DIVISION OF COMPUTER SCIENCE

Testing should help to Insert New Bugs or How to Modify Programs Predictably?

Technical Report No.144

B. Christianson
R. Barrett

September 1992

Testing Should Help to Insert New Bugs

or

How to Modify Programs Predictably?

Bruce Christianson and Ruth Barrett
School of Information Sciences, Hatfield Campus, University of Hertfordshire, England

Abstract. Software maintenance is often regarded as consisting of two different activities, fixing bugs and modifying the code to adapt to changing requirements. We argue that these activities are really two faces of the same coin, but with modification corresponding to bug insertion rather than removal. This has an implication for the way testing is supported. Test tools should view system behaviour in the context given by the designers' models. Modification to a program will change the behaviour of one or more software components, and the designer must predict which components will be affected. If we think of a bug in a program as a modification from the desired behaviour, then, in a similar way, it should be possible to predict which design components are causing the deviation. We propose a stratagy for testing the design which will help with both forward modifications and error repair.

We begin by exploring the observation that the process of introducing modifications to an existing program as a result of changes to the system requirements is very closely analagous to the process of finding bugs in a program, in the sense that good strategies for the one activity correspond to good strategies for the other.

On a conventional level this assertion appears to be obvious - if we change the specification without changing the program, then the program will fail to meet the specification, and can therefore be regarded as having bugs in it. Finding and removing these bugs will result in a program which has been modified in the required way. Conversely, debugging a program can be regarded as a minature case of incremental development. In this conventional view (which we shall reject), when we are debugging we are attempting to alter the behaviour of the program from that specified by what the program currently does, to that specified by what it should do, ie to the original specification.

But let us look more closely at what we actually do when we undertake each of these activities in turn.

When we are called upon to modify a program we begin from a requirement to alter the behaviour of the program in some specified way. We attempt to isolate the areas of the program that will be required to change, and use some strategy to identify a proposed (minimal) set of changes to the code. We then consider (by exercising some conceptual model of the system) whether the proposed set of changes will have all and only the desired effects on the program behaviour. We iterate this process, alternately refining and widening our proposals, until some stable point is reached, at which point we cut in the changes and evaluate the entire resulting program against the (new) specification.

When we are searching for a bug in a program we begin with an observed

deviation from the required behaviour. We attempt to isolate the areas in which the bug might lie, and to identify a (minimal) set of "errors" or wrong lines of code which constitute the bug. We then consider (by exercising some conceptual model of the system) whether the proposed set of errors would have the effect of explaining exactly the observed departure from the specification. We iterate this process, alternately tightening and enlarging our proposed bug, until some stable point is reached, at which point we correct (cut out) the bug and evaluate the entire resulting program against the (original) specification by regression testing.

Note that this account of debugging is subtly different from the conventional account which we gave earlier and now reject. In the account we advocate here we regard the process of debugging as starting from the original specification, modifying this to say that the program should in fact do what it actually does now instead of what the specification says it ought to do, and then seeking to find a minimal set of changes which will secure exactly this altered behaviour. We then examine the program text, hoping to find that these "changes" are already present in the program text, and to remove them.

To secure an exploitable analogy between the two processes of maintenance and debugging, we must ensure that our thinking proceeds in the same direction in both cases, that is from the familiar to the unfamiliar. Presumably we are more familiar with the current version of the program than with the version which will have been modified in the required way. And (we hope) we are more familiar with a mental model of the program as it should work than we are with a mental model of the buggy version. After all, if our intuitions about the actual current behaviour of the program could be trusted, there would be no bug in the code to begin with.

Consequently, we regard the key step in debugging as being the test of whether the presence of the (candidate) bug explains the deviant behaviour, not whether the absence (removal) of the candidate bug would be consistent with expected behaviour. We re-emphasize - until just before the deviant behaviour was detected the developers believed that the program as it stood was consistent with expected behaviour. There is generally even less evidence to suppose that the patched program will be, unless we have proof that the bug has been correctly identified.

In our experience, most of the dependent errors introduced by would-be bug fixes stem primarily from a failure to consider whether the bug has been correctly identified. Subsequent failures to consider the effects of the proposed fix are themselves a symptom, rather than cause, of the process leading to the introduction of dependent errors.

Finding and removing a bug thus corresponds more nearly to the effect of identifying and undoing a modification, rather than proposing and making one. Where a bug has not yet been positively identified, the search for it closely resembles the attempt to modify the ideal program which we have in our minds (and which we have modelled in the specification) in such a way as to produce the deviant behaviour symptomatic of the bug.

Conversely, this means that we can regard the process of modification to meet new requirements as an attempt to insert a bug, rather than to remove it. While we are seeking to identify changes, a good strategy is to treat the required new behaviour as if it were a deviant (unwanted) behaviour actually exhibited by the existing program, and ask: what could constitute the bug

producing this effect, and where in the code would it be located?

In the case of a bug, we can play about with the program to try and view the bug from different angles, but at the end of the day we must find and eliminate the bug which is actually there, not some other possible explanation for the observed behaviour. In the case of a desired modification, we are willing to accept (almost) any change which produces the required change in behaviour, but we cannot use the existing program directly in such a way as to learn more about the conseqences of our changed requirements. However, in both cases we are attempting to refine our specification of the deviation in behaviour, to the point where the textual changes are well defined and obvious, either by using the program as an animated specification tool, or by applying (possibly automated) tools to (other representations of) our mental model.

The conclusion from this is that code structures and tools which make it easy to find and fix bugs should also make it easy to identify modifications in response to changing requirements.

This also has implications for the kind of testing which we should do, and the way in which we should go about doing it. In particular, black box testing (where the test plan is generated from a specification which is structurally independent of the implementation) is of limited usefulness. Black box testing doesn't help us find the bug, and (even with a regression suite) it doesn't give us any assurance that we've fixed the bug properly.

Exhaustive unit testing is also of limited cost-effectiveness. In practice, and often even in principle, it is not possible to isolate the components of a system to the necessary degree. The interfaces usually contain a great deal of covert history and in consequence the problem of making the test harness behave like the rest of the system in full generality is, especially in the presence of concurrency, a harder problem than getting the component coded correctly from the specificiation in the first place.

We need tools which assist us to integrate our inferences from the specification with our other conceptual and structural models of the product, possibly via an extension of an existing CASE tool. This would allow a form of adaptive testing via assertion violation, using the rest of the system as a harness for each component in parallel, and using the specification (indirectly) to generate the external test sequences so as to drive the interfaces into the appropriate state.

The challenge is to develop this kind of white box testing to the point where exposure to a deviant behaviour pattern is sufficient to allow the test plan generator to propose areas of code which might be infected by the bug, and (in conjunction with a human designer) to propose test sequences which would refine these areas.

A tool to support a strategy of this type would be sufficiently intimate, not only with the code but also with the designers' conceptual models of the product, to be of valuable assistance in adapting the model and the product to changing requirements. In particular, it would be able to generate the right scenarios to force the designer to think about the choices to make in refining the specification of the changed behaviour to the point where it can be coded.

When the designers' conceptual model can be described using a formal language such as Z [Sufrin 1985], VDM [Jones 1986], CCS [Milner 1980] or UNITY [ChMis 1988], and the required properties of the system deduced as

theorems, this needs to be done in a way which allows the components of the conceptual model to be localised to specific pieces of text in the code. Whether or not the deduced properties describe the observable (deviant) behaviour is then something that can be decided by inspection.

The testing strategy when we are designing a modification is to specify the present behaviour, and then to change this specification so that it describes the required behaviour. We proceed from this to an identification of those parts of the present system which could be modified in order to secure exactly this changed behaviour.

The testing strategy when we are debugging is to specify the required behaviour, and then to change this specification so that it describes the actual behaviour. We isolate and correct those parts of system which cause the unwanted behaviour, thus removing the cause of the error rather than just one of the symptoms.

We do not ask "what if the error is here?" but rather "what if a change is made here?"

[ChMis 1988] Chandy, K.M. & Misra, J.M. (1988). *Parallel Program Design: a foundation.* Reading, MA : Addison Wesley.

[Jones 1986] Jones,C.B. (1986). *Systematic software development using VDM.* London: Prentice-Hall.

[Milner 1980] Milner,R.(1980). *A calculus of communicating systems.* Springer Verlag, Lecture notes in Computer Science, **92.** Berlin:Springer Verlag.

[Sufrin 1985] Sufrin,B.A.,Sorensen,I.H,Morgan,C.C., & Hayes, I.J. (1985). *Notes for a Z handbook.* Oxford University Programming Research Group internal report.