

A Test Model for Software Development

József Sziray

Department of Informatics, Széchenyi University
Győr, Hungary
E-mail: sziray@sze.hu

I. INTRODUCTION

The fulfillment of the quality and reliability requirements of complex software systems necessitate that the software undergo some specific **verification** and **validation** procedures [1]-[3]. An integral part of both activities is intensive testing. The importance of testing manifests itself especially in the case of **safety-critical computer systems** [3], where the primary criterion is safe operation.

This paper presents a test model which involves a mapping scheme for describing the one-to-one correspondence between the input and output domains of a given software system. The test model incorporates both the verification and the validation schemes. The significance of the model is that it alleviates the clear differentiation between verification and validation tests, which is important and useful in the process of test design and evaluation. The second part of the paper examines the case when the software is designed by using **formal specification** [3]-[5]. Here the consequences and problems of **formal methods**, and their impacts on verification and validation are discussed

II. THE TEST MODEL

The fundamental feature of software faults is that they are present all the time during operation. The question is how their effect manifests itself in various situations. Two basic classes of software faults can be defined:

1) Specification faults: Faults which occur at the beginning of the development cycle, and manifest themselves in the malfunctioning of the software, by not meeting the real user requirements. Malfunction may be the consequence of both erroneous and incomplete specification. This category can also be referred to as **external** or **user faults**.

2) Programming faults: It involves the wide group of faults that originate from the mistakes made by the programmers in the course of design and coding of the previously specified software. Other synonyms for this category: **internal** or **development faults**.

In the development of software it is a fundamental requirement to keep the whole process in a strictly consistent progression. Here the various development phases have their own representations of realization. The

activity of proving the equivalence between the consecutive phases is called **verification**.

On the other hand, the perfect realization of the verification process does not guarantee the perfect usability of the final product. All that can be guaranteed is the fulfillment of equivalence with the starting specification. In case if there had been mistakes or imperfections in the specification, the final product will not meet the user requirements. It is necessary therefore to examine the product from the viewpoint of its original destination. The extra observation process which results in the desired proof is called **validation**.

A software system can be considered as an entity which **maps an input set to an output set**. For simplicity, combinations and sequences of input data will be considered as a single input. The response to these inputs is made by producing an output or a set of outputs.

Let the set of all possible inputs to the software under consideration be denoted by **INPD (input domain)**, and the set of all possible output responses by **OUTD (output domain)**. Now the mapping of INPD into OUTD by the software system will be defined in the following form:

$$\text{SWM (INPD)} = \text{OUTD.} \quad (1)$$

Relation (1) means that the properties of the system, i. e., the mapping SWM, determine the correspondence between the elements of INPD and the elements of OUTD. This relation will be considered to be valid at any particular phase of the software development cycle.

Furthermore, we denote the set of inputs which cause malfunction by **INER**, while the set of erroneous outputs by **OUTER**. For these sets the following mapping relation holds true:

$$\text{SWM (INER)} = \text{OUTER.} \quad (2)$$

If we involve tests for fault detection, it will result in the modification of sets INER and OUTER. Suppose a test set **INERT** is capable of detecting a subset of yet undetected faults. As a consequence, this test must result in fault removal. It means that the corrected software will produce a modified output domain with a subset OUTER which does not represent the detected and so deleted faults any longer. After this, leaving the former notations unaltered, the new mapping relation for the software is

$$\mathbf{SWM (INER - INERT) = OUTER} \quad (3)$$

where the minus sign stands for subtraction between sets.

At this point the following notations are introduced:

The set of inputs which manifest themselves in development faults: **DEFI**. The output mapping of this set is **DEFO**. The set of inputs which manifest themselves in user faults: **USFI**. The output mapping of this set is **USFO**. The set of test inputs which have been devised to detect development faults, i. e., the set of verification tests: **VERT**. The set of test inputs which have been devised to detect user faults, i. e., the set of validation tests: **VALT**.

Now we can give a general model to describe the mapping scheme of verification and validation. The mapping relations are as follows:

$$\mathbf{SWM (DEFI - (VERT \cup VALT)) = DEFO.} \quad (4)$$

$$\mathbf{SWM (USFI - (VERT \cup VALT)) = USFO.} \quad (5)$$

Here relation (4) expresses the verification mapping, while relation (5) expresses the validation mapping. It can be seen that (4) represents the undetected development faults, whereas (5) represents the undetected user faults.

III. THE USE OF FORMAL METHODS

The term **formal methods** describes the use of mathematical techniques in the specification, design, and analysis of computer hardware and software [3]-[5]. A specification must be unambiguous, complete, consistent and correct. Formal methods are based on the use of **formal languages** which have very precise and strict rules. This feature enables the specifications to be defined in a manner that can be interpreted unambiguously. It also makes possible the automatic checking of the specifications in order to find omissions and inconsistencies, i. e., to prove the completeness and consistency.

One of the greatest advantages of describing a system in a formal manner is that automated tests may then be performed on this description. This allows software tools to check for certain classes of error, but also allows different descriptions to be compared to decide if they are equivalent. As it can be seen, this process is nothing else than the verification itself.

In an ideal case, the transformation procedures in the above outlined process are completely automated, having no human interaction, and are performed without any faults at any phase. If so, the external test input sequences for verification can be completely omitted. In our test model it means that **DEFI** = \emptyset , **VERT** = \emptyset , where the symbol \emptyset represents the empty set. From this it follows that **DEFO** = \emptyset .

On the other hand, the initial formal specification of the software is always carried out by a manual way, and therefore design errors may never be excluded here, even if there is an automated consistency checking available. The reason for this is that a consistent design in itself does not guarantee the perfect fulfillment of the user requirements. Also, there is no guarantee for meeting the safety requirements either. As a consequence, the external validation testing is always necessary to apply. Thus, by taking into consideration the above ideal case, the following relation will hold true:

$$\mathbf{SWM (USFI - VALT) = USFO.} \quad (6)$$

However, in most cases, fully automatic performance is yet not applicable, that is, close cooperation and interaction of the skilled designers are further required. At present, the main advantage of formal methods is that they enable to perform the design and verification processes with greater reliability.

The total process of proving safety for safety-critical systems, i. e., the total validation process cannot be automated in principle. This is because in this case the testing has always to have elements that fall outside the system specification, i. e., these complementary elements are independent of the specification. The basic reason is that safety problems originated from incomplete specification can be revealed this way only. Of course, this statement applies not only to classical methods, but also to formal methods.

Finally, it should be added that at present there are no exact formal specification methods available which could take into account the principle of safety itself.

REFERENCES

- [1] Ian Sommerville: Software Engineering, Sixth Edition, Addison-Wesley Publishing Company, Inc., USA, 2001.
- [2] Roger S. Pressman: Software Engineering, Fifth Edition, McGraw-Hill Book Company, USA, 2001.
- [3] Neil Storey: Safety-Critical Computer Systems, Addison-Wesley-Longman, Inc., New York, 1996.
- [4] Constance Heitmeyer, Dino Mandrioli: Formal Methods for Real-Time Computing, John Wiley & Sons, Inc., USA, 1996.
- [5] József Sziray: Verification and Validation of Software Systems, Széchenyi College, Győr, 2001.