The Automatic Generation of Functional Test Vectors for Rambus Designs

K.D. Jones, J.P. Privitera
kdj@rambus.com, jpp@rambus.com
Rambus Inc.,
2465 Latham St., Mountain View CA 94040

Abstract. *We present a method for the automatic generation of test vectors for functional verification, giving the advantages of random and directed testing. We show the use of a formal specification as input to a test generator. We present techniques for the efficient implementation of the generator. We discuss our experience with this method applied to commercial designs. We show how our approach is a stepping stone towards practical formal verification.*

1.0 Introduction

1.1 Background

Current approaches to functional verification range from the ideal (various flavors of formal verification[1]]) to the pragmatic (simulation driven by directed tests). The latter, usually done in a language like Verilog[2], is generally the standard approach to verification in many industrial settings. Usually, the more comprehensive the verification, the higher is the cost associated with the method, both in terms of resources and necessary skills. We believe in the ideal but are restricted by the practicalities of commercial existence to technologies that we can immediately use to verify commercial designs. We have developed a test strategy and toolset that offers an incremental path to formal verification, combining the advantages of random and directed testing in an existing environment.

1.1.1 Rambus Designs

Rambus[1] designs Rambus DRAMS (RDRAMS), ASICS and related products. RDRAMS are 500 Megabyte/second memory chips, with a bus based protocol and a comprehensive set of masking operations. RDRAMS have an instruction set containing 30 operators and a register state of 10 registers, parameterizing the device operation. It is important to distinguish, at the outset, the problem of verifying Rambus DRAMs from the typical DRAM test problem. At Rambus, we are concerned solely with the functional verification of the Rambus protocol logic, not with verification of the DRAM core. Indeed, in our simulations the core is always modeled in software. The combined Rambus protocol interface and core model has the flavor of a simple microprocessor and our method is applicable to microprocessor verification.

1. For further information on Rambus, see http://www.rambus.com.

A significant aspect of our approach is using a formal specification to drive the test generation. Rambus has a business model that involves many partners implementing from a common specification, which means an accurate specification is fundamental. Given the existence of such a document, it seemed natural to take advantage of it, and the format of the specification has evolved to be suitable for test generation.

1.2 Our Approach

The Rambus Automated Verification System (RAVS) offers an environment for the specification-directed generation of test vectors. These vectors may be symbolic, for use with a formal verification tool, or tri-state for input to a conventional Verilog simulator. RAVS combines the strengths of randomized, X-based and directed testing, since all values will be left as general as is consistent with the constraints expressed in the specification.

Since this system is in use in an industrial setting, attention has been paid to relevant pragmatics, and RAVS is incorporated into the standard document production and simulation environments. This ensures that testing is an integral part of the specification and design process.

1.3 Overview of the paper

Section 2.0 presents the specification language, RS. Section 3.0 presents details of the vector generation. Section 4.0 presents our experience in using RAVS to verify a commercial design. Section 5.0 presents current work and discusses the incremental path to using RAVS as a practical formal verification system.

2.0 Formal Specification using RS

2.1 General

The formal specification completely describes the correct functional behavior of the device. This document serves as a design reference for implementors and as the source for generating test vectors.

33rd Design Automation Conference ®
Permission to make digital/hard copy of all or part of this work for personal or class-room use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 96 - 06/96 Las Vegas, NV, USA
©1996 ACM, Inc. 0-89791-833-9/96/0006.. $3.50

The most important characteristic of the formal specification is that it defines the *user-visible behavior* of a device. In general, the user of a device is not concerned with its internal structure, only with its behavioral characteristics. Very often in the literature of formal verification [3] a specification of a device is taken to mean an abstraction of the device. Such an abstraction might be suitably represented in an RTL description, or the behavioral component of a commercial HDL, or a specialized state-machine description language, to name a few possibilities. An RS specification is generally quite different; it expresses not the logical structure of a device, but rather its operational interface. If RS were used to describe a processor, for example, the specification would limit itself to the operations supported by the device and as little of the internal state as is necessary to forecast operational outcomes.

We have found it convenient to express the operational behavior of a hardware device as collection of constraints; thus RS is designed to be suitable for expressing such constraints. An RS specification is generally not suitable for simulation. However, it serves two purposes:

1. It defines the correct behavior of the device. Given a complete set of stimuli and responses, it defines whether this represents a correct behavior for the device. Given a set of stimuli, it defines the corresponding output from a correctly behaving part.
2. It allows the generation of sequences of stimuli and responses that characterize correct function of the device.

To ensure that there is consistency between the published specification and the generated tests, we use the same source for both, and embed the machine-readable specification within the text of the published document. The specification text is automatically extracted from the document as part of test generation. A secondary advantage of this strategy is that the specification can be presented taking full advantage of the features of a document production system to enhance legibility.

2.2 The RS Language

RS is a comparatively simple language for defining temporal constraints on the IO behavior and internal states of a device, giving a specification that is formal and intuitively clear. We do not have the space to give a full exposition, so we restrict ourselves to presenting the general concepts and illustrating the language sufficiently to motivate the following sections. A complete description of RS, together with a specification of a Rambus DRAM, can be found in the RDRAM Specification[4].

An RS specification consists of a set of primitive names and a set of constraints on the values associated with these names over time[1]. The complete set of value sequences characterized by this model identifies all correct behaviors of the device.

2.2.1 Names

The name space of a specification consists of:

- *signals* representing values that hold only when explicitly driven, such as the pins on the device. At any time for which there is no value explicitly defined, a signal will be X, unless there is an explicit default value associated with its definition.

1. We assume a base clock of sufficient discrimination.

TABLE 1. External Signals

Name	Range	Polarity	Type	Default	Gen
BusCtrl	n/a	-	Bidir	Pulldown	Any
BusData	[8:0]	-	Bidir	n/a	Any

- *state variables* representing internal values that persist until explicitly changed, such as internal registers.

Signals can be grouped into buses, with the individual signals accessed by indices. RS allows reverse polarity, and bi-directional properties to be specified.

For example, we specify a simple channel[2] consisting of a single control signal and a data bus as shown in Table 1.

Packets: Packets are an abstraction mechanism for grouping related signals over a contiguous interval of time.

For example, an element of a simplified RDRAM protocol is the channel tiled as shown in Table 2.

TABLE 2. Request Packet

Clock	BusCtrl	BusData								
		[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]:
0	Start	Op [0]	Adr [9:2]							
1	Op[1]	Op [3]	Adr [17:10]							
2	U	Adr [26:18]								
3	Op[2]	Adr [35:27];								

This table defines a packet, Request, which has the following properties:

- it occupies the channel for 4 clock ticks.
- it has fields:

  Start, Op[3:0] and Adr[35:2]

- there is an abstraction mapping relating these constructs to actual pin locations, relative to invocation time of the packet.

The packet makes it more convenient to write constraints. For example, if we want to constrain the Request.Op to be a register write, represented by the opcode constant RegisterWrite, then we

2. For historical reasons, we refer to the collection of I/O signals as a channel.

could write as part of a constraint 'Request.Op == RegisterWrite' and, relative to the current time, the signals corresponding to the bits of Op will be constrained to have values that match RegisterWrite.

State Variables: variables are named objects whose values persist over time. Once a variable is constrained to have a value, this value will persist until explicitly changed.

Below, we use a variable that indicates if the device is busy at the current time, to ensure there are no overlapping requests. The statement in Table 3 declares DeviceBusy as a variable with an

TABLE 3. State Variable

```
State DeviceBusy=1'b0;
```

initial value of **logical 0**.

2.2.2 Constraints

Not all sequences of state instantiations denote valid behaviors. RS allows the definition of sets of constraints which restrict the possible state instantiations to only those considered valid.

Constraints consist of relational operators over the state elements, together with time shifting operations -- allowing the relationships between states in the sequence to be expressed. Each constraint is implicitly a function of the current time.

A transaction representing an operation with associated data for our simplified device is shown in Table 4

TABLE 4. An Example Constraint

```
Constraint Transaction =
        Packet(Request)&
         Request.Start == 1'b1 &
         @t_DATATIME{
                DeviceBusy == 1'b0 &
                Case {
                 Request.Opcode == ReadOp =>
                        ReadData,
                Request.Opcode == WriteOp =>
                        WriteData}};
```

If this constraint can be satisfied, then the state instantiations define the well formed transactions relative to the current time.

2.2.3 Assertions

Since the specification is completely general, given infinite time and space it would generate all correct behaviors for the device. In practice, we usually need to focus our attention more specifically, to get meaningful coverage in a reasonable time[1]. Assertions are additional constraints that further restrict the valid behaviors to a subset of the full state space.

1. In Section 5.0, we discuss how we try to increase the generality we can practically handle by using symbolic values.

For example, if we wish to generate only tests that are composed of a MemoryWrite followed by a MemoryRead from the same address, we add the extra constraint shown in Table 5. Anything

TABLE 5. A Simple Assertion

```
Assert "Simple Memory Test"
  "Write Memory location and read value written"
  Local column = Arbitrary(8) {
        Request.Opcode == MemoryWrite &
        Request.Adr[10:3] == column &
        Transaction &
        @t_NEXTTRANSACTION {
                Request.Opcode == MemoryRead &
                Request.Adr[10:3] == column &
                Transaction}};
```

not explicitly constrained by the assertion is allowed to be any value consistent with the full specification, so we need only constrain a small number of items.

Note: This assertion is deceptively simple, so we should be clear about what it actually implies. Any behavior that satisfies this easily expressed relationship could be generated by this assertion. The full RDRAM supports more that 20 operations, involving bit and byte masking, sequential and non-sequential addressing, that are valid MemoryWrites. There are a number of choices for MemoryRead. The transactions can be of arbitrary data length, and may even be terminated early. Any solution involving any of these combinations is a valid instantiation of this assertion. We are allowing a great deal of freedom of choice here, insisting only on the highest level property: that it be a write followed by a read.

2.3 Summary

The RS language uses a simple state model together with a set of constraint expressions to completely define the correct behavior of a device. This specification is embedded in a document constructed using Framemaker. This document is formatted for human readability and the formal text is mechanically extracted for automatic input to the test generation tool. Assertions are used to generate "interesting" behaviors for test purposes.

In the next section, we see how such a model can be given an operational semantics and can be solved to produce meaningful tests for the device.

3.0 RAVS Test Generator

RAVS is both the name of the integrated toolset used to verify Rambus designs, and the name of the program which lies at its core: a test generator that converts specifications and assertions about device behavior written in the RS language into test vectors suitable for running on a simulator. This section describes the RAVS test generator.

RAVS is written in the Scheme Lisp[5] dialect and runs on both Suns and HP Snakes. It implements a small, specialized programming language, basically the core of the specification language RS that results from deleting RS' many syntactic sugaring devices. A RAVS program comprises a device specification, and an assertion. The output of a program is a set of tests for the assertion.

One of the principle challenges in implementing RAVS was to ensure sufficient efficiency to allow the production of enough tests within a practical time period.

3.1 The Structure of RAVS Tests

A RAVS test is a sparse array of stimulus and response patterns at the IO boundary of a device, in increasing temporal order. We refer to such an array as a *pin buffer.*

For each assertion A, RAVS generates a sequence of tests T_A, the length of which is determined by a user-specified parameter. T_A has three important properties:

1. Each test in T_A obeys the specification's definition of a valid stimulus-response pattern (Correctness).
2. If T_A is generated with no length limit, it would constitute a complete set of tests for A (Completeness).
3. Tests in the sequence T_A are produced in random order (Randomness).
4. No test in T_A can be strengthened by substituting an X for a 0 or 1 in its stimulus section (Maximal strength).

The need for Correctness is obvious. Randomness is vital when small numbers of tests are generated for an assertion for it assures that radically different variations on the same assertion can be examined within the test sequence length budget.

It may seem that Completeness is a useless property under the practical limitation of finite test sequences, but Completeness implies that there are no classes of behaviors that are in principle untestable. Coupled with Randomness, Completeness provides some assurance that the generated tests will exercise the corner cases of the design.

Maximality of strength means that the tests cannot be generalized by introducing don't-cares for 0's or 1's in the stimulus. A maximally strong test is one that may cover many billions of more specialized tests. In practice, Maximal Strength is a property that can be preserved by the test generator, but it is very dependent on the specification. It is easy to write correct specifications that are nonetheless overconstrained.

3.2 Constraint Solving

RAVS creates tests via a process of constraint-solving. A RAVS specification is a list of constraints that define the correct IO behavior of a device. An assertion is a conjunction of constraints on the behavior of the IO pins over time, based on the specification. A test for an assertion is a consistent solution of all the assertion's constraints. RAVS' task is to produce many such solutions in an efficient manner

There are many constraint solving techniques, but the ones employed in RAVS derive from logic programming languages such as Prolog[6]. Prominent among these are backtracking, unification, and memo-ization.

3.2.1 Backtracking

Backtracking is a depth-first strategy for finding solutions to a set of constraints. At various points in RAVS' execution, a set of choices may be encountered. RAVS picks one of the choices and continues execution, but if the choice just made cannot lead to a solution, if, for example, there is an incompatibility with a later constraint, RAVS returns to the choice point and tries a different alternative. If no alternatives are left, RAVS backtracks out of the choice point.

All RAVS statements are compiled into two-argument Scheme functions. The first argument is the time point at which the constraint is executed. The second argument is a function called the "continuation" of the statement. When the statement has successfully computed its value, that value is passed to the continuation. If the statement fails to compute a value the continuation is not called and the statement returns normally, giving the effect of backtracking.

3.2.2 Unification

Given two expressions e_1, e_2, unifying e_1 and e_2 is the process of determining a substitution of expressions for the variables in e_1, e_2 that makes them equivalent expressions. The definition of a unifying substitution for e_1, e_2 depends on the algebra over which e_1 and e_2 are expressions. In Prolog, this is the algebra of free terms. In RAVS it is boolean algebra.

RAVS uses the Coudert-Berthet-Madre[7] cofactoring method for boolean unification. A test sequence *TA* of length n requires n invocations of the cofactoring algorithm. Unlike Prolog, in which a unification statement $A == B$ would cause invocation of the underlying unification algorithm, $A == B$ in RAVS merely adds a new boolean fact to a collection of all known facts. The "collection of all known facts" is a boolean expression K in OBDD form; $A == B$ is processed into another expression of the same kind; adding $A == B$ to the collection entails setting K to the boolean AND of $A == B$ and K. K can become inconsistent (0) as a result of this operation, in which case backtracking is initiated, but if execution of an assertion ends with a consistent K the values of every signal and state element at every time point are cofactored with respect to K to produce test vectors. These test vectors can still contain boolean expressions; an extra step is required to instantiate all boolean variables to boolean constants.

3.2.3 Temporal Constraints and Memo-ization

RAVS is a temporal language in the sense that all constraints are executed with respect to a current time point which may be shifted by various temporal operators. RAVS is also a nondeterministic language in the sense that a call to a constraint may return one of several valid results[1]. However, when a constraint is executed twice at the same current time point and with identical arguments, it must return the same result both times. The second call must agree with the result of the first, even if the first was randomly chosen. It is inefficient to allow RAVS to re-execute a constraint under these circumstances. The solution is to "memo" all constraint computations, that is, to store in memory the result of every constraint call and to make the second invocation a simple look-up. This is implemented by a hashing scheme.

3.3 Boolean Expressions and Patterns

Boolean expressions are represented by a variant of ordered binary decision diagrams (OBDDs)[8]. The same implementation of memo-ization that is used for constraints applies to OBDDs. A

1. This is required for test sequences to exhibit the Randomness property.

popular implementation technique for OBDDs is to use two hash tables: a “unique table” to maintain the property that distinct nodes represent distinct boolean functions, and a “computed table” to hold the results of boolean function evaluations. Both the unique table and the computed table are merged with the random heap used for memo-ization.

The implications of the fact that patterns are constants which remain unchanged in meaning even when used in unifications are profound in a system such as RAVS in which unifications with patterns occur frequently. A naive implementation of patterns would force the system to create new boolean expressions every time a pattern is used. Thus, every invocation of a pattern would require new OBDDs to be created even though the OBDDs created from a pattern *P* are isomorphic up to variable relabeling.

The data structure used in RAVS to represent boolean expressions abstracts the variable labeling from the OBDD data structure using the technique of *variable shifters* as described in [9].

4.0 Case Study: an 18 Megabit RDRAM

RAVS has been applied to a number of designs, some of which are now in production as commodity parts. To show how the techniques work in practice, we present an outline of the environment and process used for the verification of an existing design.

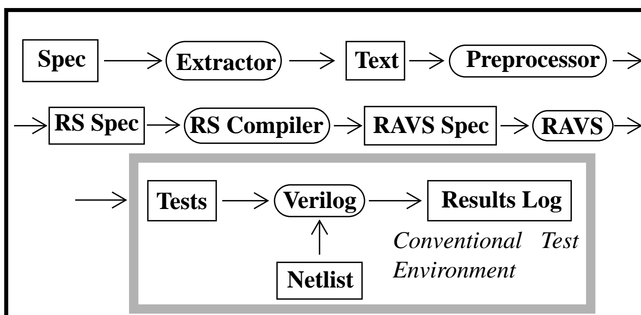
4.1 The Example

The specific project discussed here was *RB*, a generic 18 Megabit RDRAM design, implemented independently by two of Rambus' manufacturing partners.

4.2 The Environment

The RAVS environment in use at Rambus is shown in Figure 1

FIGURE 1. The RAVS Environment



The specification was developed and published as a report, using Framemaker 4. From this, we used the Frame batch programming language to extract the text. A preprocessor removes irrelevant text and the formal text is passed to the RS translator which produces RAVS input. From this, RAVS generates a number of pin buffers. Each pin buffer is used to drive a Verilog simulation against a netlist extracted from the design database. The simulation produces output logs showing either successful completion or the number and location of errors.

Human action is needed only to provide the specification and the design, and to analyze the log in the case of failures.

4.3 Specification

For the current generation of Rambus DRAMS, the specification of the device, including both the physical and the logical descriptions, is a 150 page document. The formal text within this document consists of 5000 lines of RS.

4.4 Assertions

We used an assertion suite containing 120 assertions. An assertions may describe general behaviors, such as a read following a write retrieves the correct value, or a very specific circumstance, such as a specific combination of writes and powerdowns does not cause cache loss. General assertions ensure broad coverage of functionality in a way analogous to random tests. Specific assertions allow focus on complex or difficult aspects of the design analogous to directed testing, while still allowing “unusual combinations” to happen.

4.5 Tests

For a complete regression, we generated 128 sections (a section is one complete instantiation of a pin buffer) for each assertion. This gives the equivalent of approximately 16000 directed tests, on the order of 10 Million vectors.

To generate these tests, takes approximately 136 hours on HP Snake workstations. To execute the resulting tests, took 92 hours using Verilog-XL.

4.6 Results

RAVS verification followed testing with a hand written suite of directed tests. This suite is as complete as many of the total verification suites used for some designs in current use and so the design had already been “verified” before RAVS was run.

In early tests, running only 32 sections, we found a small number of errors missed by the directed tests. For full regression, we ran 128 sections and found further problems -- some of which were detected due to unusual combinations of circumstances as a result of the random instantiation in the tests.

To date, there have been no functional errors found in designs that passed RAVS verification.

5.0 Conclusions and Future Work

5.1 The Method

Our experience shows that use of RAVS can give greater meaningful coverage, for equivalent effort, than directed or random testing. The combination of constraining the general form of the test, by an assertion, but still allowing randomness in the choices, results in concentration of effort on significant areas within the design but still allows unusual combinations to be tested. The designers admitted that some of the errors found by RAVS were unlikely to

have been discovered so early without it, since no one would have been likely to have written a test that had quite the characteristics that discovered the problem.

The major drawback of the approach is that RAVS still depends on simulation and so the results are ultimately only as good as the number of sections that can be run. It does not offer verification in any stronger sense. We believe that there is significant worth even in this level of automation. In Section 5.3, we discuss how we can make further steps towards verification using the RAVS concepts.

5.2 RAVS in Practice

Incorporating RAVS into an existing simulation/verification environment is not difficult: the greatest cost is to develop a sufficient specification. If a formal specification already exists in a suitable form, all that is necessary is to add the framework that allows the RAVS generated tests to be used as if they were manual tests. If no such document exists, then it has to be provided, but this gains the extra benefits that are derived from having a complete and concise specification of the device.

The resources needed to use RAVS are of the same order as those necessary to run Verilog. The test generation process takes approximately 60% of the runtime. The upper bound on the number of sections that can be run is a function of the memory size of the scheme implementation and the Verilog simulator. We found that 128 sections was a good compromise between coverage, time and practical limits. The limiting factor in our environment was the capacity of the existing Verilog.

RAVS is now used for verification of all RDRAM designs and has been successfully used on other projects at Rambus.

5.3 Formal Verification

RAVS can be used to create tests for symbolic simulators. RAVS' natural output is, indeed, boolean expressions; it must go through an extra processing step to instantiate boolean variables to 0 or 1 when it creates tests for standard three state simulators. We have addressed one of the practical difficulties of symbolic simulation, namely, providing input and expected output sequences for simulation

In experiments, we have used COSMOS[9] in symbolic simulation mode and we have found a single symbolic test, in our environment, can cover a billion three-state tests. Given a sufficiently general assertion, symbolic simulation represents full verification of the design. Our results have encouraged us to incorporate symbolic simulation into our future plans, and we are currently evaluating a RAVS/COSMOS combination prototype.

6.0 References

1. G. Birtwistle and P.A. Subrahmanyam, Eds., *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.
2. *Verilog-XL Reference Manua*l, Cadence Design Systems Inc., 1991
3. R. Kurshan, *Computer-Aided Verification of Coordinating Processes*, Princeton University Press, 1994.
4. R. Barth, *RDRAM Annotated Specification*, Internal Report, Rambus Inc., 1995
5. Jonathan Rees and William Clinger, Eds., “Revised Report on the Algorithmic Language Scheme,” in *ACM SIGPLAN Notices*, vol. 21, no. 12, December 1986, pp. 37-79.
6. W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
7. Olivier Coudert, Christian Berthet, and Jean-Christophe Madre, “Verification of Sequential Machines Using Boolean Functional Vectors,” in *Proc. IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design*, Nov. 1989, Leuven, Belgium, pp. 179-196.
8. Randall E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation,” in *IEEE Transactions on Computers*, vo. C-35, no. 8, August 1986, pp. 677-691.
9. Shin-ichi Minato, *Binary Decision Diagrams and Applications for VLSI CAD*, Kluwer Academic Publishers, Boston, 1996.
10. Randall E. Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, and Thomas Sheffler, “COSMOS: A Compiled Simulator for MOS Circuits,” in *Proc. 24th ACM/IEEE Design Automation Conference*, 1987, pp. 9-16