A Case Study: Formal Verification of Processor Critical Properties

Emmanuel Zarpas

IBM Haifa Research Laboratory
zarpas@il.ibm.com

1 Introduction

Over the past ten years, the Formal Methods group at the IBM Haifa Research Lab has made steady progress developing tools and techniques that bring the power of model checking to the community of hardware designers and verification engineers, making it an integral part of the design cycle for many projects. Several IBM and non-IBM design teams have successfully integrated RuleBase [2], the IBM formal methods tool, into their design cycles. In this paper we present a case study describing the formal verification of critical properties in a recent processor. Because the details of the design and the specifications are highly proprietary, this paper focuses on the process, techniques and experience involved in the formal verification of the critical properties. We report here experiences on two units, named here for confidentiality reasons unit A and B.

2 Design Under Formal Verification

Unit A The original implementation of this unit had about 15,000 flip-flops and 220,000 gates, which is a challenge for *complete* formal methods. We checked about 200 properties in order to verify the critical properties of the unit as thoroughly as possible. We found 35 bugs. Three of these bugs were found after the first tape out of the SoC. We were also able to highlight the remaining weaknesses to the design team and the SoC architect (so they could fix them).

The design cannot reach one of the critical states in less than 600 cycles. This was proved to be far too deep for Bounded Model Checking [3]. The Discovery engine, the main BDD symbolic model checking engine used by RuleBase, proved to be the only engine able to cope with the problem. Even so, as the design grew larger and significantly more complex, we had to restrict the model. By the end of the project, it became impossible to check properties without the use of severe environment restrictions, see the first line of Table 1 for average data about models and Discovery runs. SAT-based bounded model checking could still be used for the design, but only for bounds lower than 200. We made a decision to override internal design variables in order to allow the design to reach all critical states within about twenty cycles and therefore achieve a reasonable level of coverage.

At the very end of the project, progress made in model checking technologies allowed us to check properties without overriding any internal variables or any restrictions of the behavior of control input variables. Using interpolation-based techniques as in [4] allowed us to prove two thirds of our properties, including some of the most sensitive properties. Our interpolation-based engine was able to prove these properties in a surprisingly short amount of time. Indeed when this engine was able to prove a property it was usually with an interpolant computed with a low bound (e.g. 10 or 15). In general, the engine was either able to prove a property quickly or unable to do it. For the remaining properties, we used incremental bounded model checking [5]. Second line of table 1 summarizes average data about models and runs. Of course, using bounded model checking, we could prove properties only with a bound. Reachability depths computed for the intermediate models made us think that the bounds (in the k=1000-1500 range) we used were probably high enough to prove most of the properties checked. However, this approach implied solving extremely big CNFs, indeed our SAT solver had to fight with CNFs of more than 20 million variables and 60 million clauses. With such CNFs memory becomes an issue, we had to work on a 8 GB 64-bit machine. Even though, each property often took more than 24 hours to be checked up to the relevant bound.

Unit B The B unit we checked had two main phases. Phase 1 lasts for several hundred thousand cycles. This makes model checking for these properties nearly impossible as is. To circumvent this problem, we used a checkpoint generated by simulation to give our model an initial state at the beginning of Phase 2 (main unit B concerns are mainly for Phase 2). As initial states, we took a subset of the reachable states in the first cycle of Phase 2. Consequently, we did not get full coverage. A bug could be missed, for example, if the only path to this bug is through a Phase 1 state that was outside the subset used. Nevertheless, using this method, we obtained a level of verification far better than any that could be obtained using simulation or semi-formal methods.

The original implementation of B unit had about 1200 flip-flops and 12500 gates. Because this not very large, we were able to check each rule in reasonable amount of time (anywhere from a few minutes to less than an hour). The design encompassed a 2^{19} counter making reachability analysis, and therefore on-the-fly model checking, impracticable. The Discovery engine allowed us to perform to perform an over-approximation of the reachable states by disregarding the counter variables. At this point we were able to use a classical backward fix-point computation search. In general, this proved to be a good solution for this design (see third line of Table 1). As a results a dozen bugs were found very quickly in an already mature design.

3 Lessons

According to users survey [2], the three most difficult activities related to Rule-Base use are writing environments to cope with size, understanding design details and modifying design for size. As we saw in previous section, technology progresses do make a difference in tackling big size designs, however in many cases brute strength is not enough. The [2] discussion about dealing with the size problem is still up to date, so we will focus here on processes considerations usually disregarded in the literature, though critical for projects successes.

Designer support is critical For verification engineers, their relationship with designers is one of the main challenges in verification projects, especially when the verification engineers act *de facto* as consultants. It is not surprising that we found it far easier to collaborate with skilled designers. Even if designers do not carry out any formal verification on their own, they need the time and availability to support the verification efforts being done on their design. First, the specifications are generally not detailed enough for formal verification work. The designer therefore has to give further explanations to the formal verification engineers and help them define properties and models. In addition, the designer plays an essential role in reviewing traces (either false negatives or real bugs) and giving feedback in a timely manner.

Have the "classical" verification team involved The more formal verification is embedded into the "classical" verification process, the better. Ultimately, the use of formal verification should be a part of the entire verification strategy. Even the system architecture should accommodate formal verification, for example by taking into account that formal works better on small blocks than on big ones (a very light case of design for verifiability). However, if the formal verification engineers do not belong to the "classical" verification team, as it is often the case, coordination should be established. The verification lead should closely review the bugs found by formal methods on a regular basis, including properties checked or not and model restrictions made. This is very important in order to get a good cooperation with simulation teams and the maximum benefit and return on investment for formal verification.

Write general environments (top down approach) In order to create a model, the behavior of input signals of the designs need to be defined. We model input signals behavior using the PSL [1] modeling layer. A safe approach involves starting with an environment as general as possible. A non-existential property proved with an abstract environment will still hold for the "real life" environment. If false negatives appear, the environment can be refined during the verification process. In addition, abstract environments tend to be simpler and easier to write. As a result, it is usually better to start with a general environment and refine it when needed. Indeed, starting with a very precise, very detailed environment will take a long time to write, debug and tune and therefore waste designer time, a most precious resource. By refining a general environment, you very well could never have to reach such a level of detail, and even so, it is likely to be at a late stage of the verification project and after achieving some results. Bottom up approach is more risky: it is very easy to lose considerable time in tuning in a precise way complex behaviors for some inputs signals with no significant gains. Very often some abstracted behavior would have done as well.

Write simple properties The RuleBase property language is PSL. PSL is simple to learn, yet the way it is used to write properties can have a significant impact on a formal verification project. The simpler the property the better. Simpler properties are easier to write, easier to understand, and easier to maintain. Even more important, the more complex a property is, the more difficult it will be to tune it and the more designer time, a rare and precious resource, will be required. It makes sense to start writing the simplest properties you can imagine for your model. This will allow you to assess your model and determine if it represents the design, its complexity, whether it is suitable for formal methods, or whether it should be made smaller by some restrictions. Many very important properties can be expressed in a relatively simple manner. We found that checking even trivial properties uncovered bugs. For example we found two bugs in A unit by checking that a signal was actually a pulse. When you want to write a complex rule, there is often a simpler version, or a simpler rule (either stronger or weaker) that will find the same bugs. It makes sense to first seek out the simpler rule. You may not be able to avoid writing and checking complex properties, however it is a safe policy to write them during a second iteration.

4 Conclusions

In this paper, we showed how skilled use of a state-of-art formal methods tool can allow checking critical properties in very important designs, in spite of technical difficulties. The author wishes to thank I. Holmes, J. Liberty and Kanna Shimizu for their support on design verification.

	Depth	#State vars	#gates	BDD nodes allocated	Memory Usage (MB)	Discovery runtimes	BMC runtimes	Interpolant runtimes
Unit A (int.)	1420	520	4670	$1.3 * 10^7$	260	10 h		
Unit A (final)		1760	24000				48 h +	1 h
Unit B	979	305	7640	$6.7 * 10^6$	63	0.25 h		

Table 1. Average values for Unit A and B models and engines runs. Depth is the number of cycles needed to complete reachability analysis.

References

1. Accelera. PSL LRM. http://www.eda.org/vfv/
2. S. Ben-David *et al.* Model Checking in IBM. In Formal Methods in System Design, 22, 2003.
3. A. Biere *et al.* Symbolic Model Checking Without BDDs. TACAS'99.
4. K. L. McMillan. Interpolation and SAT-based Model Checking. CAV'03.
5. O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. CHARME'01, 2001.