Code Coverage:

Free Software and Virtualization to the Rescue

Franco Gasperoni, AdaCore
gasperoni@adacore.com

What is Code Coverage and Why Is It Useful?

Your team is developing or updating an embedded application based on a set of requirements. If you do not intend to conduct a testing campaign to verify compliance of the code with the requirements don't read any further. Code coverage will not be of much use. If you are still reading you probably plan to create or upgrade your collection of tests and ensure that what has been developed adheres to the requirements. Whether your testing campaign relies on unit/functional/robustness/… tests or a mixture of these you will probably wonder about the quality of your test suite and more specifically ***how to ascertain that the test suite is reasonably complete***.

Code coverage allows you to answer this question. As you run the test suite on your application, a code coverage tool tells you which portions of the application are really being exerted by the testing campaign. Let's get down to earth and work trough an example.

Let's assume that we have been asked to sort an array of floats (our requirement). To meet this requirement we implement the insertion sort routine given in Figure 1.

```
type Float_Array is array (Integer range <>) of Float;

procedure Insertion_Sort (A : in out Float_Array) is
   Value : Float;
   I     : Integer;
begin
   for W in A'First .. A'Last loop
      Value := A (W);
      I := W;
      while I > A'First and then A (I - 1) > Value loop
         A (I) := A (I - 1);
         I := I - 1;
      end loop;
      if I < W then
         A (I) := Value;
      end if;
   end loop;
end Insertion_Sort;
```

Figure 1: A simple insertion sort algorithm.

If we test our insertion sort routine using Test_1 = (1.0, 2.5, 3.141) code coverage shows that lines 11, 12, and 15 are not executed since the condition "`A (I - 1) > Value`" on line 10 and condition "`I < W`" on line 14 will always be false.

If Test_1 is our only test vector, code coverage shows that we have tested our insertion sort algorithm only roughly. If we also use test vector Test_2 = (7.5, 2.3, 1.0, 0.5) then all source code lines in Figure 1 will be executed.

What Should Be Covered?

With test vectors Test_1 and Test_2, code coverage shows that we have exerted all lines in our implementation of the insertion sort algorithm. Is this good enough? This depends on how critical is our application.

In the context of safety-critical embedded avionics applications, for instance, the DO-178B [1] "standard" focuses on requirements-based testing and distinguishes 5 levels of software criticality (from Level E – not critical at all to Level A – very safety-critical). Levels C, B, and A, are those where the DO-178B expects a certain degree of code coverage to ascertain that we have a reasonably complete test suite with respect to the requirements.

If our insertion sort algorithm is used in a level of criticality C application then test vectors Test_1 and Test_2 are good enough. As we go up to Level B and Level A criticality and as the likelihood of injuries and loss of lives increases should the embedded software malfunction, more stringent coverage requirements apply.

In Level B, for instance, covering each line of source code must be complemented with decision coverage, i.e. covering each decision branch. In our insertion sort example we must provide a test vector to try things out when the for-loop at line 7 is never executed. If our insertion sort algorithm is to fly as part of a Level B application we must add an empty array as test vector to our test suite.

For the highest level of software criticality, Level A, instruction and decision coverage have to be complemented with modified condition/decision coverage (MC/DC). In MC/DC we have to test that each condition in a decision independently affects the decision's outcome. In our insertion sort example the only decision with more than one condition is in line 10 where there are two conditions:

Condition 1: `I > A'First`
Condition 2: `A (I - 1) > Value`

Test vectors 1 and 2 have tested two of the three combinations of interest, namely:

Condition 1 = True and Condition 2 = False (test vector 1);
Condition 1 = True and Condition 2 = True (test vector 2).

We are missing a test for Condition 1 = False. This is easily addressed by providing an additional test with a single element array.

Obtaining Code Coverage Information

In the previous sections we have explained what code coverage is and its usefulness in assessing the quality of a test suite in regard to a set of requirements according to the level of criticality of the tested software. This section surveys the options to obtain code coverage information. In all of the following options coverage information is related back to the original source.

Instrumenting the Source

With this technique instrumentation is added, either by hand or with a tool, to the source code to log coverage information. This is the approach followed by several tool vendors.

This approach has the advantage of providing a straightforward coverage mapping between the source code and the information logged. Furthermore, it does not require changing the compiler and is fairly simple to put in place for unit tests that run both on the host computer and the target hardware.

This approach has several drawbacks. Firstly it is programming language specific. From a tools vendor perspective this means a tool per programming language supported. If the application being tested uses several programming languages this means using different tools for the application. In addition instrumentation needs to store the log somewhere, typically in a file, which is problematic on embedded systems without a file system. Finally this approach is very intrusive and modifies the object code generated by the compiler which may result in different behaviors between the instrumented code and the final embedded application.

To circumvent some of these problems, when possible, developers execute the instrumented test suite on the host computer, while the un-instrumented test suite is run on the target hardware. The results of the two runs are then compared to ensure that the same behavior is obtained. For large test suites this is a further disadvantage of this approach.

Instrumenting the Object

In this technique the compiler is modified in such a way that instrumentation is inserted directly at the object level during compilation. Instrumentation is inserted only when requested by the developer with a special compiler switch. GCC, the GNU Compiler Collection, has the ability to extract code coverage information by instrumenting the object following this approach. When the executable is tested, instrumentation logs are accrued in a file (one file per object). A separate tool, called gcov in the case of GCC, reads the instrumentation logs, gathers the generated information and maps it back to the original source code [2].

The advantage of this approach with respect to the previous one is its programming language independence, in the case of GCC at least. Otherwise, all the other disadvantages of direct source-level instrumentation are present with object-level instrumentation.

Extracting an Execution Trace

With this technique the object code trace is obtained by some means, typically via a JTAG or Nexus probe, or by single stepping the processor. This trace is then related back to the original source code to determine coverage. This last step isn't straightforward. It requires a mapping between the source and the object code. One possibility is exploiting the debug information included in the executable.

This approach is programming language independent. In addition, and foremost, this approach is non intrusive and allows to extract coverage information directly from the final application that is to run on the target hardware.

In terms of disadvantages this approach can be slow. In addition, when executing the application on the target hardware, a connection between the target hardware and the host where the trace is saved is still necessary. The alternative of logging the data on the board requires significant amounts of storage that is typically not available on embedded boards. If a probe is used to collect the execution trace, extra specialized hardware needs to be used. Finally, in the presence of compiler optimizations, relating trace information back to the source to obtain source code coverage is not always a simple task.

Free Software Coverage Tools: The Current Generation

The Free Software/Open Source coverage tool of reference is GCC/gcov. As previously mentioned the coverage approach taken by GCC/gcov is object-level instrumentation. This provides a convenient and easy-to-use source-coverage capability in the case of unit tests that can run on the host.

If the testing campaign is based on unit tests, and GCC is used on the host, and there is no need for DO-178B qualification material, we are almost set. One missing piece is a tool/IDE to visualize in a friendly and synthetic way what lines in the source are and aren't covered. Free Software tools such as LCOV [3] or the GPS IDE [4] address this need. Depending on the project's criticality and its risk management policy, the last missing piece may be an internal or external provider to rely on for periodic releases of the Free Software coverage tools along with the support and maintenance guarantees needed by the project. In the case of Ada and C, AdaCore [5] is one such provider. In the case of C++, CodeSourcery [6] is another.

Free Software Coverage Tools: The Next Generation

Thanks to public funds, the next generation of Free Software code coverage tools is on its way [7]. "Project Coverage" will produce a Free Software coverage analysis toolset together with artifacts that allow the tools to be used for safety-critical software projects undergoing a DO-178B software audit process for all levels of criticality.

While an important target use of the coverage toolset is safety-critical embedded applications, the design of the tools allows its use in non safety-critical circumstances.

Beyond the production of useful tools and certification material for industrial users, an important goal is to raise awareness and interest about safety-critical and certification issues in the Free Software/Open Source community.

The key insight of "Project Coverage" is as follows: code coverage can greatly benefit from recent advances in hardware virtualization technology [8] as promoted, for instance, by VMware [9] or QEMU [10]. See Figure 2.

By virtualizing the target hardware, "Project Coverage" tools can execute the binary code that is to run on the target hardware as-is on a host computer. While executing the target code on the host, "Project Coverage" tools collect binary branch information. The collected information is then analyzed off-line

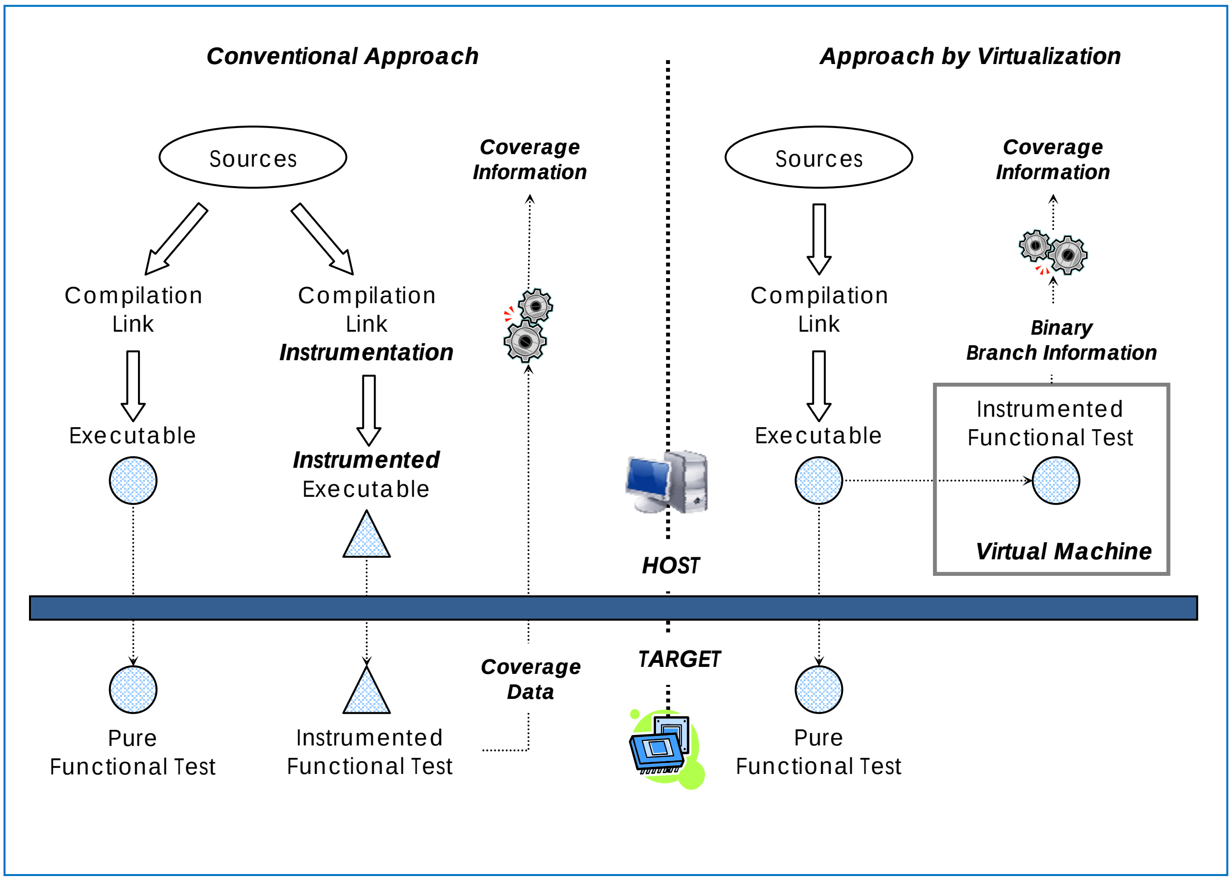

Figure 2: Obtaining code coverage information by hardware virtualization.

and mapped back to the original sources by using the source to object code mapping information extracted from the debugging information contained in the executable. We are basing this part of our work on the DWARF standard for debugging information that the majority of compilation chains are capable of generating.

Our virtualization technology is based on QEMU that we are extending, first to output binary branch coverage information, and second to make it usable in industrial contexts typically found in the avionics domain (MIL-STD-1553, ARINC 629, etc.).

Because QEMU works by compiling the target object code into the host object code and that the host computer is typically faster than the target one, virtualization is actually a plus over direct execution on the target.

The approach put forth by “Project Coverage” features several strong points:

(1) “Project Coverage” tools are easy to use and deploy since they run on the host computer;

(2) “Project Coverage” tools work for all compiled programming languages and compilers that can output DWARF debugging information and can be easily adjoined to existing development environments;

(3) Compared to a “gcov”-based approach, the “Project Coverage” solution is completely decoupled from the compiler. This means that one can change/freeze compilers and coverage tools independently of each other. This is unlike the “gcov” approach which is tightly coupled to the compiler so a change in the compiler implies a change of “gcov” and more to the point - a change in “gcov” will force a compiler change. This tight coupling between compiler and “gcov” coverage toolset is a factor of risk that is avoided with the “Project Coverage” approach.

(4) “Project Coverage” tools are non-intrusive and capable of working directly with the final executable;

(5) No specialized hardware is required to extract coverage information;

(6) Thanks to the difference in speed, memory, and file system requirements between the target and host computer, the process of extracting coverage information on the host by virtualizing the target hardware compares favorably in terms of speed with current approaches to gather coverage information;

In addition to the above:

(7) Users of “Project Coverage” will obtain DO-178B qualification material when necessary;

(8) “Project Coverage” tools will be freely available and its industrial users will have the option to purchase high-quality professional support.

In summary “Project Coverage” is a clever combination of several unrelated trends in today’s software technology landscape (Free Software, Virtualization, DWARF, DO-178B qualification …) to produce a unique code coverage solution that safety-critical and non safety-critical developers can use in their projects.

References

[1] EUROCAE, RTCA: “ED-12B/DO-178B: Software Considerations in Airborne Systems and Equipment Certification”. See http://www.eurocae.eu/ or http://www.rtca.org/.

[2] “gcov—a Test Coverage Program”. See http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[3] "The LTP GCOV extension (LCOV)". See http://ltp.sourceforge.net/coverage/lcov.php.

[4] "The GNAT Programming Studio". See http://libre.adacore.com/gps/.

[5] http://www.adacore.com/.

[6] http://www.codesourcery.com/.

[7] http://libre.adacore.com/coverage/.

[8] The Economist: "Virtualization: The Rise of the Hypervisor". January 17, 2008. Available at http://www.economist.com/business/displaystory.cfm?story_id=10534566.

[9] http://www.vmware.com/.

[10] http://en.wikipedia.org/wiki/QEMU.