A Comparison of Three Verification Techniques: Directed Testing, Pseudo-Random Testing and Property Checking

Mike G. Bartley[1]
Elixent Ltd
Castlemead, Lower Castle Street
Bristol BS1 3AG
Mikebartleyuk@Yahoo.co.uk

Darren Galpin
Infineon Technologies UK Ltd
Infineon House, Great Western Court
Hunts Ground Road, Bristol BS34 8HP
+44(0)117 952 8741
Darren.Galpin@Infineon.com

Tim Blackmore
Infineon Technologies UK Ltd
Infineon House, Great Western Court
Hunts Ground RoadBristol BS34 8HP
+44(0)117 952 8745
Tim.Blackmore@Infineon.com

ABSTRACT

This paper describes the verification of two versions of a bridge between two on-chip buses. The verification was performed just as the Infineon Technologies Design Centre in Bristol was introducing pseudo-random testing (using Specman) and property checking (using GateProp) into their verification flows and thus provides a good opportunity to compare these two techniques with the existing strategy of directed testing using VHDL bus functional models.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids – *Verification.*

General Terms

Verification.

1. INTRODUCTION

The verification of two versions of a bridge between two on-chip busses is described. The verification methodology incorporated directed testing, pseudo-random testing and property checking at module level, as well as in-system testing. Only the module-level verification is considered in detail. The incorporation of the three techniques into a verification flow is described in Section 3. In Section 4, results are used to discuss the relative strengths of each verification technology, as well as the strength of the verification provided by all three combined. Since verification is not an exact science, rather than trying to draw hard and fast conclusions, recommended use of these technologies is presented based on the experience gained in Section 5. Some ideas as to how the three technologies may more easily be used in conjunction are included.

2. BACKGROUND

2.1 The Design-Under-Test (DUT)

The LFI is a bus bridge that connects a 32-bit address, 64-bit data local memory bus to a 32-bit address, 32-bit data flexible peripheral bus. The LFI is approximately 30K gates in size. Two versions were verified - the first version (LFI-IBC32) required that the DUT could work with LMB:FPI clock ratios of 2:1 and 1:1. The second version (LFI-S) extended the LFI-IBC32 functionality to support any clock frequency ratio so long as the LMB frequency is equal to or greater than the FPI frequency, and as long as the positive edges of the clock signals align. In addition, power saving features were incorporated into the LFI-S version.

2.2 Methodology

Four main techniques were applied to the verification of the LFI:

- Directed testing using VHDL BFM's for the test bench and transaction-based tests for directed tests.
- Pseudo-random constraint driven testing using Specman from Verisity.
- Formal property checking using GateProp from Siemens.
- The LFI was placed within a larger system and verified as part of that system. This system is a reference system used solely for the purpose of verifying system level aspects of a DUT that are hard to verify at module level. For example, the LFI can go into power down mode with other parts of the system when a certain set of conditions apply and the LFI then engages in a complex handshake with the CPU and other blocks. The complete operation is better verified at system level than at module level.

This paper is more concerned with module level verification and so will concentrate on the first three techniques.

Figure 1 indicates when each technique was applied during the project. There follows a brief explanation of why these techniques were applied at these times and how they were applied.

- When the LFI-IBC32 project started the only infrastructure available was the VHDL Bus Functional Models (BFM's).

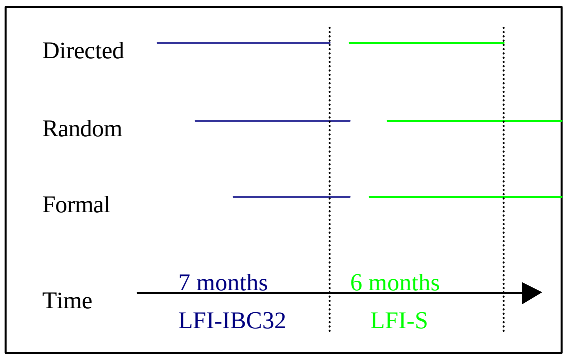

Figure 1: Verification time-line

[1] Mike Bartley was with Infineon Technologies when this work was carried out

Hence a directed test bench and directed tests was developed first.

- The directed tests were completed about halfway through LFI-IBC32 but were continually run in regression and were enhanced to achieve the structural coverage targets.
- The LFI-IBC32 directed test bench and tests were ported to LFI-S and used for regression and structural coverage there too.
- e language Verification Components (eVCs) for the FPI and LMB became available during the LFI-IBC32 project and were then used to build a LFI Specman test bench (this accounts for starting the random testing later for LFI-IBC32). This test bench was then ported to the LFI-S project.
- GateProp was a new tool to the Infineon Technologies Design Centre in Bristol and some training and infrastructure development was required before it could be used (this accounts for starting the property checking later for LFI-IBC32). The GateProp environment and properties were however very quickly ported to LFI-S.

.

3. VERIFICATION TECHNOLOGIES

3.1 Directed testing

Directed testing started with the writing of a test specification document and a test bench description document by a separate verification team to give the independent interpretation discussed in Bergeron (2000). The test specification is written using a "black box" approach from the verification engineers understanding of the DUT spec. The document aims to test fully all of the stated functionality in the DUT spec, and it is from this that the directed tests are written. The writing of the test spec forms a very good review of the DUT spec and can uncover discrepancies or holes in the DUT spec. The test bench spec is aimed at describing the test bench features such that the tests to be run on it are able to exercise all of the design features and to allow for the implementation of the test specification. The design and verification teams reviewed both specifications.

The directed test bench is built from internally supplied BFM's, written in VHDL. These are driven by a simple test language that allows the test to perform read and write transactions plus idle cycles. The test is thus written at the transaction level and the BFM translates the transactions to the appropriate signal behaviour on the bus. The BFM's allow the DUT to be stimulated across the full range of functionality specified in the bus protocol. Adherence to the protocol by all parties is monitored by a protocol checker - another VHDL module which plugs directly onto the bus.

The directed test bench used consisted of:

- Four BFM Masters and Slaves, plus a protocol checker on the LMB bus.
- Two BFM slaves and masters, plus protocol checker and local memory unit on the FPI bus.

Once the directed tests have been written and run on the DUT, structural coverage is performed using a suitable third party tool to provide defined coverage metrics such as Statement Coverage, Branch Coverage and Toggle Coverage. Pre-defined targets of 100% *justified* statement coverage ("justified" as some statements are unexecutable, such as some "when others" statements), >95% branch coverage and 100% toggle coverage. These structural coverage targets form a necessary but not a sufficient quality target. They are a safety net – whilst your verification has not reached the targets you know you haven't done enough, but reaching the targets does not mean completion of verification. Experience shows that the tests derived from the black box test spec are insufficient to reach these structural coverage targets and so extra tests are specified and implemented using a "white box" approach. The white box approach works by looking at the coverage holes with a designer and specifying specific transaction sequences that need to be performed in order to hit the previously uncovered holes. This results in a set of very targeted tests.

As well as relying on structural coverage metrics, directed testing also aims to demonstrate bus compliance. The bus protocol specifies certain scenarios that have to be performed correctly in order for bus compliance to be achieved. A compliance document with tick boxes is used, and for each sequence the test which demonstrates this is supplied.

3.2 Random testing

Random testing was performed using Specman Elite and dedicated in-house *e*VC's (*e* language verification components – comparable to BFM's). The testbench was constructed using configurable numbers of *e*VC masters and slaves on both busses, together with an *e*-protocol checker on each bus. The system clocks were driven from the VHDL top level in order to improve simulation performance, but otherwise all stimuli were generated from the *e*-code itself.

A test specification is again written in advance based on the DUT spec - the document describes the scenarios that need to be covered. Similarly a test bench specification is written. Both are reviewed with the designers who may suggest additional scenarios.

Coverage scenarios fell into two main types for both of the DUT's considered in this paper.

1) Transaction coverage. On a per-master and per-slave basis, the types of transactions applied and received are recorded, along with associated information such as acknowledge codes, byte lanes used, wait states observed. Cross-coverage of these transaction types is then performed, in order to ensure that for each opcode type we have observed every error condition, every type of wait state and so on.
2) FSM transition coverage. We hook Specman to the internal state machines within the DUT itself, and define what the

legal transitions are. During the course of the testing, we can measure coverage of every legal transition, and check that we did not cause any illegal transitions.

The combination of ensuring that we apply every possible cross-coverage point and FSM transition ensures that we achieve a high functional coverage of the DUT, and any areas that we initially miss during random testing are rapidly highlighted in the coverage results. The creation of new test scenarios with modified transaction constraints can then be added to ensure that we achieve 100% of the coverage that we have defined.

Cross-coverage of all opcode types with all byte lanes and all errors is impossible, as it can generate impossible to hit combinations. All impossible combinations and FSM transitions were removed from the coverage targets, so that a simple check for 100% coverage can be made at sign-off.

3.3 Formal

Formal verification was carried out using the Siemen's property checker, GateProp (see Bormann and Spalinger (2001) for details of this tool). The main tasks involved in property checking are writing and testing constraints and writing and running properties.

Earlier property checkers could not easily deal with large designs, suffering 'state space explosion'. One of the ways that GateProp overcomes this is not to do any reachable state analysis[1]. Hence it is the responsibility of the user to ensure that there are no false negatives due to internal signals in the design being in an unreachable state. This involves checking each failing property and developing a set of constraints on the internal signals so that they remain in a suitable superset of all possible reachable states. Both of these are very time consuming. Also, the constraints are not going to be re-usable on other designs, and in fact may be difficult to maintain across changes in design. It was therefore decided to start all properties from reset, this being the simplest valid state to define. It should be stated that more general approaches have successfully been applied in other industrial applications of GateProp, where a very high functional coverage has been achieved. In most of these projects the always-start-from-reset approach would not have touched the most interesting functionality, e.g. in telecommunication systems where you have frame lengths of thousands of cycles with corresponding serial synchronisation protocols. Admittedly, these approaches make limited use of some internal states, yet important ones only which make sense at an abstract level. They allow at least a certain degree of reusability of properties.

Given our always-start-from-reset approach, to ensure that the design did not reach an invalid state it was only necessary to constrain the inputs to model the DUT's environment. The goal here is to allow the inputs to take any legal combinations and sequences of values, in order to find corner cases. Of course, random testing also chooses inputs from inside a set of constraints. The difference here is that the property checker 'intelligently' tries to find a counter-example, or else prove, a property. However since the property checker is potentially exhaustive, it moves through clock cycles considerably slower than directed or random testing.

We split properties into two types - 'bus protocol properties' and 'transaction properties'. Transaction properties drove explicit transactions across the bridge, checking that addresses, control signals and data passed through correctly, and that the bridge responded appropriately to the various acknowledgements from bus slaves. The property checker has the advantages that all legal behaviour around the transaction was allowed, and the address and data were checked symbolically - in effect testing all possible values. The slowness of moving through clock cycles was a big limitation however. The transaction properties were sensitive to timing (e.g. the number of clock cycles it took each type of transaction to pass through the bridge). This made them more time consuming to write, with a degree of trial and error being necessary, and also to maintain since cycle timings were not specified and so could change with design updates.

Bus protocol properties checked that the LFI obeyed the protocols of the FPI and LMB. Since the bus protocol properties were checking that certain behaviour always, or never, happened they were not sensitive to timing. This also meant that they could exercise the design further just by running for longer. Another advantage they have is that, like the input constraints, they are re-usable on any module with a similar interface (discussed later).

The transaction properties and protocol properties each found bugs. Exact numbers are given later in Section 4.2.

3.4 Combining the techniques

Directed Testing, Random Testing and Formal Verification are not completely orthogonal techniques. During both the LFI projects the random test bench was used to seed both property checking and directed testing.

When a failure occurs in directed or random testing, a specific set of circumstances have caused the observed failure, and it may be possible that there are other similar signal combinations which expose the same or similar errors. Thus when the bug is "fixed" and the test re-run, the bug may no longer be observed, but a similar or new bug may still be present in the code but it is reachable only via another set of input stimuli. Consequently it is useful to test a wider set of signal inputs using property checking. If the bug failure is carefully observed, a set of properties can be deduced which must be satisfied for the bug to occur, and thus we can write a property to prove that the bug cannot possibly occur. In this way we can prove that the bug has been fixed over all circumstances, rather than the one circumstance observed in the directed or random test. However, not all directed test bench

[1] Note that versions of GateProp subsequent to that used on the LFI include approximation of the reachable state space and analysis of unreachable counter-traces.

failures may be able to be analysed in the manner needed, so we may not be able to prove all bugs in all cases.

Thus, many failing tests were converted to properties. These properties were used to check the fix and often found new bugs or that the fix had introduced a new bug. This raises the question as to why such properties were not specified initially. The reason for this is that the set of all possible properties is too large – so using this approach helps to prioritise the properties to be written and run.

In addition, a random test may exercise a piece of code that has not been exercised by directed testing before, and it may prove to be a particularly problematic piece of code. The random test gives the verification engineer a set of input conditions which need to be applied to the DUT in order to reach that piece of code, so these can then be applied via the directed test bench to explore the behaviour of this previously untested code. This can free up the random test bench resource to explore and verify other areas of the DUT while the directed test bench is used to fix the bugs that cause the random test bench to fail every time it enters this area.

4. RESULTS

4.1 Coverage

The coverage achieved from the three approaches enabled us to release a DUT with a high degree of confidence in its design. Our directed testbench achieved the following structural coverage metrics:

- 100% Justified Statement Coverage
- 98.4% Branch Coverage
- 95.2% Condition Coverage
- 84.2% Trigger Coverage
- 100% Toggle Coverage

The random testing achieved 100% FSM transition coverage, but did not attain 100% cross-coverage of the FSM's. This proved to be impossible because of the design of the DUT. The bus transaction coverage targets were met (the target was 100% after removal of impossible transactions as discussed earlier).

Coverage for Property Checking is far harder to measure as GateProp has no coverage metric. A review of the properties by the verification team established that we were convinced that all the practical transactions and all protocol issues had been covered. Indeed it is possible to establish full protocol compliance via review of the properties, or better still mathematically (see Barrett and McIsaac (1997)).

4.2 Bug Analysis

4.2.1 Bug Graph

Bugs were recorded using an issue tracking system that allowed statistics to be gathered such as the cumulative number of bugs found, the number fixed, and whether they occurred in the design or test bench. Bug find rates are often expected to follow an S-curve shape, as the number of bugs found rapidly increases as the verification effort is increased, and gradually reduces as the DUT reaches maturity. Thus if the levelling out in the curve is not seen, it can be an indication that more time is needed to debug the design.

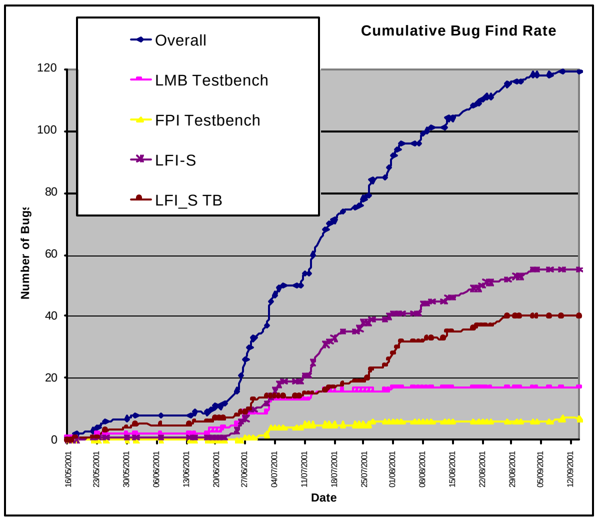

Figure 2 – LFI-S Bug Curves

Figure 2 shows the bug curve seen during the LFI-S project. It clearly shows the progression of the project through its life cycle. At the beginning of the project the test bench is constructed and debugged, then testing of the DUT begins and the bug curve rapidly moves upwards. Bugs are found in all parts of the design, and as new parts of the DUT functionality are tested, new areas of the verification IP are also tested, resulting in more test bench bugs. Eventually the test bench reaches maturity, and the DUT reaches maturity shortly afterwards. The cumulative total flattening out completely for a sufficient time can thus be used as a sign-off criteria, as it suggests that maturity has been reached.

4.2.2 What technique found what bug

Table 1 gives a breakdown of bugs found in the LFI-S by type, and by the method used to find them.

Bug Type	Prop.Chk	Direct.	Rnd.
Bus Protocol violations	11	4	8
Internal DUTqueue deallocation issues	0	2	4
Livelock/Deadlock	0	5	2
Other DUT issues	7	3	8
Total	18	14	22

Table 1 : Breakdown of LFI-S bugs

When reading these figures the reader is reminded that the various verification techniques were started at different times in the project. As stated at the start of the paper, for the LFI-S project the directed tests from LFI-IBC32 were used to prove initial functionality, and once basic functionality of the DUT had been demonstrated, random testing was started. However, property checking was started almost immediately, although time was required to modify the transaction properties due to changes in internal DUT timing.

In theory all of the bugs found by random testing could be found by directed testing. Random testing generates large sequences of transactions that can easily be replicated in the directed test bench. However, in practise there is not enough resource (to generate such large numbers of self-checking sequences) or ingenuity (to think of the corner cases that random generation creates).

Similarly, most of the bugs found by random testing could have been found using property checking given the resource to write an extensive set of properties (less ingenuity is required because property checking probes corner cases automatically). However, some bugs did require a number of cycles to cause the failure (e.g. those caused by filling up internal queues and then trying to add further transactions). The number of simulated cycles needed to get to this point can result in an excessively long run time for properties. For the same reason, property checking is unlikely to find the livelock or deadlock bugs, although if we create a livelock or deadlock by initial conditions we can prove that we cannot escape.

Property checking was most effective at demonstrating that the DUT does not violate bus protocols. Many of the violations would require a directed test to be written with a knowledge of what is happening cycle by cycle on the bus, and are generally found more by luck than judgement during random testing and than during directed testing.

Thus, during the two LFI projects, the verification strategies proved to be complementary – all stressed different areas of the DUT in different ways, and speeded the finding of bugs. They also allowed the verification manager to have several quantitative and qualitative measures on which to assess confidence in the quality of the DUT. Indeed, the following were all used in the sign-off criteria:

- directed and random test bench and test specifications reviewed and signed off by design and verification teams;
- structural and functional coverage targets met;
- all directed tests passing;
- a certain number of random transactions run successfully since last bug found;
- property specification reviewed and signed off by design and verification teams;
- all properties passing;
- bug curve flattened off and no bugs found for a given period of time.

4.3 Combined Effectiveness

Both the LFI-IBC32 and LFI-S have now been manufactured in silicon as part of larger SOC's and no bugs have reported against either.

5. RECOMMENDATIONS AND FUTURE USE

5.1 Directed testing vs. random testing

The combination of approaches improved the quality of the verification, but also increased the resources applied, especially as two test benches were built and maintained. One future course of action will be to become less reliant on directed testing (and thus more reliant on random testing and functional coverage) and to use the random test bench to perform the directed testing. However, this does result in compromise, as it requires the verification engineer to heavily constrain the random testbench and results in a loss of controllability compared to a pure directed testbench. This raises the question of whether to start with unconstrained random tests and constrain them to more directed tests over the project or vice-versa (Verisity recommend the former).

Also, without a large set of directed tests, a repeatable regression run of tests which can demonstrate DUT functionality becomes more problematic. Although the use of seeds in Specman allows repeatability, we require a list of seeds per scenario in order to guarantee full coverage. This also requires that the *e*VC's used are stable, and that the same version of Specman is also used each time the regression is run. If this requirement is not met, then the repeatability of the regression run cannot be guaranteed.

Finally, the structural coverage targets still need to be met and this requires Specman to be run with an appropriate structural coverage tool.

5.2 Productisation of verification IP

If users are to gain the benefits of verification IP reuse, then the productisation of the verification IP itself is very important.

The random test bench elements have been developed into re-usable IP (using the eVC approach recommended by Verisity). Indeed, these have been re-used successfully on a number of other projects with big reductions in test bench construction times

Some of the property checking deliverables have also been turned into re-usable verification IP. As already noted the property checking effort can be divided between the writing of constraints and the writing of properties. Each of these can be further divided. Constraints can be internal, i.e. constraining the internal signals of the design to avoid unreachable states, or external i.e. constraining the inputs to the design to reflect its environment. We've already divided the properties into bus protocol and transaction properties.

Clearly the internal constraints are not going to be re-usable, and may in fact be difficult to maintain across changes in a design during development. This was a major factor in our decision not to develop such constraints (and rely on reset to ensure properties used valid internal states). However the external constraints can be re-used, with appropriate modifications to signal names, on other modules that have a similar interface. The most commonly used interfaces are the busses, so constraints modeling busses give the largest opportunity for re-use.

Similarly the transaction properties will not be re-usable and may take some effort to maintain, due to small changes in timing. However the bus protocol properties are easily re-usable. A little care should be taken to monitor run times to ensure that the design is really being exercised, since some designs may have a ramp up time after reset where they do little and so are unlikely to break any protocols.

So it is the constraints on, and properties for, bus protocols that are re-usable. Indeed those developed for the LFI have been used on other modules that interface with the FPI and LMB busses, namely a peripheral control processor and an external bus unit. This experience of re-use will be used to develop bus protocol packages for these busses, using a simple wrapper to standardise signal names. These packages will be easy to use for someone with no experience of property checking. Of course such packages are applicable to any standardised protocol and there are also plans to develop them for other busses, such as the AHB.

5.3 Easier re-use between dynamic and static verification

During the LFI projects a set of constraints and properties were developed and maintained. There is much scope for re-use from random/directed testing to property checking and vice-versa. For example:

- The random test bench and property checking both use constraints to first define the legal sequences of inputs and then to further restrict them to a smaller set of sequences of interest to test or property in question.
- Properties should also hold during simulation.
- Dynamic verification can get the design into legal internal states from where it would be useful to start a property check (as opposed to always starting it from reset).

Fortunately, all three of these areas are being addressed in various ways. Accellera (www.accellera.org) are trying to define an industry standard formal language that can be used in both formal and dynamic verification. This should allow the first two to be realised. There are also tools appearing that enable the third (so-called semi-formal tools), such as 0-In Search.

6. REFERENCES

[1] Bergeron, J. Writing Testbenches; functional verification of HDL models (2000), Kluwer Academic Publishers

[2] Bormann, J. and Spalinger, C. (2001) Formale Verifikation fuer Nicht-Formalisten (Formal Verification for Non-Formalists*)*, Informationstechnik und Technische Informatik, volume 43, issue 1/2001, Oldenbourg Verlag

[3] Barrett, G. and McIsaac, A. (1997) Model Checking in a MicroProcessor Design Project*,* CAV'97 (Proceedings of the 9th International Conference on Computer-Aided Verification, Haifa, Israel 1997, Orna Grumberg (Ed.), Springer Lecture Notes in Computer Science 1254)