Challenges Regarding Automation of Requirements-based Testing

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

Maria Hernek, Jinesh Ramachandran
European Space Agency
Noordwijk, The Netherlands
e-mail: Maria.Hernek@esa.int, Jinesh.Ramachandran@esa.int

Allan Pascoe, Glenn Johnson
SCISYS UK Ltd.
Bristol, UK
e-mail: Allan.Pascoe@scisys.co.uk, Glenn.Johnson@scisys.co.uk

Abstract—**Testing as a method of software verification is limited in that it can only prove the presence of defects, not their absence. To be useful, a large number of test cases may be needed, a strategy that is often in conflict with project constraints such as available time and funds. Test automation may be considered as an interesting approach to alleviating this conflict. However, test automation requires accurate and computer-accessible information about the system to be tested, both in terms of the interfaces by which the system is to be stimulated as well as the desired properties of these interfaces. Within the FASTII activity (FAST=Flow-optimised Automated Source-code based Testing) the possibility of deriving this information from available requirements and design documents is being investigated. Preliminary results of this investigation as well as suggestions for future changes in the process are presented in this paper.**

Keywords: automated software test, requirements-based testing, requirement semantics, test oracles, software defects, defect identification, software verification, verification efficiency

I. INTRODUCTION

Testing as a method of verification or validation is limited in its expressiveness by the fact that it cannot prove the absence of defects. Instead, absolute statements are only possible about the behaviour of a system for the test cases applied, but not for any other cases.

Considering the internal structure of the system, the results may be extrapolated towards other cases using methods such as equivalence class analysis. Such approaches may be error-prone as they require theoretical, often manual reasoning about the actual limits of valid extrapolation.

Statistical methods can be applied to estimate the reliability of the system, but in order to provide a sufficiently accurate estimate, they may require large numbers of test cases.

At the same time, often project constraints limit the number of test cases which can be selected and applied within the confines of the effort allocated to testing. Specifically for manual test design, the number of test cases required for an appropriate reliability estimate may be prohibitively high.

Automated software testing may allow for a reduction of the mean effort spent on a single test case, thus increasing the number of test cases that can be applied within the allotted frame of effort.

It may also provide a more systematic means of selecting test cases, thereby enhancing the statistical representativity of the test results.

However, automation of software test requires that the information needed for all steps of the test process are available in a manner that can be evaluated by an algorithm. In case of automated requirements-based testing, this specifically requires information about the actual functionality described by a requirement, but also about what stimuli would trigger the functionality associated with the requirement in order to ensure that there are sufficient test cases generated exercising each individual requirement.

The FAST process (Flow-optimised Automated Source-code based Testing) is comprised of a set of procedures and tools to be used for software testing using massive stimulation. A basic breakdown of the process is shown in Fig. I-1.

Within the FAST process there are several levels of testing, each one building upon the previous one.

The most basic level is that of massive stimulation, exposing the software to large numbers of stimuli with the goal of identifying general issues such as runtime exceptions or possible non-termination. While not aimed at functional verification, the evaluation results may very well point at functional defects.

At the second level, a subset of the stimuli and the observed outputs are selected automatically as test case candidates. They are not actual test cases as the outputs in these test cases need to be confirmed against the requirements. They are also not selected with the goal of providing coverage regarding the requirements, so that the selected candidates may not sufficiently address the requirements imposed upon the software.

Currently this confirmation and assessment of requirements coverage would have to be done manually.

Within the FASTII activity we analysed the Software Requirements Document (SRD) and the Software Design Document (SDD) of a representative spacecraft on-board software in order to determine whether the information required for automated requirements-based testing is present in these documents and could be extracted, either manually or automatically. The results of this analysis shall be presented herein.

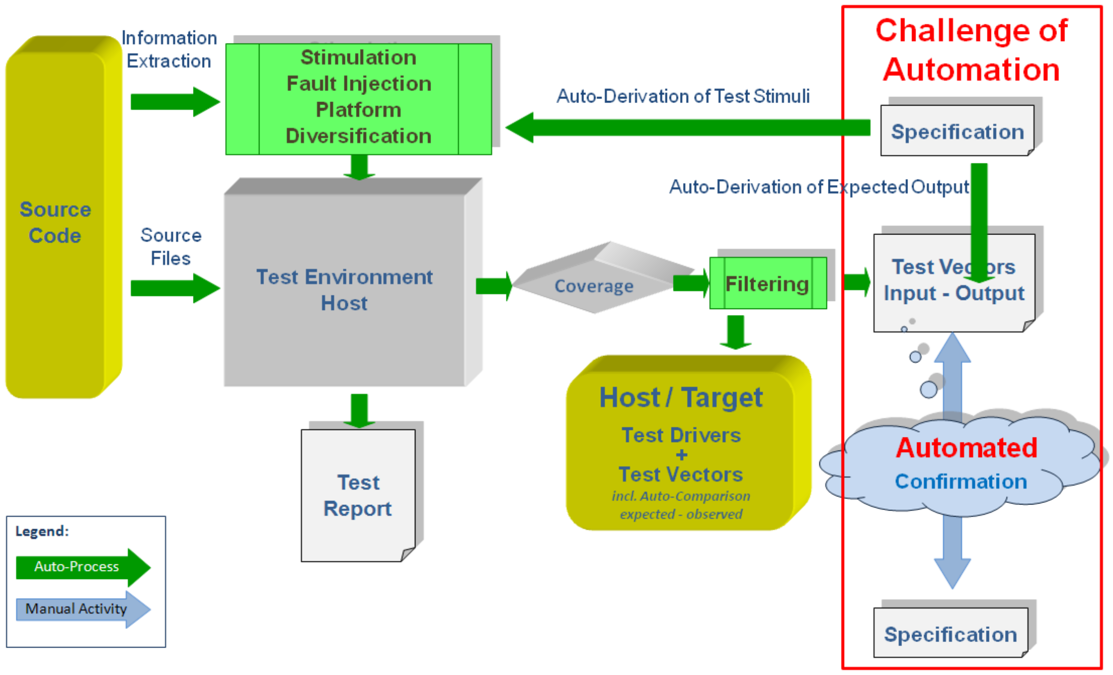

Fig. I-1: Automated Requirements-based Testing in the FAST Process

This paper is structured as follows: In Chapter II, we provide terms and definitions, followed by a presentation of the results of an analysis of requirements and design documents from a reference project in Chapter III. In Chapter IV we discuss options for the retrieval of information, including options for changing the way requirements and design documents should be written, and in Chapter V we provide conclusions and an outlook on future work.

II. Definition of Terms

Software testing aims to find faults in existing software by exposing the software – or parts thereof – to pre-fabricated stimuli, observing the reaction of the software and ascertaining whether the reaction conforms to the expected behaviour of the software given the stimuli.

Such a combination of stimuli and expected reactions/outputs is usually referred to as a *test case.* A set of test cases used in combination to test a specific part of software is referred to as a *test suite*.

If the actual behaviour of the software matches the expected behaviour, the test case is said to be *passed*, otherwise it is said to *fail*. This result is called the *verdict*.

The process of software testing consists of

- the selection of stimuli,
- the determination of the expected behaviour of the software under test,
- the injection of the stimulus,
- execution of the software-under-test,
- the extraction of the actual reaction of the software under test, and
- the determination of whether the actual behaviour of the software under test conforms to the expected behaviour.

Algorithms that specify in a generic manner how to check the conformance of the actual reaction to the expected reaction are called *oracles*.

Such oracles may also be incomplete or incorrect, meaning that they may flag test cases as failed although they succeeded – so-called *false positives* – or they may flag test cases as successful although they failed – so-called *false negatives*.

False negatives may lead to defects staying undetected, which may be acceptable to a project up to a certain level. Essentially, even with a correct oracle, testing can never be free of false negatives.

False positives lead to analysis effort without gain – a failure report needs to be understood without detecting and thus eliminating a bug. However, if the proportion of false positives is small enough, this additional effort may be outweighed by the effectiveness and efficiency gained by executing and evaluating a large number of test stimuli automatically.

The reason for implementing an incorrect or incomplete oracle may be saved effort on the side of the oracle implementation: Sometimes the cost and effort for implementing a perfect oracle may be prohibitively large, while a good but imperfect oracle might fit within the budget constraints and still improve the effectiveness and efficiency of the testing process. Ultimately, the decision needs to be based on a balance of cost and effort.

One specific property of testing in general and software testing in particular is that except for the simplest cases, testing cannot be complete and thus the absence of false negatives cannot be guaranteed.

It may be possible to partition the input space into so-called *equivalence classes*, where any stimulus from a class can be replaced by any other stimulus from the same class in terms of bug detection. This would mean that given n equivalence classes, n test cases would be sufficient to detect any bug in the respective function.

However, equivalence class testing in its pure form is a rather theoretical concept. Actually constructing equivalence classes regarding all possible bug types for a specific piece of code may be cumbersome and lead to a high number of equivalence classes which – although reduced in numbers – are as impracticable to test as the complete input space.

Instead, the adequacy of a test suite is often measured by *test coverage metrics*. Such metrics express the degree to which the test suite exercises the relevant functionality of a piece of software.

Most typically, *structural test coverage* metrics are used, which are based on the portion of structural code elements – such as statements, branches, conditions – that are reached and stimulated by executing the respective test cases. One example is *statement coverage*, which measures the relative proportion of all statements executed during the test. Another one is *branch coverage*, wherein the goal is to have each branch in the code executed in each of the possible alternatives. This is also called *decision coverage*. A variant thereof is *modified condition/decision coverage* – or MC/DC for short – where the test suite shall ensure that each of the individual parts – the conditions – of the boolean expression driving a decision has independently influenced the outcome of the decision.

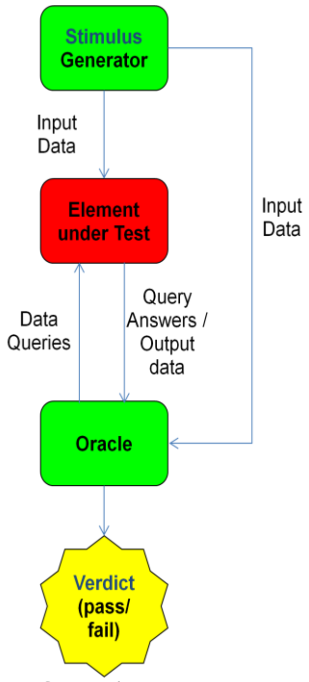

Fig. II-1: Generic Automatic Test Process

Software unit testing is software testing applied to the so-called unit level. The term *unit* usually refers to the smallest non-separable functional elements of a software product. Often these are the individual functions or procedures defined at code level, but a unit may also be composed from a group of such functions which are intended to be used in combination in order to provide atomic functionality. The latter is often the case when object-oriented methods are used in the design and implementation of the software.

Automated software testing refers to the process of performing the task of software testing in an automated manner. Within this document, the term shall be used specifically to mean the full automation of the process, consisting of all steps of the software testing process. A possible generic design of such an automated software test process is shown in Fig. II-1.

A stimulus generator selects test inputs, which are then injected into the element under test. As a reaction to the inputs, the element under test may generate outputs, which are passed on to the oracle. The element under test may also be queried for further results and the answers to these queries may be passed on to the oracle as well. The oracle also receives the original test inputs.

When all this data is available to the oracle, it delivers a verdict, indicating whether it considers the test to be passed or failed.

For all these elements to be executable by a computer, their rules of operation must be well-defined so that a computer – a non-sentient, mechanical device without additional knowledge – can execute them.

III. Analysis of Existing Requirements

In order to determine whether the information as required for automated requirements-based testing could be found in typical SRDs, and whether extraction of that information could happen automatically, manually or in a mixed manner, we considered requirements from an SRD of representative spacecraft on-board software.

Due to effort constraints only a subset of requirements was selected for analysis. In the specific case, the analysed requirements concern the interface between the central computer and the GPS receivers on-board the spacecraft. We believe that these are quite representative for interfacing requirements.

A. Method

The requirements were analysed for their semantic content concerning the functionality to be implemented in code. To identify a basic testing approach, the semantic content was elicited manually by answering a generic question for each of the requirements: When is the requirement violated?

Besides being complementary to the approach taken to requirements definition during authoring of the document – where the requirements are usually written to express positive functionality – this analysis approach is also closer to the mechanism of software testing, which aims to reveal non-compliance by way of executing test cases.

B. Requirement Classification

In total, 87 individual requirements were analysed. Each of them could be assigned to one of six requirements classes, based on their contents. A list of these requirements classes together with statistics is given in Tab. III-1.

Category	**No. of Reqs.**
Functions	46
Housekeeping	8
Communications	19
Monitoring	7
Telemetry Generation	3
Telecommand Handling	3
Total	**87**

Tab. III-1: Statistical Overview over analysed Requirements

For each of the classes, a generic testing approach was defined. This testing approach not only defines a generic – sometimes very high-level – principle on how to test the requirements from the given class, but also allows a closer look at what information is needed to test the given requirements. Some examples – in simplified form – are given in Tab. III-2.

Function requirements are defined in terms of pseudo-code, each describing a specific subprogram, including its input parameters and results.

Housekeeping requirements contain instructions for storing specific information upon fulfilment of specific conditions or occurrence of specific events for housekeeping purposes.

Communication requirements concern the format conversion and routing of telecommand (TC) and telemetry (TM) packets.

Monitoring requirements specify how the status of specific aspects of the system is to be monitored. This does not include any reactions to the change of status, such as TM generation or fault isolation and recovery.

Telemetry Generation requirements specify that telemetry packets should be generated and routed to a given destination upon given events – e.g. regular clock ticks or specific triggers.

Finally, *Telecommand Handling requirements* define the reaction towards specific telecommands being received.

1) Example: Communcation Requirements

Many of the requirements in the Communications category were found to be in this or a very similar form:

> *"Upon receipt of the XYZ telecommand the Nominal Mode Software shall route the telecommand to GPS in accordance with the format and procedures specified in [REF_ICD]."*

Here, [REF_ICD] refers to an interface control document (ICD) that describes the format and procedures mentioned.

A very basic understanding – if read by a human – reveals that according to the requirement, the nominal mode software shall convert any incoming telecommands of type XYZ to the format defined in the referenced ICD and trigger transmission towards the GPS. In how far the procedures defined in the ICD imply that additional steps are necessary – e.g. processing of acknowledgements, procedures for the initiation and teardown of communication, windowing, etc. – is not clear from the requirement itself, but hopefully would be when looking at the ICD.

Let us break down the requirement into its parts. It consists of:

- a condition/trigger: receipt of the XYZ telecommand
- a subject: the nominal mode software
- an activity: route to the GPS
- an object: the telecommand
- a qualifier: in accordance with the format and procedures defined in [REF_ICD].

Note that although structurally the qualifier seems to be merely an addition to the other elementary parts of the requirement, it can have immense impact on its implementation, as already indicated above.

Let us for a moment forget the difficulties associated with natural language parsing like resolving indirect references such as "the telecommand" as present in the "object" part of the requirement shown above. Let us instead assume that this problem was solved – which it is not – and that we have already concluded that the XYZ telecommand is being referred to.

In general, to test such a requirement, we need to initiate the given trigger or establish the given condition and observe whether the respective activity is executed, considering the aspects given by the qualifier. However, this is very general and clearly not yet on a level that can be translated into concrete operational steps.

First of all, how and where is the XYZ telecommand received? We need to know where to inject it into the subject, which is the nominal mode software. This information clearly is not present within the requirement, and hopefully is to be found elsewhere, possibly in other requirements not explicitly referenced here.

Also, what actually is the nominal mode software? How do we ensure that it is this software we are talking to?

For example, the nominal mode software can either be implemented as a specific mode of an overall software image, activated or deactivated by an in-memory switch.

It could also be a specific software image that is activated from the bootloader by directing control flow towards the entry point of this image after boot-up.

These two cases require two very different actions from the side of the test setup – either setting the mode switch appropriately or ensuring that boot-up ends within the nominal mode software image. None of these are further specified in the requirement, so we would have to search elsewhere for this information.

What does "routing to the GPS" mean and how do we verify that it is done? Usually it means sending out the packet over some specific communication channel to which both the computer running the nominal mode software as well as the GPS are connected. But which channel is this and how do we check for transmission via this channel?

Class	Text	Testing Approach
COMM	It shall be possible for the Nominal Mode SW to command both the nominal and redundant GPS units providing they are switched ON (as defined by the current satellite configuration vector).	This requirement is violated if it is impossible for the Nominal SW to command GPS units which are switched on.
	Upon receipt of the "XYZ" TC of [TC_REF] the Nominal Mode SW shall route the TC to GPS in accordance with the format and procedures defined in [REF_ICD].	Proper conversion can be checked, e.g., by back-conversion and comparison with the original.
HK	The data contents of the following GPS TM packets shall be stored within the system data pool: GPS Message TM, Primary Message TM	These requirements can be violated by not storing the data from the respective TM packets in the system data pool or not storing them separately.
FUNC	This function process_GPS_data processes the raw GPS data. Its interface is summarised in Table xxx.	The detailed description of the algorithm can be used to provide a *reference implementation* which can be used as an oracle.
	This function GPS__navigation_function shall generate the position and velocity in the inertial J2000 reference frame for the current time for the following cases: • … • … • … It will be called at n Hz in modes A and B. Its interface is described in Table yyy.	

Tab. III-2: Example requirements with Classification

Further: Is it sufficient on software level to test whether the packet is passed on to the network driver, or do we need to monitor the actual hardware connection? What about the case when both are connected to a bus which also connects to other modules? Then we need to check addressing. How are packets addressed on the respective bus? Maybe the bus is marking message types instead of destinations, similar to the CAN-bus?

We see that even if we were able to actually parse the sentence structure in a semantically meaningful manner, a lot of issues regarding the resolution of more specific information remain. The way the documents are structured now the expectation that this issue can be solved reliably by natural language parsing seems unreasonable.

2) *Low-level Function Specification*

Let us instead have a look at one of the function specifications found within the SRD. Such function specifications do not merely specify functionality, but instead concrete low-level subprograms, i.e. implementation details. It is somewhat odd to find these at the SRD level, as the Software *Requirements* Document would normally be expected to specify the problem space in terms of requirements instead of the solution in terms of an actual implementation. However, we will – again – forget this detail for a moment.

Now let us consider one of these function specifications:

“This function process_GPS_data processes the raw GPS data. Its interface is summarised in Table xxx.”

The table referred to would then contain parameter specifications, listing for each parameter

- the parameter name,
- the parameter type,
- the parameter direction (input, output or both), and
- a description of the purpose of the parameter.

The three first elements of the description represent the information typically specified for the parameters in the code. The name may even conform to typical restrictions for identifiers (no whitespace, starting with a letter, etc.) and the type might even be a formalized type name.

This could then be followed by one or more individual requirements describing steps of the function implementation in pseudo code or prose, possibly switching from one to the other and back.

Depending on the quality of the prose, such specifications should be convertible to so-called reference implementations. These implementations would reproduce the expected behaviour of the algorithm to be implemented, and therefore would allow to explicitly produce the expected output by simply supplying them with the designated inputs.

One may of course wonder about the meaningfulness of such an approach. After all, when one can produce the reference implementation from the specification, why not use the reference implementation directly as the implementation in the target system?

There may be several reasons why the reference implementation may not be suited for use as an actual implementation.

The reference implementation may depend on the availability of features – such as libraries or calculation structures – that are not available on the target. For example, the reference implementation may be based on floating point arithmetic, while the target does not provide this capability and actual calculations on the target need to be done in fixed point arithmetic instead.

Further, the way the reference implementation is specified may lead to a high computational time or memory use requirements which may not be feasible on the target or within the given runtime environment. Therefore, the actual implementation may have to be a heavily optimised version of the reference implementation.

And as a very basic modification that is necessary in almost all cases, the pseudo code has to be translated into the actual programming language used for implementation, which may lead to deviations between the implementation and the specification, and thereby imply the need for verification – e.g., by testing.

For all of these issues, the function specification is a very useful *reference oracle*.

3) Monitoring Requirement

Sometimes, even requirements other than function specifications can be testable in an automatic manner, typically when they contain pseudo code as well. One example for such a requirement was lifted from the monitoring requirements class:

"At 8Hz set the validity flags for the GPS Validity monitors to true if the corresponding GPS unit is active and the AOCS mode is Mode1, Mode2 or Mode3.

```
If the GPS unit X is active
            and the AOCS mode is Mode1, Mode2
            or Mode3
  GPS_X_validity_monitor = true.
else
  GPS_X_validity_monitor = false.
end"
```

Here, the second portion of the requirement essentially repeats the functional part of the prose at the beginning of the requirement in pseudo code.

This of course implies the question about which part of the requirement is the normative one in case they should be conflicting – which is actually the case here.

Consider the case that the GPS unit is inactive. The prose does not specify any action for that case. It only says that the validity monitor flags should be set to true if the GPS unit is active (and some other conditions apply).

The pseudo code however explicitly specifies that in this case the validity flag would have to be set to false. Thereby it does specify a more strict requirement than the prose itself.

Again, the pseudo code could be used as a reference oracle. Note that by expressing the requirement in pseudo-code we can also provide a formal coverage criterion, such as MC/DC. The results of this approach can be seen in Tab. III-3.

Test Case	Input			Output
	Status	Mode	Validity	Validity
1	active	Mode1	false	true
2	active	Mode2	false	true
3	active	Mode3	false	true
4	active	Mode4	true	false
5	inactive	Mode2	true	false

Tab. III-3: Test Cases for the Monitoring Example

Note that here the validity – the variable that is to be set – is also part of the input or rather the precondition. This way one can actually determine whether the value was modified by the function under test or whether it had already been set to that value before. This could theoretically be derived from the interpretation of the pseudo code.

While we are considering what the function under test would be doing, we might notice that that function is never mentioned. This is not surprising, as this is a functional, but not a function requirement. How these monitoring checks are implemented is left to the designer – as it should be.

4) Non-Functional Aspects

Another aspect of the requirement considered in 3) is not covered by the pseudo code, but rather by the prose: The update frequency.

The requirement can be understood in a way that specifies a minimum update frequency of (at least) 8Hz. To test that, one could establish a situation under which the validity flag would have to be changed to true, wait for 125ms and then check whether the validity check has changed.

However, formally the requirement does not specify that the system must react to a change in the situation, but rather should update the validity flag if – at the time of the update – the situation is such that the validity flag would have to be set to true.

Let us consider this case: During the waiting period of 125ms the GPS unit would become active, the AOCS mode would change to Mode1, and immediately afterwards would change back to Mode4. According to a possible interpretation of the requirement, the system would not need to detect changes at frequencies higher than 8Hz, so a conforming implementation would not be required to actually change the validity flag in that situation, but it would be allowed to, should the execution of the respective code happen during the short timeframe in which Mode1 is active.

Vice versa, a short change from Mode4 to Mode1 and back would not necessarily require the system to change the value of the validity flag to true.

The actual outcome depends on when the respective event happens relative to the execution of the update procedure. This is indicative of a race condition. Invisible Information

This whole issue is resolved if one interprets the "8Hz" specified in the requirement not as an explicit real-time requirement, but rather as an information to the designer to plan sufficient computing resources for the update to occur at a frequency of at least 8Hz. In that case, the test engineer can rest calmly, because the non-functional requirement does not have to be verified by test, but can be verified by design and code review, e.g. of the scheduling tables.

In discussion it was revealed that this indeed was the intention in this case. Unfortunately, such hidden information would not be available to a naive reader of the specification, as any machine performing natural language interpretation would be.

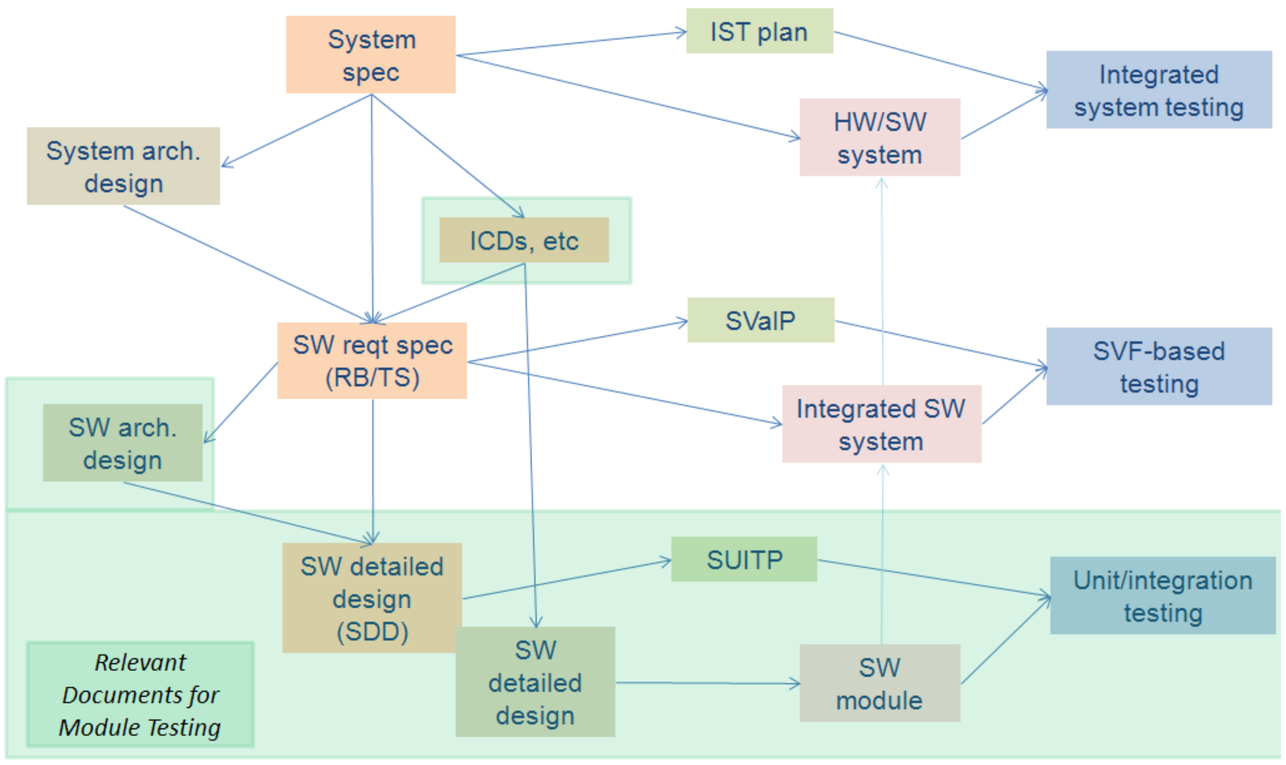

Fig. III-1: Typical Document Flow for Software

C. *Design Analysis*

While the Function requirements do have quite a direct link to the source code due to the fact that they define specific functions, a link from the other requirements to the source code-level is less apparent.

This is not surprising, as the SRD shall consider the software subsystem as an integrated system to be tested as such, and only specify what is to be implemented, not how it is to be done. The specification of the latter would be expected in the SDD. Therefore, the SDD was also analysed.

In case of the documents at hand, the SDD was more descriptive than prescriptive in nature: It described the design of the software and was not written with the intention of it being a requirements document. As a consequence, no atomic requirements describing the functionality of individual functions could be identified.

We considered whether the information from the SRD and the SDD could be joined via the information on tracing between both documents. However, the finest level of tracing found is at the level of individual code objects and functions back to the SRD requirements they implement.

Information on what physical inputs to these functions would correspond to triggers for the functionality described in the respective requirements are therefore not readily available at the level necessary for automated requirements-based testing at source-code-level. The only exception to this are the Function requirements already mentioned.

D. *Documents in the Process: The Larger Picture*

Looking at the larger picture we can confirm the suspicion that indeed according to current processes the SRD as a document is at the wrong level of detail for most attempts at requirements-based unit testing.

Let us first take a step back and try to get a more general, conceptual understanding of the process of breaking down requirements in any engineering project, not just software engineering. At some top level, there need to be general requirements on what the system that is to be built is expected to achieve. These requirements essentially state the problem to be solved.

In a first step, one would try to determine a most general solution and describe it from a very top-level perspective. So while the top-level requirements would describe, *what* is to be done, the design would describe, *how* it shall be done.

However, as the top-level design is only concerned with a top-level perspective, it will have to be broken down into solution components. For a satellite mission these components could be the space, the operations and the user segment. For a satellite they could be the satellite bus and the payload subsystem.

Here, the design again poses requirements – not to the system as a whole, but rather to the subcomponents. This means that a design is also a requirements specification, only for the levels of detail below it.

Following this principle, the SDD should be both a requirements document and a design document. It describes a design of the software element of the whole system, and at the same time it imposes requirements onto that software element.

In

Fig. III-1 we see a typical document flow for a spacecraft development project according to the ECSS standards.

Within the figure we find the typical refinement structure over the different levels of detail, starting from the system level, over the software element level to the individual software modules and the actual implementation.

The root of all artefacts for the software is the Software Requirements Document (SRD) – shown as Software Requirements Specification in the figure – together with any relevant Interface Control Documents (ICDs).

The SRD is primarily aimed at describing the problem space – although evidently, it may contain function and other requirements that actually describe the solution already quite in detail. Its connection to the actual software implementation is only indirect, incorporating the software architecture design and the Software Design Document (SDD) in its path.

Conceptually, the SDD shall describe the solution, answering to the problem description in the SRD, and the software architecture design describes the breakdown of the implementation of the software requirements into software modules, as well as their interfaces.

What we also see from the figure is that the Software Unit and Integration Test Plan (SUITP) is not directly derived from the SRD, but rather from the SDD. This is not surprising, as the Software *Unit* and Integration Test Plan needs to refer to software units, which are defined in the SDD, but not in any higher-level document.

According to this logic, the proper document to search for requirements for unit level source-code-based tests would be the SDD.

However, our analysis shows that the SDD currently seems not to be considered to be a requirements document, and therefore is not structured as one. Rather, it contains a description of the software design, without explicitly specifying functional or non-functional requirements imposed on the individual components of the software.

Thus, no requirements to test against can be found in the SDD, but while the SRD would contain requirements, they are not generally expressed at a level suitable for use in requirements-based unit testing. The only exception to this are the Function requirements found in the SRD.

E. *Results*

The analysis of requirements for the specific application highlights the difficulty of writing requirements such that they express the intended functionality of the software in a way that can be verified, e.g. by test. Many requirements were not written with pass/fail criteria in mind, but the emphasis is rather on communicating to a software designer what is required in the software design.

The results also show that there is a potential for formalisation based on common schemes. A large portion of the requirements analysed so far can be sorted into one of a few requirement categories, most of which are functional in nature and could be expressed using temporal logic formalisms.

The Functions requirements seem to be those that are most suited for source-code-level testing, as they effectively specify a reference implementation for these functions which could be used as an oracle: Both the actual implementation and the reference implementation extracted from the requirements are run on the same inputs and their results are compared to each other. The test passes if and only if they match. Interestingly, these requirements also make up the largest individual category in our sample.

IV. Options for Information Retrieval

The analysis of requirements so far seems to indicate that the information required for automatic requirements-based testing is not provided in the available documents, or at least is not provided in a concise manner. In addition, the documents are usually written in natural language, the interpretation of which by software is difficult.

Assuming that the information is available in the documents in principle – which is not necessarily the case (c.f. Sect. 0 –, there are several options for going forward. All of these options come with additional effort and cost, which must be weighed against possible improvements in quality and cost savings due to the use of automatic testing, and also against possible positive effects intrinsic to the respective method.

In the short-term perspective, only manual extraction of information seems applicable. This likely comes with significant additional effort.

Automatic extraction of information from natural language documents may be possible, but comes with its own issues impacting the complexity and correctness of interpretation. Natural language contains many constructs that are simple for humans to decipher – such as implicit references – but are difficult for software to resolve.

In a mid- to long-term perspective, more rigorous structuring of SRDs and SDDs may lead to an improvement of the situation. Again, this is connected to additional cost for the authoring of these documents.

Another possibility is the formalisation of requirements and design documents. The results from the MATTS activity[1] hint at possible quality gains from the effort of formalisation alone, even without the added gain from test automation, but the size of the gains is still unknown. Also to be considered is the decreased comprehensibility of formal notation to the human reader, which may lead to misunderstandings.

In general, any additional effort spent in authoring requirements and design documents would fall into the project phases before PDR and CDR, respectively. But benefits could come from fewer issues during qualification and improved product quality.

A. *An Example*

Let us once again consider the requirement from Sect. III.B.3). How could we make that more clear to work with when automatically generating test data?

First of all, we would need to get rid of the duplication in prose. Second, we would split it into its functional and non-functional aspects. Third, we would introduce some more information into the pseudo code.

Keep in mind that the syntax is practically irrelevant – except for the single requirement that it must be a formal syntax, a syntax that can be automatically parsed. What is important is the semantic content and the fact that it can be automatically extracted.

Let us see how that could look:

```
“The GPS_monitor_func shall
do
    If GPS_unit_X.status == active and AOCS_mode in
        (Mode1, Mode2, Mode3) then
            GPS_X_validity_monitor = true
    else
            GPS_X_validity_monitor = false
    end
  enddo”
```

The 8Hz repetition rate – the non-functional aspect – is not present in this description. Depending on the intent of this aspect of the requirement one could of course design a way of specifying it. However, our focus shall be on the functional aspects.

Note how the pseudo code gives the function – both in terms of a procedure but also in terms of functionality – a name: GPS_monitor_func. This name would not necessarily be the name in the final source code, but it should be a unique identifier which could be used to map the function or functions in the code onto the requirement.

Similarily, GPS_X_validity_monitor, GPS_unit_X.status and AOCS_mode are unique features of the specification, which could be mapped to the respective elements of the implementation.

V. Conclusions and Future Work

The analysis of available documents – SRD and SDD – indicates that they only contain part of the information required for automatic requirements-based testing, even if specific issues such as the form of representation of this information is ignored. Some of the requirements are very detailed and actually can be used for automatic testing, but they cover only a small part of the actual functionality described in the SRD.

There are several options for changing that situation in the future, but they all come at a – yet unknown – cost, that has to be balanced against the – also yet unknown – benefits.

In preparation for a cost-benefit-analysis, a simple exercise shall be executed during the on-going FASTII-activity, considering a small subset of requirements from a reference project and manually deriving oracles from these which can be used in the already established and to-be-improved FAST[2] process to introduce requirements-based information into the automated source-code-based testing approach.

Acknowledgment

The results presented above are an intermediate outcome of the FASTII activity, ESA Contract No. 4000116014/16/NL/AF/as.

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