PROVING THE CORRECTNESS OF DIGITAL HARDWARE DESIGNS USING PROLOG

INFORMATICA 1/88

Z. Brezočnik
B. Horvat
University of Maribor

UDK 681.3:325.6.08

ABSTRACT. An approach is presented for automatic formal verification of digital hardware designs using Prolog. Prolog is used both as a representational language for specifying the structure and the behaviour of a design and also as an inference mechanism for proving its functional correctness. A design in this model is composed of hierarchicaly organized modules. Each module is represented as a finite state machine. Validation of design correctness is made by formal proof as an alternative to the traditional approach which utilises simulation. The verification proceeds as follows: a) writing a design specification and a description of its realization in Prolog, b) deriving a design behaviour from the interconnections of its components and their behaviours, c) showing equivalence between the specified and the derived behaviour. The system has enough domain specific and general mathematical knowledge to perform the proofs largely automatically. Designs can be handled from the lowest transistor level up to the architectual levels. Some large designs including a simple computer have already been verified.

1. INTRODUCTION

A hardware or software designer must be able to decide whether the design meets its functional specification. Currently three different approaches exist for answering this question.

The first approach to enshure functional correctness is to develop the design from the specification by such a methodology that ensures it can't be incorrect. In software design it is exemplified in research of automatic programming. In hardware design automated techniques exist for dealing with elements of designs that are most tedious and prone to human error (such as wire routing or PAL generation). Silicon compilation techniques for automatic generation of complete designs from specifications are under development. This approach is perhaps the most attractive, but it is also the most difficult to achieve, because it faces an eventual astronomically large search space of design alternatives. A problem of automatic synthesis is so difficult, that useful general purpose systems for automatic synthesis are not expected in the near future.

The second and the most common approach to validate a design is an accurate simulation of it and trying it on test cases. In exhaustive simulation every possible combination of inputs should be tried for every possible internal state. The number of all possible input patterns can be extremely large even for simple devices. A multiplier for two 16-bit integers faces over four billion different inputs. It's clear that exhaustive simulation is no longer feasible. We can only select a subset of input patterns and hope to extrapolate from them to determine the correctness or failure of the design. The selection of an adequate subset of test cases is very difficult. Such partial simulation can detect the existence of an error, but can never guarantee that there are no more errors in the design.

There is an alternative third approach - a formal verification of a design with mathematical proof. If the formal verification succeds, the design will match its specification for all input patterns. Research in formal verification involves artificial intelligence techniques. It requires a formalism for representing the structure and the behaviour of digital systems and also an inference mechanism, which will expertly manage astronomically large search spaces. A success of the verification depends much on a built-in general mathematical knowledge base. Because of uncompleteness of the simulation and a bad prognosis for the soon automatic synthesis this approach seems to be the most promising in the nearer term.

Wagner was a pioneer with his research in this field. He has used a nonprocedural functional language for digital design description and a theorem prover in a first order predicate logic. The proof must be guided manually. Because of the description language the use is limited to low levels of a design. After that, Gordon has developed his methods for hardware modeling and verification based on different hierarchical levels with an interactive theorem prover (1). Initially, proofs were made manually, but more recently with an interactive theorem prover.

In this paper we present our verification system, named VERDIS (2) for formal verification of digital hardware designs using Prolog. VERDIS represents an automatized version of a

variant of Gordon's approach. In Fairchild Laboratory for Artificial Intelligence the VERIFY system was developed based on the similar principles (3). VERDIS has successfully verified some experimental designs with an interesting degree of complexity.

2. DESIGN REPRESENTATION IN VERDIS

A digital system in VERDIS is represented as a collection of hierarchically organized modules and their interconnections. A module is considered as a finite state machine (FSM)

$$A = \{X, Y, Z, \delta, \lambda\} \qquad (1)$$

It has a finite set of input (X) and output (Z) ports and a finite set of internal state variables (Y). VERDIS supports FSM of types Mealy and Moore and also ordinary decision circuits, if Y={}.

A module is either a primitive with no internal structure (basic building block of the design) or composition of the form

$$||(N_1, .., N_k),$$

where the components N_i are themselves modules. The input (output) ports of a compound module are the input (output) ports of its components. Compound modules are formed by linking some output ports of some components to some input ports of other components. External ports of the module are those ports that are not linked. Each port has an associated signal type which specifies the domain for signals passing through it. Signals in digital system are viewed differently according to the conceptual level in which they are considered: as voltage levels, as logic values ,as bits , as numbers or even as higher-level objects. Signals in VERDIS may have following types:

-*booole* (Boolean truth values true or false)
-*bit* (binary digits 0 and 1),
-*integer(N)* (integers in the range 0-2^{N+1}-1),
-*integer* (natural numbers),
-*booleZ, bitZ, integerZ(N)* and *integerZ*
(high impedance signal types).

Structural and signal hierarchy allow more succint description. Signals on a higher hierarchical level don't show unnecessary details of lower-level signals, but cary the same information.

For conciseness the behaviour of the FSM is described by two sets of equations , rather than an exhaustive table:

$$D^1y = \delta(x,y) \ ; \ x \in X, \ y \in Y \qquad (2)$$
$$z = \lambda(x,y) \ ; \ x \in X, \ y \in Y, \ z \in Z \qquad (3)$$

Equation (2) gives internal states as functions of inputs and current state. Equation (3) gives output signals as functions of inputs and current state. A dictionary of available functions that can be used in expressions for describing module behaviour currently consists of arithmetic (+,-,*,^) , logic *(not,and,or,xor)* and branching functions *(if,case)* and also some special functions for such operations as wiring signals on a bus and work with memories *(joinfn, bushfn, rfetch, fetch,store,...)*.

Module definition consists of Prolog facts and rules. In addition to constructs for specifying type of the module, ports, states, components, internal connections and behavioural equations the description language supports several useful constructs: constants, parameters, arrays, bit-wise connections and *equ* operator for calling some predefined modules from the system library. Let's illustrate some constructs mentioned so far by considering an example of a one-bit multiplier. It's constructed from a collection of 2-to-1 multiplexors, each of which has inputs *inx, iny*, control input *ctrl* and output *out*.

```
% Definition of a one-bit multiplier in
% terms of 2-to-1 multiplexsors

module(bitmult(N)).

port(bitmult(N), in(Bitmult),input,integer(N)).
port(bitmult(N), ctrl(Bitmult),input, boole).
port(bitmult(N),out(Bitmult),output,
     integer(N)).

constant(Bitmult(N),
         null(Bitmult),0,integer(N)).

part(bitmult(N), mplx(Bitmult, I), mux2):-
     index(0,I,N).

linked(bitmult(N),bit(I,in(Bitmult)),
       inx(mplx(Bitmult,I))):- index(0,I,N).
linked(bitmult(N), ctrl(Bimult, I),
       ctrl(mplx(Bitmult, I))) :-index(0,I,N).
linked(bitmult(N),    out(mplx(Bitmult,    I)),
       bit(I,out(Bitmult))):- index(0,I,N).

output_equation(bitmult(N), out(Bitmult) :=
if(ctrl(Bitmult), in(Bitmult), null(Bitmult))).
```

In this example, *N* is a parameter specifying the bit wide of the multiplicand (i.e., the most significant bit represents 2^N). Due to declarative power of Prolog a suitable indexing of part and connections can be used. The construct *index(0,I,N)* means simply that *I* can take any value from *0* to *N* .

3. THE VERIFICATION PROCESS

The key principle of VERDIS is that given the behaviour of components of a system and their interconnections, it is possible to derive a description of the behaviour of the whole system. The derived behaviour can then be compared with a specification of the intended behaviour of the system. If a design contains an error, a discrepancy between both behaviours can be detected and the design corrected.

Before of the verification some basic checks are made to enshure that nontristate outputs are not wired, that connected signals have equivalent types, that every output and state variable has an equation ... . These checks have proved to be very useful in finding typing and logical mistakes in design specification.

On request for verification of a module VERDIS checks, if the correctness of this module type has been already proved, in which case another proof is not necessary and the verification succeeds immediately. If a module is a primitive, its correctness can be assumed. If the correctness of a module is unknown, VERDIS uses a depth first search to recursively verify each of its part and then a module as a whole.

The next step in verifying a module is to derive a behavioural description of a composite module from the behaviours of its components with their interconnections. Each component module has a set of output and a set of state equations. The union of all these equations is in fact an implicitly specified behaviour of a module. It contains internal variables (signals at parts inside the module and state variables of component modules). The unnecessary internal information should be hidden. With subsequent substitutions of internal variables with

behaviour equations of component modules all internal variables are eliminated. If any frequently used internal variable has a very complicated equation it may be better not to eliminate it to avoid even larger equations. In such cases VERDIS first derives and evaluates the expression for the immediate variable and refers to it in behaviour equations of the module where needed. The final result of this step is a set of derived output equations and a set of state equations.

At this point we can try to prove that the derived behaviour description of the module is equivalent to the behavioural specification. In most cases a mapping between them is an exact equivalence - isomorphism. We have to show that corresponding equations in both automata are identical. The proof of identity is generally a hard problem. It requires much mathematical knowledge about functions, which can be used in equations. In more complex cases the correspondence between automata is a homomorphism rather than exact equivalence. A homomorphism may have a structural or a behavioural form or both. Structural homomorphism occurs, when automata are functionally identical but different in structural description. Behavioural or temporal homomorphism occurs, when the same automata is viewed with different time-scales. VERDIS currently works only for isomorphic machines.

The derived and the specified behaviour are compared for each output and each state. Proof of design correctness requires the ability to prove that a given equation (specified behaviour as a left side and derived behaviour as a right side)is an identity. The equation is first checked to see whether it is recognized as a trivial identity or whether it is a trivial non-identity. If the equation is not trivial, VERDIS tries to choose the best strategy for proving the indentity. A strategy selection depends on a form of left and right side of the equation and on function, operators and types of the variables involved. The repertoire of strategies contains: algebraic simplification, Boolean canonicalization and enumeration.

Algebraic simplification is the most general strategy, which tries to prove the identity of left and right side of the equation with simplification, expansion and canonicalization. Simplification is implemented with a general recursive Prolog procedure that recursively simplifies a given expression using simplification rules for the involved operators and functions. Expansion is used when a function is observed on only one side of the equation. The definition of the function is then substituted for its call and the resulting expression is simplified. Canonicalization tries to deal with combinatorial problems because of commutativity and associativity of some functions *(+,*,and,or...)*.

If only Boolean variables occur in the equation a more straightforward strategy of Boolean canonicalization is chosen. During six subsequent steps the left and the right side of equation are transformed to a lexically ordered complete disjunctive normal forms and then compared. It's faster than algebraic simplification.

Sometimes no other strategy but enumeration may be applied. An enumeration is made over a minimal necessary number of variables. For a selected set of variables each possible combination of their values is generated and substituted into the equation, which is then simplified to 'true' or 'false'. In some special cases it's not necessary to generate all possible combinations but only some of them. Partial enumeration may save much computation. VERDIS avoids the use of enumeration, because it's usually very space and time consuming.

If VERDIS doesn't have enough knowledge to select a strategy, an interactive mode is entered. Currently a user can insist in the algebraic simplification, in the enumeration on all or just some of the variables and can also simply assume the equation to be true or false. It's always possible to enlarge VERDIS's knowledge base and automatize the strategy selection for such cases.

Proving identity of an equation is the main and the most difficult problem in the verification process. The use of any procedural programming language with deterministic organization would not be a natural way for solving this problem. This fact is one of the most significant arguments for realization of VERDIS with the nonprocedural language Prolog. A Prolog program is a pattern directed system. The proof proceeds with an activation of that verification rule which is currently appliable to the left and to the right side of the equation. Such realization has a high degree of modularity. The knowledge base can simply be enlarged by adding new rules without any other changes.

4. TWO COMPLEX EXAMPLES

VERDIS has tackled some complex designs including modules *d74* and *host*. *d74* is an arithmetic unit(inputs *inA*, *inB*, *inC* and outputs *out1* and *out2)*, which is intendend to compute sums of products.

*out1 = inA*inB + inA*inC*
*out2 = inA*inC + inB*inC*

At the top level it contains three multipliers and two adders. The multipliers consist of slices, each of which contains a one-bit multiplier, a shifter and an adder. The adders are built from fulladders, which are built from invertors and 2-to-1 multiplexors. Multiplexors are built from logic gates. The logic gates are themselves described at two levels: at Boolean algebra level and at lower transistor level with tristate signals and stored charge. *d74* is parameterized in bit-wide of inputs. An instance that has 2-bit inputs involves 21 different types of module with nine levels of structural hierarchy. The design contains 1902 primitive parts, including 1016 transistors. The entire listing of the proof trace occupies about 1300 lines.

The next example is a register-transfer level description of a simple computer *host* with eight operations: HALT, JMP, JZRO, ADD, SUB, LD, ST and NOP. The computer is divided into a control section and a data section. The data section is built upon a single 16 bit data bus. Six register (the program counter, the accumulator, the instruction register, the argument register, the buffer register and the memory register) and also an arithmetic-logic unit and RAM are connected to the bus. The control section contains a microprogram ROM, a microprogram counter and a microinstruction decoder. VERDIS has proved the correctness of this computer at the microinstruction level.

5. DISCUSSION OF THE RESULTS

VERDIS is a large Prolog program that occupies about 61 kbytes of code for interpretation. It currently runs on a VAX 8800 under ISJ Prolog interpreter (4).

The main restriction on the complexity of examples which currently can be tackled is the amount of workspace required by Prolog. To deal with large systems, it is necessary to verify them a piece at a time or to restart verification when it aborts due to lack of space. Since VERDIS promptly records the progress of verification in the database, restarting does not result in duplication of work.

Due to the hierarchical decomposition of design description the work done in proving the correctness of a complete system is linear in terms of the number of types of module in its design, rather than linear in terms of the number of primitive components. For complex designs with many thousands of primitive components, the computational saving can be very great.

At present, the program performs only functional verification, with no consideration of timing. It appears relatively straightforward to introduce the notation of time in our description, but adding the appropriate inferential machinery will probably require more effort. However, dealing with truly asynchronous systems requires a rather different approach with revision of the representation and proof mechanism, perhaps in direction of a temporal logic (5).

The verification in VERDIS runs largely automatically, what is an advantage in comparison with the Gordon's approach, where the proof has to be guided by a user. VERDIS can be compared with a similar system VERIFY which has been developed in the Fairchild Laboratory for Artificial Intelligence. VERDIS introduces some useful new elements: possibility of macro calls for predefined modules in the system library; retaining of internal variables with over-complex expressions (i.e. bus output), new verification strategies of Boolean canonicalization and partly enumeration, more heuristics in variable selection for the enumeration etc.

6. CONCLUSION

VERDIS is our first attempt of formal verification which has already shown that digital designs with an interesting degree of complexity can be handled. It should be tried on many other real designs and it's knowledge base should be enlarged to become a real design tool, but abilities of this approach are yet evident.

The decision for Prolog as a description and an implementation language seems to be correct. Pattern matching and backtracking in Prolog are very powerful tools in the verification process. VERDIS is currently interpreted on a VAX 8800 under IJS Prolog interpreter(4). If we have any Prolog compiler, the speed of execution of the compiled code will increase ten times at least.

VERDIS should be only one component of larger system supporting hardware designs. This would integrate programs for design entry, simulation, verification, diagnostics etc.

REFERENCES

(1) M. Gordon , J. Herbert, Formal hardware verification methodology and its application to a network interface chip, IEE Proceedings, Vol. 133, Pt. E, No. 5, September 1986, 255-270

(2) Z. Brezocnik , Hardware verification, Master thesis, Tehniska fakulteta, Maribor, November 1986

(3) H. G. Barrow, VERIFY: A Program for Proving Correctness of Digital Hardware Designs, Artificial Intelligence, Vol. 24, No. 1-3, December 1984, 437-491

(4) J. Stojanovski, IJS Prolog Reference Manual, Institut Jozef Stefan, Ljubljana, 1986

(5) B.C.Moszkowski, A temporal logic for reasoning about hardware, Procedings Sixth International Symposium on Computer Hardware Description Languages, Carnegie-Mellon University, Pittsburgh, 1983, 79-90