VIS : A System for Verification and Synthesis

Robert K. Brayton* Gary D. Hachtel† Alberto Sangiovanni-Vincentelli* Fabio Somenzi†
Adnan Aziz* Szu-Tsung Cheng* Stephen Edwards* Sunil Khatri* Yuji Kukimoto*
Abelardo Pardo† Shaz Qadeer* Rajeev K. Ranjan* Shaker Sarwary‡ Thomas R. Shiple*
Gitanjali Swamy* Tiziano Villa*

1 Introduction

VIS (Verification Interacting with Synthesis) is a tool that integrates the verification, simulation, and synthesis of finite-state hardware systems. It uses a Verilog front end and supports fair CTL model checking, language emptiness checking, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis.

We designed VIS to maximize performance by using state-of-the-art algorithms, and to provide a solid platform for future research in formal verification. VIS improves upon existing verification tools by:

1. providing a better programming environment,
2. providing new capabilities, and
3. improving performance in some cases.

We have incorporated software engineering methods into the design of VIS. In particular, we provide extensive documentation that is automatically extracted from the source files for browsing on the World Wide Web.

Section 2 describes the major capabilities of VIS as seen by the user, and Section 3 gives a brief description of the underlying algorithms of these capabilities. Section 4 discusses the VIS programming environment, and Section 5 gives conclusions and ideas for future work.

2 Capabilities of VIS

We briefly describe the salient features of VIS. VIS has both an interactive command interface and a batch mode. For a detailed description of the full functionality of VIS, with examples of usage, refer to the VIS Manual [2].

Verilog front end VIS operates on an intermediate format called BLIF-MV, which is an extension of BLIF, the intermediate format for logic synthesis accepted by SIS [7]. VIS includes a stand-alone compiler from Verilog to BLIF-MV, called VL2MV, which supports a synthesizable subset of Verilog. VL2MV extracts a set of interacting finite state machines that preserves the behavior of the source Verilog program defined in terms of simulated results. Two new features have been added to Verilog:

1. Nondeterminism. A nondeterministic construct, `$ND`, has been added to specify nondeterminism on wire variables; this is the only legal way to introduce nondeterminism in VIS.
2. Symbolic variables. Sometimes it is desirable to specify and examine the value of variables symbolically, rather than having to explicitly encode them. VL2MV extends Verilog to allow symbolic variables using an enumerated type mechanism similar to the one available in the C programming language.

Conceptually, it would be easy to provide a translator from another HDL language, like VHDL or Esterel, to BLIF-MV.

*Department of EECS, University of California, Berkeley, CA 94720
†Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309
‡Lattice Semiconductor, Milpitas, CA 95035

Hierarchy and initialization When a BLIF-MV description is read into VIS, it is stored hierarchically as a tree of modules, which in turn consist of sub-modules. This hierarchy can be traversed in a manner similar to traversing directories in UNIX. Simulation and verification operations can be performed at any subtree of the hierarchy. It is possible to replace the subhierarchy rooted at the current node with a new hierarchy specified by a new BLIF-MV file, which might be a synthesized module or a manually abstracted module. VIS can also output the hierarchy below the current node to a BLIF-MV file.

Interaction with synthesis VIS can interact with SIS to optimize the existing logic by reading and writing the BLIF format, which SIS recognizes. Synthesis can be performed on any node of the hierarchy.

Abstraction Manual abstraction can be performed by giving a file containing the names of variables to abstract. For each variable appearing in the file, a new primary input node is created to drive all the nodes that were previously driven by the variable. Abstracting a net effectively allows it to take any value in its range, at every clock cycle.

Fair CTL model checking and language emptiness check VIS performs fair CTL model checking under Büchi fairness constraints. In addition, VIS can perform language emptiness checking by model checking the formula $EG\ \ true$. The language of a design is given by sequences over the set of reachable states that do not violate the fairness constraint. The language emptiness check can be used to perform language containment by expressing the set of bad behaviors as another component of the system. If model checking or language emptiness fail, VIS reports the failure with a counterexample, (i.e., behavior seen in the system that does not satisfy the property - for model checking, or valid behavior seen in the system - for language emptiness). This is called the "debug" trace. Debug traces list a set of states that are on a path to a fair cycle and fail the CTL formula.

Equivalence checking VIS provides the capability to check the combinational equivalence of two designs. An important usage of combinational equivalence is to provide a sanity check when re-synthesizing portions of a network. VIS also provides the capability to test the sequential equivalence of two designs. Sequential verification is done by building the product finite state machine, and checking whether a state where the values of two corresponding outputs differ, can be reached from the set of initial states of the product machine. If this happens, a debug trace is provided. Both combinational and sequential verification are implemented using BDD-based routines.

Simulation 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. VIS can generate random input patterns or accept user-specified input patterns. Any subtree of the specified hierarchy may be simulated.

3 Algorithms

This section briefly discusses the significant algorithms of VIS. The fundamental data structure for these algorithms is a multi-level network of latches and combinational gates that is created by flattening the hierarchy. It is assumed that there are no combinational cycles in the network. The primary inputs and latch outputs are referred to as *combinational inputs* and the primary outputs and latch inputs are referred to as *combinational outputs*. The variables of a network are multi-valued, and logic functions over these variables are represented by multi-valued decision diagrams (MDDs) which are an extension of BDDs.

MDD variable ordering The combinational input variables and next state variables must be ordered before MDDs can be constructed. The combinational input variables are ordered by doing a depth-first traversal of the logic that generates the combinational outputs. The order in which the output logic cones are visited is determined using the algorithm of Aziz *et al.* [1]. This algorithm orders the latches to decrease a communication complexity bound (where backward edges are more expensive than forward edges) on the latch communication graph. The traversal of an output logic cone is done in such a way that the combinational inputs farthest from the outputs appear earlier in the ordering. We use the merging technique of Fujii *et al.* to handle those variables that appear in multiple cones of logic [5]. Finally, each next state variable is inserted into the variable ordering immediately after the corresponding present state variable.

If the user has some knowledge of a good ordering, then a partial or total ordering on the variables can be read in. In addition, dynamic variable ordering is supported. We have found that forcing corresponding present state and next state variables to

remain adjacent to each other is usually beneficial. Generally, a good initial ordering followed by one or two forced dynamic reorderings gives good results.

Partitioning the network Once the description of a system has been read in and the ordering of the variables assigned, an abstracted view of the system is created in which the functions of the network are stored as MDDs. This abstracted view, called a "partition", is the input to model checking and reachability. It can be created in several ways. At one extreme, combinational output functions are defined directly in terms of combinational inputs. On the other extreme, there is an MDD corresponding to each node in the network representing the functionality of the node in terms of its fanins, i.e., a variable is introduced for each node in the network. In general, intermediate variables can be introduced to represent the functionality of a cluster of nodes in the original network. This flexibility allows very large designs to be represented and manipulated.

Image/Pre-image computation Our image/pre-image computation technique is based on an early quantification heuristic [6]. The initialization process consists of creating a bit-level relation for the next state function of each latch in the network. These bit-level relations are then ordered to optimally exploit early quantification. Next, the relations of several bits are grouped together, making a cluster whenever the MDD size of the group reaches a threshold. Next, each cluster is simplified by quantifying out the primary inputs local to that cluster. Finally, the orders of the clusters for image and pre-image are calculated and stored. Also stored is the schedule of variables for early quantification.

Reachability analysis Reachability analysis makes iterative use of image computation. The performance of reachability analysis is improved by exploiting three sets of don't cares (in the following $R_k(\vec{x})$ represents the set of states reached from the initial states in k or fewer steps):

1. Selection of the frontier set for computing $R_{k+1}(\vec{x})$, given $R_k(\vec{x})$. The frontier set $F(\vec{x})$ can be any set satisfying the following inequality: $R_k(\vec{x})\overline{R_{k-1}(\vec{x})} \subseteq F(\vec{x}) \subseteq R_k(\vec{x})$.
2. Simplification of the transition relation $T(\vec{x}, \vec{u}, \vec{y})$, by taking the generalized cofactor with respect to $F(\vec{x})$ (we care only about the transitions originating from the frontier states).
3. Simplification of the transition relation $T(\vec{x}, \vec{u}, \vec{y})$, by taking the generalized cofactor with respect to $\overline{R_k(\vec{y})}$ (we care only about the transitions to the set of states not reached thus far).

Model checking and debugging We use the algorithms presented in [3] as the basis for fair CTL model checking and debugging. In addition, a special algorithm has been implemented to improve the efficiency of checking invariants. Also, a structural pruning technique is used to eliminate those parts of the network that cannot affect the formula being checked. This is particularly useful in conjunction with the abstraction mechanism mentioned in Section 2. Finally, don't cares arising from the unreachable states, and from the fixed point computations, are used to simplify intermediate MDDs.

4 Programming Environment

One of the key goals of VIS is to serve as a platform for developing new verification algorithms. We have used as our model the object-oriented programming style of SIS. VIS is composed of 18 packages; each exports a set of routines for manipulating a particular data structure, or for performing a set of related functions (e.g., there are packages for model checking, variable ordering, and manipulating the network data structure). New packages can be added easily. This wealth of exported functions can be used by future programmers to quickly assemble new algorithms. All functions adhere to a common naming convention so that it is easy to find functions in the documentation.

Particular attention was paid to the design of the interfaces to packages that are still the subject of ongoing research (e.g., MDD variable ordering, image computation, and partitioning). This makes it easy for other researchers to plug in their algorithms for performing a particular task, and then evaluate their algorithm within the context of VIS.

Extensive user and programmer documentation exists for VIS. The creation of this documentation was aided by the tool `ext` [4], which extracts documentation embedded in the source code. For each function, the programmer provides a synopsis and a complete description, and `ext` automatically extracts this information, along with the function name and argument types, into an HTML file that can be viewed on the World Wide Web. Documentation for user commands is extracted in a similar fashion.

5 Conclusions and Future Work

We have described the verification and synthesis tool VIS, which offers a better programming environment, new capabilities, and improved performance over existing verification tools. We have implemented VIS using the C programming language, and it has been ported to many different operating systems and architectures. The capabilities of VIS have been tested on the sequential circuits from the ISCAS benchmark set and some industrial designs.

As part of future work, we intend to explore and support explicit methods for state enumeration, verification of asynchronous systems, hierarchical synthesis, partitioning schemes, language containment, and incremental techniques for synthesis and verification. In particular, we want to explore the synergy between verification and synthesis.

For more information about VIS or to get a copy, visit the VIS home page [8].

Acknowledgments

We would like to thank Adrian Isles, Sriram Rajamani, and Serdar Tasiran for their assistance in developing VIS.

References

[1] A. Aziz, S. Tasiran, and R. K. Brayton. BDD Variable Ordering for Interacting Finite State Machines. In *Proc. of the Design Automation Conf.*, pages 283–288, San Diago, CA, June 1994.

[2] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. Technical Report UCB/ERL M95, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Dec. 1995.

[3] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proc. 32nd Design Automat. Conf.*, pages 427–432, June 1995.

[4] S. Edwards. *The Ext System*, 1995. `http://www.eecs.berkeley.edu/~sedwards/ext`.

[5] H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 38–41, Nov. 1993.

[6] R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton. Efficient Formal Design Verification: Data Structure + Algorithms. Technical Report UCB/ERL M94/100, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Oct. 1994.

[7] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.

[8] The VIS Group. *VIS: Verification Interacting with Synthesis*, 1995. `http://www-cad.eecs.berkeley.edu/Respep/Research/vis`.