International Journal of Advances in Engineering Research http://www.ijaer.com

(IJAER) 2012, Vol. No. 4, Issue No. VI, December **ISSN: 2231-5152**

STUDY ON FIREWALL APPROACH FOR THE REGRESSION TESTING OF OBJECT-ORIENTED SOFTWARE

TAWDE SANTOSH SAHEBRAO

DEPT. OF COMPUTER SCIENCE
CMJ UNIVERSITY, SHILLONG, MEGHALAYA

ABSTRACT

Adherence to a defined process or standards is necessary to achieve satisfactory software quality. However, in order to judge whether practices are effective at achieving the required integrity of a software product, a measurement-based approach to the correctness of the software development is required. A defined and measurable process is a requirement for producing safe software productively. In this study the contribution of quality assurance to the software development process, and in particular the contribution that software inspections make to produce satisfactory software products, is addressed.

To be more effective detecting software defects, not only should defect detection techniques be studied and compared, but the entire software defect detection process should be studied to give us a better idea of how it can be conducted, controlled, evaluated and improved.

INTRODUCTION TO SOFTWARE FAULT DETECTION

Software fault detection is an important part of software development. The quality, the schedule, and the cost of a software product based heavily on the software fault detection process. In the development of software systems, 40% or more of the project time is spent on fault detection activities, such as, inspection, testing, and maintenance. In this thesis, maintenance means the fault detection activities after software releases, which include trouble shooting and debugging.

Software fault detection research has proposed new inspection and testing techniques, and has studied and compared different inspection and testing techniques. However, most of the research has focused on a single inspection or testing technique. At most, different inspection or testing techniques were compared to determine which one detected more faults. To be more efficient in this area, not only the study of a fault detection technique itself is necessary, but also more emphasis should be put on the fault detection process in which these techniques are applied. How can we get more from the fault detection process by a meaningful selection and combination of the available fault detection techniques? How can we assess and improve the software fault detection process? To a large extent, these questions are still open. Since there is no general advice on how to conduct the software fault detection process, many medium and small software organizations apply some techniques based on personal preference and never use the other useful techniques at all. For example, testing may be used to the exclusion of inspection.

As with much quality assurance processes software inspections impose a cost, which has to be borne, with apparent, limited, added value to the product, and it is accepted that quality assurance is an overhead on the development costs of a product. However, the cost of having a poor quality product is even greater, both in terms of actual product costs and reputation. The added value from inspections comes from increased confidence in the product from the software producers and from their customers, and further by potentially reducing programme and life cycle costs downstream from the development stage at which they are applied. By involving the correct people within an inspection team problems can be resolved early therefore eliminating unproductive rework reducing both the technical and business risks, and ultimately preventing rejection of the product by the customer.

The initial aim of this work was to review software quality assurance within the development process and more specifically the area of software inspections, with a view to establishing areas of strengths and weakness and to identify areas of work which would benefit from further research. A review of current literature shows that software inspections have been successful in identifying errors within software products close to the point of their introduction, and therefore improving software productivity. However, software inspections are still very variable in application and effectiveness, depending greatly on the ability and experience of the individual inspector. These attributes, including the human factor, which influences the effectiveness of a software inspection, have not been investigated before in detail by previous researchers.

It could then be argued that by reducing the number of faults in a product it has a positive impact on safety. The effective identification of faults close to their introduction in the software lifecycle will reduce the amount of regression testing required when faults are found down-stream and hence increase productivity.

In proper verification and all forms of software inspections, to find functional faults, the behavioural stipulation exhibited by a software artifact must be extracted from that artifact and compared to its intended stipulation. In this thesis we present techniques for deriving semantic assertions from a software artifact. These semantics represent the abstracted behavioural stipulation required to support proper verification and software inspection activities on that artifact. The repeatable techniques presented form a basis for reasoning about functional correctness and for assisting in the detection of functional faults.

The deduced semantics serve different purposes depending on the formality of the stipulation given. Although the semantic derivation techniques are manually applied to examples throughout this thesis, we place an emphasis on the definition of algorithms for extracting semantics that are amenable to automation.

B-METHOD

In one sentence, B can be regarded as a proper development method to specify and execute software that provably meets its requirements. A B development can logically be divided into three distinct parts, known as Abstract Stipulation, Intermediate Refinement and the Implementation.

Moving Z to B

Apart from B, the Z stipulation language plays currently one of the most important roles in software stipulation. The main stipulation structure used in Z is that of the Z schema. Schemas in Z are created to represent operations as well as abstract data. The following example, taken from [17], is given to illustrate the notation:

```
Class
-----------------------------------
enrolled, tested : P Student
#enrolled ≤ size
tested ⊆ enrolled
-----------------------------------
```

The first part of the schema is the declarative part, where all variables of the schema are introduced. The second part contains the stipulation itself. Here, conditions on the involved variables might be imposed. In this particular example, the abstract data to model a class of students is specified.

It contains two variables, enrolled and tested, which are both subsets of the Student set. One condition on the specified data is that the number of elements in the set enrolled does not exceed size. A further condition is that only Students from the set enrolled can appear in the set tested.

The Z stipulation language employs the same notation of a schema to specify operations. Since only predicates are allowed to appear in Z schemas, operations have to be defined as conditions on variables representing the before- and the after-state of the modified data. Commonly, dashed variables are introduced for the after-state and the respective un-dashed variables represent the before-state. To aid the notation, schema decorations have been added as part of the syntax. As for example, ΔClass described the schema that contains all variables from the Class schema as well as their dashed counterparts, along with the schema conditions for both of them.

CONCLUSION

The techniques presented in this thesis manifest in more repeatable, effective and practical techniques of inspection and proper verification.

We conclude this thesis by re-iterating and summarising the major contributions of the research, providing a critical evaluation of the techniques and techniques presented.

The techniques developed in support of this thesis are largely based on the Strongest Post-condition predicate transformer (sp). We identify problems with other proper methods for deriving semantics to support reasoning and discuss the theoretical problems with automating existing strongest post-condition calculations for assignment. We provide a definition for sp to calculate the strongest post-condition for assignment, which can be mechanised and which does not rely on the calculation of inverse functions or on the introduction of new variables.

It is anticipated that the implementation of some of the algorithms in this thesis will be problematic as a result of the fact that a number of the simplifications and solutions required to be performed by the tool are complex.

The B method as a mathematical concept has been presented. The presentation has aimed to cover the three development phases Abstract Stipulation, Refinement and Implementation.

For the abstract stipulation and refinement, validation obligations have been presented and their role has been discussed. The idea of refinement has been illustrated. Therefore, an example has been given to show how the state of an abstract machine can be re-expressed in the refinement machine. Structuring mechanisms in a B development were presented, and their relationship to the design methodologies Bottom-up and Top-down was explained.

The tool support for applying the B method has been investigated, and in particular the B-Toolkit has been presented with its most important features. Each of the environments of the B-Toolkit has been discussed separately.

Further, a development example of the game "Tic-Tac-Toe" was performed. The development has been carried out from the informal stipulation to the abstract B stipulation and finally the implementation of the component. The derivation of the machine's constant from the underlying model has been discussed. In connection with the implementation the use, of library machines was illustrated on the example of an array.

So finally we reach to result that Current techniques of verification and fault detection are, generally, not repeatable, and derivation of semantic information, including constants, from program code can support and improve the repeatability of verification and inspection tasks.

Although it is very hard to say that inspection techniques give hundred percent sureties, but we tends to achieve the quality product and minimize faults.

REFERENCES

- A Discipline of Programming, E. W. Dijkstra, Prentice Hall, 1976.
- Stipulations, programs and total correctness, E. Hehner, Technical report, University of Toronto, Canada, 1999.
- Communicating Sequential Processes, C. A. R. Hoare, Prentice Hall, 1985.
- Unifying Theories of Programming, C. A. R. Hoare and He Jifeng, Prentice Hall, 1998.
- Higher-Order Mathematics in B, J.R Abrial, D. Cansell, and G. Laffitte, In Springer, editor, Lecture Notes in Computer Science 2272, 2002.
- Reconciling axiomatic and model-based stipulations using the b method, K. Robinson., Technical report, The University of New South Wales, Sydney (Australia).
- Dynamically discovering likely program constants to support program evolution, M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, IEEE Transactions on Software Engineering, 27(2):99–123, February 2001.
- Edinburgh lcf: A mechanized logic of computation, M. Gordon, R. Milner, and C. Wadsworth, Lecture Notes in Computer Science, 78, 1979
- Logical analysis of programs, S. Katz and Z. Manna, Communications of the ACM, 19(4):188–206, 1976.
- Formal Methods for a verification based software inspection, D. Powell, PhD. Thesis Griffith University, 2002.