{coders} kitchen

How to Evaluate Embedded Software Test Tools

Jeffrey Fortin

Published on 10/29/2020

{Coders}
Kitchen

Table of Contents

Jeffrey Fortin

{Coders}
Kitchen

1 What is in your test tool?

Over the past few years the test automation tool market has become cluttered with tools that all claim to do the same thing: Automate Testing. Wikipedia lists over 100 test framework tools for C/C++ alone. Unfortunately for potential users, when viewing product literature, or simplistic demos, many of these test tools look very much alike.

The purpose of this white paper is to provide data that engineers should consider when they are evaluating software test automation tools, specifically dynamic test automation tools.

2 You can't evaluate a test tool by reading a data sheet

All data sheets look pretty much alike. The buzzwords are the same: "Industry Leader," "Unique Technology," "Automated Testing," and "Advanced Techniques." The screen shots are similar: "Bar Charts," "Flow Charts," "HTML reports," and "Status percentages." It's mind numbing.

3 What is software testing?

All of us who have done software testing realize that testing comes in many flavors. For simplicity, we will use three terms in this paper:

- **System Testing:** Testing the fully integrated application
- **Integration Testing:** Testing integrated sub-systems
- **Unit Testing:** Testing a few individual files or classes

Everyone does some amount of system testing where they do some of the same things with it that the end users will do. Notice that we said "some" and not "all." One of the most common causes of applications being fielded with bugs is that unexpected, and therefore untested, combinations of inputs are encountered by the application when in the field.

Not as many folks do integration testing, and even fewer do unit testing. If you have done integration or unit testing, you are probably painfully aware of the amount of test code that has to be generated to isolate a single file or group of files from the rest of the application. At the most stringent levels of testing, it is not uncommon for the amount of test code written to be larger than the amount of application code being tested. As a result, these levels of testing are generally applied to mission- and safety-critical applications in markets such as aviation, medical devices, and railway.

4 What does "automated testing" mean?

It is well known that the process of unit and integration testing manually is very expensive and time consuming. As a result, every tool that is being sold into this market will trumpet "Automated Testing" as their benefit. But what is "automated testing"? Automation means different things to different people. To many engineers the promise of "automated testing" means that they can press a button and they will either get a "green check" indicating that their code is correct, or a "red x" indicating failure.

Unfortunately, this tool does not exist. More importantly, if this tool did exist, would you want to use it? Think about it. What would it mean for a tool to tell you that your code is "OK"? Would it mean that the code is formatted nicely? Maybe. Would it mean that it conforms to your coding standards? Maybe. Would it mean that your code is correct? Emphatically No!

Complete automated testing is not attainable nor is it desirable. Automation should address those parts of the testing process that are algorithmic in nature and labor intensive. This frees the software engineer to do higher value testing work such as designing better and more complete tests.

The logical question to be asked when evaluating tools is: "How much automation does this tool provide?" This is the large gray area and the primary area of uncertainty when an organization attempts to calculate an ROI for tool investment.

5 Anatomy of test tools

Test Tools generally provide a variety of functionality. The names vendors use will be different for different tools, and some functionality may be missing from some tools. For a common frame of reference, we have chosen the following names for the "modules" that might exist in the test tools you are evaluating.

Parser	The parser module allows the tool to understand your code. It reads the code, and creates an intermediate representation for the code (usually in a tree structure) - basically the same as the compiler does. The output, or "parse data" is generally saved in an intermediate language (IL) file.
CodeGen	The code generator module uses the "parse data" to construct the test harness source code.
Test Harness	While the test harness is not specifically part of the tool, the decisions made in the test harness architecture affect all other features of the tool. So, the harness architecture is very important when evaluating a tool.
Compiler	The compiler module allows the test tool to invoke the compiler to compile and link the test harness components.
Target	The target module allows tests to be easily run in a variety of runtime environments including support for emulators, simulators, embedded debuggers, and commercial RTOS (real-time operating system).
Test Editor	The test editor allows the user to use either a scripting language or a sophisticated graphical user interface (GUI) to set up preconditions and expected values (pass/fail criteria) for test cases
Coverage	The coverage module allows the user to get reports on what parts of the code are executed by each test.
Reporting	The reporting module allows the various captured data to be compiled into project documentation.
CLI	A command line interface (CLI) allows further automation of the use of the tool, allowing the tool to be invoked from scripts, make, etc.
Regression	The regression module allows tests that are created against one version of the application to be re-run against new versions.
Integrations	Integrations with third-party tools can be an interesting way to leverage your investment in a test tool. Common integrations are with configuration management, requirements management tools, and static analysis tools.

Table 1: Anatomy of test tools

Later sections will elaborate on how you should evaluate each of these modules in your candidate tools.

{Coders}
Kitchen

6 Classes of test tools and levels of automation

In addition to the different methods for unit testing the code, ISO 26262 also recommends four different strategies to generate test cases during unit testing. They are listed as follows.

Manual	"Manual" tools generally create an empty framework for the test harness, and require you to hand-code the test data and logic required to implement the test cases. Often, they will provide a scripting language and/or a set of library functions that can be used to do common things like test assertions or create formatted reports for test documentation.
Semi-Automated	"Semi-Automated" tools may put a graphical interface on some Automated functionality provided by a "manual" tool but will still require hand-coding and/or scripting in order to test more complex constructs. Additionally, a "semi-automated" tool may be missing some of the modules that an "automated" tool has. Built in support for target deployment, for example.
Automated	"Automated" tools will address each of the functional areas or modules listed in the previous section. Tools in this class will not require manual hand coding and will support all language constructs as well a variety of target deployments.

Table 2: Classes of Test Tools and Levels of Automation

7 Subtle tool differences

In addition to comparing tool features and automation levels, it is also important to evaluate and compare the test approach used. For example, when you create a test project with some tools, the tool will simply load the files into its IDE but not do any of the work of creating the test harness or the test cases until you try to do something.

This may hide latent defects in the tool, so it is important to not just load your code into the tool, but to also try to build some simple test cases for each method in the class that you are testing. Does the tool build a complete test harness? Are all stubs created automatically? Can you use the GUI to define parameters and global data for the test cases, or are you required to write code as you would if you were testing manually?

In a similar way target support varies greatly between tools. Be wary if a vendor says: "We support all compilers and all targets out of the box." These are code words for: "You do all the work to make our tool work in your environment."

8 How to evaluate test tools

The following few sections will describe, in detail, information that you should investigate during the evaluation of a software testing tool. Ideally you should confirm this information with hands-on testing of each tool being considered.

Since the rest of this paper is fairly technical, we would like to explain some of the conventions used. For each section, we have a title that describes an issue to be considered, a description of why the issue is important, and a "Key Points" section to summarize concrete items to be considered.

Also, while we are talking about conventions, we should also make note of terminology. The term "function" refers to either a C function or a C++ class method. "Unit" refers to a C file or a C++ class. Finally, please remember, almost every tool can somehow support the items mentioned in the "Key Points" sections. Your job is to evaluate how automated, easy to use, and complete the support is.

8.1 Parser and code generator

It is relatively easy to build a parser for C; however, it is very difficult to build a complete parser for C++. One of the questions to be answered during tool evaluation should be: "How robust and mature is the parser technology?" Some tool vendors use commercial parser technology that they licensed from parser technology companies and some have homegrown parsers that they have built themselves. The robustness of the parser and code generator can be verified by evaluating the tool with complex code constructs that are representative of the code to be used for your project.

Key Points

- Is the parser technology commercial or homegrown?
- What languages are supported?
- Are tool versions for C and C++ the same tool or different?
- Is the entire C++ language implemented, or are their restrictions?
- Does the tool work with our most complicated code?

8.2 The test driver

The Test Driver is the "main program" that controls the test. Here is a simple example of a driver that will test the sine function from the standard C library:

```c
#include <math.h>
#include <stdio.h>
int main () {
        float local;
        local = sin (90.0);
        if (local == 1.0) printf ("My Test Passed!\n");
        else printf ("My Test Failed!\n");
        return 0;
}
```

Although this is a pretty simple example, a "manual" tool might require you to type (and debug) this little snippet of code by hand, a "semi-automated" tool might give you some sort of scripting language or simple GUI to enter the stimulus value for sine. An "automated" tool would have a full-featured GUI for building test cases, integrated code coverage analysis, an integrated debugger, and an integrated target deployment.

I wonder if you noticed that this driver has a bug. The bug is that the sin function actually uses radians not degrees for the input angle.

Key Points

- Is the driver automatically generated or do I write the code?
- Can I test the following without writing any code?
  - Testing over a range of values
  - Combinatorial Testing
  - Data Partition Testing (Equivalence Sets)
  - Lists of input values
  - Lists of expected values
  - Exceptions as expected values
  - Signal handling
- Can I set up a sequence of calls to different methods in the same test?

8.3 Stubbing dependent functions

Building replacements for dependent functions is necessary when you want to control the values that a dependent function returns during a test. Stubbing is a really important part of integration and unit testing, because it allows you to isolate the code under test from other parts of your application, and more easily stimulate the execution of the unit or sub-system of interest.

Many tools require the manual generation of the test code to make a stub do anything more than return a static scalar value (return 0).

Key Points

- Are stubs automatically generated, or do you write code for them?
- Are complex outputs supported automatically (structures, classes)?
- Can each call of the stub return a different value?
- Does the stub keep track of how many times it was called?
- Does the stub keep track of the input parameters over multiple calls?
- Can you stub calls to the standard C library functions like malloc?

8.4 Test data

There are two basic approaches that "semi-automated" and "automated" tools use to implement test cases. One is a "data-driven" architecture, and the other is a "single-test" architecture.

For a data-driven architecture, the test harness is created for all of the units under test and supports all of the functions defined in those units. When a test is to be run, the tool simply provides the stimulus data across a data stream such as a file handle or a physical interface like a UART.

For a "single-test" architecture, each time a test is run, the tool will build the test driver for that test, and compile and link it into an executable. A couple of points on this: first, all the extra code generation required by the single-test method, and compiling and linking, will take more time at test execution time; second, you end up building a separate test harness for each test case.

This means that a candidate tool might appear to work for some nominal cases but might not work correctly for more complex tests.

Key Points

- Is the test harness data driven?
- How long does it take to execute a test case (including any code generation and compiling time)?
- Can the test cases be edited outside of the test tool IDE?
- If not, have I done enough free play with the tool with complex code examples to understand any limitations.

8.5 Automated generation of test data

Some "automated" tools provide a degree of automated test case creation. Different approaches are used to do this. The following paragraphs describe some of these approaches.

MMM **Min-Mid-Max Test Cases**	MMM tests will stress a function at the bounds of the input data types. C and C++ code often will not protect itself against out-of-bound inputs. The engineer has some functional range in their mind, and they often do not protect themselves against out of range inputs.
EC **Equivalence Classes**	EC tests create "partitions" for each data type and select a sample of values from each partition. The assumption is that values from the same partition will stimulate the application in a similar way.
RV **Random Values**	RV tests will set combinations of random values for each of the parameters of a function.
BP **Basis Path Tests**	BP tests use the basis path analysis to examine the unique paths that exist through a procedure. BP tests can automatically create a high level of branch coverage.

Table 3: Automated Generation of Test Data

The key thing to keep in mind when thinking about automatic test case construction is the purpose that it serves. Automated tests are good for testing the robustness of the application code, but not the correctness (even if they provide a high level of code coverage). For correctness, you must create tests that are based on what the application is supposed to do (the requirements), not what it does do (the code).

8.6 Compiler integration

The point of the compiler integration is two-fold. One point is to allow the test harness components to be compiled and linked automatically without the user having to figure out the compiler options needed. The other point is to allow the test tool to honor any language extensions that are unique to the compiler being used. Especially with cross-compilers, it is very common for the compiler to provide extensions that are not part of the C/C++ language standards. Some tools use the approach of #defining these extensions to null strings. This very crude approach is especially bad because it changes the object code that the compiler produces. For example, consider the following global extern with a GCC attribute.

extern int MyGlobal __attribute__ ((aligned (16)));

If your candidate tool does not maintain the attribute when defining the global object MyGlobal, then code will behave differently during testing than it will when deployed because the memory will not be aligned the same.

Key Points
- Does the tool automatically compile and link the test harness?
- Does the tool honor and implement compiler-specific language extension?
- What type of interface is there to the compiler (IDE, CLI, etc.)?
- Does the tool have an interface to import project settings from your development environment, or must they be manually imported?
- If the tool does import project settings, is this import feature general purpose or limited to specific compiler or compiler families?
- Is the tool integrated with your debugger to allow you to debug tests?

8.7 Support for testing on an embedded target

In this section we will use the term "Tool Chain" to refer to the total cross-development environment, including the cross-compiler, debug interface (emulator), target board, and Real-Time Operating System (RTOS). It is important to consider if the candidate tools have robust target integrations for your tool chain and to understand what in the tool needs to change if you migrate to a different tool chain.

Additionally, it is important to understand the automation level and robustness of the target integration. As mentioned earlier, if a vendor says: "We support all compilers and all targets out of the box." They mean: "You do all the work to make our tool work in your environment."

Ideally, the tool that you select will allow for "push button" test execution where all of the complexity of downloading to the target and capturing the test results back to the host is abstracted into the "Test Execution" feature so that no special user actions are required.

An additional complication with embedded target testing is hardware availability. Often, the hardware is being developed in parallel with the software, or there is limited hardware availability. A key feature is the ability to start testing in a native environment and later transition to the actual hardware. Ideally, the tool artifacts are hardware independent.

Key Points
- Is my tool chain supported? If not, can it be supported? What does "supported" mean?
- Can I build tests on a host system and later use them for target testing?
- How does the test harness get downloaded to the target?
- How are the test results captured back to the host?
- What targets, cross compilers, and RTOS are supported off-the-shelf?
- Who builds the support for a new tool chain?
- Is any part of the tool chain integration user configurable?

8.8 Test case editor

Obviously, the test case editor is where you will spend most of your interactive time using a test tool. If there is true automation of the previous items mentioned in this paper, then the amount of time attributable to setting up the test environment, and the target connection should be minimal. Remember what we said at the start, you want to use the engineer's time to design better and more complete tests.

The key question to answer when conducting your evaluation is, how hard is it to set up test input and expected values for non-trivial constructs? All tools in this market provide some easy way to setup scalar values. For example, does your candidate tool provide a simple and intuitive way to construct a class? How about an abstract way to setup an STL container; like a vector or a map? These are the things to evaluate in the test case editor.

As with the rest of this white paper there is "support" and then there is "automated support." Take this into account when evaluating constructs that may be of interest to you.

Key Points
- Are allowed ranges for scalar values shown?
- Are array sizes shown?
- Is it easy to set Min and Max values with tags rather than values? This is important to maintain the integrity of the test if a type changes.
- Are special floating-point numbers supported (e.g.; NaN, +/- Infinity)?
- Can you do combinatorial tests (vary 5 parameters over a range and have the tool do all combinations of those values)?
- Is the editor "base aware" so that you can easily enter values in alternate bases like hex, octal, and binary?
- For expected results, can you easily enter absolute tolerances (e.g.; +/- 0.05) and relative tolerances (e.g.; +/- 1%) for floating point values?
- Can test data be easily imported from other sources like Excel?

8.9 Code coverage

Most "semi-automated" tools and all "automated" tools have some code coverage facility built in that allows you to see metrics which show the portion of the application that is executed by your test cases. Some tools present this information in table form. Some show flow graphs, and some show annotated source listings. While tables are good as a summary, if you are trying to achieve 100% code coverage, an annotated source listing is the best. Such a listing will show the original source code file with colorations for covered, partially covered, and uncovered constructs. This allows you to easily see the additional test cases that are needed to reach 100% coverage.

It is also important to understand the impact of instrumentation; the additional source code added to your application. There are two considerations: one is the increase in size of the object code, and the other is the run-time overhead. It is important to understand if your application is memory or real-time limited (or both). This will help you focus on which item is most important for your application.

Key Points
- What is the code size increase for each type of instrumentation?
- What is the run-time increase for each type of instrumentation?
- Can instrumentation be integrated into your "make" or "build" system?
- How are the coverage results presented to the user? Are there annotated listings with a graphical coverage browser, or just tables of metrics?
- How is the coverage information retrieved from the target? Is the process flexible? Can data be buffered in RAM?
- Are statement, branch (or decision), and MC/DC coverage supported?
- Can multiple coverage types be captured in one execution?
- Can coverage data be shared across multiple test environments (e.g. can some coverage be captured during system testing and be combined with the coverage from unit and integration testing)?
- Can you step through the test execution using the coverage data to see the flow of control through your application without using a debugger?
- Can you get aggregate coverage for all test runs in a single report?
- Can the tool be qualified for DO-178B/C and for Medical Device intended use?

8.10 Regression testing

There should be two basic goals for adopting a test tool. The primary goal is to save time testing. If you've read this so far, we imagine that you agree with that! The secondary goal is to allow the created tests to be leveraged over the life cycle of the application. This means that that the time and money invested in building tests should result in tests that are re-usable as the application changes over time and easy to configuration manage.

The major thing to evaluate in your candidate tool is what specific things need to be "saved" in order to run the same tests in the future and how the re-running of tests is controlled.

Key Points

- What file or files need to be configuration managed to regression test?
- Does the tool have a complete and documented Command Line Interface (CLI)?
- Are these files plain text or binary? This affects your ability to use a diff utility to evaluate changes over time.
- Do the harness files generated by the tool have to be configuration managed?
- Is there integration with configuration management tools?
- Create a test for a unit, now change the name of a parameter, and re-build your test environment. How long does this take? Is it complicated?
- Does the tool support database technology and statistical graphs to allow trend analysis of test execution and code coverage over time?
- Can you test multiple baselines of code with the same set of test cases automatically?
- Is distributed testing supported to allow portions of the tests to be run on different physical machines to speed up testing?

8.11 Reporting

Most tools will provide similar reporting. Minimally, they should create an easy to understand report showing the inputs, expected outputs, actual outputs and a comparison of the expected and actual values.

Key Points

- What output formats are supported? HTML? Text? CSV? XML?
- Is it simple to get both a high level (project-wide) report as well as a detailed report for a single function?
- Is the report content user configurable?
- Is the report format user configurable?

8.12 Integration with other tools

Regardless of the quality or usefulness of any particular tool, all tools need to operate in a multi-vendor environment. A lot of time and money has been spent by big companies buying little companies with an idea of offering "the tool" that will do everything for everybody.

The interesting thing is that most often with these mega tool suites, the whole is a lot less than the sum of the parts. It seems that companies often take four to five pretty cool small tools and integrate them into one bulky and unusable tool.

Beyond the integration with the development tool chain that we already covered, the most useful integrations for test tools are with static analysis, configuration management, and requirements management tools. Everyone wants to put their testing artifacts under configuration control so that they can re-use them, and most people want to trace their requirements to test cases

Key Points

- Which tools does your candidate tool integrate with out of the box, and can the end user add integrations?

8.13 Additional desirable features for a testing tool

OK, so we've finished the review of: "The Anatomy of a Test Tool." The previous sections all describe functionality that should be in any tool that is considered an automated test tool. In the next few sections, we will list some desirable (although

{Coders}
Kitchen

less common) features along with a rationale for the importance of the feature. These features may have varying levels of applicability to your particular project.

8.14 True integration testing, multiple units under test

Integration testing is an extension of unit testing. It is used to check interfaces between units and requires you to combine units that make up some functional process. Many tools claim to support integration testing by linking the object code for real units with the test harness. This method builds multiple files within the test harness executable but provides no ability to stimulate the functions within these additional units. Ideally, you would be able to stimulate any function within any unit in any order within a single test case. Testing the interfaces between units will generally uncover a lot of hidden assumptions and bugs in the application. In fact, integration testing may be a good first step for those projects that have no history of unit testing.

Key Points
- Can I include multiple units in the test environment?
- Can I create complex test scenarios for these classes where we stimulate a sequence of functions across multiple units within one test case?
- Can I capture code coverage metrics for multiple units?

8.15 Dynamic stubbing

Dynamic stubbing means that you can turn individual function stubs on and off dynamically. This allows you to create a test for a single function with all other functions stubbed (even if they exist in the same unit as the function under test). For very complicated code, this is a great feature and it makes testing much easier to implement.

Key Points
- Can stubs be chosen at the function level, or only the unit level?
- Can function stubs be turned on and off per test case?
- Are the function stubs automatically generated? (see items in previous section)

8.16 Library and application level thread testing (System Testing)

One of the challenges of system testing is that the test stimulus provided to the fully integrated application may require a user pushing buttons, flipping switches, or typing at a console. If the application is embedded the inputs can be even more complicated to control. Suppose you could stimulate your fully integrated application at the function level, similar to how integration testing is done. This would allow you to build complex test scenarios that rely only on the API of the application.

Some of the more modern tools allow you to test this way. An additional benefit of this mode of testing is that you do not need the source code to test the application. You simply need the definition of the API (generally the header files). This methodology allows testers an automated and scriptable way to perform system testing.

8.17 Agile testing and test driven development (TDD)

Test Driven Development promises to bring testing into the development process earlier than ever before. Instead of writing application code first and then your unit tests as an afterthought, you build your tests before your application code. This is a popular new approach to development and enforces a "test first and test often" approach. Your automated tool should support this method of testing if you plan to use an Agile Development methodology.

{Coders}
Kitchen

8.18 Bi-directional integration with requirements tools

If you care about associating requirements with test cases, then it is desirable for a test tool to integrate with a requirements management tool. If you are interested in this feature, it is important that the interface be bi-directional, so that when requirements are tagged to test cases, the test case information such as test name and pass / fail status can be pushed back to your requirements database. This will allow you to get a sense of the completeness of your requirements testing.

8.19 Tool qualification

If you are operating in a regulated environment such as commercial aviation or Class III medical devices, then you are obligated to "qualify" the development tools used to build and test your application.

The qualification involves documenting what the tool is supposed to do (the tool operational requirements) and tests that prove that the tool operates in accordance with those requirements. Ideally, a vendor will have these materials off-the-shelf and a history of customers that have used the qualification data for your industry.

Key Points

- Does the tool vendor offer qualification materials that are produced for your exact target environment and tool chain?
- What projects have successfully used these materials?
- How are the materials licensed?
- How are the materials customized and approved for a particular project?
- If this is an FAA project, have the qualification materials been successfully used to certify to DO-178B/C Level A?
- If it is an FDA project, have the tools been qualified for intended use?

8.20 Conclusion

Hopefully this paper provides useful information that helps you to navigate the offerings of test tool vendors. The relative importance of each of the items raised will be different for different projects.

Our final suggestions are:

- Evaluate the candidate tools on code that is representative of the complexity of the code in your application
- Evaluate the candidate tools with the same tool chain that will be used for your project
- Talk to long-term customers of the vendor and ask them some of the questions raised in this paper
- Ask about the tool's technical support team. Try them out by submitting some questions directly to their support team, rather than to their sales representative

Finally, remember that most every embedded software test tool can somehow support the items mentioned in the "Key Points" sections. Your job is to evaluate how automated, easy to use, and complete the support is.

Imprint

Vector Informatik GmbH
Ingersheimer Str. 24
70499 Stuttgart
Germany

Phone: +49 711-80670-0
E-mail: info@vector.com

www.coderskitchen.com