Automation of Requirements-based Testing

Ralf Gerlich, Rainer Gerlich
Dr. Rainer Gerlich BSSE System and Software Engineering
Immenstaad, Germany
Ralf.Gerlich@bsse.biz,
Rainer.Gerlich@bsse.biz

Maria Hernek
ESA/ESTEC
Noordwijk, The Netherlands
Maria.Hernek@esa.int

Allan Pascoe, Glenn Johnson
SCISYS UK Ltd.
Bristol, UK
Allan.Pascoe@scisys.co.uk,
Glenn.Johnson@scisys.co.uk

***Abstract*— Manual *requirements-based testing* is time-consuming: Input data must cover the requirements and observed output data must be checked for their compatibility with the requirements. Testcases can also be automatically generated from test models. However, these models first have to be established manually. In contrast, the approach to be presented here uses simpler ways of formalizing requirements to automatically map test data generated for *automatic robustness testing using massive stimulation* to requirements and to check the results for correctness.**

Keywords: requirements verification, automated software test, fuzzing, massive stimulation, software verification

I. Introduction

Fig. I-1 shows the classical approach to requirements verification using function or unit tests and manually designed test cases. The source code is written based on the requirements, and a mapping between functions and requirements is established manually. Test cases for unit testing are manually derived from the requirements. Input data and expected outputs are transformed into test scripts, which are used to execute the tests and generate a pass/fail verdict.

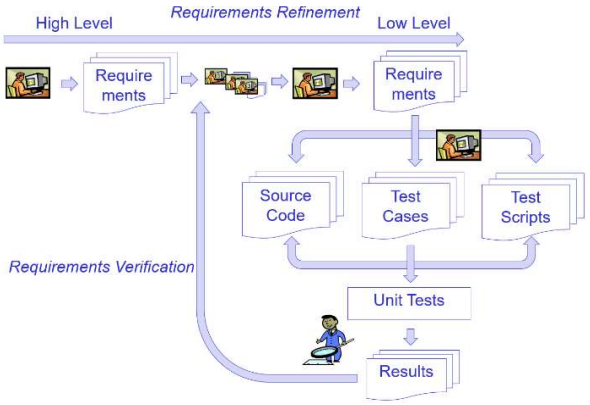


Fig. I-1: Verification by manually established test cases

II. Model-driven Approaches to Requirements Testing

In the last few years various approaches for deriving tests for requirements verification from models have been proposed. For this, the desired behaviour is modelled and annotations for determining requirements coverage and checking of requirements fulfilment are added [1]. Using random walks or similar methods test sequences and test data are derived from the models and transformed to test scripts manually or automatically.

In the ideal case a specification model is available which can be used for such a method with only few modifications. Otherwise, a test model needs to be established and verified against the specification manually – with the latter often only being present in text form.

A. Automated Robustness Testing

Automated robustness tests – sometimes also referred to by the term "fuzzing" – are used to test a component for robustness against unexpected or undesired inputs[2]. The component is stimulated using, e.g., random data, and the behaviour of the component is monitored for anomalies during execution. This may even allow identification of functional defects in the component. The simple form of test data generation using random data allows for a high test data throughput – resulting in *massive* stimulation.

The results of these tests, however, do not allow for conclusions about the fulfilment or coverage of requirements. A mapping from test inputs to requirements is missing.

B. Automated Requirements-based Testing

Automated requirements based testing on source code-level requires and aims to automatically establish a correlation between requirements and test cases. At the same time the mapping between the requirements and the affected functions in the code needs to be found. Test data shall be automatically generated and applied, in order to reduce the manual effort drastically.

The selected approach uses three mappings [3]:

- between input data to requirements (requirements coverage),
- between requirements and functions, and
- between requirements and *oracles*.

This principle is shown in Fig. II-1. machine-readable requirements are integrated with the function test and evaluated during massive stimulation. This allows determination of both requirements coverage and fulfilment. In the case of non-fulfilment the associated test data vector describes a counter-example.

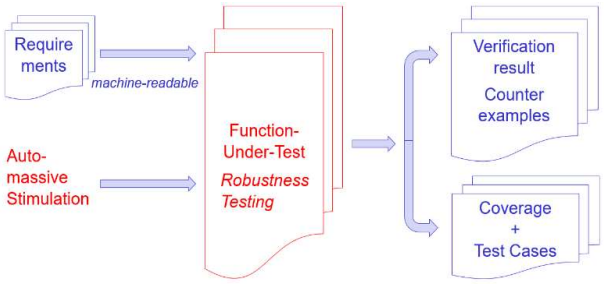


Fig. II-1: Principal Approach

Oracles are used for evaluation of requirements fulfilment. An oracle is an executable procedure which generates a *pass/fail/don't-know*-verdict for a pair of input and observed output. It is possible for single oracles to only provide definite results for specific real subsets of the input domain. Ideally all oracles together cover the whole input domain.

Fig. II-2 shows further details of the automatic approach. The input space is sampled by a configurable number of samples, using massive stimulation. The oracles are applied to each test vector and the resulting verdict is recorded. This way, counter-examples for non-fulfilment of requirements can be found.

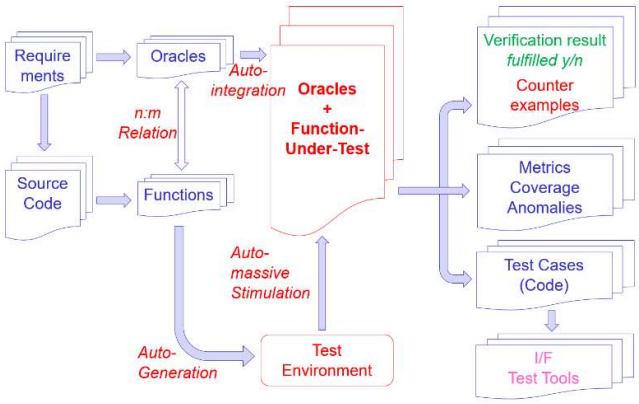


Fig. II-2: Process Details

Requirements and functions generally form an n:m relationship. Specific requirements can apply to multiple functions, such as requirements on numerical accuracy. Similarly, a single function can implement multiple requirements or contribute to their implementation.

To map requirements and functions to each other, the affected elements – data objects, structures, parameters – must be extracted from the text of the requirement and be associated to their counterparts in the Code. This can be done using naming rules which translate names from the requirements to names in the code – possibly using an identity mapping. This way oracles can be mapped to functions by concatenating the relationships of oracles to requirements and requirements to functions. An elaborate manual mapping can be avoided this way.

Fig. II-3 illustrates the logical flow in the context of hierarchical requirements. Top-down requirements should be detailed down to a level at which they can be transformed into code. This is the highest level at which oracles can be defined in a meaningful manner. Requirements on higher level are usually not suitable for verification by code-level- or unit tests.

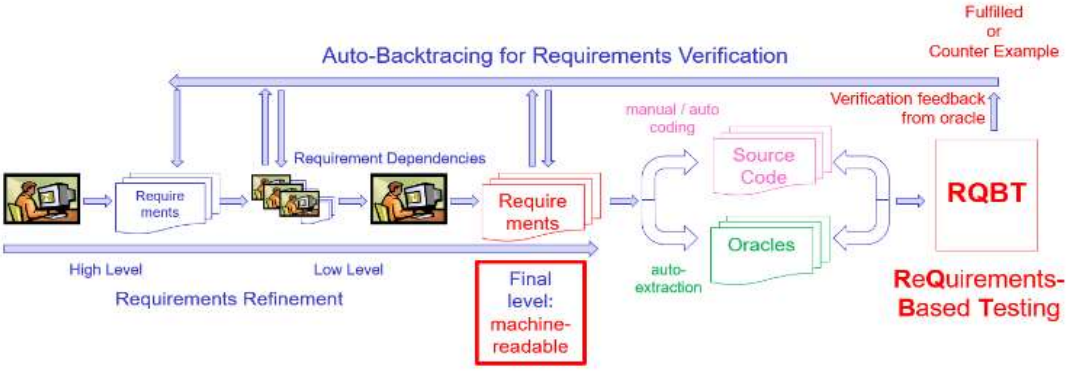
Finally, the mapping from inputs to requirements is used to determine requirements coverage and to back-trace it through the different requirement hierarchy levels up to high-level requirements. This back-tracing is done using the traceability information provided in the form of "refines"- or "implements"-relationships between the different levels of requirements.

C. *Massive Stimulation and the Derivation of Test Vectors*

For each function a test environment is established automatically, which stimulates the function using data from the input domain of parameters – including global variables[2]. Special cases such as -1 or 0 for integers are considered specifically. Other methods for targeted coverage of specific parts of the code are used, such as constraint-based or other search-based approaches to test data selection. In addition, invalid values can be injected and parts of the code can be specifically modified or supplemented, e.g. to simulate lack of memory and similar conditions.

Interesting test case candidates for generation of regression test suites are then selected from the massive set of test inputs. An input may be interesting if it contributes to requirement or code coverage, elicits exceptional behaviour or violates the rules implied by an oracle. These suites can also be re-executed with external test management tools – such as Cantata or VectorCAST – and their results can be re-evaluated.

Fig. II-3: Overall Process for Hierarchical Requirements



III. THE ORACLE APPROACH

In this approach the oracles are represented as temporal implications: *If* Condition A holds for the input *before* calling the function, *then* Condition B must hold for the tuple of input and output *after* execution of the function (Fig. III-1).

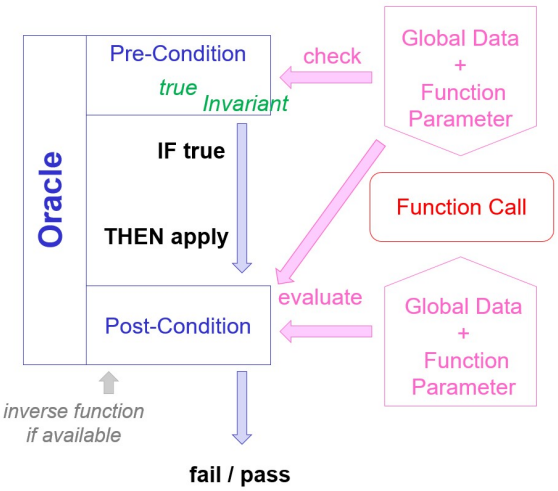


Fig. III-1: Structure of Oracles

If Condition A is not fulfilled before the function call, Condition B is not evaluated and the oracle provides *don't know* as result. Here, Condition A is called the pre-condition.

Using this structure a tautology – i.e. an expression that is always true – can be used as pre-condition. In this case, Condition B must hold for every test vector. This way, e.g., restrictions for the ranges of the results can be expressed.

Using known relations between requirements at different levels the results can be mapped to higher-level requirements. This way on each level the functions contributing to positive or negative verification results can be determined.

Although verification using massive stimulation is based on a large number of test vectors, the set of these test vectors is finite after all. Thus a complete analysis of the input space is usually not possible. However, the number of automatically generated stimuli usually exceeds by far the number of test cases providable using manual methods – which is relevant specifically for finding counter-examples.

A. Examples

Consider as an example an oracle for the square-root-function. The simple approach $x \geq 0 \Rightarrow \sqrt{x}^2 = x$ would be mathematically sound. However, it is not applicable where numerical precision is finite. A correct approach for normalised floating point values would be $x \geq f_{min} \Rightarrow \left\| \frac{\sqrt{x}^2 - x}{x} \right\| < \varepsilon$, where f_{min} is the smallest normalized floating-point number. For $0 < x < f_{min}$ absolute error limits would have to be used.

An oracle for the abs-function from the C Standard Library seems even simpler: $x \in int \Rightarrow abs(x) \geq 0$. Interestingly the value of *abs(INT_MIN)* is negative, as *INT_MIN* itself cannot be represented as a value of *int*.

Type ranges are also to be considered in other cases: The oracle $\left\| \frac{\sqrt{x^2} - x}{x} \right\| < \varepsilon$ for the square function will produce many apparent counter-examples, as for many values of x squaring them leads to a float-point overflow.

Thus also here – as for many other approaches of formalising requirements – the possibility of detecting incomplete and inaccurate requirements is present.

Truth tables, e.g. for system states, can be easily represented, as Fig. III-2 shows.

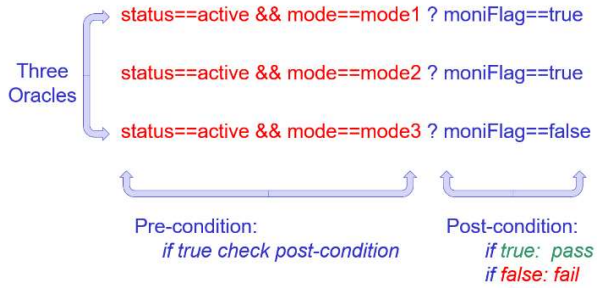


Fig. III-2: : Checking System States using Oracles

IV. OPEN ISSUES

A. Requirements Notation

At the moment most requirements are expressed in free text, which cannot be automatically evaluated. An analysis of such requirements led to the conclusion of them being incomplete, ambiguous or inconsistent, and thus not being applicable for this method. In manual verification, these shortcomings have to be compensated by creativity.

A pre-requisite for formalisation of requirements is an adequate notation. This notation can also come in a form more suitable to the user than the oracle form. However, it has to be automatically transformable into the oracle notation. Requirements presented in a formalised table structure are convenient for this.

As far as functional aspects of a software system go, there are a number of approaches to choose from for formalising requirements, such as Abstract State Machines[4], a generalisation of Finite State Machines, Petri Nets, or operation contracts in the form of pre- and post-conditions, to name just a few. Logical foundations can be found in temporal logic, with linear temporal logic being one of the most prominent variants due its use in model checking.

Sequence-based specifications[1][5] describe the behaviour of the system in terms of stimuli and the resulting responses, but also come with a concept of determining such specifications by enumerating possible stimulus-response sequences from existing preliminary or finalised requirements documents, reducing them to a finite description at the desired level of detail.

Although the domain of non-functional aspects is much larger, important concepts such as timing can be introduced in many of these models using appropriate annotations, such as for response-times or maximum buffer sizes.

These approaches describe the theoretical foundations which are at the base of the respective form of description.

While the underlying theory guides both the required semantic content as well as its basic structure, it is possible to choose from various forms of presentation. For example, abstract state machines can be represented in a graphical manner – as long as a semantical model is stored in parallel –, in the form of state-transition tables or as pre- and post-conditions establishing the relationship between source- and target-states of any transition.

Care must be taken when specification models specify features that are not explicitly present in the final software. One example might be a specification using finite state machines, without there being a variable in the software explicitly representing the current state from the state machine. Instead, the state may be implicit in that some sequences of operations simply would lead to undesired results. In such a case, abstract state machines making the preconditions of operations explicit may be a more useful modelling tool.

On a conceptual level, the idea of formalising software requirements and design is not very different from specifying, e.g., control systems. Here the underlying system dynamics is formally described using differential equations or process graphs and the requirements can be defined in terms of the desired dynamic properties of the controlled system. The use of such concepts is quite well-established in the specification, design, verification and validation of control algorithms.

While it might not be possible to express all aspects of a software system in a formal way, there is a plethora of methods to chose from.

B. Quality Assurance of Oracles

Like any other code, also oracles are subject to quality assurance. Mistakes in oracles could otherwise lead to overlooking possibly critical software faults. As each oracle can be applied to large subsets of the input domain, it is even possible for faults with huge impact to be overlooked if the oracle is incorrect. A similar risk exists in manual testing, albeit for a different reason, namely not considering relevant test cases in the first place or constructing the expected output in an incorrect manner.

V. Outlook and Future Work

As no machine-readable requirements were available, the current implementation is based on oracles manually implemented in C, with the aim of showing feasibility and advantages.

In future work a more abstract form shall be identified which is also more acceptable to users. For this, text-based requirements shall be analysed and transformed into an adequate notation which can be automatically translated into oracles. For this close contact to potential users is necessary.

Acknowledgment

The research presented here is supported by grant DLR-50PS1601 of the Space Administration of the German Aerospace Center (DLR) on behalf of the German Ministry of Economics and Energy (BMWi).

References

[1] H.-J. Herpel, G. Willich, J. Li, J. Xie, B. Johansen, K. Kvinnesland, S. Krueger, P. Barrios: “MATTS – A step towards Model Based Testing”, Eurospace Symposium DASIA’2016 “DAta Systems in Aerospace”, May 10th-12th, 2016, Tallinn, Estonia.

[2] R. Gerlich, R. Gerlich, M. Prochazka, K. Kvinnesland, B. Johansen: “A Case Study on Automated Source-Code-Based Testing Methods”, Eurospace Symposium DASIA’2013 “DAta Systems in Aerospace”, May 14th-16th, 2013, Porto, Portugal.

[3] R. Gerlich, R. Gerlich, M. Hernek, J. Ramachandran, A. Pascoe, G. Johnson: “Challenges Regarding Automation of Requirements-based Testing”, Eurospace Symposium DASIA’2017 “DAta Systems in Aerospace”, May 30th – June 1st, 2017, Gothenborg, Sweden.

[4] E. Börger, R. Stärk: „Abstract State Machines: A Method for High-Level System Design and Analysis”, Springer, 2003.

[5] S. J. Prowell: “Sequence-Based Software Specification”, Dissertation, University of Tennessee, Knoxville, 1996.