

C++ Target Testing

Neil Langmead, IPL Information Processing Ltd

This paper considers the evolution of target testing over the last decade, evaluating the host/ target testing divide, and examining current trends. Well-defined, lightweight language subsets such as embedded C++ have emerged to optimise target executable size and increase portability in embedded applications. As compiler vendors implement scalable embedded C++ solutions, the ability to test the language in a standard way becomes more difficult. Care must be taken to preserve the benefits of code size and portability while testing. The problems of testing templates and operating system calls are considered, and some solutions are presented.

The Past

Since the advent of high level languages, the practice of developing software in a different environment to the one in which it will eventually be used has become commonplace. The reasons for this are varied, but limited access to target hardware and a lack of a convenient operating system or file input and output all played their part. With access to the target restricted and time consuming, and memory expensive, most developers elected to perform as much implementation and testing work on the host as possible. Target testing was difficult; lack of automation and tool support meant that too often it was also non-rigorous.

The huge growth in the PC market during the 1990's changed the face of target testing, with PCs being used as a development host for embedded systems, and also as a host to develop software for mini and mainframe systems. For reasons of economy and efficiency host-based testing has grown in popularity [IPL 1, 1996]. This growth occurred despite the inherent philosophical flaws in host-based testing.

The Present – New IDEas

Recently, there has been an explosion in the number of complete, integrated development environments (IDEs) offering access to the real target, software emulation, host-based analysis and debugging – all under one roof. IDEs offer many convenient software-based interfaces (such as option setters and windows for viewing simulated output), simulators, debuggers and emulators. They offer developers a complete target development solution, with all the advantages and convenience of host based development. All host and target tools are integrated in a unified environment to simplify all phases of application development.

IDE deployment has led to a classic dilemma regarding testing. Are tests performed in a host-based software simulator enough to prove that the code will run successfully on the target? To do so requires proof (or faith) that the software simulation correctly mimics target execution of the same code. We must also prove that the cross compiler used produces semantically equivalent object code when targeting the simulator and the real target. This is difficult to verify, as IDEs 'hide' a lot of processing from the developer.

More importantly though, as the contrived example in Figure 1 shows, it is entirely possible for the code to behave differently on the host and the target. As the integer `x` is declared volatile, the otherwise unreachable code in the second `if` statement becomes dynamically reachable, since the value of `x` can change (by for example, a target system interrupt) between the first and second `if` conditions.

```

/* Figure 1 */
extern volatile int x;
int foo()
{
    if (x) {
/* some processing */
        if( !x) {
            }
        }
    return 0;
}

```

Thus host based testing alone on code designed for deployment in a target environment is inadequate, and it is necessary to perform at least some code validation on the target system itself. This assertion remains true in the case of IDEs unless we can prove (in the mathematical sense) equivalence between the target and host based simulation.

Target testing has also been influenced by the choice of implementation language. In fact, the testability¹ of the target application has a

dependence on the implementation language. This is especially the case with the C++ language, which presents unique challenges to the tester [Dorman, 1997].

C++ - as a language for targets

C++ is one of the most elegant and sophisticated programming languages available to the developer. Its power is rooted in the concept of the class, a natural extension to data structures found in the C language. Programmers like C++ because it is more powerful than C, providing object oriented features such as polymorphism and inheritance that greatly simplify the development process [Haden, 2001].

The downside to C++'s rich feature set is that compiled object code can be several times larger than the equivalent code compiled in C, leading to increased memory requirements and a reduction in execution speed. On host systems, these are perhaps of superficial concern – the benefits of C++ (component reuse and faster development) usually outweigh performance issues. For embedded applications though, memory and execution speed are critical.

Embedded C++ (hereafter EC++) evolved to take advantage of C++'s most useful features, but at the same time discarding those features responsible for code bloat, such as multiple inheritance, exceptions, runtime type information and virtual method tables. This makes it a suitable language for embedded developers [EC++ Technical Committee Rationale, 1998].

Testing C++

Traditionally, testing solutions are implemented in the form of reusable test 'libraries' of C++ routines. These help a tester by providing a structured framework in which to carry out testing in a controlled, repeatable, auditable manner. For testing to be carried out on the target, these libraries must have been compiled with the target compiler. The usual method of dynamic testing is to write a test script (in C++), which uses the test library routines, and calls the software under test. This script is then compiled with the source under test and linked with the test libraries, and any other libraries required by the software under test.

Memory and performance issues apply as much to dynamic test code as they do to released code. How does using EC++ affect dynamic testing? Clearly, any dynamic test script must itself conform to the embedded subset used for development. Moreover, support for those C++ features not allowed in EC++ must also be disabled in supporting test libraries. Much of the code bloat in C++ does not come from using features such as templates, but from referencing templates found in large C++

¹ i.e. how easy the code is to test. All code is testable, although some pieces of code are easier to test than others.

libraries. Including such references often means compiling and linking in many parts of the library that simply aren't needed [Haden, 2001].

Use of the EC++ library leads to around one quarter reduction in code size, compared to the full ISO C++ library [Plauger, 1998]. This code saving is only advantageous if other libraries used (e.g. test libraries) also conform to the EC++ subset.

Scalable C++ Solutions

The problem is compounded by *scalability*, offered by many embedded compiler vendors. Scalable compiler solutions enables programmers to select C++ features not included in the baseline definition, and add these to their applications. For example, some compilers will allow developers to turn on support for templates or exceptions. Thus the need arises for supporting libraries to be similarly scalable.

In Examples 1 and 2, we consider two implementations of a simple `Array` class – one which conforms entirely to EC++, the other which uses EC++ with a scalable feature: templates. Suppose for now that the class has no external dependencies.

```
// Example 1 (a)
// Array.cpp
class Array {
public:
    Array();
    Array(int);
    int& elem(int i) { return v[i];}
    int& operator[](int i);
private:
    int* v;
    int size;
};

// test.cpp
void check_func() { // test
Array ar; // instantiate object
// call and check
check(ar.elem(3), 5);
}
```

A dynamic test of the EC++ implementation should be straightforward. Initialise the software by creating an instance (called `ar` say) of the `Array` class, call methods in the class, and check their returns using the test library. A coverage analyser may be used to determine criteria for the completion of testing. Use of a test harness library implemented in EC++ should mean that the test process adds no significant overheads beyond those which are normally associated with test code and additional library linkage.

Now suppose that `Array` has a method called `status()` that checks the number of tasks in the `Array` is not more than the maximum allowed by the target's scheduler routine.

```

// Example 1 (b)
class task_manager {
public:
    task_manager();
    int getarraysize(); /* target specific function to determine
if there is enough space left in the array for allocation */
};
class Array {
public:
    Array() { size = 0; }; // default constructor
    char* Status() {
        int available_array_size = taskMan.getarraysize();
        if (available_heap_size == 0) {
            return "not enough memory, no allocation possible";
        } else {
            return "enough heap size remains, allocation possible";
        }
    } // end of status
private:
    task_manager taskMan;
    int size;
};

```

Until very recently, a dynamic test of the module in Example 1(b) would probably have consisted of a host test and a target test, performed with different compilers, different environments, and different libraries. With the host test, it is likely that there is no function equivalent to the target specific `getarraysize()` function in the `TaskManager` class. Therefore, a stub, or 'intelligent dummy routine' for this function, and perhaps the class itself, would have to be written. The implications of this are severe: the C++ stubbing problem [Dorman, 1997] has to be contended with. There is also a significant increase in test maintenance effort, due to the dependency of the stub on the original class implementation. If there is also a requirement for 100% decision coverage of this code, the stub would have to return different values on different calls, to ensure execution of both the true and false branch of the `status()` decision.

ISO C++ Issues

As well as this problem, the other main obstacle to separate host and target testing of C++ has been the lack of compliance (and convergence) to the ISO C++ standard between differing compilers. Most compilers have significant differences in the code they accept. Although the C++ standard is quite mature, the first fully compliant compiler only became available in 2001. Lack of compliance in ISO implementations has rendered separate host and target testing of C++ almost unworkable. The EC++ standard is much newer than the full C++ standard, and compiler compliance is correspondingly worse.

The introduction of IDEs, with their all-in-one approach, has changed the way host and target testing is considered. There is often no need to do separate host and target testing – all testing can be performed with the one compiler. Some target compilers can even compile for native host and target separately, and it is a simple compiler switch that determines the target architecture for which the object code is built. Thus, the problem of compiler implementations is avoided, by choosing to work with just the target compiler from the outset.

The problem of simulation remains though. Considering Example 1(b), our requirement for 100% decision coverage means that we still have to write a stub for

the `getarraysize()` function. This may not be possible – it may be a core system function that cannot be replaced. So how can we force the heap size to be zero, and thus execute the false branch of the `if` statement in the `status` function?

Wrapping

A technique called wrapping [Dorman, 1997] implements a solution to this problem. It allows the developer to ‘wrap’ calls to functions or class methods, and modify the return value being passed back to the calling function. The original function is still linked into the application, avoiding the need to provide a stub. Automatically generated “before” and “after” functions allow the tester to modify parameters passed to the function and to check and modify any return values (for a complete description of this technique, please see Dorman’s paper, and [IPL 2]). So, in Example 1(b), the tester can elect to wrap the call to `taskMan.getarraysize()` in the `status()` function, and set the return value to be 0 in the after function. With the ability to wrap on a per call basis, different returns can be passed back to the software under test on different function calls. This makes achieving the coverage criteria and ensuring correct verification of all parts of the software through dynamic testing much simpler.

EC++ With Templates

We now take the baseline EC++ implementation, and ‘scale’ it, adding support for templates. For this template version, we can assume that most `Arrays` will be `Arrays` of some pointer type. The primary reason for this is to preserve run time polymorphic behaviour. The default behaviour of most C++ implementations is to replicate the code for template functions, which is good for run time performance, but unless care is taken, code bloat can easily result [Stroustrup, 1997]. This can be avoided by careful use of template specialisations.

Example 2 implements a complete specialisation of the `Array` template. That is, there

```
// Example 2
template <class T> class Array {
public:
    Array();
    Array(int);
    T& elem(int i) { return v[i]; }
    T& operator[](int i);
private:
    T* v;
    int size;
};
// specialise to create general pointer class
template<> class Array<void*> {
public:
    void** p;
    void*& operator[] (int i);
};
template <class T> class Array<T*> : private Array<void*> {
public:
    typedef Array<void*> Base;
    Array() : Base() {}
    explicit Array(int i) : Base(i) { }
    T*& elem(int i) { return static_cast<T*&> (Base::elem(i)); }
    T*& operator[] (int i) { return static_cast<T*&>
(Base::operator[](i)); };
```

is no template parameter to specify or deduce when we use this specialisation. `Array<void*>` is used for `Arrays` declared like this: `Array<void*> vp;` Similarly, a partial specialisation is used for every `Array` of pointers and only for `Arrays` of pointers. This gives us a shared implementation for all `Arrays` of pointers.

This has been achieved without changing the interface to the class. We could of course give the general `Array` and the `Array` of pointers different names, but unsurprisingly, many forget to use the pointer classes and hence find their code to be much larger than expected. Replicated code like this can cost megabytes of code space, even in fairly small applications. By eliminating the time needed to compile additional versions of `Array` methods, compilation and link time is dramatically reduced. This is a good example of the more general approach of minimising code bloat by maximising the amount of shared code [Stroustrup, 1997].

Testing EC++ With Templates

How then can we test a scalable EC++ solution, making use of templates? As Plauger argues: “Change the type of one argument in one call to a template function and you can unwittingly double the amount of code generated to implement that function”.

Clearly, any test harness library used must also support templates. Moreover, the implementation of that library should be such that the templates used do not suffer from code bloat once any test harness templates have been instantiated. A general solution to this problem must be a highly configurable library, designed to implement EC++, but scalable to include any other features in use by the developer. A scalable solution has been developed by IPL to achieve this [IPL 3, 2001].

Thus, any test library must carefully implement both a complete, non-scaled EC++ version, and a version scaled to include whichever features of the C++ language the developer has chosen to include with his EC++ implementation.

Coverage Analysis is the objective measurement of acceptability, against which the effectiveness of test effort can be judged [IPL 4, 1999]. How can we measure the proportion of code executed, when the code is a template? On its own, the code in Example 2 is not executable (although most coverage analysers would not pick up on this fact). It is not until the template is instantiated that the compiler (usually) provides bodies and implementations for the class. Hence, the template class can be regarded as a pseudo code generator – the code does not exist until the use of that template. Also, there are theoretically an infinite number of instantiations for a given template class. Objective measurement of template code execution is possible using an IPL developed technique [IPL 5, 2001].

The Future

We can argue that target testing has come full circle. Expensive and difficult a decade ago, target testing has been simplified with the introduction of powerful host based solutions. The emergence of these new powerful IDEs has been influential in seeing a return to target only testing, although the authenticity of IDE based testing has been challenged.

Specific simulation and coverage problems have been discussed, and solutions proposed. The traditional divide between host and target testing is fast disappearing, as are many of the problems associated with the separate approaches. Problems with differing C++ implementations are also being overcome, although the plethora of scalable options now available to the embedded C++ developer will pose new challenges for testing in the future. As tool vendors catch on and implement similarly scalable solutions, the traditional difficulties with target testing seem destined to be consigned to history.

REFERENCES

Dorman, 1997

C++ - "It's Testing, Jim, But Not As We Know It"

Proceedings of the Fifth European Conference in Software Testing, Analysis and Review, Edinburgh, Scotland, November 1997

Embedded C++ Technical Committee

Version WP-RA-003, Copyright(C) 20, Nov. 1998

Rationale for the Embedded C++ specification

<http://www.caravan.net/ec2plus/rationale.html>

Hadden, Mike

Integrated Communications Design, 2001

IPL 1, 1996

Host- Target Testing

Copyright IPL Information Processing Ltd, 1996

IPL 2, 2001

Cantata++ Source Modification Tool Reference Manual

Copyright IPL Information Processing Ltd, 2001

<http://www.iplbath.com/products/prodsupp/pdf/sm21.pdf>

IPL 3, 2001

Cantata++ Test Harness Reference Manual

Copyright IPL Information Processing Ltd, 2001

<http://www.iplbath.com/products/prodsupp/pdf/th21.pdf>

IPL 4, 1999

Cantata++ - A Management Presentation

Copyright IPL Information Processing Ltd, 1999

<http://www.iplbath.com/products/prodsupp/pdf/sm21.pdf>

IPL 5, 2001

Cantata++ Coverage Instrumentation Reference Manual

Copyright IPL Information Processing Ltd, 2001

<http://www.iplbath.com/products/prodsupp/pdf/ca21.pdf>

Plauger, P.J.

Proceedings of the Embedded Systems Conference, Chicago, Illinois, March 1999

<http://www.dinkumware.com/embed9710.html>

Stroustrup, Bjarne

The C++ Programming Language, Third Edition

Published by Addison- Wesley