A Case Study on Automated Source-Code-Based TestingMethods

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

e-mail: Ralf.Gerlich@bsse.biz,
Rainer.Gerlich@bsse.biz

Kenneth Kvinnesland, Bengt Solheimdal Johansen
DetNorskeVeritas AS
Høvik,Norway
e-mail: Kenneth.Kvinnesland@dnv.com,
Bengt.Solheimdal.Johansen@dnv.com

Marek Prochazka
EuropeanSpaceAgency (ESA/ESTEC)
Noordwijk, The Netherlands
e-mail: Marek.Prochazka@esa.int

***Abstract*—We present results of a case study on a test generation approach called Flow-optimized Automated Source-code-based unit Testing (FAST) which generates test stimuli from information available in the source code, in particular taken from the detailed software interfaces. This allows automation of a significant part of testing, ranging from the test stimuli generation to the generation of the test report. A huge number of stimuli can be generated exploring the behaviour of the software under test under nominal and non-nominal conditions. Symptoms like timeouts, unexpected termination, run-time exceptions, out-of-range conditions and missing coverage are applied for defect detection. The goal of this study was to evaluate the FAST process in context of a real spacecraft flight software application and to get a feedback on its scalability regarding larger applications, its sensitivity on detecting defects in the code, the achievable test coverage, its compliance with software standards and potential limitations. We also consider the impact of coding style on suitability for automated testing. The results confirm that the approach (1) provides acceptable code coverage results without requiring manual intervention for test preparation and execution, (2) raises the probability of activation of exotic fault conditions, (3) may provide hints on locations in the code where robustness needs to be verified, and (4) identifies defects not found before by static analysis and intensive testing**

Keywords-automatic test data generation, unit testing, robustness testing, fault injection, test coverage, testability

I. INTRODUCTION

Software testing is a method for dynamic analysis of software used to determine whether the software indeed fulfils its requirements. Unit testing is one form of software testing which focuses on isolated testing of software units, such as individual modules or functions – in the sense of subroutines.

While the automatic execution of unit test suites and the automatic evaluation of results using test tools such as JUnit[1] and its other language-specific variants are widespread, preparation of test cases is still mostly performed manually. Identification of test cases may be performed targeting one or more types of code coverage or requirements coverage. Of relevance for the European space industry are the standards DO-178, versions B and C**Fehler! Verweisquelle konnte nicht gefunden werden.**, and ECSS-Q80, versions B and C**Fehler! Verweisquelle konnte nicht gefunden werden.**, all of which suggest both approaches, code and requirements coverage.

We present results from a case study applying an alternative approach to software unit testing, complementary to the traditional, manual approach: test stimuli are derived automatically from computer-readable information, mainly consisting of the source code[4]. This allows reaching a high level of test coverage quickly, irrespective of the criticality levels, especially for software for which unit testing was not previously applied or required.

For the software under test in this exercise, a part of the flight software of a European Earth observation satellite was provided by the European Space Agency (ESA). Only the source code of the software was used for the case study, but no specification or design documents. Such documents rarely contain information rigorously expressed in a computer-readable form useful for the generation of test data.

A. Structure of the Paper

After a general description of the approach and related work, we present in Chapter II some of the findings as well as coverage data, followed by a general discussion of their implications in Chapter III. In Chapter IV we derive recommendations on how specific issues, constraining this type of test automation, can be avoided by appropriate planning in software development projects that aim to apply the process. Finally we conclude and give an outlook on future developments.

B. Description of the Approach

The approach is based on the knowledge of the source code of the Software Under Test (SUT) and an extension of the process flow towards automation and is thus called Flow-optimised Automated Source-code-based Testing (FAST). The tool used is called DCRTT (Dynamic C Random Test Tool), indicating the origin of the tool from random testing – and is developed and maintained by BSSE. Another variant named DARTT – a predecessor of DCRTT – is available for Ada.

1) The Process Flow

In Figure 1 the flow graph of the traditional test approach is shown. In this approach, the specification is established first, followed by coding and manual test case preparation as parallel activities. The test cases already include the expected output of the Functions Under Test (FUT). Once these two activities are finished, the manually identified test cases are executed, and their results are evaluated by comparing the observed to the expected outputs. If the test shows that critical defects are still present – indicated by one or more test cases not being passed – corrective actions on the code and possibly the specification are applied and tests are repeated.

In contrast, Figure 2 depicts the flow graph of FAST applying automated source-code-based testing. In this case massive stimulation is performed by generating a large number of stimuli from the information found in the source code interfaces, with the intent to check robustness and to identify test case candidates. The latter is based on information recorded by the test environment – including coverage, run-time exceptions and timeouts observed. The selected candidates already include the output observed during execution. They are provided in the code of automatically generated test drivers and may achieve the same level of coverage as the overall set of stimuli does, depending on the configuration options used, when executing the test drivers.

The test case candidates are upgraded to test cases by manually reviewing the recorded output and comparing it to what is expected according to the specification. If one or more results do not conform to the specification, corrective actions must be taken as for the classical approach. Once the test case candidates have been upgraded to test cases, they can be reused at any time for regression testing.

The FAST approach thus reverses the order in which test cases are established and executed. In the classical approach, each test case is established first and executed later, while in the FAST approach, the software has already been executed with the test stimulus as input when the test case candidate is upgraded to a test case by comparison to the specification.

This approach avoids the issue that specifications are typically not formal and computer-readable and thus cannot be used for automatic test data generation without additional effort. Instead, the code is used as a formal description of the interfaces that must be stimulated.

Although not currently implemented, the verification of test case candidates can of course be automated should formal oracles be available.

The test case candidates may not be equivalent to those normally derived from the specification, therefore test cases may have to be added manually to obtain full requirements coverage.

2) Stimuli Generation

Inputs can be selected either randomly or based on a lattice spawning the input space of the individual functions as defined by their function prototypes (interfaces, black-box approach).

In case of lattice-based test data generation, the requested number of stimuli per function is distributed over the parameters. For example, if 300 samples are requested and the function has 4 integer parameters, 5 samples per parameter are used based on the 4^{th} root of 300 rounded towards positive infinity, leading to 4^5=1024 samples in total. For each parameter type some typical samples are added, including values such as 0 or NULL for pointers. Thus, the total number of stimuli actually generated may be considerably larger than the number of stimuli requested. When the resulting number of stimuli exceeds a given limit, e.g. 1,000,000 per function, the test mode is automatically switched to random testing to limit execution time.

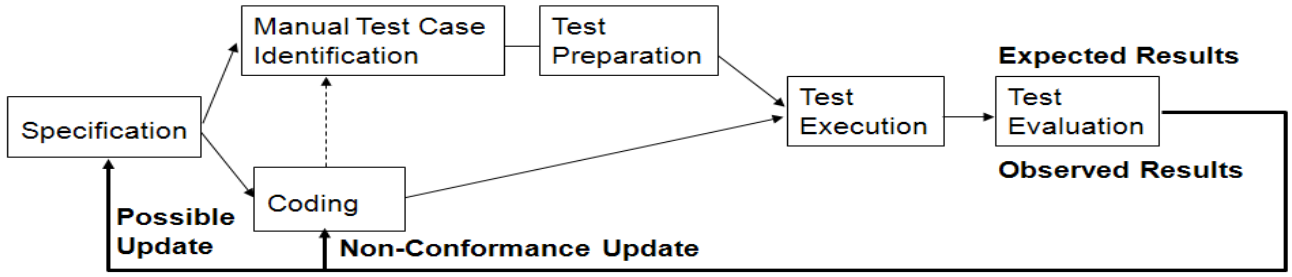

Figure 1 Flow Graph of the traditional approach to software unit testing

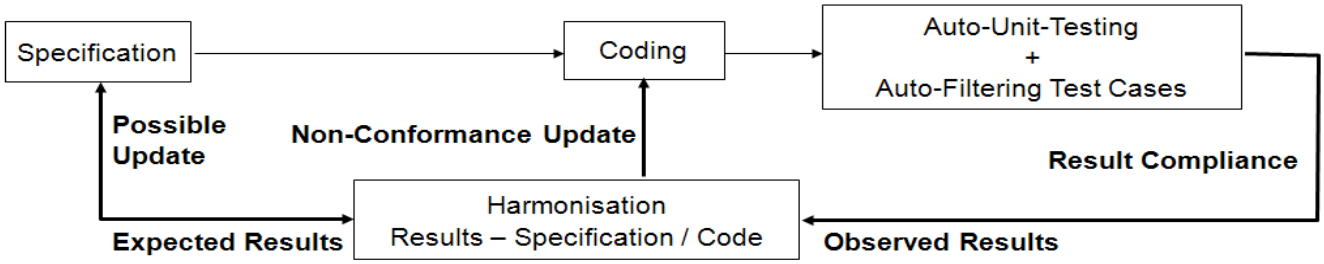

Figure 2 Flow Graph of the approach using automatic source-code-based test data generation

In case of random testing the requested number of random stimuli plus the set of typical values is injected.

Additional inputs may be determined heuristically by using constants found in the code of the function or complementing data observed during execution (white box approach). Consequently, FAST applies a grey box approach as a mix of the two.

Besides explicitly declared function parameters, the result of a function may also depend on the value of global variables. Therefore, such variables may also have to be stimulated to sufficiently exercise the functionality of the unit. As stimulation of all visible global variables for each function is not feasible in most cases, a subset of global variables applicable to the current FUT has to be identified.

A global variable is only considered for stimulation if it is not declared constant and is used inside the FUT. The set of variables used in the FUT is determined automatically by a conservative heuristic analysis.

However, if deemed necessary and feasible by the user, this analysis can be deactivated. In this case, all global variables are stimulated, except for those declared constant.

With only basic configuration data and information automatically extracted from the source code, a test environment is established that records generated input, delivered output, execution time, run-time exceptions and timeouts for each individual test stimulus executed. The source code is further instrumented automatically to record structural coverage and – optionally – to perform data range monitoring for variables used in the software. Structural coverage criteria supported are block coverage – a criterion similar to statement coverage based on continuous blocks of statements – and modified condition/decision coverage (MC/DC) **Fehler! Verweisquelle konnte nicht gefunden werden.**on decision statements (if, do-while, while, and for). Coverage measurement may be performed during integration testing as well.

Each of the selected test case candidates consists of the input provided to the function as well as the observed output. In order to form a proper test case consisting of input and expected output, the observed output must be confirmed to be correct by reference to the software design or software requirements specification. The candidate test cases are provided in a format suitable for export to other test management software.

3) Instrumentation and Stubbing

Further, the source code may be optionally modified automatically to facilitate fault injection. One variant of fault injection involves forcing specific functions to return values typically indicating errors. An example of such a modification could be to force a memory allocation function – such as *malloc* from the C standard library – to return *NULL*, indicating insufficient free memory.

Such a condition is normally very difficult to emulate in actual unit testing and automatic instrumentation replaces calls to *malloc* by calls to an intermediate function that randomly or deterministically decides on each call whether to return *NULL* or actually call *malloc* and pass on the return value from that call.

Fault injection may also include modification of global variables which are declared to be constant. This is a form of parameter sensitivity analysis and may highlight possible implicit data dependencies and thus issues which may not be critical in the respective current version of the software but may become active defects during maintenance later on or as a matter of fault propagation.

Fault injection has turned out to be an efficient means for raising probability of fault occurrence for faults otherwise occurring only sporadically.

The use of massive stimulation overcomes the limited expressiveness of structural coverage criteria as they are currently in standard use in industry [5]. By first executing the software using test stimuli in massive numbers – effectively oversampling the domain normally exercised by manual unit tests – and afterwards deciding on possible test case candidates, not only additional information from meta-analysis is made available, but also the decision on the candidates is made on the basis of actual information about their usefulness instead of engineer's intuition.

Such massive stimulation is typically not possible on the target of an embedded system, and the instrumentation required imposes an additional significant resource burden. To overcome this issue, the code is ported to a host platform with sufficient computing and storage resources. In this context, porting mainly means modifying the software such that it can be compiled and linked for running on the host platform. Of course this limits representativity of the results obtained during stimulation, which is why the tool also provides support for automatic re-execution on the target platform for the identified subset of test cases. Further, peripheral hardware is typically not available on the host platform and if it is, the interfaces used in most cases are very different from those on the target platform, so testing is limited to the functions concerning the application logic. However, this typically represents the major part of the software even in embedded systems.

To support functions not available on the host platform, to represent missing hardware interfaces or interfaces to the target operating system, or to substitute functions not yet (completely) coded, stubs may be generated for such functions with randomly or grid-based output derived from the specified output domain. More than 300 symbols, functions and data, have been generated to complement the application source code for the application under test.

Coverage in the automated approach becomes a property of the software and not only a property of the test suite. For example, low or missing coverage on a specific portion of the code may be contrary to expectation and thus hint at a defect[4].

C. Software under Test

The software under test in this exercise was the application software of a European satellite. It was classified as mission-critical meaning that anomalous behaviour would cause or contribute to a failure of the satellite system resulting in permanent and/or non-recoverable loss of the satellite's capability to perform its planned mission.

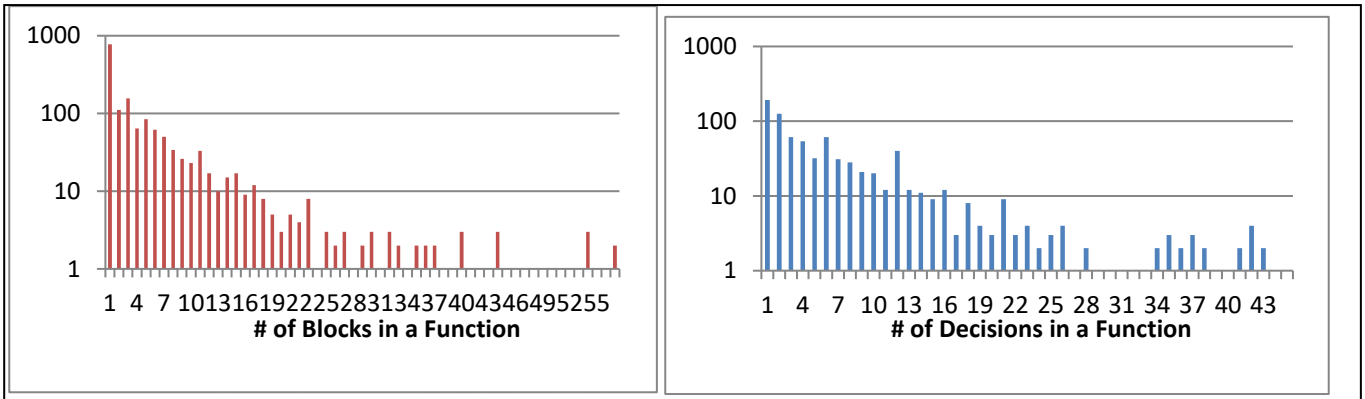

Figure 3Histograms of the distribution of block and condition counts over the functions (Y-axis: #functions+1)

The version used in the project was a near-final version that had already gone through a number of thorough test campaigns, design and code reviews and analyses, as prescribed by the applicable European space industry standards ECSS Q-80**Fehler! Verweisquelle konnte nicht gefunden werden.**and ECSS E-40[6], both in version B, and in addition Independent Software Verification and Validation (ISVV) [7]was performed due to the software criticality.

In a first step a mission-critical subset of the application software was processed. This subset consists of 1530 individual C functions over about 64 KLOC lines of code (not counting comments and blank lines). The total number of blocks is 6434, the total number of conditions 3925

In a second step, the full set of software functions was tested. This set amounts to 3375 functions, 166 KLOC and 14799 blocks and tbd conditions. This paper will focus on findings and results derived from the subset of 1530 mission-critical functions.

Slightly more than half of the functions – namely 773 – consist of a single block without any decisions, representing 12% of the blocks. In contrast, the maximum number of blocks per function found was 104 (in a single switch statement) and the highest number of decisions per function was 43, occurring in one function each.

Figure 3shows the distribution of block and decision counts over the functions. Note that the x-axis has been cut for better readability in the histogram on blocks. Only very few functions have higher block or decision counts than shown and would not be visible in the graph anyway.

A software support package could not be delivered due to matters of intellectual property rights. Therefore the missing functions were substituted by stubs as described above in order to get linkable object code.

Note that no excerpts from actual source code are given in the paper. Instead wherever we present example code, the snippets are intended to show the basic idea behind the logic present in the code.

D. Related Work

Random testing is a method for systematically selecting test data using a specific probability distribution [8][9]. If the probability distribution is suitable, estimates on the remaining single-invocation fault probability can be established for a specific distribution over the inputs under the conditions to be considered, e.g. general operational conditions.

Several refinements have been proposed to increase the efficiency of the approach. One such refinement is Adaptive Random Testing (ART)**Fehler! Verweisquelle konnte nicht gefunden werden.**. In ART, selection of a new test stimulus consists of first establishing a set of candidate stimuli from the input set. From these candidates the element that maximizes the distance from the previously selected test stimuli is used for the next test case. The distance measure can depend on the application, but typically Euclidian distance is applied. It has been proven that, given no information about the localisation of faults in the input space, the expected number of test cases required to find the first defect (the so-called F-measure) is minimal with this approach, assuming specific patterns of fault occurrence over the input space derived from typical mistakes in decisions and calculations.

Lattice-based testing [11] is a corner case of ART. Given the extents of the input space and the maximum number of sample points in that space, a grid with regularly spaced grid points maximises the distance of the individual points. In lattice-based testing, the inputs are selected from this grid, either iterating over all the points, or instead selecting random test stimuli from the set of grid intersection points. In DCRTT, the first of the two alternatives applied. This strategy has been shown to improve coverage of conditions such as direct comparisons, which are otherwise difficult to cover with strict random testing.

Use of generic criteria to detect possible defects is typically used for fuzzing [12], where software is massively stimulated with random or specifically prepared inputs in order to elicit run-time exceptions or other abnormal behaviour. The main application of fuzzing is evaluation of security and detection of possible attack vectors, but of course it can also be applied for general robustness testing.

II. FINDINGS

We executed multiple test runs for the whole software, varying configuration parameters such as the number of stimuli to be generated per function and the activation of various forms of fault injection. Depending on the

configuration, between 1 million and 27 million test stimuli were injected in total for the 1530 FUTs.

A. Analysis of Anomalies

1) Results

Depending on the configuration, between 50 and 233 of the stimulated 1530 functions showed anomalies. In this exercise, only run-time exceptions, timeouts and out-of-range conditions of indices were considered as anomalies. Data range monitoring was not used, so range violations outside index expressions were not checked.

Some of these anomalies were examined for their cause, leading to 52 findings in total, of which 50 were found to be non-critical. For two findings the project implemented corrective maintenance. Overall the software package was found to be mature, which is not unexpected given that only a near-final version was available that had already gone through quite rigorous code inspection, static analysis and classical software tests.

Most of the other anomalies could be attributed to the fact that the stimulus generator cannot know a-priori about function pre-conditions and thus cannot honour them. The anomalies occurring due to such dependency on pre-conditions do not necessarily indicate defects in the overall software as long as it is shown that the pre-condition is satisfied at all times when the function is called within the whole software.

Consequently, in the course of analysis of such an anomaly it is important to check whether fulfilment of pre-conditions has already been proven, and if not, to provide a proof.

To avoid anomalies due to conditions non-representative for the environment during the operations phase, constraints on function parameter and global data is typically provided in a generic manner.

In context of out-of-range monitoring of indices such constraints are derived automatically by the tool from information on the upper limits of arrays found in the source code.

From a general perspective, occurrence of such anomalies indicates that the software is not protected against such critical values and that a proof is necessary to show that such conditions may not occur in the context of the whole application. When the proof is already available or when it is performed due to the finding, the anomaly can be ignored or avoided by provision of a constraint.

2) Analysis Approach

Identification of the location of an anomaly is guided by information provided in the automatically generated report. Anomalies are classified into the following top-level categories:

- Run-time exception as flagged by the operating system
  e.g. access violation,
- Corrupted memory as identified by DCRTT
- Out-of-range conditions as identified by DCRTT
- Timeout conditions as identified by DCRTT
  e.g. due to a deadlock, a livelock, or a crash (corruption of essential memory)

The information required to efficiently identify the source of an anomaly is compressed and filtered. Anomalies are reported on the level of an index expression in case of an out-of-range anomaly or on block level or condition level for other anomalies depending on where the anomaly was raised. If a low-level function causes an anomaly for several FUTs, the single source is identified, so that the relevant function and block, condition or index expression can directly be entered without any need to search top-down starting at every FUT. Tab. 1 shows examples on how the anomalies are reported.

Anomaly Type	File	Function	Description				Line	Block Id	Cond Id	Test Id	#Occ
			Index Id	Dimension Index	Min/max observed	Index Expr.					
OutOfRangeLow	File1.c	Func1.c	2523	0	min=-1<0	idx1	297	1	n/a	232	2284
										236	2284
OutOfRangeLow	File1.c	Func1.c	2524	0	min=-1<0	idx2-1	298	1	n/a	232	2284
OutOfRangeHigh	File2.c	Func2.c	2844	2	max=965>27	idx3	239	1	n/a	281	2339
										275	2339
Excp	file3.c	func3.c					876	2	n/a	266	768
										267	1536
CorruptedMem	File4.c	Func4.c					639	5	n/a	314	2
DeadLock	File5.c	Func5.c					494	25	n/a	378	1

Tab. 1: Examples for Anomaly Reporting

The block id points to the block in the function where the anomaly was observed, the condition id is the id of the

condition in a logical expression, and test id is the id of the FUT which was tested when the anomaly was raised. "#Occ" is the number of occurrences of the anomaly.
For out-of-range conditions more information is provided describing the id of the index expression assigned by DCRTT, the number of the index in a list of dimensions, the minimum or maximum of the out-of-range value and the index name.
As an out-of-range condition may occur multiple times in a function – whenever the same index is used for the same dimension – the report further compresses the information to a list of critical indices of a function for which out-of-range conditions were observed during test of FUTs as Figure 4 shows.

Expr	Type	Violation	Function	File
idx1	low	min= -1 < 0	func1	file1.c
idx2-1	low	min= -1 < 0	func1	file1.c
idx3	high	max=965 > 27	func2	file2.c

Figure 4: Compressed List of Critical Indices

3) Examples of Anomalies Found

In Figure 5 an example of such a pre-condition involving the use of *memcpy* is shown. The ground station may send commands to the satellite, so-called telecommands (TC). When such a TC arrives on the satellite, the data contained in the TC packet is typically buffered and the TC is placed in a queue to be handled later.

For that, part of the TC data needs to be copied to the queue using *memcpy*. The number of bytes to be copied is derived from the length field found in the TC packet itself. In the example, this length field is found at the address *cmdData+offset*.

The function that does the queuing and copying does not check the validity of the data in the length field. Thus the length retrieved from the packet may be out of range, leading to a possible memory corruption.

As the stimulator forces using the value 0 for integer parameters at least once, there was one stimulus in which the length was 0. The calculation *len-3* thus led to the result *-3*. However, as the third parameter to *memcpy* is an unsigned integer, this is actually interpreted as $2^{32}-3$. Thus in that case the block passed to *memcpy* was equivalent to almost all of the available address space. Because the host system has a memory management unit that performs memory access checking, the copying process soon raised a memory access exception, thus highlighting the issue.

It is obvious that if such an input was received by the on-board software under test it would have grave consequences. If mechanisms for fault detection, isolation and recovery (FDIR) are implemented on the same processor in the same address space, they would be rendered useless. Data collected in RAM would be lost and the only chance for recovery would be a restart of the application software.

However, considering the context in which the function is called within the application software it can be shown that the pre-condition is indeed fulfilled: every telecommand is first thoroughly checked to be valid, before further processing its data. It must be ensured that none of the invariants is violated later by code modification during the software maintenance.

Indeed the function is protected by a mechanism similar to the one depicted in Figure 6. A table containing – amongst other information – the allowed range of values for the *len* field depending on the type of TC contained in the packet. The minimum length was shown to be at least 3 for all TC, ensuring that negative values cannot occur for the number of bytes passed to the *memcpy*.

There were several anomalies which were raised due to the existence of basic, not explicitly expressed pre-conditions. A recurring pre-condition was the correlation of two parameters, one being a pointer to a buffer and the other being expected to give the length of that buffer, as shown in Figure 7. The parameters may also be embedded in a common structure type, one member being the buffer pointer, the other being the length. This is a common pattern in C as arrays do not carry their length, different from other programming languages such as Ada, Pascal or Java.

```
char buffer[BUFFER_SIZE];
void bufferCmd(char* cmdData, unsigned int offset) {
  unsigned int len;
  memcpy(&len, cmdData+offset, sizeof(unsigned int));
  memcpy(buffer,
         cmdData+offset+sizeof(unsigned int),
         len-3);
}
```

Figure 5*memcpy* Example

In the basic FAST approach, the probability of selecting stimuli that violate this pre-condition is very high. Such parameter combinations invariably lead to run-time exceptions, often early in the function under test, so that further parts of the function cannot be exercised and receive no coverage. For this reason we needed to find a way to increase the probability of selecting inputs that fulfil the pre-condition. Because the pattern was so common in the SUT, the effort of making this constraint explicit by manually annotating all the functions using it was considered prohibitive for our case study.

However we found that there was a small set of parameter names used for such pairs, so we were able to introduce a generic pre-condition automatically correlating all parameter pairs with such names. As a consequence, coverage for the affected functions was increased considerably and the number of false-positive anomalies observed in them was reduced.

Consequently, in order to be able to use FAST efficiently, our recommendation is to use identical names for the same logical items to the extent possible.

B. Coverage Analysis

The coverage criteria considered by the tool are modified condition/decision coverage (MC/DC)**Fehler! Verweisquelle konnte nicht gefunden werden.** and so-called block-coverage.

To understand what block-coverage means, consider the instrumentation performed by the tool: At any point where a change in control flow may occur – such as in conditional statements or loops – checkpoint markers are inserted into the source code. Whenever execution traverses such a checkpoint marker, this is recorded. The markers are placed in such a way that from the sequence of traversal of these markers it is possible to derive which statements were executed, except when an exception breaks the control flow. In this case, the block is marked as being interrupted. Thus, full block coverage is indeed equivalent to full statement coverage: either an interruption is recorded or all the statements of a block were executed.

```
typedef struct TyCmdDescr{
  unsignedintminLen;
  unsignedintmaxLen;
} TyCmdDescr;

TyCmdDescr cmdDescr[]={
  {3,10},
  /* ... */
};
void recvCmd(char* cmdData, unsigned int offset,
        unsigned int entry) {
  unsigned int len;
  memcpy(&len,cmdData+offset,sizeof(unsigned int));
  if (len>=cmdDescr[entry].minLen &&
        len<=cmdDescr[entry].maxLen) {
          bufferCmd(cmdData,offset);
  }
}
```

Figure 6 Plausibility Check of Telecommands

```
void processBuffer(char* buffer, unsigned intlen) {
  unsigned int i;
  for (i=0;i<len;i++) {
        /* Do something with buffer[i] */
  }
}
```

Figure 7 Example showing Correlation of Buffer and Length

However, in statement coverage figures blocks with larger numbers of statements are overemphasised. In block coverage, only one block consisting of *n* statements without a branch is considered whereas in statement coverage *n* statements are contributing. Therefore, achieving a given percentage of block coverage is not necessarily the same as achieving the same percentage of statement coverage.

For MC/DC the full decision tables for each individual decision are considered. The decision tables do also consider short-circuit code, marking a condition as "don't care" if it does not contribute to the outcome of a decision under the circumstances represented by the table entry. An entry in the table is covered when the conditions of that decision takes the respective values associated with that entry, not considering those entries marked as "don't care". The MC/DC percentage is then calculated from the quotient of entries covered over the number of entries in total.

1) Coverage Results

In our case study, in the test runs executed with different test configurations with and without fault injection, block coverage ranges from 64.3% to 80.0%, and MC/DC was between 72.4% and 82.6%.

The number of automatically generated test case candidates was between 3,082 and 16,925, giving a mean number of test case candidates per function between 2 and 11. The higher figures reflect test case candidates related to exceptions and injected faults, which are recorded in addition to the ones considered for coverage, to make such exceptional events reproducible.

Coverage figures from different stimulation and fault injection modes are complementary to some degree. By merging test stimulus sets from different test configurations higher coverage figures could be achieved. The maximum values we got were 82.6% for block coverage and 86.6% for MC/DC.

2) Discussion of Dependencies

Coverage is subject to significant saturation even below 100%, as Figure 8 shows. For example, comparing two runs without fault injection, the first one injecting 1 million stimuli in total, the second one 20 million, we found that the execution time of the second was more than 5 times the execution time of the first one, but provided no significant increase in coverage.

Instead, other configuration options have much higher influence on coverage. For example, activating stimulation of global variables, fault injection by modification of return values or by ignoring const-qualifiers show a significant increase of coverage compared to the same configuration with these options disabled. Understanding of latter effect

requires detailed analysis of the code which has not been done so far.

Further analysis shows that, for example, fault injection leads to better coverage of error handling code that specifically checks for error codes being returned by called functions. This is not surprising, as error conditions such as insufficient memory do not normally occur without being explicitly provoked.

```
typedef enum {
   cmd1,cmd2,cmd3,cmd4
} TyCommand;

void executeCommand(TyCommand cmd) {
switch (cmd) {
case cmd1: /* ... */ break;
case cmd2: /* ... */ break;
case cmd3: /* ... */ break;
case cmd4: /* ... */ break;
default: /* error handling */ break;
   }
}
```

Figure 10Improving Coverage by using Enumeration Types

a) Optimisation Potential

However, achievement of coverage can also be difficult due to use of insufficiently constrained data types, which was observed at some places in the software. Consider the example in Figure 9. The input parameter is of type *unsigned int*, thus having 2^{32} possible values. Of these only 4 are used, all others lead to error handling. Stimulating this function randomly or even using a lattice-based approach will most probably result in many stimuli exercising the default-branch, while the other branches have a very low probability of being covered.

In case of a random stimulation with a uniform distribution, the probability of covering at least one of these with 10,000 stimuli is about $9.3*10^{-6}$.

If *unsigned char* was used for the parameter – still having 64 times more values than are actually used – the probability of hitting at least one of the non-default branches with only 300 stimuli would be slightly more than 99% in case of random testing and 100% for the lattice-based approach.

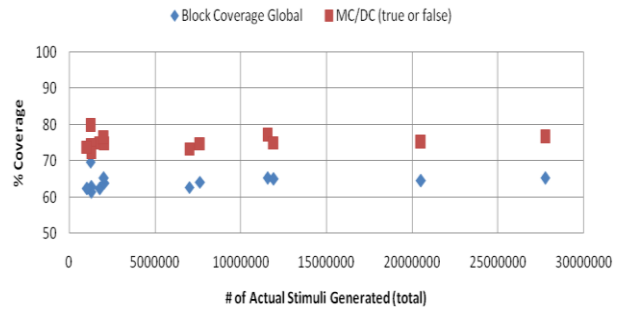

Figure 8 Achieved coverage vs. number of injected stimuli and various test modes

```
void executeCommand(unsigned int cmd) {

    switch (cmd) {
    case 1: /* ... */ break;
    case 2: /* ... */ break;
    case 3: /* ... */ break;
    case 4: /* ... */ break;
    default: /* error handling */ break;
    }
}
```

Figure 9Use of inappropriate types may prevent sufficient coverage

As a reminder: The runs requesting 10,000 stimuli per function took more than 3 days (feeding in about 27 million of stimuli in total), while runs requesting 300 stimuli per function were finished after a little more than 1 day.

Performance could be improved further if an enumeration type would be defined and used, as shown in Figure 10.

The presence of the enumeration type is a hint to the stimulus generator that these are the preferred values to be used. The generator would use only the enumeration literals defined if fault injection was deactivated, thereby covering the four branches with four stimuli in any case. Other values would be used with fault injection activated, thus covering the default branch.

Besides enhancing coverage when applying automated source-code-based test generation, the use of enumeration types also improves readability and maintainability of the source code and even for this reason alone can be considered a good engineering practice.

a) Reconsidering Block Coverage

As 773 functions have one block only one may ask how these simple functions impact the overall coverage figure and what is the average number of blocks reached in the more complex functions.

Considering a block coverage of 65% (an a reasonable figure observed for a single set of tests without merging contributions from several runs at different configurations) and the total number of blocks of 6434, the number of blocks actually covered is 4182 and the corresponding number of total blocks – not considering the single-block functions – is 6434-773=5661. This corresponds to an overall block coverage over all non-single-block functions of about 60,2%. The number of remaining functions is 1530-773=757. On the average these functions have 7.5 blocks of which 5.5 were covered.

Asking for the equivalent figures without counting the top-level block of a function we get 3425 blocks covered and 4904 blocks in total, corresponding to an overall block coverage of about 69,8%. This yields 6.5 blocks on the average and 4.5 actually covered in addition to the top-level block, equivalent to just subtracting 1 from the previous figures.

b) General Considerations

The weak dependency of achieved coverage on the number of injected stimuli (about 10% points increase of coverage while the number of injected stimuli varies by a factor of about 20 – from about 700 to about 18.000 stimuli per FUT) is a matter of the following constraints:

- Coverage in some cases is independent of the number of stimuli
  - 773 functions have one block only, full coverage is achieved with one stimulus already
  - 179 functions do not have parameters, they cannot be stimulated at all. They may depend on global data which should be stimulated as well in such a case. Remark: there may be an overlap between the 773 functions with one block and the 179 functions without input parameter.

- Parameters of enumeration type are always stimulated with the full number of literals. A variation of the number of requested stimuli does not change the number of injected stimuli for enumeration types.

- The range of a type is so large that a variation of about even 20 does not increase the probability to hit a certain value significantly.
  E.g. in case of a 32bit integer type the probability to hit a certain value is $2^{-25} \approx 10^{-8}$ at 128 stimuli and $2^{-18} \approx 10^{-6}$ at 16K stimuli.
  This strongly suggests careful selection of type ranges and application of constraints.
- Reachability conditions depend on certain complex environmental or spacecraft / application specific conditions, which cannot easily be met by selecting samples randomly or grid-based out of the input domain, e.g. in case of expected condition coincidences between data.

An example for smart selection of a type range are the following examples: for two functions each having one enumeration type parameter with 256 and 66 cases. In both cases 100% block coverage was achieved, while in another case where a 32-bit integer switch-variable was used only a few cases were and the default were hit (see Subsection a) above).

These results indicate that by an improved programming style a much better result can be achieved without any additional effort.

III. Discussion of Results

A. Defect Analysis

In general, the high number of stimuli injected – derived from the function interface – considerably increases the probability that a defect becomes active. Appropriate means of making the resulting error visible could therefore increase the probability of detecting defects. The higher sampling density better approaches the demands of standards like DO-178, versions B and C, and ECSS E-40-C for exercising boundary values and considering equivalence classes – fully independent of the criticality class of the software under test. Normally, more rigorous coverage criteria are only applied to software with higher criticality levels (such as our SUT), thus reducing the effort in case of classical manual testing for lower criticality levels, but at the same time diminishing the validity of test results for these cases.

In order for a defect to be activated, three conditions must be fulfilled: The statement containing the defect must be executed, the activation conditions for the defect must be met and the resulting fault must affect further computations, either by passing wrong values or by interrupting the control flow.

The first condition is ensured by statement coverage. However, neither statement nor decision or condition coverage, not even MC/DC generally ensures that the second condition is met.

For a simple example, consider a function that at some point is supposed to calculate n^2 but instead calculates 2^n– for both of which most computers provide single-instruction implementations. This function could be considered as covered completely by a test case that implies $n=2$, but as the desired and the actual output for that case are the same, the defect would not be detected.

Given an appropriate distribution of inputs – and in this case a way to verify the output – the chance of detection of the defect increases with the number of stimuli injected.

In order for a defect to be detected, it must be activated and the erroneous result must be detected either by an oracle [13] or via visual inspection by the engineer. An oracle which can be integrated into the automated test flow increases efficiency and is the best way to detect a defect.

The oracles used in this activity were the simple anomaly detectors, i.e. the watchdogs waiting for a timeout or an unexpected termination of the FUT or out-of-range conditions. These are pretty generic oracles which typically are not sufficient to detect functional defects, but they are sufficient for robustness testing and in some cases, run-time exceptions or timeouts occur due to functional defects.

The probability of detecting, e.g., an invalid memory access, may also be increased by platform diversification as it was performed in this case study as a matter of the FAST approach by porting to the host platform. For example, if the target platform does not provide hardware checking of memory accesses – as is the case for many microcontrollers and microprocessors used in embedded systems – testing the software on a host platform that supports such hardware checks may be advantageous, as the host platform can raise run-time exceptions as the result of an invalid memory access, while the target platform does not.

Defects of the kind where functionality was defined in the specification but not implemented in the code can only be detected if coverage of requirements by the test case candidates is evaluated. If requirements are not specified formally so that input data and/or output data can be automatically mapped onto requirements, determination of requirements coverage is bound to be a manual activity.

This task may be partially or completely automated given a fully or partially formal specification allowing determination of one requirement or a small set of requirements to which a single test case candidate applies to, just based on the input and output data associated with that candidate.

Similarly, if links between requirements and the code are established manually into the code – which often is the case in critical systems implementations – also requirements coverage can be automatically derived from code coverage.

It should be noted, however, that the correctness of such evaluation always depends on the correctness of the formal specification of the requirements or the requirements annotations in the code.

B. Coverage

Although complete coverage was not achieved, the measured coverage figures are quite impressive given the simple means of stimulation, within an acceptable timeframe between one day and several days for 1530 FUTs, requiring only computer time.

The points of manual intervention are the definition of the test configuration, identification of constraints as discussed above, the analysis of reported anomalies, the upgrade of identified test case candidates to test cases and the evaluation of the generated test suite against relevant specifications.

In addition, violation of implicit pre-conditions by the stimulus generator leads to false positives as indicated by the anomalies recorded in this project.

The effort for investigation of false positives can be reduced by covering a high number of pre-conditions by a minimum or at least reasonable number of rules, e.g. by applying good naming conventions together with well organised documentation.

In the case study about 1400 pre-conditions on parameters could be covered by 27 generic pre-conditions. This number would be even smaller if naming would have been better harmonised in the software. The number of false alarms could be reduced significantly and the coverage figures increased. The software was written with well-defined coding standard, however this does not take into account exact naming of function parameters corresponding to their intended function.

In case of out-of-range conditions of indices pre-conditions on stimulation can be derived automatically by correlating parameters occurring as indices with the boundaries defined in type and data declarations. This led to a reduction of false alarms.

Another recurring pre-condition was the use of initialisation functions before starting stimulation of a FUT. As again these pre-conditions were not formally documented we had to apply heuristics to identify candidate initialisation functions by simple data flow analysis and patterns for file and function names. One common pattern was looking for function names containing the pattern *init*. If common naming conventions for such functions are applied, their identification becomes much easier.

Such documentation could also be used by static analysis tools to verify that the pre-conditions are indeed fulfilled whenever the function is called. In this context what seems to be an overhead turns out as information which should already be available, but is not provided today due to effort constraints and missing requests in the test and verification process.

The question remains whether a static analysis tool can actually lift the burden of verification considerably. Rice's theorem states that there is no algorithm that can decide for any program provided as input whether a specific, nontrivial property holds for that program. This theorem – a consequence of the Halting Problem – limits the capabilities of static analysis tools, leading either to "don't know"-answers by the tool or the necessity to restrict the set of programs accepted. The typical approach of static analysis tools is the "don't know"-route, which again leaves the engineer to verify those properties that the static analysis tool cannot verify.

In similar situations, where the software lacks appropriate documentation of the pre-conditions usable for formal verification, stimulation ignoring the pre-conditions – just by not knowing about them – could thus highlight some critical pre-conditions, the violation of which could have grave consequences. As a consequence of the reported anomalies, appropriate measures have to be taken to ensure that these pre-conditions are always met, e.g. by provision of a formal proof or by use of additional static analysis tools.

Even if it can be proven that the pre-conditions are met, effectively rendering the anomalies hinting at the pre-conditions to be false positives, these false positives would still have additional value for the project by highlighting a condition for correctness of the software that was previously unknown or not considered explicitly. This may be of specific importance for maintenance, as the explicit documentation of these conditions may be used to ensure that the conditions are still fulfilled after changes to the software were made.

The *memcpy*-example from Section II-II.A is a case of a specific kind of pre-condition. Here we find a dependency between the contents of the configuration table listing the plausibility requirements for telecommand checking and the expression used for calculating the number of bytes to copy. In this case the – hidden – dependency was detected just by applying the range specification of a parameter. Specifically fault injection by overwriting global data marked as constant may also reveal such interdependencies. What seems to be of no use at first glance, turns out to be helpful in detecting hidden, non-documented dependencies.

An important observation is that false alarms gave valuable hints on critical issues. Even if the issue pointed to turned out as uncritical in the system context, the related review of the code often lead to identification of other issues not seen before. This can be considered as a psychological aspect: if the goal is to review all the code without having any indication for issues, the motivation is different compared to the case when the review needs to explain and understand a reported anomaly.

C. Standards

A process to integrate the FAST process in the overall test development process and stay compliant with the ECSS-E-40 and ECSS-Q-80, versions B and C and DO178 versions B and C is proposed in the project.

In order to adapt the FAST process to the objectives in ECSS-E-ST-40 or DO-178, the following process is proposed:

1. The DCRTT tool should be applied as described in this paper. In particular note the need to develop a file containing constraints to eliminate false positives.
2. Remaining issues reported by the tool should be investigated as possible errors and the software and/or test suite updated as necessary to remove those problems.
3. When all problems have been resolved all tests should pass without failing. A relatively high structural coverage should be expected, but full structural coverage will not have been obtained at this stage.
4. The auto-generated test suite must now be manually reviewed against the unit test objectives in ECSS-E-ST-40 or the objectives relevant for low level requirements

based testing in DO-178 as applicable. For each function under test the review should focus on:

a. Functional and design requirements allocated to the FUT. As in standard unit test process, test cases cannot be based on interfaces only. It will be necessary to add tests to check correctness of the requirements allocated to the FUT in order to obtain requirements coverage.
b. The need for additional robustness test cases to cover input combinations that are of special interest based on the unit testers knowledge about requirements and design and to comply with all the objectives in ECSS-E-ST-40 or DO-178.

5. Based on the output from the review, manually develop the additional test cases needed. After this step the required structural coverage is also expected to be obtained.

Note that step 5 can be done by manually adding test cases using the DCRTT tool. However, it may in the future also be done by exporting the test cases to a standard unit test tool, if the software supplier for some commercial and/or practical reasons still wants to keep such a tool operational.

A project using the process above will be able to benefit from all advantages of the method, while mitigating all the limitations and stay fully compliant with ECSS-E-ST-40C or DO-178B.

IV. Recommendations

The software under test used in this case study was not written with application of automated source-code-based testing in mind. Therefore some effort was needed for annotating the software, e.g. by adding constraints. The functionality of the software was not changed. Based on this experience a set of coding guidelines was defined with the intent to meet better the needs for this kind of test automation in future projects if followed.

The FAST approach allows early commencement of testing activities as soon as compilable code is available. This way testability issues can be detected without much effort and fixed early in the process. Further, early feedback on code quality in general and robustness and stability in particular is available and indications on pre-conditions previously not considered can be found.

Early fixing of such issues decreases the effort to be spent for analysis of false alarms. In the case study no recommendations could be implemented as the coding phase was already almost completed.

The most convenient case – not only for testing, but also for ensuring consistency – is a function without any pre-conditions. Such a function accepts all possible values in the set of inputs defined by its formal parameters and provides a meaningful output for each of these inputs. This requires proper parameterisation of the input space using orthogonal parameters, the value range of which should exactly match the range of language-provided types. In that case, violation of pre-conditions is either not possible or is reported by the compiler due to basic type checking.

Such a parameterisation is not always possible and even if it is possible, it may not always be desirable, as a complex mapping may increase the complexity of the software to an inacceptable level. Nevertheless it is worthwhile to pursue pre-condition-free functions wherever possible.

If pre-conditions cannot be avoided, it is recommended to use concepts provided by the programming language to hint at these pre-conditions. One example is the use of enumeration types in C. Another is the use of the *const* specifier in C to constrain directionality of a parameter.

In the case that the programming language does not provide a way of expressing the pre-condition or hinting at it, introduce common rules for example for naming the parameters involved in an often-used pre-condition to both increase comprehensibility of the code but also to allow easy introduction of generic pre-condition based on naming patterns or similar.

Another alternative can be the introduction of new language features. The tool DCRTT, for example, uses special tags *_IN_*, *_OUT_* and *_INOUT_* to specify directionality of parameters. The tags are removed automatically before the source code is submitted to the compiler, but the parser of DCRTT sees the tags and uses them.

If these two alternatives of specifying pre-conditions are not applicable, common forms of expressions for such pre-conditions could be defined in annotations. To avoid unnecessary complexity for the developer, specialized forms of expression for specific, recurring types of constraints should be preferred over generalized forms such as higher order logic.

For example, C does not support specific ranges on parameters, but DCRTT does. However, such concepts need to be known before coding starts to make it efficient.

Pre-conditions which are documented in a formal and computer-readable way may also be of advantage for static analysis.

If post-conditions or other properties of the FUT are documented, this may also be used for static analysis or for establishing oracles. For example, symmetries in functions can be exploited for fault detection [14].

The process may be used to aid in establishing basic unit test suites. The effort of confirming the outputs from the test case candidates to be the actual expected outputs is of course still remaining. No conclusion on the amount of this effort so far is possible, but it should be possible to compare the mean number of test case candidates per function to the typical mean number of test cases per function used in classical test processes. Further reduction of the number of test case candidates seems to be possible and we recently made progress in that regard.

V. Conclusions

The case study has shown that the approach can be applied to real-life embedded software without any unreasonable constraints imposed on the way developers implement the software.

The FAST process can be used in the nominal unit test process without any changes to the process as described in

the ECSS-E-ST-40C standard and also in projects that must comply with safety standards and guidelines, e.g. DO-178B.

However, the test cases automatically generated by the DCRTT tool must be complemented with manually derived test cases to meet all the objectives of the standards. The Final Report of the study describes a process that will make it possible to benefit from all the advantages of the method, while mitigating all its limitations and stay fully compliant with ECSS-E-ST-40C or DO-178B.

Problems that were neither detected by the static analysis tool used in Independent Software Verification and Validation (ISVV), nor by the static analysis tools used by the supplier, were found in the project. This illustrates the benefits of using tools that are as different as possible for ISVV, and that it therefore could be cost-effective to use the DCRTT tool instead of a static analyser to complement the manual code inspection in ISVV.

The FAST process does not contradict European space software standards ECSS-E-40 and ECSS-E-80 and can complement it with additional test cases, and also covers some of testing requirements of DO-178B, especially regarding robustness.

The main conclusions are:

- The process is an excellent way to reach a reasonably high level of coverage quickly and with not much effort.
- The large number of test stimuli may select much more combinations than achievable by a unit tester, thereby increasing the quality of robustness testing and the number of findings.
- The tool may ease the transition to more rigorous standards like ECSS-E-40C which focus more on robustness.
- Tool adaptation may be required, in particular when coding practices as discussed above are not applied.
- The auto-generated test suite must be evaluated against test completeness criteria in relevant standards. E.g. if 100% requirements coverage and 100% code coverage cannot be achieved automatically, the remaining test cases still have to be implemented manually. This may either be done by defining test cases by file input or writing test drivers in context of DCRTT or using another tool supporting definition of test cases.

We have seen that adaptations in coding style – such as use of properly constrained parameter types – may increase the efficiency of source-code-based automated testing significantly.

Minor annotations may be necessary to allow for more efficient test data generation and recording of test results, such as annotating parameters with their directionality or using appropriate names to facilitate application of generic pre-conditions. Test data generation still works without these annotations, although it is less efficient and more effort may have to be spent in the later stages in evaluation of the data or complementation of the test cases for coverage.

The process of introducing these annotations may be eased by applying the concepts of source-code-based automatic testing early in projects as soon as compilable source code is available. An early feedback may help increasing the robustness and testability as well as the general quality of the code at lower effort than when this is done as an afterthought shortly before or during the test activities.

Unfortunately, no effectively untested software was available for the case study, so that no conclusions are possible on whether this approach in general and the suggested test cases in particular are as effective in detecting defects as manually selected test cases.

The selection of inputs by an automaton free of bias may on one hand improve the fault detection capability by spreading the test cases out more evenly over the input space, but on the other hand an engineer might know quite well which are the critical cases which are important to test.

A considerable reduction of effort is achieved by automatic generation of the test environment, all of the test scripts and the test drivers for regression testing on host and target platform.

The approach is clearly suitable for evaluating robustness and fault tolerance of the functions involved. This feature may be of special interest in complementary verification and validation activities such as ISVV, which is performed by a contractor independent of the original provider of the software and with tools independent from the tools the original provider used for verification and validation.

The test case candidates already provide a considerable amount of the coverage required, and thus may form a good basis for an actual unit test suite. Complete coverage is still difficult to achieve with random or lattice-based testing, even if heuristics such as using constants found in the source code are applied.

If complete structural coverage is achievable at all – generally meaning that no dead-code is present inside the system under test – it could be achieved automatically using constraint-based test data generation approaches[15][16][17]. Such an approach is currently in development at BSSE and will be integrated with the already available tools for automatic testing. Due to the high computational effort associated with constraint-based approaches priority will be given to random and lattice-based testing, as these have higher stimulus throughput and thus can also provide the oversampling that is considered one of the advantages of the approach. Only for those parts that cannot be covered by random or lattice-based testing, constraint-based test data generation will be applied.

ACKNOWLEDGMENT

This case study was funded by national Norwegian and German budgets in context of the ESA GSTP programme, ESA Contract No. 4 000 102 645.

REFERENCES

[1] JUnit Homepage, http://www.junit.org/, last retrieved April 30th 2013

[2] RTCA/DO-178 / ED-12: Software Considerations in Airborne Systems and Equipment Certification, versions B and C

[3] Space product assurance: Software product assurance, ECSS-Q-ST-80, version B dated 10 October 2003, version C dated 6 March 2009

[4] R.Gerlich, C.Dietrich, R.Gerlich: „Fault Identifcation Strategies“, Eurospace Symposium DASIA'09 "Data Systems in Aerospace", 2009, Istanbul

[5] Richard Hamlet and Ross Taylor, „Partition Testing does not inspire confidence,“ IEEE Transactions on Software Engineering, vol. 16, Dezember 1990, pp. 206-215.

[6] Space Engineering: Software, ECSS-E-ST-40, version B dated 28 November 2003, version C dated 6 March,2009

[7] ESA Guide for Independent Software Verification & Validation, v2.0, 29 December 2008

[8] Richard Hamlet, “Random testing,” in Encyclopedia of Software Engineering, J. Marciniak, Ed. Wiley, 1994, pp. 970-978.

[9] Rainer Gerlich and G. Fercher, “A Random-Testing Environment for Ada Programs,” Eurospace Symposium “Ada in Aerospace”, 1993.

[10] T.Y. Chen, Hing Leung and I.K. Mak, “Adaptive random testing,” In Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making. Springer, 2005, pp. 320-329.

[11] Johannes Mayer, „Lattice-based adaptive random testing,“ Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE ’05), ACM, 2005, pp. 333-336.

[12] Patrice Godefroid, “Random testing for security: blackbox vs. whitebox fuzzing,” Proceedings of the 2nd international workshop on Random testing (RT’07), ACM, 2007, p. 1.

[13] Douglas Hoffman, “Using Oracles in Test Automation,” Proceedings of the Pacific Northwest Software Quality Conference (PNSQC 2001), 2001, pp. 99-107.

[14] Arnaud Gotlieb, „Exploiting Symmetries to Test Programs,“ Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2003, pp. 365-374.

[15] A. Jefferson Offutt, Zhenyi Jin and Jie Pan, „The dynamic domain reduction procedure for test data generation,“ Softw. Pract. Exper., vol. 29, 1997, pp. 167-193.

[16] Arnaud Gotlieb, Bernard Botella and Michel Rueher, „A CLP Framework for Computing Structural Test Data,“ Lecture Notes in Computer Science, vol. 1861, 200, pp. 399-413.

[17] Ralf Gerlich and Rainer Gerlich, „Potentials of Constraint-based Methods in Software Verfication and Validation,”Proceedings of Data Systems in Aerospace 2012 (DASIA’2012), SP-701, ESA Communications, 2012