Shaping Formal Traces without Constraints:

A Case Study in Closing Code Coverage on a Crypto Engine Using Formal Verification

David N. Goldberg, Adriana Maggiore, David J. Simpson
Ubicom, Inc.
San Jose, CA
{dgoldberg, amaggiore, jsimpson}@ubicom.com

***Abstract* — This paper shows a successful cooperation between simulation and formal verification which enabled code coverage closure with minimum effort. The case study presented is the cryptography engine of Ubicom IP8000 processor. Formal verification, in the form of unreachability analysis, was first applied to refine code coverage results by identifying the unreachable targets, among the uncovered targets, for all the components of the IP8000 core. Among the remaining reachable targets there were data dependent blocks of the 3DES (Triple DES, Data Encryption Standard) engine. Covering those targets with simulation alone would have required a large number of random simulation cycles, with no guarantee of hitting the coverage target, or a very time-consuming unrolling of the DES algorithm iterations. Instead, we used formal verification to find the 3DES inputs that were needed to reach 100% coverage. This block had not undergone assertion based verification, therefore there were no constraints to model the environment in formal analysis and the cost of developing them from scratch would have been too high. However, several existing tests, similar to those required, provided examples of legal traces for the block. In this paper we show how we captured the essence of those traces, the key control signals and their timing, in an SVA (SystemVerilog Assertion Language) sequence and how we used that sequence in an SVA cover statement to generate values that could be directly used in a system level test in order to close the coverage loop.**

Keywords: unreachability analysis; coverage closure; formal verification; simulation

I. INTRODUCTION

In this paper we present a case study of combining the usage of formal verification [1] and simulation techniques to close code coverage [2] on a 3DES [3,4] block, a component of the Ubicom IP8000 processor [5]. The 3DES block is part of the IP8000 hardware crypto security engine, which includes encryption, decryption, authentication and key exchange with support for DES/3DES as well as MD5 (Message Digest algorithm), AES (Advanced Encryption Standard), SHA1, and SHA2 (Secure Hashing Algorithm) to accelerate networking algorithms and wireless security. We describe how we used existing simulation results and formal verification without constraints to find the 3DES inputs that were needed to reach 100% code coverage.

Code coverage is an effective technique to identify functional verification holes. While 100% code coverage does not guarantee the full exploration of the design functionality, less than 100% clearly indicates that some code was never exercised. Code coverage is one of the key metrics in Ubicom's functional verification methodology whose objective is to deliver very high quality design in a cost-effective way. The case study presented in this paper is an example of how best results can be achieved with minimum effort by using the most suitable tool for the task at hand and by exploiting complementing technologies such as simulation and formal analysis. Coverage is measured towards the end of the project to assess the quality of the verification effort. New directed tests, which target the unreached coverage objectives, are then written.

A key step in coverage analysis is to separate the achievable from the unachievable coverage targets, a task for which, in the IP8000 project, we used unreachability analysis. This consists of a formal proof that a given coverage target cannot be reached. Unreachability analysis is an example of how formal verification can be used to support a simulation based verification flow. This technique has been previously used [6,7] and it is becoming a standard feature of formal tools [8]. In this paper we take the cooperation of formal and simulation a step further by showing how formal analysis and simulation can be efficiently used to close the coverage loop. In particular we show how to exploit the result of unreachability analysis, existing simulation tests and formal verification to create, with very limited effort, the missing test cases for the 3DES engine.

The rest of the paper is organized as follows: section II gives a description of Ubicom IP8000 and its verification strategy; section III describes the application and results of unreachability analysis to Ubicom IP8000 core; section IV outlines the functionality of the crypto engine and the missing test cases; section V describes how the missing tests were created by using a combination of formal verification and simulation; section VI concludes this paper.

II. VERIFICATION OF UBICOM IP8000

Ubicom is a fabless semiconductor company headquartered in San Jose, CA that develops processor systems that optimize data flow where Media Meets Networking. The IP8000 processor family is the fifth generation of Ubicom's line of 32-bit processors for high performance embedded applications.

IP8000 has an 800MHz 12-threaded CPU with 64KB level-1 instruction/data caches and a separate 256KB on-chip

memory. Significant differences from previous generations include the additions of MMU (Memory Management Unit), FPU (Floating Point Unit), BTB (Branch Target Buffer) and pipe-depth within the CPU; support for multiple outstanding misses in-flight from the caches; and DDR3, PCIe Gen-2 and USB 3.0 external interfaces. Unique to Ubicom processors is the ability to schedule 12 threads down a single non-stalling pipeline and an efficient memory-to-memory ISA.

The verification team was involved from the architecture phase all the way through to post-silicon performance validation, with bugs being progressively more expensive to fix the later they were found. The design team was also "persuaded" to help-out: using design-for-verification techniques, one example being a run-time switch on the MMU that allowed legacy pipeline tests to be run on the new CPU.

Given the large number of product changes and a small team with limited simulation farm resources, we had no choice but to work smart – selecting the most suitable tool for any task to get the best result with the least effort. When targeting the IP8000 core module the team employed directed tests, constrained random tests, assertions and formal proof based techniques to verify different blocks.

CPU tests were written in UBICOM32 assembly language. Given the effort to write tests at this level, Ubicom engineers use a lightweight BNF context-free language to expand their individual directed test into a constrained random one; this format is supported by the internally-developed tool DRAG (Directed Random Algorithm Generator).

These CPU tests were passed through the UBICOM32 toolchain to produce instruction and data memory images that were loaded into a Verilog simulator. Traces were captured by monitors during the simulation and used to compare off-line against a golden reference model in a post-process step.

Coverage analysis was used to determine when constrained random testing was asymptotically approaching its productive limit for any block. At that point other techniques such as directed tests or formal proofs were used to hit the remaining coverage holes.

Towards the end of the project we booted Linux on the RTL simulator, a task that took 96-hours to complete and which found numerous software bugs along the way. For completeness the verification team also ran gate-level simulations: hierarchical, flat, and SDF-annotated; looking for problems in the reset state of the machine, scan connectivity and "don't-care" design space left uncovered by LEC tools.

III. Refining Code Coverage Results: Unreachability Analysis

As described in the previous section, code coverage was collected towards the end of the IP8000 project, in order to assess the quality of the verification effort.

The block coverage for the IP8000_core module (see Table 1) was 93.41%; 10 out of 14 components had less than 100% block coverage, a total of 967 uncovered targets. Unreachability analysis was used to identify unreachable targets. This consists in a formal proof that no sequence of input values exists that covers the block of code under analysis. Details of how unreachability analysis was applied are given in the next subsection. We first discuss the results we obtained.

Block Coverage	Name
93% (13710/14677)	IP800_core (cumulative)
Immediate sub-instances	
93% (5303/5679)	Unit1
83% (59/71)	Unit2
78% (142/182)	Unit3
96% (880/915)	Unit4
78% (757/969)	Unit5
96% (1328/1378)	Unit6
93% (255/273)	Unit7
96% (3705/3852)	**Unit8 (including security engine)**
92% (649/701)	Unit9
96% (561/586)	Unit10
100% (58/58)	Unit11
100% (5/5)	Unit12
100% (6/6)	Unit13
100% (2/2)	Unit14

Table 1. Initial block coverage for the IP8000_core and its components

Unreachability analysis identified 579 targets as unreachable (see Table 2). Those were reviewed to rule out design bugs; it was established that the unreachable targets were default statements or blocks not used in the chosen mode of operation, selected using either input values or parameter values.

Block Coverage	Name
97% (13710/14098/579)	IP800_core (cumulative)
Immediate sub-instances	
98% (5303/5393/286)	Unit1
91% (59/65/6)	Unit2
88% (142/162/20)	Unit3
99% (880/886/29)	Unit4
88% (757/856/113)	Unit5
97% (1328/1365/13)	Unit6
93% (255/273)	Unit7
98% (3705/3768/84)	**Unit8 (including security engine)**
95% (649/680/21)	Unit9
97% (561/579/7)	Unit10
100% (58/58)	Unit11
100% (5/5)	Unit12
100% (6/6)	Unit13
100% (2/2)	Unit14

Table 2. Block coverage after unreachability analysis of the IP8000_core components (covered blocks/total reachable blocks/unreachable blocks).

Of the remaining 388 uncovered targets, 346 were identified as unreachable by the designers, leaving 42 blocks for which new tests needed to be developed. Five of those blocks belonged to the 3DES module of the security engine (a sub-instance of Unit8); in this paper we report on the technique used to craft the tests for these 5 targets.

The final block coverage was 99.91%; the last 12 uncovered targets belonged to the Data Cache arbiter. It was decided to formally verify this module and the remaining coverage was waived. Table 3 summarizes the uncovered blocks throughout the coverage review process.

	Uncovered blocks
Initial uncovered blocks	967
After unreachability analysis	388
After designer review, to be covered by new tests	42
Final uncovered blocks, to be addressed by formal verification	12

Table 3. Uncovered blocks throughout the coverage review process

A. Applying Unreachability Analysis

Unreachability analysis uses formal analysis to assess if a given coverage target can ever be reached. If a target is identified as unreachable, there is no sequence of inputs which covers the target, while matching the constraints of formal analysis.

We used Cadence Incisive Comprehensive Coverage (ICCR) to analyze coverage and Cadence Incisive Formal Verifier (IFV) to perform unreachability analysis. At the time of the project, the two tools had not yet been integrated into one flow, now the Code Coverage Unreachability flow of Cadence Incisive Enterprise Verifier-XL (IEV). Therefore we used scripting to convert coverage results to formal targets and vice versa.

A first script *iccr2ifv* analyzed the ICCR coverage report and translated uncovered blocks into IFV deadcode automatic assertions. There are 2 limitations to the this part of the flow: a) there is no IFV deadcode equivalent of "implicit else" blocks, which occur when an "if" statement without an "else" clause is used; b) care needs to be taken when targets belong to included files because the two tools use different qualifiers for the line numbers. The IEV unreachability flow removes these limitations and extends the unreachability analysis to expression coverage.

The selected deadcode automatic assertions were run in IFV. A failed assertion indicates that the target code is indeed "dead" and therefore unreachable in simulation. A second script *ifv2iccr* translated back failed deadcode assertions to blocks to be filtered out in ICCR.

We had to choose the hierarchical level at which to run the formal deadcode analysis. Although we made an attempt at running at IP8000_core level, it was deemed more efficient to perform unreachability analysis on each IP8000_core component with uncovered targets. The only abstraction technique used was blackboxing of the memories. The total run time of unreachability analysis was less than 2 hours, while the maximum memory consumption was less than 2.8GB. The size of the IP8000_core components ranged from 70 FFs to 35K FFs. Notice that only the uncovered targets were formally analyzed, as the covered targets were obviously not deadcode. This resulted in a more efficient usage of the formal tool.

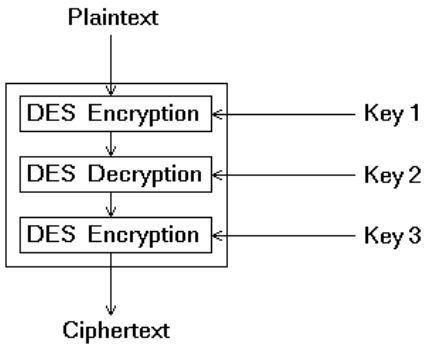

Figure 1. Triple DES – Overview of Encryption

No constraints were used in the unreachability analysis, even when they were available, as the objective was to identify structurally and unquestionably unreachable code. The only settings specified were the clocks (some components have multiple synchronous clocks) and the reset state.

IV. CRYPTO ENGINE AND MISSING TEST CASES

Ubicom's IP8000 network processor provides hardware acceleration for network security protocols such as IPSEC, VPN and SSL via its encryption, decryption and hashing algorithm implementations. Its security engine includes blocks that implement the most prevalent cryptographic algorithms, including AES (Advanced Encryption Standard), MD5 (Message Digest algorithm), DES and 3DES (Triple DES, Data Encryption Standard), and Secure Hashing Algorithms variations: SHA-1, SHA-256, and SHA-512, SHA-224 and SHA-384.

In this paper we describe the 3DES engine and the verification methods used to complete its design verification block coverage. The 3DES cipher logic in the IP8000 processor reuses a DES encryption/decryption hardware block (see Fig. 1) along with 3 key registers and a control state machine in order to produce its results. While we examine DES as our example, the same coverage hole issues and solutions, including formal methods, could be applied to any of the security engine's crypto blocks.

The 3DES function performs a series of three DES encrypt/decrypt calculations with three keys, so its hardware consists of a DES cipher function block and a control state-machine.

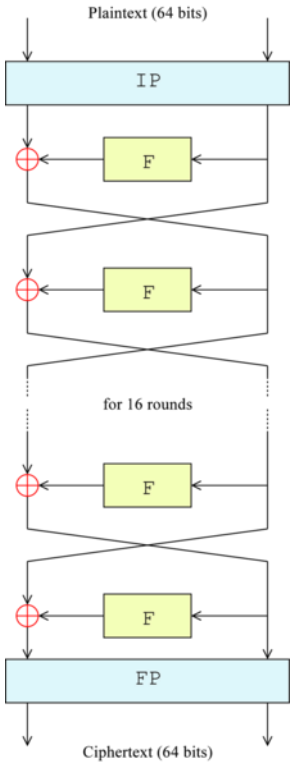

Figure 2. Internal structure of DES (from Wikipedia)

Briefly, the IP8000's DES hardware block takes in a 64-bit data block and a 64-bit key. Its cipher function consists of 16 rounds of logical processing (Feistel function [9]), preceded by an initial permutation and followed by a final permutation (see Fig. 2 and Fig. 3). The result is a 64-bit output. The DES function is symmetric, in that it uses the same key for encryption and decryption. Each of the crypto algorithm function blocks in the IP8000 design has a set of directed verification suites, which include mode variations, timing variations and data patterns along with pre-calculated results.

In addition, some of the blocks, such as the DES/3DES cipher block use a C-reference model to enable directed random input data and key generation and result checking for its encryption and decryption modes. (It should be noted that DES is now considered obsolete and 3DES, a variation of DES that is repeated 3 times with up to 3 independent keys, has been superseded for many applications by AES, which is also supported by the IP8000 security engine.)

The design verification effort for the 3DES block made use of DRAG, Ubicom's internally-developed tool, to support its directed random testing. This allowed us to generate repeatable random tests and feed data patterns to the Design-Under-Test (DUT) as well as the C-reference model and use the reference result for checking of the DUT.

Of course, given the number of possible patterns, this still meant that it could take a long time before random testing would cover all the logic. Note that for 3DES, there are 2^{168} or 3.74×10^{50} possible keys, for DES; there are 2^{56} or 7.2×10^{16} keys.

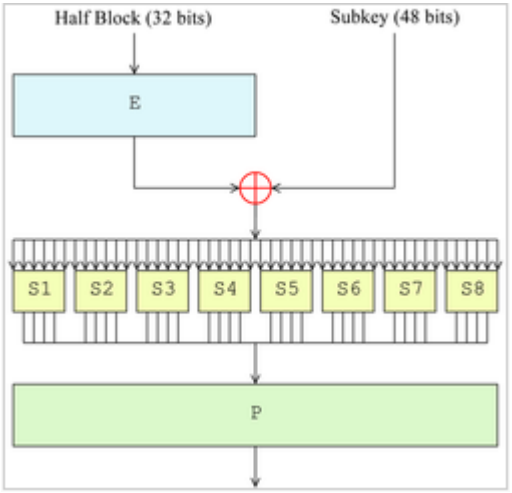

Figure 3. Feistel function (F-function) of DES (from Wikipeida)

Our IP8000 design verification coverage effort yielded a number of uncovered code blocks within the security engine. Some were timing cases, such as concurrent events and gaps between events, e.g. result data ready versus new data inputs. Others were variations of algorithm mode combinations, such as cipher chaining. These control-based cases were straightforward to create and add to our test-suite. However, there were still a number of uncovered blocks that were data dependent.

The approaches that we considered for solving this problem included more random simulation, modifying the C-reference model, and manually creating additional tests.

In fact, when examining the uncovered blocks, we found that the 3DES block still had some intermediate logic terms that were not covered, even after much simulation with random inputs. Given an intermediate result, it is possible to "unroll" and rewind or reverse-engineer the encryption calculation to find sets of intermediate patterns for the multiple stages which eventually map back to block input patterns, but this can be a time-consuming job and might be best served by permuting and using the C-reference model.

In our case, there were a number of intermediate terms that were uncovered from various stages of the block, so modifying the reference model could also be time-consuming and one-time/throwaway work. Given enough simulation cycles, we could probably have covered the remaining logic using our existing random methods, but that would consume simulation cycles and be slower due to coverage collection. Finally, we could have tried to manually derive the input values from the known uncovered intermediate values, but this would have been a difficult and time consuming task even after reducing the number of unknown by using fixed key values.

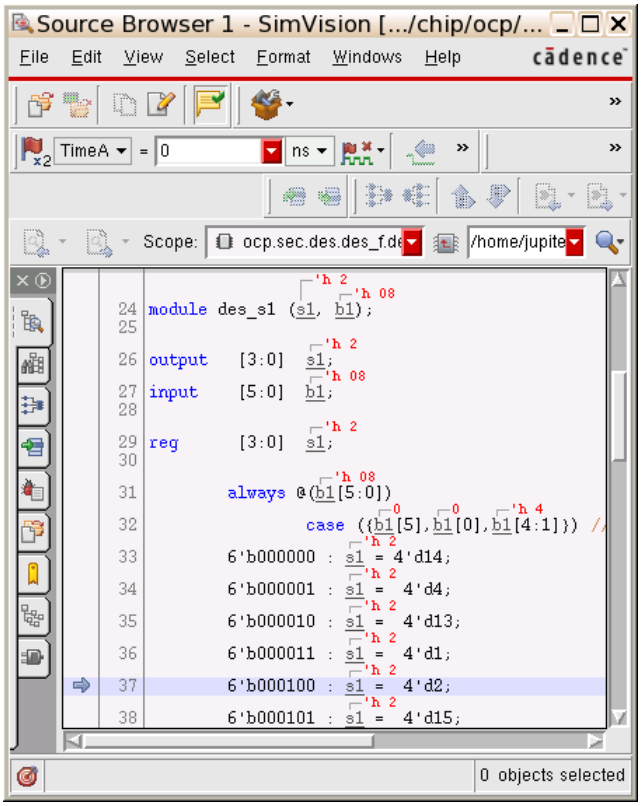

Figure 4. Source code annotation from the witness of the unreachability analysis of line 37; it indicates that the line is reached when the signal *b1* has the value 'h08.

Instead, we considered the possibility of having a formal verification tool do the work of reverse-engineering the patterns. We were already using the formal tool to find deadcode in the security engine, so we explored using the results to produce a set of possible input patterns and keys. In fact, we realized that we could make the problem easier because we could constrain the key value. In this case, we needed to not only derive the value of the intermediate causal input, but we needed to derive the original input pattern that was presented to the block itself in order to most easily feed the patterns into proper test cases.

V. Closing the Loop: Creating the Missing Tests using Formal and Simulation

The first attempt in generating the missing tests for the 3DES block was based directly on the results of the formal unreachability analysis. IFV found the uncovered code to be reachable and generated a witness, which showed how to reach those lines at the module level (Fig. 4 shows the source code annotation from the witness of the deadcode assertions for line 37 of the module *des_s1*). Tracing the driver of the input *b1* showed that the witness value of *b1* was derived from the value of an uninitialized register. This was neither a scenario that could be reproduced in simulation or a significant one for functional verification. At the same time, the local value of *b1* alone was not enough to craft a meaningful test case. What was needed was the input of the entire 3DES block or even better

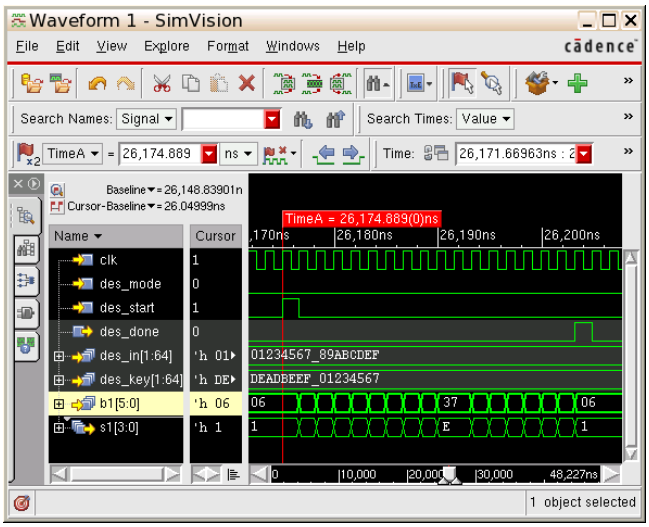

Figure 5. Simulation waveform for test *sec_3des*

the values of the input registers that would be loaded by the CPU to start a 3DES calculation.

A more interesting witness could have been obtained by modeling the environment of the 3DES block. However this block had not undergone assertion based verification, therefore there were no constraints to model its environment in formal analysis and the cost of developing them from scratch was deemed too high with respect to the risk of bugs in the block. Modeling of the environment would have required a) the formal verification engineer to become familiar with the functionality of the crypto engine and its integration within the system and b) to describe the environment in SVA [10].

The alternative was to examine existing tests that exercised the same logic to understand how similar blocks were covered under normal operating condition.

Finding the tests was fairly straightforward thanks to the way the tests were named and organized. Once a candidate test was identified, it was confirmed that it was of interest by looking at the block coverage for the test itself. The specific test, called *sec_3des*, covered line 64 of the *des_s1* module.

In order to understand how the *sec_3des* test achieved coverage of the *des_s1* line, the inputs of the 3DES module (which contained *des_s1*) and the signals *b1* and *s1* of *des_s1* were displayed in a waveform viewer (see Fig. 5).

It was then clear that the desired value of the signal *b1* (shown in the witness of the deadcode assertion and equal to 'h08) had to be produced at some point in the 16 cycles between *des_start* and *des_done*. This behavior could be easily captured in an SVA sequence consisting of two parts: a) a sequence describing the behavior of *des_start* and *des_done*; b) a sequence describing that *b1* takes the desired value at least once. The two SVA sequences were then combined by the *intersect* SVA operator, so that the first sequence, which has a fix length of 18 cycles, constrained the length of the second one. Finally, the resulting sequence was instantiated in an SVA cover statement:

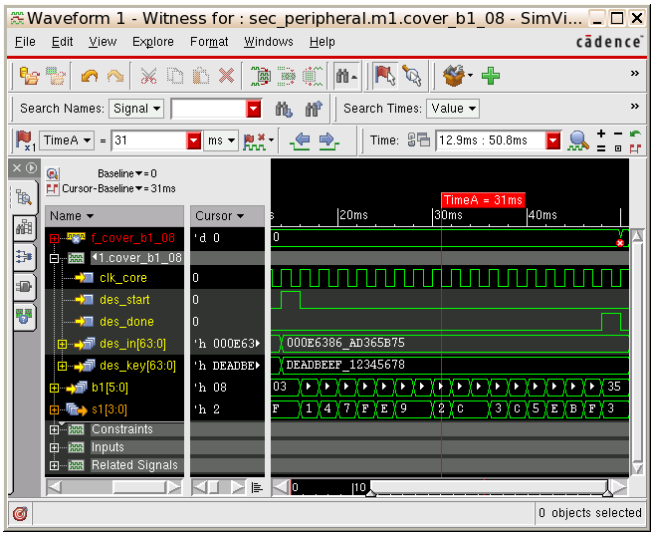

Figure 6. Witness for cover_b1_08

```
cover_b1_08: cover property
(@posedge clk_core)
 (des_start && !des_stop
  ##1 !des_start && !des_stop [*16]
  ##1 !des_start && des_stop)
 intersect
 (##1 (des.des_f.des_s1.b1[5:0] == 'h08)[=1]));
```

Running the cover statement in IFV produced a witness that showed the desired value of signal *b1* within *des_start/des_done* cycle and, more importantly, the values of *des_in* and *des_key* needed to drive *b1* (see Fig. 6). These two signals correspond to the registers that the CPU loads when starting 3DES and their values can be used in a directed system level test.

The trace shown in Fig. 6 was actually obtained by fixing the value of *des_key*. This could be easily achieved by adding a third subsequence to the cover statement:

```
cover_b1_08: cover property
…
 intersect
 (##1 (des.des_f.des_s1.b1[5:0] == 'h08)[=1])
intersect
((des_key[63:0] == 'hdeadbeef12345678)[*]));
```

The value of *des_in* needed to close coverage on the given block was des_in[63:32] == 'h000e6386 and des_in[31:0] == 'had365b75.

The same technique was applied to generate *des_in* values for the other 4 uncovered blocks. To avoid copy and paste and improve readability, a parametric SVA property *stage_and_value* was defined:

```
property state_and_value(stage, value);
(des_start && !des_stop
  ##1 !des_start && !des_stop [*16]
  ##1 !des_start && des_stop)
 intersect
 (##1 (stage == value)[=1])
intersect
((des_key[63:0] == 'hdeadbeef12345678)[*]);
endproperty
```

The above property was then instantiated for each uncovered block:

```
cover_b1_08: cover property (
  stage_and_value (
   des.des_f.des_s1.b1[5:0], 'h08));
…
cover_b3_17: cover property (
  stage_and_value (
   des.des_f.des_s3.b3[5:0], 'h17));
```

Similarly to the described example, the *stage* and *value* for each of the other uncovered blocks were derived from the witness of the corresponding IFV automatic deadcode assertion. The witness of each cover statement then provided the *des_in* value for the system level test.

In this case study we had no specific difficulty in working with the *cover* statements, which provided the desired results within just a few minutes of runtime. However, it is not uncommon when trying to cover a specific behavior to have to deal with a "fail" result. In such a case no trace is provided, as "fail" means that no trace exists. In order to make progress and understand why the behavior cannot be covered, one would relax parts of the behavior description to the point where a trace can be produced. For example, in the cover statements of this case study, one could start with removing one of the 3 sequences joined by the *intersect* operator. Or, if constraints were specified by means of *assume* statements, one or more constraints would be temporarily commented out. Once a trace is generated, it is examined to understand in what way it contradicts the desired behavior. The reason might be a mistake in the behavior description, or an over-restrictive set of environment constraints, or one might conclude that the specific behavior is indeed unreachable and use the information in the trace to understand why.

A. *Creating the Missing Tests and Verifying Coverage*

Once the formal tool was able to give us a pattern, it was simple to take an existing directed test and replace the input data with those produced by formal analysis and prove that the resulting calculation indeed covered the logic that we had targeted. For directed tests, we used a known good DES calculator (Linux OpenSSL command line tool) to calculate the expected result for comparison to the simulated result.

Given the new directed tests, we were able to run simulation with coverage collection turned on and confirm that the targeted blocks were indeed covered.

VI. CONCLUSIONS

Coverage closure is generally regarded as a resource intensive and time consuming task. It requires two main

activities: a) separating the uncovered target into reachable and unreachable; b) developing new test cases for the reachable targets. Formal verification can ease both tasks. However, applying formal verification has a cost in itself, in particular, when environment modeling by means of constraints is needed. In this paper we presented a case study in which we combined formal verification and existing simulation results, instead of environment modeling, to create the missing test cases. This way of applying formal verification was very cost-effective, while the effort required by simulation or formal alone would have been a) well in excess of the few hours required by the proposed technique and b) too high compared to the risk of bugs in the block. Without the proposed technique, we would have had to settle for a lower coverage.

The case study described is one of several successful applications of formal verification in a mostly simulation-driven verification methodology. During the IP8000 design project, formal verification complemented the simulation effort in the verification of critical blocks, protocols and algorithms. Formal techniques were also very successfully applied for post-silicon debugging and system level connectivity verification.

ACKNOWLEDGMENT

We thank Bin Ju and Joseph Hupcey III of Cadence, Viranjit Madan, Nihar Shah, Scott Asakawa and Toshi Morita of Ubicom and Dan Benua of the DVCon Technical Program Committee for their comments and valuable suggestions.

REFERENCES

[1] T Kropf, "Introduction to Formal Hardware Verification", Springer-Verlag, New York, 1999.

[2] S. Tasiran, K. Keutzer, "Coverage Metrics for Functional Validation of Hardware Designs", IEEE Design and Test of Computers, July 2001, pp. 36-45.

[3] U.S. Department of Commerce – NIST: National Institute of Standards and Technology "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher", Revised 19 May 2008 William C. Barker, http://csrc.nist.gov/publications/nistpubs/800-67/SP800-67.pdf

[4] U.S. Department of Commerce – NIST: National Institute of Standards and Technology, "Data Encryption Standard (DES)" Federal Information Processing Standards Publication, October 25, 1999, http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

[5] Ubicom IP8K Network Processor Family, http://www.ubicom.com/products/pdfs/IP8100_IP8200_Product_Brief_r12_PUB.pdf

[6] N. Kulshrestha, P. Chatterjee, S. Sridharan, M. Munishwar, "Automatic Coverage Closure Using Magellan", SNUG San Jose 2011 Proceedings.

[7] G. Faux, J. Müller, "Using Static Formal Analysis to improve Dynamic Code Coverage", CDNLive! EMEA.

[8] Cadence Incisive Enterprise Verifier-XL User Guide.

[9] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, "Handbook of Applied Cryptography", CRC Press, page 251.

[10] F. I. Haque, J. Michelson, K. A. Khan, "The Art of Verification with SystemVerilog Assertions", Verification Central.