

A Feasibility Study in Formal Coverage Driven Test Generation

Abstract.

We claim that the verification crisis may be solved by bridging the gap between formal verification and simulation. This paper presents a study of a functional verification methodology using the coverage of formal models to specify tests. This methodology was applied to a modern superscalar microprocessor and the resulting tests were compared to tests generated using existing methods. The results show that hybrid techniques can indeed improve functional verification.

1. Introduction

There is now widespread acceptance in the EDA community that the resources devoted to functional verification are increasing disproportionately as designs become more complex resulting in a “verification bottleneck”. Traditional techniques based on **simulation** can only verify operation over a small subset of the state space and it is often hard to generate tests which hit interesting corner cases. A more recent approach uses **model checking**, but this does not scale well due to state space explosion. While the capabilities of formal verification will improve over time, the need for a more immediate practical solution has resulted in increasing interest in the possibility of combining formal methods with simulation based testing[1,2,3,6].

The work described in this paper had two objectives. The first was to develop a verification methodology that bridged the gap between formal verification and simulation in a way that would integrate into an existing design flow. The second was to make a quantitative comparison with our existing simulation based verification techniques.

This paper starts by introducing coverage driven test generation. It then describes our methodology and introduces GOTCHA, a prototype coverage driven test generator implemented as an extension to the Mur ϕ model checker. The second part describes a feasibility study where we applied this technology to the verification of the decoder of a modern superscalar microprocessor. By tracing the tests on a VHDL simulation of the microprocessor we were able to measure actual state and transition coverage. We compared these statistics with those of tests previously used to verify the decoder prior to tape-out.

We are able to conclude that coverage driven test generation can improve functional verification. During the feasibility study we demonstrated some 50% improvement in transition coverage with less than a third the number of test instructions. In addition the methodology has the twin advantages of explicitly capturing verification knowledge within a model and giving quantifiable results.

Bridging the Gap: Model Based Verification

2. Coverage driven test specification

Coverage is a measure of the completeness of a set of tests. Each measurable action is called a coverage task, and together these form a coverage model. This may be quite simple such as measuring which lines of code have been executed or how many state bits have been toggled. The usefulness of coverage depends on how well the chosen coverage model reflects the aims of the testing program. In the case of functional verification, state and transition coverage are very powerful measures because of their close correspondence to the behaviour of a design.

It is also possible to measure coverage not only on a design but also on a model of the design such as a functional simulation. Given a description of a design, either the actual implementation or a model such as a specification, it is possible to reverse the above process and use coverage as a means of specifying tests. Each coverage task can be viewed as defining a set of test cases which satisfy that task. In the case of transition coverage there is a coverage task for each possible transition which defines the set of tests that pass through that transition.

Using a finite state machine model of a design and a coverage model, it is possible to automate test specification. A **coverage driven test generator** finds paths through the finite state machine model and its goal is to find a path satisfying each task in the model. These paths are called abstract test cases. By contrast we define a concrete test case to be a simulation trace of the hardware modelled by the finite state machine. A test specification is an intermediate form between abstract and concrete which can be used to drive a test generator.

The selection of a coverage model is a pragmatic choice depending on the resources available for test generation and simulation. We say that one coverage model is stronger than or implies another coverage model if every test suite which achieves 100% coverage of one model also achieves 100% coverage of the other. In this hierarchy of coverage models it is clear for example that path coverage is stronger than transition coverage which is in turn stronger than state coverage. However coverage models may also be incomparable, in the sense that neither one is stronger than the other.

The coverage efficiency of two test suites may also be compared. The most useful measure is probably the amount of simulation resources required, for example the total length of all the test cases.

Some classes of coverage models are of particular interest, since they arise in a natural way in hardware verification based on a finite state machine model of the micro-architecture. In the context of hardware design, this corresponds to choosing certain of the signals, latches, and flip-flops as characterizing the state of an implementation, and ignoring the rest. We refer to these variables as the coverage variables.

3. The methodology

Previous work has used the actual design as the model [1], or else a very highly controlled simulation environment[2]. We sought a more flexible solution integrated into our existing design flow and producing tests that could be run on a variety of platforms from software simulations to real silicon. This led us towards a methodology that could accommodate imprecision, change, and even errors. The main stages of the methodology are illustrated in figure 1.

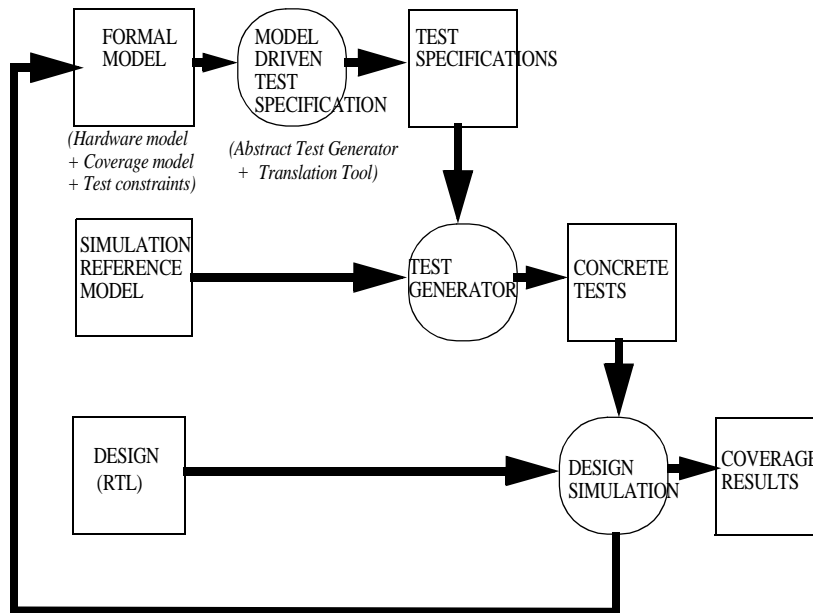


Figure 1: The main stages of the methodology

Abstract test specifications are derived from a formal model using a coverage model to specify tests that exercise all interesting corner cases. This is a difficult procedure to perform manually. Using the correspondence between the micro-architectural events at the boundary of the unit under test and architectural events it is possible to translate these traces into test specifications. The translation from abstract traces to concrete tests can be difficult, however in practice it is often simplified by the architectural structure inherent in the partitioning of the design, and the freedom to model the environment in ways that assist the process.

Test generation uses a separate architectural model to define the expected results. The coverage measurements are done on yet another model, the RTL of the design, using an architectural simulator - the golden reference for verification. This means that neither modelling errors, abstractions, nor over naive test translation can result in invalid tests, however some tests may fail to exercise the intended coverage task.

Information on the expected micro-architectural events in the unit under test can be extracted during coverage directed test generation and used to measure how well the actual tests are meeting their intended coverage objectives. This information can be used as feedback to generate additional tests to replace those that do not satisfy their original specification. It can also be used to identify errors in the model. Thus for example if the actual design is hitting states or transitions not covered by the model this suggests the model is over constrained, while conversely a failure to reach all the specified conditions may indicate either an error in the model or a weakness in the translation process.

4. The GOTCHA tool

GOTCHA (**G**enerator of **T**est **C**ases for **H**ardware **A**rchitectures) is a prototype coverage driven test generator which supports state and transition coverage. It is based on the Mur ϕ verification system[7] - an explicit state model checker from Stanford University. The resulting tool can still be used as a model checker it also incorporates language extensions and new algorithms to support the generation of abstract test specifications using a state or transition coverage model.

The Mur ϕ definition language is a high level language for describing finite state machines with the use of abstract data types, sequential code, function and procedure calls. This is extended to allow the modeler to designate coverage variables and characterize final states when modelling the unit under test, and the resulting language is called the GOTCHA Definition Language (GDL).

The GOTCHA compiler builds a C++ file containing both the test generation algorithm and the embodiment of the finite state machine. The finite state machine is explored via a depth first search or a breadth first search from each of the start states. On completion of the enumeration of the entire reachable state space, a random coverage task is chosen from amongst those that have not yet been covered or proved to be uncoverable. A test is generated by constructing an execution path to the coverage task (state or transition) then continuing on to a final state. There is also some degree of randomization of the path chosen and the tool incorporates two algorithms for test extension, the first is extension by a random walk, and the second uses a breadth first search of the state space to find "nearby" uncovered tasks. A task is deemed uncoverable if no path which exercises the task has an extension to any designated final state.

The Feasibility Study

5. Motivation

The purpose of the feasibility study was two fold. Its first objective was to develop a verification methodology that bridged the gap between formal verification and simulation in a way that would integrate into an existing design flow. The second objective was to gather some quantitative evidence to support the idea that such a methodology would offer advantages over our existing simulation based verification techniques.

The feasibility study targeted the decoder of a 64-bit superscalar processor designed by STMicroelectronics[4]. This selection reflected the importance we place on the functional verification of leading edge processors, however the methodology is applicable to most designs. The choice of the decoder made it possible to investigate the problems of modelling and test specification while simplifying other tasks, in particular the translation from abstract to concrete tests.

6. Model development

The intention of the model was to capture the functional implementation of the decoder control logic (Figure 2). This contains a number of identifiable, but not necessarily independent mechanisms, that can cause the types of hazards that result in implementation errors. For example limited branch prediction that can result in pipe stalls and certain instructions which are expanded into multiple micro-instructions which may only be accepted as the first of a pair of superscalar instructions.

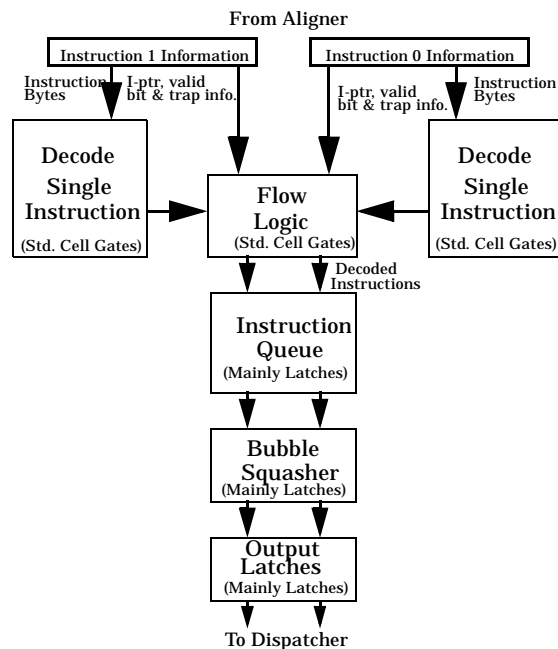


Figure 2: The basic blocks of the Decoder

The modeling of the internal logic of the decoder was done by a manual - but quite mechanical - translation of the VHDL into GDL.

The model of the environment of the decoder has the dual role of ensuring a good mapping from micro-architectural events to architectural events and also constraining the behaviour of the unit under test. If the model is under constrained then it may specify unimplementable tests. Conversely if it is over-restrictive it may be unable to reach all the states or transitions that are reachable in the real design.

Some abstractions were used that simplified the model while still giving good results. These included:

- **Temporal abstractions:** In general we were more concerned with the order of events than their precise timing.
- **Data abstraction:** Rather than modelling the entire instruction set it was possible to simply define classes of instructions.
- **Partitioning:** It is possible to simplify the model by separating functions. For example a model describing the decoder without a bubble squasher.

The most difficult thing to model was the surrounding environment, and specifically, the timing of external events such as interrupts, feedback, or memory accesses.

The methodology does not require that the environment model be totally accurate or complete, hence it was possible to make a number of simplifications. In particular:

- External events were not required as other architectural events could cover the same conditions in a more controllable fashion.
- The timing of feedback loops was measured for arbitrary architectural tests and typical values chosen for the model.
- The timing of memory accesses was made largely deterministic by using hot caches, where the processor caches are preloaded with the test program and data.

7. The Experiment

Test specifications were generated using the model as input to the GOTCHA tool. Both state and transition coverage models were used to generate two different test suites. The coverage variables were chosen to cover all the internal control state relevant to the control mechanisms.

The abstract test suites produced by GOTCHA were translated by a straightforward substitution algorithm to produce test specifications for input to an existing test generator. Each test specification was run repeatedly through the test generator till a valid concrete test was obtained. If the model permitted the specification of architecturally invalid tests this could show up at this stage and be used to correct the model. We observed this with an early GOTCHA model that permitted multiple leaps without changing the jump address, behaviour which is architecturally invalid. Any test generator will produce useful tests provided it correctly implements the test specification, however a sophisticated test generator can provide additional biasing independent of the test specification, for example biasing the data values for the instructions.

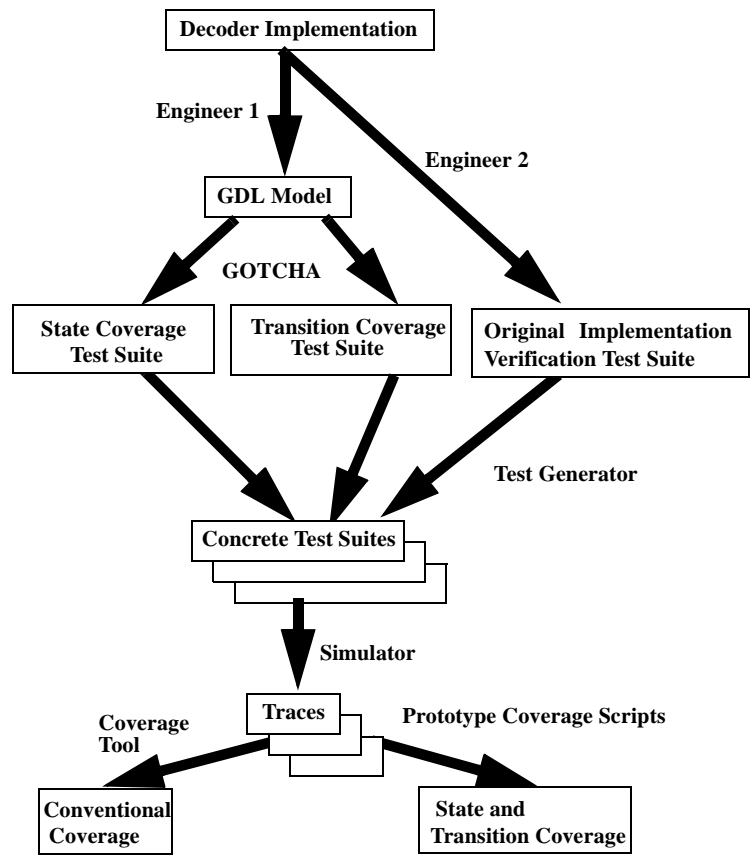


Figure 3: The flow of the experiment

These tests, and a control test suite were simulated on the complete processor design using a VHDL simulator and trace information was gathered and compared to the original trace specifications. In the specification, the behaviour of the unit under test defines the coverage objective. In the test simulation, the behaviour is used to measure the coverage achieved. This is the basis for making the methodology quantitative and providing feedback to enhance coverage. The test results are measured on the actual design, and it is possible to observe not only the behaviour of the unit under test, but also the full architectural and micro-architectural state, making it possible to observe unpredicted side effects.

Trace comparison between the specification and the simulation at the interface between the unit under test and its environment was used to identify errors in the model, and could equally well be used to localise design errors. In principle this could also provide feedback to correct timing errors in a test and hence improve coverage.

In order to provide a reference point for evaluating the results we took the tests used to verify the decoder implementation prior to tape-out. The specifications were developed by a verification engineer with the aim of identifying likely errors in the design. These were then used to generate a family of concrete tests using an advanced model based test generator called Genesys[4,8]. In an attempt to improve this process several tests were generated from each test specification giving a shotgun effect. Additional tests were generated by biasing the generator towards significant classes of instructions. Coverage measurements were then used to write new families of test-cases to improve testing the parts of the design that did not achieve 100% of reachable statement or toggle coverage. This process resulted in 93 different test specifications specifically aimed at exercising the decoder which in turn produced a database consisting of 321 concrete tests. It was this database which we used to provide comparisons with the results produced by GOTCHA.

The flow of the experiment is illustrated in Figure 3.

8. Results

Table 1 compares the size of the various test sets, and it is clear that the GOTCHA test suites consume far fewer resources than the decoder’s implementation verification test suite. Human resources were not readily comparable given the experimental nature of the feasibility study.

Test Suite	Tests	Simulated Instructions	Abstract Instructions
Implementation verification	321	269941	200000
State coverage	30	17645	360
Transition coverage	139	85278	1560

Table 1: Tests size

Several coverage measures were applied to the tests. All test suites achieved complete coverage of the decoder according to line and branch coverage metrics. The exception was multiple sub-condition coverage where none of the test sets performed well. This is because many of the coverage tasks in the model are unreachable in the design.

The verification engineers observed that the most common coverage metrics, such as statement and branch coverage are weak and do not provide a meaningful measure of quality. Other coverage models are required to differentiate quantitatively between the test suites. In Table 2 we give the state and transition coverage measurements using the

state machine based on the coverage variables used by GOTCHA in generating the tests and also the coverage of pairs of input instructions.

Test suite	States	State Coverage	Transitions	Transition Coverage	Inst. Pairs	Inst. Pairs Coverage
Implementation verification	40	74.07%	159	53.9%	49	40.50%
State coverage	45	83.33%			33	27.27%
Transition coverage	47	87.04%	236	80.00%	45	37.19%
Coverage tasks	54	100%	295	100.00%	121	100.00%

Table 2: Coverage results

While the headline percentage for instruction pair coverage is highest for the decoder implementation verification tests an investigation revealed that this was because they included several functionally equivalent cases where the second instruction was never accepted by the decoder. The GOTCHA transition coverage tests avoided this redundancy.

The cumulative coverage graphs (figures 4 and 5) show that the GOTCHA test cases required fewer instructions to achieve a given level of state or transition coverage and this coverage continued to improve while the implementation verification tests saturated. This was especially true in the case of transition coverage, which was the strongest coverage metric used to compare the tests.

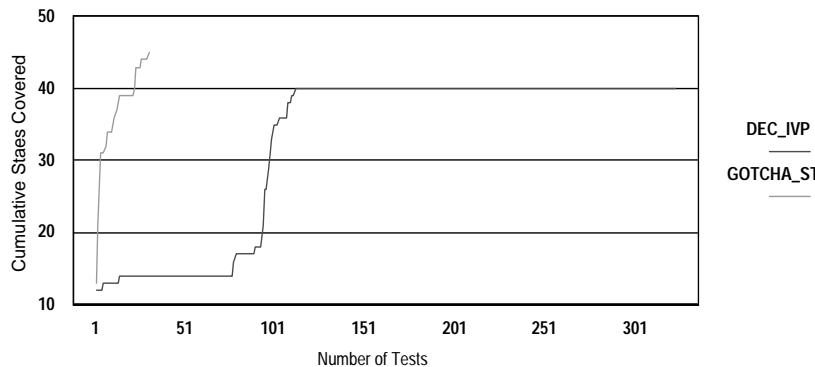


Figure 4: State Coverage

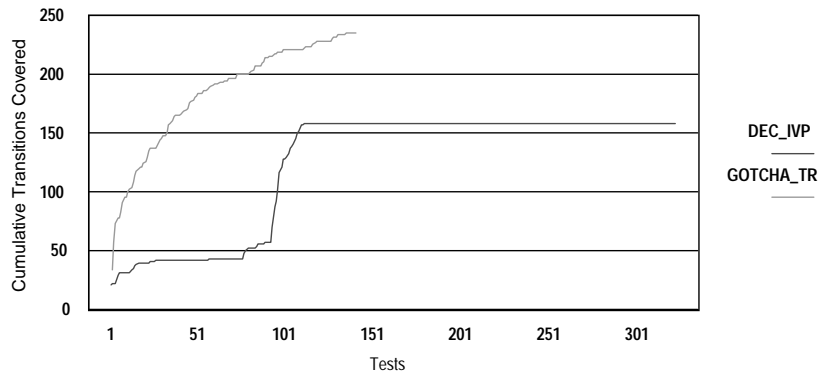


Figure 5: Transition Coverage

We also observe an interesting rise in the coverage generated by tests 100-105 in the implementation verification test suite. These were random tests generated from a specification that defined interesting groups of instructions, rather than specific scenarios. They were intended to try and cover cases not predicted by the verification engineer. It is also apparent that the strategies used in writing the decoder implementation verification tests failed to improve the coverage after a certain point. Prior to measuring the state and transition coverage there was no meaningful quantitative information on the relative quality of these tests.

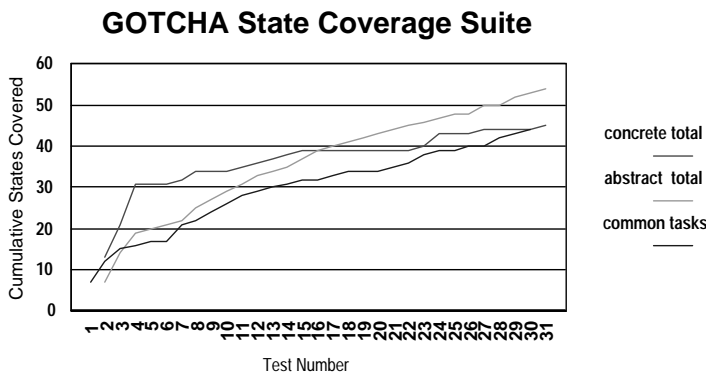


Figure 6: Abstract Versus Concrete Coverage

As illustrated in figure [6] the GOTCHA tests did not reach 100% state coverage. The failure of the tests to fully implement the specification is attributed to abstractions in the model and to the naive test generation process. The model of the environment gave feedback at the first legitimate time period whereas this feedback could be at any later time cycle in the actual design. This explains the difference between tasks covered by

the simulation of concrete tests and tasks covered in the abstract tests generated by GOTCHA. These limitations could largely be overcome by introducing feedback into the methodology.

9. Conclusions

The results in the previous section show that coverage driven test generation can improve functional verification. During the feasibility study we demonstrated some 50% improvement in transition coverage with less than a third the number of test instructions. In addition the methodology has the twin advantages of explicitly capturing verification knowledge within a model and giving quantifiable results.

Combining formal methods, functional simulation, and coverage offers definite advantages over simple testing or model checking:

- The use of a model makes it possible to automatically identify corner cases and specify corresponding tests.
- The formal model is only used to identify test cases, not predict the test results, hence it is safe to tackle the problem of state explosion with potentially unsound simplifications and abstractions.
- The test simulations can detect unexpected errors and side effects because they compare architectural, and potentially also micro-architectural state, for the entire design.

An important lesson of the feasibility study was the potential benefits to be gained by using feedback from simulations to guide the test generation process. This can compensate for excessive abstraction in the formal model and a naive translation process from abstract test specifications to concrete tests. Feedback is also an essential part of the iterative process of refining a model.

Using quantitative coverage models provides an objective basis for evaluating test sets. Without this information it is difficult to develop an effective verification plan, or conversely to set objective standards for key design decisions such as tape-out. While existing coverage tools can provide quite powerful metrics such as multiple sub-condition coverage these are of limited use given their inability to distinguish between reachable and unreachable coverage tasks.

The choice of coverage model is still subjective. Weak coverage models such as line coverage can make it impossible to distinguish between good and poor test sets. State and transition coverage would seem to be especially suitable as the basis for coverage models because of their close correspondence to the functionality of the design. In general even if one does not achieve 100% on an aggressive coverage metric it will push the resulting tests towards giving good coverage on weaker metrics.

REFERENCES

- [1] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. *Architecture validation for processors*. In International Symposium of Computer Architecture 1995, pages 404–413, 1995.
- [2] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal, *Coverage directed test generation using symbolic techniques*. In FMCAD 96, Nov. 1996, Palo Alto.
- [3] D.L.Dill, *What's Between Simulation and Formal Verification?*. In DAC 98: 35th Design Automation Conference, June 1999, San Francisco.
- [4] F.Casaubieilh, A.McIsaac, M.Benjamin, M.Bartley, F.Pogodolla, F.Rocheteau, M.Belhadj, J.Eggleton, G.Mas, G.Barrett, C.Berthet, *Functional Verification Methodology of Chameleon Processor*. In DAC 96: 33rd Design Automation Conference, June 1996, Las Vegas.
- [5] *Fesibility Study Report*. ESPRIT Project 25314: GENEVIEVE, V1.0, June 1998.
- [6] *0-In Design Automation*, <http://www.0-In.com/>
- [7] *Murφ*: <http://sprout.stanford.edu/dill/murphi.html> A public domain model checker available from Stanford University.
- [8] *GeneSys* An automatic test program generator for uni-processor designs developed at the IBM Haifa Research Laboratory. See A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. *Test program generation for functional verification of PowerPC processors in IBM*. In 32nd Design Automation Conference, DAC 95, pages 279–285, 1995.

Acknowledgments.

We would like to acknowledge the support of the European Union for this work through the Esprit programme.

