Formal Verification of Hardware Correctness:

Introduction and Survey of Current Research

Paolo Camurati and Paolo Prinetto

Politecnico di Torino

"To verify is to prove the truth of something by presenting evidence for it." To benefit from this dictionary-like, generic definition, we must tailor it to the particular domain we want to consider: the design of hardware.

Every design, no matter how strategic or complex, requires multifaceted verification before marketing. Starting from final manufacturing and moving back through previous phases of the process, we might find many objects for verification. Low-level design rules, timing, high-level design rules, firmware, functional correctness, and base software might attract our interest. The domain of verification spans all phases of design, covering hardware, firmware, and software. We restrict our discussion in this article to one particular item—the verification of functional correctness through formal techniques.

To formally verify hardware correctness, we need suitable representation systems and automated proofs. Logic, techniques from software verification, and automated synthesis benefit the process.

Let us begin by giving two definitions of *functional correctness*, which we will use throughout. First, at every design step, the designer specifies what the system under development should do and how it should do it. What the system should do is called its *specification*, while any one of the possible devices that realizes the specification is called an *implementation*. The design of a system may reduce to an iteration of specification/implementation steps, performed either top-down or bottom-up, where the implementation at level *i* becomes the specification for level *i* + 1. Any piece of hardware is functionally correct if we can somehow prove that its implementation realizes the specification (see Figure 1).

Other concepts of correctness exist. Sometimes it is useful to consider an existing design and to verify some of its properties. We can group such properties into two main classes:

- safety properties and
- liveness properties.

Safety properties express conditions of the form "bad things will never occur." Within the framework of branching time temporal logic (with multiple evolutions in the future, each consisting of a "path" connecting some "states" and expressing before/after relations between the events), an example of a safety property looks as follows:

> "for every path in the future, at every node on the path, if the Request signal is low, it must remain low until Acknowledge goes low"

Liveness properties express conditions of the form "good things will occur in the future." A branching time temporal logic example looks as follows:

> "for every path in the future, if there has been a Request signal, then eventually there will be an Acknowledge

0018-9162/88/0700-0008$01.00©1988 IEEE

signal in response to the request on at least one node on the path"

Safety and liveness properties play the role of *partial* specifications; that is, they do not describe device behavior globally, rather they cover limited aspects. We call a design correct with respect to the properties if we can demonstrate that the properties are true. This is our second definition. Since specifications are partial, we must take particular care in their selection, trying to cover all or as many good and bad things as possible.

Obviously, correctness is not an autonomous concept, but rather a relation between two entities: a specification and an implementation, or a property and a design. Verification to first principles is thus impossible—a verified design is only as good as its specification. Specification languages fail in being so involved and detailed that no practitioner would ever use them. Moreover, the specifications are as likely to have errors as the implementation, and it is unlikely that anyone would ever first write a formal specification and then implement it.

A possible classification of the relations between a specification α and an implementation β or a property α and a design β that must be demonstrated in a proof follows:

- equality: $\alpha = \beta$,
- equivalence: $\alpha \leftrightarrow \beta$,
- logical implication: $\alpha \rightarrow \beta$,
- homomorphism: $\mathrm{M}(\phi(a)) = \mathrm{M}(b)$

The last item, used in model-based approaches in first-order predicate calculus, consists of defining a function (the homomorphic function ϕ) and a mapping M that makes possible the comparison between a model of the specification and a model of the implementation. Specifications and implementations often fall at different levels of abstraction, thus we must transform them through a ϕ function: structural abstraction consists of hiding internal lines, data abstraction transforms a data type into another data type, and temporal abstraction takes into account timing models and time's granularity.

In this article we analyze formal verification techniques focusing on two key points: suitable representation systems and mechanizable proofs. But before we look at current research efforts, we will briefly discuss related topics to make them understandable to novice readers.

First we report on different approaches to hardware verification. We compare formal verification and automated synthesis to show how they cooperate in producing zero-defect designs. Next we present the necessities for formal verification techniques: formal representation systems and the related reasoning facilities. For each technique we consider some of the main efforts related to it and evaluate them. A reading list covers the specific topics for those interested in exploring further. The reading list includes all works on which we based our discussions. Finally, we evaluate the different approaches and show the promise of research in this field.

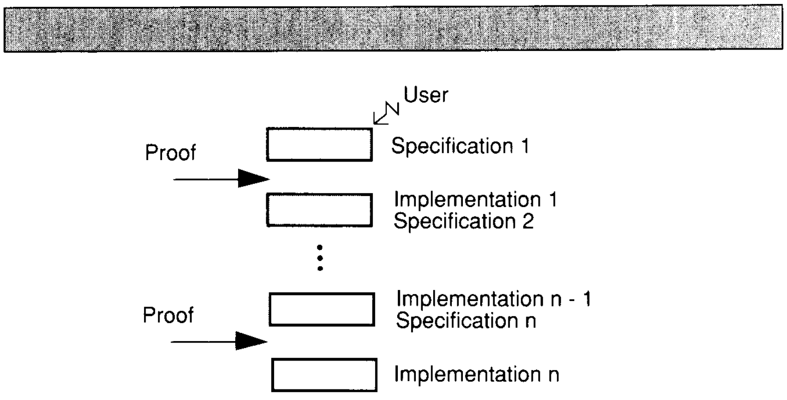

Figure 1. Correctness.

Approaches to hardware verification

We can verify the correctness of hardware in many different ways, such as through breadboarding, simulation, and formal proof.

Rising design and fabrication costs, market demands, circuits moving toward very-large-scale integration, and growing system complexity made the use of a prototype for extensive testing almost impossible. Breadboarding has also dwindled because parasitics, thermal and electrical characteristics, and component-matching properties differ for discrete components and VLSI. Nowadays, breadboarding for VLSI is restricted to very special applications, such as systems used in space flights. Breadboarding's limits forced the extensive use of simulation in establishing hardware correctness.

Simulation differs from breadboarding mainly in the presence of a model of the system under consideration. Although there are various notations to describe the models, the ultimate goal of using them is to allow the operation of a simulation engine.

Simulation-based design evaluation steps through the following phases:

(1) Model description in a suitable language
(2) Test input stimuli generation
(3) Simulation
(4) Result extraction from simulation runs
(5) Extracted and expected data comparison
(6) Circuit/system redesign

The last three phases have been successfully applied in commercially available systems. A major drawback of Step 3 as a means of establishing correctness or computing performances lies in the generation of input stimuli. If we must prove correctness with certainty and/or evaluate performance with accuracy, we should consider all possible input combinations and their sequences. This leads to an explosion of cases, which soon makes the approach unfeasible.

Thus Step 2 replaces exhaustive simulation. The designer prepares a set of test cases that he or she considers sufficient to establish correctness. If extracted and expected data differ, something has gone wrong; otherwise, nothing can be stated unequivocally. As E.W. Dijkstra has remarked, "Non-exhaustive testing can be used to show the presence of bugs, but never to show their absence." The degree

```
Specification:
two-bit: box;
        input start;
        output register c(0:1);
        | c = 0 ∧ start | c = 1;
        | 0 < c < 4 | c = c+1;
end box;
```

(a)

```
Implementation:
two-bit: box;
        input start;
        output c0, c1;
        c0 = JKFF(tt,cs,tt);
        c1 = JKFF (tt,clock,ts);
        ts = OR2(tt,start);
        tt = OR2(c0,c1);
        cs = OR2(c1,ct);
        ct = AND2(clock,nt);
        nt = NOT1(tt);
        end box;
```

(b)

Figure 2. A two-bit counter with a behavioral specification under the form of a state transition description (a) and a structural implementation under the form of a network diagram description (b).

Case	Specification result
$C=0 \wedge START=1$	$C=1$
$C=0 \wedge START=0$	$C=0$
$C<>0$	$C=C+1$

Case	Implementation result
$C0=0 \wedge C1=0 \wedge START=1$	$C0=1$ $C1=0$
$C0=0 \wedge C1=0 \wedge START=0$	$C0=0$ $C1=0$
$C0=1 \vee C1=1$	$C0=C1$ xor $C0$ $C1= \sim C1 \wedge C0$

Figure 3. Given beginning specification and implementation values, we can find and manipulate symbolic values to establish equivalance.

of confidence the designer has in the simulated system depends on the quality of the test cases.

Another crucial point, common to simulation and formal verification, is Step 1. The model must, in fact, represent the real system accurately, otherwise it is not only useless, but misleading, too.

Having established the limits of simulation, we could continue with an enhanced simulation-based approach to verification; abandon simulation, resorting to formal methods to prove hardware correctness; or integrate various approaches, both formal and simulation-based.

As far as the first choice is concerned, we could resort to *symbolic simulation*, an offspring of conventional simulation because it uses a model for hardware and a simulation engine, but differing from it in considering symbols rather than actual values for the circuit under consideration. In this way we can simulate the response to entire classes of values with a notable improvement over traditional techniques.

Symbolic simulation extends to verification of correctness because specifications and implementations can be run concurrently and the results manipulated and compared to establish a proof. As an example, let us consider a two-bit counter[1] with a behavioral specification under the form of a state transition description and a structural implementation under the form of a network diagram description (see Figure 2). Note that the descriptions resort to different timing models and that there is no explicit conversion function from natural numbers to bit-vectors. The correspondence between states and output values of the two machines is given by a simulation relation that must hold at each clock tick:

Specification		Implementation
start	=	start
c(0)	=	c0
c(1)	=	c1

Starting specification and implementation with c = C and start = START, we find and manipulate symbolic values to establish equivalence as shown in Figure 3. Trivial equivalence cases are easily solved. We leave nontrivial ones, such as proving that some bit-vector operations realize integer addition, to theorem provers.

A novel and promising approach to hardware verification is formal verification. The key concept lies in the word "formal": it means that the proof is mathematical, rather than experimental. Mathematical demonstration overcomes the limits of test-case simulation, since it is valid for all input stimuli under specified assumptions.

Formal verification needs suitable systems to represent the objects it considers (specifications, implementations, properties) and means to perform proofs. Formal systems must be mathematically sound and tailored to the domain of application, that is, to the classes of designs to be verified.

Hybrid approaches use formal techniques and exhaustive simulation, such as enumeration on a restricted set of variables, trying to balance proof power and computational efficiency.

We can also use formal techniques to transform a design. In this case we talk of "correctness-preserving transformations." Such techniques are particularly useful when integrating verification and automated synthesis in a cooperative approach to correct hardware design. A correctness-preserving transformation takes a correct implementation of a specification and derives another correct implementation. We use these transformations to generate design alternatives to improve the quality of some original solution, to explore the design space, and to prove the equivalence of two hardware descriptions.

Formal verification versus automated synthesis. The goals of automated synthesis and of verification seem mutually exclusive: the former aims at correct-by-construction designs, while the latter targets the proof of post-factum designs. Formal verification does not consider aspects such as area, cost, and performance. We can regard formal verification as an ancillary approach, replaceable by synthesis as soon as synthesized designs surpass handmade ones.

Given the restricted domain of formal verification, we can reasonably suppose that it will reach its goals sooner than synthesis, although a circuit 90-percent synthesized is more useful than a circuit 90-percent verified—the subtle bugs will occupy the unverified 10 percent. Moreover, formal verification helps in defining the concept of correctness and correct-by-construction design methodologies. Thus, both approaches benefit from the advancement of the same theoretical studies.

Formal systems for hardware representation

Since the difference between formal and simulation-based verification lies in the presence of a mathematical proof, it is essential to have a formalism to represent hardware systems at all levels of abstraction. Such a formalism requires a complete, precise, and coherent definition of the underlying semantics. Consequently, we associate with each formal system a set of calculation properties allowing mathematical proofs.

We can distinguish semantics as operational, denotational, or axiomatic.[2]

Operational semantics defines the meaning of any term of a formalism in terms of the actions performed by an abstract state machine that interprets the statements written in the formalism.

Denotational semantics entails an abstraction of the interpretation mechanism, since it introduces a model that views a term as a function transforming states into other states. Denotational semantics assigns meaning in terms of set and function theories.

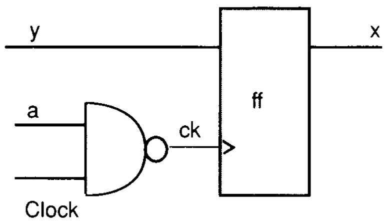

Figure 4. A synchronous circuit.

Axiomatic semantics in the theory of programming languages means that some formulas of predicate calculus are associated with a program. From a mathematical point of view, this is just a transformation of a formal language not part of a formal system into a formal language for which a syntactical concept of derivability exists.

The next section presents logic, the most widely known and used formal system. Subsequently, we will present the use of programming languages for hardware representation, although they aren't formal systems. Programming languages are widely used in conjunction with logic in software verification, and some authors extend their use to hardware, too, cross-fertilizing some ideas with software verification techniques.

Formal logic

Logic is that branch of knowledge concerned with truth and inference. It investigates the structure of propositions and deductions, resorting to a method that abstracts from the contents of the propositions in question and is related only to their form. A proposition is a declarative statement to which we can attach a true or false value. The meaning of a proposition is given by an interpretation in some "world," while syntax is the only key available for us to manipulate and reason about. Ad hoc formalisms enhance the distinction between form and meaning. Logicians have proposed various formalisms. We will briefly outline the most relevant ones in the domain we consider:

- First-order predicates
- Higher-order predicates
- Specific calculi
- Temporal logic

First-order predicates. First-order predicates deal with propositions and propositional functions, that is, those functions where the domain of the independent variable ranges over propositions. All usual logical connectives are defined as well as the existential ($\exists$) and universal ($\forall$) quantifiers. Interpretations of first-order predicates require a specified domain and symbols referring to entities within this domain. Different domains lead to different interpretations.

The language of first-order predicates consists of terms, atoms, and formulas. The traditional approach outlined above is purely logical and does not provide the common concept of function. Thus, it is mainly used by Prolog-oriented researchers. Other authors prefer to introduce truth values in their formal systems rather than assigning them by means of an interpretation. Hence, they do not deal with predicates, rather they use functions with truth values as the range.

Model theory offers another approach to first-order predicates.[3] Model theory deals with the relationship between a formal system and an algebraic structure that gives a meaning to it.

Example. Consider the simple synchronous circuit[4] of Figure 4, consisting of a D-type flip-flop with clock input gated by a NAND gate. The goal is to show how

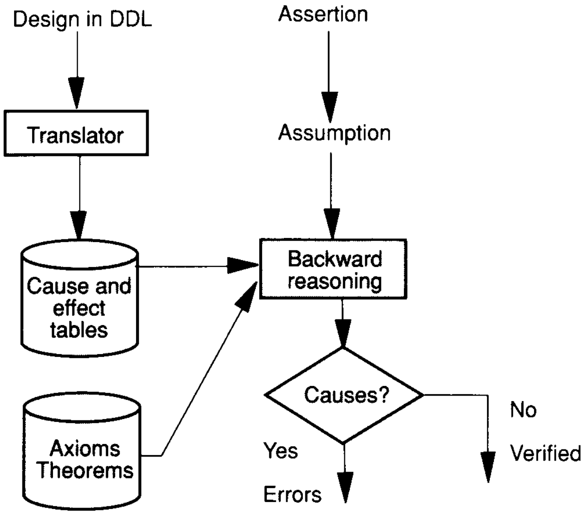

Figure 5. DDL verifier.

first-order predicate calculus can describe structure and behavior. The device resorts to a precise-delay timing model.

The description in terms of first-order predicates expresses behavior as a set of time functions, possibly with initial or generic conditions. It looks as follows:

$$\begin{aligned}
&ck(t) = nand(a(t - d_{nand}),\\
&\qquad\qquad clock(t - d_{nand})),\\
&ff(0) = L,\\
&\sim(t=0) \wedge (re(ck(t)) = L)) \Rightarrow\\
&\quad (ff(t) = ff(t-1)),\\
&\sim(t=0) \wedge (re(ck(t)) = H)) \Rightarrow\\
&\quad (ff(t) = y(t)),\\
&x(t) = ff(t - d_{ff})
\end{aligned}$$

Function *re* detects the rising edge of a signal, d_{nand} is the delay of the NAND gate, and d_{ff} is a propagation time through the flip-flop.

Research. T.J. Wagner of Stanford University, California, first attempted to apply predicate calculus to the verification of hardware using an available theorem-prover for a number of simple proofs of unit-delay descriptions.

A.S. Wojcik of Illinois Institute of Technology, Chicago, presented Aura, the Argonne Automated Reasoning Assistant, an experimental system for formal verification. The axiom set includes rules for Boolean algebra, signal specifications, and structure handling. Wojcik described specifications and implementations at the register-transfer level using a clause form.

J.C. Barros of Shell Development, Houston, Texas, and B.W. Johnson of the University of Texas, Dallas, used first-order predicates to describe some classes of commonly used synchronous circuits, such as arbiters, synchronizers, latches, and inertial delays. They defined axioms in a ternary algebra which, in addition to the standard values true and false, has the unknown value *u*. They introduced two kinds of axioms, constraining the outputs or allowing partial inference of output behavior from input behavior. The predicates included parameters to specify timing conditions. Theorem proving demonstrated correctness, but no mechanization of this approach has been reported.

N. Suzuki of the University of Tokyo explored a hybrid methodology between simulation and formal verification. The specifications, represented by input/output assertions in first-order predicates, tie the method to formal verification. Instead of showing that the implementation for all inputs satisfying input assertions satisfies output assertions, Suzuki showed that this holds only for selected inputs, called *test data*. The main interest of this approach lies in the language the author used for description: Concurrent Prolog, a logic-programming language that allows concurrency and efficient backtracking by means of guards and commit operators. Suzuki's approach was applied to the memory of Dorado, a high-performance personal computer designed at Xerox Palo Alto Research Center.

T. Uehara, T. Saito, F. Maruyama, and N. Kawato of Fujitsu Laboratories, Kawasaki, Japan presented a system called the DDL Verifier (see Figure 5), applied to synchronous systems at the functional level. Implementations are described in the DDL language and specifications are first-order predicates. A translator reduces the circuit under consideration to cause/effect tables, which show necessary and sufficient conditions for circuit operations. The proof method uses backward reasoning and proof by contradiction. First, the specification is negated and then traced backward using cause/effect tables to show that it is always false. The DDL Verifier takes time into account, exploring truth and falsity in present and past states of the circuit under consideration. In a later paper, the authors abandoned first-order predicates as a means of expressing specifications and resorted to temporal logic.

H. Eveking of the Technische Hochschule Darmstadt, Federal Republic of Germany, introduced a fundamental distinction between vertical and horizontal verification. *Vertical* verification means compliance between the specification and the implementation of a circuit. When the behavior of a device is described, a model is created and associated with the real object. Such a model is valid only if the environment in which it operates satisfies a set of constraints; thus, the device behaves as specified in the description. These constraints, called *assertions*, may refer to any point in the device. When devices interconnect in a more complex structure, you must extract a set of interface assertions that have no ties to internal points and that guarantee—if satisfied—that internal assertions also hold. Eveking calls this process of composing assertions and propagating them outwards *horizontal* verification. A support tool for such a methodology is Vertico, an expert system for generating interface assertions through predicate transformation.

For vertical verification, Eveking considered synchronous systems described in

SMAX, a nonprocedural hardware description language belonging to the Conlan family. Eveking considers HDLs formal languages but not formal systems. Thus, HDL descriptions must be axiomatized and transformed into first-order predicates. The predicates associated with a description may be classified as logical axioms, that is, general-purpose axioms; HDL-specific axioms; and description-specific axioms. The formal system provides a set of inference rules; axioms and inference rules form a *theory*. This set of entities allows a precise definition of correctness: an implementation is correct with respect to a specification if the nonlogical axioms of the former are theorems in the latter's theory. When specification and implementation occupy different abstraction levels and/or use different timing models, you must introduce an interpretation of a language (or a theory) in another language (or theory), thus generating a suitable mapping.

Mechanization of this approach is under study, but for efficiency's sake, instead of a general-purpose theorem-prover, Eveking uses a set of smaller and specialized provers whose activation is controlled by an expert system integrating frames and production rules.

W.A. Hunt of the Institute for Computing Science and Computer Applications at the University of Texas, Austin, used Boyers-Moore logic to describe and verify the FM8501 microprocessor. The specification is the microprocessor viewed as an instruction interpreter; that is, for every possible instruction, a new microprocessor state is defined. The implementation is a graph of logic gates. The formalism consists of quantifier-free first-order predicates. Recursive functions provide the primary means of description for hardware devices. Time is only implicitly modeled as a stream of values, which constitutes a weakness of this approach. The objects the author considered are bit-vectors of arbitrary size, natural numbers, and integer numbers. Specifications and implementations are both expressed as recursive functions. Implementations, or hardware formulas, can be automatically expanded to yield structural descriptions.

Evaluation. First-order predicates represent a substantial share of current research efforts. They are important both from methodological and practical points of view. In particular, studies with first-order predicates have contributed to the understanding of their expressive limits and fostered research on higher-order predicates.

Higher-order predicates. Higher-order predicates represent an enhancement of first-order predicates. They extend the notation of first-order predicates in that the domain of variables also ranges over functions and predicates, and functions can take functions as arguments and return functions as results. As an example, consider how we could express a falling clock signal:

$$\forall clock\ t.\ (Fall\ clock)\ t\ =\ clock(t)\ \wedge\ \sim clock(t+1)$$

The variable *clock* has time functions as domain—that is, functions from time—modeled as positive integers, to Boolean values. *Fall* is a higher-order function accepting *clock* as an argument and returning a predicate on positive integers.

Unrestricted higher-order predicates suffer from a number of paradoxes, avoided by resorting to type theory and type hierarchy. Higher-order predicates generally contain the axioms of infinity and choice. The *infinity* axiom states that the domain of individuals is infinite; the *choice* axiom allows us to introduce new primitive formulas.

Example. This example illustrates the use of higher-order predicates to represent parameterized systems such as an n-bit adder.[5] An n-bit adder computes an n-bit sum and one-bit carry-out from two n-bit inputs and a one-bit carry-in. The lines c_{in} and c_{out} carry one-bit words and the lines a, b, and sum carry n-bit words. One-bit words are modeled as Booleans; n-bit words are modeled as functions mapping natural numbers into Booleans. The specification follows:

$$\begin{aligned}&Adder(n)(a,b,c_{in},sum,c_{out}) \equiv\\ &(2^{n+1} * Bit\_Val(c_{out}) + Val(n,sum) =\\ &\quad Val(n,a) + Val(n,b) + Bit\_Val(c_{in}))\end{aligned}$$

The higher-order function *Adder* applied to the n-bit number yields a predicate specifying the corresponding adder. The function *Bit_Val* relates the logical values true and false with numbers 1 and 0; *Val* transforms bit-vectors recursively into numbers.

You can implement an n-bit adder by connecting n full adders. The inputs are a single bit-carry in c_{in} and two n-bit words, $a_{n-1} \ldots a_0$, $b_{n-1} \ldots b_0$. The outputs are an n-bit sum, $sum_{n-1} \ldots sum_0$, and a one-bit carry-out, c_{out}. The implementation uses a recursively defined higher-order function, *Add_Imp*, which when applied to a number n yields the predicate specifying the implementation of an n-bit adder. The recursive part of the function says that you build an n-bit adder by first building an $n-1$-bit adder and then connecting its carry-out to the carry-in of a one-bit adder:

$$\begin{aligned}&Add\_Imp(n)(a,b,c_{in},sum,c_{out}) \equiv\\ &\exists c.Add\_Imp(n)(a,b,c_{in},sum,c) \wedge\\ &\qquad\quad Add1(a(n),b(n),c,sum(n),c_{out})\end{aligned}$$

Research. F.K. Hanna and N. Daeche of the University of Kent, United Kingdom, presented a system called Veritas for specification and verification of hardware. Specifications, written in higher-order logic, are partial and hierarchical, and they can take into account low-level timing issues. A designer's knowledge of digital systems is structured as a set of theories, each defined by a theory presentation consisting of a set of symbol declarations and a set of axioms. Theorems can be deduced from axioms using inference rules. The Veritas system is supported by various software tools for establishing and handling the theory database using a functional programming language, ad hoc parsers, user-defined inference rules, and goal-directed theorem-provers. The Veritas approach is interesting from a methodological rather than from a practical point of view, examples being limited to very simple gates.

M.J.C. Gordon of the University of Cambridge, United Kingdom, is another British author who migrated to higher-order logic. We describe his earlier work in the section "Specific calculi." His formal system based on higher-order predicates is called HOL, but the same name is given to the automatic theorem-prover. HOL as a formal system is polymorphic (it allows types containing type variables) and has Hilbert's operator to build the choice axiom.

To avoid paradoxes, Gordon imposed some restrictions on terms (they must be "well-typed") and introduced some variations in terminology:

- A *sequent* is a set of assumptions from which a conclusion follows.
- A *theorem* either is an axiom or follows from other theorems by means of inference rules.
- A *theory* is a set of types, type operators, constants, definitions, axioms, and theorems.

In pure logic, a theory contains all possible theorems, whereas in HOL it contains only axioms and already proved theorems. Gordon gave two theories as primitive: Booleans and Individuals, introducing for the latter the axiom of Infinity to generate infinite sequences.

M.J.C. Gordon, J. Joyce, and G. Birtwistle of the University of Calgary, Canada, with the HOL system verified a microprocessor originally specified and verified with LCF-LSM (logic of computable functions/logic of sequential machines). D. May and D. Shepherd of Inmos, Bristol, United Kingdom, are using HOL to derive correct microcode for the IMS T800 floating-point transputer.

Evaluation. Higher-order predicates have more expressive power than first-order predicates, but the trade-off for this advantage involves more difficulty in performing proofs and mechanizing. The integration of higher-order predicates with functional approaches looks like a trend for future development.

Specific calculi. Many authors, especially in the domain of hardware verification, create their own formal systems reducible to first-order or higher-order predicates. We will call these formalisms "specific calculi." Such calculi are defined along the lines of logic by giving legal connectives, terms, and rules for constructing formulas and performing inferences.

Example. Consider the case of a unit-delay directional wire,[6] shown in Figure 6. This device operates synchronously within a discrete model for time according to the state-transition graph shown in the figure.

The device is defined by two input ports (α and t), an output port (β), and two states (W and W') through which the circuit evolves at every clock tick according to the events on the input port α. Input to α may only occur when the timer ticks, so an α signal only occurs with a t signal. The signal will propagate through the wire and is output on the next t signal, either on its own or simultaneously with a further input on α. The timer may tick without any input appearing, since it models the passage of real time (which cannot be halted).

$$W \leftarrow tW + (\alpha t)W'$$
$$W' \leftarrow (\beta t)W + (\alpha\beta t)W'$$

Research. G.J. Milne of the University of Edinburgh, United Kingdom, presented Circal, a calculus to describe and analyze circuit behavior. Circal is based on dot calculus, a descriptive language consisting of construction operators together with objects representing primitive concepts. Objects are composed to yield descriptions and communicate through labeled ports, called "sorts." Circal describes the behavior of a computing agent by the actions it wishes to perform with other agents in the environment. Circal descriptions are based on the notion of event, that is, a trigger that starts a particular evolution through time, resulting in a state.

When composing more devices, you often must reduce their complexity to keep them manageable. You can do this by hiding internal ports, making them invisible to an external observer.

The use of Circal is not restricted to a specific domain: it is both a very special hardware-description language and a framework for device analysis, in particular for formal verification of partial or total specifications and for "constructive" simulation. The last concept represents a variation with respect to standard simulation: instead of simulating a whole device composed of many elements, Milne simulated the elements one by one and composed the results. A proof demonstrated the equivalence of the standard and constructive approaches.

Circal does not just verify single designs, but also demonstrates the correctness of classes of designs generated by the same simple silicon compiler using NOR expressions as specifications. Circal tools in a Lisp-based environment and related prototype systems support expression manipulation and Circal simulation, but not correctness proofs as such.

An interesting library of LSI and VLSI components has been created and is reported in Circal.

M.J.C. Gordon produced the LCF-LSM system. LCF, or logic of computable functions, is a verification-oriented programming environment extensible to the manipulation of specifications. LSM, or logic of sequential machines, is a specification language for synchronous systems, currently used at the register-transfer and gate levels. The interface between user and LCF is provided by ML, an interpreted metalanguage with side effects similar to Lisp's. LSM is tied to OL, a predicative object language assuming some concepts from the CCS, or calculus of communicating systems, proposed by Milner.

LSM interfaces to LCF through four types: terms, constants, formulas, and theorems. In OL, theorems are derived from axioms through rules of inference. Rules allow folding and unfolding, renaming, combining, pruning, and unwinding equations. Axioms are grouped in theories, and theories in taxonomies. For the verification process, the user creates separate theories for specifications and implementations, and generates a proof interactively by directing the system to use rules. Gordon reported some examples of the application of his methodology to a simple computer and to an emitter-coupled-logic chip used in the Cambridge Fast Ring hardware.

Gordon's early work presented a formalism to describe synchronous circuits based on recursive functions and lambda calculus. Specifications are given behaviorally, while implementations are structural connections of modules with assumed primitive behavior. Elementary behaviors are composed using catenation, connection, and hiding operators. Syntactic identity of expressions demonstrates equivalence of implementation and specification.

Gordon introduced the concepts of *microcycles* and *macrocycles* to represent systems that do not work in lockstep, that is, systems where a single operation at a higher level is implemented by a sequence of operations at a lower one. He also introduced the induction principle to take into account complex cases. The paper reported some examples of application to register-transfer systems and NMOS circuits. Gordon's early work inspired H.G. Barrow, D. Weise, and J.L. Paillet.

H.G. Barrow of the Fairchild Laboratory for Artificial Intelligence Research, Palo Alto, California, presented a verification system written in Prolog, called Verify, which stemmed from Gordon's early work. Verify can check the correctness of finite-state machines. Behavioral specifications and structural implementations are both described in Prolog. The system extracts from the structure the derived behavior and compares it with the intended one. To prove the identity of specification and implementation, Verify uses symbolic manipulations, such as simplification, expansion, and canonicalization. When they become too heavy computationally, it resorts to evaluation or enumeration, and thus to exhaustive simulation on selected variables. The interactive system has been applied to verify the same simple computer presented by Gordon, and D74, a module for computing sums of products.

D. Weise of the Massachusetts Institute of Technology, Boston, Massachusetts, presented a methodology and a tool, called Silica Pithecus, to prove functional correctness for designs at the switch level. Although it refers to NMOS, this approach is almost technology-independent. An ad hoc language describes specifications, while implementations are described as schematics. The system extracts signals from the schematics, possibly still analog, and abstracts from these signals their digital behavior. Specified and intended behavior are compared to prove syntactic equivalence through symbolic manipulation and rewrite rules.

J.L. Paillet of the Université de Provence, Marseille, resorted to operative expressions to describe specifications and implementations. He then extracted the function of an implementation by composing the functions of the submodules and eliminating the internal variables. Eventually he compared, specified, and extracted behaviors to demonstrate equivalence. Internal variables are eliminated either by substitution or by a "development" operation (when looped). Concerning time, Paillet defined a "past" functional to allow references to past values of carriers. This approach focuses on synchronous circuits described at the register-transfer level. Mechanization is under way.

Evaluation. Specific calculi are stand-alone formal systems especially tailored for hardware verification. From a theoretical point of view, they resemble first- or higher-order predicates. Some authors abandoned them, since they considered the trade-off between ad hoc formalisms and costs unfavorable.

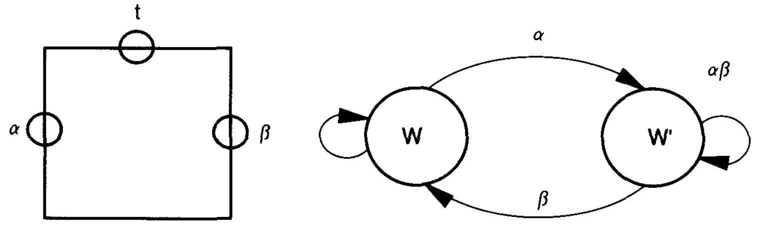

Figure 6. A unit-delay directional wire and its state-transition graph.

Temporal logic. In the domain of hardware representation, we must cope with a particular aspect of reality: time and temporal evolutions. Traditional logic is very powerful when dealing with static situations, but it fails when dealing with dynamic ones. To satisfy hardware's requirements, two choices seem possible:

(1) an explicit introduction of the time variable *t* into predicate logic, or
(2) a generalization of predicate logic to encompass the temporal domain.

Following the first path, authors introduce time functions, treating time just as any variable, with the usual rules for terms, formulas, and inference. Following the second path, authors augment standard logic to cover temporal evolution.

First-order and higher-order predicates stand for eternal truths. *Modal logic*, on the other hand, introduces the concepts of possibility and necessity in the future. Modal logic, although more expressive than traditional predicate logic, still lacks the ability to cope with changes, an essential feature in hardware behavior. To handle changes, we need a formal system that can reason from past events to what can or must be true at present and in the future. Such formal systems are generally grouped under the label "temporal logic."

Temporal logic includes all usual connectives and adds some typical operators. Although there are many variants, the basic operators are

- henceforth $\Box$
- eventually $\Diamond$
- next $\bigcirc$
- until $\cup$

We can classify temporal logic systems according to the way they consider time. The framework being generally a discrete model for time, there are different ways of considering the future. The past is always linear, while the future may be either a unique world or a set of possible worlds. In the first case, time is linear in the future, too; such logic is called "linear temporal logic." In the latter case, time branches in the future; such a system is called "branching temporal logic." In the former case, a system supposedly has a unique evolution along time, whereas in the latter case, a system has a set of possible evolutions.

Linear and branching time are not the only distinctions in temporal logic. Another important one is instant-based versus interval-based logic. Temporal logic is instant-based when propositions are asserted on single states (that is, instants in discrete time) only. Temporal logic is interval-based when propositions are asserted on sequences of states (that is, on intervals in discrete time). "Interval temporal logic" may be either global or local. It is global when the truth of propositions is determined over an entire interval, that is, on all its states. It is local when truth is determined on the first state of an interval and holds unchanged on the rest.

Linear temporal logic example. Consider part of the handshaking protocol[7] in Figure 7. Explanations appear between quotes.

$\Box(\sim \mathit{Hear} \rightarrow \Diamond \mathit{Call})$

"if Hear is low, then sooner or later Call will rise"

$\Box(\mathit{Call} \rightarrow \mathit{Call} \cup \mathit{Hear})$

"if Call is high, it will stay high until Hear is high"

$\Box(\mathit{Call} \rightarrow \Diamond \mathit{Hear})$

"if Call is high, then sooner or later Hear will rise"

$\Box(\sim \mathit{Call} \rightarrow \Diamond \sim \mathit{Hear})$

"if Call is low, then sooner or later Hear will fall"

Research. G.V. Bochmann of the University of Montreal, Canada, presented one of the first attempts to use temporal logic as a formalism to describe and reason about hardware. In his approach, time is linear and discrete. He presented some

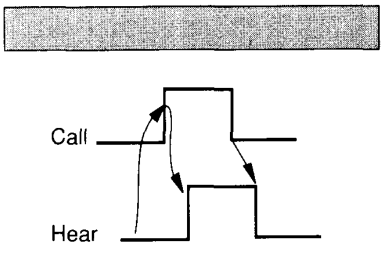

Figure 7. A handshaking protocol.

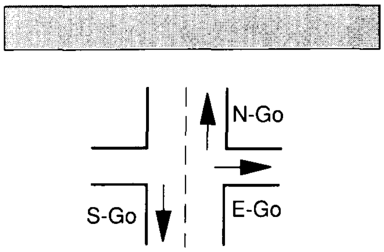

Figure 8. A traffic-light controller.

of the axioms bound to the modalities he used, then described and verified an arbiter. Verification relies upon a kind of reachability analysis, in which possible states and transitions are generated and symbolically executed.

M. Fujita, H. Tinker, T. Moto-oka, and S. Nishiyama of the University of Tokyo, Japan, presented an approach to formal verification based on linear temporal logic and Prolog. The domain of application is currently restricted to finite-state machines used to implement the synchronization part of a complex system. Implementations are described in standard HDLs, such as HSL or a subset of DDL, at the gate or register-transfer level. Specifications are expressed in temporal logic with weak and strong until (U) operators. Implementations are automatically translated into C-Prolog clauses, while specifications are transformed into state diagrams using temporal logic's decision procedure. This state diagram and the implementation are simultaneously traversed to look for a counterexample, where negated specifications satisfy the implementation. Some features expedite the system's efficiency, including filtering of descriptions, storing state transitions, and using external specifications. The authors used this approach for automatic synthesis of state diagrams starting from temporal logic specifications.

Branching temporal logic example. Consider the expression of some safety and liveness properties for a traffic-light controller[8] (see Figure 8). The traffic-light controller is stationed at the intersection of a two-way highway going north and south and a one-way road going east. No turns are permitted. At the north, south, and east of this intersection, a sensor goes high for at least one clock cycle when a car arrives. When the intersection is clear of cross traffic, the controller raises a signal indicating that the car may cross the intersection. Once the car has crossed, the sensor that indicated the arrival will go low. The sensors are named *N, S,* and *E*; the output signals for each end of the intersection are *N-Go, S-Go,* and *E-Go.*

A description in branching temporal logic follows:

$\forall G \sim (E\text{-}Go \wedge (N\text{-}Go \vee S\text{-}Go))$

"for every path in the future, at every node on the path, there will never be green signs in both directions"

This formula is a safety property that is true if the controller does not permit collisions.

The following liveness properties state that every request to enter the intersection is eventually answered, so the controller is starvation-free. If all three of these formulas are true, the controller is deadlock-free as well.

$\forall G (\sim N\text{-}Go \wedge N \rightarrow \forall F\, N\text{-}Go)$

"for every path in the future, at every node on the path, if the light is red and there is a northbound car requesting to cross, then for every path there will be at least a node on the path where the *N-Go* signal will be high"

$\forall G (\sim S\text{-}Go \wedge S \rightarrow \forall F\, S\text{-}Go)$

"for every path in the future, at every node on the path, if the light is red and there is a southbound car requesting to cross, then for every path there will be at least a node on the path where the *S-Go* signal will be high"

$\forall G (\sim E\text{-}Go \wedge E \rightarrow \forall F\, E\text{-}Go)$

"for every path in the future, at every node on the path, if the light is red and there is an eastbound car requesting to cross, then for every path there will be at least a node on the path where the *E-Go* signal will be high"

$\exists F (N\text{-}Go \wedge S\text{-}Go)$

"for some path there is a state on the path where both the *N-Go* and the *S-Go* signals will be high"

Research. E.M. Clarke, B. Mishra, D.L. Dill, M. Browne, E.A. Emerson, and A.P. Sistla of Carnegie Mellon University, Pittsburgh, presented an approach and the supporting tool for formal verification of synchronous and asynchronous control parts of digital systems. Implementations are described by either structural HDLs or an ad hoc language called SML, for state-machine language. Specifications are expressed in the temporal domain using computation tree logic, or CTL, a branching time propositional temporal logic. CTL extends all temporal logic operators to take into account the introduction of *possibility* in the future: a circuit may evolve along many possible paths. Implementations expressed in structural form are automatically transformed into a state-transition graph (a finite Kripke structure) by means of simulation. If the implementation is already expressed as a finite-state machine, the state-transition graph generation process is almost immediate. Verification relies upon state-graph transition, possibly exhaustive, looking for a counterexample of the specification.

The tool supporting this approach, called MC for model checker, comes in a C-language version and a Franz-Lisp version. Since state-graph complexity increases exponentially with the size of designs, the authors proposed multilevel verification to keep the task manageable. Lower-level modules are composed to yield complex ones, thereby shrinking complexity by restriction. In a later version, called EMC for extended model checker, the updated tool deals with a more expressive branching time temporal logic called CTLF. The authors have applied their approach to a FIFO cell and to an asynchronous communication interface adapter.

Interval time temporal logic example. Consider the expression of a positive pulse signal[9] with quantitative timing information (see Figure 9).

The corresponding interval temporal logic formula looks as follows:

$$\begin{aligned}&\uparrow\downarrow^{l,m,n} X \equiv\\ &[(x \approx 0 \wedge \mathit{minlen}(l)); \mathit{skip};\\ &[(x \approx 1 \wedge \mathit{minlen}(m)); \mathit{skip};\\ &[(x \approx 0 \wedge \mathit{minlen}(n))]\end{aligned}$$

The construct *minlen*(n) is true for an interval at least n units long. *Skip* is a construct to test for intervals having length one.

Research. B. Moszkowski of the University of Cambridge, United Kingdom, presented a general-purpose formalism to specify hardware behavior, called ITL for interval temporal logic. ITL is general-purpose because you can use it to describe register-transfer operations, as well as flow graphs and transition tables. ITL's syntax allows variables, expressions, and formulas. Each formula is associated with a sequence of states (points in discrete time). A function maps variables, formulas, and states into data values. Through this mapping function, it is possible to define when a sequence of states satisfies a formula. ITL is local in that truth or falsity of a formula depends only on the value the mapping function returns on the first state of the interval. Some operators have an interval-dependent meaning, which distinguishes ITL from all other temporal logic formalisms.

Moszkowski presented Tempura, a logic programming language with imperative constructs for assignment, as a logical consequence of his work. Tempura functions as a conventional programming language or as an HDL for structural descriptions. A ring network and a multiplier exemplify its use in the hardware description domain. As a logic programming language, Tempura resembles Prolog, but has no incorporated unification and backtracking facilities. It is supported by a software tool implementing its interpreter. This evaluates formulas, splitting them recursively into values at present and next state, in a way similar to temporal logic's decision procedure.

Moszkowski did not present evidence for the utility of his approach in the domain of formal verification of hardware; rather, he claimed that such an extension would not be difficult.

Within the major framework introduced by Moszkowski, T. Aoyagi, M. Fujita, T. Moto-oka, S. Kono, and H. Tinker of the University of Tokyo presented Tokio, a concurrent logic programming language. Although based on Prolog, it extends it by incorporating local ITL concepts. Since Prolog lacks the concept of time, assignment occurs as a tricky side-effect. This is Prolog's main disadvantage with respect to conventional programming languages. Tokio introduces a discrete time model, with intervals defined by their starting and final events. ITL operators have been added to Prolog, and the interpreter can be directed to execute goals in specified intervals. The authors presented no examples of Tokio's application to formal verification, but we expect them in the near future.

Evaluation. Temporal logic substantially advances traditional logic because it can capture time and dynamic behaviors—essential features in hardware descriptions—with concise and clear notation. For example, it avoids the introduction of explicit time functions and time variables. Moreover, temporal logic is useful in different environments, such as programming, although within the domain of hardware verification no one has yet exploited its full power. Timing anomalies, races, and hazards could be described easily, but we are not aware of similar research efforts.

A lively debate among temporal logicians focuses on the relative merits of linear and branching time temporal logic. The common opinion is that linear time temporal logic has more expressive power and branching time temporal logic has more powerful decision procedures. The choice between the two strongly depends on the applications.

Cross-fertilization with software verification techniques

Since software verification is older than its hardware counterpart, one of the first approaches for hardware adapted some software verification techniques to the particular domain under consideration.

Programming languages constitute immediately available description formalisms. Although not, strictly speaking, formal systems, they have been extended for hardware description and verification in conjunction with logic. Most descriptions of hardware are procedural; that is, a locus of control governs the evaluation. This heavily restricts the application domain of such formalisms. In fact, they are used only at very high levels of design, namely those in which behavior is described algorithmically. Procedural descriptions allow the expression of some useful features typical of a software verification environment, such as termination properties, preconditions, and postconditions. Normally, you would resort to predicative forms to express these features, which play the role of specifications or properties, while procedural descriptions represent implementations.

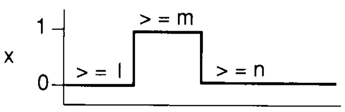

Figure 9. A pulse with timing details.

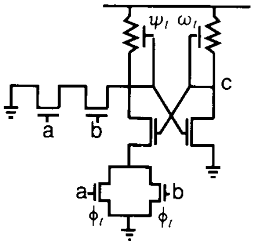

Figure 10. A Muller C-element.

Examples. One example of the application of Floyd's inductive assertion method considers the expression of preconditions, postconditions, and loop assertions on a Muller C-element[10] (see Figure 10). We can easily state the operation of the C-element. The device has two binary input signals, *a* and *b*, and a binary output signal, *c*. The output becomes 1 whenever both inputs become 1; it becomes 0 whenever both inputs become 0. Otherwise, it stays the same. Since no particular assumptions are made about the input signals *a* and *b*, the corresponding assertion

ϕ is true at every time t. Because the circuit has only one cycle, a single loop assertion ψ suffices and is coincident with the output assertion ω. The proof employs the STP theorem prover.

$$\phi_t = \text{true}$$

$$\omega_{t+2} = \psi_{t+2} = \text{if } (a_t = b_t) \text{ then } a_t \text{ else } \omega_t$$

Research. R.E. Shostak of SRI International, Menlo Park, California, modeled hardware as a graph with circuit elements for nodes and signals for arcs. Each circuit element is characterized by a transfer predicate, which expresses how the element transforms its inputs to yield its outputs. Assertions are placed on inputs, outputs, and loops. As in Floyd's method, verification conditions are generated by tracing back output assertions to the inputs and comparing them with input assertions using the STP theorem-prover. To ease the task, acyclic subgraphs are extracted from the complete graph and verified one by one. Shostak's examples concerned some analog and asynchronous circuits, without explicitly considering time or reporting any automation.

A very similar approach taken by V. Pitchumani, E.P. Stabler, and Z.D. Umrigar of Syracuse University, Syracuse, New York, featured assertions, preconditions, postconditions, and a theorem-prover. The authors used a register-transfer language similar to Pascal and AHPL to describe hardware. This language allows concurrent constructs, and the authors described proof methods for parallel control sequences. The domain of application of such a method is limited to synchronous circuits. The authors reported no mechanization.

An approach based on Floyd's inductive assertion method but with interesting variations comes from M.C. McFarland and A.C. Parker of Carnegie Mellon University, Pittsburgh. Instead of verifying that a design implementation complies with its specification, they considered a more general case. They demonstrated that certain transformations used to optimize designs are correct, so that the behaviors of original and transformed designs are equivalent.

McFarland and Parker used the description language called ISPB, a subset of ISPS. ISPB is axiomatized and has a set of rules used to manipulate expressions. Like ISPS, it is procedural but allows concurrent statements. The way hardware behaves is captured by its interactions with the environment, which the authors called *events*. Sequences of events represent *histories*, used to give meaning to descriptions. Specifications are expressed as *behavior expressions*, that is, regular expressions augmented by predicates. The system was developed within the Carnegie Mellon University Design Automation, or CMUDA, project.

Evaluation. Applying software verification techniques to hardware suffers from some drawbacks. First, devices are described in a procedural fashion and only at very high levels of abstraction (algorithmic and behavioral). Moreover, this approach follows closely the evolution of its software counterpart, including its difficulties. Thus, such research represents a minority of the current efforts in this area.

Simulation has evident theoretical limits, but thanks to the efficiency of its tools, it is widely used in commercially available systems. Although they overcome simulation's nonexhaustivity, formal techniques introduce additional concerns: they need formal description and proof systems.

Formal techniques are now mainly academic efforts, with the majority of research efforts located in Europe. US and Japanese researchers seem to concentrate more on automated synthesis.

Let us now analyze current work. It looks like a big research effort is under way: a lot of projects are already under development and new ones are continually being started. Nevertheless, it proves very difficult to benchmark different approaches, and not just because of the vast amount of material to take into consideration. We deliberately avoided presenting comparisons or figures lacking merit from a theoretical or practical point of view. The domains to which the authors applied their methodologies differ markedly, ranging from the switch to the system level. Each author tailored the formal system and proof method to his domain, sometimes losing generality and applicability. For example, formal systems especially intended to model synchronous circuits are not easily extendible to asynchronous ones, and vice versa.

Another observation concerns the domain of application: many authors restricted their examples to "special" circuits, that is, either circuits already well-suited for verification or old-fashioned gate-level designs, bound more to MSI/LSI than to VLSI technology. Moreover, the authors seldom considered gate arrays and programmable logic devices despite their growing importance in the world of application-specific IC design, nor did they take into account low-level timing issues, such as races and hazards.

From a practical point of view, formal techniques face a major limitation: the majority of approaches are purely theoretic. Even when some software tool is implemented, it is in general a prototype, its performance is hard to evaluate, and it cannot be easily incorporated in commercial systems.

Although formal verification techniques suffer from evident limits, we believe that they are important not only from a theoretical, but also from a practical, point of view. The ability to verify a design implies understanding of its profound semantic meaning; this helps both manual design and automated synthesis. Tools are becoming more efficient as we gain in expertise and take advantage of recent developments in parallel processing and innovative architectures. Within a few years, perhaps five or six, we can hope to see commercial systems including formal techniques, although it is hard to believe that they will be general-purpose.

Industry, abandoning its initial skepticism, is becoming more and more interested in formal verification, since it can guarantee correct designs and shave costly development time. Some major European manufacturers plan to include in their private CAD systems some of the formal verification tools currently under development.

If formal verification keeps in touch with the latest developments in VLSI, so computer-aided design does not lag behind design, both fields will benefit and contribute significantly to the advancement of computer science. □

Acknowledgments

We are grateful to all those people who kindly provided us with the papers, reports, and material used throughout this article. In particular, we would like to thank Dominique Borrione, Mario Barbacci, Hans Eveking, and the referees for their help in reviewing the article, their suggestions, and their valuable cooperation.

References

1. J.A. Darringer, "The Application of Program Verification Techniques to Hardware Verification," *Proc. ACM IEEE 16th Design Automation Conf.*, June 1979, pp. 375-381.
2. J.E. Donahue, *Complementary Definitions of Programming Language Semantics*, Springer-Verlag, Berlin, 1976.
3. J. Barwise, *Handbook of Mathematical Logic*, North-Holland Publishing, Amsterdam, 1977.
4. H. Eveking, "Formal Verification of Synchronous Systems," *Formal Aspects of VLSI Design: Proc. 1985 Edinburgh Conf. VLSI*, G.J. Milne and P.A. Subrahmanyam, eds., North-Holland Publishing, Amsterdam, 1986, pp. 137-151.
5. M.J.C. Gordon, "Why High-Order Logic Is a Good Formalism for Specifying and Verifying Hardware," *Formal Aspects of VLSI Design: Proc. 1985 Edinburgh Conf. VLSI*, G.J. Milne and P.A. Subrahmanyam, eds., North-Holland Publishing, Amsterdam, 1986, pp. 153-177.
6. G.J. Milne, "Circal: A Calculus for Circuit Description," *Integration, the VLSI J.*, July 1983, pp. 121-160.
7. M. Fujita, H. Tinker, and T. Moto-oka, "Verification with Prolog and Temporal Logic," *Proc. CHDL '83: IFIP 6th Int'l Symp. Computer Hardware Description Languages and their Applications*, Pittsburgh, May 1983, pp. 105-114.
8. M. Browne et al., "Automatic Verification of Sequential Circuits Using Temporal Logic," *IEEE Trans. Computers*, Dec. 1986, pp. 1035-1044.
9. B. Moszkowski, "A Temporal Logic for Multilevel Reasoning about Hardware," *Computer*, Feb. 1985, pp. 10-19.
10. R.E. Shostak, "Formal Verification of Circuit Designs," *Proc. CHDL '83: IFIP 6th Int'l Symp. Computer Hardware Description Languages and their Applications*, Pittsburgh, May 1983, pp. 13-30.

Further reading

This section presents some additional reading for those who want a deeper understanding of some of the approaches discussed in the article. References are grouped according to topic. They represent a summary of each of the research efforts covered here.

Symbolic simulation

Carter, W.J., W.H. Joyner, and D. Brand, "Symbolic Simulation for Correct Machine Design," *ACM IEEE 16th Design Automation Conf.*, June 1979, pp. 280-286.

Cory, W.E., "Symbolic Simulation for Functional Verification with Adlib and SDL," *ACM IEEE 18th Design Automation Conf.*, June 1981, pp. 82-89.

First-order predicate calculus

Barros, J.C., and B.W. Johnson, "Equivalence of the Arbiter, the Synchronizer, the Latch, and the Inertial Delay," *IEEE Trans. Computers*, July 1983, pp. 603-614.

Hunt, W.A., "FM8501: A Verified Microprocessor," *IFIP WG 10.2 Workshop, From HDL Descriptions to Guaranteed Correct Circuit Designs*, North-Holland Publishing, Amsterdam, Sept. 1986, pp. 85-114.

Suzuki, N., "Concurrent Prolog as an Efficient VLSI Design Language," *Computer*, Feb. 1985, pp. 33-40.

Uehara, T., et al., "DDL Verifier and Temporal Logic," *CHDL '83: IFIP 6th Int'l Symp. Computer Hardware Description Languages and their Applications*, May 1983, pp. 91-102.

Wagner, T.J., "Verification of Hardware Designs Through Symbolic Manipulation," Int'l Symp. Design Automation and Microprocessors, Feb. 1977, pp. 50-53.

Wojcik, A.S., "A Formal Design Verification System Based on an Automated Reasoning System," *ACM IEEE 21st Design Automation Conf.*, July 1984, pp. 641-647.

Higher-order predicates

Hanna, F.K., and N. Daeche, "Specification and Verification of Digital Systems using Higher-Order Logic," *IEE Proc.*, Vol. 133, Pt. E, No. 5, Sept. 1986, pp. 242-254.

Specific calculi

Barrow, H.G., "Verify: A Program for Proving Correctness of Digital Hardware Designs," *Artificial Intelligence*, Vol. 24, 1984, pp. 437-491.

Gordon, M.J.C., "LCF-LSM," *Tech. Report No. 41*, Computer Laboratory, Univ. of Cambridge, Cambridge, United Kingdom, 1983.

Paillet, J.L., "A Functional Model for Descriptions and Specifications of Digital Devices," *IFIP WG 10.2 Workshop: From HDL Descriptions to Guaranteed Correct Circuit Designs*, Sept. 1986, Grenoble, France, pp. 19-40.

Weise, D.W., "Automatic Formal Verification of Synchronous MOS VLSI Designs," verbal presentation to IFIP WG 10.2 Workshop on Formal Verification, Darmstadt, West Germany, Nov. 1984.

Temporal logic

Bochmann, G.V., "Hardware Specification with Temporal Logic: An Example," *IEEE Trans. Computers*, Mar. 1982, pp. 223-231.

Fujita, M., H. Tinker, and T. Moto-oka, "Logic Design Assistance with Temporal Logic," *CHDL '85: IFIP 7th Int'l Symp. Computer Hardware Description Languages and their Applications*, Aug. 1985, pp. 129-137.

Software verification techniques

Floyd, R.W., "Assigning Meanings to Programs," *Int'l Symp. Applied Mathematics*, Vol. 19, Mathematical Society, 1967, pp. 19-32.

McFarland, M.C., and A. Parker, "An Abstract Model of Behavior for Hardware Description," *IEEE Trans. Computers*, July 1983, pp. 621-637.

Pitchumani, V., and E.P. Stabler, "Verification of Register Transfer Level Parallel Control Sequences," *IEEE Trans. Computers*, Aug. 1985, pp. 761-765.

Shostak, R.E., "Formal Verification of Circuit Designs," *CHDL '83: IFIP 6th Int'l Symp. Computer Hardware Description Languages and their Applications*, May 1983, pp. 13-30.

Paolo Camurati is currently a PhD student in the Dept. of Computer Science and Automation at the Politecnico di Torino. He received the MS degree in electronic engineering from that institute in 1984.

Camurati's interests include CAD for VLSI, including hardware description languages, design for testability, testing and ATPG, and formal verification; and application of AI techniques to CAD, CAT, and CAR.

Paolo Prinetto has since 1978 been assistant professor and since 1982 researcher at the Dept. of Computer Science and Automation at the Politecnico di Torino. Also, he is currently a full professor at the University of Udine. In 1980 he joined the EEC Study Team on CAD for VLSI: Languages and Data Structures. In July 1983 he joined IFIP Working Group 10.2. He has been the leader of the ART* simulation system project.

Prinetto's interests include CAD for VLSI, including hardware description languages, design for testability, testing and ATPG, and formal verification; application of AI techniques to CAD, CAT, and CAR; and microprogramming, including microinstruction modeling and simulation, integrated CAD systems for microprogrammable architecture development, and automated microprogram synthesis.

Prinetto received the MS degree in electronic engineering from the Politecnico di Torino in 1976.

Readers may write to the authors at the Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi 24, I-10129 Turin, Italy. Their electronic mail address is cao@itopoli.bitnet.