Post-Silicon Debug Using Formal Verification Waypoints

C. Richard Ho, Michael Theobald, Brannon Batson, J.P. Grossman, Stanley C. Wang, Joseph Gagliardo, Martin M. Deneroff, Ron O. Dror, David E. Shaw*

D. E. Shaw Research, New York, NY 10036, USA

{Richard.Ho, Michael.Theobald, Brannon.Batson, JP.Grossman, Stan.Wang, Joe.Gagliardo, Marty.Deneroff, Ron.Dror, David.Shaw}@DEShawResearch.com

***Abstract*—Applying formal methods to assist in the post-silicon debugging of complex digital designs presents challenges that are distinct from those found in pre-silicon formal verification. In post-silicon debug, a set of observed events or conditions describes a failure scenario. The task is to identify a reasonably general set of input and hardware state conditions that inevitably produces that failure scenario. That set of conditions may be represented in the form of a counterexample to a desired property. Modern formal verification methods are especially adept at finding counterexamples to properties, and can often do so efficiently in large state spaces. This paper describes a method of assisting the discovery of counterexamples using user-hypothesized preconditions, or *waypoints*, of the failure. Each waypoint is an event that is believed to occur prior to the observed failure of the target property. By guiding formal analysis through a sequence of waypoints, the time required to find a counterexample of the target property can be significantly reduced. A specific case study is presented to illustrate the application and performance of our method using an actual example from the post-silicon debug of a 33-million–gate chip.**

I. Introduction

The post-silicon debug of functional errors in large, highly complex Application-Specific Integrated Circuits (ASICs) frequently requires extensive detective work to isolate symptoms and identify underlying causes. Lack of observability, long runtimes to reach the error state, and imprecise control of event timing make many post-silicon bug hunts tedious and time-consuming endeavors.

In this paper, we describe one such bug hunt involving the Anton ASIC [1], a 33-million–gate chip designed to accelerate molecular dynamics (MD) calculations. In this case, the ASIC exhibited erroneous behavior resulting in occasional memory corruption. The symptoms of the error (the *error signature*) were analyzed and a hypothesis of how the error occurred was formulated. This hypothesis involved certain complex corner-case conditions and particular event sequences. Extensive random simulation targeting the bug, however, did not succeed in validating this hypothesis. This was primarily a result of the fact that the bug appeared only in a specific, hard-to-reach hardware state whose occurrence was dependent on the precise timing of input stimuli.

The bug was eventually isolated and reproduced through a process of formal verification based on model checking [2]. In particular, we used an approach based on targeting sets of conditions called *waypoints*, which are hypothesized by the user to necessarily occur en route to the bug in question. The bug was found to lie beyond the practical reach of standard (bounded) model checking from a reset state, which could only complete exhaustive analysis to 65 cycles within a three-day time limit and a 32-GB memory limit. Using the method described here, however, the hypothesized cause of the bug was analyzed to generate waypoints, which were then targeted by model checking. Once an input sequence was found that led to a given waypoint, a state trace was generated, then used as the initialization sequence for model checking to the next waypoint or to the eventual error condition.

In this way, formal verification was guided to find the bug at a depth of 69 cycles from reset within ten hours of computation. Although the bug was only four cycles beyond the exhaustive analysis from reset, those additional cycles have high computational complexity, which would have made analysis using standard model checking impractical to complete within a reasonable amount of time. By using waypoints to reduce the amount of analysis needed to find the error trace, however, we were able to validate the hypothesized cause of the bug without a prohibitive expenditure of computational resources. This approach also allowed the analysis of conditions around the bug, and ultimately confirmed that the error would no longer occur after the design was corrected.

In the remainder of this paper, we discuss each of the major steps in our method, including (1) converting error symptoms into assertions, (2) finding the right level of logic to analyze so that the bug can be exhibited, (3) choosing the appropriate places in the design to abstract logic, (4) setting the necessary input constraints, and (5) finding the trace to the bug. We also present runtime data comparing standard model checking of the error assertion to guided model checking using waypoints.

* Correspondence to David.Shaw@DEShawResearch.com.
David E. Shaw is also with the Center for Computational Biology and Bioinformatics, Columbia University, New York, NY 10032.

II. Post-Silicon Formal Verification

Most complex ASICs require post-silicon debug. Much previous work has occurred in developing techniques to increase observability of the internal state in the fabricated design [3] as well as in techniques for automatically isolating erroneous logic [4]. One of the steps in post-silicon debug is to confirm that a hypothesis about an error can really account for the observed symptoms. The design team wants to know the exact hardware state and input sequence that would trigger the error.

Formal analysis, in particular model checking, has proved useful for finding the specific cause of a known error condition [5]. If the symptoms of the error can be represented as a property, then a counterexample (cex) to the property gives the sequence of stimuli and design-state transitions (a *trace*) that lead to the error.

Prior knowledge of the existence of such a counterexample sidesteps some of the difficulties of standard model checking on a property whose truth is not known for certain. This section discusses the optimizations that can be applied for post-silicon debug with formal verification. First, the selection of model-checking algorithms can be restricted to those optimized for finding counterexamples; second, the accuracy and generality of input constraints can be relaxed with information from the error signature; and third, model checking can be applied with waypoints to reduce the amount of analysis needed to find the trace.

A. *Model-Checking Algorithms*

Model checking is the mathematical process of determining whether a property holds true for a particular *model* of a system. In the context of this paper, a model is a representation of the hardware design. Model checking can produce either (1) a proof that the property holds in the model; or (2) a counterexample showing the input- and state-sequence that demonstrates a violation of the property within the model. In practice, a third result (undetermined) often occurs because the model checking algorithm runs out of time or memory before either of the definitive results can be obtained.

There are several algorithms for model checking [6], including explicit state enumeration, symbolic model checking with binary-decision diagrams, satisfiability-based model checking (also known as SAT), induction, interpolation, and variations and improvements of above algorithms (symmetry, abstraction refinement, etc.). For the purpose of finding a trace to an error signature, we can limit the choice of model-checking algorithm to those which are well-suited to finding counterexamples, such as SAT.

Finding long counterexamples is a key problem in model checking. Yang and Dill [7] use preconditions of a property to guide model checking. Ganai et al. [8] use manually and automatically-generated hints from a design description for difficult to reach coverage points. Bjesse and Kukula [9] exploit abstractions to generate long counterexamples. Wang et al. [10] have developed a technique that targets long bugs that have a regular pattern, which can be proved by induction.

B. *Accuracy of Input Constraints*

In pre-silicon verification, the accuracy of input constraints is a major factor in the success of formal verification. Without an accurate and complete set of input constraints, counterexamples may be found that utilize illegal input sequences. Analyzing such illegal counterexamples not only wastes time, but also prevents the possibility of a proof being found for the property until all illegal sequences that can produce a counterexample are excluded from analysis.

Fortunately, in post-silicon debug model checking, this stringent requirement can be relaxed. In particular, since model checking is typically deployed only if the trace of the error signature is difficult to obtain in simulation, it is often the case that even traces with some illegal stimuli can shed light on the conditions necessary to exercise the bug.

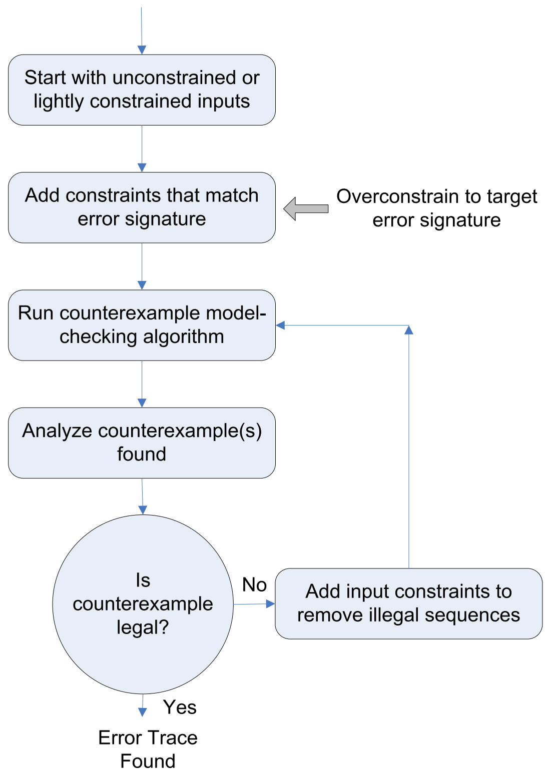

Start with unconstrained or lightly constrained inputs
Add constraints that match error signature
Overconstrain to target error signature
Run counterexample model-checking algorithm
Analyze counterexample(s) found
Is counterexample legal?
No
Add input constraints to remove illegal sequences
Yes
Error Trace Found

Figure 1. Setting input constraints

In addition, some aspects of the input sequence needed to exercise the bug may be known from the observed error; it is therefore possible to overconstrain the input space (relative to the full set of legal stimuli) to limit or “guide” the formal analysis. Consider the situation, for example, of a bug that only occurs on a read operation following a cache-miss. The input constraints can then be set so that analysis only occurs down paths that include a cache-miss followed by a read. Such over-constraining of inputs would not be appropriate in pre-silicon verification because it may mask bugs.

Some commercial SAT implementations do not actually limit formal analysis to legal input sequences, but do limit the traces presented to the user, so they may not observe a performance improvement from over-constraining. There is still a benefit, however, from the reduction of traces to be debugged, as the only traces to be considered are those where known input events occur. Noting these relaxed requirements, the methodology for post-silicon debug, shown in Fig. 1, becomes a mixture of over- and underconstraining. One starts with constraints that match the observed error signature. These constraints may limit the possible input sequences to a set that is much smaller that the full set of legal inputs. All other inputs should be lightly constrained so as not to prevent any legal input sequences. Additional input constraints can then be added to remove illegal sequences observed in counterexamples that are found.

C. *Formal Verification Waypoints*

When navigating to an unfamiliar destination, savvy travelers occasionally use *waypoints* (also known as *landmarks* or *guideposts*) to check that they are progressing on the right path towards their desired destination. A similar concept can be utilized when trying to find a trace to an error signature.

Yang and Dill [7] proposed using interesting or required preconditions (which they called "guideposts") of a property to assist explicit state-enumeration model checkers. These preconditions are events defined by engineers using knowledge about the design and the target property to provide sub-goals for formal analysis. Yang and Dill used guideposts to influence the order of state-space exploration so that paths which encountered more preconditions would be explored first.

In this work, we use similarly defined preconditions. Our use of the preconditions ("waypoints"), however, is to propel the initial state to be used in model checking deep into the state-space of the model. The steps for using waypoints to find a trace to an error signature are:

1. Identify one or more waypoints.
2. Order the waypoints *1, …, n*, with waypoint *n* as the error signature.
3. Starting with waypoint 1 and an initialization sequence that is just the reset sequence:
    a. Set the waypoint as the target property.
    b. Set the initialization sequence for model checker.
    c. Model check.
    d. If no trace is found, return to step (1) and identify additional waypoints prior to waypoint 1.
    e. If a trace is found, save the trace in the format to be used as initialization sequence in step (b) for model checking.
    f. Set the next waypoint as the target property.
    g. Return to step (c).

When one uses a trace to a waypoint as the initialization sequence for model checking the next waypoint, some state values of the model that the user believes are required to reach the target property get set up. Using waypoints to set up these intermediate states ensures that only reachable combinations of states are used, as opposed to setting up the state in some arbitrary way that may not be reachable from reset. Doing so reduces the scope of analysis at each step, as shown in Fig. 2.

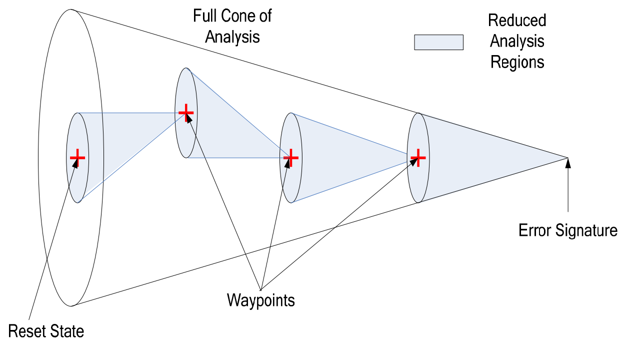


Figure 2. Waypoints reduce analysis region

Waypoints, deployed in this manner, are useful for targeting a particular property believed to be false, i.e. a property for which a counterexample exists. There is, however, no guarantee that the trace to an error signature will pass through any or all of the waypoints defined by a user. It is possible for a user to misunderstand an error signature and define waypoints in such a way as to prevent a model checker from finding a trace. This will become evident when a trace cannot be found to any one of the waypoints in the sequence *1, …, n* of waypoints within reasonable limits of time and memory. The only recourse when this happens is to re-examine the waypoints and alter them or their sequencing.

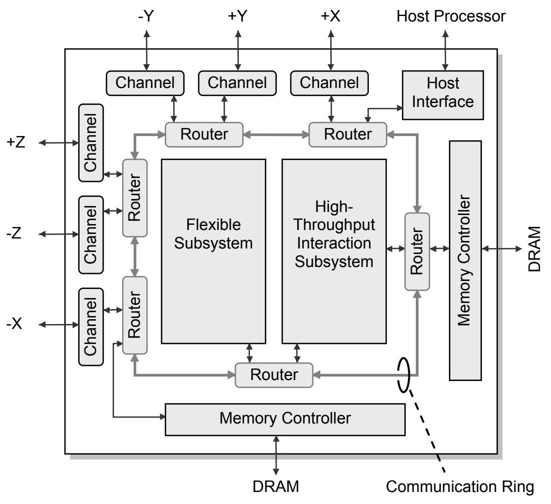


Figure 3. Anton ASIC Block Diagram

III. POST-SILICON DEBUG CASE STUDY

This section provides a case study of applications of formal verification with waypoints for the post-silicon debug of the Anton ASIC. We start with a brief overview of the Anton architecture. A more detailed explanation of the architecture and how it is used to perform MD computations can be found in [1].

A. The Anton ASIC

Anton is designed to accelerate MD computations, which model the motion of a collection of atoms according to Newton's laws of physics. An MD computation divides continuous time into a sequence of discrete *time steps*. Typically, each time step represents a few femtoseconds of physical time; Anton is intended to run MD computations for up to milliseconds of physical time (close to a trillion discrete time steps).

Anton achieves the speed required for computations of this scale through a combination of specialized hardware, high-bandwidth communication, and fine-grained parallelism. The Anton ASIC (block diagram shown in Fig. 3) consists of two main computational subsystems: the *high-throughput interaction subsystem* (HTIS) [11], which computes pairwise interactions, and the *flexible subsystem* [12], which contains a number of programmable processors. Two memory controllers are connected to off-chip dynamic random-access memory (DRAM). A host interface communicates with an external host processor used to control and monitor the ASIC, and six communication channels connect the ASIC to its neighbors in the three-dimensional torus network. These components communicate with one another by sending packets over a bidirectional on-chip communication ring, which consists of six identical routers connected in a loop.

The first Anton ASICs were fabricated in late 2007. In the bring-up process, various sample MD simulations were run. The successful runs are instructive, but during bring-up, it is the runs that exhibit problems that provide the most valuable (hardware and software) debug information. A run can encounter problems in two ways: (1) the run can produce an internal error; and (2) the MD simulation can behave in an unexpected manner, for example, if the energy of the MD system deviates substantially from the expected range. In each case, root-cause analysis can be undertaken using hardware and software instrumentation that reveals details about the operation of the hardware. In many cases, the embedded software of Anton required modifications, but in several cases hardware errors were indicated. None of the hardware errors were critical; each had an acceptable software workaround. Nevertheless, investigations were launched to find the root cause of each error, so as to ensure we completely understood the problem and to consider possible hardware fixes for future versions of the Anton ASIC.

One particular error found was described as follows (implementation-specific names have been replaced with functional descriptions for ease of understanding):

```
"Forces for packet 0 of
      <memory area 1> and <memory area 2>
are sometimes wrong on
      time step #447 of a <MD system> run.

It appears that the HTIS forces that are
addressed to
    packet 0 of <memory area 2>
were delivered to
    packet 0 of <memory area 1>."
```

This application-level error symptom translates to a memory corruption problem in the hardware. It says that occasionally (only on a certain time step of the MD run), some data (representing forces calculated by the HTIS) that should accumulate in one memory location ended up at the wrong address in memory.

B. Developing a Theory of the Error

Analysis of the communication patterns between **<memory area 1>** and **<memory area 2>** revealed that one explanation that would be consistent with the error was if a race condition arose between the *fill* and *evict* operations of the memory system.

Specifically, if an atomic accumulate-store data packet arrives at the memory controller such that it is supposed to evict **<memory area 1>** from the cache, the sequence of events that should occur is (Fig. 4):

1. Eviction: **<memory area 1>** written back to DRAM
2. Store: **<packet data>** is stored in cache line
3. Fill: **<memory area 2>** is fetched from DRAM and added to **<packet data>** in cache line

(The eviction and store operations are atomic)

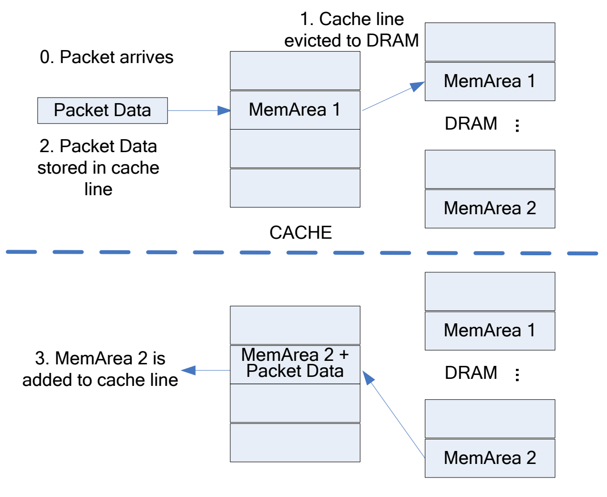
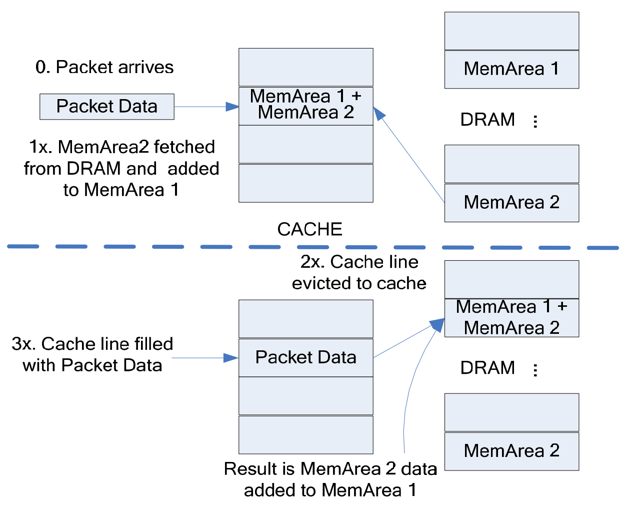


Figure 4. Correct memory operation



0. Packet arrives
Packet Data
MemArea 1 + MemArea 2
MemArea 1
DRAM
1x. MemArea2 fetched from DRAM and added to MemArea 1
MemArea 2
CACHE
2x. Cache line evicted to cache
MemArea 1 + MemArea 2
3x. Cache line filled with Packet Data
Packet Data
DRAM
MemArea 2
Result is MemArea 2 data added to MemArea 1

Figure 5. Race condition causes memory corruption

Now suppose that a race condition causes step 3 to occur first:

1x. Fill: **<memory area 2>** is fetched from DRAM and added to **<memory area 1>** in the cache line

2x. Eviction: incorrect **<memory area 1>** + **<memory area 2>** is written back to DRAM

3x. Store: **<packet data>** is stored in cache line

The result is that the forces that have been accumulating in **<memory area 2>** have been incorrectly added to **<memory area 1>**. Immediately after step 1x, all the forces that are intended to be summed at **<memory area 2>** have effectively been transferred to **<memory area 1>.** This incorrect update is then saved to DRAM in step 2x (Fig. 5).

This hypothesized cause of the error, among all the theories brainstormed by the engineering team, was the only one consistent with the observed symptoms. The next step was to identify how such a race condition might occur in the hardware implementation.

C. Identifying Error Mechanism in Implementation

With a conceptual theory of the error in hand, the next step was to match the steps of the theory to operations in the hardware implementation. This step requires detailed knowledge of the register-transfer level (RTL) description to identify the corresponding logic and operations to match the events in the conceptual theory of the error. Fig. 6 shows the relevant blocks in the memory controller of the Anton ASIC.

The error sequence starts with a *read-modify-write* (rmw) operation arriving at the memory controller and causing a cache *miss/evict* operation. This simply means that the memory line that the rmw-operation refers to is not currently stored in the cache and needs to be brought in from DRAM. To process the *miss/evict* operation, the control pipeline (CP) issues a *miss/evict* command to the data pipeline (DP) to save (back to DRAM) the current contents of the cache line that will be used for the operation. Simultaneously, it issues a *read request* to the bank controller (BC) to fetch the targeted memory line.

To prevent a race condition between the *miss/evict* and the *read* operations, the CP issues a 3-bit field (*dpstall*) with both operations. The FIFO to queue up the *miss/evict* operations is five levels deep, so three bits to hold *dpstall* is sufficient to prevent duplicate values. The *read* operation (in the BC) must wait for the corresponding *dpstall* value to be sent from the DP before it issues the *read* operation. For its part, the DP should only issue the corresponding *dpstall* value when it is complete with the *miss/evict* operation. In this way, the implementation is supposed to guarantee the order of operations shown in Fig. 4.

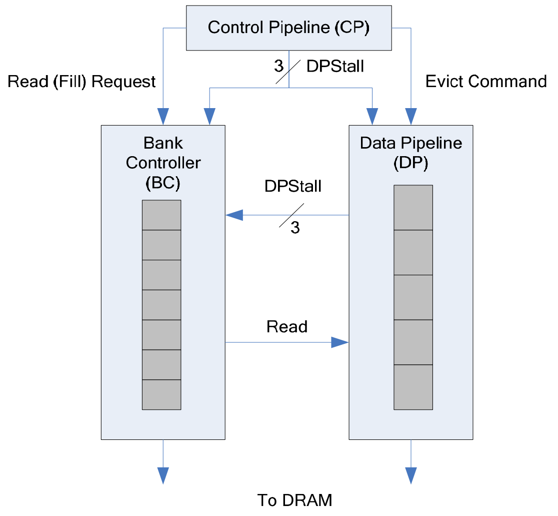


Figure 6. Memory controller blocks

The only event sequence that would allow the fill operation to overtake the evict operation would be if the *dpstall* signal should wrap and issue the same *dpstall* tag to two *miss/evict* operations. Then, when the first *miss/evict* operation completes and forwards its *dpstall* tag to the BC, both corresponding *read* commands are unblocked. This would enable the second *read* operation to get ahead of its corresponding *miss/evict* operation.

Hence, the error signature could result from the following sequence of events: (1) a single *dpstall* tag sent from the DP to the BC unblocks multiple read operation (Waypoint 1); (2) fill of cache line with particular *dpstall* tag (Waypoint 2); and (3) store of cache line with same *dpstall* tag, indicating that the fill operation has occurred before the evict operation (Error Signature). This sequence becomes the set of waypoints used to find the error signature.

D. Confirming Theory and Verifying Fix

Once an error mechanism in the hardware implementation is identified as a candidate root cause, it is necessary to confirm that the sequence of events in the mechanism occur as predicted and result in the error signature. In many cases, this can be accomplished in simulation with modifications to existing test environments. In other cases, ours included, simulation cannot activate the full sequence of events hypothesized in the error mechanism. Formal verification (model checking) provides an alternative method to do so. The necessary steps follow the standard formal verification methodology:

1. Write assertions to detect error mechanism. Our assertions were created with the SystemVerilog Assertion (SVA) language as well as the Open Verification Library (OVL). The three main assertions written covered the two waypoints identified and the final error signature.

2. Determine correct cone of analysis. The cone of analysis is the logic within the transitive fanin of the target property. Ideally, the entire design is analyzed while looking for the error. This is not practical in any but the smallest designs. Choosing a subset of the design is common practice for FV; on the other hand, choosing a design subset that is too small leads to greater difficulties in defining the input constraints for FV. Also, it is necessary to ensure that all the logic involved in the error is captured within the design subset. For our particular case study, the logic needs to include the CP, DP and BC blocks of the memory controller. It turns out that these blocks encapsulate most of the logic on the memory controller subsystem. In all, this logic is estimated be about 800k gates.

3. Abstract unnecessary logic. This step in the standard FV methodology identifies logic that can be simplified to reduce the analysis region. Common abstractions include counters and deep FIFOs. These can usually be reduced to versions that have abstract values that represent groups of concrete values, such as a counter that only has values for *empty*, *partially full* and *full*. If the value of the counter between when it is *empty* and when it is *full* is irrelevant to other events, this simplification reduces the state space of analysis enormously. Abstractions of this sort over-approximate the state space of the design and are safe for proofs of properties. For counterexamples, however, they frequently lead to false traces that are not possible in the full logic, or to traces that are inconsistent with the events that the designer is expecting to see. Hence, when searching for an error signature, abstractions must be made with even more care, and avoided if possible. The example in our case study was performed with no abstractions.

4. Set input constraints. The method of setting input constraints, when none are present, was described in section II.B. For our error signature, a full set of input constraints for the analysis region (the memory controller) was available from pre-silicon verification.

5. Find trace using waypoints. This step is simply the iterative targeting of successive waypoints, using the trace to the previous waypoint as the initialization sequence. For the first waypoint, the initialization sequence is the normal reset sequence for the design. The detailed performance numbers for the waypoints and error signature of our case study are shown in section IV.

Once the error signature is found and the theory of the error is confirmed, the next step is usually to fix the error and validate that it has been removed from the design. In our case, the underlying cause of the error was that the *dpstall* counter was missing a stall term that would prevent it from wrapping and was thus allowing multiple stall releases. Once fixed, the RTL was re-checked, with each waypoint and the error signature targeted by FV. We were able to quickly obtain proofs that the first two waypoints were unreachable. Although the complexity of the design prevented us from obtaining a full proof that the error signature is unreachable, the fact that the necessary preconditions for the error were no longer possible gave us confidence that the fixed RTL no longer has the error.

IV. Comparing Direct and Indirect Formal Analysis

The case study in section III highlights the difference between model checking from a reset state versus model checking using waypoints. The complexity of the target property is just beyond the reach of bounded model checking, using standard commercially available model checkers. Applying waypoints, however, narrows the scope of analysis sufficiently to reach the target (Table 1).

Table 1. Direct FV of Target vs. FV using Waypoints

Target	Initialization sequence	Mem.	Time (CPU sec)	Analysis depth	CEX found?
Error Signature	Reset state	32 GB	> 260000 (3 days)	65	no
Waypoint 1	Reset state	9 GB	2135	28	yes
Waypoint 2	Waypoint 1 (28 cycles)	9 GB	32637	40 (68 from reset)	yes
Error Signature	Waypoint 1 (28 cycles)	32 GB	> 260000 (3 days)	40 (68 from reset)	no
Error Signature	Waypoint 2 (68 cycles)	9 GB	1250	1 (69 from reset)	yes

Table 1 shows that the error signature is only a few cycles away from the achieved analysis depth of the FV run from reset (as shown in row 1). What is not shown in this table is that the analysis depth of the run from reset had been at 65 for approximately 50% of the runtime and appeared incapable of making further progress. It is common for bounded model checkers to hit an analysis depth limit, beyond which it is impractical to continue within a reasonable time.

Also compare row 4 ("Error Signature target from Waypoint 1") against row 5 ("Error Signature target from Waypoint 2"). Row 5 shows that only one additional cycle of analysis was needed from Waypoint 2 to find the error signature; but that single cycle required a large amount of analysis. It is easy to fall into the trap of thinking that with only a few cycles of analysis needed to reach the next target property, it is simply a matter of providing a small amount of additional time or memory. One characteristic of SAT-based model checking, however, is that the amount of CPU time and memory required to complete each additional cycle of analysis has little correlation to the amount of CPU time required for any previous cycle; it is dependent only on the model and the hardware state. Exponential explosion in time or memory frequently occur at large analysis depths. In this case, this single cycle of analysis pushed the target assertion out beyond the reach of analysis from Waypoint 1.

V. Summary

This paper describes a method of finding the root cause of an error signature, which is particularly valuable in post-silicon debug, using model checking of multiple waypoints to reduce the scope of formal analysis. The methodology includes guidelines for selecting the type of model-checking engine to use, a procedure for selecting input constraints that utilizes known information from the error signature, and a procedure for identifying waypoints and using them to find a path to the error signature from a reset state of the design. The method was illustrated with a detailed case study from the post-silicon debugging of the Anton ASIC, providing examples for each step of the method. Finally, performance numbers were presented for an example in which the method of using waypoints is able to discover a path to an error signature that would not be found within a practically feasible amount of time using a standard formal verification methodology. (The error in question, which was handled using a software workaround, was fixed in a subsequent version of the Anton chip.)

Acknowledgment

We would like to acknowledge and thank the engineers from our formal verification tool vendors, Jasper Design Automation and Mentor Graphics (0-In Functional Verification), who provided invaluable support throughout our project.

References

[1] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J.P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles and S. C. Wang, "Anton: A special-purpose machine for molecular dynamics simulation," in Proc. 34th International Symposium on Computer Architecture (ISCA '07), San Diego, CA, June 9–13, 2007, pp. 1–12.

[2] E. M. Clarke, O. Grumberg, D. A. Peled, Model Checking. Cambridge: MIT Press, 1999.

[3] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in Proc. 43rd Design Automation Conference (DAC '06), San Francisco, CA, July 2006.

[4] S. B. Park and S. Mitra, "IFRA: Instruction footprint recording and analysis for post-silicon bug localization in processors," in Proc. 45th Design Automation Conference (DAC '08), Anaheim, CA, June 2008.

[5] J. Kumar, C. Ahlschlager, P. Isberg, "Post-silicon verification methodology on Sun's UltraSparc T2," in High Level Design Validation and Test Workshop (HLDVT '07), Irvine, CA, Nov 7–9, 2007.

[6] M. R. Prasad, A. Biere, A. Gupta, "A survey of recent advances in SAT-based formal verification," in International Journal on Software Tools for Technology Transfer (STTT), 2005.

[7] C. H. Yang and D. L. Dill, "Validation with guided search of the state space," in Proc. 35th Design Automation Conference (DAC '98), San Francisco, CA, June 1998.

[8] M. Ganai, P. Yalagandula, A. Aziz, A. Kuehlmann, and V. Singhal, "SIVA: A system for coverage-directed state space search," in Journal of Electronic Testing: Theory and Applications, February 2001.

[9] P. Bjesse and J. Kukula, "Using counter example guided abstraction refinement to find complex bugs," in Design, Automation and Test in Europe (DATE '04), March 2004, pp.10156–10161.

[10] C. Wang, A. Gupta, and F. Ivancic, "Induction in CEGAR for detecting counterexamples," In Proc. International Conference on Formal Methods in Computer Aided Design (FMCAD), November 2007.

[11] R. H. Larson, J. K. Salmon, R. O. Dror, M. M. Deneroff, R. C. Young, J.P. Grossman, Y. Shan, J. L. Klepeis and D. E. Shaw, "High-throughput pairwise point interactions in Anton, a specialized machine for molecular dynamics simulation," in Proc. 14th International Symposium on High-Performance Computer Architecture (HPCA '08), Salt Lake City, UT, Feb. 16–20, 2008, pp. 331–342.

[12] J. S. Kuskin, C. Young, J.P. Grossman, B. Batson, M. M. Deneroff, R. O. Dror and D. E. Shaw, "Incorporating flexibility in Anton, a specialized machine for molecular dynamics simulation," in Proc. 14th International Symposium on High-Performance Computer Architecture (HPCA '08), Salt Lake City, UT, Feb. 16–20, 2008, pp. 343–354.