# How Can We Build More Reliable EDA Software?

David E. Wallace
Scott Aron Bloom
Blue Pearl Software
Santa Clara, CA
+1 (408) 961-0121
{dave,scott}@bluepearlsoftware.com

## Abstract

*Many EDA development groups rely on full tool testing as the only means of determining program correctness. Meanwhile, other software development communities have made use of extensive unit tests as a method of assuring high-quality components. We believe that the value of such testing is derived from principles familiar to the EDA community from hardware test generation: the need for both controllability and observability of a unit under test to verify correct behavior.*

*We believe that unit testing has value both for the development of new code and for the understanding and adoption of existing code, though it may require substantial refactoring*[*] *to make older code testable.*

*We explore both the value and costs of unit testing for EDA software, as well as noting some alternatives to stand-alone unit testing that may provide some degree of observability and controllability when full unit testing is impractical. We describe tools and resources we have used in putting together a unit test infrastructure at Blue Pearl Software.*

## 1. INTRODUCTION

The field of Electronic Design Automation has developed a great understanding of how to build and test massively complex hardware designs to assure a high degree of reliability. Yet, like the proverbial shoemaker's children going barefoot, we EDA developers have often failed to apply this understanding to the design and development of our own software. All too often, we content ourselves with full system testing of our tools on a combination of sample designs with known results and customer designs with presumed good results. While this is often sufficient to detect gross misbehavior, such as crashes, it does little to ensure that the underlying components are robust, performing optimally, or are adaptable to new requirements. This is the software equivalent of shipping any microprocessor that can execute a few sample instruction sequences correctly and boot the operating system without causing a hardware panic. Few of us would argue that such a testing regime would be sufficient to assure quality hardware performance in the field, given the complexity of modern hardware designs. Neither is the software equivalent sufficient to assure robust performance of the tools we provide.

A related question is: how can we understand and adapt existing EDA software once the original developers have moved on? Much EDA software is based on legacy code that was developed elsewhere, or is being maintained by persons other than the original developers. The EDA landscape is full of companies that are acquired, development teams that have moved on, and university software that is adapted to use in a commercial environment. How well do we understand the strengths and limitations of inherited code? Even if it is extensively field-tested (and thus likely free of gross errors), will it work properly when faced with new conditions and new designs? Can we determine how to extend and modify such code while keeping the core algorithms correct, or must we treat it as a black box that can be called and assumed to yield correct results, but never touched?

While there are few if any reliable surveys of development methods across EDA companies, the two of us have a collective 51 years of experience with EDA development at 13 different companies. We are confident that the problems we discuss are widespread throughout the EDA industry, not the isolated pathologies of one or two individual companies.

These ideas are not a panacea. There is no quick or easy fix that will guarantee an absence of bugs. We view this as the start of a multi-year effort to transform the culture and the software of an EDA company in order to get more reliable and maintainable software, and a more predictable development cycle.

## 2. Background: Controllability and Observability

What lessons can we draw from the hardware test community? Two of the most significant advances that gave rise to modern ATPG were scan testing[2,3] and the D-algorithm for combinational test pattern generation[4]. These two advances supported each other: the D-algorithm

[*] Refactoring is modifying code to improve its structure without changing its behavior. [1]

allowed the automatic generation of test patterns that would expose faults in a combinational network, while scan testing decoupled the sequential behavior of a system from its combinational behavior, allowing these patterns to be applied to logic gates throughout a chip and the results observed.

Two key concepts used here are controllability and observability. In order to detect a presumed fault on a unit under test (modeled as a stuck-at value on a logic connection), we need to be able to control the inputs so as to activate the faulty behavior (controllability), and we need to also to propagate the effects of the fault to a point where we can observe the difference between the good system behavior and the faulty system behavior (observability).

Scan testing allows us to decompose complex hardware systems into smaller units that can be individually tested. It allows us to move controllability and observability points from the system I/Os to the state elements of the design, allowing more faults to be tested, and simplifying both the generation of tests and the interpretation of the results.

These same principles apply to software as well as hardware. If a software test is to expose faulty behavior in a unit of code, we need to be able to control the inputs of the unit so as to activate the faulty behavior, and we need to be able to observe the consequences of that fault in the outputs. Just as with hardware, testing using only the full system inputs and outputs is usually not sufficient to cover all the possible faults of subsystems, and certainly makes certain tests harder to write, even if we can theoretically express that test as a functional test. For thorough test coverage, we need to be able to access internal observation and control points in the software – what Michael Feathers refers to as “seams” [5, p. 31] in the code, which serve an analogous function to scan registers in hardware testing.

These seams will be used to allow testing of individual functions, methods, and classes in test harnesses.

## 3. The Case For Unit Testing

What is a unit test? Various sources [5 p. 12, 6 pp. 4-7, 7, 8, 9, 10 p. 499, 11 p. 8] agree that a unit test is a test of an individual unit of software, generally a method, function or procedure, but possibly an entire module or class interface. It often involves a separate test harness, involving drivers and stubs, to test a single unit or group of units in relative isolation.

In his critique of Extreme Programming, Robert Glass called unit testing possibly “the most agreed-upon software best practice of all time.”[12] However, this kind of unit testing has not been routinely used in EDA in our experience. Neither of us has ever worked in an EDA team where such stand-alone unit tests were developed and shared by any other team member. If any such testing was done, it was kept private by individual developers and discarded once their code had been integrated into the rest of the tool. There is, however an alternate definition that has been used by two of the teams we have worked with, where unit test is used to refer to a small system test case that tests a single feature or scenario of the software tool. In this paper, we use “unit test” to refer to the first usage; the latter may be referred to as a small functional test.

### 3.1 Bug Example

Consider the following line of C code:

```
/* Round t up to next multiple of m */
t = t + t%m;
```

Does it work? Is it reliable? Does it accomplish what the comment suggests it should do?

This calculation appeared in nearly a dozen different places in a commercial EDA tool one of us once worked on. It was attempting to calculate what internal clock cycle would correspond to the next user clock edge, for a user clock that occurred every m internal cycles. It had never been the subject of a user bug report, and the tool passed every regression test without complaint.

Yet a bit of hand simulation shows that this calculation is incorrect. For t=7, m=3, the code rounds t up to 8, not 9. For t=8, m=3, it rounds to 10. For t=17, m=4, it rounds to 18 rather than 20.

The primary reason this bug had remained undetected for so long was that all but one of the regression tests, and most of the user experience to date, had been operating on blocks with a single primary user clock, where m=1 or 2. This buggy calculation works whenever t is an integer multiple of m/2, which is always the case for these values of m. But the company was about to support more accurate models for multiple clocks, which would require larger values of m.

This brings up one aspect of EDA development: we frequently reuse code under new conditions that will stress it in ways not previously experienced. Just because code was used without apparent errors in a previous deployment does not mean that it will continue to be reliable when the requirements change.

To test a single line like that in the middle of a several hundred line routine is challenging. Not only do we have to set up the conditions that will trigger the faulty behavior, we need to trace what conditions on other variables and data structures will result in a change in the observable behavior on the outputs of the big routine. But suppose instead that we had refactored this particular calculation into a separate, one-line function (correcting the faulty calculation in the process):

```
int RoundUpToMultiple(int num, int mod) {
  return num%mod? num+mod-num%mod: num; }
```

Now we have observability and controllability points directly before and after the suspect code, and we can call it directly from a test harness:

```
EXPECT_EQUAL(4, RoundUpToMultiple(4, 2));
EXPECT_EQUAL(6, RoundUpToMultiple(5, 2));
EXPECT_EQUAL(6, RoundUpToMultiple(4, 3));
EXPECT_EQUAL(6, RoundUpToMultiple(5, 3));
EXPECT_EQUAL(3, RoundUpToMultiple(3, 3));
```

By testing so close to the code, we can validate the functionality directly. This does not imply that every calculation should be its own function, but note that in this case, doing so allows the different places performing this calculation to reuse a single tested implementation instead of copying their own. In general, coding for testability will involve writing more modular and shorter routines and methods than code written without such considerations.

## 3.2 Seams in the Code

Feathers [5] defines a seam as "a place where you can alter behavior in your program without editing in that place." Calling a function from a test harness rather than production code as above might seem to qualify. His usage, though, is mostly focused on creating seams that allow us to modify either the arguments passed to a function/method or routines called by the function/method in order to break dependencies and allow the code to be tested in relative isolation.

He defines three types of seams that can be used to get code under test: preprocessor seams, link seams, and object seams. Preprocessor seams make use of the preprocessor to change the behavior, such as using a #define to distinguish between testing mode and system mode. Link seams change behavior by linking different object libraries with the code under test – for example, linking with stubs or mock objects for testing rather than the real thing (for the distinction between the two, see [13]). Object seams use the inheritance hierarchy of an object-oriented language like C++ or Java to substitute test objects with a common interface in place of real objects in a test harness.

One example of creating an object seam to break build dependencies occurred in a early unit test we wrote at Blue Pearl. We needed an instance of a Net object from our internal database, which required instantiating a Module. (Faking out the Net object – or the Module – was considered impractical, because of the amount of Net behavior the class under test required.) Linking the Module code required accessing a method called getPrimitivePinName from a global instance of a big DesignAttributes class, and trying to link that class would have required linking in a lot of additional code. So instead, since this method was the only behavior that Module needed from DesignAttributes, we created an abstract class AbstractPinRenamer, that just supports a getPrimitivePinName method with the same interface as the DesignAttributes one. Then we had both DesignAttributes and a new test class inherit from AbstractPinRenamer, and changed the Module code to interact with an AbstractPinRenamer rather than a DesignAttributes. Now the test code could interact with the smaller test class, while the production code still used a DesignAttributes. This is an example of the AbstractInterface pattern from Feathers[5].

## 3.3 Reducing the Development/Test Cycle

One of the key reasons to adopt unit testing is to speed up the development/test cycle to give near-instantaneous feedback on code that we develop. The quicker we can get testing feedback, the easier it is to understand and fix our mistakes, and the less likely it is that our mistakes will inconvenience our fellow developers by getting checked in to a shared repository trunk.

This cycle time has important qualitative effects on how we use test feedback. A test suite that involves lots of long-running user test cases that takes a day or so to run through will probably not be run before developers check in code. Even if they do, the odds are that someone else will make a change that must be merged before checkin, so that the code being tested will not be exactly the same as the code that is checked in. Moreover, debugging such examples when they fail also takes a long time, so when the trunk is broken it may stay broken for a few days.

If we can reduce the run time to several minutes on a single machine (either by running a moderate number of quick-running small functional tests, or very few larger tests), we can get a test suite that can be run before checkins and at the conclusion of major blocks of work. This is a big improvement over the overnight test suite, and decreases the chances of checking in bad code. Most EDA organizations have or develop such a suite. But several minutes is still a significant barrier to routine use of tests during development. It requires a significant interruption of work to switch from developing code to running the tests, which discourages developers from running the tests except at natural breaks in their work flow. Also note that any test suite that relies on running a full EDA tool will have a certain amount of startup time overhead for each test case, such as getting licenses and reading startup files, which limits the number of test cases that can be included if the suite must run in a few minutes.

Finally, if we can reduce the run time of a test suite to a few seconds, rather than minutes, we get tests that can be routinely run as part of normal development activity. This is one goal of unit testing. If it only takes us a few seconds to verify that the interface behavior of a class we are modifying hasn't changed (unintentionally), we can run these tests repeatedly as we modify the implementation. (In such a development loop, we would typically run just the unit tests associated with the files or libraries we are modifying – we would run a larger suite before checkin.) This can facilitate development techniques such as Test-Driven Development (TDD) [14]. We want individual unit tests to run extremely quickly – milliseconds or even microseconds – so that we can run hundreds or thousands of them in just a few seconds.

## 3.4 Documenting behavior

One of the key values of unit tests is that they document (by example) the usage of the routines they cover. Unlike comments, unit tests that are regularly run will be known to reflect the current behavior of the code. If a routine has unusual requirements or preconditions (such as requiring that the *foo* routine must be called before calling *baz* on the same object), that will generally be reflected in the setup for the tests of *baz*, rather than waiting to be sprung on the unsuspecting programmer who reuses what looks like a perfectly useful routine in another context.

The full behavior of many EDA routines is not necessarily apparent from the call or short description alone. One EDA tool that we worked on in a previous company had three different static routines to determine if a primitive cell was combinational, all named isCombinational. All three agreed that an AND cell was classified as combinational and a DFF was not, but they disagreed on the classification of other cells, such as LUTs and bus keepers. Unit testing could make this difference explicit (and probably suggest renaming two of these to names that were more descriptive).

The use of unit testing is helpful for exploring the full range of behavior for existing code, particularly if the original developers are no longer available to ask questions about the code. Often, we develop our understanding of what such code does by reading through it and perhaps tracing through it with a sample test case or two. This method quickly breaks down as the code becomes more complex, and it is easy to overlook corner-case behavior that can be explored by appropriate unit tests.

Getting this code into a test harness may require doing some relatively safe and conservative refactorings to break dependencies on other code, as described by Feathers[5]. We note, however that few modifications are guaranteed not to change existing behavior of legacy code that may contain latent bugs and inadvertent use of undefined behavior. We have seen even such a trivial change as moving a global variable definition from one library to another produce a change in some system test results.

## 3.5 The Pro-Active Search for Bugs

By more fully characterizing the behavior of our functions and methods, we can identify design flaws in our current implementation. As we identify the full range of possible outputs from a function for various inputs, we can ask if the users of these outputs can handle unusual values under some input conditions that give rise to that value. This allows us to be more pro-active about identifying possible faults before a customer can encounter them.

Not every potential fault is worth worrying about. For example, few of us try to write code that is absolutely bulletproof against integer overflow or running out of memory in the absence of evidence that these are likely to be issues. But highlighting the potential interactions is useful for identifying failure modes for further investigation.

## 4. The Case Against Unit Testing

Not everyone agrees with a primary focus on unit testing for reliability. Brian Marick [11] argued for larger subsystem testing rather than individual unit tests, arguing that unit testing was too expensive and too brittle to be sustainable. We will examine these arguments in more detail below, but we should also note that Marick was writing in 1995, before the growing popularity of Extreme Programming brought a renewed focus on the advantages of unit testing. We should also note that many of our unit tests do in fact link in production versions of various subsystems – the difference is that Marick argued that such testing should be done *instead of* separately validating the underlying components, while we believe it should be done *in addition to* such validation.

## 4.1 Cost/Time

While the benefits of unit testing can be significant, it is not cost-free. Having good unit test coverage involves writing a lot of non-customer facing code. Moreover, this coding will generally be done by developers, rather than QA engineers – it requires primary skill in development languages such as C++ or Java rather than Verilog or VHDL, and requires a focus on the internal code details. (Although with the right skill mix, a QA engineer who can work with the team to help refine their unit tests from the beginning can be invaluable.) As such, it is in competition for resources with the activities of coding and debugging existing code.

The amount of effort required to develop good unit tests may be comparable to or even exceed the effort to develop the code being tested in the first place. Table 1 lists some ratios of production code to test code for some non-EDA C++ projects that use gtest[15] as a unit test engine. In terms of lines of code, production code and test code are roughly comparable for many projects. It is true that developing unit tests for a given class may be simpler and more stereotyped than the details of the algorithms for the code, so extending the unit tests for a class once some are available may go faster than developing equivalent lines of functional code, but it is clear that the effort to develop unit tests that completely cover a large project will itself be large. On the other hand, it is possible to develop them a few at a time.

**Table I: Functional:Test lines of code for gtest projects**

| Project | Functional LOC | Test LOC | Func:Test Ratio |
|---|---|---|---|
| Gmock | 14273 | 15082 | 1:1.06 |
| Gtest | 28694 | 25192 | 1.13:1 |
| AutoComp[16] | 80000 | 70000 | 1.14:1 |

We believe that the improved component quality that comes from well-tested components and better-understood components will pay dividends that will free up development time in the long run and improve the quality of the tool, leading to a better-planned and less interrupt-driven development cycle. However, this is a view that requires a long-term investment perspective. We believe that a fair amount of unit-testing literature and discussion oversells the ability to immediately recover the costs of unit testing within a very short timespan. This is not necessarily true for EDA code, and underscores the importance of management that is willing to take a longer-term view.

## 4.2 Code Stability

One of us once worked for a manager who objected to our early efforts to experiment with unit tests on the grounds that only customer-facing interfaces were likely to be stable enough to be more or less guaranteed not to change, for explicit reasons of backwards compatibility. Internal EDA code interfaces, he thought, were too likely to change to be worth testing. Indeed, the (only semi-automated) unit tests this author developed for a new algorithm there did become obsolete when the algorithm needed to be moved earlier in the design flow, to operate on a different data structure (a precursor to the one it had originally operated on). The unit tests were tied to features of the old data structure that did not exist for the new structure, and could not be easily adapted.

This is one type of code stability issue with unit tests – the risk that the code will change in a way that makes the unit tests obsolete. Another risk is that excessively detailed tests may make it difficult to make necessary changes to the code while preserving the tests. Another example from our past experience: a test system that used lots of small functional tests to verify the behavior of a tool. There were hundreds of tests verifying that the tool could detect all possible illegitimate combinations of command line flags and options, each of which would cause the tool to put out a usage string and stop. The problem was that each of these was implemented with a comparison to a golden log file for that test. This meant that any change – however trivial – to that usage string would break hundreds of tests, each of which needed to be individually checked. A five minute code change could take hours to validate and update golden files for each of the failing tests.

These problems are real, though addressable. It takes effort to develop a truly robust set of tests that is neither too brittle nor too constraining. The problem of brittle tests can be addressed in part by coding strategies that modularize and partition information so that no individual test is dependent on the details of many classes and data structures. It helps to be working in an object-oriented language, where behavior can be composed from layers of individually testable classes. It helps to test primarily the class interfaces, rather than details of their implementations. And some tests will indeed become obsolete as code changes, as will some of the code itself. As a matter of responsibility, it should be the developer's responsibility to maintain and update appropriate unit tests for the code he or she develops, and to retire tests that are no longer appropriate.

For minimizing rigidity, we should strive not to have too many tests test the same functionality repeatedly, so that not too many tests will have to change when the functionality changes. It is fine to have lots of functional tests that verify that a usage string is produced when illegal options are invoked, but only one or two should test the actual value of that string. If the rest could use regular expressions or filters to gloss over the actual contents of the string, the suite would have been more robust to changes in usage. In general, unit tests that focus on specific behaviors of a class under test are likely to be more focused on distinct behaviors than functional tests, which of necessity must include a fair amount of canned common behavior.

In addition, unit test fixtures written in an object-oriented language can benefit from inheritance to factor out common behavior that must be checked. Test fixtures in gtest are C++ classes that can inherit from other test fixtures. So if there is common behavior that must be checked for several different test fixtures, such checks can be coded in a parent test fixture class that the different test fixtures inherit from, making sure that there is only one place that needs to be fixed when this behavior changes.

## 4.3 The Altruist's Dilemma

What we call the "Altruist's Dilemma" is rooted in the costs of developing unit tests. Many of the benefits of unit testing cited in section 3 accrue to the entire team of developers using unit tested code – the ability to more quickly understand it, the ability to modify it without breaking interface behavior required by the original developer, and the higher productivity resulting from being able to reuse this code without having to spend much time debugging it. Moreover, these benefits are distributed over time. On the other hand, the costs of developing the unit tests are paid up front, and are paid by the original developer. Particularly if there is not already a good unit testing infrastructure in place, there may be considerable overhead in getting things set up before any benefits at all can be realized.

This asymmetry poses a considerable barrier to adoption for the first developer to try to use unit testing on a project. This developer will be paying all the costs initially, but realizing only a fraction of the benefit. As a result, without strong management support, this developer risks being seen as less productive, with the associated career risk. It is much better if a team can adopt or experiment with unit testing together, where they can realize benefits from each other's unit tests and share the risks. But in a culture such as EDA development, where there is no tradition of unit testing, it may be difficult to persuade others to go along with such a change in development style.

## 5. Alternatives to Unit Testing

While persistent unit tests that are shared by the whole development team are the "gold standard" of testability, many EDA developers may find themselves in environments where it is not practical to develop such tests for code they are working on. Perhaps management is deeply skeptical of the value of unit testing, or policy is such that all persistent tests need to be run against a full tool executable. Or perhaps you have a unit test infrastructure, but there isn't time to develop persistent tests that cover the particular code you are working on. What can the conscientious developer who wants to develop reliable code do in such an environment? Here are some suggestions that one or both of us have used at various times in our careers.

### 5.1 One-off testing (in debugger, or ad-hoc use)

Debuggers are not just static tools that allow you to observe the behavior of running code without affecting it. Most debuggers have the ability to set variables and call functions while in the debugger, though those abilities may be somewhat limited (e.g., the MS Visual Studio C++ debugger does not allow you to call templated functions directly from a watch window – but you can call a standard function that invokes a template call). This allows the debugger to be used for one-off testing – if you want to see what a function does when the second argument is 10, set a breakpoint just before a call to that function, change the variable value, and observe the results. Dynamically-typed languanges are particularly useful to test in a debugger, since you can construct arbitrary arguments and observe the results. Statically-typed languages don't usually allow you to create new variables on the fly, but it can sometimes be useful to have some global pointers of appropriately-selected types coded into a program just for debugging.

Likewise, it can sometimes be useful to write a short throw-away program that exercises a particular function, even in the absence of a larger unit test environment. Such testing is not as useful to the rest of the team as persistent unit tests can be in documenting and verifying the code against future changes – but at any rate you can have some assurance that the code has been tried at least once on a particular corner case.

### 5.2 Log inputs and outputs; look at usage

What happens when you want to know what a function does in various corner cases, but aren't able to or don't have the time to write individual unit tests? In that position, we once inserted temporary logging code at the start and end of a function, writing the input and output values to a log file. Then we ran a full system test suite, and dredged through the logs to see if the cases we were interested in happened to occur naturally. We've used a similar approach to find which test cases in a large suite exercise particular functions in an existing code base.

### 5.3 Export function interface to tool command line

If your test organization and infrastructure only allows tests to be performed on the full tool, one way to convert unit tests into system tests is to export the inputs and outputs of internal functions into (undocumented) command line functionality, such as exporting it into a Tcl function that can be called from a tool command line. Then the unit test can be written against such "system" behavior. One has to be careful with this approach, as it can expose behavior and information about the internal workings of the tool to end users – but at least it allows the code to be tested.

### 5.4 Beyond Unit Testing

A fair number of bugs can be caught today by static checkers such as Coverity® Static Analysis[17] or Gimple Software's PC-lint[18]. Dynamic checking using symbolic execution in KLEE[19] has shown promise in finding bugs in C code and covering execution paths not found in manual tests. Perhaps someday this approach may be used for the automated generation of unit test cases, much as ATPG is used today to generate hardware test patterns.

## 6. Coding for Testability Guidelines

Here are some rough guidelines for writing more testable code:

1) Keep it short. Massively long routines are hard to test thoroughly. A block of lines that implements a single concept should usually be extracted into its own routine, which creates a seam for observability and controllability.

2) Make it easy to instantiate stuff. If you can construct an object without a lot of preliminaries, it makes it easier to construct one to support another test.

3) Beware mega classes. A class with dozens of methods and responsibilities is hard to test. A

class that is composed of simpler (and separately testable) interfaces is much easier.

4) Keep files focused. A single file that defines lots of classes is hard to test, and likely to lead to link dependencies. A file should generally define a single class, or a small set of related classes.
5) Modularity is your friend. Testable code tends to be more focused and modular.
6) Write tests along with the code. Even if you don't adopt full-out Test Driven Development (and the complexity of EDA algorithms can make this difficult), writing tests in parallel with writing code can make it easier to test.
7) Write readable tests. The easier it is to look at a test and see what it is doing, the easier it is to understand what the code is expected to do, and whether the test covers what we think it does.

## 7. Blue Pearl Experience

At Blue Pearl Software, we have been refactoring and developing unit tests for a large (635K lines of C++, not counting external libraries) EDA tool over the past several months. This is the start of a multi-year project to improve the quality of our software and cover much of our software with unit tests.

Our primary build tool is Cmake [20]. We use the open-source Google Test (gtest) and Google Mock (gmock) [15,21] to define the unit tests and build one or more unit test executables per library, and Ctest[20] as the framework to run the test executables. Gtest and gmock come with Cmake configuration files, and were easily integrated into our source tree as external imports. For additional refactoring support, we use Visual Assist X[22].

The full set of unit tests developed to date is run on all platforms to qualify a build for QA system testing – if the unit tests fail, the build is rejected and the QA tests are not run. Individual developers can run the full set of unit tests or just the tests associated with a particular library or suite.

For the moment, developing unit tests has been primarily a background task as we experiment with how to bring significant sections of code under test and explore how unit tests fit with various projects. As of this writing, we have written around 5000 lines of unit tests. If we assume that we will eventually need around a 1:1 ratio of test code to product code, we stand at around 0.8% of our goal over the next few years. Our overall quality strategy remains multi-pronged, with system-level tests still carrying the majority of the quality burden. But we have found that unit-level tests offer much more detailed testing of the code, which will be important as we move forward and bring larger sections of the code under test.

Given our current level of coverage, most of the case studies we can offer are either examples where we have used unit tests to explore surprising behavior of existing code, or where unit tests could have saved time and broken builds had they been in place.

### 7.1 Platform Conversion

One area where even minimal unit tests were useful to us was when adding support for a 64-bit build on a new version of one of our platforms. The unit tests flagged some minor differences between the 32 and 64 bit builds during the build process. Unfortunately, we did not yet have unit tests covering another area of the code, where a sentinel value of (size_t) -1 was being used to flag deleted objects that had not yet been removed. Some of the legacy routines manipulating those objects passed an unsigned int as the type instead, which failed to match on a 64-bit platform. Having unit tests covering basic insertion and deletion of objects from the enclosing data structure would have saved us a couple of days of broken builds while trying to debug some rather bizarre behavior resulting from an inconsistent data structure.

### 7.2 Identifying Surprising Behavior

Unit tests can be helpful in identifying surprising behavior from legacy code. For example, here are some tests we developed for a utility data value class that was being used both in simulation and to encode module parameter values:

```
drDataValue dv1;
dv1.setDecimalValue(10);
EXPECT_EQ(10,dv1.getDecimalValue());
EXPECT_DOUBLE_EQ(10.0,dv1.getRealValue());
EXPECT_EQ("10", dv1.getValueString());

dv1.setRealValue(10.0);
EXPECT_EQ(10, dv1.getDecimalValue());
EXPECT_DOUBLE_EQ(10.0, dv1.getRealValue());
EXPECT_EQ("10", dv1.getValueString());
```

So far, it seemed to be a nice utility class that provided type conversion as a bonus. But the test of the setStringValue method proved surprising (note: these are not the original expected values, but the values that were reported back from our first, failing, unit test as to what the API actually returned):

```
dv1.setStringValue("10");
EXPECT_EQ(2,dv1.getDecimalValue());
EXPECT_DOUBLE_EQ(2.0,dv1.getRealValue());
EXPECT_EQ("2'b10", dv1.getValueString());
```

The setStringValue method interpreted its argument string as a four-valued (01XZ) binary value. This surprising behavior (especially since the API also provided a method called setBinaryValue that did almost the same thing) was not documented in the header file that defined the class. The behavior was described in a comment in the implementation code, but not every user of an API can be expected to read the full implementation. Such surprising behavior is a bug waiting to happen when the code is reused or extended. This API has since been revised and documented in both the unit tests and the header file.

### 7.3 Testing Complex Data Structures

One reason that EDA unit tests can be challenging to write, at least initially, is that so much of our code is data-structure driven. Our use cases tend to involve various configurations of nets, ports, instances, and so on, all of which must be hooked up properly in the test setup because the code under test will be navigating those data structures directly. Crucial differences in behavior may be dependent on the exact settings of various flags and attributes of different objects within a netlist. Keeping tests readable as the setup gets longer and more complex is challenging.

These issues have come up for us in writing unit tests for some code that interacts with a third-party HDL parser. One of the key questions has been whether to include the full parser in these unit tests, driving them from Verilog or VHDL sources, or instead to build the test cases up out of objects from the parser database API. For the moment, we have taken the latter approach, which allows us greater control of the generated data structures, and removes possible dependencies on the parser version or options used. There is a tradeoff here: testing against the actual parser would mean that all our test cases could arise from actual user input, but that we might not cover all possible data structure configurations. Testing with constructed data structures means that we might construct cases that could not actually arise in practice, but allows us to help make our code more robust against any possible configuration.

We have been able to make these tests more readable by developing some test helper functions that construct certain stereotyped patterns of objects relevant to groups of tests. This takes more time initially, but makes development of subsequent tests easier.

## 8. Conclusions

The principles of controllability and observability govern software as well as hardware testing. If we are going to catch faults with our tests, we need to be able to make the faulty behavior happen, and we need to notice when it does. Traditional testing methods used in EDA don't do a very good job of providing controllability and observability of internal software components. Unit testing, coupled with refactoring of existing code to make it more testable, offers far more control and observability of software component behavior. At Blue Pearl Software, we are using unit testing to improve the quality of our software components and deliver better quality software to our customers.

## 9. REFERENCES

[1] M. Fowler et al, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[2] E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," *Proceedings of the 1977 Design Automation Conference*. Reprinted in *25 Years of Design Automation*, pp. 358-364.

[3] M. J. Y. Williams and J. B. Angell, "Enhancing Testability of Large Scale Integrated Circuits vis Test Points and Additional Logic," *IEEE Trans. Computers*, C-22 (1973), pp. 46-60.

[4] J.P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, Vol. 10, No. 4, July 1966, pp 278-291.

[5] M. C. Feathers, *Working Effectively With Legacy Code*, Prentice Hall PTR, Upper Saddle River, NJ, 2005.

[6] R. Osherove, *The Art of Unit Testing with Examples in .Net*, Manning Publications, Greenwich, CT, 2009.

[7] http://en.wikipedia.org/wiki/Unit_testing, retrieved 3/4/2012.

[8] http://softwaretestingfundamentals.com/unit-testing/, retrieved 3/4/2012

[9] http://c2.com/cgi/wiki?StandardDefinitionOfUnitTest, retrieved 3/4/2012

[10] S. McConnell, *Code Complete, Second Edition*, Microsoft Press, Redmond Washington, 2004.

[11] B. Marick, *The Craft of Software Testing: Subsystem Testing, Including Object-Based and Object-Oriented Testing*, Prentice Hall PTR, Englewood Cliffs, NJ, 1995.

[12] R.L. Glass, "Extreme Programming: The Good, the Bad, and the Bottom Line," *IEEE Software*, November/December 2001, pp 111-112.

[13] M. Fowler, "Mocks Aren't Stubs," http://martinfowler.com/articles/mocksArentStubs.html, January 2007, retrieved 3/6/2012.

[14] K. Beck, *Test-Driven Development: by Example*, Addison-Wesley, 2003.

[15] http://code.google.com/p/googletest

[16] J. Oravec, personal communication.

[17] http://www.coverity.com

[18] http://www.gimpel.com

[19] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), December 2008.

[20] http://www.cmake.org

[21] http://code.google.com/p/googlemock

[22] http://www.wholetomato.com