Architecturally Speaking, Are We Cool?

Effectiveness of Qualifying C/C++ Hardware Model and Simulator Environment Using Certitude C++

A Gutmann
John Hayden
David Brownell
Marty Rowe

Analog Devices, Inc.
Norwood, MA. USA

www.analog.com
www.synopsys.com

ABSTRACT

In the fast-changing world of semiconductor development, the complexity of ASIC/SoC designs continues to grow rapidly. Integrating a large amount of complex functionality into high-performance single-chip implementations with embedded software is very common. Such large and complex hardware designs require increasingly large amounts of simulation to verify and validate. Therefore, the C/C++ programming language is widely used in design flow for various hardware modelling tasks, including proof of concept, simulators, and RTL verification. Architect engineers and hardware designers rely on C/C++ hardware reference model simulators to exhaustively validate the hardware design to check for faults and weaknesses. Therefore, functional qualification of C/C++ hardware model and its verification effectiveness are critically important to the correctness of the hardware design. This paper describes a new approach using a functional qualification tool utilizing a mutation-based technique, Certitude C++, to assess hardware architecture model simulator verification effectiveness and to measure the simulator environment ability to verify, which translates into the quality of the hardware itself.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Given today's high complexity and the size of modern ASIC/SoC (System on Chip) designs, the use of C/C++ to represent complex portions of the design continues to increase. For large systems, traditional SystemVerilog and UVM techniques become too complex, time consuming, resource intensive, and difficult to manage for high-level verification. Simulating hardware RTL representation in C/C++ enables designers to accelerate simulation and achieve speeds several times faster than a comparable SystemVerilog UVM verification environment.

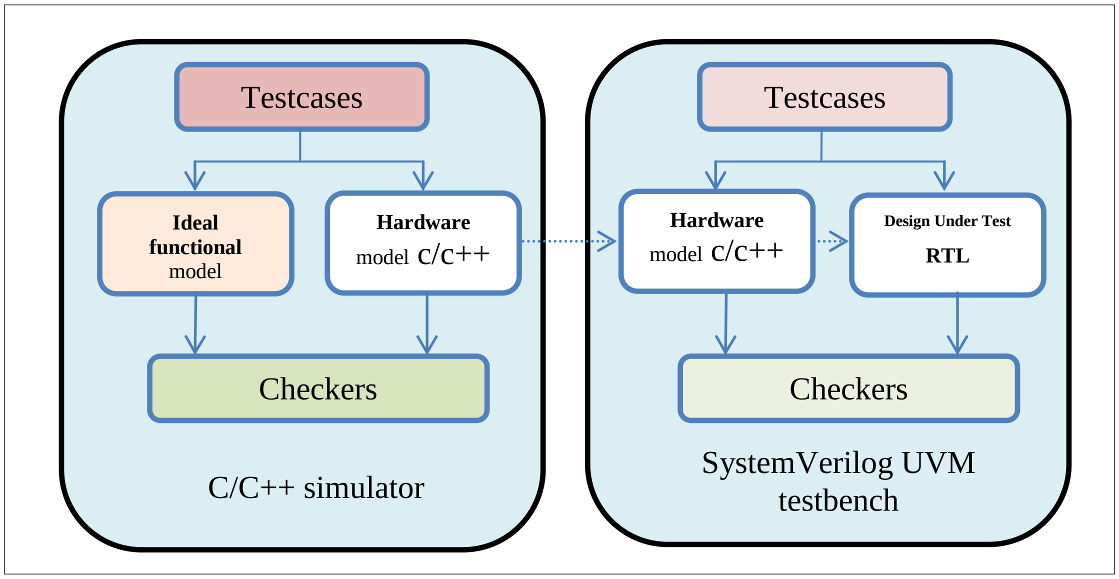


Figure 1. C/C++ simulator vs. SystemVerilog UVM testbench environment

A complex C/C++ environment can speed up a SystemVerilog UVM testbench significantly under the same amount of computing resources. In our particular case study, a C/C++ simulator sped up as much as 50 times and such strategy was a must. A bit-accurate design-equivalent RTL reference model was created as well as its C/C++ ideal reference model and checker. Therefore, the goal was not only to confirm that our hardware architecture was sound but also to create a bit-accurate hardware equivalent model, which can be translated into (or equivalent to) RTL and verification checker as well. It also saves effort in RTL verification and minimizes complexity. Therefore, completeness of C/C++ simulator environment must be correct and fully cover all aspects to decrease the probability of bugs in the hardware reference model. The functional qualification tool with a mutation-based technique, Certitude C++ can help objectively quantifying the effectiveness of the C/C++ simulator environment which results in higher confidence in the architecture, modelling, and design within the simulation environment.

2. What is Functional Qualification?

Research of mutation analysis started in the software development domain in the 1970's and where research has been ongoing since [1]. The fundamental idea of mutation analysis is that one can identify weaknesses in testing of a software program by systematically introducing a single behaviour change, called a "mutation", into the testing program, and check whether the mutation is observed by the validation suite. If there is a difference in the output of a test between the mutated code and the original code then the mutation is considered "killed". However, if no difference is observed then the mutation is considered "live"and indicates a potential weakness in the testing of the software program. The next principle of mutation analysis is that, by improving the testing of the program to kill the mutants, one can potentially identify real bugs that are not capable of being detected in the current environment. Table 1 below shows example mutations that could be inserted into a program[2].

Table 1. Mutation Examples

Original Code	Mutated Code
A = B + C;	A = B – C;
If (A || B)	If (A && B)

The major limitation of mutation analysis is that it can be quite computationally expensive; all mutations are considered in isolation and the entire test suite must be run for each mutation to determine if it is killed or live. In large software systems, the amount of time required to consider all possible mutations can exceed the amount of resource availability.

In today's Functional Qualification software for semiconductor verification, the term "mutation" has been replaced with "fault", but the underlying principle remains the same. The software systematically introduces faults into the design code and executes the verification test suite to activate and propagate the fault, and the faults should be detected by the testbench. A fault is considered detected if a passing test fails, and any fault that is not detected highlights a potential weakness in the verification environment. Fixing these weaknesses, by adding new tests or improving the testbench checkers, increases the chances of exposing any real bugs that remain in the design. Today's software also employs algorithms for test ordering, duplicate fault analysis, and fault dropping to improve the computational efficiency of the analysis.

It is important to understand that the faults introduced by functional qualification tool are not intended to replicate bugs that might be in the design; they are only intended to introduce a change of functionality into the design. If the verification suite is not capable of detecting this change, the claim of functional qualification users is that there is a potential weakness in the verification environment. However, improving the testbench to detect this change will significantly increase the likelihood of exposing real bugs that may exist in the design. The purpose of functional qualification is to find weaknesses in the verification environment or the simulator, and, by this process, we can confirm or deny that each design logic or line of code are validated by a checker. Furthermore, Functional Qualification does not identify or correct bad checkers in the verification suite or simulator that also have corresponding bugs within the design.

A common argument raised against functional qualification is that it is not possible to model every coding mistake that users will make, and therefore there can still be bugs in the design after completing the effort. Both of these statements are true, however, they are also not claims made by users of functional qualification tools.

3. Architect and Design engineers' Challenges

Regardless of whether the C/C++ hardware reference model is used as a proof of algorithm concept or an exhaustive simulation, the correctness of the reference model is equally significant to RTL design. In many cases, the C/C++ hardware reference model can be mapped 1-to-1 to the final RTL either by RTL designers or by using an HDL translation tool. Therefore, the C/C++ hardware reference model must be comprehensively checked to ensure that we are architecturally correct. The completeness of the checkers of C/C++ stimulus is critical. A bug that exists in the reference model can easily propagate its faults into the hardware design; consequently, the RTL is only as good as the hardware reference model.

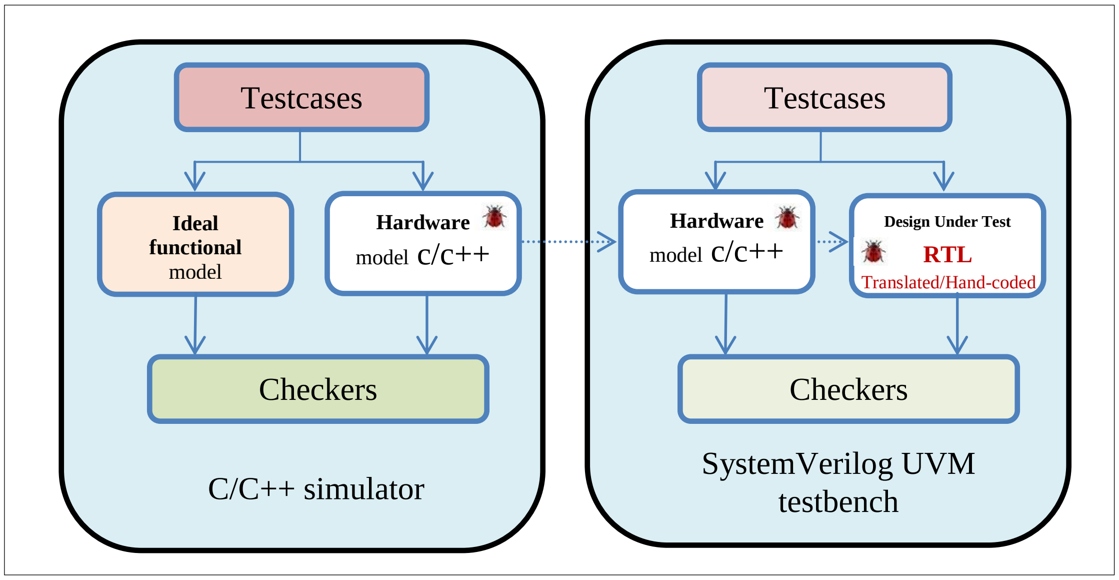


Figure 2. Bugs in hardware model in C/C++ can propagate into RTL

So, how do architect, design, and verification engineers measure effectiveness of the hardware reference model checkers and its simulator environment quickly and objectively? For every feature and functionality in the design, does the simulator environment have at least one capability to validate the hardware reference model? Given the size and complexity of the modern designs and that human mistakes are likely, it is risky to assume that the code coverage and code review are a good measure of the capability of simulator environment and subsequently the quality of the hardware reference model. New comprehensive ways, using Functional Qualification tool, to qualify the C/C++ model and its testbench effectively and quickly are needed.

Traditionally, architect and design engineers implement their C/C++ hardware reference model and its verification simulator without involving verification engineers. The hardware reference model, software, and simulator are often built and checked by one single source or tool, which can lead to a common cause bug. The way to determine if a C/C++ verification methodology is effective is by analysing code and code coverage. The more complex the designs we have at hand, the more likely there exists human error during reference model design and during analysis of these verification matrices. Significant errors in a reference model can be costly, as it can propagate killer bugs into the design or, worse, blind us to real architectural issues, which can lead to a design re-spin. Therefore, we must find ways to qualify our C/C++ model and its testbench effectively and quickly.

The best way to examine whether the C/C++ environment is capable of verifying the correctness of the hardware reference model is to insert artificial bugs or faults in the model and measure the ability of the verification environment to activate, allow faults to propagate, and detect faults. This qualification can identify missing or broken checkers or incomplete test scenarios. However, performing fault insertion tasks by hand and collecting comprehensive results for every single line of code is not robust, is time consuming, and is prone to error. In addition, modifying the design reference model code to insert faults is risky. This is where the new functional qualification, Certitude C/C++, can aid these problems safely and comprehensively.

4. Experience with Functional Qualification and Results

Bringing up Certitude C/C++ on top of our C/C++ hardware reference model and simulator was relatively quick and nearly effortless. Certitude C/C++ runs on top and the calls the C/C++ simulator which manipulates hardware reference model.

The Certitude C/C++ qualification tool analyses the C/C++ verification effectiveness in several steps. First, it parses every single line of code in the reference model to determine which faults to insert and creates instrumented mutated reference code that can be used to inject these faults. Next, it activates the faults in each line of the mutated copy of reference model one at a time, to see which faults get propagated to checkers in the simulator environment. This provides users with a comprehensive idea of the verification quality by determining whether the C/C++ verification environment is capable of detecting the faults inserted by the qualification tool. This same technique has been successfully exercised in SystemVerilog RTL testbenches for the past two years on separate projects in three different organizations within our company. The goal is to reach all lines of code within C/C++ reference model without any non-activated logic or non-detected faults. Additionally, designers will have a greater chance to find dead code, redundant logic, and coding guideline issues within the reference model under test as they review the Certitude C/C++ results by analysing the reports and adding more testcases. Designers can effectively improve their model which results in optimization in the final design area and power.

Like all design, modelling, and verification tools, functional qualification tools do not prove that integrated functions in your reference model produce the correct result or provide the correct verification suite, but they do prove that your model environment has checkers for each function. The tools do not guarantee that your verification environment is immaculate, flawless, and bug free. However, by adding functional qualification to the design process, designers can identify absent checkers quickly in their environment which would not be found easily by generic code coverage or code review. At the end of the process, the quality of the C/C++ verification environment and the hardware reference model will be improved, and it will be confirmed that every single line of code is functionally checked and qualified comprehensively.

Verification Environment Is Difficult

Whether the verification environment is for testing hardware, software and system, engineers universally struggle with these same questions.

- Have all scenarios been covered effectively?
- Are there monitors and checkers for every single feature and functionality?
- Are infrastructure or environment process issues masking bugs?
- Does the design meet standards to ensure absence of functional bugs?

Qualification Steps with C/C++

Eventually, measuring simulation environment comes down to **activation**, **propagation**, and **detection**. To detect a bug:

- The stimulus must **activate** the bug.
- An effect of the activated bug must **propagate** to a checker.
- The environment checker must **detect** the fault behaviour due to the bug.

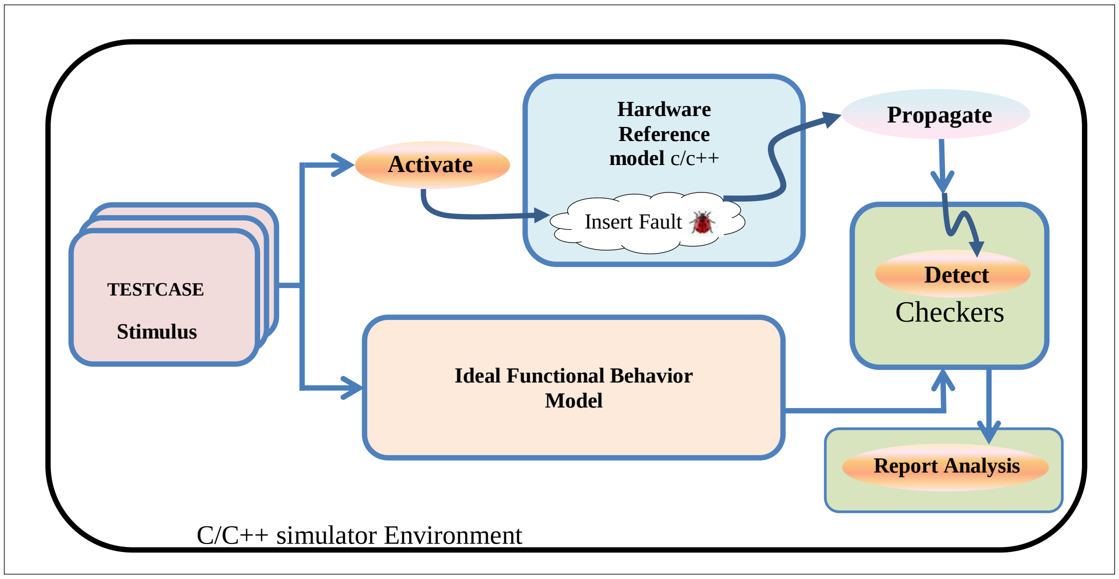

Hardware Reference model c/c++
Propagate
Activate
Insert Fault
Detect
Checkers
TESTCASE
Stimulus
Ideal Functional Behavior Model
Report Analysis
C/C++ simulator Environment

Figure 3. Effective Verification in a C/C++ Environment

Traditional code coverage only measures activation by reporting which lines of code have been successfully executed and covered. It does not provide insight into whether there were any checkers for them, and worse, whether these checkers can detect a bug. Comprehensiveness of functional logic on any covered line of code is likely unknown. For example:

If (A >0) { B += 8; C >>= A; }

Although this line of code may be covered by stimulus activation; however, traditional code coverage does not report whether there is a checker for **B** and **C**. It does not tell whether checkers of **B** and **C** are capable of detecting a fault, if inserted. This is a key of mutation testing which goes beyond the traditional coverage.

A Functional qualification tool, Certitude C++, automatically inserts "artificial bugs" called ***faults*** into the hardware reference model, as seen in Figure 3. It measures the ability of the verification simulation environment to **activate**, **propagate**, and **detect** faults. One or more of the following facts must be uncovered:

- Checkers :
    - Missing checkers. There is a hole in the test or testplan.
    - Broken checkers. There exist bugs within the checkers. When *faults* are not detected by the checkers, it is likely that the checker is missing or broken. Checkers can have a bug as well.
    - Checker is disabled or forgotten or incomplete.
    - Checker does not flag a failure.
- Simulation environment and process issues: If checkers exist and are correct, the environment reports a false positive which can mask the existing bugs.
    - Process issues or script errors. Tests never run but reported false positive.
    - Failed tests ran but never propagated and reported

  - Over-constrainted environment
- The hardware reference model or software may be overdesigned or redundant code.
  - There may exist redundant code which the design engineer is not aware of. If a fault is inserted and not detected, design engineer can examine whether a check must be added or not. If no checker should be added, it is likely the line of code does not belong in the design.
  - Inefficient coding style. Wasteful hardware design logic or instruction code. If a line of code is not detected because the inserted fault is being corrected by another line of code, then the design may not be fully optimized as it contains redundant logic.
  - Bad or invalid coding style, i.e. a code with injected faults does nothing; therefore, the code will not do what the design is intended to.
- There may exist dead code within hardware reference model or software.
  - Line of code cannot be reached.

How Fault injection Works?

Here is a simple line of code:

If (A == 0 && B > -2) { C += 8; D >>1; }

A single fault is injected one at a time by the qualification tool. A modified C/C++ code from above with inserted faults will look as follows:

First fault:

If (0) { C += 8; D >>1; }

Second fault:

If (1) { C += 8; D >>1; }

Third fault:

If (!(A == 0 && B > -2)) { C += 8; D >>1; }

Fourth fault:

If (A != 0 && B > -2) { C += 8; D >>1; }

Fifth fault:

If (A == 0 || B > -2) { C += 8; D >>1; }

Sixth fault:

If (A == 0 && B < -2) { C += 8; D >>1; }

Seventh fault:

If (A == 0 && B >= -2) { C += 8; D >>1; }

Eighth fault:

If (A == 0 && B > +2) { C += 8; D >> 1; }

Ninth fault:

If (A == 0 && B > -2) { C; D >> 1; }

Tenth fault:

If (A == 0 && B > -2) { C -= 8; D >> 1; }

Eleventh fault:

If (A == 0 && B > -2) { C += 8; D << 1; }

For subroutine or function call, an example code is below:

function_helloWorld();

An injected fault shall be code removed:

/* code removed */

Every single fault injection is activated, propagated, and detected by qualification tool. The following must be true in order for the environment to be effective and robust enough to detect all bugs within hardware reference model.

- All original RTL tests compliant with environment, i.e. all original tests must ***pass*** and be detected by the qualification tool. The qualification tool should not move on to the next step unless this is true.
- All tests running with modified RTL containing faults must ***fail*** and be detected by the qualification tool.

As can be seen above, for every single line of code, fault injections can consume a lot of time if performed by humans. Modifying the hardware reference model source code can be risky, tedious, and prone to more errors. Collecting results and regression can be an engineering nightmare. Functional qualification tool, Certitude C++, can be extremely helpful and easy to integrate. Here is a list of Fault types.

Table 2. Type of Faults Injection

Type of Faults	Fault Insertion
Condition Faults	Condition True, Condition False, Negated
Dead Faults	Dead Assign, Dead Call, Dead Operator, Dead Statement
Array Faults	Value At Zero, Value At Max
Operator Faults	Swap Operations including prefix and postfix

Easy Integration

In our experience, Certitude C++ bring-up was fast and straight-forward. Our simulator environment in GNU C/C++ with add-on library was up and running within two hours time by modifying a few certitude files. Certitude C++ runs on top of the simulator environment. It compiles, executes, runs, and gathers result from existing testcases.

Table 3. Simulator Environment Qualified Statistic

Source Code	Line Count
C/C++ hardware reference model	9658
C/C++ Ideal reference model and Testbench	9642
RTL SystemVerilog	12,808
SystemVerilog UVM Testbench	17,313

Setting up simulator environment for evaluation consists of creating configuration files. The following is the list of important files:

- **certitude_config.cer** file contains certitude setup config
- **certitude_compile** file contains our environment compilation steps
- **certitude_batch** file contains lsf jobs command
- **certitude_execute** file contains scripts that execute testcases and capture ***passed*** and ***failed*** results from simulator
- **certitude_testcases.cer** file contains all the testcases' names
- **certitude_hdl_files.cer** file contains hardware reference model files and any qualifying waivers
- **cert_testlist** file contains command line flag for each testcase

Fault Classification

At the end of a functional qualification effort every fault will be categorized. Users will know for every line of code if the simulator verification environment is at least capable of detecting a bug if one exists on the line. Here is a list of fault types:

- **Non-Activated fault** : No testcase capable of activating fault on any particular line of code

- **Non-Propagated (Weak)** : Fault is activated, but no testcase has propagated it to detection phase. E.g. an if-statement which is always false.
- **Detected:** At least one testcase has activated, propagated, and detected the fault
- **Non-Detected**: The fault has been activated but no testcase has detected it. This fault is propagated to the boundary of the output of hardware reference.
- **Not Yet Qualified**: The fault has not been fully processed yet. There are two possible reasons why:
    - The activation phase has not been executed yet.
    - The detection phase has not been executed yet or has been interrupted before detection phase has been completed for this fault.

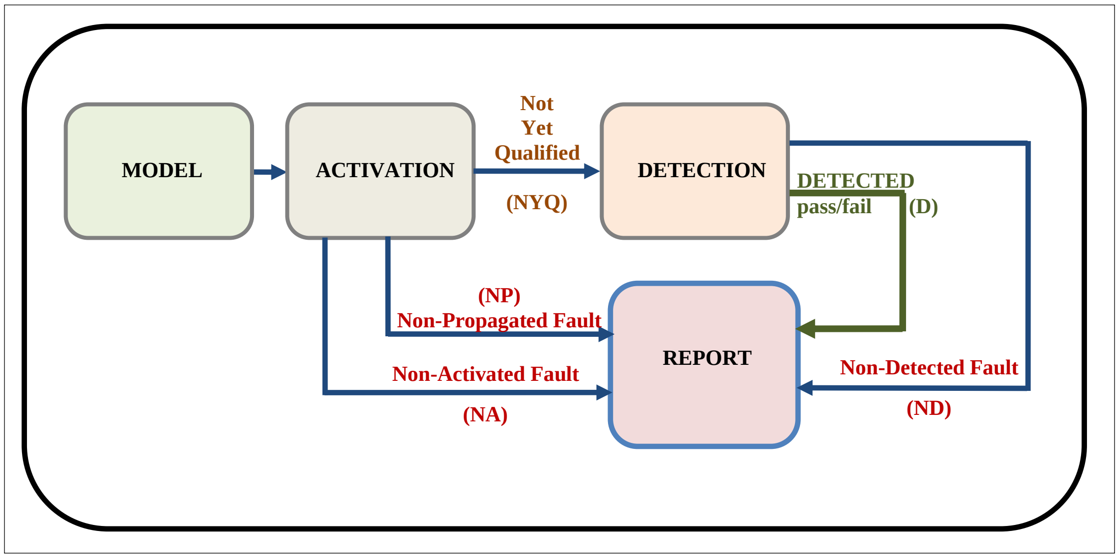


Figure 4. Fault Classification and Process Flow

Process and Flow

There are 3 important phases.

- **Model** phase : Analyse the design (static), line by line, to determine the possible faults. Upon analysis completion, the functional qualification tool writes instrumented RTL and identifies fault locations in source code.
- **Activate** phase : Analyse the verification environment by simulating original tests once. Identify ***Non-activated*** faults and correct faults and testcases.
- **Detect** phase : Measure the ability to detect bugs by injecting faults and simulating one fault at a time. Identify ***Non-Detection*** faults, ***Non-Propagated(Weak)*** faults, and ***Detect*** faults. Report details of fault ID, fault type/condition, and status.
- **Report Analysis** phase : Generate comprehensive reports of all associated faults, details of faults in changed logic, testcases, and its results. Report of results can be generated anytime while detection is running in the background.

Runtime and Data Volume

Runtime is the primary limiter of functional qualification. Depending on the number of faults placed in a design and the efficiency of the testbench in detecting those faults, the runtime can become extremely large if faults are not efficiently detected by the test environment.

Activate Runtime =(# tests * test time) / Number of Servers

Detect Runtime = (# faults * # tests * test time) / Number of Servers

The case study environment consists of a randomized short regression suite and an exhaustive regression suite. For an objective measurement, we provide data here from our short regression suite which consists of 135 testcases. Exhaustive regression suite includes short regression testcases and 81 advanced tests with all input combinations exercised.

Table 4. Runtime and Statistic for short regression suite

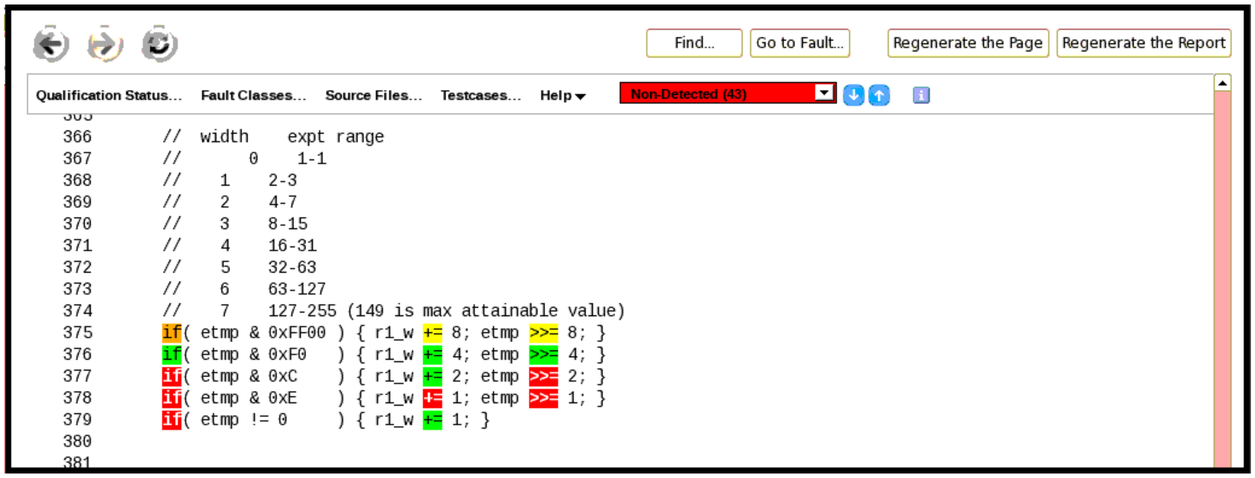
Phase	Runtime
Model phase	1 minutes
Activate phase	45 minutes
Detect phase	21 hours

The other major limiter to productivity and cost with the functional qualification tool is the amount of data generated by the tool. Just as the runtime puts pressure on computing resources, the data generated by the tool can be also a bottleneck if disk space is limited.

Example of Certitude C/C++ Code Report

Below is an example of Certitude C/C++ internal HTML report from our case study. In Figure 5 below, fault color coded highlight indicates detected Fault types. All Certitude C/C++ Fault Type colors are explained below:

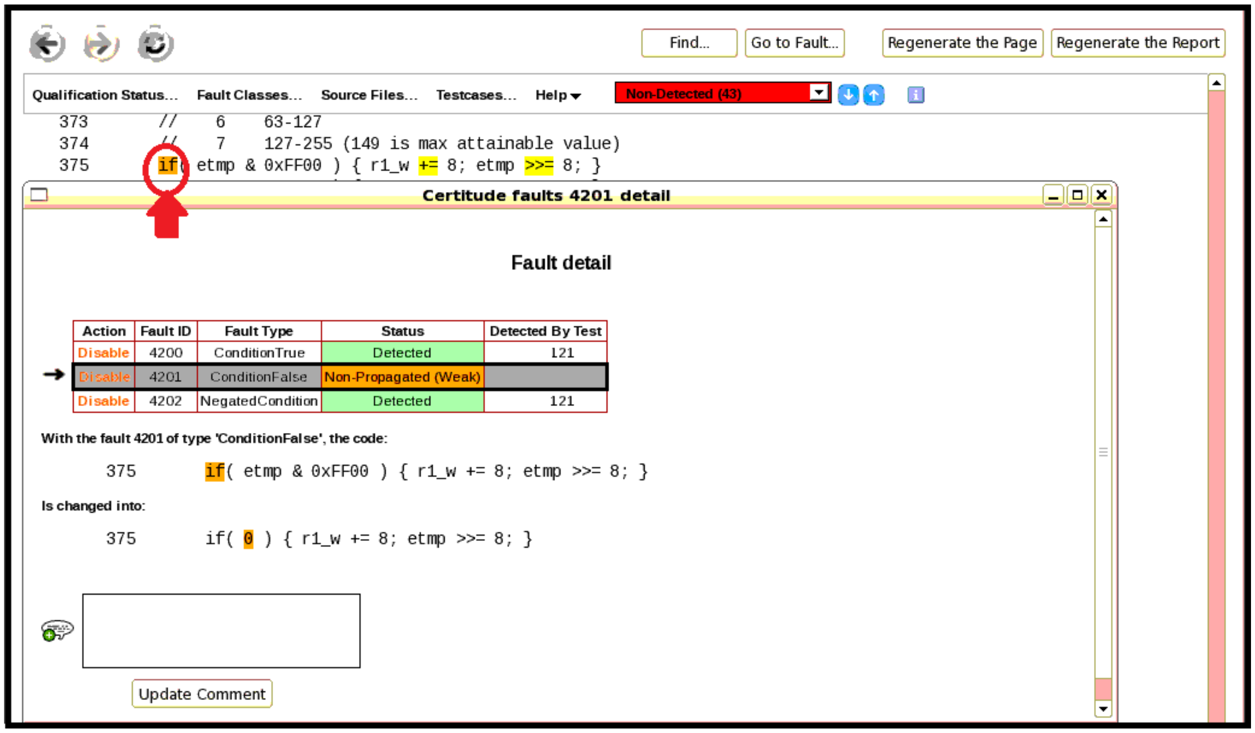
- Yellow represents non-activated faults.
- Orange represents non-propagated(weak) faults.
- Green represents detected faults.
- Red represents non-detected faults.



```
366      //  width    expt range
367      //      0    1-1
368      //    1    2-3
369      //    2    4-7
370      //    3    8-15
371      //    4    16-31
372      //    5    32-63
373      //    6    63-127
374      //    7    127-255 (149 is max attainable value)
375      if( etmp & 0xFF00 ) { r1_w += 8; etmp >>= 8; }
376      if( etmp & 0xF0   ) { r1_w += 4; etmp >>= 4; }
377      if( etmp & 0xC    ) { r1_w += 2; etmp >>= 2; }
378      if( etmp & 0xE    ) { r1_w += 1; etmp >>= 1; }
379      if( etmp != 0     ) { r1_w += 1; }
380
381
```

Figure 5. Example of Certitude C/C++ Code internal HTML Report

From our case study, Figure 6 illustrates a non-propagated(weak) fault. On line 375, a non-propagated faults displayed in orange with a hyperlink. The root cause of this fault was missing a testcase which can activate condition True in the if-statement. When testcase only generates false in if-statement, no "conditionFalse" fault will be detected at the output. A testcase which generates input combination resulting in condition-true in the if-statement would propagate a "conditionFalse" fault to be detected at the output of unit under test. Of course, if such input combination is invalid, then this line of code may be deadcode or invalid.



```
373      //    6    63-127
374      //    7    127-255 (149 is max attainable value)
375      if( etmp & 0xFF00 ) { r1_w += 8; etmp >>= 8; }
```

Fault detail

Action	Fault ID	Fault Type	Status	Detected By Test
Disable	4200	ConditionTrue	Detected	121
Disable	4201	ConditionFalse	Non-Propagated (Weak)	
Disable	4202	NegatedCondition	Detected	121

With the fault 4201 of type 'ConditionFalse', the code:

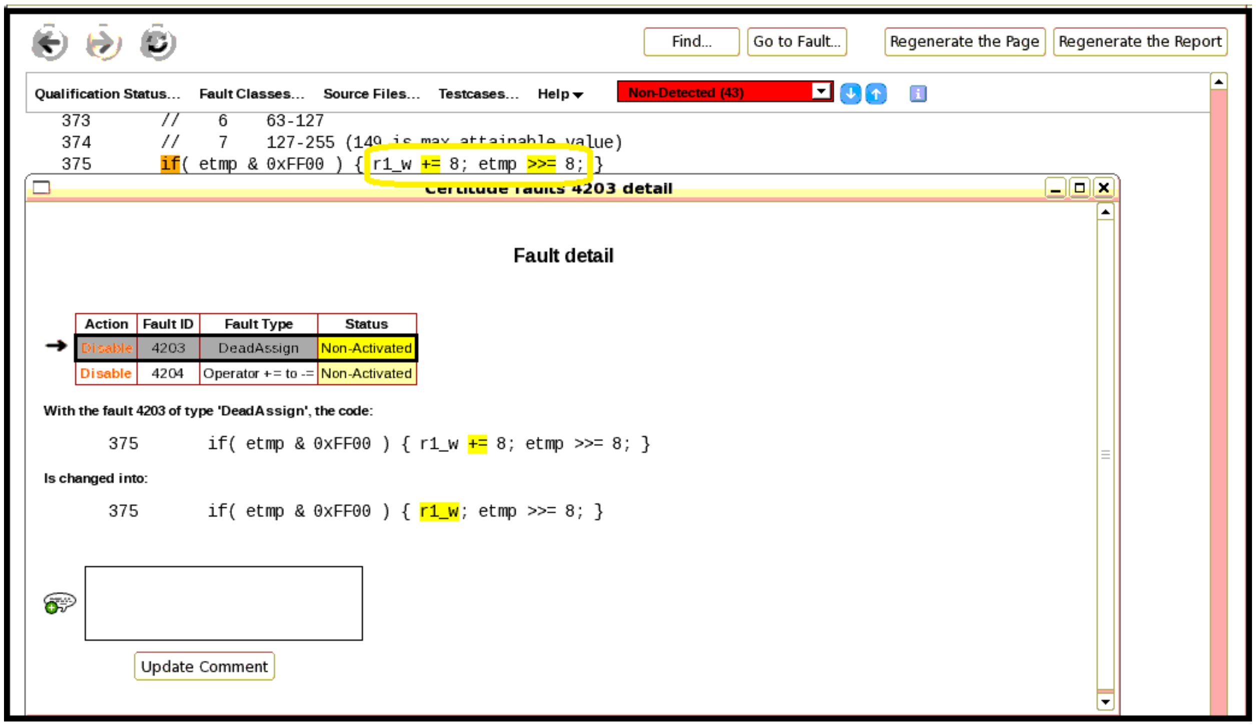
```
375      if( etmp & 0xFF00 ) { r1_w += 8; etmp >>= 8; }
```

Is changed into:

```
375      if( 0 ) { r1_w += 8; etmp >>= 8; }
```

Figure 6. Example of Non-Propagated Fault(Weak) report

In the same example, the code inside the curly brackets on line 375 in Figure 7 had never been reached because the condition of if-statement is always false as explained earlier. The faults inside the curly brackets were never activated and were reported as non-activated faults. Again, adding a testcase which resulted in a Condition-True on Line 375 if-statement would allow these fault ID 4203 and 4204 to be activated.



```
373        //    6    63-127
374        //    7    127-255 (149 is max attainable value)
375        if( etmp & 0xFF00 ) { r1_w += 8; etmp >>= 8; }
```

Fault detail

Action	Fault ID	Fault Type	Status
Disable	4203	DeadAssign	Non-Activated
Disable	4204	Operator += to -=	Non-Activated

With the fault 4203 of type 'DeadAssign', the code:

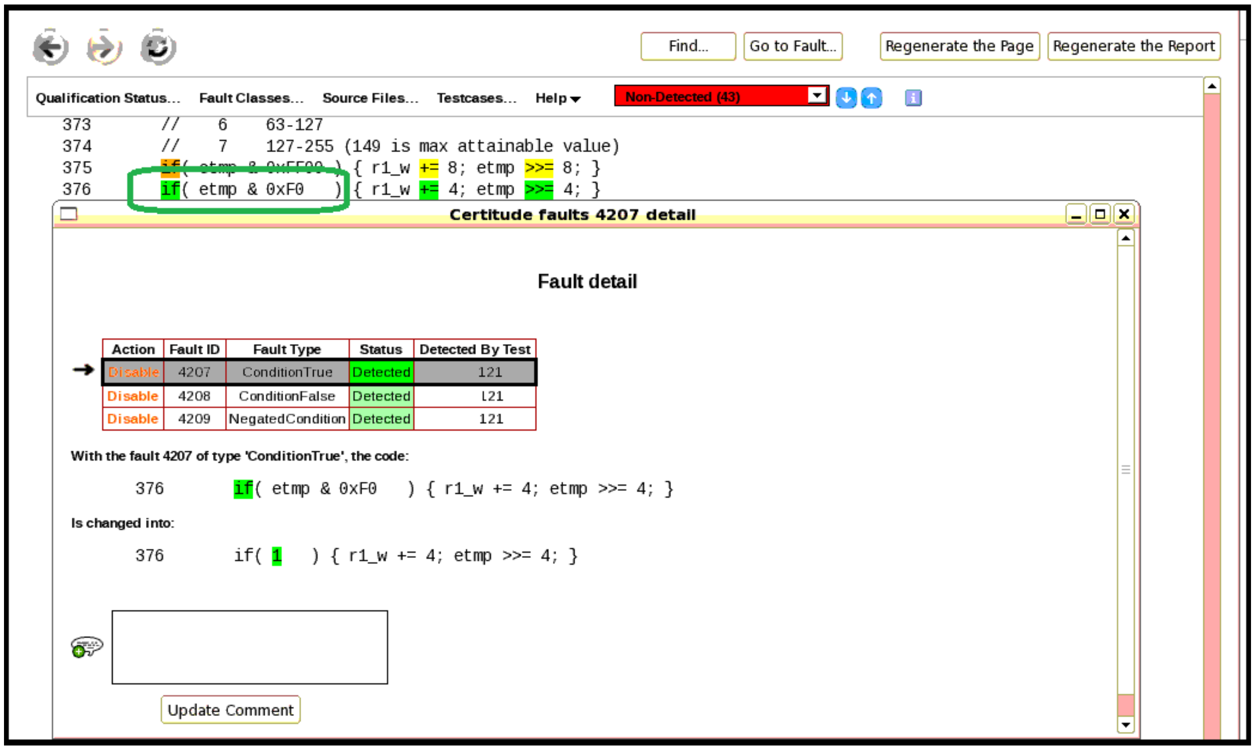
```
375        if( etmp & 0xFF00 ) { r1_w += 8; etmp >>= 8; }
```

Is changed into:

```
375        if( etmp & 0xFF00 ) { r1_w; etmp >>= 8; }
```

Figure 7. Example of Non-Activated Fault report

It is worth noting that, even if this line of code is dead, it may not be eliminated by a development tool. Whether this code is being translated into RTL or it is being compiled into machine instruction code, this line of deadcode will generate more hardware logic or will use up more instruction memory than needed. In a limited resource, need-for-speed, and “memory is never big enough” world, identifying deadcode is a positive side-effect of functional qualification tool Certitude C++, that we found valuable.



```
373      //    6    63-127
374      //    7    127-255 (149 is max attainable value)
375      if( etmp & 0xFF00 ) { r1_w += 8; etmp >>= 8; }
376      if( etmp & 0xF0   ) { r1_w += 4; etmp >>= 4; }
```

Fault detail

Action	Fault ID	Fault Type	Status	Detected By Test
Disable	4207	ConditionTrue	Detected	121
Disable	4208	ConditionFalse	Detected	121
Disable	4209	NegatedCondition	Detected	121

With the fault 4207 of type 'ConditionTrue', the code:

```
376        if( etmp & 0xF0   ) { r1_w += 4; etmp >>= 4; }
```

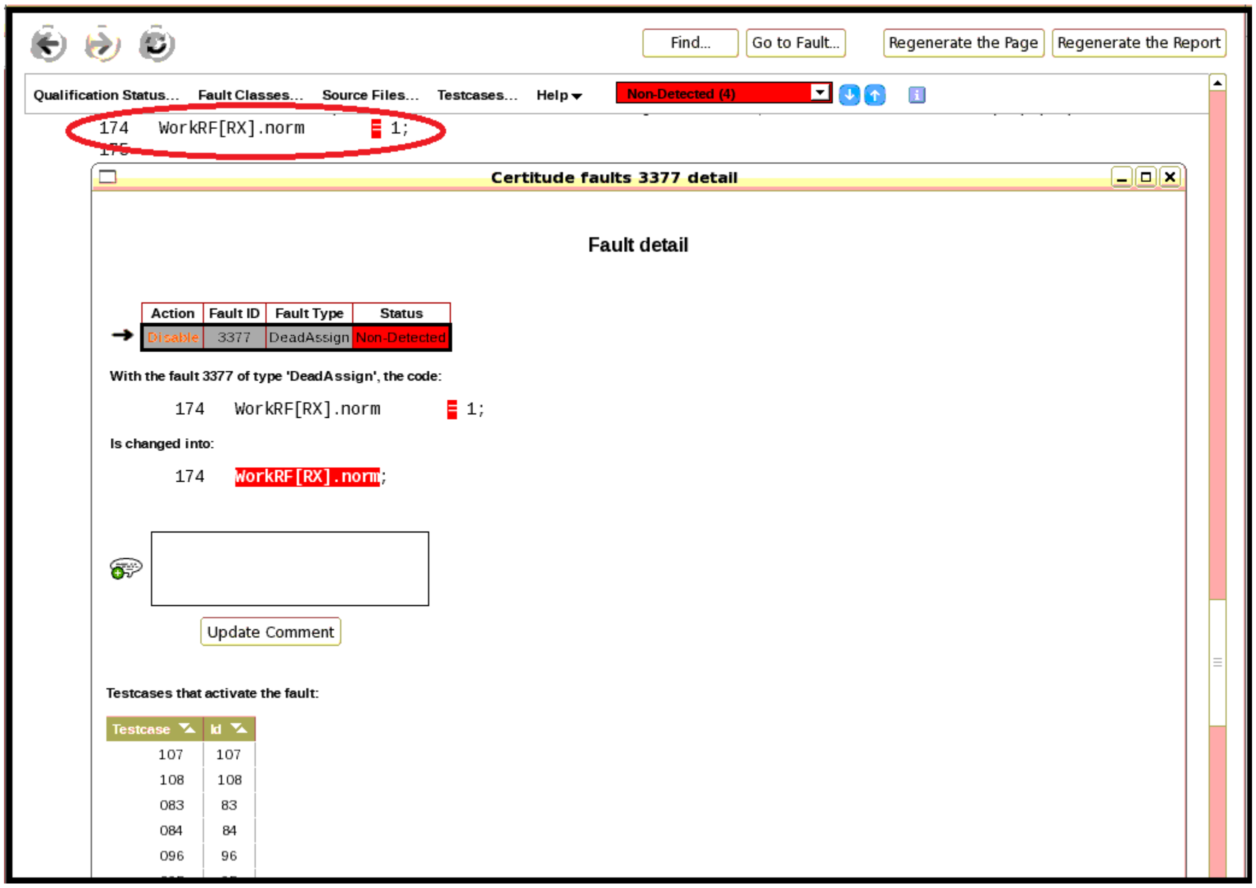
Is changed into:

```
376        if( 1   ) { r1_w += 4; etmp >>= 4; }
```

Figure 8. Example of Detected Fault report

In the same example, line 376 logic in Figure 8 were all detected when faults was inserted. The report shows the mutated logic which inserted ConditionTrue, ConditionFalse, and NegatedCondition faults, and reports the tests in which each fault was detected. We concluded that that our test 121 can efficiently generate inputs which meet this constraint; therefore, no new additional tests or checkers were required.

Figure 9 illustrates a non-detected fault report caused by the lack of a checker or no testcase for the feature. Traditional coverage would report that this line was covered and that the output was toggled by a stimulus in the verification environment. However, the behaviour of output signal below were never been checked for its correctness.



```
174    WorkRF[RX].norm       = 1;
```

Fault detail

	Action	Fault ID	Fault Type	Status
➔	Disable	3377	DeadAssign	Non-Detected

With the fault 3377 of type 'DeadAssign', the code:

```
174    WorkRF[RX].norm       = 1;
```

Is changed into:

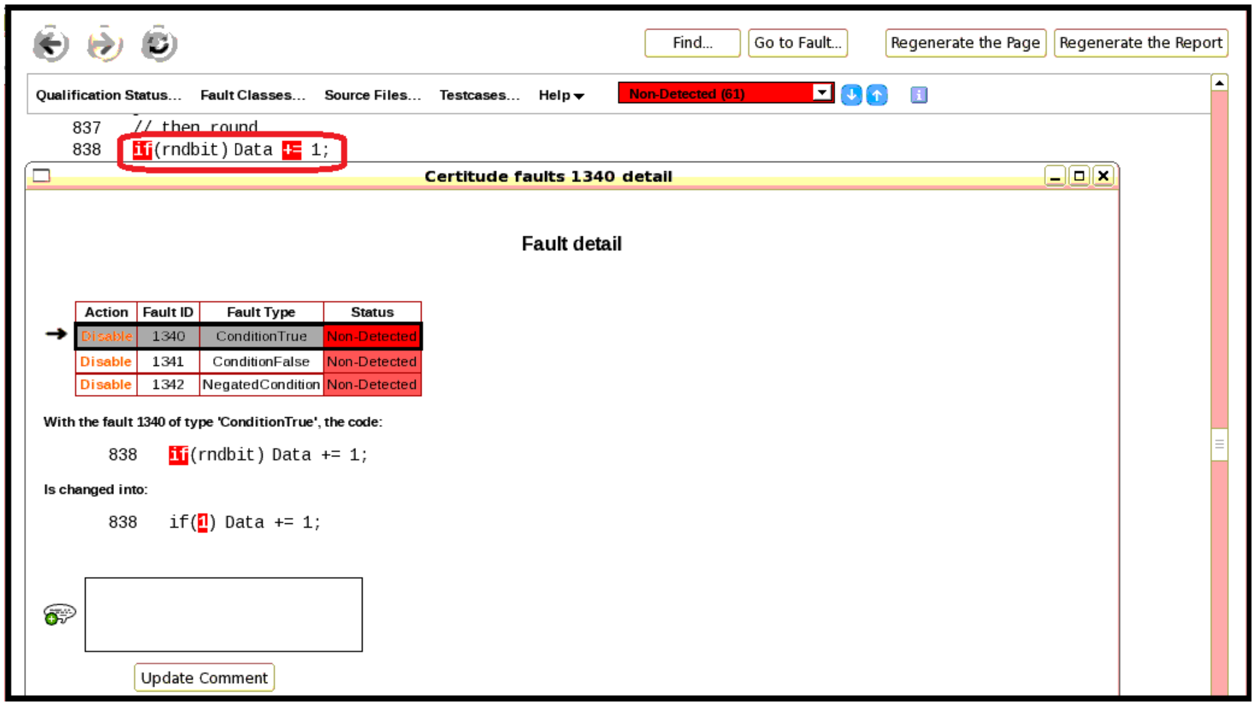
```
174    WorkRF[RX].norm;
```

Testcases that activate the fault:

Testcase	Id
107	107
108	108
083	83
084	84
096	96

Figure 9. Example of Non-Detected Fault report

Figure 10 illustrates a non-detected fault covering unnecessary logic, which turned up in our case study. Mathematically, rounding is often necessary in arithmetic computation but not required depending on the accuracy of hardware requirement. We have found both “rounding twice” issue and “unnecessary rounding” logic. Below is an interesting example which rounding logic was not necessary for the specification of our design. An adder could be saved here.



```
837    // then round
838    if(rndbit) Data += 1;
```

Fault detail

Action	Fault ID	Fault Type	Status
Disable	1340	ConditionTrue	Non-Detected
Disable	1341	ConditionFalse	Non-Detected
Disable	1342	NegatedCondition	Non-Detected

With the fault 1340 of type 'ConditionTrue', the code:

```
838    if(rndbit) Data += 1;
```

Is changed into:

```
838    if(1) Data += 1;
```

Figure 10. Example of Non-Detected Fault code

All in all, an architecture flaw or a killer bug can cause a design re-spin. An over-designed architecture can cause resource issues. It is important to find all fault types in the design and the environment which can be anything from missing checkers, broken checkers, over-constrainted testcases, redundant code, inefficient coding style, to deadcode. By finding out these faults, designers can investigate and correct the piece of unoptimized code in reference model or the flaws in verification environment. In our case study, we were able to eliminate a number of redundancies in the code and instances of deadcode as well as finding killer bugs and architecture flaws.

Additionally, we also found qualification tool to be useful for qualifying software because each line of code, which cannot be optimized by development tools, translates into instruction codes storing in memory on chip.

5. Conclusions

We have found Certitude C/C++ functional qualification to be effective, quick to identify holes, and easy to integrate and bring up. The resulting analysis report exposes important issues including missing coverage, missing testcases, bad coding guidelines, dead code, redundant logic, and severe bugs. The unbiased systematic approach of Functional Qualification tools not only measures the effectiveness of the C/C++ its environment, but it also provides greater confidence in our hardware reference model and its environment.

6. References

[1] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4):34-41. April 1978.

[2] David Brownell, "Who's Watching the Watchmen? The Time has come to Objectively Measure the Quality of Your Verification Environment" SNUG, September 12, 2013.

[3] M. Hampton, "Functional Qualification," EE Times, June 25th 2007.