated type-inferencing capability is available. Polymorphism, i.e. types containing type variables, is a special feature supported by this logic. Semantically, types denote sets and terms denote members of these sets. Formulas, sequents, axioms, and theorems are represented by using terms of Boolean type.

The sets of types, type operators, constants, and axioms available in HOL are organized in the form of theories. There are two built-in primitive theories—*bool* and *ind*—for Booleans and individuals (a primitive type to denote distinct elements), respectively. Other important theories, which are arranged in a hierarchy, have been added to axiomatize lists, products, sums, numbers, primitive recursion, and arithmetic. On top of these, users are allowed to introduce application-dependent theories by adding relevant types, constants, axioms, and definitions. New types are introduced by specifying an existing representing type, a predicate that identifies the subset isomorphic to the new type and by proving appropriate theorems about them. Currently, a user is allowed to introduce arbitrary axioms into the system. This is potentially dangerous, since there is no method to check for consistency. An alternative that is being explored and strongly encouraged is to restrict the form of new axioms to be definitions, i.e. binding of a constant to a closed term. Additional theories that have only definitions for axioms cannot introduce any new inconsistencies and are therefore guaranteed to lead to safe extensions.

The HOL logic is embedded in an interactive functional programming language called ML. In addition to the usual programming language expressions, ML has expressions that evaluate to terms, types, formulas, and theorems of HOL’s deductive apparatus. The overall HOL system supports a natural deduction style of proof [6], with derived rules formed from eight primitive inference rules. In addition, there are special rules for help in automatic theorem-proving, e.g. a collection of rewrite rules. All inference rules are implemented by using ML functions, and their application is the only way to obtain theorems in the system.

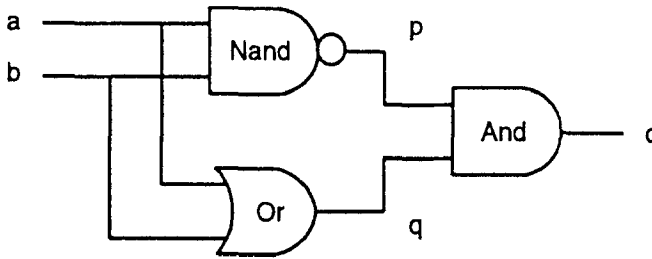


Figure 2. Implementation of an Exor-Gate.

Once proved, theorems can be saved in the appropriate theories to be used for future proofs. Most proofs done in the HOL system are goal-directed and are generated with the help of tactics and tacticals. A tactic is an ML function that is applied to a goal to reduce it to its subgoals, while a tactical is a functional that combines tactics to form new tactics. The tactics and tacticals in HOL are derived from the Cambridge LCF system (which evolved from the Edinburgh LCF [53]). The strict type discipline of ML ensures that no ill-formed proofs are accepted by the system.

3.1.3.2. Hardware model in HOL. In the HOL approach to hardware representation, circuits can be described in terms of both their behavior and structure [40]. Hardware devices are behaviorally described by higher-order predicates, which hold true for a set of arguments exactly when these argument values could occur on the corresponding ports of the device. Time-dependent behavior can be easily expressed by having arguments be functions of time. Circuit structure in terms of component parts is represented by taking a simple conjunction of the component behavior predicates, with interconnections between ports represented as common arguments. Hiding of internal line names can be accomplished by existential quantification of the corresponding variables. For example, a behavioral description for an Exor-gate can be represented as a predicate $Exor(a, b, c) \equiv (c = \neg(a = b))$, and its structural implementation in terms of simpler Boolean gates (shown in Figure 2) can be represented as $Exor_Imp(a, b, c) \equiv \exists p, q. Nand(a, b, p) \wedge Or(a, b, q) \wedge And(p, q, c)$.

Typically, primitive components in a hardware system are represented at the switch and gate levels. An oversimplified switch model is used in most cases; it models each transistor as a switch, with the gate controlling the connectivity between source and drain. Gate-level combinational circuits are easily described by using Boolean functions. Description of sequential circuits generally involves reasoning explicitly about time. The axiomatization of primitive recursion allows representation of parameterized hardware also.

3.1.3.3. Verification framework with HOL. Verification tasks in the HOL system can be set up in a number of different ways. The most common is to prove that an implementation, described structurally, implies (in some cases, is equivalent to) a behavioral specification. Other tasks can be formulated in terms of abstraction mechanisms useful for hardware verification that have been identified by Melham [59, 61]. As mentioned before, abstractions are essential for any verification system that needs to deal with large and complex designs. They help to relate different levels of a design hierarchy, enabling verification to proceed one level at a time, thereby making the process more tractable. The different abstraction mechanisms—structural, behavioral, data, and temporal—have been described in detail in Section 2.1.2. Verification tasks corresponding to each of these can be formulated in HOL, as summarized in Table 1.

Table 1. Typical HOL verification tasks.

Type of abstraction	HOL formulation
Structural	$\forall a\ b. \text{Spec}(a, b) \equiv \exists x\ y. \text{Imp}(a, b, x, y)$ where a, b are external signals, and x, y are internal signals
Behavioral	$\forall a\ b. \text{Imp}(a, b) \Rightarrow \text{Spec}(a, b)$ where $\text{Spec}(a, b)$ is a partial specification of behavior.
Data	$\forall a\ b. \text{Imp}(a, b) \Rightarrow \text{Spec}(f(a), f(b))$ where f is a data abstraction function, mapping data values of Imp to that of Spec .
Temporal	$\text{Incr}(f) \wedge \forall a\ b. \text{Imp}(a, b) \Rightarrow \text{Spec}(a.f, b.f)$ where f is a temporal abstraction function, mapping values on the time scale of Spec to that of Imp ; $\text{Incr}(x)$ denotes that x is an increasing function, and $\cdot$ denotes function composition.

3.1.3.4. Applications of HOL. Since its introduction, the HOL system has been applied in the verification of numerous hardware designs. Camilleri, Gordon, and Melham demonstrated the correctness of a CMOS inverter, an n -bit CMOS full-adder, and a sequential device for computing factorial function [40]. Herbert has described the verification of memory devices with low-level timing specifications and modeling of combinational delays [55] and verification of a network interface chip implemented in ECL logic [51]. Dhingra has used HOL to formally validate a CMOS design methodology called CLIC [45]. Other examples like the “Gordon’s” multiplier and a parity circuit have also been verified [48, 49]. An illustrative example of verification of a simple microprocessor called TAMARACK-1 (based on “Gordon’s” computer) is provided by Joyce [57]. Interestingly, a design error that was missed by formal verification was discovered after fabrication—a

reset signal to initialize the microinstruction program counter was found missing! The source of this omission was an invalid assumption that a relevant signal was bi-stable. Joyce extended his original work to verify TAMARACK-3, which included handling of hardware interrupts and asynchronous interactions, in the context of a formally verified system [58].

Other researchers outside the group at Cambridge have also used HOL for hardware verification. One such project was the use of formal methods for the design of a microprocessor called Viper at the Royal Signals and Radar Establishment in England [44]. Unlike other microprocessors that have been developed by similar methods (e.g. Hunt's FM8501, Joyce's TAMARACK), Viper was intended for serious use, and was amongst the first to be commercially produced. An important feature of Viper's formal development was its specification at decreasingly abstract levels of description. The specifications for the top two levels were first given informally by Cullyer, and later formalized in HOL by Cohn, who also gave a formal proof of correspondence between the two [42]. She has reported that the complete proof took about six person-months of work, and resulted in the generation of over a million inferences. According to her, the proofs were difficult both because of the size of theorems and due to computationally expensive operations like rewriting. She gives an interesting set of statistics for the proofs of some theorems in terms of the number of inference steps and CPU time used, but warns against taking them too seriously. Cohn has also written an interesting critique on the notion of proof for hardware systems [43].

3.1.3.5. Strengths and limitations. HOL is a powerful hardware verification system, deriving its strength on one hand from the expressiveness of higher-order logic, and on the other from the effectiveness of the automated theorem-proving facilities it provides. Unlike approaches that concentrate on only circuit behavior or structure, it can capture both. The ability to work with various abstraction mechanisms and hierarchical descriptions makes HOL very useful for handling large designs. The logic itself can be extended to allow formalization of virtually any mathematical theory. For example, packages have been formulated for defining recursive data types for hardware and for reasoning about them using induction [60]. This allows verification of parameterized hardware, e.g. an n -bit adder, tree-structured circuits, etc. As for the theorem-prover, it includes a wide variety of derived inference rules that contribute to its generality. In fact, the HOL system has been successfully used for reasoning about programs also [50]. An attractive feature of the system is its ability to evolve continuously. New theories and associated theorems become part of the system, which can be drawn upon for future proofs. Complex derived rules, found useful in a particular context, can be saved and reused elsewhere. With particular reference to hardware verification, for example, it allows technology changes to be incorporated easily into the model used to represent circuits.

On the negative side, apart from the usual disadvantages that higher-order

logic systems suffer from (mentioned at the beginning of this section), HOL has its own share of problems as well. One of these has been commonly referred to as the “false implies everything problem” [40]. Correctness statements in HOL are usually stated in the following form:

$$\begin{aligned} \forall i_1 i_2 \dots i_m o_1 o_2 \dots o_n. \text{Imp}(i_1, i_2 \dots i_m, o_1, o_2 \dots o_n) \\ \Rightarrow \text{Spec}(i_1, i_2 \dots i_m, o_1, o_2 \dots o_n), \end{aligned}$$

where, $i_1, i_2 \dots i_m$ are inputs and $o_1, o_2 \dots o_n$ are outputs. Since a false antecedent makes the implication trivially true, an *Imp* predicate that were equivalent to “False” would satisfy every specification! An easy example of such an *Imp* is a circuit where a line is connected to both power and ground. The usual definitions of power and ground equate them to True and False, respectively. Thus, the implementation asserts that the line is both True and False at the same time, which is clearly false. A couple of solutions to the above problem have been proposed. In the first one, a conjunct is added to the above statement, which asserts that for all inputs there exist some outputs that satisfy the predicate *Imp*, thus precluding *Imp* from being universally false. This solution is not quite satisfactory since it destroys the bidirectionality property of HOL circuit descriptions. Another alternative is to have better circuit models that would consider the drive strengths of various sources connected to a line, thus enabling it to handle short-circuits, etc.

Better circuit models are needed for other important reasons too. The primary reason, which applies to modeling in general, is accuracy of the verification task at hand. An example illustrating this point is described by the HOL Group itself [40], where an implementation of a CMOS Exor circuit is shown to be correct under the assumptions of the simple switch model. However, this circuit cannot be expected to work in practice due to threshold effects. (Threshold effects cannot be modeled by the simple switch.) Another example, also mentioned earlier, is related to Joyce’s microprocessor verification [57]. An error failed to get detected by formal verification due to an invalid assumption of bi-stability of a signal. Such assumptions clearly do not hold for primitive level components, and other models, e.g. those which consider an unknown or don’t-care value X , may fare better than binary models. Thus, inaccurate models can result in *false positives* (i.e. incorrect circuits being labeled correct). They can also result in *false negatives* (i.e. correct circuits being labeled incorrect), e.g. simple switch models cannot account for low-level effects like charge-sharing, which may be crucial to the working of a circuit like a precharged bus.

Another reason for better circuit models, though related in part to reliable verification, addresses the applicability aspect of a circuit model. For example, when formal verification of a cell library was attempted using HOL [52], it was realized that none of the HOL circuit models [45, 55, 60] could handle all relevant aspects of all the cells. (The simple switch model described earlier is appropriate for handling only static CMOS circuits.) In particular, only a

unidirectional model that represents combinational delay explicitly could handle static memory devices, and a unidirectional tri-state model that considers charge storage could handle tri-state bus drivers. Also, causality has largely been ignored in HOL circuit representations. Admittedly, use of predicates allows expression of bidirectionality in circuits. However, when applied in an uncontrolled manner, this can lead to strange conclusions with no bearing to reality. For example, using the HOL description of an inverter circuit [48], it is possible to derive ($i = \neg o$) which corresponds to an unreal “reverse” inverter.

3.1.3.6. Related work. Hanna and Daeche first proposed the use of higher-order logic for hardware verification [54] and provided inspiration for most of the work that followed including HOL. The overall approach, called VERITAS, demonstrated the effectiveness of a theory of circuits axiomatized within a higher-order logic setting and the theorem-proving techniques used by this system (based on those of the ML/LCF theorem-prover [53]). Other interesting features of Hanna’s approach include the detailed timing description used in the representation of analog waveforms, use of partial (as opposed to complete) specifications for circuit behavior, and the hierarchical development of theories, which enables verification to proceed at different levels of hardware abstraction.

Gordon’s work on the HOL system was preceded and influenced by his earlier work on the LCF_LSM system [46]. The LCF (Logic of Computable Functions) part of this system consisted of the programming environment for generating formal proofs interactively [53] that was used by both VERITAS and HOL. The other part was a specification language called LSM (Logic of Sequential Machines), which was used, as the name suggests, to specify sequential machines and other hardware. Special terms (based on behavior expressions of Milner’s CCS [24]) were used to specify sequential behavior in the form of output and next-state equations. Structural description consisted of an identification of circuit components, renaming of common lines, joining of components (a kind of parallel composition), and hiding of internal line names. As Gordon admitted himself, the LSM system was not entirely satisfactory. Inspired by the work of Hanna and Moszkowski [112] (to be described later), he adopted the use of terms in HOL logic in place of CCS-like terms for hardware description. Using a well-defined logic in place of relatively ad hoc expressions led to several advantages. Firstly, logic made it easier to organize descriptions hierarchically. The three essential operations for hierarchical representation—parallel composition, hiding, and renaming of variables—are realized in logic in a simple manner by conjunction, existential quantification, and substitution, respectively. Secondly, use of predicates instead of functions made it possible to describe bidirectional circuit devices more naturally. Finally, logical reasoning was also made more methodical by using standard inference rules instead of ad hoc rules.

Inspired by Gordon’s work on LCF_LSM, Barrow developed one of the first completely automated hardware verification systems, called VERIFY [39]. This system also represents a hardware design as a collection of hierarchically organized

modules, with each module represented as a finite-state machine. Verification consists of proving the equivalence of a structural description (of an implementation) and a behavioral specification. VERIFY, which has been implemented in Prolog (a logic programming language) [41], provides a variety of interesting automated techniques for proving equivalences. It checks for an identity, followed by an enumerative approach to test all cases, and when the latter is impractical it tries powerful algebraic manipulation methods. By way of examples, the hardware designs verified by VERIFY include Gordon's computer and a complex design to compute sums-of-products, which is parameterized in the number of bits of input and is described using nine levels of structural hierarchy. VERIFY's strengths lie in its ability to handle hierarchical descriptions, and in its degree of automation. Its main drawbacks are a restrictive notation for hardware description (allows representation of only finite-state machines), poor circuit models and use of enumerative techniques.

3.1.4. Temporal logic. So far, the logic formalisms we have described deal with static situations, i.e. the truth of propositions does not change over time. Temporal logic is an extension of predicate logic that allows reasoning about dynamically changing situations. It is a specialized form of *Modal Logic* which is best understood by considering the development of logic in terms of an increasing ability to express change. The brief description given here follows that given by Manna and Pnueli [105]; theoretical details can be found in a standard text [121], or in a recent article by Emerson [83].

Propositional logic deals with absolute truths in a domain of discourse, i.e. given a domain, propositions are either true or false. Predicate logic extends the notion of truth by making it relative, in that truth of a predicate may depend on the actual arguments (variables) involved. Since these arguments can vary over elements in the domain of discourse, the truth of a predicate can also vary across the domain. Extending this notion further, modal logic provides for additional variability, where the meaning of a predicate (or a function) symbol may also change depending on what "world" it is in. Variability within a world is expressed by means of predicate arguments, whereas changes between worlds are expressed by using *modal operators*. The dynamic connectivity between worlds (represented as states) is specified by an accessibility relation. In most cases this relation is never used explicitly, and modal operators are used to characterize properties that are true for states accessible from a given state. There are two basic modal operators—the necessity operator represented by $\Box$ (also called the Box) and the possibility operator represented by $\Diamond$ (also called the Diamond). The intended meaning is that a property $\Box P$ ($\Diamond P$, respectively) is true in state s if the property P is true in all states (at least one state, respectively) accessible from s .

To summarize, modal logic essentially consists of regular predicate logic enhanced by modal operators. Modal formulas are interpreted with respect to a state in a universe (where the universe consists of a set of states), a domain

of discourse over which appropriate logic symbols are interpreted by each state, and an accessibility relation between states. Temporal logic is derived from this basic framework by placing additional restrictions on the accessibility relation to represent passage of time. In other words, a state s is accessible from another state s' if it denotes a future state of s' . Thus, one can talk about the past, present, and future in the temporal development of (the state of) the universe. In addition to the basic modalities $\Box$ and $\Diamond$ described above, two other operators $\bigcirc$ and U —are frequently used. Within the temporal framework, these four are also referred to as *Always (Henceforth, 'G')*, *Sometimes (Eventually, 'F')*, *Next-time ('X')*, and *Until ('U')* respectively. The intended meaning of these operators, which shall be formalized subsequently, is as follows:

- $\Box P$ is true in state s , if P is true in all future states from s
- $\Diamond P$ is true in state s , if P is true in some future state from s
- $\bigcirc P$ is true in state s , if P is true in the next state from s
- $P U Q$ is true in state s , if either Q is true in s itself, or it is true in some future state of s , and until then P is true at every intermediate state

Some researchers have also considered past time duals of the above future time operators.

3.1.4.1. Specifications in temporal logic. Historically, temporal logic was first applied to the task of specifying and verifying concurrent programs by Pnueli in a landmark paper [116], followed by researchers too numerous to mention (for an excellent survey, see Pnueli's article [3]). Since reasoning about concurrent programs focuses more on their behavior in time, rather than their input–output behavior, temporal logic is particularly well suited for this purpose. In this context, the following classes of correctness properties have been identified, all very easily expressible in temporal logic:

- **Safety properties**—assert that nothing “bad” happens, typically represented as $\models \Box P$, i.e. P holds at all times in all models; e.g. partial correctness (no wrong answers are produced), mutual exclusion (no two processes are in the critical section simultaneously), deadlock freedom (no deadlock state is reached), global invariants (no violation of the invariants takes place)
- **Liveness properties**—assert that eventually something “good” happens, typically represented as $\models P \Rightarrow \Diamond Q$, i.e. in all models, if P is initially true then Q will eventually be true; e.g. total correctness (termination eventually occurs with correct answers), accessibility (eventually a requesting process will enter its critical section), starvation freedom (eventually service will be granted to a waiting process)
- **Precedence properties**—assert the precedence order of events, typically represented as $\models P U Q$, i.e. in all models, P will hold until Q becomes

true;

e.g. safe liveness (nothing bad happens until something good happens), fair responsiveness (responses are granted in the order of requests placed)

The application of these ideas to hardware verification is accomplished by considering appropriate semantic models for hardware, as described later in this section.

3.1.4.2. *Strengths and limitations.* The most important advantage provided by temporal logic is its ability to express dynamic properties naturally, along with a method of reasoning that focuses specifically on temporal operators. Another advantage is that it can be used at virtually any level of abstraction, thus providing a common technique across the entire abstraction hierarchy. Lamport provides arguments in favor of the temporal logic approach to verification [98], e.g. its axiomatic approach of specifying properties instead of an abstract model, which facilitates hierarchical specification, and its emphasis on states rather than operations of the system to be verified.

On the other hand, temporal logic is not very well suited for functional verification tasks. Though theoretically it can handle them, it would fare no better than simple predicate logic. A more serious limitation of temporal logic is with respect to its expressiveness. As formulated above, it is useful for reasoning only qualitatively about time. Specification and verification of quantitative timing properties requires additional technical machinery. Moreover, even with qualitative specifications, it cannot express many interesting properties that one might want to verify. Wolper has addressed this aspect of temporal logic [129], which we shall describe in detail in Section 3.1.5 on Extended Temporal Logic.

3.1.4.3. *Classification of temporal logics.* Note that we have still not specified details of the semantic model with respect to which temporal formulas are interpreted. In fact, many variants on this have led to the development of different kinds of temporal logic. One distinction is based on whether the truth of a formula is determined with respect to a state, or with respect to an interval between states. The latter has given rise to what is commonly known as *Interval Temporal Logic* and is described towards the end of this section. Within the former, there has been further categorization based on the difference in viewing the notion of time. In one case, time is characterized as a single linear sequence of events, leading to *Linear Time (Temporal) Logic*. In the other case, a branching view of time is taken, such that at any instant there are a branching set of possibilities into the future. This view leads to *Branching Time (Temporal) Logic*. Contrary to what one might naively imagine, this almost philosophical difference has far-reaching consequences, and has been the subject of many a lively debate between proponents on each side. We shall return to the historical development of this interesting issue after we have described the main approaches

using each of these formulations.

3.1.4.4. Linear time temporal logic (LTTL). This variation of temporal logic, popularized by Manna and Pnueli [105], is based on models that are infinite sequences of states. The accessibility relation is specified such that for an ω -sequence of states $\sigma = s_0, s_1, \dots; s_j$ is accessible from s_i if and only if $i \leq j$. The i -truncated suffix of σ is denoted $\sigma^{(i)}$, i.e. $\sigma^{(i)} = s_i, s_{i+1}, \dots$.

Formulas are formed from terms and operators in much the same way as in first-order predicate logic, with only local variables and propositions (Boolean-valued variables) allowed to vary over states. The truth of a formula is determined with respect to a model that consists of a global interpretation of signature symbols I , an assignment of values to global variables α , and an ω -sequence of states σ , where each state of the sequence assigns values to the local variables and propositions. (In the following, I and α are not mentioned explicitly and the focus is on σ . Amongst the classical operators, the subset consisting of $\wedge$, $\neg$, and $\exists$ is considered, since the other operators $\vee$, $\Rightarrow$, $\equiv$, $\forall$ can be expressed in terms of this subset. “If and only if” is abbreviated as “iff”.) Truth of a formula with respect to σ is inductively defined on the structure of formulas as follows:

- if ϕ is an atomic formula,
 $\sigma \models \phi$ iff $s_0 \models \phi$, i.e. ϕ is simply interpreted over the first state of σ
- $\sigma \models \neg \phi$ iff $\sigma \not\models \phi$
- $\sigma \models \phi \wedge \psi$ iff $\sigma \models \phi$ and $\sigma \models \psi$
- $\sigma \models \exists x. \phi(x)$ iff there exists a value d (where $d \in \text{domain of discourse}$),
such that $\sigma \models \phi(d/x)$ (where $\phi(t_2/t_1)$ indicates substitution of t_1 by t_2 in ϕ)
- $\sigma \models \Box \phi$ iff $\forall k \geq 0, \sigma^k \models \phi$
- $\sigma \models \Diamond \phi$ iff $\exists k \geq 0, \sigma^k \models \phi$
- $\sigma \models \bigcirc \phi$ iff $\sigma^1 \models \phi$
- $\sigma \models \phi U \psi$ iff $\exists k \geq 0$ such that $\sigma^k \models \psi$, and $\forall i, 0 \leq i < k, \sigma^i \models \phi$

(Note: The form of *Until* used in the above description is also known as *Strong Until*, since it requires ψ to hold true in some state. A weaker version, called the *Weak Until*, admits the case where ψ may never become true, in which case ϕ remaining true forever would satisfy the until-formula).

Some examples of interesting properties expressible in LTTL are

- $\Box \Diamond \phi$: ϕ holds infinitely often along a path
- *Request* $\rightarrow \Diamond$ *Grant*: if a resource has been requested, it will eventually be granted
- $\Diamond \Box \neg$ *Enabled* $\vee \Diamond$ *Chosen*: a process cannot be enabled infinitely often without ever getting chosen

3.1.4.4.1. Hardware model with LTTL. To characterize the behavior of concurrent

programs in terms of sequences of states, Pnueli has proposed an abstract computational model called “fair transition systems” (FTS) [3]. An FTS consists of a set of states (not necessarily finite), some of which are specified to be initial, and a finite set of transitions. Nondeterminism is allowed by representing each transition as a function from a given state to a set of states. In addition, *justice* and *fairness* requirements are included by specifying a justice set $\mathcal{J}$ and a fairness set $\mathcal{T}$, each of which is a set of subsets of transitions. An admissible computation of an FTS is a sequence of states and transitions, such that the starting state of the sequence is one of those designated initial, each state follows from the previous one by an appropriate transition, and the computation terminates only if no transitions are enabled. It is also ensured that each admissible computation is just and fair, i.e. if an element of the justice (fairness) set, which is itself a set of transitions, is enabled continuously (infinitely often) beyond a certain state, then a transition belonging to that element will be taken at least once (infinitely often) beyond that state. LTTL formulas are interpreted over sequences of states that correspond to admissible computations of an FTS.

The properties of *justice* and *fairness* arise quite often in dealing with concurrent processes. This is because concurrency in a system of processes is frequently modeled as an interleaving of the individual process executions. In order for this kind of serialization to not introduce anomalies, one has to ensure that all enabled processes are given a fair chance to contribute the next transition of the overall system. Various notions of fairness conditions have been identified to achieve this [88, 100, 120]. Of these, *impartiality*, *justice*, and *fairness* have often been found useful. Typically, impartiality requires that all processes execute infinitely often. Justice requires that if a process is enabled continuously beyond a certain time, then it will be executed eventually. Fairness is even stronger, and requires that if a process is enabled infinitely often, it will be executed infinitely often.

3.1.4.4.2. Verification framework with LTTL. Both theorem-proving and model checking techniques (explained in Section 3.1) have been applied to the task of verifying LTTL assertions. In the following we give examples of each.

Manna and Pnueli studied deductive proof systems for LTTL within the context of concurrent program verification [104, 106, 107]. Their system [106] consists of three main parts—the uninterpreted logic part (that gives general axioms for first-order temporal logic with equality), the domain part (which considers models with a fixed interpretation of predicate and function symbols and in which variables range over specific domains, e.g. integers, lists etc.) and the program part (which further restricts the class of models to be computations of fair transition systems). Axioms for the first and second parts are very general and do not change across different programming languages. They also present an axiom schema for the third part, which is proved to be relatively complete with respect to the first two. This axiom schema provides a unified framework for reasoning about different languages, and can be tailored to a particular

one through appropriate definitions of atomic transitions, justice and fairness. They give concrete examples with a shared variables computational model and CSP [56].

Lichtenstein and Pnueli presented a model checking algorithm for determining satisfiability of propositional LTTL formulas with respect to finite state models similar to the fair transition systems described above [101]. To check if a formula ϕ is satisfied by a program P , a product graph G is constructed from the states of P and $Cl(\phi)$ (the closure of subformulas of ϕ). The construction of G is such that ϕ is satisfied by P if and only if there is an infinite path in G from a starting state that contains ϕ . This involves finding strongly connected components of G , and the overall complexity is $O(|P| \cdot 2^{|\phi|})$. With slight modifications, the same algorithm can handle various notions of fairness (impartiality, justice, fairness, generalized fairness) as well as past temporal operators.

3.1.4.4.3. Related work with LTTL. Owicki and Lamport presented an independent proof method (using proof lattices) for proving liveness properties with LTTL [115]. One of the first examples of using LTTL for hardware verification was provided by Bochmann in verifying an asynchronous arbiter through reachability analysis done by hand [66]. Malachi and Owicki identified derived temporal operators (e.g. *while* operator) useful for formal specification of self-timed systems, using a version of temporal logic similar to that described above [103], but did not provide any proof methods.

Manna and Wolper used propositional LTTL for the specification and synthesis of the synchronization part of communicating processes [108]. Sistla and Clarke proved that the problems of satisfiability and model checking in a particular finite structure are NP-complete for the propositional LTTL logic with only (F) , and are PSPACE-complete for the logics with various subsets of operators— (F, X) , (U) , (X, U) , (X, U, S) [123].

One of the severe criticisms of the Manna–Pnueli proof system approach described above is that it is inherently global and non-compositional. One needs to reason about the global state of the complete program (including all its associated variables) in order to prove a temporal property. To remedy this situation, several efforts have been made towards development of compositional proof systems. One of the techniques uses edge propositions (and edge variables) to distinguish between transitions made by a module and those made by the environment, as suggested by Lamport [97], and also used by Barringer, Kuiper, and Pnueli [64]. Another technique is to partition the interface variables into sets, such that each module may modify only those variables that it owns [3]. In any case, past temporal operators have been found convenient and extended temporal operators necessary for completeness of the compositional proof systems [3]. Pnueli generalized these ideas further within the context of an “assume-guarantee” paradigm to characterize an interface between a module and its environment [117]. In general terms, a guarantee specifies the behavior of a module, under the assumption that constrains the environment. He also

gave pragmatic recommendations highlighting the differences between global-style and compositional-style proof systems for temporal logic.

3.1.4.5. Branching Time Temporal Logic (BTTL). Different kinds of BTTL logics have been proposed depending on the exact set of operators allowed, the common feature being that they are interpreted over branching time structures. The usual temporal operators (F , G , X , and U) are regarded as state quantifiers. Additional quantifiers, called the path quantifiers, are provided to represent all paths (A) and some path (E) from a given state. The propositional versions of these logics can be best identified within a unified syntactic framework consisting of *state-formulas* (abbreviated *state-f*) and *path-formulas* (abbreviated *path-f*), defined as follows [86]:

$$\langle \text{state-f} \rangle ::= \langle \text{atomic-proposition} \rangle \mid \quad (1)$$

$$\neg \langle \text{state-f} \rangle \mid \langle \text{state-f} \rangle \wedge \langle \text{state-f} \rangle \quad (2)$$

$$A(\langle \text{path-f} \rangle) \mid E(\langle \text{path-f} \rangle) \quad (3)$$

$$\langle \text{path-f} \rangle ::= \langle \text{atomic-proposition} \rangle \mid \quad (4)$$

$$\neg \langle \text{path-f} \rangle \mid \langle \text{path-f} \rangle \wedge \langle \text{path-f} \rangle \quad (5)$$

$$F \langle \text{state-f} \rangle \mid G \langle \text{state-f} \rangle \quad (6)$$

$$X \langle \text{state-f} \rangle \mid \quad (7)$$

$$\langle \text{state-f} \rangle U \langle \text{state-f} \rangle \mid \quad (8)$$

$$F \langle \text{path-f} \rangle \mid G \langle \text{path-f} \rangle \quad (9)$$

$$X \langle \text{path-f} \rangle \quad (10)$$

$$\langle \text{path-f} \rangle U \langle \text{path-f} \rangle \quad (11)$$

Some of the studied BTTL logics, in terms of numbered parts of definitions above, are:

- BT—set of state formulas generated using definitions (1), (2), (3), and (6)
- BT⁺—set of state formulas generated by adding definition (5) to those of BT
- UB—set of state formulas generated using definitions (1), (2), (3), (6), and (7)
- UB⁺—set of state formulas generated using definition (5) to those of UB
- CTL—set of state formulas generated using definitions (1), (2), (3), (6), (7), and (8)
- CTL⁺—set of state formulas generated by adding definition (5) to CTL
- CTL*—set of state formulas generated using all eleven (1)–(11) definitions above

(Note: In general, for a BTTL logic L that allows a path quantifier to prefix a single state quantifier, the L⁺ version of the logic allows a path quantifier to prefix a boolean combination of state quantifiers.)

In fact, different LTTL logics can also be described within the same framework as

- $L(F)$ – set of path formulas generated by definitions (4), (5), and (9)
- $L(F, X)$ – set of path formulas generated by definitions (4), (5), (9), and (10)
- $L(F, X, U)$ – set of path formulas generated by definitions (4), (5), (9), (10), and (11)

We will describe the CTL approach in detail, followed by a brief description of related work with other logics.

3.1.4.5.1. Verification framework with CTL (Computation Tree Logic). Clarke and Emerson first proposed CTL and presented efficient algorithms for CTL model checking, within a larger framework of automatic synthesis of synchronization skeletons from CTL specifications [74]. Clarke, Emerson, and Sistla demonstrated the effectiveness of using CTL for automatic verification of finite-state systems [75]. In their approach, a finite-state system is modeled as a labeled state-transition graph. Formally, this graph can be viewed as a finite *Kripke structure* [93] and is represented as a triple $M = (S, R, P)$, where S is a finite set of states, R is a total binary relation on states and represents possible transitions, and P is a mapping that assigns to each state the set of atomic propositions true in that state. A *path* within this structure is naturally defined as an infinite sequence of states, with each adjacent pair related by R .

As its name suggests, CTL interprets temporal formulas over structures that resemble computation trees. In the context defined above, given an M and a state s_0 , it considers the infinite computation tree rooted at s_0 , generated by considering all possible nondeterministic transitions at every state. The truth of a CTL formula is defined inductively as follows:

- $(M, s_0) \models p$ iff $p \in P(s_0)$, where p is an atomic proposition
- $(M, s_0) \models \neg\phi$ iff $(M, s_0) \not\models \phi$
- $(M, s_0) \models \phi \wedge \psi$ iff $(M, s_0) \models \phi$ and $(M, s_0) \models \psi$
- $(M, s_0) \models A X\phi$ iff for all states t such that $(s_0, t) \in R$, $(M, t) \models \phi$
- $(M, s_0) \models E X\phi$ iff for some state t such that $(s_0, t) \in R$, $(M, t) \models \phi$
- $(M, s_0) \models A(\phi U \psi)$ iff for all paths $(s_0, s_1, s_2 \dots)$,
 $\exists k \geq 0$ such that $(M, s_k) \models \psi$ and $\forall i, 0 \leq i < k, (M, s_i) \models \phi$
- $(M, s_0) \models E(\phi U \psi)$ iff for some path $(s_0, s_1, s_2 \dots)$,
 $\exists k \geq 0$ such that $(M, s_k) \models \psi$ and $\forall i, 0 \leq i < k, (M, s_i) \models \phi$

The other operators— $AF\phi$, $EF\phi$, $AG\phi$ and $EG\phi$ —are regarded as abbreviations for $A(True U \phi)$, $E(True U \phi)$, $\neg EF(\neg\phi)$ and $\neg AF(\neg\phi)$, respectively. The combination of path and state quantifiers in these operators is best illustrated with examples as in Figure 3. (In the figure, shaded (unshaded) nodes represent

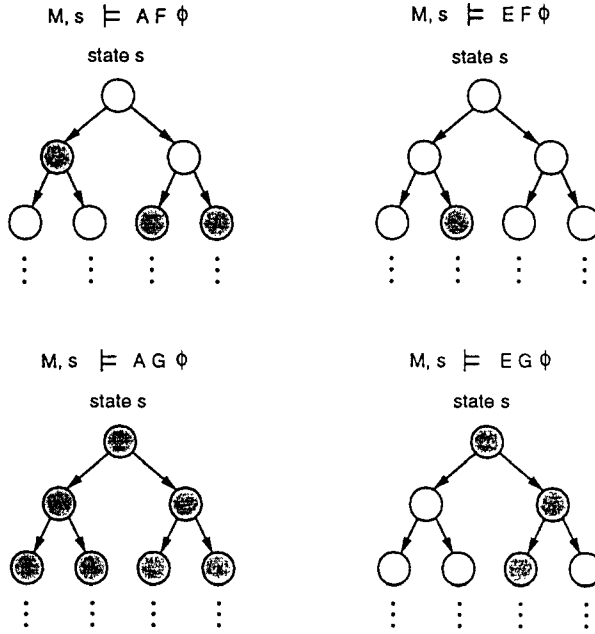


Figure 3. CTL operators.

states with ϕ true (false), respectively.) Some examples of interesting properties expressible in CTL are

- $AG(AF\phi)$: ϕ holds infinitely often along all paths
- $AG(Request \Rightarrow AFGrant)$: it is always true that if a request is made, it will eventually be granted
- $EF\phi$: it is possible to reach a state where ϕ holds

To check whether a formula ϕ holds in a state s of a structure M , Clarke et al. give a model checking algorithm of complexity linear in the size of both M and ϕ . This algorithm operates iteratively on the length of the formula ϕ —at the end of stage i , each state is labeled with the set of subformulas of length $\leq i$ that hold true in that state. Each stage involves at most a depth-order traversal of the underlying state-transition graph. At the end of n stages (where $n = |\phi|$), if state s is labeled with ϕ , then ϕ holds in s ; otherwise it does not. In the latter case, the model checker tries to find a counter-example, i.e. a path that demonstrates the negation of the formula to be true. An example is shown in Figure 4 for checking the CTL formula $EF\phi$. Starting initially from a graph as shown on the top left, iterative approximations for $EF\phi$ are computed as shown ($U1, U2, U3$; marked as shaded nodes), until a fixed point is reached ($U3$). At

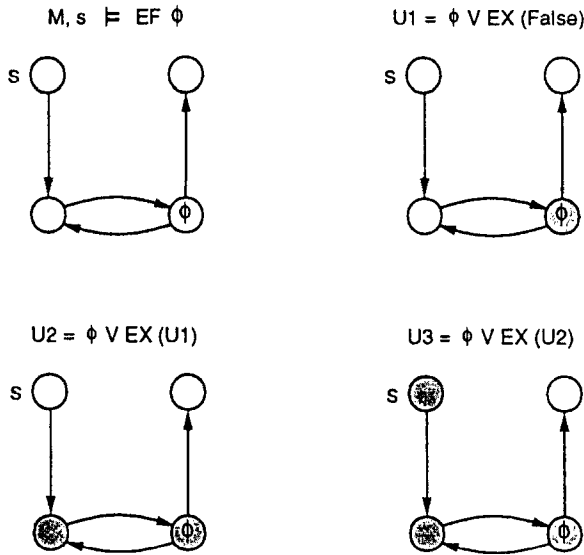


Figure 4. Example of CTL model checking.

this point, since the node s is shaded, we conclude that the formula is true in state s .

Since fairness cannot be expressed in CTL [86], Clarke et al. modify the semantics of CTL such that path quantifiers now range over only fair paths. (Fair paths in this context are defined as those along which infinite states satisfy each predicate that belongs to a fairness set F). This new logic, termed CTL^F , can handle various notions of fairness, including those of *impartiality* and *weak fairness* (but not *strong fairness*), by appropriately defining the corresponding fairness sets. Model checking for CTL^F is done by first identifying fair paths (by using strongly connected components in the graph of M), followed by application of the model checking algorithm to only these paths. This results in additional complexity linear in the size of the fairness set F .

3.1.4.5.2. Applications of the CTL framework. The basic framework described above has been applied to numerous hardware and protocol verification problems by Clarke and his coworkers. Different techniques have been used to generate a state-transition graph with respect to which CTL model checking is performed:

- A global state-transition graph is constructed by considering all possible interleavings of individual processes described in CSP. This has been used in the verification of an asynchronous communication protocol called the Alternating Bit Protocol [75].

- For sequential circuits, the state-transition graph can be automatically generated either by circuit simulation (using a mixed gate- and switch-level simulator with a unit delay timing model) or by direct compilation from a high-level state machine description language called SML [68]. The former has been used towards verification of (and finding a bug in) a published design for a self-timed queue element [70]; the latter has been used for verification of numerous hardware controller circuits, e.g. a traffic controller [70], a DMA controller [73], etc.
- The unit-delay timing model does not work very well for asynchronous circuits, since their operation depends upon components with varying delays. To verify their correctness, a *speed-independent* timing model is often used, which assumes arbitrary finite delays on every output in the circuit. A technique was presented for state-transition graph extraction under the speed-independent model [82]. This technique utilizes *flow-tables* to describe the operation of each primitive circuit element (typically a Boolean gate, a Muller C Element, or a Mutual Exclusion Element), which are then combined to yield transitions for the whole circuit. A timing error was found in a published design for an asynchronous arbiter circuit, and a modified version was subsequently verified to be correct.
- A speed-independent model is too conservative in practice, since designers typically use reasonable assumptions about relative component delays to obtain faster/smaller circuits. Another method was proposed [69] that allows such delay assumptions (constant lower and upper bounds on individual delays, bounds on differences between delays) to be included in the state-transition graph construction. This was used to verify the correct operation of a queue-element under particular delay assumptions, where the same circuit could not be proved correct under the speed-independent model.

3.1.4.5.3. Related work on other BTTL logics. Ben-Ari, Pnueli, and Manna studied the UB (Unified Branching) logic and presented a procedure for deciding the satisfiability of a UB formula with respect to a structure similar to the Kripke structure described above [65]. This decision procedure is based on construction of a semantic tableau⁵ and is of exponential complexity. They also provided an axiomatization (axiom-based proof system) for the logic and proved it to be complete.

Queille and Sifakis independently proposed a model checking algorithm for a logic with CTL modalities (without the *Until*) [119]. Formulas are interpreted with respect to transition systems that are derived from an interpreted Petri-net description of an implementation (translated from a high-level language description), within a verification system called CESAR. In their algorithm, interpretation of temporal operators is iteratively computed by evaluating fixed points of predicate transformers. However, they do not provide any means for handling fairness in their model checking approach.

Emerson and Halpern proved the small-model property of CTL, provided exponential time tableau-based decision procedures for CTL satisfiability, and

extended the axiomatization given by Ben-Ari et al. to cover CTL along with a proof of its completeness [85]. They also studied the expressiveness of various BTTL logics and showed that $UB < UB^+ < CTL \equiv CTL^+$ [85].

Emerson and Lei considered additional linear time operators denoted by $\overset{\infty}{F} p$ ("infinitely often p ", same as GFp) and $\overset{\infty}{G} p$ ("almost always p ", same as FGp) [87]. They defined FCTL by extending the notion of fairness in CTL, to consider fairness constraints that are Boolean combinations of the $\overset{\infty}{F}$ and $\overset{\infty}{G}$ operators. Combinations of these operators can express strong fairness (as well as other notions of fairness found in literature). Model checking for FCTL is proved to be NP-complete in general, but is shown to be of linear complexity when the fairness constraint is in a special canonical form. They also presented a model checking algorithm for CTL^* , which is shown to be PSPACE-complete [75].

3.1.4.5.4. Model checking and the state explosion problem. One of the serious limitations of the model checking approach is its reliance on an explicit state-transition graph representation of the hardware system to be verified. Typically, the number of states in a global graph increases exponentially with the number of gates/processes/elements (parallel components) in the system, resulting in what is popularly called the *state explosion* problem. This restricts the application of direct state enumeration approaches to small circuits only. Several alternatives have been explored in order to alleviate this problem. Some rely upon variations in the logic and methodology (described in the remainder of this section) in order to reason about an arbitrary number of processes, or to reason about components, thereby using smaller (non-global) graphs. Others use effective techniques such as symbolic manipulation (described in the next section) in order to explore the state-space implicitly. These two approaches can often be combined, resulting in substantial computational savings.

Apt and Kozen proved that it is not possible, in general, to extend verification methods for a finite-state process in order to reason about an arbitrary number of processes [62]. However, several researchers have addressed special cases of this problem. Clarke, Grumberg, and Browne introduced a variant of CTL^* , called Indexed CTL^* (ICTL*), which allows formulas to be subscripted by the index of the process referred to (without allowing constant index values) [77]. A notion of *bisimulation* is used to establish correspondence between Kripke structures of two systems with a different number of processes, such that an ICTL* formula is true in one if and only if it is true in the other. However, the state explosion problem is not really avoided, since the bisimulation relation itself uses the state-transition relations explicitly. The notion of correspondence between Kripke structures was later extended, such that a *process closure* captures the behavior of an arbitrary number of identical processes [76]. Reasoning with the process closure allows establishment of ICTL* equivalence of all systems with more than a finite number of processes. However, this process closure has to be provided by the user. (Similar approaches using network invariants were

proposed within the context of a more general process theory and automata techniques [167, 170], described in Section 3.2.2.)

Sistla and German also addressed this problem in the context of concurrent CCS processes [125]. They give fully automatic procedures to check if all executions of a process satisfy a temporal specification (given in propositional LTTL) for two system models—one consisting of an arbitrary number of identical processes, and the other consisting of a controller process and an arbitrary number of user processes. These algorithms can also be used for reasoning about global properties (e.g. mutual exclusion) and about networks of processes (e.g. token rings). However, the high complexity of these algorithms (polynomial and doubly exponential in process size, respectively) limits their practical application to some extent.

A related problem was addressed by Wolper for reasoning about an infinite number of data values [126]. He shows that a large class of properties of a process stated over an infinite number of data values are equivalent to those stated over a small finite set, provided the process is *data-independent*. Informally, a process is data-independent if its behavior does not depend upon the value of the data. (In general, determining data-independence for a process is undecidable, but certain syntactic checks can be used as sufficient conditions.) This has been used to specify correctness of a data-independent buffer process (i.e. given an infinite sequence of distinct messages, it should output the same sequence) by showing that it is enough to specify the buffer for only three distinct messages. (An unbounded buffer cannot be characterized in propositional temporal logic otherwise [124].) This significantly adds to the specification power of propositional temporal logic, and also extends the applicability of the associated verification methods.

Another different track explored by various researchers has been in the direction of promoting hierarchical/modular reasoning in the hope of reducing the size of the state-transition graphs. Mishra and Clarke proposed a hierarchical verification methodology for asynchronous circuits [111], in which restriction on the language of atomic propositions is used to hide internal nodes of a system. They then identified a useful subset of CTL without the next-time operator, called CTL^- , such that truth of CTL^- formulas is preserved with respect to the restriction operation.

A compositional approach was presented by Clarke, Long, and McMillan [79], in which an interface rule of inference allows modeling of the environment of a component by a reduced *interface* process, while still preserving the truth of formulas. Simple conditions have been identified for the rule to be valid within a general process model and an associated logic. Examples have been given for the case of both asynchronous and synchronous process models, with variants of CTL^* , and with appropriate notions of compositions. The language SML is also extended to handle modular specifications (called CSML for Compositional SML) [80]. This approach is best utilized for loosely coupled systems where the resulting interface process can be kept simple.

More recently, Grumberg and Long have also proposed a framework for compositional verification with the logic $\forall\text{CTL}^*$ (a subset of CTL^* without the existential path quantifier) [90]. It uses a preorder on finite-state models that captures the notion of a composition (as having less behaviors than a component). The truth of logic formulas is preserved by the preorder, such that satisfaction of a formula corresponds to being below the structure representing its semantic tableau. An assume-guarantee style of reasoning [117] within this framework allows verification of temporal properties for all systems containing a given component. This methodology has been demonstrated for compositional verification of $\forall\text{CTL}$ formulas (CTL formulas without the existential path quantifier) with respect to Moore machine models.

Another recent method proposed by Clarke, Grumberg, and Long is based on the use of abstractions with model checking of formulas in $\forall\text{CTL}^*$ [78]. Data abstractions (mappings) constitute a homomorphism from a given model of a system to an abstract model, such that the truth of a $\forall\text{CTL}^*$ formula in the abstract model implies its truth in the original model. In practice, a conservative approximation of the abstract model is obtained by automatic symbolic execution of a high-level program over the abstract domain (by using abstract interpretations of the primitive relations). This method is particularly useful for reducing complexity of verification of datapaths, as has been demonstrated by its application to multipliers, a pipelined ALU, etc.

3.1.4.5.5. Symbolic methods for model checking. Symbolic model checking has received a great deal of attention from various researchers lately. It was initially explored by McMillan [109] and was proposed independently by Bose and Fisher [67], and by Coudert, Madre, and Berthet [81]. The underlying idea common to these approaches is the use of symbolic Boolean representations for states and transition functions (or relations) of a sequential system, in order to avoid building its global state-transition graph explicitly. Efficient symbolic Boolean manipulation techniques are then used to evaluate the truth of temporal logic formulas with respect to these models. In the case of CTL model checking, this typically takes the form of fixpoint computations.⁶ Symbolic representation allows the regularity in state-space of some circuits (e.g. datapaths) to be captured succinctly, thus facilitating verification of much larger circuits in practice than is possible with direct state enumeration techniques, as demonstrated by Burch et al. [72]. McMillan and Schwalbe successfully applied these methods for verification of the Encore Gigamax cache consistency protocol and found some critical design errors [110], thus demonstrating the effectiveness of symbolic methods for real-life industrial applications.

It is also interesting to compare the differences between these approaches. The method used by Burch et al. [72] is very general and can handle nondeterministic systems, thus allowing its application to both synchronous and asynchronous circuits. However, this generality is gained at the cost of increased complexity of representing the complete transition relation symbolically (using Bryant's

BDDs [12]). Bose and Fisher, on the other hand, model systems as deterministic Moore machines, and use symbolic representations of the next-state functions (not relations) [67]. The latter are derived directly from symbolic simulation of the circuit to be verified using the switch-level simulator COSMOS [17]. Coudert et al. also use a deterministic Moore machine model with symbolic representation of the next-state function [81]. However, they use more sophisticated Boolean manipulation operations (e.g. “constraint”, “restrict” operators) to keep down the size of their internal data representations called TDGs (Typed Decision Graphs). (TDGs are similar to BDDs and provide an equivalent canonical representation of Boolean formulas.)

Bryant and Seger have presented another extreme in this spectrum of symbolic methods [71]. They avoid explicit representation of even the next-state function. Instead, they use the simulation capability of COSMOS to symbolically compute the next-state of each circuit node of interest. This restricts them to using a limited form of temporal logic that can express properties over finite sequences only (unlike the other approaches that can handle full CTL). They reason within a symbolic Ternary algebra (with logic values 0, 1, and X) to compute the truth values of formulas.

3.1.4.6. LTTL versus BTTL. As mentioned before, LTTL logics take a linear view of the underlying notion of time and interpret formulas over linear sequences of states. Operators are provided to reason about properties along a single sequence (path). With respect to validity in a model⁷, the formulas are thus implicitly universally quantified to reason about all possible state sequences. On the other hand, BTTL logics take a branching view of time, where all possible futures are considered at every state. In effect, BTTL logics use explicit path quantifiers A and E to reason about paths in an entire execution history, these paths themselves being represented by linear time formulas.

The controversy between these two was first sparked by Lamport [96]. He focused on $L(F, G)$ and BT as examples of LTTL and BTTL logics, respectively, and provided interpretations of the former over paths and the latter over states of a model. A notion of equivalence of two formulas (A and B) was defined to mean that they are either both valid or both invalid, for all models M with a given set of states (i.e. $M \models A \equiv M \models B$). Lamport then showed that the expressiveness of $L(F, G)$ is incomparable to that of BT , since each can express a certain formula to which no formula of the other is equivalent. Differentiating clearly between the nondeterminism used to model concurrency and that which is inherent in some programs, he argued that LTTL is better for reasoning about concurrent programs, since BT cannot express strong fairness (i.e. $FG \neg (\text{Enabled}) \vee F(\text{Chosen})$). He also maintained that since it is usually required to reason about all possible computations of a concurrent program, the implicitly universally quantified LTTL formulas are better suited for the task. On the other hand, he argued that BT is better suited for reasoning about inherently nondeterministic programs, since LTTL cannot express existential properties at all (e.g. one of the possible

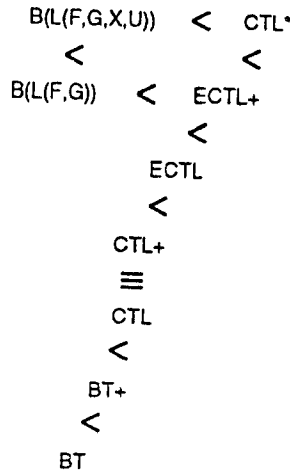


Figure 5. Relative expressiveness of temporal logics.

executions terminates).

This controversy was revisited by Emerson and Halpern [86]. They presented various versions of LTTL and BTTL logics within a unified framework consisting of state and path formulas (described earlier). They also pointed out technical difficulties with Lamport's notion of equivalence and used a modified definition to prove various expressiveness results, as shown in Figure 5 (where $B(L)$ denotes the associated branching time logic for a linear time logic L , and a logic at the bottom/left of a ' $<$ ' sign is strictly less expressive than the logic at the top/right). Lamport's results regarding incomparability of $L(F, G)$ and BT do hold, but are not true in general about linear and branching time logics. In fact, CTL^* with infinitary operators $\bar{F}$ and $\bar{G}$ can express strong fairness and is strictly more expressive than $L(F, G, X, U)$.

Apart from issues of expressiveness, another main contention between proponents of the two logics has been complexity of the model checking problem. A model checking algorithm of linear time complexity was first presented by Clarke, Emerson, and Sistla for CTL (described earlier), and was demonstrated to be effective for a variety of finite-state systems [75]. On the other hand, model checking for a variety of LTTL logics was proved to be PSPACE-complete by Sistla and Clarke [123]. In a practical approach to the same problem, Lichtenstein and Pnueli presented a model checking algorithm for $L(F, G, X, U)$, which was linear in the size of the model but exponential in the size of the formula to be verified [101]. They argued that since most formulas to be verified are small in practice, using LTTL model checking was a viable alternative to BTTL. Finally, Emerson and Lei proved that given a model checking algorithm for an LTTL logic (e.g. $L(F, G, X, U)$), there exists a model checking algorithm of

the *same* complexity for the corresponding BTTL logic (e.g. CTL*) [87]. Since BTTL is essentially path-quantification of LTTTL formulas, this result implies that one gets this quantification for (almost) free. Thus, they argued, the real issue is not which of the two is better; rather, it is what basic modalities are needed in a branching time logic to reason about particular programs, i.e. what linear time formulas can follow the path quantifiers.

Within the larger framework of formalization of reactive systems, Pnueli offers an insightful article on the dichotomy that exists between the linear time and branching time views, and which cuts across various methodological issues [118].

3.1.4.7. Interval temporal logic (ITL). The temporal logics discussed in the earlier sections are state-based logics, i.e the truth of atomic propositions (and variables) depends on states. In ITL, as the name suggests, this depends on intervals of states. ITL has an additional operator, called the “chop (;)” operator (borrowed from Process Logic [92]), in terms of which all the usual temporal operators can be represented. In the following we describe the propositional part of ITL, as presented by Halpern, Manna, and Moszkowski [91].

Since ITL formulas are interpreted over intervals of states, a model for propositional ITL consists of an interval from a set of states S and an interpretation function P mapping each propositional variable p and a non-empty interval $s_0 \dots s_n \in S^+$ to a truth-value. The length of an interval $s_0 \dots s_n$ is n , zero-length intervals (consisting of a single state s_0) are permitted, an initial subinterval is of the form $s_0 \dots s_i$, and a terminal subinterval is of the form $s_i \dots s_n (0 \leq i \leq n)$. Truth of ITL formulas is inductively defined as follows:

- $(P, s_0 \dots s_n) \models p$ iff $P(p, s_0 \dots s_n) = \text{true}$, where p is an atomic proposition
- $(P, s_0 \dots s_n) \models \neg \phi$ iff $(P, s_0 \dots s_n) \not\models \phi$
- $(P, s_0 \dots s_n) \models \phi \wedge \psi$ iff $(P, s_0 \dots s_n) \models \phi$ and $(P, s_0 \dots s_n) \models \psi$
- $(P, s_0 \dots s_n) \models \bigcirc \phi$ iff $n \geq 1$ and $(P, s_1 \dots s_n) \models \phi$
- $(P, s_0 \dots s_n) \models \phi ; \psi$ iff
 $\exists i, 0 \leq i \leq n$ such that $(P, s_0 \dots s_i) \models \phi$ and $(P, s_i \dots s_n) \models \psi$
i.e. there is at least one way to chop the interval $s_0 \dots s_n$ into two adjacent subintervals $s_0 \dots s_i$ and $s_i \dots s_n$ such that ϕ is true on the first subinterval and ψ on the second.

Additional operators for subintervals, initial subintervals, and terminal subintervals can be defined in terms of the chop operator. It has been proved by Halpern et al. that satisfiability of arbitrary propositional ITL formulas is undecidable [91]. However, if all propositional variables are restricted to be local, i.e. a propositional variable p is true of an interval $s_0 \dots s_n$ iff p is true in its first state s_0 , then the satisfiability problem becomes decidable. Even in this case, the best result obtained is that the complexity of the problem is elementary in the depth of the operators $\neg$ and $;$. This limits the practical usefulness of the logic to some extent.

On the other hand, the expressiveness of ITL has been demonstrated for expressing various properties of an interval, e.g. checking its length, checking the truth of a formula in its initial state, in its terminal state, etc. The full first-order version of ITL has been used to explore different ways of expressing detailed quantitative timing information for hardware models, e.g. signal stability, temporal delay parameterized by propagation times, gate input and output delays, etc. Based on the ease with which intervals can be used to specify timing details, Halpern et al. proposed the use of this logic not only for verification but also for providing a rigorous basis for describing digital circuits. Several examples of this, ranging from simple latch elements to the Am2901 bit-slice ALU, can be found in Moszkowski's work [112, 114]. Another interesting research direction was the work done on the programming language Tempura [113], which offers a way to directly execute hardware descriptions based on a useful subset of ITL. (Arbitrary ITL descriptions are not executable.)

Leeser has also used ITL for functional and temporal specification of CMOS circuits [99]. Circuits at the transistor and gate levels are hierarchically described using the logic programming language Prolog. These Prolog descriptions of the implementation are converted to ITL descriptions using a rule-based system. Verification is performed by showing that the specification formula logically implies the implementation formula. Leeser's main contribution lies in extending the ITL switch model used by Moszkowski to include capacitive effects with an associated delay. She also uses constraints to impose conditions on inputs (like setup and hold times etc.). She has demonstrated this methodology on examples that include a dynamic latch and a dynamic CMOS adder.

3.1.5. Extended Temporal Logic. Temporal logic, as described in the previous section, is good at expressing certain properties of programs like partial correctness, mutual exclusion, liveness, etc. However, it cannot express some other simple properties like regular properties of execution sequences, which have been found useful by the program verification community [128]. (For example, a sequence of alternating *request* and *grant* events can be specified as a regular expression $(request; grant)^*$.) Since the propositional version of LTTL (called PTL) is as expressive as a first-order theory of linear order [89], some means of introducing higher-order constructs is needed to extend the expressiveness of standard PTL. One such method was suggested by Wolper in a classic paper [129].

Wolper proved that it is impossible to express properties like "a proposition p should hold in every even state of an execution sequence" in PTL. He proposed introduction of new operators corresponding to right-linear grammars⁸, leading to an Extended Temporal Logic (ETL) corresponding to each grammar. The basic idea is that given a right-linear grammar G and a particular assignment of formulas to the terminals of G , a word $w = v_{w_0}v_{w_1} \dots$ generated by G describes exactly those sequences of states $\sigma = s_0, s_1 \dots$ such that the formula associated with v_{w_i} is true

in state s_i . Since right-linear grammars are equivalent to regular expressions, this enables regular properties of sequences to be specified. More formally, consider a right-linear grammar $G = (V_T, V_{NT}, P, V_0)$ where $V_T = \{v_0, v_1 \dots v_n\}$ is a set of terminal symbols, $V_{NT} = \{V_0, V_1, \dots V_k\}$ is a set of nonterminal symbols, P is a set of productions, and V_0 is the starting nonterminal. The ETL corresponding to G consists of PTL augmented with a set of k operators $\mathcal{G}_i(f_0, \dots f_n)$, one for each nonterminal symbol V_i , such that

- $\sigma \models \mathcal{G}_i(f_0, \dots, f_n)$
iff there exists a word $w = v_{w_0} v_{w_1} \dots (0 \leq w_j \leq n)$ generated by G from V_i such that $\forall j, \sigma^j \models f_{w_j}$, where σ^j is the suffix of σ starting with s_j .

For determining the satisfiability of ETL formulas, an exponential-time decision procedure similar to the semantic tableau method of Ben-Ari et. al. [65] was presented. A sound and complete axiomatic system for ETL was also provided by adding axioms for grammar operators to those for PTL.

The original work on ETL was generalized subsequently by Wolper in collaboration with Vardi and Sistla [130]. They redefined the notion of grammar operators in terms of n -ary automata operators ($A = (\Sigma, S, R, s_0, F), |\Sigma| = n$). Each application of an automata operator $A(f_0, f_1, \dots, f_n)$, uses a correspondence between the alphabet (Σ) and the n -ary arguments as follows:

- $\pi \models A(f_0, f_1, \dots, f_n)$
if there is an *accepting* run $\sigma = s_0, s_1 \dots$ of $A(f_0, f_1, \dots, f_n)$ over π (where π represents an infinite sequence of truth assignments to atomic propositions).
- A run of a formula $A(f_0, f_1, \dots, f_n)$ over π
is a sequence $\sigma = s_0, s_1, \dots$ of states from S , such that
 $\forall i, 0 \leq i \leq |\sigma|, \exists a_j \in \Sigma$ such that $\pi^i \models f_j$ and $s_{i+1} \in R(a_j, s_i)$.

By considering three different ways of defining *accepting* runs, they obtained the following three logics:

- ETL_f : A run σ is accepting iff some state $s \in F$
- ETL_l : A run σ is accepting iff it is infinite
- ETL_r : A run σ is accepting iff some state $s \in F$ occurs infinitely often in σ

The above three notions of acceptance of infinite words by nondeterministic finite-state automata are often referred to as finite, looping, and repeating acceptance, respectively. They proved that the above three logics are expressively equivalent, despite the fact that the finite and looping acceptance automata define strictly smaller classes of languages than those defined by repeating acceptance (also called Büchi) automata. In fact, these logics are shown to be expressively equivalent to Büchi automata⁹. Though expressively equivalent, the complexity

of the satisfiability problem varies greatly amongst these logics. It is proved to be PSPACE-complete for ETL_f and ETL_i , and in EXPSpace for ETL_r .

Clarke, Grumberg, and Kurshan presented an application of automata operators in the context of specifying branching-time properties, resulting in a logic called ECTL [127]. They considered four different types of automata on infinite words—Büchi automata (described in Section 3.3.2), Muller automata, L -automata (described in Section 3.2.2), and $\forall$ -automata. They described an efficient model checking algorithm for operators corresponding to deterministic Muller automata, which is of complexity linear in size of the model and length of the formula, and polynomial (of low order) in size of the largest Muller automaton used. They leave open the problem of extending their results to handle the nondeterministic versions of the automata discussed.

3.1.6. Mu-Calculus. Mu-Calculus provides another framework for extending expressiveness within the temporal logic context. It provides operators for denoting fixed points of predicate transformers (functions from relations to relations), in addition to the usual predicate logic operators. Mu-Calculus was developed independent of temporal logic, with various versions proposed and studied in the context of program verification [84, 133, 135, 137]. The following description is a summary of Mu-Calculus as used for hardware verification [131].

The signature of the logic consists of individual variable symbols (x_1, x_2 , etc.) and predicate variable symbols of some fixed arity (X, Y, Z , etc.). Formulas consist of the usual predicate logic formulas ($True, \neg\phi, \phi \wedge \psi, \exists x_1\phi$) and application of relational terms to individual variables. Relational terms (P, Q , etc.) consist of (1) predicate variables (2) abstraction terms (of the form $\lambda x_1, x_2 \dots x_n[\phi]$), and (3) fixpoint terms (of the form $\mu Z.[P]$). The n -ary fixpoint term $\mu Z.[P]$ represents the least fixed point (lfp) of an n -ary relational term P (also regarded as a predicate transformer), which is formally monotone¹⁰ in the predicate variable Z . Another operator denoted ν is defined as a dual to μ and represents the greatest fixed point (gfp) of a relational term. (A formal description of the full Mu-Calculus and the associated fixpoint theory is beyond the scope of this article. The interested reader may see the references cited earlier for details.)

Emerson and Lei proposed viewing CTL not as a sublanguage of the branching time temporal logic CTL* (described in Section 3.1.4.5), but as a sublanguage of propositional Mu-Calculus [132]. In effect, the lfp and GFP operators μ and ν , respectively) are used to give fixpoint characterizations of the basic CTL modalities as follows:

- $EF\phi \equiv \mu Z.[\phi \vee EX Z]$
- $EG\phi \equiv \nu Z.[\phi \wedge EX Z]$
- $E(\phi U \psi) \equiv \mu Z.[\psi \vee (\phi \wedge EX Z)]$

They also demonstrated the unified framework provided by Mu-Calculus for incorporating various extensions described earlier—handling fairness constraints,

past-tense operators, and extended temporal operators. For example, the FCTL fairness assertion $E \bar{F} p$ is expressed as $\nu Z_1. \mu Z_2. [EX((p \wedge Z_1) \vee Z_2)]$ and the ETL assertion that “ p holds in every even state” as $\nu Z. [p \wedge AXAX Z]$.

They presented a model checking algorithm for the propositional Mu-Calculus, which is of exponential time complexity for the full calculus, but is of polynomial time complexity for restricted fragments called $L\mu_k$ (where depth of alternating μ and ν operators is bounded by $(k-1)$). They also showed that $L\mu_2$ is sufficient for formalizing both CTL and FCTL (as well as some other process logics), thus reaffirming efficient model checking approaches for these logics within a unified framework.

Recently, there has been renewed interest in the generality offered by the Mu-Calculus approach. Burch et al. explored a new angle in the solution to the model checking problem [131]. They use symbolic methods (based on Bryant’s BDDs [12]) to represent both the formulas/terms of the calculus and the state-space of the associated model. They demonstrate the effectiveness of the unified Mu-Calculus model checking approach by its application to seemingly diverse problems, such as

- CTL model checking
- testing satisfiability of propositional linear temporal logic formulas
- testing observational equivalences for finite transition systems
- deciding language containment between ω -automata (finite-state automata on infinite words)

The use of symbolic methods allows them to verify much larger circuits in practice than is possible by direct state enumeration methods.

3.1.7. Functional approaches and special calculi. In this section we describe approaches that use some form of functional equivalence as the basis for verification. A specialized language, rather than a conventional logic, is chosen for expressing the specification. (As mentioned earlier, we include this section here because of its syntactic similarities with conventional logic.) Typically, both the specification and the implementation are represented as functions within the language. Special calculi or rules are formulated for the task of proving equivalence between the two.

Wagner was amongst the first researchers to use formal techniques for hardware verification [145, 146]. He used a non-procedural register transfer language both to describe the hardware and to specify its intended behavior. To show that a circuit satisfies its specifications, a series of transformations are applied to the lower-level component definitions, to show that they produce the same results as the specification. Elementary Boolean reduction rules, clock transition rules, etc. are provided in the form of theorems, and all manipulations are carried out symbolically by a theorem-prover with assistance from the user. Conditions

for detection of some types of races and hazards are also provided. Though its accomplishments might seem modest by today's standards, as one of the first systems to verify LSI chips and microprocessors, it effectively demonstrated the feasibility of formally verifying hardware.

Milne designed a special-purpose calculus called CIRCAL with which to describe, specify and verify hardware [141, 142]. Each device in CIRCAL terminology is described structurally by a set of communication port names (inputs and outputs) called its "sort". Its behavior is described by using CIRCAL expressions with operators for nondeterminism, choice, abstraction, composition, guarding, and termination. A complex system is modeled hierarchically as an interconnection of simpler devices and its behavior derived from that of its parts. Timing information can be modeled by including a clock port in the system description. Specifications are also represented as CIRCAL descriptions and are typically abstracted forms of the implementation itself. Functionality and timing equivalence of the two descriptions can be proved syntactically by using the CIRCAL rules associated with the different behavior operators. An interesting feature of this system is that, apart from verification by mathematical proof, it also allows for simulation. A single test pattern is represented as a CIRCAL expression, which is then composed with the device description to yield the response. By using associativity and idempotency properties of the composition operator, it is possible to perform *constructive* simulation, i.e. compose the results of simulation on components to produce the overall response. CIRCAL has been used to verify an automated silicon compiler as well as a library of VLSI components.

Sheeran developed a VLSI design language called μ FP for the design of synchronous regular array circuits [143, 144]. This language, based on the functional programming language FP, employs higher-order functions that are given semantic as well as geometric interpretations. Circuits are hierarchically described in μ FP as either combinational arrays or as stream arrays (that employ memory). Behavioral descriptions of circuits are transformed by applying algebraic laws to obtain layouts that are correct by construction. Circuit transformations are also performed by application of appropriate laws of transformation, in order to explore various design alternatives. The main contribution of her approach was in demonstrating the feasibility of using a functional approach for the detailed design of correct hardware.

Another major effort in this area has been that of Weise. His system, called Silica Pithecus [147], focuses on verification of low-level synchronous MOS circuits. He differentiates clearly between analog- (signal-) level behavior and digital-level behavior of a circuit, utilizing an abstraction function mapping the former to the latter. A novel feature of his methodology is the use of constraints, i.e. Boolean predicates, for the following:

- to ensure valid inputs (only inputs that meet the constraint conditions are considered during verification)
- to ensure valid outputs (these constraints are automatically generated in order

- that the intended behavior is met)
- to specialize behavior (as provided by the designer)

Verification is performed by proving that for all inputs such that the constraints are met, the abstracted signal behavior is equivalent to the intended digital behavior. Rewrite rules and a tautology-checker are used to prove the equivalence. Large circuits can be handled by hierarchical verification. At any given level of the hierarchy, constraints can either be accepted (they are shown to hold), rejected (shown not to hold) or propagated upwards, to be proven later. The major strengths of Silica Pithecus lie in its powerful circuit model (capable of handling charge-sharing, ratioed circuits, threshold effects, races, and hazards), its novel way of using constraints, and its hierarchical operation. Its main weakness lies in the combinatorial nature of its proof method—more powerful theorem-proving tactics are likely to be needed for it to be effective on complex circuits.

Borrione, Camurati, Paillet, and Prinetto also used a functional methodology for verification of the MTI microprocessor [138], developed at CNET in France. Their functional model uses a discrete representation of time, supported by difference equations and reference to past instances. The operation of the microprocessor is modeled at the machine-instruction, microprogram, and micro-instruction levels, with a functional semantics imparted to each machine- and micro-instruction. Proofs of correspondence between these levels are accomplished through semi-automated use of a tautology-prover, a theorem-prover (called OBJ2), some ad hoc Lisp functions and proofs done by hand. Borrione and Paillet used a similar functional methodology for the verification of hardware described in VHDL (a hardware description language) [139]. Essentially, VHDL descriptions of both the behavioral specification (functional and temporal) and the structural implementation are converted to a set of functional equations, respectively. Verification is performed by showing the equivalence (or the logical implication) between these two sets of equations.

Special calculi have also been used to provide compositional models for circuits (different from traditional non-compositional models used in switch-level simulators [17]). Winskel originally proposed a compositional model for MOS circuits [148]. He combined ideas from Bryant's switch-level model [17] (using a lattice of voltage values) with Milner's Calculus of Communicating Systems (CCS) [24] and Hoare's formulation of Communicating Sequential Processes (CSP) [56]. Circuit behaviors are modeled as static configurations, each static configuration characterized by voltage values, signal strengths, internal voltage sources, and signal flow information. Composition of circuits is allowed if the ports that are connected impose consistent constraints on the environment (as captured by static configurations). However, the inherent complexity of the model makes it difficult to use it in practice. More recently, a simpler compositional switch model has been developed by Chaochen and Hoare [140]. They also present a simulation algorithm for their switch-model and develop a theory of assertions for the specification and verification of synchronous circuits.

3.2. Automata / Language theory

In this section, we describe approaches that use an automaton, or equivalently a language, to represent a specification. Various kinds of finite-state automata and languages have been explored in this context. We include their definitions where appropriate. We also describe approaches that use a closely related concept to finite-state automata, called finite-state machines. Both automata and machines are defined in terms of an underlying state-transition structure with an input alphabet. However, automata are typically regarded as *acceptors* of words, with a single binary output (accept/reject) based on some form of acceptance conditions. Machine models, on the other hand, incorporate an output alphabet into their definition, and outputs are associated with either states or transitions [155]. The focus is mostly on output sequences in response to input sequences, rather than on a notion of acceptance of input sequences.

3.2.1. Machine equivalence. In this section we describe approaches that represent both the implementation and the specification as finite-state machines. To show that every behavior of the implementation satisfies the specification, an equivalence between the corresponding machines is proved. The main limitation of this method springs from the fact that equivalence is sometimes too strict a relationship than is desired for *satisfaction* of a specification by an implementation. For example, a verification method for checking equivalence cannot always be modified to account for partial specifications.

3.2.1.1. Machine model for sequential circuits. Sequential circuits are frequently represented in the form of deterministic finite-state machines. One such machine model, called the Moore machine model, is formally denoted by a six-tuple (S, I, O, NF, OF, s_0) , where

- S is a finite set of states,
- I is an input alphabet,
- O is an output alphabet,
- NF is a next-state function, mapping $S \times I$ to S ,
- OF is an output function, mapping S to O , and
- s_0 is a starting state ($s_0 \in S$).

In the Moore machine model, an output is a function of state alone. Another variation, where the output is a function of both state and inputs, is called the Mealy machine model. To represent a sequential circuit as a Moore machine, its logic level description is organized in the form of three basic units as shown in Figure 6—a set of latches (corresponds to S), next-state logic (purely combinational—corresponds to NF), and output logic (purely combinational—corresponds to OF).

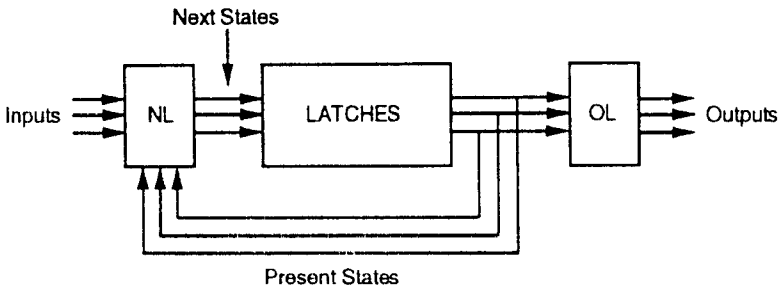


Figure 6. Moore machine model for sequential circuits.

3.2.1.2. Verification framework. The Moore and Mealy machine models correspond naturally to deterministic finite-state automata (DFAs). The classic algorithm for checking the equivalence of two DFAs [155] involves the construction of a composite automaton that accepts only the exclusive-or of the two corresponding languages. If this composite automaton accepts a non-empty language, then the two DFAs are not equivalent. This algorithm forms the basis of several formal approaches for checking the equivalence of sequential circuits. For example, to prove the equivalence of two Moore machines M_1 and M_2 (with the same input and output alphabet), a product machine is formed with an output in every state that checks the equality of the outputs of the individual machines. Thus M_1 and M_2 are equivalent if and only if the composite machine produces a “true” output in every state reachable from the start state. However, the problem is intractable in general, since an exponential number of input patterns need to be tested exhaustively to establish the equivalence. In this section we describe two approaches that mark a significant improvement in performance over other such methods, due to their use of efficient graph algorithms such as depth-first traversal and symbolic breadth-first search.

3.2.1.3. Example approaches. Devadas, Ma, and Newton focused on sequential machine descriptions at different levels of abstraction—the register-transfer (RT) level and the logic level. They addressed the problems of verifying equivalence of (1) an RT-level description against a logic-level description, and (2) two logic-level descriptions [154]. Both algorithms involve extraction of a state-transition graph (STG) from the machine description. They use a special technique for using the “don’t-care” information (e.g. invalid input and output sequences) from the STG of the first machine, to reduce the size of the STG of the second machine. This results in considerable time and memory savings during STG processing in rest of the algorithm. Once the two STGs are available, their equivalence is verified. For problem (1), since the starting state for the RT-level machine is specified, an extension of the standard DFA equivalence algorithm is used to incorporate the don’t-care transitions. This algorithm has been shown to extend easily to

Mealy machine models as well. For problem (2), if the starting states in the logic-level descriptions are known, they use an enumeration/simulation approach where acyclic paths in the STG of the first machine are enumerated one at a time, simulated on the second machine, and the outputs compared. Since only one path is enumerated at any time, using depth-first-search from the starting state, the memory requirements are considerably less than having to store the entire STG.

Another major contribution in this area has been made by researchers at the Bull Research Center in France. They developed a tautology-checker called PRIAM [150], which was used to verify the equivalence of a specification (expressed as a program in a hardware description language called LDS) and an implementation (also an LDS program, extracted from a structural description of the circuit, e.g. layouts, gate-level description, etc.) [156]. Basically, each LDS program is reduced by symbolic execution to a canonical form of Boolean function representation called a Typed Decision Graph (TDG) [149], thereby reducing the task of checking equivalence to that of checking syntactic equality. The main drawback of this early work was that both the specification and the implementation programs were required to have the same states and the same state encodings, thus severely limiting its application.

This verification framework was generalized by Coudert, Berthet, and Madre [151, 152]. They use the standard algorithm for comparison of two Mealy machine models, i.e. the output of the two machines should be the same for every transition reachable from the starting state. The significant contribution of their approach is the idea of using a *symbolic breadth-first search* of the state-transition graph of the composite machine, instead of the usual depth-first techniques used in other methods. This technique is best illustrated with the help of Figure 7. For the composite machine description, its states, inputs, and outputs are represented as Boolean vectors, and its next-state and output functions (NF and OF , respectively) as vectors of Boolean functions (obtained from symbolic execution of the LDS programs). The algorithm proceeds in stages, where stage k ensures the correctness of the symbolic outputs of transitions from states that are reachable, from the initial set of states, through k transitions (labeled *From* in the figure). For the next iteration, the next-states of *From* are symbolically evaluated (by using the next-state function NF and the symbolic inputs X), thus giving the set of states (*New*) that are reachable in $(k + 1)$ transitions. The algorithm ends when no more *New* states can be found, at which point the outputs of all reachable transitions have been examined. In effect, symbolic manipulation allows all transitions and states reachable from a certain set of states (*From*) to be evaluated using only one operation, without explicit enumeration of either the states or the input patterns. Also, efficient symbolic manipulation techniques (e.g. simplification of a function under a constraint) [153] are used to reduce the size of TDGs where possible. Savings in both time and memory are thus achieved, enabling them to handle much larger circuits than possible by use of non-symbolic methods.

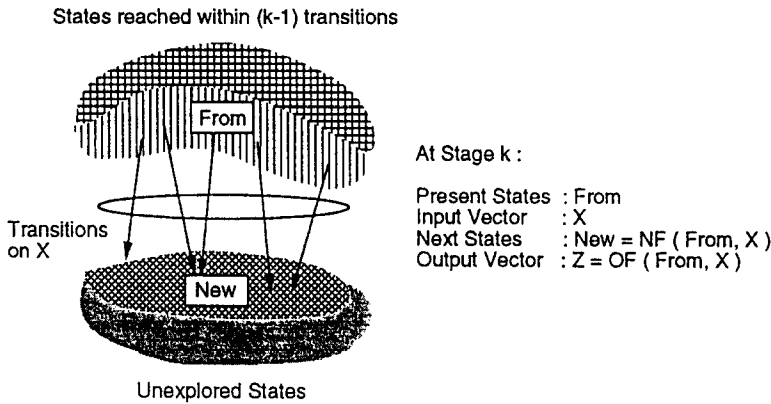


Figure 7. Symbolic breadth-first search.

3.2.2. Language containment. A generalization of the method used in the previous section is to consider containment rather than equivalence between the languages representing the implementation and the specification ($L(Impl)$ and $L(Spec)$, respectively). In other words, it is formally proved that $L(Impl) \subseteq L(Spec)$ in order to verify that every behavior of the implementation satisfies the property expressed by the specification. This allows easier handling of partial specifications as well as abstractions, thereby facilitating hierarchical verification across different levels of abstraction. Such an approach has been used by Kurshan and his coworkers. Their work is described in this section.

3.2.2.1. Models for specifications and implementations. Finite-state automata that accept sequences (infinite words, ω -words), rather than strings (finite words), are useful for modeling non-terminating processes. Several such automata have been studied; an introductory description can be found in a recent article by Thomas [169]. Among languages accepted by these automata, an ω -regular language $\mathcal{L}$ is one that can be represented as $\mathcal{L} = \bigcup_{i=1}^n U_i \cdot V_i^\omega$ where U_i and V_i are regular sets (of strings) over the language alphabet, V_i^ω denotes sequences of strings in V_i and $\cdot$ denotes concatenation. Büchi automata¹¹ accept exactly the set of ω -regular languages [158].

Kurshan has defined modified versions of finite-state automata and finite-state machines that accept sequences, called L -automata and L -processes, to represent specifications and implementations, respectively. (The description that follows is summarized from his recent presentation [166]). A primary difference in Kurshan's models is an algebraic structuring of inputs to the automaton/machine. A transition between states is enabled by a *set* of input values rather than one input value (as is the case in traditional models). Any such set can be represented by a predicate over input values. These predicates form a Boolean algebra¹²

denoted L . Thus, L represents the set of all subsets of input values. The transition structure of the automaton/machine is viewed as an adjacency matrix over L , with the (i, j) -th entry specifying the enabling predicate for a transition from state i to state j . The definitions of both an L -automaton and an L -process are in terms of this underlying L -matrix (hence the name L -automaton/process).

An L -automaton is defined¹³ to be a four-tuple $\Gamma = (M_\Gamma, I(\Gamma), R(\Gamma), Z(\Gamma))$, where

- M_Γ is the transition L -matrix, which may be nondeterministic, i.e. there may be more than one transition enabled by the same input; the set of states is implicitly regarded as the set of vertices corresponding to the adjacency matrix M_Γ
- $I(\Gamma)$ is a set of initial states of Γ
- $R(\Gamma)$ is a subset of transitions, called recurring edges of Γ
- $Z(\Gamma)$ is a set of subsets of states, called cycle sets of Γ

The alphabet of the language accepted by an L -automaton implicitly consists of $Atoms(L)$, the set of atoms¹⁴ of the Boolean algebra L . A *run* of the automaton on an infinitary word $x \in Atoms(L)^\omega$ is a sequence of states $s_0 s_1 \dots$ such that $x_i * M_\Gamma[s_i, s_{i+1}] \neq 0$ (where $*$ indicates the conjunction operation and 0 indicates the additive identity of L). In other words, the i^{th} input x_i has to enable the i^{th} transition predicate $M_\Gamma[s_i, s_{i+1}]$ in the sequence of states that form a run. A run from an initial state is said to be *accepting* if either

- the set of states that appear infinitely often in the run belongs to $Z(\Gamma)$ (the run is called Γ -cyclic), or
- transitions from the set $R(\Gamma)$ appear infinitely often in the run (the run is then called Γ -recurring).

The set of all accepting runs is defined to be the language accepted by Γ , denoted $\mathcal{L}(\Gamma)$. Given an arbitrary ω -regular language over $Atoms(L)$, there is an L -automaton that accepts the same language [165]. In this manner an L -automata can be used to specify any ω -regular property. The relationships between L -automata and other automata on infinite words, e.g. Büchi, Muller automata, etc., have also been studied [159, 165].

An L -process $A = (M_A, S_A, I(A), R(A), Z(A))$ is similar to the L -automaton, with an additional component S_A that represents the nondeterministic selections (outputs) of the machine. The function S_A maps every state to the subsets of L , the Boolean algebra underlying its transition matrix M_A , such that every transition from a state s is enabled by some selection $z \in S_A(s)$ from s . Again, the alphabet of the language accepted by A consists of $Atoms(L)$, with runs being defined in the same manner. However, the acceptance conditions for an L -process are defined to be reverse of those for an L -automaton, i.e. the run should be neither A -cyclic nor A -recurring. This facilitates expression of fairness

constraints for the machine. It is also shown that an arbitrary ω -regular language over $Atoms(L)$ can be accepted by some L -process [166].

Using the L -process model, the behavior of a system consisting of a finite number of coordinating component processes $A_1, \dots, A_k$ can itself be modeled as a “product” L -process $A = A_1 \otimes \dots \otimes A_k$ (where $\otimes$ denotes a tensor product operation [166]). The nondeterministic selections associated with each component (which also serve as inputs to other coordinating components) allow handling of nondeterminism inherent in the system represented by A . Essentially, in each process A_i , a selection z_i in the current state s_i is chosen nondeterministically from the set of possible selections $S_{A_i}(s_i)$. The product $z = z_1 * \dots * z_k$ denotes the global selection of the system A . This z then determines a set of possible next-states in each A_i , i.e. those states to which transitions are enabled by the selection z . Under appropriate restrictions on the form of A_i ’s, each A_i can be regarded as separately resolving the current global selection z , by nondeterministically choosing one of the possible next-states. This process of alternate *selection* and *resolution* forms the basis of the *s/r model* of coordinating concurrent processes [161] and has been illustrated for modeling of communication protocols [157] and other discrete-event systems. By including a “pause” selection, which enables a self-loop in every state, asynchronous delays can also be modeled within the *s/r model*. The *S/R language* [164] provides a syntax for using the *s/r model*.

3.2.2.2. Verification framework. A specification is typically represented by a deterministic L -automaton T (called a “task” in Kurshan’s terminology) and an implementation by a nondeterministic L -process A . Verification is cast in terms of testing for language containment, i.e. testing if $\mathcal{L}(A) \subset \mathcal{L}(T)$ ¹⁵. The complexity of performing this test has been shown to be $O(|M_A \otimes M_T| \cdot |Atoms(L)|)$ [166], i.e. it is linear both in the size (number of transitions) of A and in the size of T .

Kurshan uses additional techniques to further reduce complexity of the verification problem. One of these is based on the fact that given an ω -regular language $\mathcal{L}$, a finite number of deterministic L -automata $\Gamma_1, \dots, \Gamma_n$ can be found such that $\mathcal{L} = \cap_{i=1}^n \mathcal{L}(\Gamma_i)$ [165]. (Such a property is *not* true of Büchi automata.) Thus, to verify $\mathcal{L}(A) \subseteq \mathcal{L}$ one may decompose $\mathcal{L} = \cap_{i=1}^n \mathcal{L}(\Gamma_i)$ and verify the smaller cases $\mathcal{L}(A) \subset \mathcal{L}(\Gamma_i)$ for all $i = 1 \dots n$, each such case of complexity linear in the size of A and in the size of Γ_i . Another effective technique developed by Kurshan is the use of *reductions*, which form the basis of both complexity management and hierarchical verification. We describe them in the next section; details can be found in Kurshan’s presentations [165, 166].

3.2.2.3. Reductions. A reduction is a validity-preserving transformation from a complex domain to a simpler one, such that validity of a (simple) assertion in the simpler domain implies validity of the corresponding assertion in the complex domain. In a verification problem, an implementation description A can be transformed to A' using a reduction transformation such that

$$\mathcal{L}(A') \subset \mathcal{L}(T') \Rightarrow \mathcal{L}(A) \subset \mathcal{L}(T) \quad (12)$$

where T' is the corresponding reduction of the original specification T . In this manner, complexity of the original language containment problem can be significantly reduced by working in the simpler domain consisting of A' and T' . A reduction transformation is potentially more powerful than equivalence or minimization, since it can be chosen *relative* to a particular specification T .

Reduction transformations also provide a basis for a methodology of hierarchical verification. If A and A' are implementation descriptions at different levels of the abstraction hierarchy, then the inverse of a reduction transformation from A to A' can be seen as a refinement transformation from A' to A . In this context, when a property T' is verified of A' at one level, it implies that the image property T remains true of A at the lower (more detailed) level. Successive refinements during the top-down development of a design can use inverse reduction transformations in this manner to ensure that verification performed at any level holds for all lower levels. This helps in introduction of correctness properties at appropriate levels of abstraction, thereby simplifying the task of overall verification.

Kurshan uses a *homomorphic* reduction transformation from A to A' , with its basis rooted in a language homomorphism relating their respective languages. The latter, in turn, is based on a Boolean algebra homomorphism¹⁶ on their respective Boolean algebras L and L' . It has been proved that if two homomorphisms $\Phi: A \rightarrow A'$ and $\Psi: T \rightarrow T'$ (which agree on their underlying Boolean algebra homomorphisms) satisfy certain conditions, then condition (12) holds. These conditions for $\Phi(\Psi)$ can be checked algorithmically in time linear in the size of $A(T)$. To further reduce this complexity, a homomorphism $\Phi: A \rightarrow A'$ can be constructed as a product of component homomorphisms $\Phi_i: A_i \rightarrow A'_i$ on the respective components of A and A' .

3.2.2.4. Examples and related work. An automated facility called COSPAN has been developed to implement the verification techniques and successive refinement design methodology described above [163]. It takes as input a system description expressed in S/R syntax that is compiled into C code, followed by a state-space analysis conducted to check for arbitrary user-given ω -regular properties. Given a homomorphic reduction provided by the user, it can also check its validity and then conduct the language containment test in the reduced domain. It has been used effectively in practice, e.g. to formally develop and verify the layered design of a communications protocol used by one of the development projects at AT&T [163]. Other examples of using reductions for modeling and verifying hardware have also been shown, e.g. verifying an arbiter circuit through a four-level hierarchy formed of transistor-level, gate-level, flip-flop level, and arbiter-level descriptions [160] and a semi-algorithmic method for extraction of finite-state circuit models (from an analog description of the circuit) such that correctness of circuit behavior at the model level is preserved in the

analog domain across a continuous range of circuit parameters [168].

Another interesting direction that this research has taken has been in extending its applicability to verification of non- ω -regular properties, e.g. reasoning about induction and recursion. A limited form of induction has been shown to work within the existing automata framework [167]. Essentially, an “invariant” process is used to perform the inductive step along the partial order provided by the language containment relation. Though this method is general and has been shown to apply to other partial-order relations, a limitation is that an invariant may be hard to find, since it has to represent the joint behavior of an arbitrary number of processes. (A similar method using network invariants in the context of a general process theory was independently proposed by Wolper and Lovinfosse [170].)

3.2.2.5 Strengths and limitations. One of the greatest strengths of Kurshan’s approach is its use of reductions both to control the complexity of state-space analysis and to provide a basis for hierarchical verification. Since most techniques based on state-space analysis suffer from the problem of state-explosion, i.e. an exponential increase in the number of states with an increasing number of components, complexity management becomes a critical issue in practice. This is especially so since formal verification is expected to work on large problems that are beyond the reach of traditional simulation methods. Devoting effort to develop an underlying semantics that supports reduction methods has potentially many advantages, as has been demonstrated by the work with COSPAN. Also, a method of hierarchical verification that allows stepwise refinement of a specification allows much larger systems to be handled in practice.

Another strength of this approach is its powerful modeling paradigm. In particular, the modifications to the standard ω -automata and Moore machine model definitions—algebraically structured inputs, nondeterministic outputs, and special acceptance conditions—have several benefits. The structuring of the inputs in the form of a Boolean algebra forms the basis of both the theory of homomorphic reductions and of a decomposition method for L -processes (whereby the underlying transition matrix of a product L -process is viewed as a tensor product of the matrices of its component L -processes). Nondeterminism is a useful abstraction mechanism for modeling hardware. A controlled form for its representation can provide the conceptual simplicity it affords, without having to sacrifice efficiency of analysis. Within the automata-theoretic approach described above, nondeterministic outputs help in modeling uncertainty in a closed system of coordinating processes, allow modeling of asynchronous delays and behavior of processes in continuous time. In addition, nondeterminism serves a role in the hierarchical development of a design—details to be added at a lower level are typically modeled as nondeterministic choices at a given level of the abstraction hierarchy. Finally, the form of the acceptance conditions used for an L -automata allow decomposition of an arbitrary ω -regular property into an intersection of the languages of (potentially less complex) L -automata. They also allow easy

specification of safety and liveness properties [166].

Though the approach described above has a strong theoretical basis with due regard to complexity issues, its application in practice is sometimes limited by the fact that the burden of providing a reduction transformation lies with the user. In order to realize the full potential of the system, some means of using its reduction mechanism is required. It is not always obvious which transformation works best, though the automated facility to check its validity does help in exploring different options.

3.2.3. Trace conformation. Another language model that has been used to describe circuits is called trace theory. Trace theory models a system's behavior as a set of traces, each of which is a sequence of events. It has been successfully used to model and specify various kinds of asynchronous systems, e.g. Hoare's CSP [56], delay-insensitive circuits [177, 178]. Its application to verification of speed-independent circuits was developed by Dill [173, 174] and is summarized in this section. (Technically, delay-insensitive circuits are different from speed-independent circuits—the former operate correctly under all possible finite delays in the gates *and* wires [178], whereas the latter are supposed to operate correctly under all possible finite gate delays only [172], i.e. wire-delays are assumed to be zero.)

3.2.3.1 Modeling speed-independent circuits. The transitions on the input and output wires of a circuit correspond naturally to events of a trace-theoretic model, i.e. a trace is a sequence of wire symbols, each symbol representing a transition on the corresponding wire. More formally, Dill models a speed-independent circuit as a *prefix-closed* trace structure $T = (I, O, S, F)$ where

- I and O correspond to the input and output wires, $I \cup O = A$,
- $S \subset A^*$ is a regular set of successful traces, representing the circuit's behavior on legal inputs, and
- $F \subset A^*$ is a regular set of failure traces, representing the circuit's behavior on undesirable inputs.

The prefix-closed part refers to the sets S , and $P = S \cup F$ (called set of possible traces) being prefix closed, i.e. all prefixes of a trace in the set also belong to the set. (This is a technical requirement for which details can be found in Dill's thesis [173].) Trace structures for simple components of circuits can be constructed by explicitly describing the finite-state automata that accept the regular sets S and F . For example, the state-transition graph for a Boolean-Or gate with inputs a, b and output c is shown in Figure 8. The set of accepting states for the associated F -automata consists of the single state marked FF , and that for the S -automata consists of all other states. Note that transitions corresponding to hazards, i.e. those where an input changes without enough time for the output to change, lead to the state FF and thus to a failure trace.

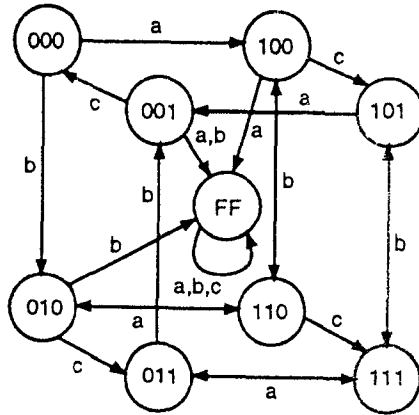


Figure 8. State-transition graph for a Boolean-Or gate.

Trace structures for complex circuits are hierarchically constructed from those for simpler components, by using the operations of hiding, composition, and renaming. Hiding internalizes some output wires, thus making them invisible, and is implemented using a delete operation on traces. Composition of two traces can be defined when their output sets are disjoint, and is defined in terms of an inverse deletion and intersection. Renaming is simply a substitution of new wire names for old ones.

3.2.3.2. Verification framework. Trace structures are used for describing both implementations and specification of speed-independent circuits. The notion of verification is based on the concept of a *safe substitution*, i.e. a trace structure is regarded as a (correct) implementation, if it preserves the correctness of a larger *context* when substituted for the specification in that context. Formally, a context is defined to be a trace theory expression with a free variable, denoted by $\mathcal{E}[]$. A trace structure T is said to *conform to* another structure T' , denoted $T \preceq T'$, if for every context $\mathcal{E}[]$, if $\mathcal{E}[T']$ is failure-free, then so is $\mathcal{E}[T]$. (A trace structure T is failure-free if its failure set F is empty).

The above definition of trace conformation is not directly operable, since it is not possible to test an infinite number of contexts. However, a worst-case context (called the mirror of T') can be demonstrated, such that $T \preceq T'$ holds if and only if the composition of T and the mirror of T' is failure-free (or $S' = \emptyset$). Intuitively, the mirror of a trace structure represents the strictest environment conditions that the trace structure is expected to operate correctly under. Therefore if an implementation operates correctly when composed with the mirror of the specification, then it is a safe substitution under all environments.

The theory outlined above forms the basis of an automated verifier that can verify the conformation of an implementation with respect to a specification,

the latter being in a special canonical form that makes it easy to obtain its mirror structure. It has been demonstrated on several examples, including one where a bug was found in the published design of a distributed mutual-exclusion circuit [176].

3.2.3.3. *Strengths and limitations.* The main strength of the approach described above lies in its ability to perform hierarchical verification, i.e. specifications at one level of the abstraction hierarchy serving as implementation descriptions for higher levels. This is made possible by (1) using the same formalism, i.e. trace structures, for representing both the specification and the implementation, and (2) development of appropriate hierarchical operations for traces, i.e. hiding, composition, and renaming.

An obvious limitation is its need for explicit construction of a state-transition graph associated with the composite trace structure. Though the automated procedure constructs states only as needed, and in most incorrect cases manages to isolate a failure by examination of only a few states, yet it may have to explore the whole graph in the worst case. Also, the theory in terms of prefix-closed trace structures can handle specification of safety properties only. Extensions have been developed to handle liveness and general fairness properties, but they are not practically implementable [173].

3.3. *Hybrid formalisms*

In the last two sections we have described approaches that formulate specifications using logic and language/automata theory, respectively. In the logic case, a specification is expressed using some kind of logic appropriate for the property being verified. Typically, verification proceeds either by deductive reasoning with the logic descriptions of the implementation and the specification (i.e. theorem-proving) or by showing that the implementation provides a semantic model for the specification (i.e. model checking). In the language/automata case, a specification is represented as a language (or an automaton) and so is the implementation. Verification typically consists of testing language containment or equivalence.

Though these two approaches to verification may seem unrelated at first, in fact it is quite the opposite, due to the special relationships that exist between various kinds of logics, languages, and automata. Hybrid methods exploit this relationship in order to gain from relative advantages of both. Logic is naturally suited to expressing specifications. For example, temporal logic provides special operators like $\Box$ (Always) and $\Diamond$ (Eventually) to explicitly reason about time. However, logics are not always easy to reason about. Slight variations in the semantics can require vastly different deductive reasoning methods, model checking procedures, etc. On the other hand, automata/language theory includes classic algorithms that are very resilient to minor modifications. Also, automata/language theory is more widely understood, because of its applications in virtually all branches of

computer science and engineering today.

In the remainder of this section we describe hybrid approaches that combine the advantages of specification in logic and analysis in the automata/language theory. More specifically, a logic specification is first converted to an equivalent automata description, followed by verification in the latter domain. As for the opposite direction, some recent work has been done in this area, e.g. Loewenstein casts a finite-state automata equivalence problem in terms of an HOL theorem, to be proved within an HOL theory developed for reasoning about such automata [183, 184]. However, since theorem-proving is generally harder than verification with automata, such approaches are not very common. They do offer the advantage of not having to work with explicit state-transition graphs as most automata techniques need to do.

3.3.1. LTTL and finite-state automata. In most tableau-based techniques for checking satisfiability of propositional temporal logic formulas [65, 85, 129, 189], a formula is decomposed into two parts—one that should hold in the present state and another that should hold in the next state (along with any *eventualities* that need to be satisfied). This decomposition forms the basis of a semantic tableau construction, such that the tableau for a formula encodes all possible models of the formula. The decompositions for propositional LTTL formulas are summarized in Table 2, where the next-state formulas are those preceded by the next-state symbol $\bigcirc$. The formula within the curly braces $\{ \}$ indicates eventualities, e.g. in the decomposition of $\Diamond\phi$, the requirement that ϕ should be eventually satisfied is indicated by adding $\{\phi\}$ to the next-state part.

Table 2. Decomposition of LTTL modalities.

Formula	Decomposition
$\Box\phi$	$\phi \wedge \bigcirc\Box\phi$
$\Diamond\phi$	$\phi \vee (\neg\phi \wedge \bigcirc(\Diamond\{\phi\}))$
$\bigcirc\phi$	$\bigcirc\phi$
$\phi U \psi$	$\psi \vee (\phi \wedge \neg\psi \wedge \bigcirc(\phi U \psi))$

This tableau construction forms the basis of a verification technique used by researchers at the University of Tokyo and the Fujitsu Labs in Japan. As described by Fujita, Tanaka, and Moto-oka [182], they recursively decompose an LTTL formula according to Table 2. This process results in a state-transition graph with states labeled by formulas that they need to satisfy, and transitions labeled by Boolean combinations of atomic propositions. The decomposition is

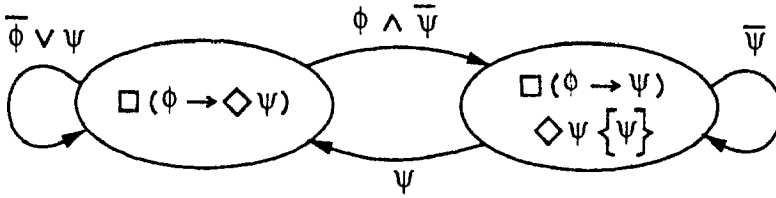


Figure 9. State-transition graph for $\Box(\phi \rightarrow \Diamond\psi)$.

recursively repeated until all states in the graph fall in a loop and no more new states are needed. For example, the formula $\Box(\phi \rightarrow \Diamond\psi)$ can be represented as shown in Figure 9 [182].

This state-transition graph (with eventualities) is, in some sense, a finite-state automata (FSA) equivalent of the corresponding LTL formula. Thus, given an FSA description of an implementation, traditional automata techniques can be used to verify that it satisfies an LTL specification. Fujita et al. actually construct the product of the automata for the implementation and the automata for the *negation* of the specification. This product state-transition graph is then checked for the presence of an infinite path (that satisfies the eventualities), which, if found, proves that the implementation does not satisfy the specification. (Note that this is equivalent to checking the emptiness of the language represented by $L(Imp) - L(Spec) = L(Imp) \cap \overline{L(Spec)}$.) They describe algorithms for both forward and backward exploration of the product state-transition graph [185].

This verification facility has been developed within the larger context of a unified CAD framework supporting hierarchical design of synchronous circuits. The system uses hardware description languages (called DDL and HSL) to represent design implementations at the gate and register-transfer levels. These descriptions are translated to finite-state automata representations by tracing causality from effects to causes [185]. Earlier implementations of the system used the logic programming language Prolog [41] to automatically perform the search for infinite paths using its inbuilt mechanisms for backtracking and pattern-matching [181]. Due to reasons of inefficient performance, a change was made later to using Time Extended BDDs for internal representation, which are basically Bryant's BDDs [12] extended to express states and eventualities [180].

Several other tools, e.g. a graphics editor for handling timing diagrams, a rule-based temporal logic formula generator, and a simulator, have been developed to facilitate the task of specification and verification. Techniques have also been developed to efficiently use the approach in practice, since its complexity is limited by tautology-checking of Boolean formulas, which is an NP-hard problem [20]. Filtering of a design description is done to isolate the parts that are actually needed for verification. Since specifications are frequently expressed as conjunctions of smaller conditions (especially when an interval-

based methodology for specification of sequentiality is used [180]), this results in substantial savings in the size of the implementation to be considered. Also, state-transitions are frequently cached for future reference, thereby improving performance.

The main contribution of this approach lies not so much in its theoretical basis, but in its practical approach of providing a unified, efficient, and usable framework for specification, verification and synthesis of circuits.

3.3.2. Temporal logics and Büchi automata. Vardi and Wolper presented a more general method than that described above, for converting a temporal logic model checking problem to a purely automata-theoretic problem [187] (similar to their work on modal logics of programs [188]).

The essential idea of their approach is simple. A PTL (propositional linear time temporal logic) formula is interpreted with respect to a computation consisting of an infinite sequence of states. Since each state can be completely described by a finite set of propositions, a computation can be expressed as an infinite word over the alphabet consisting of truth assignments to these propositions. (In the case of branching time logics, a similar argument can be made, except that computations are now expressed as trees, not sequences, of truth assignments). A constraint on the computations, as is placed by a PTL formula, directly translates to a constraint on the form of these infinite words. Thus, given any PTL formula, a finite-state automaton on infinite words can be constructed that accepts the exact set of sequences that satisfy the formula [130]. In this sense, temporal logic formulas can be viewed as *finite-state acceptors* of infinite words.

On the other hand, a finite-state program, which constitutes a model for the temporal logic formulas, can be viewed as a *finite-state generator* of infinite words (the computations). The model checking problem, that every computation of a program P should satisfy a formula ϕ , therefore reduces to the problem of verifying that the language of infinite words generated by the program $L(P)$ is *contained* in the language accepted by the formula $L(\phi)$, i.e. $L(P) - L(\phi)$ is empty.

Given an arbitrary PTL formula ϕ , Vardi and Wolper give an effective construction for the corresponding finite-state acceptor, which is in the form of a Büchi automaton. (Büchi automata are named after Büchi who first studied them [158]). Formally, a Büchi automaton over a finite alphabet Σ is of the form $M = (\Sigma, S, s_0, T, F)$, where

- Σ is a finite alphabet,
- S is a finite set of states,
- s_0 is the initial state,
- T is a transition relation—a transition from state s_i to state s_j on input letter $\alpha_i \in \Sigma$ is denoted as $(s_i, \alpha_i, s_j) \in T$, and
- $F \subseteq S$ is a set of final states.

A run of the automaton on an infinite word $\alpha = \alpha_0\alpha_1\dots$ is a sequence of states $\sigma = s_{\sigma_0}s_{\sigma_1}\dots$ such that $s_{\sigma_0} = s_0$ and $(s_{\sigma_i}, \alpha_i, s_{\sigma_{i+1}}) \in T$ for all $i \geq 0$. A run is called accepting if some state $s \in F$ occurs infinitely often in the run.

The size of the constructed Büchi automaton, equivalent to a formula ϕ , is exponential in the length of ϕ . The automaton actually consists of the product of two automata—one called the *local* automaton and the other called the *eventuality* automaton. The former consists of states formed from subsets of subformulas of ϕ that have no local (propositional) inconsistencies. The latter is responsible for checking that eventualities (of the form $\Diamond\phi$ and $\phi U\psi$) are eventually satisfied. The exponential number of states results due to the fact that all subsets of subformulas of ϕ and their negations (also called closure of ϕ) have to be considered.

Using this construction and the previously known results for the emptiness problem for Büchi automata (i.e. it is solvable in linear time [87, 179] and is logspace complete for co-NLOGSPACE [186]), they show that the general model checking problem for program P and a PTL formula ϕ is of time complexity $O(|P| \cdot 2^{|\phi|})$ and space complexity $O((\log|P| + |\phi|)^2)$. This approach is extended to deal with other variants of temporal logic, e.g. incorporating past-time operators by extending the equivalent Büchi automaton construction. It can also include fairness assumptions of the program by using special Büchi acceptance conditions. Finally, it can be easily extended to handle model checking with Extended Temporal Logic (Section 3.1.5), the major difference being that the Büchi automaton constructed for an ETL_r formula is of size exponential in the square of the length of the formula, thereby resulting in a higher complexity for the overall model checking.

The biggest contribution of this approach is the unifying theoretical framework that allows handling of temporal logic model checking problem in general, without entrenching it in the syntactic and semantic details of a particular logic (as is common in most other tableau-based techniques). It also allows extensions to be incorporated in a unified manner. Apart from the conceptual simplicity of the clean framework, it eases the task of assessing complexity of the associated verification problem and isolates the difficult part—construction of the finite-state acceptor—from the other simple ones.

3.4. Relative expressiveness of various formalisms

Having described the various formalisms in some measure of detail, a natural question we face is this: What are their expressive powers relative to each other? We have already mentioned some results regarding LTTL and BTTL logics (Section 3.1.4). In this section we summarize major results regarding other formalisms—various modal logics, standard predicate logics, and automata/languages—first for linear structures and then for branching structures.

3.4.1. Linear structures. Kamp showed that the LTTL language with only (F, X) operators (called $L(F, X)$), is expressively incomplete with respect to a first-order predicate language of linear order¹⁷ [94]. He also showed that with the addition of U and its past-time dual S (*Since*), the language (called $L(F, X, U, S)$) becomes as expressive. The addition of past-time operators was also suggested independently by other researchers [63, 95, 102], on grounds that it simplifies modular specifications as well as safety properties. However, it was proved by Gabbay et al. [89] that the future fragment (with initial semantics) is itself expressively complete, and is therefore equivalent to having both past-time and future-time operators. They also present a deductively complete proof system for linear orders, i.e. every temporal formula that is valid in all linearly ordered domains is provable in their system.

Wolper's Extended Temporal Logics, with different notions of automata acceptance, are expressively equivalent to Büchi automata [130]. Büchi showed that Büchi automata are equivalent to ω -regular languages, which are equivalent to the monadic¹⁸ second-order predicate logic of linear order (called S1S) [158]. Mu-Calculus has also been shown equivalent to S1S by Park [136]. Thus, for linear structures, $ETL \equiv \text{Büchi automata} \equiv \omega\text{-regular languages} \equiv \text{S1S} \equiv \text{Mu-Calculus}$.

3.4.2. Branching structures. Schlingloff has compared the expressive powers of several logic languages on branching structures with respect to ω -tree languages (languages on infinite trees) [122]. He proves that temporal logic is equivalent in expressive power to first-order predicate logic with b successors (where b is the branching factor of the underlying ω -tree). He also shows that ETL on branching trees is at most as expressive as weak second-order logic, and conjectures that they may be equivalent. Finally, it has been shown by Niwinsky that propositional Mu-Calculus is equivalent to monadic second-order predicate logic with b successors (also called SbS) [134]. (Since other technical details are beyond the scope of this article, we refer the interested reader to the references cited.)

4. A classification framework

Having described the major research efforts in formal hardware verification in some measure of detail, we present a big-picture view in this section. As alluded to previously, we regard the implementation representation, the specification representation, and the form of proof method employed to be important dimensions along which approaches differ. These three axes form the basis of our classification. Note that in Section 2 we have already described the relevant issues involved while choosing a particular form for each of these. This section presents, in some sense, a summary of the *actual* choices that have been made by different researchers.

For the implementation representation axis, the popular forms used to describe

hardware fall under the following categories:

- logic description (L)—hardware described using terms and relations in a particular logic (with functional representations also included in this category)
- state-transition graph (STG)—a graph description of hardware, with nodes representing states and labeled edges representing transitions labeled with inputs/outputs; traditionally associated with operational machine models
- automaton (A)—a description in terms of states and transitions along with acceptance conditions; traditionally associated with languages and grammars
- trace structure (TS)—a behavioral description in terms of traces (values on externally visible input/output ports)

The specification representation axis consists of the following categories:

- logic description (L)—expressed as formulas in a particular logic; typically used to specify functional/behavioral properties
- temporal logic property (TL)—though subsumed by the above category, we have chosen to consider this separately due to its specialized nature; typically used to specify behavior over time
- state-transition graph (STG)—typically used for specification of a machine model at a different hardware abstraction level
- automaton (A)—used to specify behavior in terms of the language accepted by the automaton; (we also include here those approaches that use operators in logic based on automata semantics)
- trace structure (TS)—used to specify behavior in terms of the allowed traces

Finally, the following forms of proof method have been popularly used:

- theorem-proving (TP)
- model checking (MC)
- machine equivalence (ME)
- language containment/equivalence (LC)
- trace conformation (TC)

The appropriate classification of the works of various researchers (and research groups) is shown in Figure 10. Each row label represents an item of the implementation representation axis, while each column label represents an item of the specification representation axis. A box in this two-dimensional array is marked with various approaches (represented by mnemonic labels) that use the corresponding representations. The proof method used is indicated along with each approach. Table 3 shows the correspondence between the mnemonic labels used to represent the approaches and the references cited earlier. We also summarize in Figure 11 the hardware level(s) that each approach has modeled using the associated implementation representations.

Table 3. List of labels and references for Figures 10 and 11.

Label	References
Barrow	[39]
BM	[32, 33, 35]
Bull-1	[150, 151, 152, 156]
Bull-2	[81]
Burch	[171]
CGK	[127]
CMU-1	[69, 70, 72, 73, 75, 77, 82]
CMU-2	[67]
CMU-3	[71]
Cosmos	[10, 11, 14, 16]
Cospan	[160, 163, 165, 166, 167, 168]
Dev	[154]
Dill-1	[173, 174]
Dill-2	[175]
Fujita	[180, 181, 182, 185]
Hol	[40, 42, 45, 48, 52, 55, 57, 59]
LP	[101]
MPB	[65, 105, 106, 107, 116]
VW	[187]
Weise	[147]
Wolper	[129, 130]

With the framework of Figure 10, it is easy to see the broad similarities and differences between various approaches, as well as to roughly estimate their relative strengths and weaknesses. For example, the HOL (Hol in figure) and Boyer–Moore (BM) verification approaches work with different logics, but are similar in that they both use logic descriptions to represent the implementation and the specification, and use theorem-proving techniques. On the other hand, these approaches are significantly different from that of Clarke et al. (CMU-1), which uses model checking of temporal logic formulas with respect to state-transition graphs. Both the HOL and Boyer–Moore approaches derive their main strengths from a natural specification style with logic, combined with compositional and hierarchical methods allowed by theorem-proving techniques. Their weaknesses

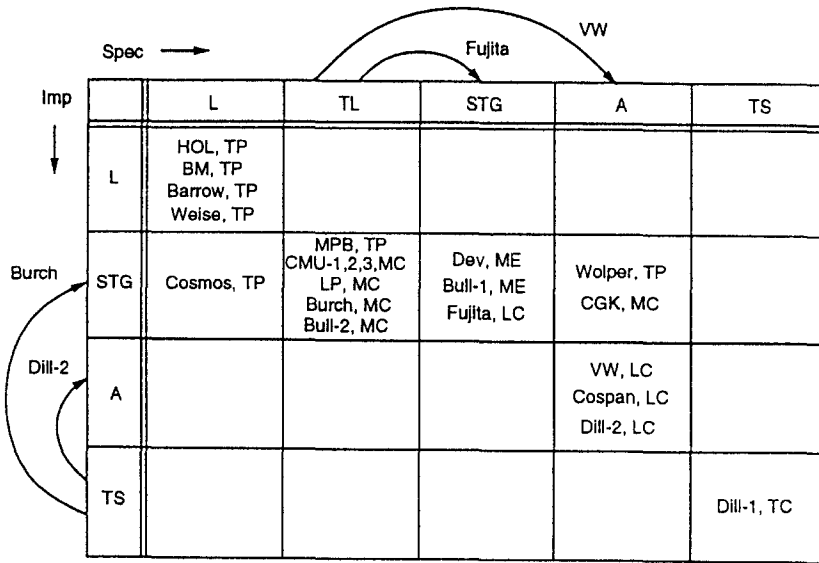


Figure 10. Classification of hardware verification methods.

lie in poor circuit models and a high complexity of analysis with theorem-proving. With the CMU-1 approach, however, the main strength consists of the efficiency of model checking, with the major weakness being its reliance on an explicit construction of a state-transition graph. (Approaches that use an STG for the underlying implementation representation are, in general, likely to suffer from state-explosion. We present some general conclusions regarding approaches that use theorem-proving vs. model checking in Section 5.1. A useful comparative study of various theorem-provers has been presented by Stavridou et al. [217].)

Another interesting observation, as can be seen in Figure 10, is that a large number of approaches use logic (including temporal logic) to represent specifications, whereas a state-transition graph is a popular choice for an implementation representation. This reflects largely on the suitability of logic as a natural formalism for specification, and on the suitability of an operational model for representation of an implementation.

We have also indicated some arcs in Figure 10, which join boxes along the same axis. These arcs indicate relationships between different, though equivalent, forms of representation. Note that hybrid approaches (as described in Section 3.3) fit this category, since they convert specifications expressed in logic to equivalent automata form. Thus, the Vardi-Wolper (VW) approach is shown as an arc from TL to A (temporal logic to automata), and the approach used by Fujita et al. (Fujita) as an arc from TL to STG (temporal logic to state-transition graphs). We have examples of equivalent representations along the implementation axis

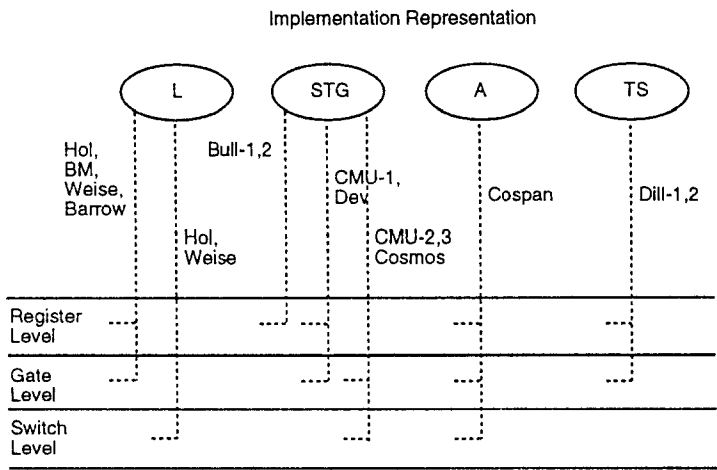


Figure 11. Representation of hardware levels.

also. Dill converts trace structures to equivalent automata descriptions and then uses language containment proof techniques to perform verification of real-time properties [175] (also see Section 5.2)—indicated as an arc from TS to A. Similarly, Burch converts trace structures to state-transition graphs and then performs usual CTL model checking [171]—indicated as an arc from TS to STG. The motivation for using equivalent representations is to combine the advantages of a natural representation style (e.g. trace structures) with simpler analysis methods (e.g. those provided by automata and state-transition graph representations).

A natural question that arises in the context of Figure 10 concerns the boxes that are empty. Do the empty boxes really indicate mutual incompatibility of the corresponding representations, or is it just the case that these have not been explored yet? Also, there has been an increasing interest in the area of equivalent representations. It would be interesting to see which new arcs, if any, are explored in the future. As for Figure 11, it can be seen that most approaches work at higher (register/gate) levels of hardware abstraction. More work needs to be done to either extend these to effective switch-level models, or to integrate them with other approaches that can handle low-level circuit effects.

5. Recent trends and future directions

As research in formal hardware verification has progressed rapidly through the recent years, some trends have begun to emerge. We describe these briefly in this section and also point towards directions that are likely to be the focus of

increased activity in the next few years.

5.1. Theorem-proving versus model checking

By far the biggest polarization trend perceived within the verification community that uses logic has been with respect to the proof methods employed. One group of independent researchers has advocated the use of theorem-proving while the other group has believed firmly in model checking. There has been a tradition of rivalry between these two groups, each touting its own advantages against the disadvantages of the other. There are, in fact, basic differences between these two approaches that have significant implications.

Theorem-proving, by its very nature, is a deductive process. This raises both theoretical and practical concerns regarding management of its complexity. Automation can be, and has been, provided to some degree (e.g. in the form of rewrite rules, specialized tautology-checkers, etc.). However, most of the “automated” theorem-provers available today are semi-automated at best, in that they require some form of human interaction to guide the proof searching process. In effect, theorem-“provers” are more like theorem-“checkers” in most cases. On the other hand, theorem-proving systems are very general in their applications. Logic allows representation of virtually all mathematical knowledge in the form of domain-specific theories. This allows formalization of both sophisticated hardware systems (e.g. floating-point arithmetic units) as well as any interesting property that one might want to specify about them. The ability to define appropriate theories, and reason about them using a common set of inference rules, provides a unifying framework within which all kinds of verification tasks can be performed. In fact, it is for this very generality that theorem-proving systems pay the price of increase complexity.

Model checking, in contrast, is a relatively modest activity. Since attention is focused on a single model, and there is no need to encode incidental knowledge of the whole world, the complexity of this task is much more manageable. In most cases, a clear algorithm can be provided that can be made completely automatic. These algorithms usually have a counter-example mechanism also. This feature comes in handy for debugging purposes, and is important from a practical standpoint. The drawback of such systems is that they are not general in the way that theorem-provers are. A model checking verification system will work only for the kind of logic and models that it is designed for. For example, a model checking algorithm for evaluating the truth of CTL formulas with respect to Kripke structures is very different from one that applies to LTTL formulas with respect to fair transition systems. Some efforts have been made towards unification of model checking ideas in terms of automata theory (Section 3.3.2), but the problem largely remains domain specific.

Apart from the issues of complexity and generality, there are other concerns also, more specifically related to the field of hardware verification, that tend to

divide the two groups. One such concern is regarding the ability to perform hierarchical verification. As has been mentioned earlier, in order to manage the inherent complexity of hardware systems encountered today, hierarchical verification has become a very desirable feature in a verification approach. Techniques that do not provide for this fare badly on large, complex hardware designs. Most theorem-proving approaches find it easy to incorporate hierarchical methods, due to the natural abstraction mechanisms available for representation of hardware as terms (and their combinations) in logic. The same cannot be said of traditional model checking approaches. Most of these use non-hierarchical, state-based descriptions of hardware. An increase in the number of hardware components results in a combinatorial explosion in the number of global states, thus leading to poor performance on large problems. This problem has been alleviated to some extent by more recent efforts (Section 3.1.4.5.4 and 3.1.4.5.5) that include use of symbolic methods and abstractions. Recent results on modular verification also provide a platform for further improvement in supporting hierarchical verification within a model checking framework.

Another concern stems from the kind of models used to represent hardware. Theorem-proving approaches typically use a structural representation of hardware, with predicates in a logic representing hardware components. Thus, they are suitable for reasoning about functional specifications (by developing a theory of circuit functions within the proof system) and for reasoning about parameterized descriptions (by using induction methods for proofs). On the other hand, model checking approaches typically use a state-based hardware description that is oriented towards expressing its behavior (i.e. what atomic propositions are true in each state) rather than its structure. These models allow relatively easier formalization of issues like concurrency, fairness, communication, and synchronization. Thus, theorem-proving approaches have traditionally performed better at verification of functional specifications of datapaths, while model checking approaches have been better at reasoning about the control aspects of a circuit. Theorem-proving approaches that do attempt to deal with temporal aspects, e.g. axiomatic proof systems for temporal logics, have not been very successful in practice due to their inherent complexity. However, successful attempts have been made for datapath verification within a model checking context through use of data abstractions (Section 3.1.4.5.4) and for a limited form of induction within the model checking/language-containment framework (Section 3.1.4.5.4/Section 3.2.2).

These differences notwithstanding, there has recently been an increased awareness of the relative merits of both approaches. Several researchers are now exploring ideas for combining the two in order to enhance the advantages achievable by either one alone. Aside from the fact that better circuit models are needed with each approach (in order to improve the quality of verification), it is generally regarded that model checking can more easily deal with low-level circuit details and is also more efficient, while theorem-proving is better for higher-level reasoning in a more abstract domain. By applying the better

technique to subproblems of any given verification problem, one can hope to effectively increase the size and complexity limits of problems that can be handled in practice. This would also bring closer the ultimate goal of complete verification of a given hardware system.

5.2. Real-time verification

An interesting application of hardware verification research has been in the area of real-time systems verification. Interest in real-time systems has grown rapidly in the past few years as these systems have found application in numerous areas. Due to the critical nature of typical applications, their verification has become an active area of research. In this section, we give pointers to extensions of hardware verification techniques (in roughly the same order of presentation as in Section 3) that have been found useful in the verification of real-time systems.

Jahanian and Mok developed a restricted form of first-order logic, called Real-Time Logic (RTL), with the objective of specification, verification, and synthesis of real-time systems [201]. They use a special *occurrence* function to keep track of the n th occurrence of an event, which leads to a fairly expressive logic. The biggest drawback of RTL is that it is provably undecidable [202]. However, decision algorithms have been developed for certain subclasses of the full logic, within a framework based on mode-charts [200, 203]. Barbacci and Wing suggested the combined use of a hardware description language called Larch and RTL to specify both functional and timing properties [194]. Higher-order logic has also been combined with RTL by MacEwen and Skillicorn [206] to develop Distributed RTL (DRTL)—a specification language that exploits the modularity and hierarchy provided by HOL to develop distributed systems specifications.

Several researchers have extended the basic temporal logic formulation in order to reason about time quantitatively. Different representation mechanisms have been used to introduce the notion of time, impacting both the syntax and the semantics of the associated logic. The popular mechanisms include use of bounded-time temporal operators, introduction of an explicit clock variable, and temporal quantification of time variables (by binding variables to the time of the current temporal context). Bernstein and Harter used the notion of time-bounded eventuality (for temporal implication), and an axiomatic proof lattice methodology to prove real-time assertions [195]. Koymans, Vytupil, and de Roever also used timed-operators, e.g. $U_{=t}$ (*Until* in real-time t), within the basic LTTL framework with past-time operators [95]; more recent work on this logic, called Metric Temporal Logic (MTL), has been presented by Koymans [204]. Ostroff used an extension of Pnueli's fair transition systems to obtain semantic models called Timed Transition Models (TTMs) [207, 208]. Constant upper and lower time-bounds are associated with the transitions of a TTM, with reference to a fictitious global clock represented as a special variable. Specifications are expressed in Real-Time Temporal Logic (RTTL), which allows explicit reference to the

clock variable, and verification is performed by reachability-graph analysis. Two extensions of LTL proposed by Pnueli and Harel [209]—Global Clock Temporal Logic (GCTL, later called XCTL) and Quantized Temporal Logic (QTL)—are similar to (restricted) RTTL and MTL, respectively; decision procedures for the validity problem and the model checking problem for XCTL have been presented recently [199]. Emerson et. al. studied a real-time extension of CTL (called RTCTL), which uses bounded-time temporal operators over a discrete time domain, and described algorithms for model checking and checking satisfiability of formulas [198]. Alur, Courcoubetics, and Dill also considered a similar extension (called TCTL), but over a dense time domain, and presented a model checking algorithm with respect to timed graphs [190]. Alur and Henzinger proposed temporal quantification of time variables (also referred to as freeze quantification), and presented a tableau-based decision procedure and a model checking algorithm for the associated logic called Timed Propositional Temporal Logic (TPTL) [192]. They also showed the expressive equivalence of TPTL (with freeze quantification) and MTL (with time-bounded temporal operators) in a useful comparative study of the complexity and expressiveness of various real-time temporal logics [193]. It has also been shown that the expressive powers of TPTL and XCTL are incomparable [199]. Moszkowski used the interval semantics of ITL itself (Section 3.1.4.7) to reason about real-time properties [112].

The other major influence from hardware verification has been in the area of using traces and automata/languages as the basis for real-time verification. Zwarico and Lee extended Hoare's CSP model [56] by adding timing constraints to the traces describing program execution [210]. Burch also extended traces to incorporate time events (marking passage of discrete time) [196], and is studying conservative approximation of continuous time traces by discrete time traces as part of his thesis work [197]. Dill extended his work on verification of speed-independent asynchronous circuits (Section 3.2.3) to incorporate use of timing assumptions given as constant upper and lower bounds on continuous-time delays between trace events [175]. An interesting feature of his technique is the construction of a Büchi automaton that accepts exactly the set of traces as are allowed by the time-constrained behavior of the implementation. Since specifications are also expressed in the form of Büchi automata, standard automata techniques can be used to verify the language containment relation. (Note: This is an example of using equivalent implementation representations, as mentioned in Section 4.) A similar problem of fixed upper and lower bounded continuous-time delays was addressed by Lewis [205]. Based on these works, Alur and Dill proposed the notion of Timed Büchi Automata (TBA), which accept languages of timed traces (traces with each event associated with a real-time value) [191]. Language containment for TBAs is proved to be undecidable, and the methodology is extended to deterministic Timed Muller Automata (DTMA), for which it is decidable.

As hardware verification techniques improve, especially with respect to verification of timing properties, we expect to see corresponding progress in the area

of verification of both hardware and software real-time systems.

5.3. *Emergence of symbolic methods*

As described in Section 3, symbolic methods for Boolean function manipulation have attracted a great deal of interest amongst researchers and have been used in conjunction with several verification methods. Their use is almost orthogonal to other verification issues, in that the focus is on compact internal data representation without affecting the high-level verification algorithm. This results in an increase in efficiency and practical usefulness of the overall approach.

Boolean formulas (representing circuit outputs) formed from Boolean-valued symbolic variables (representing circuit inputs) have been directly used for automatic functional verification of combinational circuits [12, 13, 15]. Symbolic methods have also been used for Hoare-style verification of finite-state sequential circuits [10, 11, 14]. Typically, the states of a given circuit (implementation) are encoded by symbolic Boolean variables. The circuit's sequential behavior is described in terms of next-state and output functions over these state variables and other variables representing the circuit inputs and outputs. Verification is performed by checking post-conditions of the Hoare-style assertions (specifications), for all transitions corresponding to states and inputs that satisfy the pre-conditions. The use of a symbolic simulator like COSMOS [17] allows multiple transitions to be simulated in one step. It also avoids construction of an explicit state-transition graph for the given circuit.

Extending the ideas from single transitions of a sequential system to multiple transition sequences, these methods have been used to perform automatic state-space exploration also. The states are, again, encoded by symbolic Boolean variables, with the transition behavior described in terms of a next-state function (or a transition relation, using a separate set of next-state variables). The key idea is that a Boolean formula over the state variables implicitly represents a set of states, i.e. those corresponding to valuations of variables that make the formula true. This allows handling of *sets* of states, rather than individual states (as in direct state enumeration approaches), by appropriate symbolic manipulations on the corresponding Boolean formulas. For example, set intersection is performed by conjunction and set union by disjunction. Finding the set of successor states to a given set of states is also accomplished easily by a composition with the next-state function (or by a relational product using the transition relation). Finding the set of all reachable states is accomplished by iteratively finding the successor states till a fixed point is reached. Canonical representations of Boolean formulas (BDDs or TDGs) enable simple syntactic checks to determine Boolean functional equivalence, and also allow efficient graph algorithms for symbolic manipulations in practice.

The main advantage of symbolic (as opposed to enumerative) state-space exploration is that it avoids construction of an *explicit* state-transition graph.

This is useful in handling large systems, especially in tackling the problem of state explosion encountered in most systems consisting of a number of parallel components. It has been applied within the framework of several verification efforts that rely upon exploring finite-state systems. For example, finite-state machine equivalence has been verified by symbolic reachability analysis of the product machine [151, 152]. Another application is in the area of model checking for temporal logics, where each state corresponds to a unique valuation of the atomic propositions (represented by symbolic state variables). By exploiting the fixpoint interpretations of temporal modalities, the set of states that satisfy a given temporal formula can be computed symbolically as a fixpoint of a Boolean functional on state variables [67, 71, 72, 81, 109]. A generalization to fixpoint operators allows symbolic model checking of Mu-Calculus formulas also [131], which provides a unified framework for model checking of various logics. Recent efforts on symbolic methods, specifically BDDs, include

- use of partitioned transition relations for system description, represented by implicitly conjuncted/disjuncted BDDs [212]
- use of additional variables to obtain more compact circuit representations, as in the case of multipliers [211]
- parallel implementations of the basic symbolic manipulation algorithms [216]

5.4. Formal verification and synthesis

A recent trend that is gaining popularity is the use of formal verification not only as a post-design activity, but also its incorporation within the design phase itself. Formal verification can be used to verify the procedures used in automated synthesis programs. It can also be used to verify that correctness-preserving transformations (an essential component of most automated synthesis systems) are indeed correct and produce equivalent representations. This idea is not new—an excellent article on the relationship between verification, synthesis and correctness-preserving transformations was presented by Eveking [214]. However, it is only recently that actual design systems based on formal verification methods have started to make an appearance [213, 215, 218, 219]. We expect to see more of combined synthesis and verification methodologies in the future.

6. Conclusions

We have presented a high-level view in terms of the basic verification problem of the multi-faceted research done in the area of formal hardware verification. It is fairly clear that there does not exist any one method that is appropriate for handling all verification aspects of a complete hardware system. What we have at present is similar to a toolbox of multiple tools, each adept at solving a subproblem of the bigger problem. It would be interesting to see what approaches

could interact and communicate with each other in a useful manner, in order to achieve the goal of a completely verified system.

Another landmark awaited in the maturing of formal hardware verification is its active adoption by the industry. Though successful instances of its industrial applications exist, it is far from being commonly accepted by hardware designers and engineers in the field. The current perception is that a significant insight into the theoretical basis of the verification techniques is needed in order to use them effectively. Since most designers do not have a formal training in logic or automata theory, there is reluctance in using even the few tools that are available. Several efforts can help improve this situation. More work in integrating formal verification tools within the traditional design environment, consisting of synthesis and simulation tools, will help provide a familiar interface to designers. Also, an effort needs to be made through education and training to make the formal methods and their benefits better understood. We hope that our survey will be a useful step in that direction.

Acknowledgments

I would like to thank Allan Fisher for his continued guidance and support during the writing of this article. His constructive suggestions for the organization of the survey, the numerous discussions with him regarding the contents, and his comments on a preliminary draft were all of invaluable help. I would also like to thank Carl Seger, Sharad Malik, and the referees for their critical comments and suggestions for further improvement.

Notes

1. This is usually called a *representation* function in Hoare's work.
2. There is implicit universal quantification of formulas with free variables.
3. Induction is not usually used as a rule of inference with first-order logics.
4. This is with respect to standard models [6].
5. A semantic tableau is a structure that encodes all possible models for a formula.
6. Fixpoint computations for CTL operators are described in Section 3.1.6.
7. This is a little different from the usual notion of *validity*, which means truth of a formula in *all* models.
8. Equivalent to regular grammars; RHS of productions have nonterminal symbols only on the right [155].
9. Büchi automata are described in Section 3.3.2.
10. A relational term P is formally monotone in a predicate variable Z , if all free occurrences of Z in P fall under an even number of negations.
11. These are described in Section 3.3.2.
12. Informally, a Boolean algebra is a set closed under the operations of meet ($*$, conjunction), join ($+$, disjunction), and negation ($\neg$); with a multiplicative

identity 1 and an additive identity 0 [162].

13. Only a simplified version is described here, since a treatment of all technical requirements in the definition is beyond the scope of this article.
14. Atoms of a Boolean algebra are its minimal elements with respect to the ordering $\leq$, defined as $x \leq y$ iff $x * y = x$.
15. For this section, we follow Kurshan's terminology in using the symbol $\subset$ to denote containment, not necessarily *proper* containment.
16. A Boolean algebra homomorphism is a map that is linear with respect to $*$, $+$, and $\neg$.
17. A theory of linear order refers to all formulas that are true when interpreted over domains that are linearly ordered, i.e. the alphabet of the logic includes a built-in dyadic linear order predicate " $<$ " which is not interpreted, unlike the other signature symbols.
18. "Monadic" logic refers to a logic with single-argument predicate symbols.

References

Other Surveys

1. P. Camurati and P. Prinetto. Formal verification of hardware correctness: Introduction and survey of current research. *Computer*, 21(7):8–19 (July 1988).
2. E.M. Clarke and O. Grumberg. Research on automatic verification of finite-state concurrent systems. *Annual Review of Computer Science*, Carnegie Mellon University, Pittsburgh, PA, 2:269–290 (1987).
3. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, J.W. de Bakker, W.-P.de Roever, and G. Rozenberg (eds.), volume 224 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1986, pp. 510–584.
4. M. Yoeli. *Formal Verification of Hardware Design*. IEEE Computer Society Press, Los Alamitos, CA, 1990.

Logic

5. J. Barwise (ed.). *Handbook of Mathematical Logic*. North-Holland, Amsterdam, 1977.
6. D. Gabbay and F. Guenther (eds.). *Handbook of Philosophical Logic*, volumes 1, 2, and 3. D. Reidel, Boston, 1983.
7. W.S. Hatcher. *The Logical Foundations of Mathematics*. Pergamon Press, Oxford, England, 1982.
8. G. Hunter. *Metalogic: An Introduction to Metatheory of Standard First Order Logic*. University of California Press, Berkeley, 1971.
9. E. Mendelson. *Introduction to Mathematical Logic*. Van Nostrand, New York, 1964.

First-Order Logic

10. D.L. Beatty, R.E. Bryant, and C.-J.H. Seger. Synchronous circuit verification by symbolic simulation: An illustration. In *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI*, W.J. Dally (ed.), MIT Press, Cambridge, 1990, pp. 98–112.
11. S. Bose and A.L. Fisher. Verifying pipelined hardware using symbolic logic simulation. In *Proceedings of the IEEE International Conference on Computer Design*, IEEE Computer Society Press, Silver Spring, MD, 1989, pp. 217–221.
12. R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691 (August 1986).

13. R.E. Bryant. Algorithmic aspects of symbolic switch network analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(4):618–633 (July 1987).
14. R.E. Bryant. A methodology for hardware verification based on logic simulation. Technical Report CMU-CS-87-128, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, June 1987.
15. R.E. Bryant. Symbolic analysis of VLSI circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(4):634–649 (July 1987).
16. R.E. Bryant. Verifying a static RAM design by logic simulation. In *Proceedings of the Fifth MIT Conference on Advanced Research in VLSI*, J. Allen and F.T. Leighton (eds.). MIT Press, Cambridge, 1988, pp. 335–349.
17. R.E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffer. COSMOS: A compiled simulator for MOS circuits. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, Los Alamitos, CA, June 1987, pp. 9–16.
18. J.A. Darringer. The application of program verification techniques to hardware verification. In *Proceedings of the Sixteenth ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, Los Alamitos, CA, June 1979, pp. 375–381.
19. R.W. Floyd. Assigning meaning to programs. *Proceedings of Symposia in Applied Mathematics: Mathematical Aspects of Computer Science*, 19:19–31 (1967).
20. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco, 1979.
21. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580 (1969).
22. C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281 (1972).
23. Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, New York, 1978.
24. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.
25. E.F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, C.E. Shannon (ed.), Princeton University Press, Princeton, NJ, 1956, pp. 129–153.
26. R.E. Shostak. Formal verification of circuit designs. In *Proceedings of the Sixth International Symposium on Computer Hardware Description Languages and their Applications*, T. Uehara and M. Barbacci (eds.). IFIP, North-Holland, Amsterdam, 1983.

Boyer-Moore Logic

27. W.R. Bevier. Kit and the short stack. *Journal of Automated Reasoning*, 5(4):519–530 (1989).
28. W.R. Bevier, W.A. Hunt, Jr., J.S. Moore, and W.D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428 (1989).
29. R.S. Boyer and J.S. Moore. Proof-checking, theorem-proving and program verification. *Contemporary Mathematics*, 29:119–132 (1984).
30. R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.
31. A. Bronstein and C.L. Talcott. String-functional semantics for formal verification of synchronous circuits. Technical Report 1210, Stanford University, Stanford, CA, 1988.
32. A. Bronstein and C.L. Talcott. Formal verification of synchronous circuits based on string-functional semantics: The 7 Paillet circuits in Boyer-Moore. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1989, pp. 317–333.
33. S.M. German and Y. Wang. Formal verification of parameterized hardware designs. In *Proceedings of the IEEE International Conference on Computer Design*, IEEE Computer Society Press, Silver Spring, MD, 1985, pp. 549–552.
34. W.A. Hunt, Jr. FM 8501: A verified microprocessor. Ph.D. thesis, Technical Report ICSCA-CMP-47, University of Texas at Austin, 1985.
35. W.A. Hunt, Jr. The mechanical verification of a microprocessor design. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione (ed.). North-Holland, Amsterdam, 1987,

pp. 89–129.

36. W.A. Hunt, Jr. Micropocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460 (1989).
37. J.S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492 (1989).
38. W.D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518 (1989).

Higher-Order Logic

39. H.G. Barrow. Proving the correctness of digital hardware designs. *VLSI Design*, 5:64–77 (July 1984).
40. A.J. Camilleri, M.J.C. Gordon, and T.F. Melham. Hardware verification using higher-order logic. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borriane (ed.). North-Holland, Amsterdam, 1987, pp. 43–67.
41. W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, New York, 1981.
42. A. Cohn. A proof of correctness of the VIPER microprocessor: The first level. In *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P.A. Subrahmanyam (eds.). Kluwer Academic Publishers, Boston, 1987, pp. 27–71.
43. A. Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(4):127–139 (1989).
44. W.J. Cullyer. Implementing safety critical systems: The VIPER microprocessor. In *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P.A. Subrahmanyam (eds.). Kluwer Academic Publishers, Boston, 1987, pp. 1–26.
45. I. Dhingra. Formal validation of an integrated circuit design methodology. In *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P.A. Subrahmanyam (eds.). Kluwer Academic Publishers, Boston, 1987, pp. 293–322.
46. M.J.C. Gordon. LCF.LSM: A system for specifying and verifying hardware. Technical Report 41, Computer Laboratory, University of Cambridge, 1983.
47. M.J.C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, Computer Laboratory, University of Cambridge, May 1985.
48. M.J.C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. Technical Report 77, Computer Laboratory, University of Cambridge, September 1985.
49. M.J.C. Gordon. HOL: A proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P.A. Subrahmanyam (eds.). Kluwer Academic Publishers, Boston, 1987, pp. 73–128.
50. M.J.C. Gordon. Mechanizing programming logics in higher order logic. In *Current Trends in Hardware Verification and Automatic Theorem Proving*, G. Birtwistle and P.A. Subrahmanyam (eds.). Springer-Verlag, New York, 1989, pp. 387–439.
51. M.J.C. Gordon and J. Herbert. Formal hardware verification methodology and its application to a network interface chip. *IEEE Proceedings*, 133, Part E(5):255–270 (September 1986).
52. M.J.C. Gordon, P. Loewenstein, and M. Shahaf. Formal verification of a cell library: A case study in technology transfer. In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, 1989, L.J.M. Claesen, (ed.), North-Holland, Amsterdam, 1990, pp. 409–417 (Volume II).
53. M.J.C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1979.
54. F.K. Hanna and N. Daeche. Specification and verification of digital systems using higher-order predicate logic. *IEEE Proceedings*, 133 Part E(5):242–254 (September 1986).
55. J.M.J. Herbert. Formal verification of basic memory devices. Technical Report 124, Computer Laboratory, University of Cambridge, 1988.

56. C. A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–667 (August 1978).
57. J.J. Joyce. Formal verification and implementation of a microprocessor. In *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P.A. Subrahmanyam (eds.). Kluwer Academic Publishers, Boston, 1987, pp. 129–158.
58. J.J. Joyce. Multi-level verification of microprocessor-based systems. Ph.D. thesis, Technical Report 195, Computer Laboratory, University of Cambridge, 1990.
59. T.F. Melham. Abstraction mechanisms for hardware verification. In *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P.A. Subrahmanyam (eds.). Kluwer Academic Publishers, Boston, 1987, pp. 267–291.
60. T.F. Melham. Using recursive types to reason about hardware in higher order logic. In *Fusion of Hardware Design and Verification*, G.J. Milne (ed.). North-Holland, Amsterdam, 1988, pp. 27–50.
61. T.F. Melham. Formalizing abstraction mechanisms for hardware verification in higher order logic. Ph.D. thesis, Technical Report 201, Computer Laboratory, University of Cambridge, 1990.

Temporal Logic

62. K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309 (1986).
63. H. Barringer and R. Kuiper. A temporal logic specification method supporting hierarchical development. Technical report, University of Manchester, November 1983.
64. H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, ACM, New York, 1984, pp. 51–63.
65. M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20(3):207–226 (1983).
66. G.V. Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers*, C-31(3):223–231 (March 1982).
67. S. Bose and A.L. Fisher. Automatic verification of synchronous circuits using symbolic simulation and temporal logic. In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, 1989, L.J.M. Claessen, (ed.), North-Holland, Amsterdam, 1990, pp. 759–764.
68. M.C. Browne and E.M. Clarke. SML: A high level language for the design and verification of finite state machines. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione (eds.). North-Holland, Amsterdam, 1987, pp. 269–292.
69. M.C. Browne, E.M. Clarke, and D.L. Dill. Automatic circuit verification using temporal logic: Two new examples. In *Formal Aspects of VLSI Design*, G.J. Milne and P.A. Subrahmanyam (eds.). North-Holland, Amsterdam, 1986, pp. 113–124.
70. M.C. Browne, E.M. Clarke, D.L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1044 (December 1986).
71. R.E. Bryant and C.-J.H. Seger. Formal verification of digital circuits using symbolic ternary system models. In *Proceedings of the Workshop on Computer-Aided Verification (CAV 90)*, E.M. Clarke and R.P. Kurshan (eds.), volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Springer-Verlag, New York, 1991.
72. J. Burch, E.M. Clarke, K. McMillan, and D.L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, Los Alamitos, CA, June 1990, pp. 46–51.
73. E.M. Clarke, S. Bose, M.C. Browne, and O. Grumberg. The design and verification of finite state hardware controllers. Technical Report CMU-CS-87-145, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, July 1987.
74. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1981, pp. 52–71.

75. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263 (April 1986).
76. E.M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York, August 1987, pp. 294–303.
77. E.M. Clarke, O. Grumberg, and M.C. Browne. Reasoning about networks with many identical finite-state processes. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York, August 1986, pp. 240–248.
78. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, January 1992.
79. E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, Washington, D.C., June 1989, pp. 353–361.
80. E.M. Clarke, D.E. Long, and K.L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In *International Symposium on Computer Hardware Description Languages and their Applications*, J.A. Darringer and F.J. Rammig (eds.). IFIP. North-Holland, Amsterdam, 1989, pp. 281–295.
81. O. Coudert, J.C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In *Proceedings of the Workshop on Computer-Aided Verification (CAV 90)*, E.M. Clarke and R.P. Kurshan (eds.), volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Springer-Verlag, New York, NY, 1991.
82. D.L. Dill and E.M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEEE Proceedings*, 133 Part E(5):276–282 (September 1986).
83. E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, volume B, J. van Leeuwen (ed.). Elsevier Science Publishers, Amsterdam, 1990, pp. 995–1071.
84. E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In *Proceedings of the Seventh International Colloquium on Automata, Languages, and Programming*, volume 85 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1981, pp. 169–181.
85. E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*. ACM, New York, 1982, pp. 169–180.
86. E.A. Emerson and J.Y. Halpern. ‘Sometimes’ and ‘Not Never’ revisited: On branching time versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178 (1986).
87. E.A. Emerson and C.L. Lei. Modalities for model checking: Branching time strikes back. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, January 1985, pp. 84–96.
88. N. Francez. *Fairness*. Springer-Verlag, New York, 1986.
89. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, 1980, pp. 163–173.
90. O. Grumberg and D.E. Long. Model checking and modular verification. In *Proceedings of CONCUR '91: Second International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, August 1991.
91. J. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals. In *Proceedings of the Tenth International Colloquium on Automata, Languages, and Programming*, volume 154 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1983, pp. 278–291.

92. D. Harel, D. Kozen, and R. Parikh. Process logic: Expressiveness, decidability and completeness. *Journal of Computer and System Sciences*, 25(2):144–170 (1982).
93. G.E. Hughes and M.J. Creswell. *An Introduction to Modal Logic*. Methuen, London, 1977.
94. H.W. Kamp. *Tense Logic and the Theory of Linear Order*. Ph.D. thesis, University of California, Los Angeles, 1968.
95. R. Koymans, J. Vytöpil, and W.-P. de Roever. Real-time programming and asynchronous message passing. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York, 1983, pp. 187–197.
96. L. Lamport. ‘Sometime’ is sometimes ‘Not Never’—On the temporal logic of programs. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, 1980, pp. 174–185.
97. L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222 (April 1983).
98. L. Lamport. What good is temporal logic? In *Proceedings of the IFIP Congress on Information Processing*, R.E.A. Mason (ed.). North-Holland, Amsterdam, 1983, pp. 657–667.
99. M.E. Leeser. Reasoning about the function and timing of integrated circuits with Prolog and temporal logic. Ph.D. thesis, Technical Report 132, Computer Laboratory, University of Cambridge, April 1988.
100. D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proceedings of the Eighth International Colloquium on Automata, Language, and Programming*, volume 115 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1981, pp. 264–277.
101. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, 1985, pp. 97–107.
102. O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proceedings of the Conference on Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1985, pp. 196–218.
103. Y. Malachi and S.S. Owicki. Temporal specifications of self-timed systems. In *VLSI Systems and Computations*, H.T. Kung et. al. (eds.). Computer Science Press, Rockville, MD, 1981, pp. 203–212.
104. Z. Manna and A. Pnueli. Verification of concurrent programs: Temporal proof principles. In *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1981, pp. 200–252.
105. Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In *Correctness Problem in Computer Science*, R.S. Boyer and J.S. Moore (eds.), Academic Press, London, 1982, pp. 215–273.
106. Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, 1983, pp. 141–154.
107. Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4(3):257–290 (1984).
108. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6:68–93 (1984).
109. K.L. McMillan. *Symbolic Model Checking, An approach to the state explosion problem*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992.
110. K.L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, 1991 (sponsored by Information Processing Society, Tokyo, Japan), pp. 242–251.
111. B. Mishra and E.M. Clarke. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science*, 38:269–291 (1985).

112. B. Moszkowski. *Reasoning about Digital Circuits*. Ph.D. thesis, Stanford University, Stanford, CA, 1983.
113. B. Moszkowski. Executing temporal logic programs. Technical Report 55, Computer Laboratory, University of Cambridge, August 1984.
114. B. Moszkowski. A temporal logic for multi-level reasoning about hardware. *Computer*, pp. 10–19 (February 1985).
115. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495 (July 1992).
116. A. Pnueli. The temporal logic of programs. In *Proceedings of the Eighth Annual Symposium on Foundations of Computer Science*, IEEE, New York, 1977, pp. 46–57.
117. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, K. Apt (ed.), Volume 13 of *NATO ASI series, Series F, Computer and System Sciences*, Springer-Verlag, New York, 1984, pp. 123–144.
118. A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proceedings of the Twelfth International Colloquium on Automata, Languages, and Programming*, volume 194 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1985, pp. 15–32.
119. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1982, pp. 337–351.
120. J.P. Queille and J. Sifakis. Fairness and related properties in transition systems. *Acta Informatica*, 19:195–220 (1983).
121. N. Rescher and A. Urquhart. *Temporal Logic*. Springer-Verlag, Berlin, 1971.
122. B.-H. Schlingloff. Modal definability of ω -tree languages. In *Proceedings of the ESPRIT-BRA ASMICS Workshop on Logics and Recognizable Sets*, Germany, 1990.
123. A.P. Sistla and E.M. Clarke. Complexity of propositional linear temporal logic. *Journal of the ACM*, 32(3):733–749 (July 1985).
124. A.P. Sistla, E.M. Clarke, N. Francez, and A.M. Meyer. Can message buffers be axiomatized in temporal logic? *Information and Control*, 63(1):88–112 (1984).
125. A.P. Sistla and S. German. Reasoning with many processes. In *Proceedings of the Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, Washington D.C., 1987, pp. 138–152.
126. P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, January 1986, pp. 184–192.

Extended Temporal Logic

127. E.M. Clarke, O. Grumberg, and R.P. Kurshan. A synthesis of two approaches for verifying finite state concurrent systems. In *Proceedings of Symposium on Logical Foundations of Computer Science: Logic at Botik '89*, volume 363 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, July 1989.
128. A.C. Shaw. Software specification languages based on regular expressions. Technical report, ETH Zurich, June 1979.
129. P. Wolper. Temporal logic can be made more expressive. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, IEEE, New York, 1981, pp. 340–348.
130. P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, IEEE, New York, 1983, pp. 185–194.

Mu-Calculus

131. J. Burch, E.M. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, Washington, D.C., June 1990, pp. 428–439.

132. E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, Washington, D.C., 1986, pp. 267–278.
133. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354 (December 1983).
134. D. Niwinski. Fixed points vs. infinite generation. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, Washington, D.C., July 1988, pp. 402–409.
135. D. Park. Finiteness is mu-ineffable. Theory of Computation Report No. 3, University of Warwick, Warwick, England, 1974.
136. D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the Fifth GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1981, pp. 167–183.
137. V. Pratt. A decidable mu-calculus. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, IEEE, New York, 1981, pp. 421–427.

Functional Approaches

138. D. Borriore, P. Camurati, J.L. Paillet, and P. Prinetto. A functional approach to formal hardware verification: The MTI experience. In *Proceedings of the IEEE International Conference on Computer Design*, IEEE Computer Science Press, Silver Spring, MD, 1988, pp. 592–595.
139. D. Borriore and J.L. Paillet. An approach to the formal verification of VHDL descriptions. Research Report 683, IMAG/ARTEMIS, Grenoble, France, November 1987.
140. Z. Chaochen and C.A.R. Hoare. A model for synchronous switching circuits and its theory of correctness. In *Proceedings of the Workshop on Designing Correct Circuits*, G. Jones and M. Sheeran (eds.), Springer-Verlag, New York, 1990, pp. 196–211.
141. G.J. Milne. CIRCAL: A calculus for circuit description. In *Integration, the VLSI Journal*, Volume 1, Nos. 2 & 3, pp. 121–160 (October 1983).
142. G.J. Milne. A model for hardware description and verification. In *Proceedings of the 21st ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, Los Alamitos, CA.
143. M. Sheeran. μFP , An Algebraic VLSI Design Language. Ph.D. thesis, University of Oxford, England, 1983.
144. M. Sheeran. Design and verification of regular synchronous circuits. *IEE Proceedings*, 133 Part E(5):295–304 (September 1986).
145. T.J. Wagner. *Hardware Verification*. Ph.D. thesis, Stanford University, Stanford, CA, 1977.
146. T.J. Wagner. Verification of hardware designs through symbolic manipulation. In *Proceedings of the International Symposium on Design Automation and Microprocessors*, IEEE, New York, 1977, pp. 50–53.
147. D. Weise. Multilevel verification of MOS circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(4):341–351 (April 1990).
148. G. Winskel. A compositional model of MOS circuits. In *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P.A. Subrahmanyam (eds.), Kluwer Academic Publishers, Boston, 1987, pp. 323–347.

Machine Equivalence

149. J.P. Billon. Perfect normal forms for discrete functions. Technical Report 87019, Bull Research Center, Louveciennes, France, June 1987.
150. J.P. Billon and J.C. Madre. Original concepts of PRIAM, an industrial tool for efficient formal verification of combinational circuits. In *Fusion of Hardware Design and Verification*, G.J. Milne (ed.), North-Holland, Amsterdam, 1988, pp. 487–501.
151. O. Coudert, C. Berthet, and J.C. Madre. Verification of sequential machines using Boolean functional vectors. In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, 1989, L.J.M. Claesen, (ed.), North-Holland, Amsterdam, 1990, pp. 111–128.

152. O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1989, pp. 365–373.
153. O. Coudert and J.C. Madre. Logics over finite domain of interpretation: Proof and resolution procedures. Technical report, Bull Research Center, Louveciennes, France, 1989.
154. S. Devadas, H-K. T. Ma, and A.R. Newton. On the verification of sequential machines at differing levels of abstraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, June 1988, pp. 713–722.
155. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, 1979.
156. J.C. Madre and J.P. Billon. Proving circuit correctness using formal comparison between expected and extracted behavior. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 205–210.

Language Containment

157. S. Aggarwal, R.P. Kurshan, and K.K. Sabnani. A calculus for protocol specification and validation. In *Protocol Specification, Testing and Verification III*. North-Holland, Amsterdam, 1983, pp. 19–34.
158. J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science*, E. Nagel et al (ed.). Stanford University Press, Stanford, CA, 1960, pp. 1–12.
159. E.M. Clarke, I.A. Draghicescu, and R.P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω -automata. In *Proceedings of the Fifteenth Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, May 1990.
160. I. Gertner and R.P. Kurshan. Logical analysis of digital circuits. In *Proceedings of the Eighth International Symposium on Computer Hardware Description Languages and their Applications*, M.R. Barbacci and C.J. Koomen (eds.). IFIP, North-Holland, Amsterdam, 1987, pp. 47–67.
161. B. Gopinath and R.P. Kurshan. The Selection/Resolution model of coordinating concurrent processes. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1980.
162. P. Halmos. *Lectures on Boolean Algebras*. Springer-Verlag, New York, 1974.
163. Z. Har'El and R.P. Kurshan. Software for analytical development of communication protocols. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, January 1990.
164. J. Katzenelson and R.P. Kurshan. S/R: A language for specifying protocols and other communicating processes. In *Proceedings of the Fifth IEEE International Conference on Computer Communications*, IEEE, New York, 1986, pp. 286–292.
165. R.P. Kurshan. Reducibility in analysis of coordination. In *Discrete Event Systems: Models and Applications*, volume 103 of *Lecture Notes in Control and Information Sciences*. Springer-Verlag, New York, 1987, pp. 19–39.
166. R.P. Kurshan. Analysis of discrete event coordination. In *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (eds.). Springer-Verlag, New York, 1989.
167. R.P. Kurshan and K.L. McMillan. A structural induction theorem for processes. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York, 1989, pp. 239–247.
168. R.P. Kurshan and K.L. McMillan. Analysis of digital circuits through symbolic reduction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(11):1356–1371 (November 1991).
169. W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, volume B, J. van Leeuwen (ed.). Elsevier Science Publishers, Amsterdam, 1990, pp. 133–191.

170. P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1989, pp. 68–80.

Trace Theory

171. J.R. Burch. Combining CTL, trace theory and timing models. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1989, pp. 334–348.
172. T.-A. Chu. On the models for designing VLSI asynchronous digital systems. *Integration, the VLSI Journal*, 4:99–113 (1986).
173. D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. Ph.D. thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, 1988. Also published in ACM Distinguished Dissertations Series, MIT Press, Cambridge, MA, 1989.
174. D.L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. In *Proceedings of the Fifth MIT Conference on Advanced Research in VLSI*, J. Allen and F.T. Leighton (eds.). MIT Press, Cambridge, MA, 1988.
175. D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1989, pp. 197–212.
176. A.J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In H. Fuchs, ed., *Proceedings of the 1985 Chapel Hill Conference on VLSI*; W.H. Freeman, New York, 1985, pp. 245–260.
177. M. Rem. Concurrent computation and VLSI circuits. In *Control Flow and Data Flow: Concepts of Distributed Programming*, M. Broy (ed.). Volume 14 of NATO ASI series, series F. Computer and System Sciences, Springer-Verlag, New York, 1985, pp. 399–437.
178. J.L.A. van de Sneepscheut. *Trace Theory and VLSI Design*. Ph.D. thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands, 1983.

Hybrid Approaches

179. E.A. Emerson and C.L. Lei. Temporal model checking under generalized fairness constraints. In *Proceedings of the Eighteenth Hawaii International Conference on System Sciences*, 1985, Western Periodicals Company, North Hollywood, CA, pp. 277–288, (Vol. I).
180. M. Fujita and H. Fujisawa. Specification, verification, and synthesis of control circuits with propositional temporal logic. In *Proceedings of the Ninth International Symposium on Computer Hardware Description Languages and their Applications*, J.A. Darringer and F.J. Rammig (eds.). IFIP, North-Holland, Amsterdam, 1989, pp. 265–279.
181. M. Fujita, H. Tanaka, and T. Moto-oka. Verification with Prolog and temporal logic. In *Proceedings of the Sixth International Symposium on Computer Hardware Description Languages and Their Applications*, T. Uehara and M. Barbacci (eds.). IFIP, North-Holland, Amsterdam, 1983, pp. 103–114.
182. M. Fujita, H. Tanaka, and T. Moto-oka. Logic design assistance with temporal logic. In *Proceedings of the Seventh International Symposium on Computer Hardware Description Languages and their Applications*, C.J. Koomen and T. Moto-oka (eds.). IFIP, North-Holland, Amsterdam, 1983, pp. 129–138.
183. P. Loewenstein. Reasoning about state machines in higher-order logic. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, M. Leeser and G. Brown (eds.). volume 408 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1990.
184. P. Loewenstein and D.L. Dill. Verification of a multiprocessor cache protocol using simulation relations and higher-order logic. In *Proceedings of the Workshop on Computer-Aided Verification (CAV 90)*, E.M. Clarke and R.P. Kurshan (eds.). volume 3 of *DIMACS Series in Discrete*

Mathematics and Theoretical Computer Science. American Mathematical Society, Springer-Verlag, New York, 1991.

185. F. Maruyama and M. Fujita. Hardware verification. *IEEE Computer*, 18(2):22–32 (February 1985).
186. A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. In *Proceedings of the Twelfth International Colloquium on Automata, Languages, and Programming*, volume 194 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, pp. 465–474.
187. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, Washington, D.C., June 1986, pp. 332–344.
188. M. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32(2): 183–221 (April 1986).
189. P. Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, 28:119–136 (1985).

Real-Time Verification

190. R. Alur, C. Courcoubetis, and D.L. Dill. Model checking for real-time systems. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Washington, D.C., June 1990, pp. 414–425.
191. R. Alur and D.L. Dill. Automata for modeling real-time systems. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages, and Programming*, volume 443 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1990.
192. R. Alur and T.A. Henzinger. A really temporal logic. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, IEEE, New York, pp. 164–169, 1989.
193. R. Alur and T.A. Henzinger. Real-time logics: Complexity and expressiveness. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, Washington, D.C., June 1990, pp. 390–401.
194. M.R. Barbacci and J.M. Wing. Specifying functional and timing behavior for real-time systems. In *Proceedings of the Conference on Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, June 1987.
195. A. Bernstein and P. Harter. Proving real-time properties of programs with temporal logic. In *Proceedings of the Eighth Annual ACM Symposium on Operating Systems Principles*, ACM, New York, December 1981, pp. 1–11.
196. J.R. Burch. Modeling timing assumptions with trace theory. In *Proceedings of the IEEE International Conference on Computer Design*, IEEE Computer Science Press, Silver Spring, MD, October 1989, pp. 208–211.
197. J.R. Burch. *Automatic Verification of Real-Time Concurrent Systems*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991 (forthcoming).
198. E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. Presented at the International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, June 1989.
199. E. Harel, O. Lichtenstein, and A. Pnueli. Explicit clock temporal logic. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Washington, D.C., June 1990, pp. 402–413.
200. F. Jahanian, R.S. Lee, and A.K. Mok. Semantics of modechart in real time logic. In *Proceedings of the 21st Hawaii International Conference on System Sciences*, IEEE Computer Science Press, Washington, D.C., January 1988, Volume II, pp. 479–489.
201. F. Jahanian and A. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904 (1986).
202. F. Jahanian, A. Mok, and D.A. Stuart. Formal specification of real-time systems. Technical Report TR-88-25, Department of Computer Science, University of Texas at Austin, June 1988.

203. F. Jahanian and D.A. Stuart. A method for verifying properties of modechart specification. In *Proceedings of the IEEE Real-Time Systems Symposium*, IEEE, New York, December 1988, pp. 12–21.
204. R. Koymans. *Specifying Message-Passing and Time-Critical Systems with Temporal Logic*. Ph.D. thesis, Eindhoven University of Technology, The Netherlands, 1989.
205. H. Lewis. A logic of concrete time intervals. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Washington, D.C., June 1990, pp. 380–389.
206. G.H. MacEwen and D.B. Skillicorn. Using higher-order logic for modular specification of real-time distributed systems. In *Formal Techniques in Real-time and Fault Tolerant Systems: Proceedings of a Symposium*, M. Joseph (ed.), Volume 331 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1988, pp. 36–66.
207. J. Ostroff. Real-time computer control of discrete event systems modeled by extended state machines: A temporal logic approach. Technical Report 8618, University of Toronto, September 1987.
208. J. Ostroff. Automated verification of timed transition models. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1989, pp. 247–256.
209. A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems. In *Formal Techniques in Real-time and Fault Tolerant Systems: Proceedings of a Symposium*, M. Joseph (ed.), Volume 331 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1988, pp. 84–98.
210. A. Zwarico and I. Lee. Proving a network of real-time processes correct. In *Proceedings of the IEEE Real-Time Systems Symposium*, IEEE, New York, December 1985, pp. 169–177.
211. J.R. Burch. Using BDDs to verify multipliers. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, Los Alamitos, CA, June 1991, pp. 408–412.
212. J.R. Burch, E.M. Clarke, and D.E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, Los Alamitos, CA, June 1991, pp. 403–407.
213. H. Busch and G. Venzl. Proof-aided design of verified hardware. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, Los Alamitos, CA, June 1991, pp. 391–396.
214. H. Eveking. Verification, synthesis and correctness-preserving transformations—cooperative approaches to correct hardware design. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione (ed.). North-Holland, Amsterdam, 1987, pp. 229–239.
215. W. Luk and G. Jones. From specifications to parameterized architectures. In *Fusion of Hardware Design and Verification*, G.J. Milne (ed.). North-Holland, Amsterdam, 1988, pp. 267–268.
216. H. Ochi, N. Ishiura, and S. Yajima. Breadth-first manipulation of SBDD of Boolean functions for vector processing. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, Los Alamitos, CA, June 1991, pp. 413–416.
217. V. Stavridou, H. Barringer, and D.A. Edwards. Formal specification and verification of hardware: A comparative case study. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, 1988, pp. 197–204.
218. D. Verkest, P. Johannes, L. Claesen, and H. De Man. Formal techniques for proving correctness of parameterized hardware using correctness preserving transformations. In *Fusion of Hardware Design and Verification*, G.J. Milne (ed.). North-Holland, Amsterdam, 1988, pp. 77–97.
219. D. Verkest, P. Johannes, L. Claesen, and H. De Man. Correctness proofs of parameterized hardware modules in the Cathedral-II synthesis environment. In *Proceedings of the European Design Automation Conference*, Glasgow, 1990. IEEE Computer Society Press, Washington, D.C., 1990.