A low risk, high reward approach to adopting formal methods

James Pascoe - STMicroelectronics R&D Ltd. - April 08, 2013

At STMicroelectronics, we recently investigated using formal methods as a complement to the constrained random testing of our ARM based CPU subsystems. Our motivations were to:

1. Achieve verification closure with appreciably less time and effort than that required by a constrained random approach
2. Encourage designers to develop formal properties for their blocks. Functional insights can be expressed as PSL or SVA assertions in the RTL. These properties then provide follow-on benefits in the subsequent design stages
3. Augment or replace legacy in-house flows with mature industry tools. This reduces maintenance overhead and promotes a more robust approach
4. To address the engineers’ reservations about formal’s ease of use, we took a stepwise approach that would enable them to accumulate expertise incrementally. This approach also significantly reduced the project’s risk.

Accordingly, we applied formal methods in parallel with constrained random testing to three designs, each more complex than the previous one, each yielding different insights into the effectiveness of formal methods.

Along the way, we found that, using a formal approach, we verified in four engineering weeks what had taken six engineer weeks of constrained random test-bench development effort. We could understand the functionality of a block for which the functional specification was not available; we found errors in blocks that had been verified by other means and we clarified ambiguities in specifications quickly and efficiently. This process steadily built our confidence that formal methods could deliver a return that was worth the investment.

This article describes our work and the results that we achieved.

Our verification challenge

We are responsible for the CPU subsystem side in a CPU/GPU design team. We develop ARM based subsystems that are integrated into a wide range of system-on-chip (SoC) design platforms (see Figure 1). The SoCs are targeted at a wide range of applications from mobile devices to Big Data, Cloud and High Performance Computing.
Our subsystem verification strategy is focused on three levels of abstraction:

**Unit testing**: The objective of unit testing is to ensure that the blocks developed by ST are ready to enter verification. Unit testing is typically performed by the designers, using HDL-based test-benches. These test-benches are often not reusable or of limited use when the block enters verification.

**Block-level verification**: The objective is to ensure that each block conforms to its specification in isolation from the rest of the system. Block-level projects are typically conducted by verification engineers using a constrained random approach.

**System-level testing**: Once the individual blocks have been verified in isolation, they are integrated into a subsystem. Point-to-point connectivity checks are performed to detect wiring errors in both the functional design and the power management circuitry. This is followed by functional testing to ensure that the subsystem operates as a whole. Functional testing is performed using both constrained random and directed tests. The subsystem may also undergo performance testing, for example using approaches defined by the Embedded Processor Benchmark Consortium (EEMBC) [1].

We used a three-step process to apply formal methods to each of these verification challenges.

**Step 1: A well-understood block**

We started our use of formal methods by verifying an in-house developed IP called the Sensor Control Block (SCB). This design periodically monitors on-chip temperature and generates the appropriate power control interrupts to enable circuit switch-off.

The SCB is well specified and well understood, so it was a good test case for our first steps in formal. The SCB enabled us to identify formal’s complementary added value to the constrained random approach. Also, the SCB is small enough to avoid the state-space explosion problem — an undesirable complication for our initial study. The block had been extensively unit tested by the designer prior to verification. In addition, we developed a constrained random test-bench so that we could compare the two approaches.

We used JasperGold to check the APB interface; the interrupt properties; the min, max and averaging functionality; the output signals and the register interface (see Figure 2).
The green areas indicate the proportion of properties that have been proved while yellow denotes the properties that have not yet been evaluated. Red areas show properties that have failed. Each failed property is accompanied by a counter example that can be analyzed to understand the root cause.

**Results**

We found three bugs, all in the first week of using formal. One bug was a problem with the ‘PREADY’ signal on the APB interface. This issue had not been previously detected despite constrained random testing. The second bug was an RTL problem, namely an erroneously inverted ‘or’ concatenation on the sensor overflow / underflow bits. The third problem was a specification error. Two registers were specified as requiring a ‘Write 1’ to be cleared, whereas, the desired (and verified) behavior was that these registers were also clearable by writing 0.

In the case of the SCB, our constrained random test-bench detected only one further issue of significance in addition to those found by the formal tool. Moreover, it took approximately seven engineer-weeks of development for the constrained random test-bench to reach sufficient maturity. By contrast, formal methods discovered three bugs within the first week of use, and produced a list of Property Specification Language (PSL) properties that can be embedded in the RTL. These properties can now be reused in both the system-level verification and the SoC integration.

**Step 2: A not-so-well-understood block**

Our next step was to verify a more complex design called the Clock and Reset Manager (CRM). The CRM takes high-level clock and reset inputs from the SoC and generates the appropriate signals to the individual blocks in the subsystem. For example, the SoC may request a ‘power on’ reset at any time and it is the CRM’s responsibility to ensure that bus transactions are completed and that the individual blocks are reset in the correct manner. The CRM also enables the subsystem to be fully asynchronous and so must conform to the ARM sign-off criteria.

We had good micro-architectural design documentation for the CRM — lots of schematics — but no functional specification or timing diagrams. It was difficult to devise a verification plan because it was unclear what the effect of stimulating each functional input should be. We required a means to extract the features from the micro-architecture. This would allow us to review them with the designers and architects to establish the block’s correct behavior. Feature extraction can be time-consuming using a constrained random approach. This is because each exploratory test-case can require a significant amount of coding. However, we had more success using the ‘visualization’ capability of our formal tool.

How does the feature extraction work?

Consider one situation where the CRM functionality was not well specified. We would typically
receive (often verbally) the following types of behavioral description:

Once ‘cmd_in’ rises ‘cmd_ack’ should rise after ‘N’ cycles

Once ‘cmd_in’ falls, ‘cmd_ack’ will fall after ‘M’ cycles

where ‘N’ and ‘M’ are unknown.

To determine N and M, we used the tool’s visualization capability (see Figure 3). For this case, we wrote two properties which allowed us to determine N and M’s maximum values. We then refined these properties to determine the lowest value of N and M for which the properties hold. These values could then be reviewed by experts to establish their correctness. The key advantage of using formal in this context is that hypothesis testing can be performed quickly — significantly faster than using a constrained random approach. Using formal, it is often possible to perform feature extraction in real-time with the designer.

![Figure 3: Using the JasperGold® visualization tool to perform feature extraction](image)

**Results**

Formal verification of the extracted feature set found no functional bugs. To some extent this was expected as the block had been verified previously. However, confidence in the previous verification was low due to the specification’s lack of clarity. The payback was two-fold:

- **Understanding**: feature extraction enabled us to complete the functional specification. This provided an approach to solving a big (and increasingly common) problem.

- **Increased confidence**: The formal verification results yielded 73 fully-proven properties; 88 covers, all covered; and no evidence that we had over-constrained the design.

In addition, the tool’s dead code analysis identified cases relating to unreachable reset branches and false functional assumptions.

Moreover, 28 of the 73 properties that we wrote did not initially converge. However, using the tool’s bounded proof coverage analysis we determined that all of our cover items were reachable in 268 cycles or less. We were exploring all of the properties to a search depth of 773 states, so we concluded that the bounded proofs were acceptable. Subsequently, we used more powerful proof engines and showed that all of the properties converged.

**Step 3: An IP integration challenge**
Once the blocks have been verified in isolation, they are integrated into the subsystem. Certain parts of the integration process are performed manually. Thus, in order to verify that there are no wiring errors, we used formal methods to run a series of point-to-point checks. These checks are not only used in a functional context, they can also be re-used to verify connectivity in power-aware design scenarios. Thus, formal point-to-point testing provides a useful sanity check before the main system-level functional testing begins.

Results

Using an in-house developed documentation-driven point-to-point flow, we generated 2,568 connectivity properties. The setup of the formal tool was completed in around one day, and we required three engineer-weeks to prove all 2,568 properties. Through this activity, we detected and resolved 11 connectivity issues.

Progress and coverage

As with dynamic verification, we used coverage metrics to assess the project’s status and drive its progress. We were faced with the usual verification questions, for which formal methods provides analytical answers. These questions included:

Q1: Completeness: Are there enough properties to verify all of the functionality?
A1: The formal tool enabled us to identify and measure coverage holes outside of the areas being analyzed. This provided a good indicator of the project’s progress and enabled us to steer the work to completion.

Q2: Constraints: How do we know that the environment is not over or under-constrained? Over-constraint implies that bugs may be hidden, whereas, under-constraint means that the block could potentially be subjected to invalid states.
A2: The formal tool measured stimuli coverage to ensure that only scenarios which were valid in the context of the design’s intended operation were exercised and covered.

Q3: Bounded proofs: Do we need to increase the depth of the search space?
A3: In all cases, the search depth we were exploring was greater than the depth required to trigger all of the design’s cover points. This gave us confidence that the bounded proofs were traversing the state space to a sufficient depth.

Q4: Mixed formal and dynamic verification: How do we interpret coverage data from the formal tool in combination with constrained random?
A4: We manually reviewed the formal coverage alongside that produced by our constrained random test-bench. The formal tool’s use of the Unified Coverage Interoperability Standard (UCIS) API [2] will eventually enable us to merge these results into other Unified Coverage Databases (UCDB). Thus, it will be possible to combine the coverage results from the two verification approaches.

Wrap up

How do formal methods stack up against our hopes and expectations? We have achieved our cost, time and effort reduction goals. We have achieved our property reuse goal. And we have made significant progress in transforming the legacy aspects of our verification flows into a more
comprehensive and robust approach, with a lower maintenance overhead.

With our step-wise adoption strategy, we found that formal verification is a powerful tool, even in the hands of engineers new to formal methods. The technology helped with interpreting specifications; detecting errors, omissions and inconsistencies; feature extraction from the RTL in the absence of a specification; testing assumptions, both explicit and implicit; unit testing with a reusable test-bench; identification of dead code; block testing, and verifying properties such as the absence of deadlock.

We're only a few months into our adoption ramp, and formal methods have already proven their worth.

References


About the author

James Pascoe is a Microprocessor Architect currently working on the verification of ARM-based CPU subsystems at STMicroelectronics. James is based in Bristol, UK.

If you found this article to be of interest, visit EDA Designline where you will find the latest and greatest design, technology, product, and news articles with regard to all aspects of Electronic Design Automation (EDA).

Also, you can obtain a highlights update delivered directly to your inbox by signing up for the EDA Designline weekly newsletter – just Click Here to request this newsletter using the Manage Newsletters tab (if you aren't already a member you'll be asked to register, but it's free and painless so don't let that stop you).