Synopsys Users Group

Use of Verification Point Tools

Michael Thompson, Thomas Zboril

Huawei Technologies Canada

Ottawa, Ontario, Canada

http://www.huawei.ca

ABSTRACT

A "point tool" is defined as any tool that is used at one or more specific short intervals (points) in the ASIC/SoC/IC/FPGA development flow. This is in contrast to other tools, such as a SystemVerilog simulator, that are used throughout the entire development flow. Synopsys offers a selection of such "point-tools" for Verification, three of which are discussed: ***XPROP****,* ***FCA*** *and* ***Certitude****. This paper and its associated presentation are intended to assist the reader's understanding of when their process may benefit from the use of these tools.*

Table of Contents

Table of Figures

1. Introduction

This paper attempts to explain how to get the best return on your investment in the use of XPROP, FCA and Certitude. Unlike other tools, such as VCS, these tools are not used throughout the development process. This is partially due to their intentionally limited scope.

This paper is not intended to be a tutorial in the use of these point-tools. All have been the topic of one or more SNUG papers [2][3][4][6] and at least one figures prominately in a Master's Thesis [5]. In addition, all have support and user documentation available from Synopsys. Nevertheless, it is reasonable to assume that not all readers are familiar with all of these tools, so a very quick overview is offered here.

1.1.1 XPROP

XPROP is a simulation tool that adds "X-pessimism" to RTL simulations. It is useful for finding reset and initialization issues that do not always manifest themselves in RTL simulations, but do exist at the gate level.

1.1.2 FCA

Formal Coverage Analyzer is a static tool that looks for unreachable code in the RTL and creates code coverage exclusions for the unreachable code.

1.1.3 Certitude

Certitude is a fault-injection tool that can be used to measure the quality of your Verification Environment. It works by injecting faults into your RTL and determining whether or not the Verification Environment is capable of detecting them.

1.2 Goals of this Paper

The reader will be forgiven if he or she assumes that the goal of this paper is to help the reader determine whether or not to use XPROP, FCA and Certitude. This is not the goal. Development teams are encouraged to work with their local Synopsys support team to determine whether and which of these tools fit with your process. Rather, the goal of this paper is to assist the reader to determine the best way to apply each of these tools, and to help determine the return on investment of each.

As it turns out, "the best way" to use these point-tools primarily translates to "when" to use them. Using any of these tools too early can be a waste of time, while using any of these tools too late may mean unnecessary delays to your schedule. In order to discuss when a tool should be used, we introduce the concept of a generic development process, and use it to define where each of these tools fit in that process. While it is a given that the process used by any one team will be different than the generic process discussed here, it is expected that any real process can be easily mapped to our generic process in a way that allows us to achieve our goal of explaining when to use the point-tools in question.

2. A Generic Development Process

For the purposes of this discussion, a "development process" shall mean whatever process (or flow) is used to develop an ASIC/SoC/IC/FPGA (hereafter referred to simply as an"ASIC"). These processes range from ad-hoc, with only a modest amount of program management, to rigorous, with detailed compliance checks of tightly defined milestones. Due to the increasing logic complexities of virtually all ASIC developments, development teams are adding increasing amounts of rigor to the processes they follow, and completely ad-hoc processes are virtually unheard of today. Another trend is the ever increasing use of agile techniques. The concept and application of a typical development process is of interest to us here as it helps to define the concept of a point tool, and more importantly, how and when they can be best applied.

Near the turn of the century it was possible to discuss the concept of a “typical” process, such as the one detailed by Doerre and Lackey in 2002[1]. Fifteen years later it is difficult to define a “typical” process due to the wide range of devices being developed. To facilitate discussion, we will define a generic process, illustrated in Figure 1. This generic process is loosely based on Figure 4(b) of [1], with increased emphasis on front end development and excluding everything after the “Release to Final Layout” stage (since the point-tools in question are not applicable after Release to Layout).

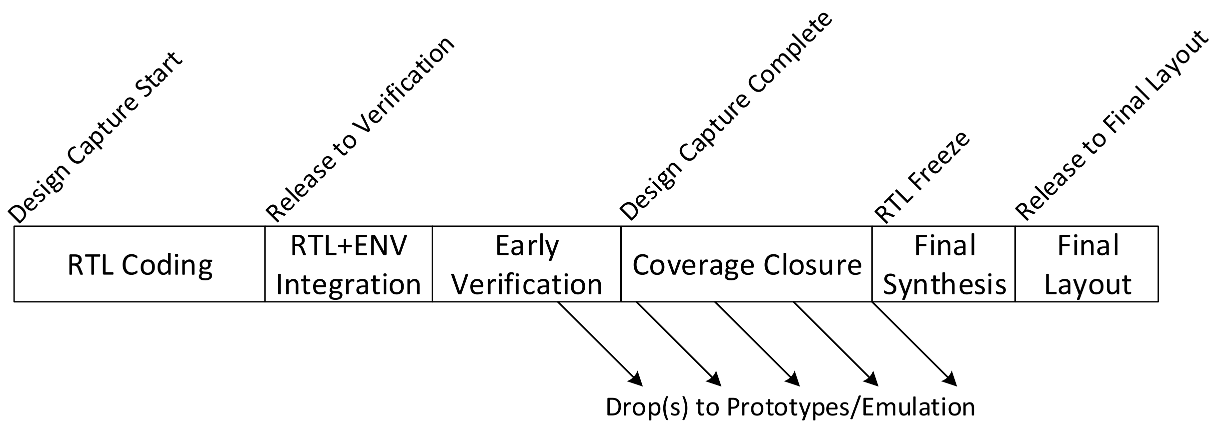

Figure 1 : Generic Development Process

This generic process is deliberately defined in such a way that it can be easily mapped to the specific process followed by virtually any ASIC team. The important milestones of the process in Figure 1 are:

- **Design Capture Start**: In the (rare) case where the DUT is all new, this is when the first coding begins. Perhaps more typically, this would be the point where the Design team starts to update code that is re-used from a previous project.
- **Release to Verification**: is the point where the RTL is first integrated into the verification environment. This may be after most or all of the RTL is captured, as in a traditional waterfall development process, or as soon as possible after coding starts, as in an agile development process.
- **Design Capture Complete**: is the point where all RTL code is captured. All changes after this point are expected to be for timing, optimization and/or functional fixes.
- **RTL Freeze**: is the point where all front-end design and functional verification is complete.
- **Release to Final Layout**: the netlist generated from the code at RTL Freeze is handed off to physical implementation. After this point, the front end design cannot change without putting the schedule at risk.

Between these milestones in our generic process are activities that will be familiar to all readers. The details of the execution of these activities are dependent on the specific process followed and usually look something like this:

- Between **Release to Verification** and **Design Capture Complete** is a very busy time. The RTL and Verification Environment are integrated for the first time, and the first bugs are flushed out. After integration it is typical for coding of both the RTL and Verification Environment to continue in parallel. New code is integrated, verified and regressed as defined in the process.

- At some point close to **Design Capture Complete**, the RTL may be released to Emulation or prototyping. It is typical that there will be multiple code "drops" to Emulation/prototyping before Release to Final Layout.
- Between **Design Capture Complete** and **RTL Freeze** is where the verification team really earns their money. The primary objective here is to achieve whatever completion metrics are used by the specific progress. Typically, these are some combination of test-case completion, code coverage and functional coverage.
- **RTL Freeze** to **Release to Final Layout** is typically a short interval. In the ideal case, all front-end work is completed by RTL Freeze and so the only activities here are final synthesis and the other activities required to generate the final netlist for the physical design team.

2.1 Pressure Points in the Process

Any point in the process that is identified using the phrase "Release to" is generally a point where the team is under pressure. This is either because the device is being handed off to a new team, or because a set of time-critical tasks must be completed before the project can move onto the next phase. Typically, point-tools are introduced by EDA vendors to address these pressure points and teams will deploy point tools just ahead of a pressure point in an effort to reduce risk and effort.

In this paper we will argue that this can be a good strategy. In some cases, it can be a better strategy to think of the use of these point-tools in the context of the end-to-end process.

2.2 What *Not* to Do

An understandable first instinct is to apply each of these tools near the end of the process. The reason this may be your first approach is because these tools present themselves as final-sign off criteria: did you find all the x-propagation scenarios; have the impacts of dead-code been factored into the code coverage results; can your Verification Environment find all faults. Since the final answer to these questions cannot come until the final regressions have been run on the final version of the RTL, it is natural to run these tools as close as possible to RTL Freeze. This is a mistake for several reasons:

- In all cases, tool set-up time is non-zero. If you wait until the end, you risk schedule delays.
- Analysis of the results of these tools also involves significant effort. Once again, waiting too long has schedule risks.

In our "shift left" world another tendency is to try to do work as early as possible in the process. As stated previously, one of the goals of this paper is to provide guidance on when is the best (read: earliest) point in the process to apply each of these point tools.

3. XPROP

Of the three point-tools discussed in this paper, XPROP is the most mature and the one whose value is easiest to recognize. XPROP was first introduced to VCS in 2011. At that time, users reported significant set-up times and significantly increased run-time costs [1]. Our experience using XPROP began with VCS2014. It has been our experience that set-up is now almost trivial and the run-time impact is on the order of 15% to 20%. So the cost of deploying XPROP is negligible and the cost of using it is manageable. It should be understood that currently XPROP is a separately licensed product so the true cost of using XPROP is not in its set-up or run-times, but in its license management.

Evans, Yam and Forward in [3] suggested that XPROP could be used to reduce and/or eliminate the need to gate-level simulations. They also suggest a flow whereby XPROP is applied twice throughout the process: once just after Release to Verification and a second time as a sign-off criteria at Release to Final Layout. With respect to our colleagues, our recommendations differ slightly. At this time, it does not appear that XPROP can be a replacement for gate simulations. Both of the merging functions used by XPROP are semantically different from what gate models provide [2]. Rather, the real value in XPROP is finding most of your X-propagation issue early in the design cycle, well before a gate-model is available. It has been our experience that X-propagation issues uncovered in gate-simulations are significantly reduced when XPROP simulations have been previously deployed. Given that gate simulations occur at one of the most significant pressure points in the process (often between RTL Freeze and Release to Final Layout), this is significant. We have also found that XPROP significantly reduces bring-up issues for prototyping and emulation. This makes sense when you consider that synthesis for prototyping and emulation generates a gate-netlist, albeit with a different layout target. Both prototyping and emulation are becoming more prominent is ASIC flows and between Release to Verification and RTL Freeze it is not uncommon to have multiple releases to prototyping/emulation. In these situations, it is useful to run XPROP before each prototyping/emulation release. Given the low cost of XPROP simulations, we recommend running XPROP at regular intervals as a matter of course. This will minimize the probability of X-propagation issues in your prototypes/emulation netlists and de-risk the creation of the final netlist for the Release to Final Layout milestone. As an added benefit, it also provides continuous feedback about code constructs that create X-propagation issues to the design team.

3.1 XPROP Recommendations

Our specific recommendations for the use of XPROP are:

- We recommend that XPROP be used in virtually all ASIC development process. The set-up costs are now negligible, the run-time costs are manageable and the benefits in reduced debug time are significant.
- Start running XPROP near Design Capture Complete. Running the tool earlier yields too many false negatives, largely due to incomplete RTL code.
- If your flow involves prototyping and/or emulation, XPROP should be used prior to the first 'drop' to prototype/emulation.
- A full regression using XPROP should be run on the RTL at regular intervals. Depending on the size of your device and the number of XPROP licenses at your disposal, this can be once per week and not less than once per two weeks. Due to the increased run-times (~15% to 20%), we recommend running XPROP regression over the weekend.
- XPROP should not be used as a substitute for gate-level simulations.

4. FCA

Formal Coverage Analyzer (FCA) was first introduced by Synopsys as a production tool in 2015. FCA is promoted as a tool that will identify dead-code in your RTL, and automatically generate exclusion files that can used by Synopsys tools to remove the dead-code from the code coverage report.

In our experience, discussions about the practically of FCA have centered on three topics: how to use the exclusion data produced by FCA, the cost/benefit of the tool and the set-up/effort required to obtain exclusion data. We will discuss these topics separately in the next three sub-sections.

4.1 Use of Exclusion Data

It is tempting to apply the exclusion data generated by FCA directly without further consideration. Doing so will generally produce a small, yet satisfying increase in your code coverage results without significant effort. We view this as very risky behavior. It is important to understand that FCA uses static (formal) methods to analyze the RTL. As such, it is susceptible to the two well-known issues in static verification[1]:

1. FCA relies on the user to provide the proper input constraints to the RTL. If these are not correct, then the generated coverage exclusions cannot be either.
2. Like all static tools, it is sometimes impossible for FCA to generate an unbounded proof.

For the purposes of this discussion, issue #2 is not a concern since FCA will not generate an exclusion unless it can generate and unbounded proof that a segment of code is, in fact, unreachable or dead-code. The problem that concerns us here is issue #1. Using incorrect constraints could result in the tool incorrectly identifying dead-code. For this reason, we strongly recommend that both the input constraints and the output exclusions should be independently reviewed and that a human with designer-level knowledge of the code signs-off on all code classified as dead-code by FCA before the exclusions are used.

A third issue is that FCA generates exclusion data on a per-instance basis, rather than a per-module basis. In some cases it can be difficult to review the exclusions on RTL code with many instances of the same module, unless each instance gets the same coverage (a highly unusual outcome in real DUTs). This situation motivated one of us (Zboril) to develop a script that cloned all instance coverage into module coverage, which greatly eased the effort reviewed to review code coverage holes. We would be very pleased to see such a feature supported directly by FCA.

Reviewing the output exclusions generated by FCA is critically important. We have encountered situations in which we were confident that a segment of code inputs was tied off, only to learn later that the tool did not recognize these as constraints. Whether this is user or tool error is immaterial: to guarantee accurate results, a review is always necessary.

4.2 Cost of Using FCA

The recommendations for these reviews should cause the reader to think about the cost/benefit of using FCA. Assuming that code-coverage closure is an important part of a development flow, using FCA is beneficial only if the cost of generating the exclusions with the tool is less than the cost of generating the exclusions manually. In making this determination, a development team must consider:

i. The set-up and run-time effort of using FCA. This is discussed in the next sub-section.
ii. The effort required to review the input constraints used by FCA to generate the exclusions.
iii. The effort required to review the actual exclusions generated by the tool.
iv. The effort required to manually review the RTL and manually generate the exclusions without the aid of FCA.

Clearly, FCA is only going to be useful if the cost of (i)+(ii)+(iii) is less than the cost of (iv). In our experience, there is considerable disagreement within the team about the relative cost of (iii) vs. (iv). Some have suggested that a competent engineer should be able to quickly identify dead-code in the RTL, particularly if it is code that they wrote. Others have suggested that the unambiguous identification of dead code is difficult, even for the most competent and experienced engineers. One of the major reasons for this is that it is extremely unlikely that all of the code in question will be written by a single person, and the developers of the original code may not be available for a review. Another reason is that code coverage closure often occurs months after the code was first written, so

[1] This statement should not be construed as a criticism of formal verification. All verification technologies have short-comings – it is important to understand these so that the best mix of verification technologies can be applied.

even the original developer may be unfamiliar with the code.

The authors make their living identifying functional issues in RTL code written by extremely experienced and competent engineers. It has been our experience that the cost of reviewing FCA generated exclusions is less than the cost of reviewing RTL to manually generate exclusions.

4.3 FCA Set-up

It has been our experience that the FCA does a good job for small and moderately sized DUTs. In these cases, the set-up is straight forward. Run-times are reasonable and you can expect to have a completed set of exclusions available for review and use within 24 hours. FCA also has a handy feature that allows for the generation of useable exclusions from a partial analysis, so reviews can be done concurrently with FCA runs.

For larger DUTs, it is required to use a divide-and-conquer approach in order to achieve useful results in a timely manner. This can significantly add to effort required to use FCA, so care should be taken when applying the tool to RTL blocks that synthesize to more than approximately 250Kflops.

4.4 When to Use FCA

It is typical to review code-coverage holes late in the process, in the last few weeks prior to Release to Final Layout. At this point the verification team should have high coverage and the question is whether or not the remaining holes are a significant risk. The problem with this approach is that it is very close to one of the most critical milestones of the project. At this point, the team is very busy and this can make it difficult to spend the time required for meaningful reviews of code-coverage holes. An unexpected benefit of using FCA is that since it can identify dead-code independent of the simulation results, there is no need to wait until late in the process. FCA can be applied any time after Design Capture Complete. This will reduce the workload required for reviewing coverage holes close to Release to Final Layout. Another benefit is that applied earlier, FCA provides opportunity for RTL designers to remove dead code and identify code that is intended to be "live", but really isn't.

Teams should keep in mind that the RTL code-base can be expected to change between Design Capture Complete and Release to Final Layout, so a final FCA run should be done as close to Release to Final Layout as possible. Since all of the set-up required will already be in place, the cost of this is simply machine time.

4.5 FCA Recommendations

Our specific recommendations for the use of FCA are:

- FCA is a fit for those teams that use code-coverage as a primary sign-off criterion. If your process does not target specific code-coverage goals including a detailed review of code coverage results, then FCA will not provide added benefit.[2]
- FCA input constraints should be reviewed as part of the code-coverage sign-off criteria.
- All code identified as dead-code by FCA should be reviewed to double-check that the code is eligible for exclusion from code-coverage.
- FCA should be applied as soon as possible after Design Capture Complete, and again on the final RTL that is used as the source code for the netlist used at Release to Final Layout.

[2] The authors seriously question the validity of any development process that does not use code-coverage, but we are aware that not all developments teams do.

5. Certitude

Most verification tools and processes focus on the quality of the DUT. Certitude is interesting in that it focuses on the quality of the Verification Environment. Certitude also provides new metrics for verification completeness that are independent from all others used in typical development projects.

5.1 What is Certitude?

As previously stated, it is not a goal of this paper to provide a tutorial on the use of tools. Having said that, the authors suspect that Certitude is the least known of the three point tools discussed in this paper. Therefore, some amount of introduction is warranted in this case. Keep in mind that a detailed review of Certitude is well beyond the scope of this paper, so what follows is a very superficial introduction.

In essence, Certitude works by inducing structural faults (or mutations, as in [4]) in the RTL code[3] and running your regression to determine if your Verification Environment can detect them. For example, a statement in the RTL **a = b || c;** could be mutated to **a = b && c;**. Certitude then runs your regression to determine whether the verification environment can detect the fault (mutation). Non-detected faults are assumed to be indicative of a deficiency in the Verification Environment.

Below are two simplified illustrations of the types of faults (mutations) that Certitude injects into the RTL. It is important to note that this is *not* how Certitude actually works, but it does illustrate the effect of a Certitude-injected fault on your RTL code.

5.1.1 Faults on Module I/O

The code snippet below, left, shows a trivial module port list. The snippet on the right shows how an artificial 'stuck-at' type fault could be added to the request_1 input of the module. Certitude has the ability to add stuck-at faults on all module I/O for the entire DUT. Both stuck-at-0 and stuck-at-1 faults can be generated. These faults are assumed to have impacts on the function of the DUT and it is expected that a thorough and complete Verification Environment would be able to detect them.

```
module simpleton_arbiter (
  input  clk;
  input  reset_n;
  input  request_1;
  input  request_2;
  output grant_1;
  output grant_2;
);
```

```
module simpleton_arbiter (
  input  clk;
  input  reset_n;
  input  request_1;
  input  request_2;
  output grant_1;
  output grant_2;
);

initial force request_1 = 1'b0;
```

Figure 2 : Illustration of a Certitude 'Stuck At' Fault on an Input Port

[3] Don't panic. ☺ This is done by instrumenting a snap-shot of the RTL. Your actual RTL code is not touched.

5.1.2 Faults on Reset

The example below is somewhat more interesting:

```
always @(posedge clk) begin

  if (!reset_n) begin
    grant_1 <= 1'b0;
  end
  else begin
    grant_1 <= rdy_1&&msk_1
  end

end
```

```
always @(posedge clk) begin

  if (1'b1) begin
    grant_1 <= 1'b0;
  end
  else begin
    grant_1 <= rdy_1&&msk_1
  end

end
```

Figure 3 : Reset Fault

The code snippet on the right shows a fault has been injected such that the sequential block can never come out of reset. Certitude can inject such 'never comes out of reset' faults as shown into each sequential block of the DUT. It is important to note that the reset faults are added to each sequential block individually, not all at once. The idea here is that we want to know that if one-and-only-one sequential block does not actually do anything that the Verification Environment will detect it. In a typical-sized DUT, this can be many hundreds faults.

5.2 Using Certitude-injected Faults to Measure Verification Quality

After a Certitude regression, Certitude will assign a status to faults as one or more of the following: not-activated, activated, propagated and detected (or not). A fault is **activated** when a simulation causes the result of a fault to be evaluated. Looking again at Figure 2, it is expected that at least once during a regression, the Verification Environment will attempt to drive stimulus into the DUT that results in request_1 being asserted. Certitude will run your regression to determine if this is, in fact, the case. Assuming that it is, Certitude assigns the fault to the activated status. Of course, this is not sufficient for proper verification of the DUT. Consider the case in which the Environment is not checking request_1 directly. In this situation the Environment needs to drive additional stimulus into the DUT that will cause a result from the fault to be seen on an output that is observed. Certitude calls this fault **propagation**. Lastly, the Environment must be sensitive to the fact that the resulting output will be incorrect due to the injected fault. This is known as fault **detection**.

In this way, Certitude provides two easy to understand and track metrics: a finite list of faults that should be detected in your Environment and a status (activated, propagated, detected) to help you understand what your Environment is (or is not) capable of, with respect to those faults.

This finite list of faults produced by Certitude is its key value proposition. Certitude will analyze your RTL and generate a list of faults that the tool can inject into the code. This list can be used as a completion metric, just as functional coverage is. An important thing to note here is that Certitude's list of faults is completely independent of the implementation or quality of the Verification Environment.

5.3 Cost of Using Certitude

Unlike the previous two tools we have discussed, the cost, in terms of time and effort, of using Certitude is significant. The cost has three components: setup, fault-inserted regression and results analysis. Each will be discussed in turn.

5.3.1 Certitude Setup

Even the most casual reader of this paper will have recognized that the number of faults that can be created by Certitude can be very large. For example, a typical DUT has thousands of module ports that can have stuck-at-0 and stuck-at-1 fault variants. In addition, there are many other types of faults that Certitude can inject in the code. For a reasonable sized DUT, the total number of injected faults can number in the many tens-of-thousands.

Fortunately, the tool supports a classification system for faults that allows development teams to focus on specific types of faults that are deemed to be of high value. It is strongly recommend that development teams work closely with their Synopsys Application Consultant or Corporate Applications Engineer to determine which types of Certitude faults will be most beneficial for their specific project.

5.3.2 Fault-Inserted Regression

Certitude will run the tests in your regression as many times as necessary to determine whether or not your Verification Environment can detect all faults. In the worst case, this can mean that all tests will be run number-of-fault times. Of course, this worst case can only be realized if your regression is capable of both activating and propagating all injected faults, but incapable of detecting any faults. This is a highly unlikely scenario and Certitude applies knowledge gained during the fault-insertion stage to greatly minimize the number of tests it needs to run. Nevertheless, you can expect a full Certitude regression to run more than 100 times longer than a typical regression.

5.3.3 Certitude Results Analysis

The most significant cost of using Certitude is the review of regression results. Given that the faults are added directly to the RTL code itself, a proper understanding of the results can only be gained with Designer-level knowledge of the RTL. The example below is a somewhat contrived example that illustrates this point.

```
…
parameter THREESTAGE = 1;
…

always @(posedge clk) begin
  if (!reset_n) begin
    s1  <= 1'b0;
    s2  <= 1'b0;
    out <= 1'b0;
  end
  else begin
    s1 <= in;
    s2 <= s1;
    if(THREESTAGE) begin
      out <= s2;
    else
      out <= s1;
    end
  end
end
```

```
…
parameter THREESTAGE = 1;
…

always @(posedge clk) begin
  if (!reset_n) begin
    s1  <= 1'b0;
    s2  <= 1'b0;
    out <= 1'b0;
  end
  else begin
    s1 <= in;
    s2 <= s1;
    if(1'b0) begin
      out <= s2;
    else
      out <= s1;
    end
  end
end
```

Figure 4 : Is this Injected Fault a Functional Bug?

The snippet above will propagate "in" to "out" in either two or three clocks, as determined by the value of the parameter THREESTAGE. By default, THREESTAGE is set to 1. Let's assume that the top-level of the DUT that uses this RTL has set the parameter THREESTAGE to 0 and the Verification Environment has a check for this, such as *assert property (@posedge clk) in ##2 out;*. However, the fault shown does not change the behavior of the RTL from what is intended and Certitude would report this as an undetected fault. Our experience has shown that there are a significant number of faults like this in every DUT, and the time-and-effort to review non-detected faults to determine if they can be waived is significant.

5.4 When to Use Certitude

The significant time-and-effort cost of Certitude should be a primary consideration when determining whether, when and how to deploy this point-tool on your project. The first question, whether to deploy, will be driven primarily by your project's quality requirements. If your project's tolerance for functional defects is high, such as often experienced in consumer electronics, Certitude may not be appropriate. Other devices, such as those targeting autonomous automotive applications, should be expected to be most sensitive to functional defects in silicon.

Having made the decision to deploy Certitude, you may be tempted to start Certitude regressions as soon as possible, say immediately following Release to Verification. We recommend against this for two reasons:

1. Certitude works by injecting faults on the RTL code itself. Therefore, Design Capture Complete is the earliest time that Certitude should be deployed.
2. This early in the development of the verification environment, coverage can be expected to be low and this would mean data from Certitude would be of low value. A minimum level of completion of the environment is required before Certitude is deployed. For example, coverage and/or assertions on the DUT's primary outputs.

On the other hand, waiting until just before RTL Freeze will almost certainly mean that there will not be sufficient time to complete any meaningful Certitude regression and results analysis.

This situation creates a dilemma: starting too early produces useless results (because new code is being added all the time and the Environment is too fresh to find all faults) and starting too late means not being able to use the results (because the boss will tape-out before the Certitude results can be fully analyzed). The solution to this dilemma is to take a step-wise approach and make use of code-coverage to let you know when to take the next step.

5.4.1 Step-Wise Approach

As hinted at earlier, and discussed in [6], Certitude classifies faults according to a classification system that starts with very course faults, such as the port connectivity fault shown in Figure 2, to the reset-condition fault shown in Figure 3, all the way to the very subtle "ComboLogic" faults which are essentially single-bit-flips of wide vectors. This classification system can be used to reduce the turn-around-time for Certitude regressions and analysis by reducing the number of faults generated and analyzed at each step.

5.4.2 Leveraging Code Coverage

The distinction between an activated fault as determined by Certitude and code coverage, as measured by VCS is largely semantics. In terms of time-and-effort, code coverage is much less expensive than Certitude, so we recommend that use of Certitude be guided by code coverage results. For example, the first class of Certitude faults is called "TopOutputsConnectivity" faults. These are essentially stuck-at-0, stuck-at-1 and inversion faults added to the primary output of the DUT. It is cheaper, in terms of time-and-effort to measure toggle-coverage on the top-level outputs of the DUT than it is to run a Certitude regression. A review of the code coverage will inform your decision about when to start the step-wise deployment of Certitude, as discussed in the previous sub-section. It will also provide guidance about any specific outputs which could be excluded from fault-insertion[4]. Toggle coverage can also be used to determine when it would be fruitful to run a Certitude regression using the Internal Connectivity class of faults. Similarly, line coverage can be used to gate deeper fault classes, such as reset condition faults.

5.5 Certitude Recommendations

Our specific recommendations for the use of Certitude are:

- Consider deploying Certitude for Designs with very high sensitivity to functional defects in silicon.
- A pre-deployment review will save you a lot of time in the end. Things to consider:
  - The types of faults that are important to your project.
  - Exclude faults on areas that are known to have low code coverage (e.g. dead-code as identified by FCA or human review).
- Use a step-wise approach to perform multiple iterations of Certitude regressions, each focused on a specific fault class, starting with the Connectivity classes and progressing through each of the fault classification levels.
- Once a fault has achieved the "detected" status, exclude it from future Certitude runs to save on regression time.
- Use code coverage to determine whether or not your Environment is thorough and complete enough to detect the expected faults at each classification level.

Immediately following RTL Freeze, perform a final Certitude regression of all the fault-classes used in your project.

6. Conclusions

Three "point tools" for Verification, XPROP, FCA and Certitude, have been discussed in the context of an end-to-end verification process. Within the context of this paper, the central questions are: should these tools be used, and if so, how to get the best value. Specific recommendations for each tool have been provided and these can be further reduced to:

- **XPROP** has a low cost in terms of time-and-effort and a very high return in terms of saved timed and actionable results. It is recommended that all development teams deploy this point-tool.
- **FCA** is applicable to those projects that place a high value on code coverage results and that spend effort to analyze code coverage holes. FCA has moderate cost in terms of time-and-effort and in some cases may be reduce the time required to analyze code coverage holes.
- **Certitude** is applicable to those projects with a very low tolerance for functional defects in silicon. Certitude has high time-and-effort costs and up-front planning is necessary to mitigate these.

[4] Some primary outputs, such as DFT outputs, may be static for the entire regression. These could be excluded from Certitude fault-insertion.

7. References

[1] G. W. Doerre, D. E. Lackey, "The IBM ASIC/SoC Methodology – A recipe for first-time success", IBM Journal of Research and Development, Vol. 46, No. 6, November 2002.

[2] Rafi Spigelman, "Improved X-Propagation using the xProp technology", SNUG Israel, 2012.

[3] Adrian Evans, Julius Yam, Craig Forward, "X-Propagation: An Alternative to Gate Level Simulation, SNUG San Jose, 2012.

[4] Alex Lorgus, Tyler Bennett, "Speed Up Coverage Closure with FCA and Echo", SNUG Canada 2015.

[5] Samuli Rahkonen, "Mutation-Based Qualification of Module Verification Environments", Master's Thesis, Tampere University of Technology, Tampere, Finland. June 2015.

[6] Shahid Ikram, Craig Barner, Joseph Derrico, Marty Rowe, "Using Certitude Efficiently", SNUG Boston 2015.