The Design of a Test Case Definition Language

David Byers Magnus Engström Mariam Kamkar
Department of Computer Science
Linköping University
{davby,x96magen,marka}@ida.liu.se

Abstract

Unit testing of software requires the construction of large amounts of supporting software. Test drivers and stubs are needed to isolate the unit under test from other units, and test cases specifications need to be translated into compilable code. Constructing test software is as time-consuming and as fault-prone as any other software, and sometimes more so. To support testers there are tools that can automatically generate test code from test case specifications that are similar to the test case specification. These tools include a test case definition language, in which the test cases are stated, and a test script generator, that generates compilable code from the test case definitions. Even so, testers often have to resort to writing source code for test cases requiring features unavailable in the test case definition language. To learn more about how a better test case definition language could be constructed, we have designed and implemented a test script generator and a test case definition language. This paper gives an overview of some of our experiences and some of our ideas for future work.

1 Introduction

Software testing, particularly unit testing, requires the construction of significant amounts of test scaffolding software: drivers for executing functions under test and checking results and stubs for simulating code that is not yet in place. Such scaffolding software has a number of properties that are undesirable in testing.

Test scaffolding software may be as large and as complex as the unit under test. Since the effort spent developing such software may not significantly impact the quality or reliability of the software under test and since testing and test software development is often done under severe time pressure, there is little incentive to produce anything other than a "quick and dirty" solution. If the test scaffolding software if faulty it may classify faulty software as correct or correct software as faulty. The latter type of misclassification is not so serious, but the former can be extremely expensive if the faults are not discovered until late in the testing process or even after delivery. Re-use of test-cases is an important factor in regression testing. Test cases embedded in program code are seldom easy to re-use.

Instead of writing test scaffolding code manually, test cases can be defined using a specialized language, a test case definition language. The test scaffolding

code is then generated from the unit under test and the test case definitions by a test script generator (see figure 1. This arrangement relieves the tester of much of the work not directly related to testing and can hopefully prevent faults in test code.

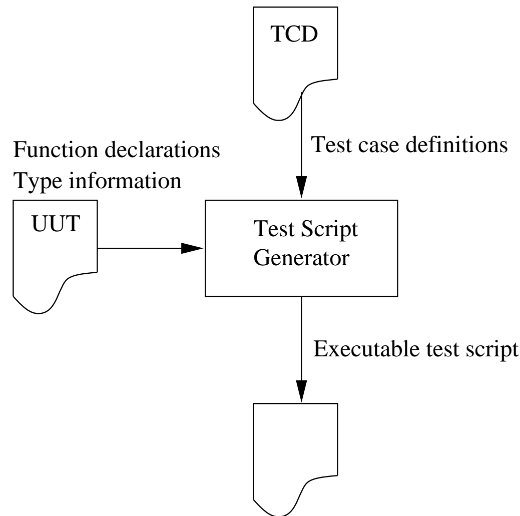

Figure 1: System structure

Test script generators exist as commercial tools, but very little has been published on the design of test case definition languages or the principles of how such tools could work. We have designed and implemented a test case definition language based on similar principles as the Cantata Test Case Definition language [2] and ideas from the work done on TSL [3, 1]. In this paper we present some of our experiences with designing the language and implementing a test script generator for it.

2 Design Issues

The exact requirements of a test case definition language depends on the environment in which it is used. The implementation language used in the software under test impacts what type of features can be tested, and the type of application impacts what properties of the software need to be tested.

In a procedural language such as C, the test case definition language should support calling procedures and functions, examining return values and contents of output parameters, setting and examining global variables, and the definition of stubs. In an object-oriented language such as C++, the language should also support creation of objects, calling methods, setting and examining instance variables, calling of overloaded function and perhaps scaffolding not only of code but of classes.

When testing real-time software the test-case definition language should support checking real-time aspects such as maximum response time or that response time does not exceed some threshold more than a certain amount of the time.

In safety-critical systems the test case definition language should be able to determine if the system ever arrives in an unsafe state. In state machines the test case definition language should be able to check that the test case traverses the state graph in the correct manner. The list goes on and on.

We have decided to start by defining a test-case definition language suitable for testing application software written in C or C++. Our goal was to design a readable language that is fairly complete with respect to features in the C and C++ languages so that testers would not be forced to resort to source code insertions. We have not examined the problems associated with object-orientation in great detail, but can demonstrate aspects of object-orientation that need special consideration in the design and implementation of the test case definition language.

2.1 The Structure of a Test Case

The basic structure of every test case in our language is shown in figure 2. The test case starts out with a descriptive section which can be used to identify a certain test case. That section is followed by a number of function calls, each of which has its own internal structure.

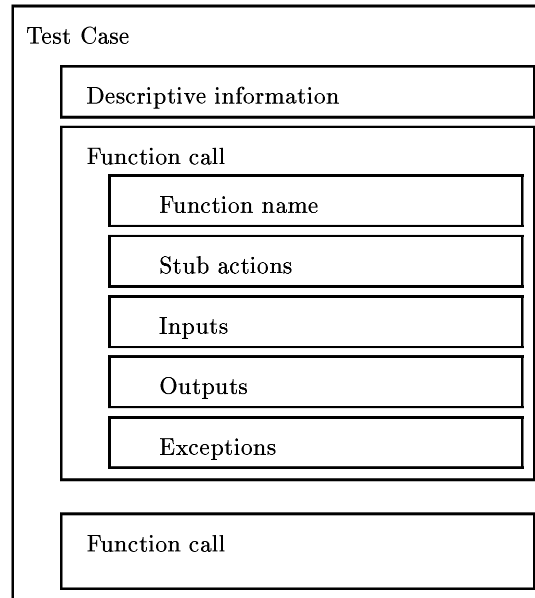

Figure 2: Test case structure

Each function call specification contains directions that determine the behavior of stubs, if any; the inputs to the function, the function itself and the expected outputs. There may also be code that checks which, if any, exceptions are thrown by the function.

To support object-oriented software, C++ in particular, the test case may contain directions for creating instances of classes, calling methods in specific instances, checking instance variables and calling overloaded functions.

There are a number of specific points which we found require careful attention when designing a test case definition language and a test script generator.

2.2 Function Calls

The simplest type of test case is a single function call. The test case needs to specify which function to call, which inputs to call it with, and which outputs are expected. Figure 3 shows an example of how a function can be called.

```
CALL
    FUNCTION integrate
    INPUTS
        function    "x^2"
        a           0
        b           5
        h           1
    OUTPUTS
        result      41.25
        MAP rt      0.208333
ENDCALL
```

Figure 3: A simple function call

2.2.1 Specifying Inputs

For most function, specifying inputs is not a problem. The way our test case definition language is designed, test cases are specified with no regard to the order between parameters. This property is useful in languages where a caller may use named parameters. By naming all parameters, the test case designer does not have to memorize their order and the test cases become more readable.

Some languages, such as C, allow functions with a variable number of parameters. This can be handled by a special name in the input list that represents additional parameters, but one problem remains. Our approach to function calls is to create variables that correspond to all input parameters, rather than supply values directly. To do so the test script generator must know the type of all input parameters. Normally it infers the type by examining the declaration of the function being called, but this is not possible for arguments beyond the formal parameter list. We overcome this problem by requiring the test case designer to specify the type of all additional parameters, as is shown in figure 4. Another possibility would be to attempt to infer the type from the value specified for the parameter.

2.2.2 Checking outputs

For most calls it is a simple matter to check that results are as expected, but there are a few exceptions.

In many systems there are functions that return user-defined types. Initially the test script generator knows nothing about these types, which prevents it from generating code to compare values of user-defined types. If user-defined types are to be checked, the tool needs to be told how to compare them. In our

```
CALL
    FUNCTION printf
    INPUTS
        fmt         "%s: result = %.2f"
        ARG1        string      "Warning"
        ARG2        double       24.67
ENDCALL
```

Figure 4: Calling a function with a variable number of arguments

tool the user can specify a comparison function for user-defined types, and the test script generator will generate calls to this function when required. A better solution would be to have the test script generator automatically infer how to compare user-defined types whenever it could.

Functions that return floating-point numbers pose another problem. Numerical calculations using floating point numbers are notoriously imprecise, and it is often the case that a small imprecision in the result is acceptable. For this type of function the user needs a way to specify how much imprecision is allowed before failing the test case.

2.3 Pointers

When a function in C or C++ has a pointer argument the test script generator must allocate space for the function to write in through the pointer. For instance, the standard function `fread` reads data from a file to a buffer allocated by the caller and indicated by a pointer argument. The test case definition language must also provide facilities for specifying the value a pointer points to before a call and the value it is expected to point to after a call.

2.4 Stubs

Stubs are used to isolate a unit under test from other units it depends on, by replacing functions in other units. A stub normally normally implements a subset of the full functionality of the function it replaces. Nevertheless, creating stubs can be a tedious and time-consuming exercise since they must closely simulate the true behavior of the stubbed function. We feel that a test case definition language should provide facilities for rapid development of stubs and for checking that the stubs are called in the manner expected in the test case.

There are several ways to facilitate writing stubs. A simple method, which is used by Cantata, a commercial testing tool, is to hard-code a number of actions into a stub and with each test case specify which action is to be taken on each invocation of the stub. Figure 5 shows a stub with two actions. The first action consists of checking that the input to the stub from the unit under test is the string `"Alyssa P. Hacker"` and then returns 45250. The second action is to check that the input to the stub is the string `"Ben Bitdiddle"`, in which case it is to return 32500.

Test cases in which the tested function calls stubs must contain a specification of which action is to be taken on each invocation of the stub. The

```
STUB get_salary
    ACTION 1
        INPUTS
            name    "Alyssa P. Hacker"
        OUTPUTS
            result  45250
    ENDACTION
    ACTION 2
        INPUTS
            name    "Ben Bitdiddle"
        OUTPUTS
            result  32500
    ENDACTION
ENDSTUB
```

Figure 5: A simple stub

example in figure 6 shows how the actions in `get_salary` are used. The first time `salary_difference` calls `get_salary`, action two is taken. The second time it is called, action one is taken.

```
CASE 1
    STUBS
        get_salary#2,get_salary#1
    END STUBS
    CALL
        FUNCTION salary_difference
        INPUTS
            name_1     "Alyssa P Hacker"
            name_2     "Ben Bitdiddle"
        OUTPUTS
            result 12750
    ENDCALL
ENDCASE
```

Figure 6: A test case specifying how a stub is called

A different approach to stub definition that might be appropriate under certain circumstances would be to declare a partial mapping between inputs and outputs. Figure 7 shows how the `get_salary` could be declared.

We think that it would be worthwhile to explore the use of stubs in the software industry and to develop better methods of defining stubs.

2.5 Source Code Insertion

No matter how much thought goes in to designing a test case definition language, there will be test cases that cannot be expressed in the language. Therefore it is necessary to allow testers to insert arbitrary source code in the test case definitions. Our language allows source code to be inserted at various points, such as before all test cases, at the beginning of a given test case and as an initializer for an input variable.

```
STUB get_salary
    "Alyssa P. Hacker" = 45250
    "Ben Bitdiddle"    = 32500
    *                  = FAIL
ENDSTUB
```

Figure 7: Declarative definition of stubs

2.6 Data abstraction

One of the much-touted features of modern programming languages is the ability to hide data and code from other software units. This abstraction is a desirable feature in software development. Testing on the other hand does not benefit from abstraction.

It is possible to test a software unit merely by calling its published interface, but this can impact negatively on the efficiency of testing. The ability to set local variables, call internal routines and check the value of hidden data can make testing much more efficient and accurate.

In languages such as Ada where the wall between units is impenetrable a test script generator should be able to generate code that bypasses the limitations of the language. This might even mean modifying the unit under test.

We have not addressed this issue in our tool, but recognize that it must be examined and dealt with more carefully.

2.7 Isolating Classes

Most object-oriented languages feature inheritance. A class inherits some methods from one or more other classes, overrides some and defines a few of its own. If classes are viewed as single units, then it is necessary to isolate classes not only from code that use methods in the classes, but from the entire class hierarchy. The situation is similar to, but somewhat more complex than, stubbing in procedural languages.

We have not examined this issue at all.

3 Related Work

We have been unable to find much work in the area of test case specification languages. Of the few commercially available languages, we have examined the language used by Cantata, a testing tool available from IPL Information Processing Ltd. Our language is similar to Cantata's in many ways, but we have given much thought to avoiding some of the difficulties encountered using Cantata, in particular the frequent need to write C or C++ code in test cases. Our model of stubs is nearly identical to that of Cantata.

In the research community, we have found very little published. Tom Ostrand and colleagues [3, 1] have presented a language for formal test specifications, TSL, that supports the category-partition method for generating test cases [3].

The category-partition method is based on the idea of classifying inputs to a program in categories, and then testing one instance of each category. With several inputs, all legal combinations of categories are tested. TSL supports generating test cases using the category-partition method by allowing the tester to specify which categories are available and how they may be combined. The test script generator then generates test cases corresponding to all legal combinations of categories.

A somewhat different approach is taken by Sun Microsystems' ADL [4] (Assertion Definition Language) and TDD (Test Data Description) languages. ADL is a formal grammar for specifying the interfaces and semantics of functions. TDD is a structured way of describing test data. Together, ADL and TDD allows automatic generation of complete test scripts. This is by no means a complete overview of the capabilities of ADL; ADL also has applications outside the area of testing.

There are a number of differences between the approach taken by ADL and our approach. Most of these differences stem from differences in the goals of the two languages. ADL tries to address a much wider spectrum of testing-related problems than we do, and this is reflected in the design of the language and the tools. The most important difference between the two languages is that in ADL the semantics of a function are specified separately from the test data, whereas our approach requires the tester to specify the expected output from a function explicitly for each test case. Both approaches have merits. The ADL approach is very flexible and allows for a high degree of reusability and test automation, at the cost of added complexity. Our approach is very easy to understand and to learn, at the cost of flexibility. Another difference is our mechanism for specifying stubs. ADL appears to have no provision for this at all.

4 Conclusions

A good test case definition language has the potential of automating much of the work that is currently done manually by testers. Such a language can also support automatic testing in maintenance through the use of databases of test case definitions and automatic generation of test code. The commercial test-case definition languages currently available support some of this but still require manual coding for complex test cases. Our attempt at defining a test case definition language that supports the features of several different languages show that there are many areas where research has an opportunity to provide better solutions, which in turn could help testers in a very tangible way.

Some of the points we have identified are

- The test case definition language could provide better support for defining stubs. Our model is useful but too primitive in many cases.
- Handling of user-defined types could be improved through more sophisticated code analysis.
- Testing in an object-oriented environment gives rise to certain problems that are absent in traditional programming languages, such as scaffolding of data types in addition to code. These issues require further research.

- Data abstraction is troublesome in test script generation. The test script generator needs to be able to bypass the restrictions imposed by the language used in the unit under test.

References

[1] Marc J. Balcer, William M. Hasling, and Thomas J. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the ACM SIGSOFT'89 Symposium on Software Testing Analysis and Verification (TAV3)*, number 5, Key West, Florida, December 1989. ACM SIGSOFT.

[2] IPL Information Processing Ltd., Bath. *Cantata Test Case Definition Language Reference Manual.* See also `http://www.iplbath.co.uk/`.

[3] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[4] Sun Microsystems, Inc. *ADL Translator User's Guide.* Sun Microsystems, Inc., 1996. See also `http://ADL.opengroup.org/`.