Combining Dynamic Testing and Static Verification with VectorCAST and Polyspace

Introduction
This paper describes how to leverage the benefits of dynamic testing and static verification. The salient features of both software testing techniques are presented and how they are best implemented in automated tools. It explains how both of these techniques complement each other – and how to leverage the synergies between them. Finally, it presents the VectorCAST integration with MathWorks® Polyspace®.

Dynamic Testing and Static Analysis
The differences between dynamic testing and static analysis at large can be confusing. This is to be expected, after all, both techniques (not to mention tool vendors!) promise similar benefits. Benefits such as; higher code quality, less bugs, faster testing, and so on. But the fact is that both techniques focus on very different methods.

Dynamic Analysis
Dynamic analysis refers to how a software system reacts to various input data. This is done by actually executing the software and verifying that the output, or expected result is as expected. There are different types of testing that encompass dynamic testing. Unit Tests, Integration Tests, and System Tests, are all considered dynamic testing methodologies.

Static Analysis
Static analysis is a large category of techniques with different approaches to code quality but with a common methodology in that the source code is analyzed versus being executed. On one end of the spectrum, there are tools for coding standards checking. The idea behind such tools is that code is written to respect certain rules of a programming language. If the rules are followed, the software will be inherently safer and more maintainable.

Other static analysis techniques focus on error detection rather than error avoidance. But these techniques are limited to a certain subset of bugs – specifically, runtime errors. These errors are born out of arithmetic violations of the language that lead to unreliable program operation. Examples of such errors include; out-of-bounds arrays, divide by zero, and faulty pointer operations.
What is static verification
Among the different static analysis techniques, static verification stands out for its ability to exhaustively detect runtime errors such as the ones mentioned previously. This technique accomplishes this by running different sets of algorithms to determine the realistic range of values that can be experienced on specific variables at different points in the program. These algorithms use the very structure of the code to progressively refine ranges of values from the full range of the variable type, down to a subset of that range, to no valid values at all.
Here is a trivial code example of how static verification can accomplish this task.

```c
static int m_w;
int m_y;
int m_z[10];
void myFunction(int x)
{
    int i;
    if(x > 10)
    {
        m_y = i / x;
        x = m_y;
    }
    else
    {
        if(x >= 0)
        {
            m_y = x/m_w;
            m_z[m_y] = 0;
        }
    }
    for(i = 0; x > 10 && i < 10; i++)
    {
        x++;
    }
}
```

One of the first potential runtime errors we must be aware of is the possibility of a division by zero on the operation contained in the first “if” statement. But clearly, “x” cannot be zero in the context. We can therefore conclude that this error would never occur. On the other hand, the division “x/m_w” (part of the else statement) would always return an error if we start the analysis on this function, as “m_w” is a static int and will be initialized to zero.

There are also two other areas of potential problems that may go unnoticed.

First, it should be noticed that the “m_z[x]” access may be out of bounds, depending on the input values of “x”. Finally, the “for loop” will never execute, as “x” at this point in the code cannot be higher than 10.
The MathWorks® Polyspace® tool implements static verification. Based on this technique, it would color the code according to whether a potential runtime error (1) can never occur (in green), (2) will never occur (in red), or (3) cannot be mathematically reached (in gray). Finally, when (4) an error can sometimes occur depending on specific input values to the program or (5) the algorithm were unable to definitively determine whether a runtime error may occur or not occur, the operand will be highlighted in orange.

Comparison between static verification and dynamic analysis

Static verification can be of great help detecting runtime errors. What it does not do, however, is determine whether the functions defined in the program actually do what the requirements expect them to do. In other words, static verification will not find functional errors. A runtime error free program will run reliably, but it does not mean it will run correctly.

Dynamic testing is used for verifying functional correctness. The idea behind dynamic testing is to create test cases that associate specific input values to output values (expected values) and execute the tests to ensure the program is accomplishing its goals. These values need not be scalars – they can be ranges of values, lists of values, etc. But in each case, we are testing the program against its functional requirements. Obviously, the closer the execution is to the actual target, the more relevant the tests are, which is why industry standards recommend or mandate tests be conducted on the target processor and board, or on a simulator.
From this analysis, we can clearly see that static verification and dynamic testing are highly complementary to each other. It is not a question of which technique is better. The fact is that for software to be of high quality, it should be both functionally correct and exempt of runtime errors.

**Other Requirements**
There are some areas of overlap. For instance, you can still find a runtime error using dynamic testing with specific test case data – but you then need to use the specific combination of values that will trigger the error. This is useful when testing the robustness of a particular function by using the maximum or minimum type values as input. But static verification would be able to accelerate the discovery of such bugs by alleviating the need to guess what these combinations of values might be.

**Going beyond complementarity - synergies between static verification and dynamic testing**
One might say that some overlap – minimal as it is – is necessary to ensure bugs do not slip through the cracks. But it goes much further – these areas are actually the key to exploit the synergies existing between static verification and dynamic testing.

One important area of synergy is to further refine the runtime error analysis in the case of potential intermittent errors. In Polyspace, these errors are categorized as an unproven warning – they may either be an error that can occur from time to time, or it may actually never occur but the analysis could not sufficiently restrict realistic variable ranges to confirm it.

Dynamic testing may help here by providing a way to determine what needs be to done for such operations. Based on the range of values provided by static verification, it is possible to devise additional test cases that will attempt to trigger the error. Some of these errors will cause the program to fail, while others will simply corrupt it, so it is important to build a test case that can detect both situations.

A unit test tool like VectorCAST is particularly adept at this because it enables you to quickly create test cases at a unit level and test a variety of values in one test case. After all, if an error can be triggered at the unit level, the possibility exists that it will be triggered during operation.

On the other hand, if the potential error cannot be triggered, we cannot conclude that it could not occur, as it is impractical and most of the time impossible to run all test cases. But it would significantly increase the level of confidence in the code and enable testers to make better judgments about which of these warnings could be real errors and which ones are unlikely to cause trouble.
Another synergy that can be exploited is to accelerate code coverage. This metric is based on the idea that ideally all code should be executed during testing – if some code is not run then we cannot conclude that these snippets of uncovered code will be functionally correct and not cause problems in the future. However, it can sometimes be difficult to figure out how to cover specific parts of very complex code.

Static verification can help here, as the analysis determines a subset of realistic values that variables can take at different code locations. This information can then be used to devise additional test cases – which would be particularly easy to do at unit level. For instance, if a tool like Polyspace reports some code is mathematically unreachable, we won’t be able to cover it no matter the number of tests we generate. We must therefore try to understand why that code was implemented in the first place and amend it as to allow it to be covered while meeting the program’s requirements.
Using the Polyspace integration within VectorCAST

In order to fully exploit the synergies existing between dynamic testing and static verification, Vector Software has developed, in collaboration with MathWorks, a seamless integration between the two tools. The rationale behind the integration is to enable users to easily create Polyspace environments from VectorCAST by recycling the project information already collected.

Whenever a VectorCAST unit/integration testing environment (in either Ada, C, or C++) is open, the Polyspace integration can be easily launched from the VectorCAST graphical user interface. The integration can also be launched from VectorCAST/Cover. This will allow an analysis on Ada or C files (Polyspace only allows C++ files to be analyzed on a per class basis, and thus support for this language in the integration is limited to unit/integration test).

Upon launching the integration, the following window will appear. In unit/integration testing, the integration will detect the programming language used automatically. Please notice that if you launch from VectorCAST/Cover, you will need to specify the language of the files you wish to analyze.
The following options are available:

- You can select to have the Polyspace analysis run on the Client (local machine) or to be spooled to a remote server.
- If you are using Polyspace R2009b or earlier, you will need to check the first checkbox, as the way to call Polyspace from the command line changed from R2009b to R2010a.
- If you wish to use the “includes” or macros set in VectorCAST during the Polyspace analysis, you may do so at this point in time. You can also specify that you do not want Polyspace to generate a dummy main function (which is the behavior by default of the integration).
- You can specify any number of additional Polyspace arguments in the text box.

Once you press on the “Launch analysis” button, VectorCAST will call Polyspace from the command line and run the analysis. If using C++, the integration will request you identify the class to be analyzed (from a selection of classes based on an analysis of the code under test). Once the analysis is finished, it will close the command line window it opened (unless you override this behavior by clicking on the last check box, which may be useful for debugging purposes), and if using the Polyspace Client the results of the analysis will be opened automatically (if spooling to a Server, you will need to retrieve the results manually).

Advantages

The VectorCAST / Polyspace integration is a prime example of Vector Software’s best-of-breed approach to verification and validation. Instead of providing a variety of tools based on different techniques, Vector Software focuses on building the best dynamic test tools in the industry, while providing meaningful integrations with top-of-their-class tools, such as Polyspace.

Furthermore, MathWorks recommend running Polyspace tools on a file-by-file (or class-by-class) basis. This helps reduce the time it takes to analyze code, and it also helps users ensure their code is not only reliable, but robust as well. Thanks to the VectorCAST Polyspace integration, it is possible to launch such an analysis seamlessly from any unit test environment. Likewise, having both a unit test environment and a Polyspace project based on the same code configuration running in parallel greatly helps exploit the synergies we explained above, as it becomes easy and practical to switch between both tools to accelerate code coverage and potential runtime error (“oranges”) confirmation.
About Vector Software

Vector Software, Inc., is the leading independent provider of automated software testing tools for developers of safety critical embedded applications. Vector Software’s VectorCAST line of products, automate and manage the complex tasks associated with unit, integration, and system level testing. VectorCAST products support the C, C++, and Ada programming languages.