CAMBRIDGE

Cambridge University Press
978-0-521-11532-2 - Higher Order Logic and Hardware Verification
T. Melham
Excerpt
More information

Chapter 1

Hardware Verification

Continued advances in microelectronics have allowed hardware designers to build devices of remarkable size and complexity. With increasing size and complexity, however, it becomes increasingly difficult to ensure that these devices are free of design errors. Exhaustive simulation of even moderately-sized circuits is impossible, and partial simulation offers only partial assurance of correctness.

This is an especially serious problem in safety-critical applications, where failure due to design errors may cause loss of life or extensive damage. In these applications, functional errors in circuit designs cannot be tolerated. But even where safety is not the primary consideration, there may be important economic reasons for doing everything possible to eliminate design errors, and to eliminate them early in the design process. A flawed design may mean costly and time-consuming refabrication, and mass-produced devices may have to be recalled and replaced.

A solution to these problems is one of the goals of formal methods for verification of the correctness of hardware designs—sometimes just called *hardware verification.* With this approach, the behaviour of hardware devices is described mathematically, and formal proof is used to verify that they meet rigorous specifications of intended behaviour. The proofs can be very large and complex, so mechanized theorem-proving tools are often used to construct them.

Considerable progress has been made in this area in the past few years. Notable large-scale applications of hardware verification include Graham's verification of the design of an SECD chip using the HOL proof assistant [8], Joyce's verification in HOL [78, 79] of a simple microprocessor originally verified by Gordon using the LCF_LSM theorem prover [44], Hunt's verification of the FM8501 microprocessor using the Boyer-Moore theorem prover [73], and the verification by Narendran and Stillman of an image processing chip using a method based on rewriting [101].

Hardware verification techniques are now considered mature enough to be applied to simple circuits intended for safety-critical applications. Cohn's work on verifying the Viper microprocessor is a preliminary experiment in this area [29, 27]. Viper is a simple microprocessor designed at the Royal Signals and Radar Establishment with formal verification in mind and intended for use in safety-critical applications [31]. Cohn's formal proof of the correctness of Viper is not complete, but it covers some important aspects of the machine's design.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-11532-2 - Higher Order Logic and Hardware Verification
T. Melham
Excerpt
More information

1.1 The hardware verification method

Two things are needed for any method of hardware verification based on rigorous specification and formal proof. The first is a formal language for describing the behaviour of hardware and expressing propositions about it. The ideal language is expressive enough to describe hardware in a natural and concise notation yet still has a well-understood and reasonably simple semantics. The second requirement is a deductive calculus for proving propositions expressed in this language. This must, of course, be logically sound, and it should be powerful enough to allow one to prove all the true propositions about hardware behaviour that arise in practice.

Various formal languages and associated proof techniques have been proposed as a basis for hardware verification. These range from special-purpose hardware description languages with *ad hoc* proof rules to systems of formal logic and subsets of ordinary mathematics. Formal methods for reasoning about hardware behaviour have been based, for example, on algebraic techniques [6, 22, 90], various kinds of temporal logic [11, 33, 80, 100], functional programming techniques [105], predicate calculus [34, 71, 73, 113], and higher order logic [21, 45, 57, 81].

The details of the verification methods based on these different formalisms vary, but many of them share a common general approach. This typically involves the following four steps.

1. Write a formal specification S to describe the behaviour that the device to be verified must exhibit for it to be considered correct.

2. Write a specification for each kind of primitive hardware component used in the device. These specifications are intended to describe the actual behaviour of real hardware components.

3. Define an expression D which describes the behaviour of the device to be proved correct. The definition of D has the general form

   $$D \stackrel{\text{def}}{=} P_1 + \cdots + P_n$$

   where $P_1, \ldots, P_n$ specify the behaviours of the constituent parts of the device and $+$ is a *composition operator* which models the effect of wiring components together. The expressions $P_1, \ldots, P_n$ used here are instances of the specifications for primitive devices defined in step 2.

4. Prove that the device described by the expression D is correct with respect to the specification S. This is done by proving a theorem of the form

   $$\vdash D \text{ satisfies } S$$

   where 'satisfies' is some *satisfaction relation* on specifications of hardware behaviour. This correctness theorem asserts that the behaviour described by D satisfies the specification of intended behaviour S.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-11532-2 - Higher Order Logic and Hardware Verification
T. Melham
Excerpt
More information

When the device to be proved correct is large, this method is usually applied hierarchically. The design is structured into a hierarchy of components and sub-components, and specifications that describe 'primitive components' at one level of the hierarchy then become specifications of intended behaviour at the next level down. The structure of the proof mirrors this hierarchy: the top-level specification is shown to be satisfied by an appropriate connection of components; at the next level down, each of these components is shown to be correctly implemented by a connection of sub-components, and so on—down to the lowest level, where the components used correspond to devices available as hardware primitives.

1.2 Limitations of hardware verification[1]

A correctness proof cannot guarantee that a real device will never malfunction; the *design* of a device may be proved correct, but the hardware actually built can still behave in ways not intended by the designer. There are some obvious reasons for this. There may, for example, be defects in fabrication. Or, since the theorem-proving tools used for hardware verification are not yet integrated with the CAD systems used to generate circuit layouts, the design which is verified may not correspond exactly to the device which is built. But in addition to these pragmatic problems, there are also two more fundamental reasons why a physical device, although built to a verified design, may still fail to behave as intended by the designer. Both are results of obvious but important limits to what can be established by formal proof.

First, a proof can never demonstrate that a circuit will behave 'as intended' by the designer; a proof can show only that its design behaves as prescribed by a written, and possibly inaccurate, specification. A 'verified' device may therefore exhibit unexpected behaviour because its design was proved correct with respect to a specification that fails to reflect the designer's intent. This is an obvious point. But it is an especially important one when the specification is large or the behaviour being specified is complex, for then it may be far from clear to the designer that the specification is itself correct.

Second, a formal proof cannot, of course, demonstrate anything about physical reality. All that can be proved is that a mathematical *model* of a device has the properties prescribed by a specification of required behaviour. But a model may fail to capture important aspects of the real hardware it is intended to describe, and design errors can therefore escape discovery because the undesirable behaviour resulting from them is not reflected in this model.

Because of these fundamental limits to the scope of formal proof, verification can never establish with complete certainty that a physical device will function exactly as intended. A correctness theorem can be only as good as the model and the

[1] See also Cohn's discussion of the limitations of hardware verification in the context of the Viper verification project [28]. For a discussion of certain extra-logical aspects of Viper, see [85].

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-11532-2 - Higher Order Logic and Hardware Verification
T. Melham
Excerpt
More information

specification it relates. There are, however, two important practical measures that can be taken to help justify confidence in the significance of such results. First, specifications can be made as clear and concise as possible, so that their fidelity to the designer's intent can be evaluated easily. This reduces the likelihood of a proof failing to yield a meaningful result because the specification is itself incorrect.[2] Second, models can be used that describe the empirical behaviour of real devices as accurately as possible. The more accurate the model, the less likely it is that a design error will escape discovery.

Adopting these two measures is essential if design verification by formal proof is to be an effective way of dealing with the problem of hardware correctness. An important consequence is that the notion of *abstraction* must play a central role in effective formal methods for hardware verification.

1.3 Abstraction

Abstraction is the process by which the important properties of a complex object are isolated for further use and the remaining ones ignored as being irrelevant to the task at hand. An example is the process of procedural abstraction in programming. Once a procedure has been defined, it is treated as an atomic operation with only one important attribute, namely *what* the procedure does. The exact sequence of computation steps which achieve this operation—*how* the procedure does it—is ignored [50]. Programming language constructs that support abstraction in this way are fundamental tools for dealing with the complexity of programming. By allowing the attributes of a complex object which are important to be isolated from those which are not, software abstraction mechanisms provide a way of limiting the amount of detail that must be considered at any one time [52].

Abstraction plays a similar role in hardware verification. Here, an abstraction mechanism establishes a relationship of abstraction between a complex description of hardware behaviour and a simpler one. This provides a means for controlling the complexity of both specifications and proofs of correctness. By suppressing the irrelevant information in detailed descriptions of hardware behaviour, and thereby isolating the properties of these descriptions which are most important, effective abstraction mechanisms help reduce the size and complexity of the specifications at each level in a hierarchically-structured correctness proof. Abstraction mechanisms and hierarchical structuring help to control the complexity of proofs in the same way that software abstraction mechanisms are used to manage the complexity of program development.

Considered in the context of the limitations of proof discussed above, it is clear

[2] Another approach is to make specifications executable, so that the designer can run them to gain confidence in their accuracy (see, for example, [20, 92, 110]). This is a special case of the more general strategy of evaluating specifications by deriving consequences of them.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-11532-2 - Higher Order Logic and Hardware Verification
T. Melham
Excerpt
More information

that abstraction must play a central role in two areas of hardware verification—first, in formulating meaningful assertions about correctness, and second, in assessing the accuracy of the models on which correctness proofs are based. These two aspects of the role of abstraction in hardware verification are discussed briefly below.

1.3.1 Abstraction and correctness

In proving the correctness of a design, it is generally desirable to use a model whose mathematical properties reflect as accurately as possible the empirical behaviour of the physical device itself. This does not mean that it is always necessary to model the physics of electronic circuits in as much detail as possible; a simplified model can often be justified by the fact that the device to be verified is implemented in a particular technology or design style. But a model should nonetheless reflect as accurately as possible the actual behaviour of a device built using the technology or design style in question.

Specifications, on the other hand, must be clear and concise, so that they are intelligible enough to be seen to reflect the designer's intent. Most of the details about the actual behaviour of a device must therefore be left out the specification of its intended behaviour. Only the essential aspects of its required behaviour can be included. Furthermore, the larger and more complex the device being verified, the more detailed information about its actual behaviour must be ignored in order to keep the specification small.

This means that specifications must, in general, present more abstract views of the behaviour of devices than models do. The concept of a relationship of abstraction between two formal descriptions of hardware behaviour is therefore fundamental to expressing what it means for a device to be 'correct'. Formally, the correctness of a device is stated by a theorem which asserts that a mathematical model of its actual behaviour in some sense 'satisfies' a specification of its intended behaviour. For all but the simplest devices, the satisfaction relation used to formulate this correctness relationship must relate a detailed design model to a more abstract specification. This notion of correctness as a relationship of abstraction is explored in detail in this book.

1.3.2 Abstraction and the accuracy of models

Although the value of a correctness result depends on how accurately a model reflects the actual behaviour of the physical device it represents, a very accurate model of hardware behaviour may be unnecessarily complex. It may be possible to adopt a circuit design style for which a simpler model will do. The functional correctness of a fully complementary CMOS circuit, for example, does not critically depend on transistor size ratios [114]. A very accurate model of CMOS transistor behaviour, which takes into account transistor size ratios, would therefore be inappropriate for this conservative design style. In this case, a less accurate—but also simpler and

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-11532-2 - Higher Order Logic and Hardware Verification
T. Melham
Excerpt
More information

more tractable—model can be used. The validity of using this simpler model can be justified on empirical grounds, in the light of what is known about the actual behaviour of fully complementary CMOS circuit designs.

In general, a simplified model can often be justified empirically by the fact that the device which is to be proved correct is implemented in a restricted style of circuit design. Such a model may be less accurate than a more complex one, but the restrictions on device behaviour imposed by the design style itself will ensure that the extra accuracy of a more complex model is not needed. In this case, the simple model of hardware behaviour will in fact be an abstraction of a more accurate model. Both models will describe the same design, but the simpler model will describe only some of the aspects of device behaviour that are captured by the more accurate model.

Although it may be possible to justify this use of a simplified model of device behaviour empirically, it is also desirable to assess the accuracy of models by more rigorous means. This can be done by means of a proof which demonstrates that a simplified model is sufficiently accurate (with respect to a more complex model) for the particular circuit design style in question. Such a proof shows that, for a restricted class of circuit designs, a simple model is in some sense a valid abstraction of a more complex model. This notion of an abstraction relationship between models of hardware behaviour and the connection between this idea and the concept of the relative accuracy of models are discussed in detail in chapters 4 and 7.

1.4 Hardware verification using higher order logic

The choice of a formalism for hardware verification involves a compromise between expressive power and ease of proof. A simple and restricted formalism may make the proofs of some devices easy, but it may also make it hard to specify complex devices easily and concisely. A powerful formalism will be expressive enough for a wide class of devices but may make proofs difficult. The formalism used for hardware verification in this book is a version of type theory called *higher order logic* [24, 41]. Higher order logic is a foundational formalism that makes available the results of general mathematics. This allows one (in principle) to construct within the logic whatever mathematical tools are needed for the verification task at hand.

Chapters 2 and 3 introduce higher order logic and explain the basics of hardware verification using this formalism. One of the main aims of this book is to give a clear account of the role of abstraction in hardware verification and to show how certain fundamental abstraction mechanisms can be formalized in higher order logic. Chapter 4 provides a general account of the basic principles of abstraction, and chapters 5–7 give a selection of examples to illustrate these principles. The aim of these examples is to provide clear illustrations of some specific aspects of hardware verification using higher order logic. The examples are therefore generally very simple. The reader should, however, be able to see how the methods illustrated by

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-11532-2 - Higher Order Logic and Hardware Verification
T. Melham
Excerpt
More information

these examples can be applied on a larger scale.

Two fundamental principles underlie the approach taken in this book. The first is that all reasoning about hardware should be done by strictly formal proof, using only the deductive calculus provided by the primitive basis of higher order logic. This provides the greatest possible assurance that only logically sound reasoning is used. The second principle is that any special-purpose notation needed to specify hardware behaviour and to formulate correctness should be introduced by definitional means only. That is, the syntax of the logic is extended with new notation not simply by postulating axioms to give meaning to it, but rather by defining it in terms of existing expressions of the logic that already have the required semantics. This ensures that inconsistency is not introduced by postulating *ad hoc* axioms intended to characterize non-primitive mathematical objects.

An important consequence of adopting these two principles is that mechanized theorem proving support becomes essential in practice, since only for the simplest theorems of higher order logic is it feasible to carry out fully detailed formal proofs manually. The HOL theorem-prover [42, 47] is a proof generating system suitable for conducting large-scale proofs in the version of higher order logic used in this book. The basic ideas behind the HOL system are explained in chapter 2. It is, however, beyond the scope of this book to discuss theorem-proving in HOL in detail; for this, the interested reader should consult the HOL system manual [47]. A substantial case study is described in [49].

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-11532-2 - Higher Order Logic and Hardware Verification
T. Melham
Excerpt
More information

Chapter 2

Higher Order Logic and the HOL System

This chapter provides an overview of the formulation of higher order logic used in this book for reasoning about hardware. A brief account is also given of the mechanization of this logic in the HOL theorem proving system.

The version of higher order logic described in this chapter was developed by Mike Gordon at the University of Cambridge [41]. Gordon's version of higher order logic is based on Church's formulation of simple type theory [24], which combines features of the λ-calculus with a simplification of the original type theory of Whitehead and Russell [115]. Gordon's machine-oriented formulation extends Church's theory in two significant ways: the syntax of types includes the polymorphic type discipline developed by Milner for the LCF logic PPλ [48], and the primitive basis of the logic includes rules of definition for extending the logic with new constants and types.

The description of higher order logic given in this chapter is not complete, though it does cover all the aspects of the logic important to an understanding of later chapters. This book is concerned more with specifications than with proofs, and verification of theorems will be left mainly to the reader's logical and mathematical intuition. This chapter therefore deals mostly with notation. See Gordon's paper [41] or the HOL system manual [47] for a full account of higher order logic, including a list of the primitive rules of inference and a set-theoretic semantics.

In this book, the phrase 'higher order logic' should generally be understood to mean the particular formulation described in this chapter. Related versions of higher order logic are discussed by Andrews [2] and Hatcher [64].

2.1 Types

Higher order logic is a typed logic; every term of the logic has an associated logical *type*, which names the kind of value it denotes. A term is syntactically well-formed only if its type is consistent with the types of its subterms. As a syntactic device, types are necessary to eliminate certain 'paradoxical' statements which, if they could be expressed in the logic, would make it inconsistent (e.g. formulations of Russell's paradox). Requiring every term to have a type that is consistent with those of its

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-11532-2 - Higher Order Logic and Hardware Verification
T. Melham
Excerpt
More information

subterms makes such paradoxical expressions syntactically ill-formed and thereby eliminates them from the logic.

But as well as preventing inconsistency, logical types also play an important role in writing natural and concise propositions about hardware behaviour. Types can be used in hardware verification, as in mathematics and programming, to organize data and to make distinctions between variables that range over different kinds of values (e.g. bits, numbers, pairs, lists, recursive structures). This pragmatic use of logical types is discussed in chapter 5.

2.1.1 The syntax of types

The syntax of types in higher order logic is given by

$$\sigma \quad ::= \quad c \mid v \mid (\sigma_1, \ldots, \sigma_n)op$$

where σ, σ_1, ..., σ_n range over types, c ranges over type *constants*, v ranges over type *variables*, and op ranges over n-ary type *operators* (for $n \geq 1$). Type constants and type variables are called *atomic* types. Types constructed using type operators are called *compound* types.

Type constants are identifiers that name fixed sets of values. Examples are the primitive type constant *bool*, which denotes the two-element set of boolean truth-values, and the type constant *num*, which denotes the set of natural numbers.

Type variables are used to stand for 'any type'. They are written α, β, γ, etc. Type expressions that contain type variables are called *polymorphic*. A *substitution instance* of a polymorphic type σ is a type obtained by substituting types for all occurrences of one or more of the type variables in σ. Type variables occur in Church's formulation of higher order logic as metavariables ranging over types; in the version used here they are part of the object language. This allows a limited form of implicit universal quantification over types within the logic, since theorems that contain polymorphic types are also true for any substitution instance of them.

A compound type of the form $(\sigma_1, \ldots, \sigma_n)op$ denotes a set constructed from the sets denoted by the types σ_1 through σ_n. The n-ary type operator op is the name of the operation that constructs this set. The compound type $(bool, bool)fun$, for example, denotes the set of all total functions from values of type *bool* to values of type *bool*. This compound type is constructed using the binary type operator *fun*, which stands for the function space operation on sets.

2.1.2 Primitive and defined types

There are two primitive type constants in higher order logic, namely *bool* and *ind*. The type constant *bool* denotes the two-element set of boolean truth-values. The type constant *ind* denotes the set of 'individuals', which in this formulation of higher order logic is simply a set with infinitely many distinct elements. There is only one

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-11532-2 - Higher Order Logic and Hardware Verification
T. Melham
Excerpt
More information

primitive type operator, the binary type operator *fun*. If σ_1 and σ_2 are any two types, then the compound type $(\sigma_1, \sigma_2)fun$ denotes the set of all total functions from values of type σ_1 to values of type σ_2.

In principle, every type needed for doing proofs can be written using only type variables, the primitive type constants *bool* and *ind*, and the primitive type operator *fun*. But in practice it is desirable to add more type constants and operators to the logic than are strictly necessary to prevent inconsistency. In the version of higher order logic used here, this is done by defining new types and type operators in terms of primitive types or other already-defined types. A type definition extends the language of types by introducing a new type constant or type operator not already present in it. Formally, a type definition is an axiom that is added to the logic to give meaning to a new type expression. The primitive basis of the logic includes an explicitly-stated *rule of definition* for introducing axioms of this kind. This rule is explained in detail in section 2.7.

Two types that can be defined formally using this rule are *num*, the type of natural numbers, and $(\sigma_1, \sigma_2)prod$, the cartesian product of σ_1 and σ_2. A summary of some notation associated with these two basic types is given in section 2.2.6; a full account of how they are defined can be found in [88]. Other types defined specially for hardware verification are also introduced in later chapters.

2.1.3 Notational abbreviations for types

Some notational abbreviations are used to make type expressions more readable. Compound types constructed with the type operators *fun* and *prod* can be written using the infix notation shown below.

Infix Abbreviations for Types	
Type	*Abbreviation*
$(\sigma_1, \sigma_2)fun$	$\sigma_1 \rightarrow \sigma_2$
$(\sigma_1, \sigma_2)prod$	$\sigma_1 \times \sigma_2$

Expressions written using this infix notation are metalinguistic abbreviations for the corresponding object-language types; unlike defined types, they are not part of an extended syntax of object-language type expressions. For example, the type expression $bool{\rightarrow}(bool{\rightarrow}bool)$ is simply shorthand for the rather less readable type expression $(bool, (bool, bool)fun)fun$.

By convention, it is assumed that the infix symbols $\rightarrow$ and $\times$ associate to the right. The expression $bool{\rightarrow}(bool{\rightarrow}bool)$, for example, can be written without parentheses as $bool{\rightarrow}bool{\rightarrow}bool$. In addition, $\times$ is assumed to be more tightly binding than $\rightarrow$. So, for example, the expression $bool \times bool \rightarrow bool$ means $(bool \times bool) \rightarrow bool$.

© Cambridge University Press www.cambridge.org