

Complete Test Generation for Extreme Programming

Mike Holcombe¹ and Florentin Ipate²

¹ Department of Computer Science
University of Sheffield, UK
m.holcombe@dcs.shef.ac.uk

² IFSOft, Romania
www.ifsoft.ro
fipate@ifsoft.ro

Abstract. Test generation is a key part of the *Extreme Programming* approach. This paper describes a very powerful functional testing method that generates test sets which can detect *all* possible faults in a computer program, provided some design for test conditions are satisfied. The basis for this *complete test generation* method is the *X-machine*, a simple and elegant way of visualising the dynamics of a program.

Keywords: Functional testing, unit testing, acceptance testing, test set generation, X-machines

1 Introduction

Testing is a major part of software development and in Extreme Programming the generation of test cases is a vital part of the initial phases of a project. In this paper, we present a method for generating test cases that provides a well founded approach to the problem of detecting *all* faults. The method is based on the computational modelling with *X-machines*, a sort of extended finite state machines, and can be integrated into Extreme Programming in a simple and designer-friendly way. It is a generalisation of the original X-machine based method [3], [1] that extends significantly its applicability and simplifies the testing process [4]. This generalisation is called in what follows the *complete* X-machine testing method. As the original method, the complete method assumes that some *design for test conditions* are satisfied.

The original X-machine based testing method [3], [1] assumes that the basic functions of the system are correct. This can be checked by a separate testing process, an effective way of doing this is to apply a functional method such as category partition and boundary value analysis. The simplest scenario is when tried and trusted components are used, for example, functions that take a keyboard input and echo it to a screen or put it in a register or perhaps a function that accesses a cell in a database table. However, if this is not the case, the original method will implicitly assume that each basic function can be tested *in*

isolation from the rest of the system. This is not always a realistic assumption since the implementations of the basic functions are not always distinct units of code (e.g. subroutines, modules, etc.) that can be separated from the rest of the system.

The complete testing method removes this condition and allows the testing of the basic functions to be performed along with the main testing process and the test cases generated for the basic functions to be integrated into the overall test cases generated for the entire X-machine.

2 A Simple Example

In order to illustrate the application of the complete testing method we use a simple customer orders database as described in [2]. We can identify a number of stories such as: entering customer details, editing customer details, placing orders, editing orders, etc.. Then we identify from these stories what is prompting change (inputs), what internal knowledge is needed (memory), what is the observable result (output) and how the memory changes after the event. From the diagram (Figure 1) one can see how the basic functions are organised. Each state, in this example, has associated with it an appropriate screen with buttons, text fields etc.

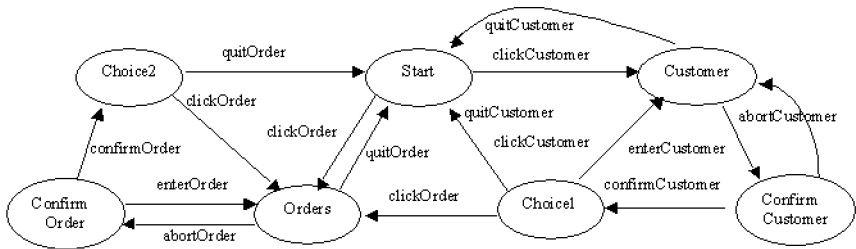


Fig. 1. The state transition diagram for the customer orders database

It is straightforward to generate a test set U_ϕ for each basic function ϕ using traditional functional methods such as category-partition and boundary value analysis. However, many of these functions do not fire in the initial state of the X-machine, so the context in which they are tested has to be taken into account as well, e.g. in order to apply the test set for *confirmCustomer*, one will first have to click the enter customer button and enter the customer details. Consequently the testing of a basic function will have to rely on the correctness of the functions that have been applied to set up the appropriate context, otherwise the testing results may be affected by the errors introduced earlier by these functions, e.g. when the test set for *confirmCustomer* is applied, one must assume that the correct customers details have been entered and recorded.

The solution to this problem is to test the basic functions in the order that they can be reached in the X-machine diagram. In this case, a basic function ϕ can always be reached by a sequence v_ϕ of other basic functions that have already been tested (and therefore been shown to be correct). Let us illustrate the process for our example.

We start with the basic functions that emerge from the initial state, *Start*. There are two such function, *clickCustomer* and *clickOrder*. Obviously the sequence $v_{clickCustomer} = v_{clickOrder} = \epsilon$, the empty sequence, and the test sets $U_{clickCustomer}$ and $U_{clickOrder}$ can be applied straight to the initial state of the X-machine.

We look now for the states that can be reached from *Start* by sequences made up only of the functions *clickCustomer* and *clickOrder*. There are two such states: *Customers* and *Orders*; *clickCustomer* takes the X-machine from the initial state *Start* to *Customer*, while *clickOrder* takes the X-machine from *Start* to *Order*. There are two functions, *enterCustomer* and *quitCustomer* that emerge from *Customer*. The sequence that reaches these functions is $v_{enterCustomer} = v_{quitCustomer} = customer_button_clicked$, where *customer_button_clicked* is the event (or input) corresponding to the clicking of the customer button on the start screen, which triggers the function *clickCustomer*. In order to test *enterCustomer* and *quitCustomer*, this input will be concatenated with the elements of $U_{enterCustomer}$ and $U_{quitCustomer}$, respectively. The two functions *enterOrder* and *quitOrder* are tested in a similar manner.

The next states visited will be those accessed by sequences made of the functions that have already tested (i.e. *clickCustomer*, *clickOrder*, *enterCustomer*, *enterOrder*, *quitCustomer*, *quitOrder*) and the basic functions that emerge from those states will be tested. The procedure will continue until all functions have been reached and tested. It is sufficient to test a basic function only once even if it appears in the diagram many times, e.g. *quitCustomer* will not be tested again when the state *Choice1* is reached.

Once the basic functions have been tested and shown to be correct we can proceed to generate the test set for the whole system. This is done in the following way. We start at the initial state *Start* with the initial memory value and the aim is to visit every state of the X-machine, e.g. in order to visit *ConfirmCustomer*, the sequence *clickCustomer* :: *enterCustomer* is processed (Here :: means concatenation or sequence connector.) When we have reached a state we need to confirm that it is the correct state and this is done by following more simple paths from that state until we get outputs that tell us, unambiguously, what the state was.

Then we repeat the path to that state and check what happens if we try to apply every basic function from that state, some will succeed but some should fail. Have the correct ones passed and failed? This is then repeated for every state. Some example functions sequences are:

$$clickCustomer :: enterCustomer :: confirmCustomer,$$

$$clickCustomer :: enterCustomer :: clickOrder.$$

The first test has tried to apply a correct function (i.e. *confirmCustomer*) from the state *ConfirmCustomer* and should pass, the second has tried to apply an incorrect function (i.e. *clickOrder*) from that state and should fail. Now, this test set is not quite what we want since it is based on the set of functions which we cannot access directly, it needs to be converted to a sequence of inputs. So we choose suitable inputs that will trigger the correct functions as we trace through the diagram along the paths of functions, generating sequences of inputs which are our actual tests. The design for test conditions allow this to happen, the mathematical details and proof of correctness are in [4]. Thus we have the following test sequences corresponding to the sequences above:

customer_button_clicked :: customer_details_entered :: confirm_button_clicked,
customer_button_clicked :: customer_details_entered :: orders_button_clicked.

Of course, as this is a high level test set, the input *customer_details_entered* represents a more complex series of activities. e.g. entering the *customer_name*, *customer_address*, etc. Since all basic functions have already been tested and shown to be correct, at this stage the input that triggers a function can be chosen at random.

The test generation, which is fully automated, will generate all the input sequences needed to establish whether the implementation is correct, i.e. agrees with the X-machine model.

3 Conclusions

The use of smart test strategies in XP can provide substantial gains in quality. This paper is an attempt to explain how one of the most powerful test generation approaches, the complete X-machine testing method, could be put to use. The complete testing method generalises a previous X-machine based method by extending significantly its applicability and simplifying the testing process.

Further work for interfacing the method with XP is, however, needed. Ultimately we need to build smart test tools which interface naturally with the XP process. The development of such tools is currently in progress. The use of such tools will be reported in further papers.

References

1. Holcombe, M. and Ipate, F. 1998. *Correct Systems: Building a Business Process Solution*. Springer Verlag: Berlin.
2. Holcombe, M., Bogdanov, K., Gheorghe, M. 2001. Functional Test Generation for Extreme Programming *Proceedings of XP2001*: 109-113.
3. Ipate, F. and Holcombe M. 1997. An Integration Testing Method That is Proved to Find all Faults. *Intern. J. Computer Math.* **63**: 159-178.
4. Ipate, F. 2004 Complete Deterministic Stream X-machine Testing. *Formal Asp. Comput.*, to appear.