Introduction to Formal Verification

Hardware Verification Group
Concordia University
Department of Electrical & Computer Engineering
Montreal, Quebec, Canada

May 2001

The rapid growth in the market gives rise to the demand in lesser and lesser cycles in the design of a VLSI circuit. With ever increasing complexity of the design of digital systems and the size of the circuits in VLSI technology, the role of design verification has gained a lot of importance. High costs (time, money, security and possibly lives) are incurred because a system is typically delivered with design errors, and hence must go through several design iterations. It takes a very large amount of time and effort to correct an error, especially when the error is discovered late in the process. For these reasons, we need approaches that enable us to discover errors and validate designs as early as possible. Verification is defined as the validation of the circuit for its correctness. The verification process is required at every level of the design flow as shown in (Figure 1) [16].

The designer first manually derives the requirements of the system as the system behavioral specification. This specification is then refined manually or using CAD tools into more detailed descriptions such as register-transfer (RT), logic and mask level descriptions. As the late detection of design errors is largely responsible for unexpected delays in realizing the hardware design, it is extremely important to ensure correctness in each design step. With correct-by-construction design style, automatic tools using behavioral and logic synthesis techniques can be used to ensure behavioral and gate level design correctness. However, the refinement process from high-level specification to synthetizable design usually requires manual fine tuning to achieve high performance. More progress is needed to automate the design process at higher levels in order to produce designs of the same quality as is achievable by hand. It is thus essential that the specification (or behavior) and the intermediate design stages be verified for consistency and correctness with respect to some user-specified properties or a previous level of the specification, thus making post-design verification

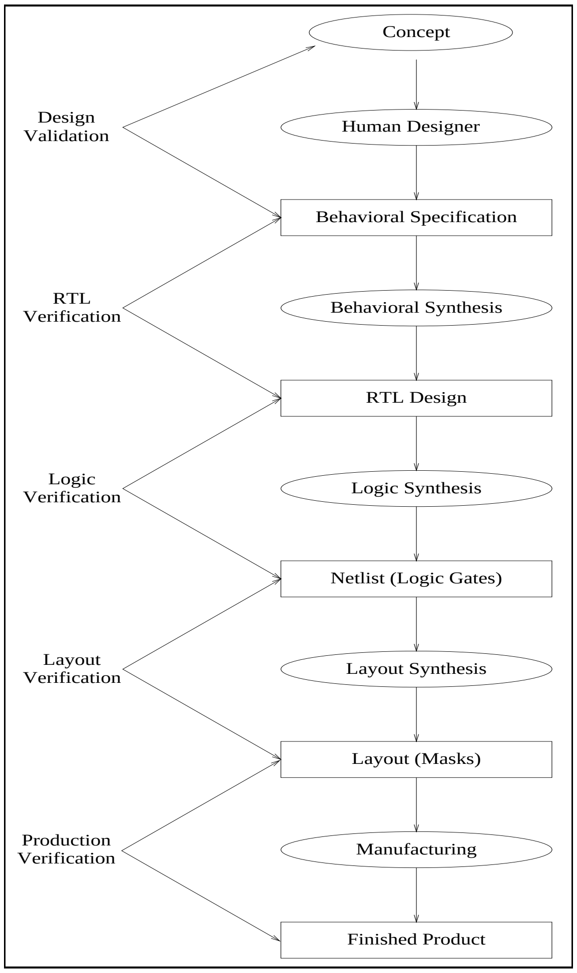

Figure 1: Hierarchical Design Flow [16]

essential.

Validation techniques are simulation, testing, prototyping and formal verification. Traditionally, testing and simulation are used to check the design's correctness (see Figure 2).

Despite the major testing and simulation efforts, serious design errors often remained undetected because simulation catches some problems, but not exhaustively. Due to the increasing concurrency and complexity of designs, detecting every bug resulting from the complex interaction of concurrent events by simulation becomes highly improbable (or prohibitively time consuming). Because testing and simulation are inadequate to certify that a system behaves correctly, the evolution of alternative verification approaches has emerged, such as formal methods [18, 24]. Formal methods have the potential for significantly reducing the number of design faults in VLSI and at the same time reducing the cost of that design. Formally verifying designs

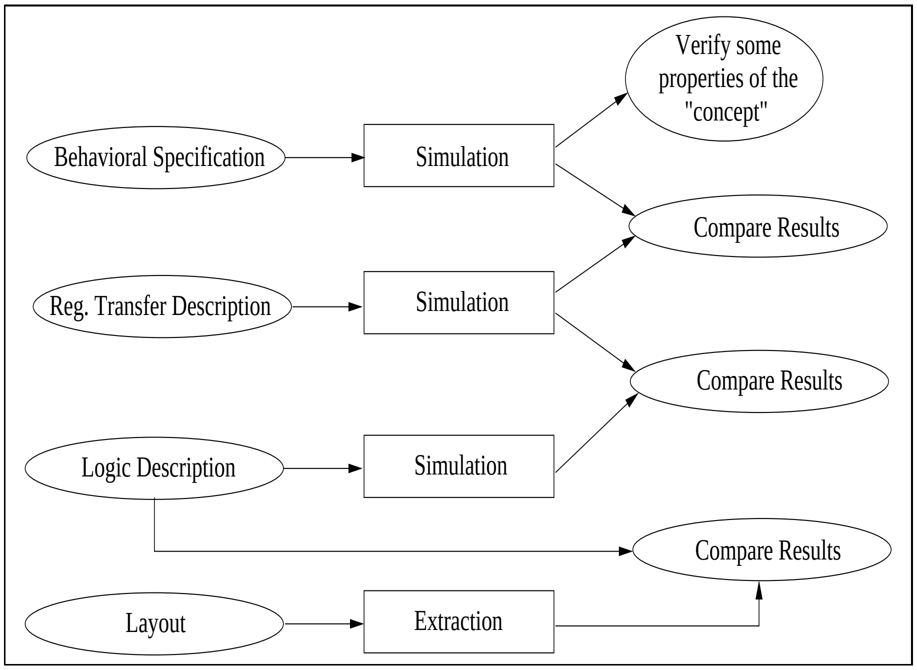

Figure 2: Verification by Simulation

may be cost effective in ”safety critical” applications, for systems in high volume or remotely placed, and for systems that will go through frequent redesign because of changes in technology. It has been recently recognized that formal verification is a powerful complementary approach to simulation and it has made exciting progress [18, 24].

This situation has prompted interest in verification techniques. Formal methods have long been developed and advocated within the computing science research community as providing sound mathematical foundation for the specification, implementation and verification of computer systems. These methods exploit representations with formally defined semantics in order to describe abstractly (independent of details of implementation) the desired functional behavior of a system [3]. Such formalization methods provide precise and unambiguous system specifications which can be checked for completeness and internal logical consistency. The mathematical nature of these specifications enable reasoning about consistency (i.e., whether the system dynamics is consistent with system's static properties) and the deduction of consequences of the specification. These can be checked against the user's expectations and used to generate tests for the system implementation. The application of formal methods requires a formal framework to describe the specification and implementation of a design . A specification refers to the description of the intended/required behavior of the hardware design. An implementation refers to the hardware design that is to be verified. Various formalisms, such as temporal logic, propositional logic, higher-order-logic, computational tree logic are used for the specification and implementation description framework.

Specifications in an executable formal language allow direct simulations (animations) of system behavior, giving early feedback to be compared with user requirements before full system development has begun. Equally important in the system's development process, a formal specification is a yardstick against which implementations are verified, or implementation steps through mathematical proof of the equivalence of abstract and concrete representations of system operations or data structures [21]. This way, a hierarchical verification can be achieved where a lower level specification is made equal to a higher level implementation (Figure 3) [18]. A formally based development methodology requires that a mathematical theory of

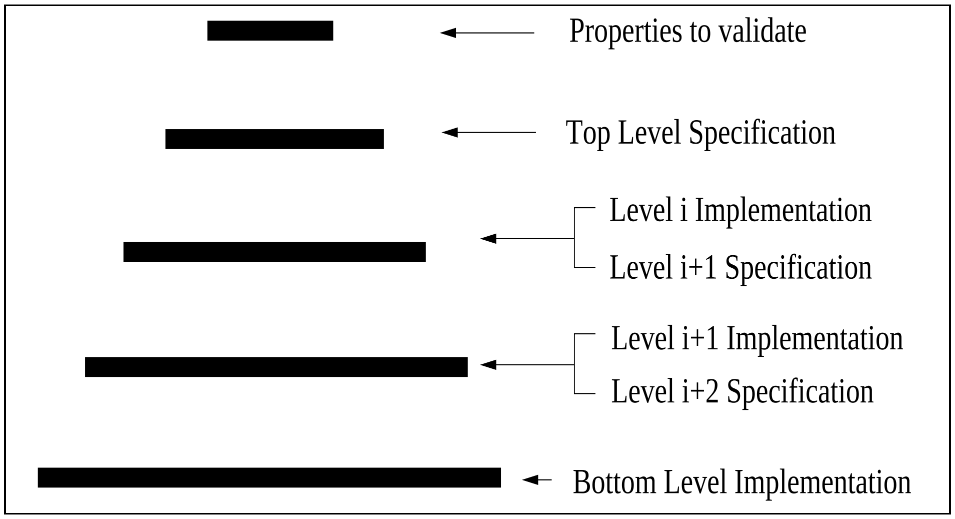

Figure 3: Hierarchical Verification

the desired system be created, documented and analyzed. This foundation activity entails a greater proportion of time and effort being invested in the initial pre-design phases of system development than is now commonly the case. Thanks to the rigorous discipline imposed by these methods, system development phases are rendered less error-prone, more systematic and amenable to computer assistance, hence higher quality products is achieved. Thus, formal verification is proposed as a method to help certify hardware and software, and consequently, to increase confidence in new designs.

1 Formal Verification Techniques

There are several approaches to formal hardware verification such as theorem-proving, model checking, equivalence checking and symbolic simulation [18, 24, 26, 35]. Each of them has its own strengths and weaknesses.

1.1 Theorem Proving

Theorem proving consists in expressing the specification and implementation in first-order or higher-order logic formulae. Their relationship, stated as equivalence or implication, is regarded as a theorem to be proven within the logic system, using axioms and inference rules. Powerful mathematical techniques such as induction and abstraction are the strengths of theorem proving and make it a very flexible verification technique. Theorem proving is a powerful verification technique that can provide a unifying framework for various verification tasks at different hierarchical levels. Theorem-proving methods have been around for over 35 years, and definitely have their staunch adherents. They have been extensively used in government pilot projects, notably in NASA. In spite of impressive demonstrations in the hardware domain and elsewhere, theorem-proving methods never achieved the broad level of acceptance in industry for which their advocates had hoped. The reason undoubtedly lies in the need for expert users, and an application cycle which evolves generally slower than a normal product design cycle, so even just keeping up with the project development schedule is a problem. Theorem provers or proof checkers are tools developed to partially automate the proof process or to check a manual proof. Some of the well-known theorem provers are HOL (Higher-Order Logic) [17], PVS (Prototype Verification System) [32], Boyer-Moore [4] and ACL2 [23].

1.2 Decision Diagram based Methods

Decision graph based methods use state space algorithms on finite-state models to check if the specification is satisfied. In case the verification fails, the user can track with a produced counter-example as to why it failed. The two main techniques in this category are property checking and equivalence checking. Equivalence checking verifies that an implementation has the same outputs as that of the specification, for all input sequences, wherein both the implementation and the specification are modeled as FSMs. Property checking verifies the validity of a specification, expressed as a set of properties, on an implementation, modeled as an FSM. In property checking, we have invariant checking where safety properties are verified, and model checking where safety as well as liveness properties can be checked. In equivalence checking, we differentiate combinational equivalence and sequential equivalence checking.

1.2.1 Model Checking

Model checking [11] is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols. Specifications are expressed in a propositional temporal logic, and the reactive system is modeled as a state-transition graph. An efficient search procedure is used to determine automatically if the specification are satisfied by the state-transition graph. The technique was originally developed by Clarke et al [9]. Later at AT&T Bell Laboratory, there was an alternative approach based on showing inclusion between automata [27]. Model checking has several important advantage over mechanical theorem proving, the most important being the high automacity of the procedure. Typically the user provides a high level representation of the model and the specification to be checked. The model checker will either terminate with the answer true indicating that the model satisfies the specification or give a counterexample execution that shows why the formula is not satisfied. The counter-examples are particularly important in finding subtle errors in complex reactive systems. The first model checkers were able to find subtle errors in small circuits and protocols. However they were unable to handle very large examples due to the state explosion problem. The problem arises in systems composed of multiple state holding elements operating in parallel: the total number of states in the system generally grows exponentially with the number of state holding elements. Verifying complex systems dramatically changed with the discovery of reduced ordered binary decision diagrams (ROBDDs) [6] to represent the transition relation. Used with the original model checking algorithm, it is called symbolic model checking [29]. By using this combination, it is possible to verify large systems. Because of this breakthrough it is now possible to verify reactive systems with realistic complexity and a number of major companies including Intel, Motorola, Fujitsu and IBM are using symbolic model checkers to verify real circuits and protocols. In several cases, errors have been found that were missed by extensive simulation. A number of tools have been developed. Some well-known model checkers are SMV (Symbolic Model Verifier) [30], VIS (Verification Interacting with Synthesis) [5], SPIN [19], Murphi [13]. Other commercial model checking tools also emerged like FormalCheck [8] or RuleBase [2].

While symbolic representations have greatly increased the size of the system that can be verified, many realistic systems are still too large to be handled. Thus, it is important to find techniques that can used in conjunction with the symbolic methods to extend the size of the systems that can be verified. Current well-known techniques are compositional reasoning and abstraction [31].

Recently a number of ROBDD extensions such as Binary Moment Diagrams (BMDs) [7], Hybrid Decision Diagrams (HDDs) [10] or K*BMDs [14] have been also developed to represent arithmetic functions more compactly than ROBDDs. An improvement is the EOBDDs [28] that can have leaf nodes labeled by terms containing abstract sorts. Multiway Decision Graphs (MDGs) [12], the successor of EOBDDs, allow the labeling of edges to be first order terms and non-terminal nodes to be abstract variables.

1.2.2 Equivalence Checking

Equivalence checking is used to prove functional equivalence of two design representations modeled at the same or different levels of abstraction. It can be divided into two categories: one is *combinational equivalence checking*, and the other is *sequential equivalence checking*. Equivalence checking verifies for all input sequences that an implementation has the same outputs as the specification, both modeled as finite state machines (FSM). Some commercial tools have been used in industries, for example Chrysalis's Design Verifier, Cadence Affirma, Synopsys Formality, IBM BoolEye, etc. They are often used to verify the equivalence between RTL and synthesized gate-level design. Also they are used to ensure the correctness of manual optimization during the design process. Combinational equivalence checking is based on the canonical representations of boolean functions, typically binary decision diagrams (BDDs) or their derivatives. The functions of the two descriptions are converted into canonical forms which are then structurally compared. The major advantage of BDDs is their efficiency for a wide variety of practically relevant combinational circuits. If the BDD size does not grow too large, this type of Boolean reasoning is fast and independent of the actual circuit structure. Moreover, if structural similarities of the two designs are exploited, BDDs can effectively find implications between nets even if they are away from the primary inputs. However, since the current design are mainly clock-driven synchronized designs, to perform the combinational equivalence checking between two different sequential models, we have to cut the designs into pieces, map each register (or flip-flop) of one model into another, and compare their combinational circuits between every two consecutive registers. The tool maps each register of one model into another and compares their combinational circuits between every two consecutive registers. However, combinational equivalence checking cannot handle the equivalence checking between RTL and behavioral models because these models are developed separately and it is not possible to map each register in the RTL model to that of a behavioral model. Sequential equivalence checking is used to verify the equivalence between two sequential designs at each state. Sequential equivalence checking considers the behavior of two designs while ignoring their implementation details such as register mapping. This is done by building the product machine of the FSMs and checking a simple invariant stating the equivalence of the outputs of both FSMs for every reachable state of the product machine. It can verify the equivalence between RTL and netlist or RTL and behavioral model which is very important in design verification. The disadvantage of sequential equivalence checking is that it cannot handle a large design because it encounters the state space explosion problem very fast. Examples of sequential equivalence checkers are VIS [5] and MDG [12].

2 Formal Verification Tools

There are many formal verification tools available at academic and industry level. Some of the main tools are HOL, PVS, SMV, VIS, FormalCheck, MDG along side others. Significant research work is going on at present to make the tools better to suit the needs of academia and industry.

2.1 HOL

The HOL System is an environment for interactive theorem proving in a higher-order logic, developed at University of Cambridge, U.K [17]. Its most outstanding feature is its high degree of programmability through the meta-language ML [33]. The system has a wide variety of uses from formalizing pure mathematics to verification of industrial hardware. Academic and industrial sites world-wide are using HOL. HOL has a formally defined syntax and semantics. It supports both 'forward' and 'goal-directed' proofs and it is secure, i.e., it can not prove false theorems.

2.2 PVS

The logic of PVS (Prototype Verification System) [32] is a theorem prover, developed at SRI, California. The logic used in PVS is a strongly typed higher-order with a rich type system, which includes dependent types and predicate subtypes. Type checking in this logic is undecidable and requires the assistance of the theorem prover and possibly human intervention to discharge automatically generated type correctness conditions. Definitions and theorems can be grouped into parameterized theories whose parameters may have constraints attached. Proofs in PVS are backward proofs using a sequent representation for proof obligations. The inference rules operate at a higher level than the primitive inferences of higher-order logic and include instantiation, application of theorems, equality rewriting, as well as complex propositional rewriting. Higher-level proof strategies analogous to HOL tactics can be constructed from the basic inference rules using a specialized strategy language. To discharge certain classes of obligations, PVS employs an integrated decision procedure for equality reasoning, linear arithmetic, arrays, etc., as well as a BDD-based procedure for propositional logic. The decision procedures are integrated with the type checker to make use of additional constraints available from information. PVS also includes a model checker for finite-state μ-calculus that can be accessed through a formalization of the μ-calculus within the PVS logic.

2.3 SMV

Symbolic Model Verifier (SMV) [29] is a model checker developed at Carnegie Mellon University. It allows to check that a finite-state system satisfies specifications given in CTL (Computational Tree Logic) properties [15]. It uses the OBDD-based symbolic model checking algorithm. A specification for SMV is a collection of properties. When the tool fails to check properties of a certain model, it will produce a counter-example. A counter-example is a behavioral trace that violates the specified property. This makes SMV a very effective debugging tool as well as a formal verification system. A newer version of SMV developed at Cadence [30] includes the concepts compositional verification [31] and abstraction which enhance the ease of verification. For large designs, especially those including substantial data path components, the user must break the correctness proof down into parts small enough for Cadence SMV to verify. This is known as compositional verification. Cadence SMV also provides a number of tools to help the user reduce the verification of large, complex systems to small finite state problems.

2.4 VIS

Verification Interacting with Synthesis (VIS) [5], developed at University of California, Berkeley, integrates the verification, simulation and synthesis of finite-state hardware systems. It uses a Verilog front-end and supports model checking, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis. VIS performs symbolic model checking and reports the failure with a counter-example which is called the “debug” trace. Also, VIS provides the capability to check the combinational equivalence and sequential equivalence of two designs. Sequential verification is done by building the product finite-state machine, and checking that the outputs are equal for every reachable state of the product machine. VIS also provides traditional design verification in the form of a cycle-based simulator that uses BDD techniques. Since VIS performs both formal verification and simulation using the same data structures, consistency between them is ensured.

2.5 FormalCheck

FormalCheck [8] is an industrial model checking tool based on ω-automata [36]. It was first released in April 1997 by Bell Labs Design Automation and now part of the Cadence Design Systems Affirma tool set. FormalCheck supports the synthetizable subsets of the industry standard hardware description languages, VHDL and Verilog. The user supplies FormalCheck with a set of queries to be verified on the design model which is written in synthetizable Verilog or VHDL code. The queries (properties and constraints) are simple temporal statements (formalizations) describing behavioral aspects of the specification. FormalCheck can handle larger designs by using a number of embedded reduction techniques.

2.6 MDG

Multiway Decision Graphs (MDGs) [12], developed at University of Montreal, represent and manipulate a subset of first-order logic formulae suitable for high-level hardware verification. With MDGs, a data value is represented by a single variable of an abstract type and a data operation is represented by an uninterpreted function symbol. The MDG operators and verification procedures are packaged as MDG tools and implemented in Prolog. The MDG tools provide facilities for invariant checking, verification of combinational circuits, equivalence checking of two state machines and model checking.

2.7 Hybrid Tools

No single formal method is suitable for describing and analyzing every aspect of a complex system. A practical solution is to combine different methods. When combining methods it is important to find a suitable style for using different methods together, and also investigate how these different methods could be used together. Today, there exist a number of hybrid tools which are mainly the combination of theorem proving and model checking. Examples are [1, 20, 22, 25, 32, 33, 34].

References

[1] M.D. Aagaard, R.B. Jones, and C-J.H. Seger. Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher* Order Logics, Lecture Notes in Computer Science 1690. pages 323-340, 1999.

[2] I. Beer, S. Ben-David, and A. Landver C. Eisner. RuleBase: an Industry-oriented Formal Verification Tool. *Proc. Design Automation Conference(DAC'96)*, pages 665–660, June 1996.

[3] H.E. Berg, W.E. Boebert, W.R. Franta, and T.G. Moher. *Formal Methods of Program Verification and Specification.* Prentice-Hall Inc., 1982.

[4] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[5] R.K. Brayton. VIS: A System for Verification and Synthesis. Technical Report UCB/ERL M95, University of Berkley, December 1995.

[6] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[7] R.E. Bryant and Y. Chen. Verification of Arithmetic Circuits with Binary Moment Diagrams. *Proc.* 32^{nd} *ACM/IEEE Design Automation Conference(DAC'95)*, June 1995.

[8] Cadence, USA. *Formal Verification Using Affirma FormalCheck, Version 2.3*, October 1999.

[9] E. Clarke, E.A.Emerson, and A.P Sistla. Automatic Verification of Finite State Concurrent System Using Temporal Logic Specification. *ACM Transactions on Programming languages and Systems*, 8(2):244–264, 1986.

[10] E. Clarke, M. Fujita, and X. Zhao. Hybrid Decision Diagrams. *Proc. IEEE International Conference on Computer-Aided Design (ICCD'95)*, June 1995.

[11] E. Clarke, O. Grumberg, and D.E. Long. *Model Checking.* In Nato ASI, 152 of F. Springer Verlag, 1996.

[12] F. Corella, Z. Zhou, X. Song, M.Langevin, and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. *Formal Methods in System Design*, 10(1):7–46, 1997.

[13] D. Dill. The Murϕ Verification System. *Proc. International Conference on Computer-Aided Verification(CAV'96)*, pages 390–393, August 1996. Lecture Notes in Computer Science, Vol. 1102, Springer-Verlag.

[14] R. Drechsler, B. Becker, and S.Ruppertz. K* BMDs:A New Data Structure for Verification. *Proc. IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods(CHARME'95)*, October 1995.

[15] E. A. Emerson. *Temporal and Modal Logic. Handbook of Theoretical Computer Science.* Elsevier Science Publisher B.V., 1990, Chapter 16.

[16] A. Ghosh, S.Devadas, and A.R.Newton. *Sequential Logic Testing and Verification.* Kluwer Academic Publishers, 1992.

[17] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, Cambridge, U.K., 1993.

[18] A. Gupta. Formal Hardware Verification Methods: A Survey. *Journal of Formal Methods in System Design*, 1(2/3):151–238, 1992. Kluwer Academic Publishers.

[19] G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, May 1997.

[20] J. Hurd. Integrating Gandalf and HOL. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 1690. pages 311-321, 1999.

[21] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 1986.

[22] J.J. Joyce and C.J.H. Seger. Linking BDD-based Symbolic Evaluation to Interative Theorem Proving. *In Proceedings of the* 30^{th} *Design Automation Conference*, pages 469–474, June 1993.

[23] M. Kaufmann and J.S. Moore. ACL2: An Industrial Strength of Nqthm. *Proc.* 11^{th} *Annual Conference on Computer Assurance(COMPASS'96)*, pages 23–34, June 1996. IEEE Computer Society Press.

[24] C. Kern and M. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of E. Systems*, 4:123–193, April 1999.

[25] S. Kort, S. Tahar, and P. Curzon. Hierarchical Verification Using an MDG-HOL Hybrid Tool. *Proc. IFIP Conference on Correct Hardware Design and Verification Methods (CHARME'2001)*, September 2001.

[26] T. Kropf. *Introduction to Formal Hardware Verification.* Springer Verlag, 1999.

[27] R.P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes.* Princeton University Press, 1994.

[28] M. Langevin and E. Cerny. An Extended OBDD Representation for Extended FSMs. *Proc. EDAC-ETC-EUROASIC*, 1994.

[29] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, Massachusets, 1993.

[30] K.L. McMillan. *Getting Started with SMV: User's Manual.* Cadence Berkley Labratories, USA, 1998.

[31] K.L. McMillan. Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking. *Proc.Computer Aided Verification(CAV'98)*, June/July 1998.

[32] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Sirvas. PVS: Combining Specification, Proof Checking, and Model Checking. *In Computer Aided Verification.* Lecture Notes in Computer Science 1102, Springer-Verlag, pages 411-414, July 1996.

[33] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, U.K., 2nd edition, 1996.

[34] K. Schneider and D.W. Hoffmann. A HOL Conversion for Translating Linear Time Temporal Logic to ω-Automata. *Theorem Proving in Higher Order Logics.* In Y. Bertot, G. Dowek, A. Hirschowitz,C. Paulin, and L. Thery, editors, *Lecture Notes in Computer Science 1690*, Nice, France. September 1999.

[35] C.J. Seger. An Introduction to Formal Hardware Verification. Technical Report 92-13, Dept. of Computer Science, University of British Columbia, Vancouver, B.C., Canada, June 1992.

[36] W. Thomas. Automata on Infinite Objects. *volume B of Handbook of Theoretical Computer Science*, 1990.