On the State of the Art in Requirements-based Validation and Test of Software

Johannes Ryser, Stefan Berner, Martin Glinz
Department of Computer Science of the University of Zurich
Winterthurerstrasse 190, CH-8057 Zurich, Switzerland
{ryser, berner, glinz}@ifi.unizh.ch

Abstract

Validation of requirements and verification of software are crucial activities in the software development process since they essentially determine the quality of the software product and account for a big part of the overall development cost and use of resources. Only these activities can ensure that user requirements - whether they be explicit or only implicit - are met. This report overviews current methods and approaches towards validation of requirements and requirements-based testing.

Keywords

distributed systems, embedded systems, requirements specification, specification-based testing, system test, test specification, validation

1.0 Introduction

During the last decade, size and complexity of software embedded in industrial products has exploded. Companies are investing enormous amounts in information technology and currently there are no signs of a decrease. As software is increasingly being integrated into products, it is becoming a key competitive advantage and the company that is able to deliver software of high and predictable quality with a minimum of development cost in a timely fashion has a big advantage over any competitor.

The timely delivery of good quality complex software depends on several critical factors: Of special interest in this report are requirements validation (i.e. ensuring that the requirements adequately, consistently and completely state the needs of the users) and system test (i.e. establishing that the implemented system conforms to its requirements).

In industry these two activities are often not performed in a very systematic manner; they depend heavily on intuition and experience of the tester. As a direct result software quality and required testing effort tend to be unpredictable. Different approaches have been developed to overcome this problem: Methods to derive test cases systematically, to find “good” test cases - a good test case being one that has a high probability of discovering a yet undetected fault or anomaly - or to generate test cases automatically have been contributed by the research community. However, many open issues remain, in particular systematic validation and testing of distributed, embedded systems is a widely unexplored field.

2.0 State of the Art in the Research Field

2.1 Definition of Validation and Verification

Validation and verification of software are defined in this report according to [Boeh81] and the IEEE Standard Glossary of Software Engineering Terminology [IEEE90]:

> *Verification and validation (V&V). The process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements.*

Approaches to validate requirements are considered in Section 2.2, system test as part of the verification process, the two most fundamental test strategies and some established test methods are summarized in Section 2.3, Section 2.4 categorizes test methods according to the formality of the specification and in Section 2.5 requirements-based test approaches are presented. The final chapter points out the limits and drawbacks of the presented methods.

2.2 Requirements Validation

Requirements validation is the process of establishing the adequacy, completeness and consistency of a requirements specification. As requirements validation takes place in the early phases of the software engineering process and faults in requirements discovered at a later time are very expensive to fix [Boeh81], requirements validation is the most crucial of all verification and validation activities.

There are five standard approaches to evaluate adequacy, completeness and consistency of a requirements specification regarding the users needs and expectations:

1. conducting reviews of the requirements specification
2. building and evaluating a prototype
3. running a simulation
4. using test and analysis functions integrated in specification tools / automated completeness and consistency analysis

In addition to explicit validation activities, formalization of requirements specification helps to find omissions and contradictions in a specification, that is formalization helps to check a specification for completeness and consistency. Cause and effect graphing for example aids in identifying requirements that are incomplete and ambiguous [IEEE90]. State transition diagramming might discover missing states and subsequently even missing actions or behavior.

Reviews as a means of quality management involve a group of people examining a document - e.g. the requirements specification - with the aim of finding faults and discovering existing anomalies. All of the groups findings, conclusions and decisions are collected and passed to the team who is responsible for their correction. There are different review techniques: Inspections [Faga76], Walkthroughs [Your77, Wein71] and technical reviews

[Free82].While most authors agree that reviewing should be a formal process, there is no agreement to what method or what procedure should be used to inspect a requirements specification.

Prototypes implement an executable model of the future system or of parts of it. A prototype of a software system is an incomplete implementation that mimics the behavior we think the user needs [Stak89]. Users work with the prototype as they would (or will) with the system to be built. In doing so requirements are validated as the system is shown to meet users expectations. Prototypes are mainly useful to show the adequacy of a model regarding user interface, interaction and functional visible behavior of a system. They are less apt to verify completeness or consistency of requirements. Prototypes should be easy to build, modify and extend. Nevertheless, building a prototype is considerably more expensive than reviewing.
Building a prototype instead of writing a requirements specification is dangerous because a prototype does not cover **all** requirements. Moreover it is difficult to distinguish the essential features of a system that is being built on the basis of a prototype from the accidental ones. Executable requirements specifications (i.e. specifications that can automatically be converted - translated - into a prototype) try to overcome these problems.

Simulations help in validating the adequacy of the behavior of a given system under specific circumstances. Different scenarios (description of a set of interaction sequences) can be simulated.

Specification tools not only help to capture requirements, to model the system according to a chosen specification method and to build a repository / data dictionary, but they also help in testing completeness and consistency of requirements specifications. Exemplary are tests of the form: ‘Are all data objects defined in the dictionary? Are definitions consistent?’ or tests specific to a given method, e.g. ‘Has every process in a DFD on every level of the hierarchy the same number of data in- and outflows?’
Requirements that have been elicited can be further analyzed by checking them for logical self-consistency, for testability and feasibility. Functional completeness can be checked by analysis of existing processes, process descriptions and existing systems (if there are any) carrying out the same tasks as the system to be built.

A problem arising in any of the approaches mentioned above is how to systematically cover both functional and non-functional requirements when validating. How is a prototype to be validated systematically? What actions have to be taken to ensure a review completely and adequately covers all the requirements called for by users and procurers? How can specification tools validate non-functional requirements, e.g. UI requirements?

2.3 System Test

Software testing is aimed at finding faults by executing the system or parts of it in a well defined environment with specific equally well defined input data (test cases) [Myer79]. In the system test the final system is examined to assure compliance with the requirements specification prior to its release and deployment. Testing in general and system testing in particular are tedious, time-consuming activities which prompt fatigue and inattentive work. Moreover as tests are conducted at the end of the development process, time schedules are becoming tighter and tighter. Detecting faults causes additional delays. As a consequence, both test preparation and execution are frequently performed only superficially.

In order to conduct successful tests a test plan has to be created and the necessary test cases have to be developed. This preparatory work is costly and time-consuming, but it is a prerequisite for any useful test. Despite the fact that test preparation accounts for a considerable percentage of overall test cost, only little research has been directed to the problem of systematic construction of system tests. In particular we are not aware of specific methods for testing distributed embedded systems.

As it is impossible to construct and conduct exhaustive tests for any system of reasonable size due to combinatorial explosion of input data and parameter combinations, software is partially tested only. Choosing the right test cases becomes a crucial issue. The chosen set of test cases should ideally uncover a maximum of faults with a minimum of cost and effort. There are different strategies for test case selection: Basic is the distinction between structure and functional testing.

In *structure-oriented* approaches, so called White-Box or Glass-Box-Tests, test cases are chosen based on the control flow or data flow of the program to be tested. Program code as well as the specification are used in developing test cases. The significance of test cases can be measured in terms of their coverage: Coverage measures the percentage of internal structures (statements, branches, data structures) that are executed or accessed when executing all test cases.

White-Box-Tests are not apt to test a whole system. It is hardly ever - if at all - possible to successfully construct a set of test cases with sufficient covering; and if so, it can only be done with prohibitive expenditures. Moreover a White-Box-Test by definition can not uncover missing features. Therefore the structure-oriented approaches are not appropriate for system test.

In *function-oriented* approaches - so called Black-Box-Tests - the tester does not know the internal structure of the program. Test purpose and test cases are chosen according to the requirements stated in the specification. Test data for test cases is selected following a particular method. Test methods are summarized in the following sections.

The existing methods for test case selection are mostly informal and fairly sketchy. The quality of the test (i.e. the number of faults it detects) heavily depends on the experience and expertise of the persons who design the tests. Therefore any research aiming at a systematic derivation or even an automatic generation of test cases is of great practical importance. As it has been proved that a complete and general automation of the entire testing process is impossible [Goed31] the most promising approach to an improved test case selection and derivation from the requirements is a systematic manual or partially automated procedure.

The focus of this paper is on methods to generate test cases for system test. Testing strategies for unit and integration test are not considered.

2.4 Methods Classification: An Overview

It has been mentioned that testing in parts is a tedious and wearisome task. Therefore it would be desirable to support persons who design test plans and devise test cases with a method or even a semiautomatic procedure to help them tackle these tasks. There are a number of different approaches to derive test cases from specifications - a coarse overview is given below.

2.4.1 Solution strategies

1. *Heuristic methods*: Testing is intrinsically a heuristic approach to find faults in a program. All the classical test and test case selection methods as presented in many books (e.g. [Myer79], [Beiz90]) fall in this category. Equivalence (class) partitioning, boundary value analysis and tests, domain testing, error guessing, random input tests (all mentioned are Black-Box-Tests) are part of this category as well as Coverage Tests / Control-flow-testing (path-, condition-, branch coverage,...), data-flow-testing, syntax-testing and state-transition-testing (White-Box-Testing).
   Most of these methods can be supported by tools, but automatic test generation is not feasible or yields insufficient coverage.

2. *Methods to improve heuristic test case selection*: These approaches extend and integrate heuristic methods. They systemize test case selection and refinement by describing general exemplary procedures, giving ready made partitioning or step-by-step guidance on how to select effective test cases (see for example [Beiz95]). Test methods are classified according to their aptness to specific tests (e.g. unit, integration or system test) and the error assumptions they base on - what faults will and what faults can not be uncovered by a specific test method - are indicated. The limits and drawbacks of different heuristic methods are pointed out and by combining different methods the shortcomings of the individual methods are diminished or even overcome. Tools support for testing staff is a main concern in these approaches.
   Yet much of test case design and test case selection has to be done manually. Because of the integration of different methods coverage by automatically generated test suites is increased, but still sufficient coverage is hardly achieved.
   These methods will be called ´*procedure-based approache*s´ in this paper.

3. *Automatic test case selection / generation*: There are two approaches to automatically generate test cases, namely test case generation based on code analysis and test case generation based on formal specifications.
   In the former approach, flow of control and value boundaries in control and data structures of a program are analyzed to extract information for test case generation. There exist supporting tools to this approach, many of which are language dependent and can not guarantee sufficient coverage by automatically created test cases only. [Howd76] proves that the automatic generation of a finite test set that is sufficient to test a routine by path coverage is not a computable problem. Obviously these approaches rely on code. They can not easily be extended to informal or semiformal specifications because these kind of specifications are in general inconsistent, ambiguous, inaccurate, indistinct and vague. Furthermore these methods are not suitable for system test, for whole systems are far to complex to be analyzed in practice with justifiable expense and test suites generated automatically are not of reasonable size anymore. As we are looking at requirements-based testing in this paper we will not explore this approach any deeper.

   The second approach requires formal specifications: By writing a formal specification[1]

and by formalizing test purpose(s) test cases can be generated automatically or derived in a formal way. Long known examples are Cause-Effect-Graphs, decision tables and the Stimulus-Response-Approach as shortly described in Section 2.5.3. Most research in improving test methods and automating test case generation is centered around the field of different formal approaches. Some representative methods are listed in Section 2.5.3.
Based on formal approaches automatic test suite generation is feasible. Yet size of test suites is still a problem, for test suites tend to be very large. Sufficient coverage is achievable only under restricting assumptions.

4. *Statistical testing*: Based on the idea of random testing - randomly choosing any input data to generate test cases that are equally distributed across the whole input data space, in statistical testing test data is chosen according to some given probability distribution of the input parameters in the input data space [Dura84, Loo88, Thev95]. Thus: while random testing is done by a uniform sampling of the input domain, statistical testing relies on a operational profile - the expected runtime distribution of inputs to the program. From random testing / statistical testing quantitative estimates of a program's operational reliability can be inferred [Thay78]. It is difficult though to determine what distribution captures the runtime distribution of inputs in the given application best. A lot of experience and statistics on the use of this particular or at least a related application are needed. Yet Thayer et al. recommend that "final testing should be done with input cases chosen at random from [a programs] operational profile" [Thay78].

A further method might be mentioned. It is not a method to derive test cases though, for it is rather aiming at avoiding testing at all, making testing unnecessary and superfluous:

5. *Formal, mathematical proof* (proof of correctness): In this approach software is guarantied - at least theoretically - to be error free by mathematical proof of correctness. Formal proofs really aren't a test method and they don't help in test case selection. As valuable as they can be in establishing a system´s correctness we have to be careful as they entice to believe that testing of formally proven software is not necessary any more. However, formal proofs are just as prone to faults as any other human activity.
Formal proofs are only applied in small projects or on the safety critical parts of a system because of the required effort and cost.

2.4.2 Influence of the formality of specifications on test case selection

1. *Informal and formatted specifications* can be further subdivided in
- unstructured natural language specifications: These are specifications in form of an essay. They describe the requirements in narrative form, at most using enumerations, but no further structuring or highlighting (e.g. numbering system, tables, ...).
- structured natural language specifications: use formatting as a way of structuring the specification. Sections, paragraphs, indents, font type, face, style and size, numbering and enumerations, tables, sketches and keywords like *if... then... else... endif* or *for... do... endfor* are applied and utilized to highlight, subdivide and structure the specification and make it more understandable. This kind of specification can even take on the form of pseudo code: describing functions in an algorithmic way similar to program

1. The term "Formal specification" as used in this paper includes all formal (based on set theory and predicate calculus, pre- /postcondition approach), all algebraic and model-based (Larch, Z, VDM) specifications. See the classification scheme on 1page 6.

code.
The main advantage of this descriptive type of specification is to be less abstract and often more understandable to users than the later mentioned. They are simple to write and readily understood. Because of their heavy reliance on natural language, informal specifications suffer of incompleteness, omissions and inconsistency. They are imprecise, ambiguous and requirements can be overspecified. That's why this kind of specifications is not apt to serve as a basis to generate test cases automatically. Heuristic and extended heuristic methods have to be used to create test cases. Test case selection has to be done by the tester and thereby depends heavily on the expertise, intuition and experience of the tester. Statistical testing may be applied if the runtime distribution of input values is known. The completeness and consistency of the specification can hardly be checked.
This type of specification can be successfully applied to small projects that are easily comprehensible and of manageable size, but it is not suited for large projects.

2. *Semiformal / partially formal specifications* are increasingly used today. Semiformal specifications are a constructive approach to build a model of the problem domain, defining some formal rules on how to construct the model, yet relying on non-formal natural language description in other parts. To contrast them with formal methods they are often also called structured methods. They can be further categorized in
- specifications based on procedural abstraction (e.g. structured analysis)
- specifications based on data abstraction (prominent representative being the ERM Entity Relationship Model)
- specifications based on behavioral abstraction (e.g. finite state machine, state transition diagramming, message sequence charts)
- object oriented methods (e.g. Booch, Rumbaugh, Shlaer-Mellor or Wirfs-Brock).
Semiformal methods are quite diverse as to their formality: On the one hand there is methods like structured analysis for example where big parts of a specification are informal using data flow diagrams and a data dictionary, relying heavily on natural language and constructs, the semantics of which is not precisely defined. On the other hand there may be parts in a semiformal specification that are totally formal, e.g. finite state machines[2]. Test case selection and generation therefore can only be automated in parts and still depends heavily on the tester. Most arguments applying to informal specifications can be carried over to semiformal specifications. The big advantage semiformal specifications have compared to informal specifications is that they are accessible to checks for completeness and consistency. Moreover test cases for formal parts of the specification can be generated automatically. Semiformal specifications are therefore best tested by a procedure-based approach as these integrate heuristic methods and often offer tools support. Beside procedure-based approaches statistical testing might be applied ensuring further coverage and allowing for an estimate of the reliability of an application.

3. *Formal specifications* are based on formal calculus founded on mathematics. They can be constructive or descriptive. An additional subdivision partitions formal methods into
- logic based formal specifications, algebraic specification: Specifications are based on first or higher order logic and predicate calculus or on function theory respectively.

2. Finite state machines could even readily be transformed into an executable program. They are formal, but they capture only the behavioral aspects of a system. That's why they need to be combined with other methods to describe all aspects of a system and consequently they are listed as a semiformal approach.

Specifications are descriptive. Logic (or functional) programming languages may be used to write the specification.
- model- or language based formal specifications: In this category fall all specifications written in a specific special purpose specification and modeling language. Example are Z [ZOCL92, Spiv92a, Spiv92b], VDM [BSI89, Dawe91, Jone90], Larch [Gutt93] or CSP [Yama92].
- functional specifications
- specifications based on behavioral models (e.g. Statecharts, Petri nets, I/O automata)
- executable specifications like Gist [Cohe83, Balz85] or Paisley [Zave84, Berl87] incorporate a mixture of logic, functional and procedural programming to enable the analyst to write a specification that can be executed and validated by users directly. Executable specifications are very helpful in validating user requirements, but are not contributing any new insight or help to testing. An executable specification represents just a kind of logic or function-based formal specification and thus methods applicable to those approaches are also applicable to executable specifications. However, it has to be emphasized that validation is a crucial activity in the software development process and that any methods improving validation are precious as an error discovered during this phase is saving a lot of time and money in later phases (see prototypes Section 2.2, Requirements Validation).

To formal specifications formal test case derivation may be applied and test cases can be generated automatically. As will be shown in the following paragraphs there still are some problems to the approach of automatic test case generation, the two major problems being the size of test suites that are generated automatically - they tend to be huge, far to extensive to be of practical use - and the coverage that can be achieved as it is hardly ever sufficient. Yet automatic test case generation is valuable as it simplifies the job of a tester by taking off a lot of the laborious, trouble- and wearisome work of creating test cases and as it can be improved and augmented by testers actions and experiences.

In the following paragraphs methods are listed and shortly described according to the category they belong to. Heuristic methods and the procedure-based approaches are only mentioned as they are state of practice. Formal requirements-based testing is described in more depth.

2.5 Approaches to Automatic Test Case Generation in Requirements-Based Testing

As opposed to testing strategies which draw test cases from design and implementation of a program, in requirements based testing test cases are derived from requirements and specifications respectively.

In this section the classification of Section 2.4 is resumed. Formal test case derivation and approaches to automatic test case generation are investigated in more depth. Different approaches are introduced and shortly described.

2.5.1 Heuristic methods

Any defined non formal testing procedure is incorporating some kind of input partitioning. Heuristic methods are ubiquitous for they represent an intuitive way of testing. As they are in use in most companies today it only remains to be said that they are just as good as the

process they are embedded in and only if they are applied conscientious, persistent and regular to fulfill a test purpose. A method suitable to test purpose and position in the development process has to be chosen (e.g. White-Box tests for unit testing, Black-Box tests for system testing).
Heuristic methods are not suitable for automatic test case generation based on requirements though. They are not further investigated here.

2.5.2 Procedure-based Approaches to Improve Test Case Selection

Procedure-based approaches[3] try to establish a well defined procedure for testing and test case selection. Heuristic methods are formalized[4] and expanded, their suitability for specific test phases (as unit, integration and system test) is evaluated and the integration of different testing methods in a test strategy is strived for. Thereby the test process is getting controllable. Metrics - be it linguistic or structural metrics - are applied to quantify complexity, size and cost. Art and craft is to be replaced by science, science begins with quantification [Beiz90]. Cost and expense, required time to test a program and staff needed can be estimated more precisely as a well established and defined procedure is followed.
Yet procedure-based approaches aren’t suited for automatic test case generation as the sole means of test case creation either. Existing testing tools for procedure-based approaches rely on code and structural information of the application under test to generate test cases. Accordingly procedure-based approaches are not inspected any further.

Be it mentioned once more: The boundaries between heuristic methods and procedure-based approaches are blurred as they depict an evolutionary gradual process.

2.5.3 Formal Test Case Derivation / Automatic Test Case Generation

To design and execute a system test is a demanding and challenging job. It would be a great help to testers to be freed of the toil and strain of test case derivation or at least to have some automated support in generating test cases. Methods in the two categories mentioned so far leave most of requirements-based test case generation to be done manually.

Trying to overcome the problems mentioned, a promising approach is being pursued: Test case generation based on **formal** specifications. Depending on the type of a formal specification (logic-based formal specifications, functional specifications, algebraic specifications, model or language-based formal specifications, specifications based on behavioral models) there are some more or less developed methods to derive test cases from specification. A representative method of each category is listed and shortly described below.

Logic-based formal specifications: Early work towards requirements-based testing has been done by the IBM Systems Development Division. In their approach the semantic content of a specification is analyzed, causes and effects are singled out forming the start and end nodes of a net. Subsequently, cause effect sequences are determined and con-

3. described in Section 2.4.1, Solution strategies, second strategy
4. meaning that a procedure is given on how to use these methods in a systematic manner to derive effective test cases and how to refine, sensitize - sort out and avoid unachievable paths - and instrument - avoid coincidental correctness, confirm that a correct result was obtained by the intended program path - test cases.

nected to start and end nodes thus forming a boolean graph [Elme73, Myer79]. In Cause-Effect-Graphs effects - all distinct output conditions or system transformations caused by changes in the state of the system - and causes - all input states and conditions - are determined based on the specification. Each effect is traced back to the conditions and the input data that caused it. Test cases are then derived or generated from valid input combinations. Cause-Effect-Graphs are a formal language founded on combinatorial logic networks. Syntactic and environmental constraints and restrictions (e.g. mutually exclusive conditions) are indicated in the graph[5].

In [Deut81] a similar approach as used by Hughes Aircraft Company is described. This approach is appropriate especially for reactive systems and uses Stimulus-Response elements. Stimuli are sensor data, input data or/and system conditions. Responses are correspondingly all system transformations and output data. Deutsch stresses the importance of properly identifying all requirements which are to be tested, and points out that requirements have to be traceable, that is: at all times it must be possible to pinpoint the function(s) and procedure(s) which implement and fulfill a given requirement, likewise that all test cases that verify a given requirement be identified. In this approach all Stimulus-Response elements, the so called threads which are derived from requirements specification directly, are depicted in a “System Verification Diagram” (SVD). Each thread is connected to a specific requirement and a identified function thereby revealing inconsistencies, redundancies and omissions in the specification and fulfilling the demand of traceability.

Algebraic specification: Bernot, Gaudel and Marre [Bern91] describe an approach based on an algebraic specification $SP = (\sum, Ax)$ whereas $\sum$ is the signature of the program and *Ax* are validation predicates stating certain equalities between terms. Starting from the ideal of an exhaustive test they show how to select test sets that are finite, decidable and maintain some good properties of an exhaustive test set. With reverence to [Good75] they assume that the test data selection problem consists of stating hypothesis *H* about the interpretation of *Ax* associated with program *P* and selecting a (finite!) data set *T* such that $H+Success(T) \Longrightarrow Correctness(SP, P)$. The *Success* oracle is a predicate deciding whether a program *P* passes or fails the test. In order to test a program *P* against its formal specification *SP*, they describe the construction and refinement of a test context (*H*, *T*, *Success*), including an automatic selection of test data *T*. Using this test context they check whether equality of terms stated in *Ax* also holds in the implementation. The approach is based on Horn clause logic. Test hypotheses are needed to represent and formalize common test practices and to reduce the size of the test set while preserving its validity.

Applying this approach Barbey and Buchs [Barb94] tested Ada abstract data types. Given a formal algebraic specification (for example by reengineering an existing component) they describe the automatic generation of test data sets that test Ada abstract data types against their formal specification. Furthermore they describe building and usage of an oracle which decides on the success or failure of a test. They show that their approach being based on rigorous semantics provides a complete test of all the axioms (properties) stated in the specification. For each operation, a complete and independent test set is generated, thus avoiding misleading combinations of operations.

5.Decision table models as described in [Beiz90, Good75] are another logic-based method based on requirements and specification and are closely related to Cause-Effect-Graphs as Cause-Effect-Graphs are converted to limited-entry decision tables to generate test cases.

Specifications based on behavioral models, functional specifications, process algebras[6]: In telecommunications conformance test methods have been developed to ensure that the implementation has specific properties described in the specification. Theoretical approaches try to prove such properties by defining conformance relations between specification and implementation. Test cases to establish confidence in the validity of these relations can be generated automatically [Holz91]. Unfortunately properties for which test cases can be generated are often limited to a behavioral coverage under restrictive additional conditions. That means that behavior must be specified in form of finite state automata and furthermore assumptions about the implementation are made thereby voiding the Black-Box-Criterion and leading to unnecessary dependencies. Another problem is the amount of necessary test cases which is often far to large for realistic tests.

Approaches in industry such as the ISO/IEC Standard 9646 “OSI Conformance Testing Methodology and Framework” (CTMF) describe a general way of testing an implemented protocol against its specification. CTMF describes general concepts for conformance testing, different test architectures, test methods and a language, the so called Tree and Tabular Combined Notation [Gies94], to describe a test suite and its implementation. Assuming a formally specified protocol (for example in SDL, Estelle or LOTOS) ISO 9646 defines the phases for the process of conformance testing. The test specification phase consists of the two main tasks test purpose development and test case development, followed by the phase test execution and test result analysis. CTMF describes the steps to be performed within a phase and the representation of results with different degrees of detail and formalization. Test purposes can be described in an informal way only. Thus test purpose and test case development have to be done manually by expert staff.

In order to improve this situation Grabowski at al. have developed the SAMSTAG approach [Grab94]: In this project they formalize the representation of test purposes by determining coverage criteria. Embedded in the ISO framework mentioned above formalization of test purposes leads toward an automatic derivation of test cases. Given a system specification written in a formal language like SDL (or Estelle or LOTOS) and a test purpose specified as a finite automaton (e.g. depicted in a Message Sequence Chart) the SAMSTAG method (or tool respectively) will try to find paths (called observations) through the SDL specification which will produce the signaling sequence that is specified by the Message Sequence Chart to accomplish the test purpose. The search for potential candidate paths which fulfill the mentioned criterion is not decidable for any SDL system. Consequently a negative search is canceled after a certain time. After a successful search a test case in the standardized CTMF language “Tree and Tabular Combined Notation” is output.

Another integrating approach for conformance testing of communication protocols based on finite state models is taken by S. Fujiwara, G. v. Bochmann et al. described in [Fuji91]. The so called partial W method is an improvement of the W method [Chow78] and yields shorter test suites. In a two-phase approach the first phase checks that all the states defined by the specification are identifiable in the implementation, at the same time checking the transitions leading from the initial state to these states for correct output and state transfer. Then the second phase checks the implementation for all the transitions defined by the specification which were not checked during the first phase. This approach relies on quite

6. asfor example λ - calculus (functions calculus) [Miln90] or π - calculus [Miln89]

a few assumptions as any test method based on finite state machines up to date does (Minimality and completeness of the specification, reachability of all states from the initial one, deterministic state machine, reset operation which needs to be implemented correctly).

2.6 Limitations and drawbacks of the current approaches to requirements-based testing

The methods mentioned in the preceding section are a step in the direction of automatic test case selection and generation. As such they are of great interest and importance, not to academia alone but to the software industry as well, as they help in reducing software development cost and significantly improve software test quality. Yet they are not widely used because of two problems that remain:

- Automated test case selection often does not yield effective test cases - effective here being defined as “of high probability to uncover yet undiscovered faults”. Testers relying on their intuition do a superior job.
- Any method that has been mentioned has some major drawbacks or relies heavily on restricting assumptions. Some of these restrictions are pointed out in the sequel.

Formal methods in general: One drawback that is true for all formal methods is, that they are difficult to understand, not only for developers, but more especially for the clients who are not trained in using/reading/interpreting them. To develop fully formal specifications and perform fully formal tests extremely skilled people are needed. When formal methods are applied, development can take three times as long as expected for a problem of a given complexity [Guih94]. Maintainability of formal specifications is a problem as well. Therefore formal methods are rarely used in industrial requirements specifications except for highly safety-critical systems, where the customer is willing to pay a very high premium for system safety.

Logic-based formal specifications, functional specifications: Cause-Effect-Graphing or the Stimulus-Response paradigm are attractive for testing reactive and distributed systems. However, the real problems lie in defining and testing the right sequences of stimuli and responses which can not (as yet) be automated. Other problems are the definition, formalization and inclusion of constraints in the analysis and the scaling to large projects (number of causes, effects and their interdependencies).

Algebraic specification: As an abstract descriptive formal method algebraic specifications suffer of all the problems mentioned above. Moreover, some distinct problems have been stated by Spence and Meudec [Spen94]:

1. It is difficult to write axiomatic specifications for complex data types.
2. A large number of test cases is generated. The oracle is not (yet?) automatically generated.

Barbey and Gaudel [Barb94] point out that both problems are minor as practical experience at an industrial level show the successful use of formal specifications in complex problems [Dauc93] and because the process of testing might be automated further.

Specifications based on behavioral models, process algebras: A serious limitation of these methods is their restriction to behavioral coverage and assumptions as have been mentioned that have to be made for test case generation.

ISO 9646 standardizes conformance testing, but lacks sufficient formalization of certain steps (e.g. formalization of test purpose). The SAMSTAG approach solves this problem, offering a semiformal representation and a properly defined process, but the approach is currently limited to protocol conformance testing. The partial W methods has the same limited domain.

3.0 Conclusions

As has been shown, on the one hand generation of test cases from formal specifications does not in general yield a sufficient coverage and, on the other hand, test suites that are automatically generated are generally far too extensive to be of practical use. There are specific fields like telecommunication where appealing results for test case generation have been obtained, but only under quite stringent assumptions and limiting testing to behavioral tests.

As it is not feasible at present to conduct automatic system tests, we will focus in our work on methods to support the test case designer in his tasks by exploiting systematic manual or semiautomatic procedures to test case design and generation.

The fundamental concept to our approach is, to use synergies between requirements validation and system test. Both activities are based on requirements, one trying to show conformity of the specification with user requirements, that is to ensure that the requirements adequately and completely state the needs of the users, the other undertaking to establish that the implemented system conforms to its requirements. Yet the two in practice are often viewed as separate, segregated activities of the software engineering process, having no relation, no connecting link one to the other. Applying the idea of scenarios to capture user requirements, validation of requirements and generation of test cases to perform system test are associated and linked: based on the selfsame scenarios the requirements specification can be validated by domain experts and test cases for the system test can be derived. Scenarios serve as a common basis to both activities.

Only little research has been done in this area up to now. We take up the notion of using scenarios or use cases to join requirements validation and system test as the topic of our research.

4.0 References

[Arth91] Arthan, R.D. On Formal Specification of a Proof Tool. In *VDM 91: Formal Software Development Methods*, Vol. 1. Prehn, S.; Toetenel, W.J. (Ed.), Springer Verlag.

[Balz85] Balzer, R.M. A 15 Year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering* **SE-11** (11), Nov. 1985, pp 1257-1268.

[Barb94] Barbey, S.; Buchs, D. Testing of Ada Abstract Data Types Using Formal Specifications. *Eurospace Ada Europe ´94 Symposium Proceedings*, Marcel Toussaint (Ed.), LNCS (Lecture Notes in Computer Sciences), no. 887, Springer Verlag, Copenhagen, Danmark, Sept. 9194: 76-89.

[Beiz90] Beizer, B. *Software testing techniques.* 2nd Ed. New York: Van Nostrand Reinhold.

[Beiz95] Beizer, B. *Blackbox Testing. Techniques for Functional Testing of Software and Systems.* New York: John Wiley.

[Berl87] Berliner, E. F.; Zave, P. An Experiment in Technology Transfer: PAISLey Specification of Requirements for an Undersea Lightwave Cable System. *Proceedings Ninth International Conference on Software Engineering*, IEEE Computer Society Press, Monterey, CA, March 1987, pp. 42-50.

[Bern91] Bernot, G.; Gaudel, M.C.; Marre, B. Software Testing Based on Formal Specifications: a Theory and a Tool. *IEE Software Engineering Journal*, **6** (6), 1991: 387-405.

[Boeh81] Boehm, B.W. *Software Engineering Economics.* Englewood Cliffs, N.J.: Prentice-Hall.

[BSI89] British Standards Institution: *VDM Specification Language: Proto-Standard*. IST/5/50.

[Chow78] Chow, T. S. Testing Design Modeled by Finite State Machines. *IEEE Transactions on Software*, 4, pp. 178-186, Mar 1978.

[Cohe83] Cohen, D. Symbolic Execution of the Gist Specification Language. *Proceedings Eighth International Joint Conference on Artificial Intelligence*, IJCAI-83, Karlsruhe, West Germany, 1983, pp. 17-20.

[Dauc93] Dauchy, P.; Gaudel, M.-C.; Marre, B. Using Algebraic Specifications in Software Testing: A Case Study on the Software of an Automatic Subway. *Journal of Systems and Software*, 21 (3): pp. 229.244, June 1993. North Holland, Elsevier Science Publishing Company.

[Dawe91] Dawes, J. *The VDM-SL Reference Guide.* Pitman London.

[DeMi79] DeMillo, R. A.; Lipton, R. J.; Sayward, F. G. Program Mutation: A New Approach to Program Testing. In Infotech int. Ltd.: *Infotech State of the Art Report on Software Testing*, Vol. 2, 1979, pp. 107-127, Infotech International Limited, Maidenhead, England.

[DeMi87] DeMillo, R. A.; McCracken, W. M.; Martin, R. J.; Passafiume, J. F. *Software Testing and Evaluation*. Benjamin/Cummings Publishing Company, Menlo Park, CA.

[Dill93] Diller, A. *Z: An Introduction to Formal Methods*. Wiley Chichester.

[Dura84] Duran, J. W.; Ntafos, S. C. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, 1984, pp. 438-444.

[Dyer92] Dyer, M. *The Cleanroom Approach to Quality Software Development*. John Wiley and Sons, New York.

[Elme73] Elmendorf, W.R. *Cause-Effect-Graphs in Functional Testing*. TR-00.2487, IBM Systems Development Division, Poughkeepsie, New York.

[Faga76] Fagan, M.E. Design and Code Inspection to Reduce Errors man Program Development. *IBM Systems Journal* **15** (3), 1976: 182-211.

[Free82] Freedman, D.P.; Weinberg, G.M. *Handbook of Walkthroughs, Inspections and Technical Reviews.* Boston, Toronto: Little, Brown and Company.

[Fuji91] Fujiwara, S.; Bochmann, G. v.; Khendek, F.; Amalou, M.; Ghedamsi, A. Test Selection Based on Finite State Models. *IEEE Transactions on Software Engineering*, **17** (6), 1991.

[Gies94] Giessler, A.; Baumgarten, B. Die OSI-Konformitätstestmethodik. *PIK* **17**, 1 (1994). München: K. G. Saur Verlag, 4-12.

[Goed31] Goedel, K. *Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme.* Unvollständigkeitstheoreme / incompleteness theorems as republished in [Goed86].

[Goed86] Goedel, K.; Feferman, S. (Editor). *Collected works.* Vol. 1. Oxford University Press New York.

[Good75] Goodenough, J.B.; Gerhart, S.L. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering,* **SE-1** (2): 156-173. (see also SIGPLAN Notices 1975, 10 (6)).

[Gord87] Gordon, M.J.C. HOL: A Proof Generating System for Higher-Order Logic. In Birtwistle, G.; Subrahmanyam, P.A. (Ed.), *VLSI Specification, Verification and Synthesis*. Kluwer, 1987.

[Grab94] Grabowski, J.; Nahm, R.; Spichinger, A.; Hogrefe, D. Die SAMSTAG Methode und ihre Rolle im Konformitätstesten. München: K.G. Saur Verlag. *PIK* **17** (4), 1994: 214-224.

[Guih94] Guiho, G. Operational Safety Critical Software Methods in Railways. *Proceedings 13th IFIP World Computer Congress 94*, vol. 3, pp, 262-269.

[Gutt93] Guttag, J. V.; Horning, J. J.; Garland, S. J. *Larch: Languages and Tools for Formal Specification*. Springer Verlag.

[Hols89] *The HOL System: Description*. SRI International, Dec. 1989.

[Holz91] Holzmann, G.J. *Design and Validation of Computer Protocols*. Englewood Cliffs, N.J., Prentice-Hall International.

[Howd76] Howden, W.E. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering* 2: 208-215.

[IEEE90] *Standard Glossary of Software Engineering Terminology.* IEEE Std 610.12-1990. IEEE Computer Society Press.

[Jone90] Jones, C. B. *Systematic Software Development Using VDM.* Prentice-Hall.

[Loo88] Loo, P. S.; Tsai, W. K. Random Testing Revisited. *Information and Software Technology,* Vol. 30, No. 7, pp. 402-417.

[Mali92] Malik, S.; Yamanchi, H. *CSP: A Developer´s Guide.* McGraw-Hill.

[Miln89] Milner, R.; Parrow, J.; Walker, D. A Calculus of Mobile Processes. *Reports ECS-LFCS-89-85 and -86,* Laboratory for Foundations of Computer Science, Edinburgh University, 1989.

[Miln90] Milner, R. Functions as Processes. *Proceedings ICALP ´90, Lecture Notes in Computer Science*, Vol. 443, pp. 167-180, Springer Verlag.

[Myer79] Myers, G.J. *The Art of Software Testing.* John Wiley & Sons, New York.

[Ostr88] Ostrand, T.J.; Balcer, M.J. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM* 31 (6): 676-686.

[Preh91] Prehn, S.; Toetenel, H. (Ed.) *Formal software development methods. VDM 91: 4th international symposium of VDM Europe*. Springer Verlag.

[Ried92] Riedemann, E.H. Testen von Software durch Mutationsanalyse: “ja bitte” wegen der hohen Qualität oder “nein danke” wegen des Aufwands und der prinzipiellen Probleme? In H.-J. Kreowski (Ed.) *Informatik zwischen Wissenschaft und Gesellschaft*, Informatik-Fachbericht 309, pp. 202-217, Springer Verlag, Berlin/Heidelberg.

[Spen94] Spence, I.; Meudec, C. Generation of Software Tests from Specifications. In M. Ross, C.A. Brebbia, G. Staples and J. Stapleton (Ed.): *SQM´94 Second Conference on Software Quality Management*, vol. 2, pp. 517-530. Edinburgh, Scotland, UK, July 1994.

[Spiv92a] Spivey, J. M.; Abrial, J.-R. *The Z Notation: A Reference Manual.* Prentice-Hall International, Series in Computer Science.

[Spiv92b] Spivey, J. M. *Understanding Z: A Specification Language and it´s Formal Semantics*. Cambridge University Press.

[Stak89] Staknis, M.E. The Use of Software Prototypes in Software Testing. *Sixth International Conference on Testing Computer Software*, Washington, DC, May 89.

[Thay78] Thayer, R. A.; Lipow, M.; Nelson, E. C. *Software Reliability*, Amsterdam, the Netherlands.

[Thev91] Thévenod-Fosse, P.; Waeselynck, H.; Crouzet, Y. An Experimental Study of Software Structural Testing: Deterministic versus Random Input Generation. In: *Proceedings 21st IEEE Symposium on Fault-Tolerant Computing (FTCS-21)*. Montreal, Canada, pp. 410-417, IEEE Computer Society Press, 1991.

[Thev93] Thévenod-Fosse, P.; Waeselynck, H. Statemate Applied to Statistical Testing. In: *Proceedings of the International Symposium on Software Testing and Analysis* (ISSTA ´93). Cambridge, Ma, pp. 99-103, 1993.

[Thev95] Thévenod-Fosse, P.; Waeselynck, H.; Crouzet, Y. Software Statistical Testing. In: *Predictably Dependable Computing Systems*. Editors: B. Randell, J.-C. Laprie, H. Kopetz, B. Littlewood. SpringerVerlag, ESPRIT Basic Research Series.

[Wein71] Weinberg, G. *The Psychology of Computer programming.* New York: Van Nostrand Reinhold.

[Word92] Wordsworth, J. B. *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering.* Addison-Wesley.

[Your77] Yourdon, E. *Structured Walkthroughs.* New York: Yourdon Press.

[Zave82] Zave, P. An Operational Approach to Requirements Specification for Embedded Systems. *IEEE Transactions on Software Engineering* **SE-8** (3), May 1982, pp. 250-269.

[Zave84] Zave, P. An Overview of the PAISLey Project. *ACM SIGSOFT Software Engineering Notes*, **9** (4), July 1984, pp. 12-19 (correction in **9** (5), Oct 1984, p.10)

[ZOCL92] *Z Base Standard*, Version 1.0, Oxford Computing Laboratory, 1992