

# How to Get Your REALLY Difficult Properties Proven

Thomas J. Thatcher

Sun Microsystems

thomas.thatcher@sun.com

## ABSTRACT

In our experience with formal verification, we have run into some properties that were very difficult to prove. When running with difficult properties, it can be difficult to see if you are making progress or not. Because some of these properties are critical to the success of our design, we have invested a lot of effort and tried several different ways to get them proven. This paper describes what we have learned. We will describe one particularly difficult property we ran into. We'll describe how we originally set up the property. We'll discuss several of the techniques we used, including Magellan setup tricks, the use of coverage goals, some divide-and-conquer approaches, and assume-guarantee proofs. We'll conclude with some results, including the number of bugs we found in the design.

## Table of Contents

1.0	Introduction . . . . .	3
1.1	High-Level View of Formal Verification . . . . .	3
1.2	Property Checking at Sun . . . . .	3
2.0	Property to Be Proven . . . . .	4
2.1	Verification Challenges . . . . .	4
3.0	Setting up the Formal Proof . . . . .	4
3.1	Writing constraints. . . . .	4
3.2	Use of Coverage Goals . . . . .	6
3.3	Use of Multiple Sessions for Different Goals . . . . .	7
4.0	Running the Tool . . . . .	7
4.1	Iteratively Adding Constraints. . . . .	7
4.2	Continuing while Waiting for a Bug Fix. . . . .	7
4.3	Verifying a Bug Fix. . . . .	7
4.4	Run Time . . . . .	8
5.0	Special Treatment for Difficult Bugs. . . . .	8
5.1	Cone-of-Influence Report . . . . .	8
5.2	Breaking up the Property. . . . .	8
5.3	Assume-guarantee Proofs . . . . .	9
5.4	Bounded Proof Mode. . . . .	10
6.0	Results . . . . .	10
6.1	Properties Proven. . . . .	10
6.2	Bugs Found . . . . .	11
7.0	Conclusions and Recommendations . . . . .	11
8.0	Acknowledgments . . . . .	11
9.0	References . . . . .	11

## Table of Figures

Figure 1.	Proof Setup for trap unit, showing pipeline model . . . . .	6
Figure 2.	Internal structure of the TLU block . . . . .	9
Figure 3.	Setup for first half of assume-guarantee proof. . . . .	10
Figure 4.	Setup for second half of assume-guarantee proof . . . . .	10

## 1.0 Introduction

Formal verification tools have proven to be very useful at verifying complex behavior and finding deep corner-case bugs. One of the formal verification tools on the market is Magellan, from Synopsys. This tool combines a formal property checking engine with a constrained random simulation engine to provide very fast bug detection along with the formal proof. It is currently being used within Sun to attack some of our most difficult properties. This paper describes the use of Magellan for formal property checking at Sun. We'll describe one particularly difficult property we worked on, then describe the methodology we developed to attack this property and other similar properties. The paper will conclude by looking at some of the results we achieved using formal property checking.

### 1.1 High-Level View of Formal Verification

The goal of any verification effort is to insure that the design functions properly under the complete set of legal stimulus. In hardware design, the most common methodology used for functional verification is simulation. Tests for the design are written, then the design is evaluated with the stimulus generated. The problem is that the set of tests may not cover the complete set of legal stimulus. Methods such as code coverage and functional coverage have evolved to give some feedback on the completeness of the test suite. Still if the functional coverage description is not complete, then the test set may not be complete.

Formal verification offers an answer to these problems, but it does have some of the same problems. Formal verification tools do a mathematical proof that the design satisfies certain properties. This mathematical proof is analogous to verifying the design over all possible stimulus. However, in most cases, constraints need to be applied, because the properties do not hold under illegal stimulus. Each constraint removes a set of illegal stimulus, but it may remove a piece of legal stimulus as well. In this occurs, the properties are all proven, and nobody is aware that the properties were proved on an over-constrained design. As a result, additional methods are needed to evaluate the completeness of a proof, just as they are needed to evaluate completeness of a simulation-based regression. This does not mean that formal verification has no value. One big value of formal verification is that writing constraints forces the designer to think about illegal stimulus. Often, questions about whether a set of stimulus is legal or illegal lead to the uncovering of a bug. This makes formal verification complementary to the traditional simulation-based verification methodology, allowing us to find bugs that would not be uncovered any other way.

### 1.2 Property Checking at Sun

At Sun Microsystems, there have been three usage models for formal verification. The first approach is a complete verification of a block using formal verification. A test plan for the block is written, and a complete set of proofs is developed to verify everything in that test plan. This type of verification may be used in the place of a stand-alone test environment. The second approach is to target certain critical properties of the design, which could cause a fatal bug if they were not satisfied. Formal proofs are run on these critical properties. This approach is used to maximize the return on the formal verification effort when resources are limited. Finally, formal verification has been used in post-silicon debug to quickly identify failures seen on the bench when the failures cannot be reproduced in simulation. This is quite useful when the failing scenario implicates a certain block in the design. Properties are written on the suspected block

asserting that the erroneous behavior observed on the bench is impossible. The formal tool will then generate an example of the corner case where that behavior can occur.

## **2.0 Property to Be Proven**

One particularly difficult property we encountered involved the trap unit on a new microprocessor design which is currently in development. This trap logic unit (TLU) is responsible for receiving error and interrupt signals coming in from all over the chip. All these signals must be prioritized in order to determine which trap should be taken. When the proper trap has been determined, the TLU flushes instructions remaining in the pipeline and redirects the program counter to the address pointed to by the proper entry in the trap table.

Inside this block are approximately 64 signals, each representing a trap to be taken. We needed to assure that it was impossible for two or more of these signals to be asserted at the same time. If this were to occur, the microprocessor could possibly take an incorrect trap. A failure of this property would have very serious consequences in the design, and needed to be avoided at all costs. That is why this property was targeted.

### **2.1 Verification Challenges**

While the property to be proven was very easy to write, the setup of the proof was much more difficult. The one-hot property was not guaranteed by the block. It relied on many different assumptions about the processor architecture. All of these assumptions had to be implemented as constraints to the design. The block did contain some prioritization logic to prioritize different exception signals, but it was not complete. If there was an architectural reason why two exception signals could never occur on the same cycle, then there was typically no logic to prioritize those two signals. Thus a constraint would need to be written to prevent the formal tool from generating this case.

The constraints that needed to be written were often very complex. Often, the constraint required understanding of the timing of other blocks within the CPU. In many cases it was very challenging to express the constraints using Open Vera Assertions (OVA), the property description language we used for this project.

Another complication to this block was the fact that there was no clean interface between any of the sub-blocks within this block. That meant that most of the proofs needed to be run at the block top level. The lack of clean interfaces made it very difficult to divide up the block and verify the components separately.

## **3.0 Setting up the Formal Proof**

### **3.1 Writing constraints**

When verifying a block with very complicated interfaces, there are two main approaches to writing the constraints. The first approach is to specify the constraints using properties. The second is to write transactor modules (also called stub modules) in Verilog to generate proper stimulus on the block. Constraining the design with properties has the following advantages:

1. Constraint Properties can be used later on as target properties to be proven on another block

2. When a proof is complete, it is very easy to link these properties in to the regression simulations. Then regressions can be run, and any firings of the constraint properties can be double-checked to insure that the constraint is correct. It would be very difficult to verify a stub module in this manner.

Constraint properties have a number of disadvantages, however.

1. Constraint properties may be prone to dead-end states. This typically occurs when the stimulus generator is not able to generate new stimulus for the design which meets all the constraints. This is typically not a problem with stub modules. Stub modules are written in procedural code, and are easier to write for complicated interfaces.
2. Constraints may be very difficult to write using properties when interfaces are very complicated. Writing them using procedural Verilog is usually more straightforward.

### 3.1.1 Avoiding Dead-End States

When OVA or SVA properties are used as constraints, there is the possibility that Magellan will run into dead-end states. This has discouraged many people from writing their constraints this way. However, there are some easy rules which can greatly reduce the risk of running into dead-end states. First, make sure all properties progress forward in time. The first code example below shows a violation of this rule.

```
event hit_only_on_valid :
    if (tag_hit) then past(valid,2);
```

This property attempts to retroactively constrain the signal valid two cycles into the past. While some formal tools would be able to handle this, the stimulus generator for the random simulation can not. If it sets tag\_hit to a 1, and signal valid was not 1 two cycles earlier, it will hit a dead-end state. This problem is easily avoided by re-writing the property as shown below.

```
event hit_only_on_valid :
    if (~valid) then #2 ~tag_hit;
```

A second rule to follow is to try not to constrain a signal with multiple properties. If multiple properties constrain one signal, then all properties must either constrain the signal to the same value, or they must have mutually exclusive conditions on the constraint. Here is another example

```
event not_valid_if_nomatch :
    if (addr != 4) then #1 ~valid;
event valid_if_req :
    if (req) then #1 valid;
```

In this example, Magellan will hit a dead-end state when signal req is one and addr is not equal to four. This can be corrected by either combining the two properties into one, or by modifying the if portion of one of the properties to make the two conditions mutually exclusive. Here's an example where the second property is modified to prevent the dead-end state:

```
event not_valid_if_nomatch :
    if (addr != 4) then #1 ~valid;
event valid_if_req :
    if (addr==4 && req) then #1 valid;
```

### 3.1.2 The Best of Both Worlds

In the case of the trap unit, we started out writing all the constraints using properties. However, the constraints grew so complicated, that something needed to be done to simplify things. That's when we decided that a combination of transactor modules and property constraints would work better. For the transactor module, we wrote a procedural Verilog model of the microprocessor pipeline, and then used the state information in that pipeline model to write constraints for the trap unit. A block diagram of this setup is shown in Figure 1.

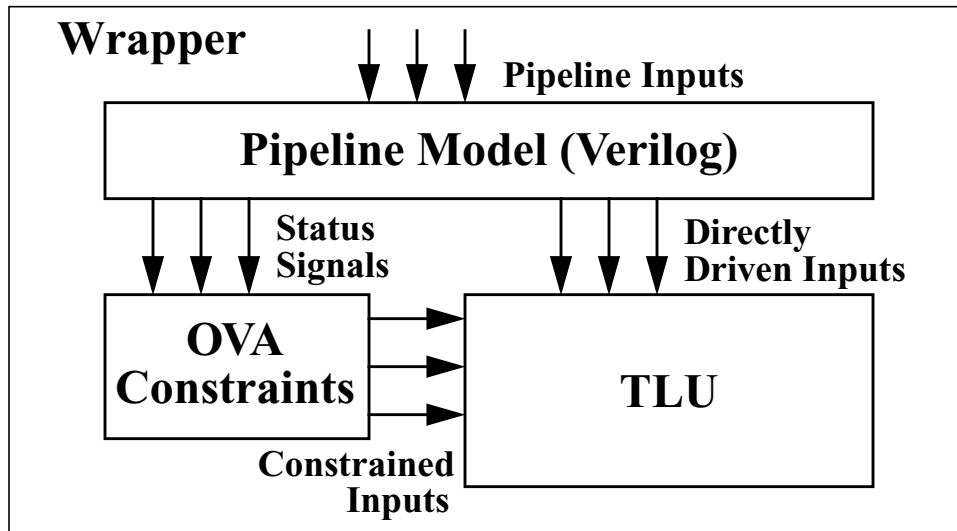


Figure 1. Proof Setup for trap unit, showing pipeline model

The pipeline model simply tracked “instructions” through the last several stages of the microprocessor pipeline. The “instruction” was an arbitrary 8-bit encoding which specified the type of instruction, and possibly the result (cache hit, cache miss, etc.). Coding the pipeline model allowed for extra randomized inputs to the design.

### 3.2 Use of Coverage Goals

One of the concerns with formal verification is knowing if the design is over-constrained or not. If the design is over-constrained, the result is often a true proof. A way is needed to evaluate whether this true proof is valid, or if it is the result of an over-constrained design. This was of particular concern in the TLU design, because of the large number of constraints needed to prove our property. To gain more confidence in the proof, we started coding coverage goals for the proof. These goals were implemented as OVA cover statements. First, a coverage point was created for each different trap signal that was part of the one-hot proof. Then coverage points were written for most significant inputs to the design. In all we wrote over 350 total coverage points. We required that all coverage points must be covered. The inability of the tool to cover a given point usually pointed to some incorrect constraint that was over-constraining the design.

In Magellan, these coverage points can be specified as goals for the formal engine. Magellan will try to hit these coverage goals, or will try to prove that they are not coverable. When the tool session completes, a report is generated showing which coverage points were covered, and which

ones were proven non-coverable. With Magellan, the coverage points have another advantage. In attempting to cover these goals, the tool may be steered toward a falsification of a target property. In one example, a session run with coverage points and properties hit a falsification within ten minutes, while a session run with the target properties only required over an hour to hit the falsification.

### **3.3 Use of Multiple Sessions for Different Goals**

In setting up the Magellan project file, we found it necessary to set up multiple sessions to achieve different goals. Early on, the goal was to get to a falsification quickly. As a result, we set up a session with properties and coverage goals as targets for the formal engine. We found that including coverage goals often steered the tool toward a falsification much faster. Later, when the focus shifted to trying to achieve a proof, we ran a session with just the property. Finally, to double-check the achievable coverage from the proof, we had a session with only coverage goals. We found that when the property is not proven or falsified within a reasonable amount of time, the tool would not spend much time on coverage goals, and they would remain uncovered. Therefore we ran the coverage-only session to rapidly evaluate the coverage that was achievable with the current set of constraints. This third session may not be necessary in the future, as the engine orchestration within Magellan may change.

## **4.0 Running the Tool**

### **4.1 Iteratively Adding Constraints**

We began running the proof with a minimum of constraints. As we analyzed each falsification, more constraints were added, usually after a conversation with the block designer. There are a couple advantages of starting with a minimum number of constraints. First, constraints can actually make the proof more complex. Many times we have added a constraint to the design, thinking that we were simplifying things, only to see the proof time go up dramatically. Second, the more constraints there are, the more likely that an error in the constraint code will cause the design to be over-constrained.

### **4.2 Continuing while Waiting for a Bug Fix**

When a bug is encountered in the design, progress can stall until the bug is fixed. However, by adding temporary constraints, the verification engineer can direct the tool to avoid the first bug, and hunt for more until the first bug is fixed. We did not use this approach extensively, because the bug turn-around time was fairly fast.

### **4.3 Verifying a Bug Fix**

When a bug is fixed, often the result of running formal verification is that a different bug is found. If the property does not pass, it's difficult to tell if the original bug is fixed, or if the tool simply found a different bug. This was especially true in our case, where a single one-hot property covered the entire design, and many different failure cases. However, by writing temporary properties which target the original bug exactly, the bug fix can be verified. The approach may also be used for duplicating bugs found by simulation. We did need to do this in a few cases when simulation regressions discovered a bug not caught by the formal tool. We tracked the problem

down to an error in the constraints that prevented the tool from hitting this bug. Once the constraints were fixed, we wrote a specific property to target the bug found in simulation, and verified that we were able to hit it with Magellan.

#### **4.4 Run Time**

Typically, we did not let Magellan jobs run more than a week. Our observation was that we rarely saw any additional output from the tool after 48 hours, so it was very difficult to quantify the value in running any further. If the tool did print additional status into the log file, it might be easier to decide whether or not to continue the run. A few experimental runs that were allowed to run for two weeks did not yield any additional insight into the design.

### **5.0 Special Treatment for Difficult Bugs**

When a property is not proven by the tool, even after a very long run time, special treatment will likely be needed to get the properties proven. Here are a few techniques that can be used to improve the chances of finding a proof, or to figure out that a proof is not feasible on the design.

#### **5.1 Cone-of-Influence Report**

Magellan has a cone-of-influence report. This prints a list of flip-flops and primary inputs that are in the cone of influence of the target properties in the session. This report has two primary uses. First, the report serves as a quick way of determining the feasibility of finding a proof. The report lists the total number of flip-flops in the cone of influence. If this number is very large, there will probably be a very small chance of finding a proof. One rule of thumb is that there should be fewer than 400 flip-flops in the cone of influence if there is to be a good chance of finding a proof. The cone of influence for our TLU proof was around 5800 flip-flops. In retrospect, this was a very good indicator that a proof was not going to be attained at the top level of the block. It's still useful to run the tool, however. In the future, we can simply plan to run the tool as a bug-finding engine, without expecting to achieve a proof, as we have seen others do[1]. It's also important to not treat this rule as an absolute. We have seen proofs achieved on much larger cones, and we have seen proofs fail to converge on smaller cones as well.

The second use for the cone-of-influence report is to analyze the flip-flops that are present. Are there flip-flops that are not expected in the cone-of-influence? If so, why? Answering these questions may lead to ways to simplify constraints.

#### **5.2 Breaking up the Property**

One approach is to break up the property into smaller properties to be proved. A one-hot property, for example, may be decomposed into many 2-bit one-hot properties. We tried doing this manually with the TLU proof. However, in this particular design, it didn't seem that a 2-bit one-hot property was much easier than the original 64-bit one-hot properties. A cone-of-influence report on the original property contained around 5800 flip-flops. Running a 2-bit one-hot property resulted in a cone of influence containing 5200 flip-flops. This indicated that this approach was not very promising for achieving a proof. We did let several 2-bit examples run for a week, but were not able to prove any of them.

### 5.3 Assume-guarantee Proofs

When the proof for this one-hot property was set up, we ran it at the block top level. This is because the designer felt that it would be difficult to specify properties on the interfaces of the sub-blocks in the design. However, since Magellan was not able to converge on the proof at this level, we decided to try to run the proof at a lower level in the design. There were two primary sub-blocks that were important to this proof. A set of properties to describe the interface between these two sub-blocks was needed. These interface properties needed to be proved on the first sub-block using the existing constraints from the top level, then used as constraints to prove the target one-hot property on the second sub-block. The setup of this two-step proof is described below.

A simplified structure of the TLU is shown in Figure 2. There are two primary sub-blocks which are important to the proof: Exception signals all go the first block, where some of the prioritization occurs. Then trap requests are passed over to the second sub block, which does the final prioritization and arbitration. The 64 trap enable signals are then combined together to create a 9-bit trap type.

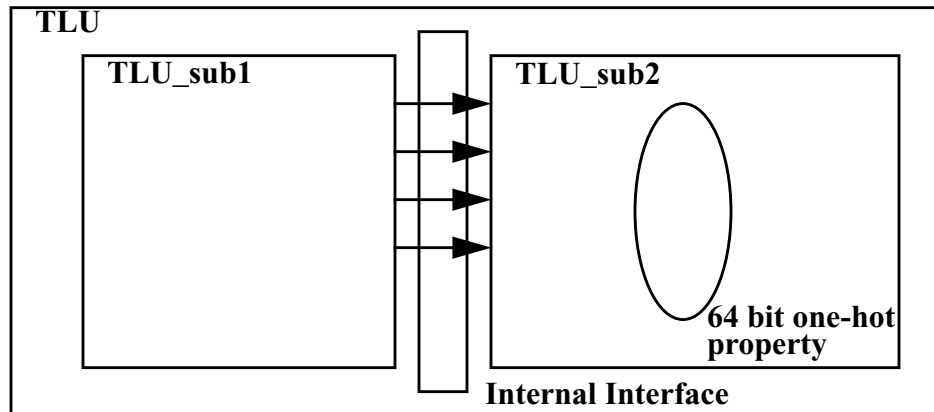


Figure 2. Internal structure of the TLU block

First, a specification for the interface between the two sub-blocks was needed. This required approximately 1500 lines of OVA code. The specification was implemented as an OVA unit with two separate bind files, one for each TLU sub-block.

Next, two separate proofs were set up. First a proof was run on block TLU\_sub1 to prove the properties described in the interface OVA description. This proof required a setup with the pipeline model and all the previous constraints that had been written for the top-level TLU. In the second proof, the properties in the interface OVA description were used as constraints for the TLU\_sub2 block in an attempt to prove the target one-hot property.

One complication that came up were important signals which did not go to both blocks. For example, a signal output from TLU\_sub1 was useful for specifying an interface property, but it did not propagate to TLU\_sub2. In order to have that signal available for the OVA code, a wrapper was created for block TLU\_sub2 so that these OVA signals could be connected to something.

The assume guarantee proof dramatically reduced the fan-in cones for the target property from 5800 to around 500, and we were hopeful we would be able to achieve a proof. Unfortunately the

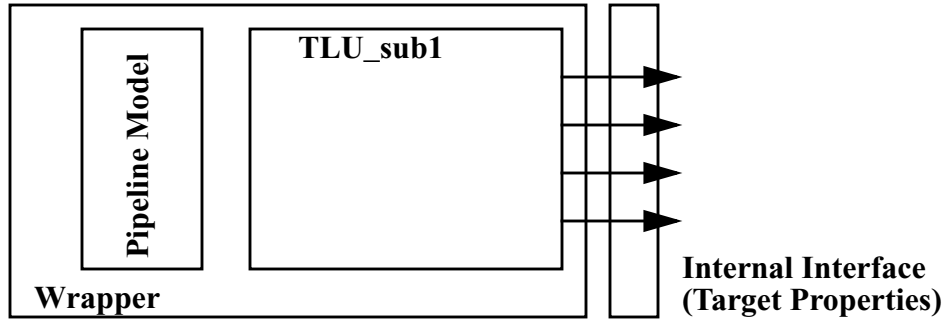


Figure 3. Setup for first half of assume-guarantee proof

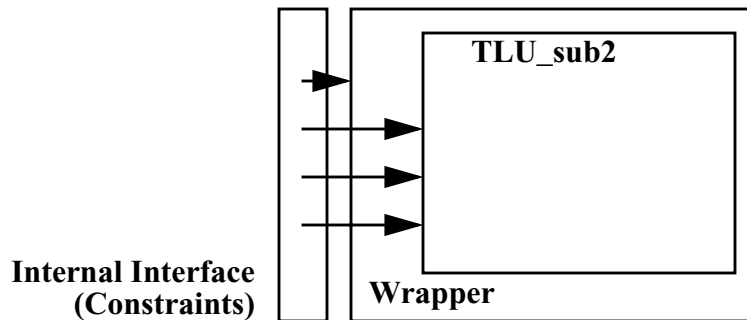


Figure 4. Setup for second half of assume-guarantee proof

proof still did not converge. However, there was a big increase in the depth of the bounded proofs reported by the tool. At the top-level, the tool reported that the property was proven for 16 cycles. With the assume-guarantee proof, the target one-hot property was proven for 155 cycles.

#### 5.4 Bounded Proof Mode

One recent enhancement to Magellan is the addition of a bounded proof mode. We ran the TLU proof in this mode trying to increase the depth of the bounded proofs. We found that we were not able to increase the depth of the proofs, but we did find falsifications that were not found in the normal mode.

## 6.0 Results

### 6.1 Properties Proven

Three sets of properties were run on this TLU. The first set was the one-hot property that we have been talking about. This property was never proven. A second set of properties we ran was a check to insure that if the pipeline was flushed for a trap, that the trap unit would re-direct the PC to the trap address within a reasonable amount of time. This property was also run on the top level trap unit, and like the one-hot property, was never proven. The final set of proofs was a set of priority proofs. These were run on one sub-block of the TLU. They checked that the different exceptions were always prioritized correctly. These proofs were much simpler and we were able to prove all of them.

## 6.2 Bugs Found

A total of 14 bugs were found in the design. Two or three of these bugs were relatively simple bugs, which were found only a couple of days before simulation-based testing would have found them. The remaining bugs were corner cases that probably would not have been caught in simulation, or at least would not have been caught until several months later. A breakdown of the bugs is shown in Table 1.

**Table 1: Summary of Bugs Found**

<b>Proof Set</b>	<b>Proven</b>	<b>Num Bugs Found</b>
Trap one-hot	No	11
Redirection	No	3
Priority	Yes	0
Total		14

## 7.0 Conclusions and Recommendations

While we are happy with the number of bugs we found, there is still room for improvement. We would like to improve our productivity in using formal verification. We want to see more bugs found in shorter period of time. There are several ways that this might be accomplished.

First, we need to start thinking about formal verification earlier in the design process. If the design is built with regular interfaces, it becomes much easier to verify. Part of the problem with the TLU design was the extreme complexity of the interfaces. The block designer never intended for sub-blocks to be verified separately, so there was little effort spent in making the interfaces between these sub-blocks simple and regular. In addition, the methodology being used by the CPU design teams is very optimized for physical implementation, not for verification, so we need to do more work to ensure that code is written in a manner that makes it easier to verify.

Second, we need better ways to triage the properties. Big properties that have a small chance of being proven need to be run in bug-finding mode, while more effort can be placed on properties that have a better chance of being proven. Here we need more help from Magellan. We need better ways of predicting the complexity of the design so that we can plan and act accordingly.

## 8.0 Acknowledgments

Thanks to Paul Jordan, who answered daily e-mails regarding the operation of the design. Thanks to Jerry Vauk and his team, who supported this effort. Thanks also to Xiaolin Chen and Dan Benua of Synopsys, who have helped us a great deal with Magellan.

## 9.0 References

- [1] R.M. Gott, J. R. Baumgartner, P. Roessler, S. I. Joe, "Functional Formal Verification on Designs of pSeries Microprocessors and Communication Subsystems," *IBM J. Res & Dev*, Vol 49, No 4/5, July/September 2005, pp 565–580.