A FORMAL POT-POURRI

Laurent Arditi
ARM, Processor Division
laurent.arditi@arm.com

INTRODUCTION

In the context of the design and validation of the ARM Cortex A9 processor, formal verification (FV) has been experimented very late in the project life. Traditional validation techniques are widely used: simulation, emulation, block-level directed-random, top-level with architectural and device specific test suites. But, except for logical equivalence checking, formal methods were initially not recognized as a solution with a sufficient Return On Investment (ROI). Recent improvements in the FV tools (we consider model-checking tools here), raised the opportunity to further evaluate that ROI.

We describe here a one-year extensive usage of FV on the Cortex A9 processor, its companion L2 cache controller, and other older designs. Our initial goal was to assess the applicability and the benefits of using state-of-the-art commercial FV tools for “deep formal verification”. That is to verify end-to-end properties related to cache coherency, the absence of deadlocks, or the completeness of hazard detections. Another goal was to develop a simple and automatic flow to formally verify the properties embedded into the designs and which were previously only checked by simulation. We quickly found new and unexpected applications of FV to solve urgent issues.

CONTRIBUTIONS

DEEP FORMAL VERIFICATION

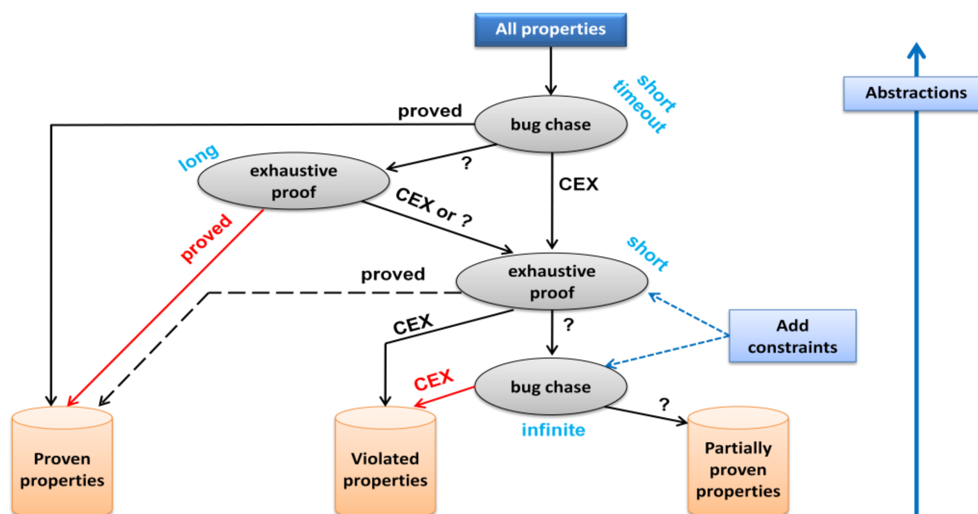
The Cortex A9 proccesor is a multi-core design and as such we wanted to formally verify some cache coherency properties. The properties we verified are about the coherency protocol implementation and the data integrity when a cache line needs to move or be copied from one core to another. We could find a bug in a development version of the design. It was identified as a bad data signalled by a scoreboard. It was very unlikely that simulation could catch that bug, and indeed it was missed.

We have experienced the use of FV to check the absence of deadlocks and livelocks. Usually, this verification task is applied at the FSM level, and verifies that a single FSM can not deadlock. But this is only a partial validation because real deadlocks are usually caused by the interactions of many blocks and FSMs. We thus think deadlock detection must be applied at the top-level, on the full design. The deadlock and livelock properties are then liveness properties about the possibility of the top-level module to activate some of its primary outputs: the VALID and READY signals of its AXI ports. The proof of a liveness property at that level seems to be a very challenging task. However, it could quickly show a valid counter-example: it was a known deadlock that the design team hadn’t fixed yet.

FORMAL FLOW FOR EMBEDDED ASSERTIONS

Assertions were already present in the RTL. The difficulty in verifying them using FV is their number (hundreds to thousands). We have developed an automatic flow to maximize the number of full proofs (unbounded) and the number of valid counter-examples (CEX), if any. This flow is based on the experimental observation that,

on the one hand, it is easier to get a full proof when the design is not completely constrained and a high-level of abstraction is used. On the other hand, valid counter-examples need a complete constraint set, but a low abstraction level. This flow iterates with different abstraction and constraint levels, with proof engines targeting full proofs or bug-chasing. It can also benefit from the distribution of the proofs on a cluster.



FORMAL VERIFICATION AS AN AID FOR OUR SUPPORT TEAM

In addition to the tasks described above, we have found other applications where FV has a very high ROI. One is the analysis of known errata (bugs found by simulation, emulation or even after tape-out). We first developed properties showing these bugs. In all cases, FV could show counter-examples. That means that if the right properties were in place and they were formally verified, the errata would have been avoided. We use FV to fully characterize the errata (exact context, revisions of the design they are in) and verify their fix.

FV has also been used to answer several customer support requests. For example, when a customer reports a problem on one IP, he usually exposes the faulty output trace, but not the input trace (the software in case the IP is a processor). FV can be used to automatically generate an input trace producing the faulty outputs. A similar technique can be used to answer questions about unpredictable behaviors of an IP. When the documentation, the RTL review and the simulation cannot give clear answers, FV is the right technique to query a design.

RESULTS

The primary tool we have used is JasperGold® from Jasper Design Automation. We have found FV very efficient to find counter-examples (bugs in general, cover traces when querying a design). This is also true for deep-formal properties, even if they seemed to be highly complicated and out of the tool capacities. Indeed, in all the experiments we have done, when a counter-example was expected (because we knew a bug was present), the formal tool was able to show this counter-example. It requires usually a short amount of time (max. a couple of hours), even on huge designs such as the Cortex A9 processor in full. We thus think that if FV was able to find known bugs, it is also able to find most unknown ones.

Getting exhaustive proofs is difficult and very time consuming. Techniques to move from a bounded to an unbounded proof require a very detailed understanding of the design-under-verification. Usually only the designer is able to do that, but [s]he has no time for that!

We claim that FV must be used as a complement to other validation techniques. The first goal should not be to prove the absence of bugs. Instead, it should be to catch new bugs. With very limited efforts and setup time, FV can find bugs when they exist. In that sense this is a very powerful technique with a high ROI.