VIS

Robert K. Brayton,[1] Gary D. Hachtel,[2] Alberto Sangiovanni-Vincentelli,[1]
Fabio Somenzi,[2] Adnan Aziz,[3] Szu-Tsung Cheng,[1] Stephen A. Edwards,[1]
Sunil P. Khatri,[1] Yuji Kukimoto,[1] Abelardo Pardo,[2] Shaz Qadeer,[1]
Rajeev K. Ranjan,[1] Shaker Sarwary,[4] Thomas R. Shiple,[1] Gitanjali Swamy,[1]
Tiziano Villa[1]

[1] Department of EECS, University of California, Berkeley, CA 94720
[2] Department of ECE, University of Colorado, Boulder, CO 80309
[3] Department of ECE, University of Texas at Austin, TX 78712
[4] Lattice Semiconductor, Milpitas, CA 95035

1 Introduction to VIS

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.

VIS was designed 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.

In addition, software engineering methods were used in the design of VIS in order to encourage further development by a diverse set of researchers. In particular, we provide extensive documentation that is automatically extracted from the source files for browsing on the World Wide Web.

In this tutorial, we will briefly describe each of the capabilities built into VIS and then concentrate on how to use and control VIS to make maximum use of its potential.

1.1 History

Many first generation tools for automatic formal verification were based on two theoretical approaches. The first is temporal logic model checking, where the properties to be checked are expressed as formulas in a temporal logic, and the system is expressed as a finite state system. In particular, Computational Tree Logic (CTL) model checking is a technique pioneered by Clarke and Emerson to verify whether a finite state system satisfies properties expressed as formulas in a branching-time temporal logic called CTL. SMV, a system developed at Carnegie Mellon University, belongs to this class of tools.

Certain properties are not expressible in CTL, but they can be expressed as ω-automata. The second approach, language containment, requires the description of the system and properties as ω-automata, and verifies correctness by checking that the language of the system is contained in the language of the property. Note that certain types of CTL properties involving existential quantification are not expressible by ω-automata. COSPAN, a system developed at Bell Labs, offers language containment.

A combination of both approaches is offered by the HSIS [2] system, which was developed at the University of California, Berkeley. Our experience with verification tools (in particular HSIS) led to the conclusion that sometimes, the simpler and more limited the approach, the more efficient it can be. A number of design decisions that we made for HSIS made it unacceptably slow for some large examples. With these problems in mind, we set about writing a tool that was more efficient, easily extendible, and offered a good programming environment, in order that it can be more easily upgraded in the future as more efficient algorithms are developed.

VIS also has the capability to interface with SIS to optimize logic modules; hence, VIS is an integrated system for hierarchical synthesis, as well as verification. We plan to pursue research on the interaction between verification and synthesis in the future; hence the name VIS, verification interacting with synthesis.

1.2 Overview of VIS

Fig. 1 presents of an overview of VIS. It has three main parts: a front-end to read

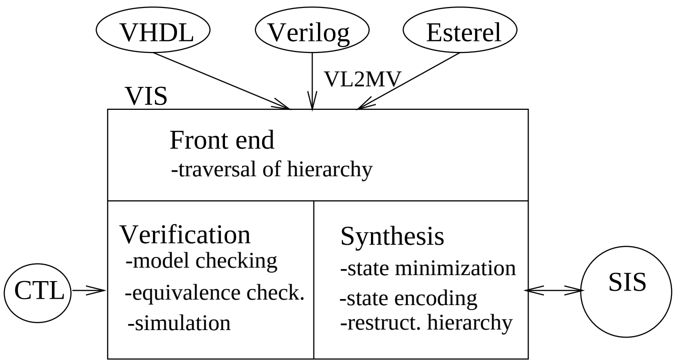

Fig. 1. Block diagram of VIS.

and traverse a hierarchical system described in BLIF-MV, which may have been compiled from a high-level language like Verilog; a verification core, to perform model checking of Fair CTL and test language emptiness; and a path to SIS, to optimize parts of the logic.

1.3 VIS Philosophy

We decided to offer limited but efficient capabilities. We felt that in the future, it would be easy to add more features, as they are required, using a well defined programming interface. In line with this **keep it simple** philosophy, VIS provides the following verification capabilities.

- Only CTL formulas can be checked. Language containment may be handled in a later release. However, we do handle language emptiness checks.
- Fairness constraints must be of Büchi type, i.e., sets of states that must be visited infinitely often. However, the internal VIS data structures do have the capability to support more complicated fairness constraints.

VIS can interact with SIS to assist the task of verification by simplifying parts of the system. Another objective is to support a full-fledged hierarchical synthesis flow, that translates a Verilog description into an optimized multi-level circuit at the gate level. Unlike existing logic optimization systems like SIS, VIS can support hierarchical synthesis.

2 Describing Designs for VIS

VIS operates on an intermediate format called BLIF-MV, which is an extension of BLIF, the intermediate format for logic synthesis accepted by SIS [4]. VIS includes a stand-alone compiler from Verilog to BLIF-MV, called VL2MV [3], 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, `$ND`, has been added to specify nondeterminism on wire variables; this is the only legal way to introduce nondeterminism in VIS. Internally, `$ND` introduces pseudo-inputs which can non-deterministically take any allowed value.
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.

Currently, a translator from a subset of VHDL to BLIF-MV is being developed and a path from Esterel to BLIF-MV is available through the POLIS system at Berkeley.

3 Verification In VIS

When a BLIF-MV description is read into VIS, it is stored as a "hierarchy" tree, which is a hierarchical description of the design; it consists of modules that in turn consist of sub-modules. The functionality of a module is represented by a network of gates with arbitrary functionality and latches. The *print_hierarchy_stats* command in VIS prints hierarchy information, and the *print_models* command lists statistics on all the models in the hierarchy. Other useful print commands are *print_io* and *print_latches*.

The hierarchy in VIS can be traversed in a manner similar to traversing directories in UNIX, i.e., a desired node in the tree can be reached by walking up and down with the *cd* command. At any node, simulation, verification and synthesis operations can be performed. The command *pwd* prints the name of the current node. The command *ls* lists all the nodes (submodules) in the current node; *ls -R* lists all the nodes in the current subtree.

We begin by describing the steps involved in converting this hierarchy description into an internal FSM representation. The compound *init_verify* command executes the entire sequence of required initialization commands.

3.1 Flatten

The first step towards verification consists of "flattening" this hierarchical description into a single network (netlist of multi-valued logic gates and latches). The output of the design is a function of its inputs and the latches; this functionality is defined by the functions at the gates, and their interconnections. The *flatten_hierarchy* command creates this network, and the *print_network* command can be used to print it. Other related commands are *print_network_stats* command that prints statistics about the network, and *test_network_acyclic* command that checks the network for combinational cycles.

Note that when a node in the hierarchy is arrived at for the first time during the traversal of the hierarchy using *cd*, there is no network for that node until *flatten_hierarchy* is called for that node.

Also *flatten_hierarchy* automatically checks each table in the network for being deterministic (any non-determinism in the description is taken care of by pseudo-inputs) and completely specified. Since this checking takes some time, it can be turned off safely using the option *flatten_hierarchy -b*, after a BLIF-MV file has been checked once.

3.2 Ordering and Partitioning

The next step towards verification consists of converting this network representation into a functional description that represents the output and next state variables as a function of the inputs and current state variables. We use the BDD (binary decision diagram) and its extension the MDD (multivalued decision diagram) to represent boolean and discrete functions. Before creating the MDDs, it is necessary to order the variables in the support of the MDD. This is accomplished by the *static_order* command, which gives an initial ordering. To undo the current ordering, reinvoke the command *flatten_hierarchy*. At any stage the current variable ordering can be written out to a file using the *write_order* command.

The *build_partition_mdds* command computes the transition function MDDs. Depending on the partitioning method selected, the MDDs for the combinational outputs (COs) are built in terms of either the combinational inputs (CIs) or some subset of intermediate nodes of the network. The MDDs built are stored in a DAG called a "partition". The vertices of a partition correspond to the CIs, COs, and any intermediate nodes used. Each vertex has a multi-valued function (represented by an MDD) expressing the function of the corresponding network node in terms of the partition vertices in its transitive fanin. Hence, the MDDs of the partition represent a partial collapsing of the network. The *inout* method represents one extreme where no intermediate nodes are used, and *total* represents the other extreme where every node in the network has a corresponding vertex in the partition. The *frontier* method creates vertices for intermediate nodes as necessary in order to control the BDD size. The BDD size at which a new vertex is created is controlled by the parameter *partition_threshold*. This partition method encompasses the other two (a large value of partition_threshold leads to "inout" method and a zero value for partition_threshold leads to "total"). The partition graph can be printed to a file with the *print_partition* command. Another related command is the *print_partition_stats* command that prints statistics on the partition graph.

3.3 Dynamic Re-ordering

Dynamic ordering of variables may be enabled and disabled using the *dynamic_var_ordering* command. Dynamic ordering is a technique to reorder the MDD variables to reduce the size of the existing MDDs. Available methods for dynamic reordering are *window* and *sift*. Dynamic ordering (especially using *sift*) may be time consuming, but can often reduce the size of the MDDs dramatically.

Dynamic ordering is best invoked explicitly (using the *dynamic_var_ordering -f <method>* option) after the *build_partition_mdds* and *print_img_info* commands. If dynamic ordering finds a good ordering, then you may wish to save this ordering (using *write_order <file>*) and reuse it (using *static_order -s <method> <file>*). With option *dynamic_var_ordering -e <method>* dynamic ordering is automatically invoked once a certain threshold on the overall MDD size is reached. Enabling dynamic ordering may slow down the verification, but it can make the difference between completing and not completing a verification task.

3.4 FSM Traversal and Image Computation

FSM traversal is the core computation in design verification. Efficient traversal requires grouping the MDDs, in a manner optimal for traversal. To traverse the FSM, the present state, input, and next state variables are organized for easy manipulation. All this information is included in an FSM data structure created in the *compute_reach* command. This also invokes traversal of the entire reachable state set of the FSM representing the design, and may be invoked with different verbosity options to get varying amounts of traversal information. On subsequent calls to *compute_reach*, the reachability computation is not reperformed, but statistics can be printed using *-v*.

The reachability computation makes extensive use of image computation. There are several user-settable options that affect the performance of image computation. The documentation for the *set* command lists these options. Use the command *set image_method* to change the image computation method, and then re-initialize verification (starting at the *flatten_hierarchy* command [5]). The *print_img_info* prints current image information. Notice that while *print_partition _stats* prints information on the next state *functions*, *print_img_info* prints information on the next state transition *relations*. The command *print_img_info* creates transition relations from transition functions by clustering several functions together. The result is a partitioned transition relation. It is often a good idea to force dynamic variable reordering (for instance, *dynamic_var_ordering -f sift*) at this point to reorder these relation MDDs. The reachability computation is an optional step of the model checking algorithm; unreachable states may be used as don't cares to minimize the BDD representation.

3.5 Fairness Constraints

Fairness constraints are used to restrict the behavior of the design. Each fairness condition specifies a set of states in the machine, and requires that in any acceptable behavior these states must be traversed infinitely often (i.e., these states must be on a cycle). Such constraints are called "Büchi fairness" constraints. Fairness constraints are stored in fairness files (with extension `.fair` by convention). A fairness file is read in by the *read_fairness* command. Active fairness conditions can be displayed by means of *print_fairness*. The *reset_fairness* command is used to reset the fairness constraint to "true"; by default, there is one fairness condition that contains all states.

Fairness constraints remove unwanted behavior from a system. They are a powerful tool, but should be used with care, because it is easy to make a faulty system pass wanted properties by a careless use of fairness constraints. Also, it has been observed that verification is slowed down considerably when there are many fairness constraints.

3.6 Language Emptiness

The language of a design is given by sequences over the set of reachable states that do not violate the fairness constraint. If the language is empty, we know that the system does *not* exhibit any legal behavior. VIS supports the command *lang_empty* as an alias for model checking the formula *EG true*. This is relevant when verification is done by adding a component which monitors the design and has fair runs precisely when the observed behavior does not satisfy the specification. Before invoking model checking, *lang_empty* can also be used to ensure

[5] Whenever a hierarchy is reinitialized, the option *flatten_hierarchy -b* can be used safely for efficiency.

that the system is non-trivial. This is pertinent because the fairness constraints specified may make the entire system "unfair", and an empty system passes all universal properties.

3.7 Model Checking, Invariants and Abstraction

The *model_check* command calls CTL model checking in VIS. Formulas to be checked are specified in a file which is an argument to the command.

An important class of CTL formulas is *invariants.* These are formulas of the form $AG\ f$, where f is a quantifier-free formula. The semantics of $AG\ f$ is that f is true in all reachable states. The command *check_invariant* implements an algorithm that is specialized for these formulas.

When performing model checking and checking invariant properties, one can use the reduce option *-r*, to perform model checking on a "pruned" FSM, i.e., one where parts of the network that do not affect the formula (directly or indirectly) have been removed.

This mechanism can be combined with the abstraction mechanism available through the command *flatten_hierarchy -a <file>*. *<file>* contains the names of variables to "abstract". For each variable `x` appearing in *<file>*, a new primary input node named `x$ABS` is created to drive all the nodes that were previously driven by `x`. Hence, the node `x` will not have any fanouts; however, `x` and its transitive fanins will remain in the network. Abstracting a net effectively allows it to take any value in its range, at every clock cycle. This mechanism can be used to perform manual abstractions.

3.8 Error Tracing

VIS produces a debug trace to help the designer understand the cause of the failure. Common corrective actions are the correction of an error in the original system description or addition of fairness constraints.

If model checking or language emptiness checks fail, VIS reports the failure with a counterexample, i.e., an error trace of sample "bad" behavior (behavior seen in the system that does not satisfy the property - for model checking, or invalid 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.

3.9 Combinational and Sequential Equivalence

In VIS it is also possible to check the equivalence of two networks. The command *comb_verify* verifies the combinational equivalence of two flattened networks. In particular, any set of functions (the roots), defined over any set of intermediate variables (the leaves), can be checked for equivalence between two networks. Roots and leaves are subsets of the nodes of a network, with the restriction that the leaves should form a complete support for the roots. The correspondence between the roots and the leaves in the two networks is specified in a file.

The default option assumes that the roots are the combinational outputs and the leaves are the combinational inputs. Two networks are declared combinationally equivalent if they have the same outputs for all combinations of inputs and pseudo-inputs. An important usage of *comb_verify* is to provide a sanity check when using SIS to re-synthesize portions of a network, as explained in Section 3.11.

The command *seq_verify* tests the sequential equivalence of two networks. In this case the set of leaves has to be the set of all primary inputs. This produces the constraint that both networks should have the same number of primary inputs. The set of roots can be an arbitrary subset of nodes. Moreover, no pseudo-inputs should be present in the two networks being compared. Sequential verification is done by building the product finite state machine. The command *seq_verify* verifies whether any state, where the values of two corresponding roots differ, can be reached from the set of initial states of the product machine. If this happens, a debug trace is provided.

3.10 Simulation

Simulation, although not "formal verification", is an alternate method for design verification. After the command *build_partition_mdds* is invoked, the network can also be simulated. In VIS we provide internal simulation of the BLIF-MV description generated by VL2MV, via the *simulate* command. Thus, VIS encompasses both formal verification and simulation capabilities. *simulate* can generate random input patterns or accept user-specified input patterns.

Any level of the specified hierarchy may be simulated; the user may traverse the hierarchy to reach the relevant level via the *cd* command and simulate only that part. The *init_verify* command must be called to set up the appropriate internal data structures before simulation.

3.11 Synthesis in VIS

VIS can interact with SIS in order to optimize the existing logic. There are two possible goals/scenarios:

1. Synthesis for verification : Synthesis can be used to optimize the logic that represents the system, for simpler verification.
2. Front-end to synthesis : Files described in Verilog and compiled into *blif_mv* (using VL2MV or another tool) can be synthesized by using VIS and SIS together.

A key fact is that only the current level of the hierarchy is sent to SIS, and not the subtree rooted at the current node. [6] Modules at a lower level are treated as external and the boundary variables are carefully preserved, by reintegrating

[6] One would need a flattening routine different from the one which starts the verification flow already in VIS, and such a routine to flatten for synthesis is not yet available.

their multi-valued status after the optimization step in SIS (SIS requires that boundary variables are completely encoded, i.e., are binary variables).

4 Concluding Remarks

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

One of the key goals of VIS is to serve as a platform for developing new verification algorithms. We have used the object-oriented programming style of SIS as our paradigm. 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.

For more information about VIS or to get a copy, visit the VIS home page [5].

Acknowledgments

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

The development of VIS was supported in part by the SRC under contract 96-DC-324 and by generous financial contributions from BNR, Cadence, California Micro, Fujitsu, and Motorola.

References

1. Kenneth L. McMillan. Symbolic Model Checking. Kluwer Academic Publishers, 1993.
2. R. K. Brayton et al. HSIS: A BDD based system for formal verification. Proc. of Design Automation Conference, 1994.
3. S.-T. Cheng. Compiling Verilog into automata. Tech. Rep. UCB/ERL M94/37, May 1994.
4. E.M. Sentovich et al. SIS: a system for sequential circuit synthesis. Tech. Rep. M92/41, May 1992.
5. VIS Home Page : http://www-cad.eecs.berkeley.edu/~vis