BSSE System and Software Engineering

Improving Test Automation by Deterministic Methods in Statistical Testing

This paper was presented during
DASIA'06: Data Systems in Aerospace
May 22-25, 2006, Palace Hotel, Berlin, Germany
organised by Eurospace, Paris
ESA Document SP-630, 2006

BSSE System and Software Engineering

Auf dem Ruhbuehl 181
88090 Immenstaad, Germany

Phone +49/7545/91.12.58
Mobile: +49/171/80.20.659
Fax +49/7545/91.12.40
e-mail: Rainer.Gerlich@bsse.biz
Ralf.Gerlich@bsse.biz
Thomas.Boll@bsse.biz

URL: http://www.bsse.biz

IMPROVING TEST AUTOMATION BY DETERMINISTIC METHODS IN STATISTICAL TESTING

Ralf Gerlich[1] Rainer Gerlich[2], Thomas Boll[2] & Philippe Chevalley[3]

[1] University of Ulm, 89069 Ulm, Germany, co-located at BSSE,
e-mail: Ralf.Gerlich@bsse.biz
[2] BSSE System and Software Engineering, Auf dem Ruhbuehl 181,
88090 Immenstaad, Germany, Phone +49/7545/91.12.58, Mobile +49/171/80.20.659,
Fax +49/7545/91.12.40, e-mail: Rainer.Gerlich@bsse.biz, Thomas.Boll@bsse.biz,
URL: http://www.bsse.biz
[3] ESA/ESTEC, TEC-SWE, Keplerlaan 1 - Postbus 299, 2200 AG Noordwijk ZH, The Netherlands,
Phone +31/71565-6539, Fax +31/71565-5420, e-mail philippe.chevalley@esa.int

ABSTRACT:

Statistical testing is of increasing interest because it allows full test automation – from test generation to evaluation - and hence reduces significantly the human test effort, while allowing a much broader test range. However, automatically generated tests based on a statistical approach have to cope with the "oracle problem" and the "small target problem". The oracle problem represents the fact that software cannot conclude on the correctness of the derived results by itself, while the small target problem consists in the challenge of hitting sporadic, but important test conditions in a large input domain. In the worst case this problem can result in zero-probability for specific test conditions. Although deterministic testing methods in principle do not suffer from the small target problem, they suffer from an oracle problem as well, which is the problem to know which test cases are needed. In general, the test criteria need to represent all viable questions a test engineer wants to pose on the respective system under test. Therefore deterministic methods are best at pointing out anticipated faults (as far as "anticipated" by a test engineer) while statistical methods can also reveal non-anticipated faults.

BSSE has started already more than ten years ago with first activities in statistical testing, and built related tools. Based on the feedback from recent activities an approach has been defined aiming to overcome the weakness of statistical and deterministic testing by making a synthesis of the best of the two worlds. This approach includes automatic test case generation based on the information in the source code (prototype specification and code structure), optimisation by improved test criteria for statistical testing and methods known from deterministic testing, automated test evaluation and comprehensive presentation of results.

1 INTRODUCTION

Test automation is mostly understood today as automation of test execution – and possibly as automation of test evaluation, mostly in the form of user-provided verification functions. However, most of the effort lies in the identification of test cases – including provision of the expected results or verification functions – and the preparation of the test, independently of whether the actual execution is automated or not. Improvement of test automation must therefore address full automation or – where not otherwise possible – at least simplification of these tasks, including the automation of execution and evaluation, of course.

1.1 Definitions

A test case is a set of inputs and execution conditions under which certain properties and requirements for the system-under-test (SUT) are to be verified. In manual testing this also includes the expected results. Deterministic testing is a method of testing where test cases are selected purely based on deterministic test criteria, functional or non-functional ones. An example for a functional test criterion is an expected result like "x=10" in response to a test case. Non-functional criteria include structural test criteria such as statement or decision coverage. Each test case matching an element of the criterion – e.g. "statement X must be executed at least once" in case of statement coverage – is considered equivalent to any other test case matching that criterion.

Statistical testing is a method of testing where test cases are selected among the possible input domain randomly w.r.t. the functional or non –functional test criteria based on a given probability distribution. Random testing is the special case where a uniform distribution is used.

1.2 Deterministic vs. Statistical Testing

Regarding the expected information about test criteria, deterministic and random testing are clearly the extremes on the scale. While in deterministic testing the test goal is assumed to be completely specified by the given formal criterion, random testing assumes a complete lack of information about test criteria. For statistical testing the criterion is at most given indirectly via the probability distribution.

Historically, deterministic testing is the most natural way of testing from a specification point of view. It is used in context of manual testing, where the given criteria are to be considered merely as a minimum. Depending on specific vendor standards and the experience and thoroughness of test engineers, test case sets may be manually designed which served the actual goal of testing and thereby typically overfulfil the required minimum criteria.

In the last years several approaches – mostly based on the advances in the field of constraint solving – have surfaced which allow automation of deterministic test case identification [Gotlieb00, Durrieu]. They focus mostly on the fulfilment of the given structural coverage criteria and assume that any test case set fulfilling the coverage criteria is equivalent to any other test case set doing so. While the number of test cases is thereby reduced a-priori together with the effort for manual evaluation of test results, experience of test engineers is lost and the goal of fault identification is replaced by the goal of coverage.

In case of statistical testing test cases are chosen randomly by automation from the input domain (“SUT is considered as black-box”), and the coverage is just measured as a result and confirmation of fulfilment of the minimum test criteria, such as statement coverage or MC/DC. This results in a more spread selection of test cases which according to our experience is also often more successful in detecting faults, specifically non-anticipated faults.

Automated deterministic testing is currently limited to simple applications: limited size and nesting of source code, limited variable types like boolean (there are specific problems with float values or pointer aliasing), simple coverage criteria like statement coverage for which a single execution of a statement is sufficient to consider it as being verified. More complex SUT and extended test criteria are likely to exceed the current possibilities of these methods.

In contrast, statistical testing employs very simple and efficient means for test case selection. Huge test case sets can be automatically generated fast and easily. Worst-case execution and memory complexity of the method is solely affected by the cardinality of the input domain of the SUT. This capability is both a strength and a weakness of statistical testing. A huge test set cannot be subject of manual verification. Therefore the benefits of statistical testing are currently mostly acknowledged in the area of robustness testing, e.g. based on the criterion that no exception shall occur under nominal and non-nominal conditions.

Statistical testing also suffers from the "small target" problem. Specific but important testing conditions may be represented by subsets of the total input domain with a small net hit probability according to the given probability distribution. The most extreme case would be a probability of zero for selecting a test case matching the given conditions, e.g. in case of a finite subset of an infinite input domain such as recursive data structures.

Therefore an extension of statistical test generation towards deterministic testing by consideration of verification criteria seems to be a promising approach. By previous activities in the context of real projects ([AISVV], [ACG]), related to subprogram stimulation from parameter type range, block coverage figures between 50% and 70% could already be achieved. To reach full 100% coverage deterministic test case generation can be considered as a complementary activity. For further improvement more specific test criteria can be introduced to reduce the large number of automatically generated test cases to a much smaller number which is adequate for manual evaluation. In any case, however, deterministic methods must not negatively influence the unbiased selection of test cases and thus the ability to identify non-anticipated faults.

1.3 Test Case Optimisation

Even though it is impossible to identify "how many faults left" with zero being the ultimate goal, the commonly used metrics are nothing more than indirect metrics regarding identification on how many faults are still not detected. In case of valid and reasonable test criteria it can only be concluded that a test case set is insufficient if the criterion is not fulfilled. The reverse assumption (e.g., "coverage criterion fulfilled" means "good test case set" w.r.t fault identification) is not generally true.

However, many of the current deterministic test approaches seem to be based purely on this reverse assumption. This is different in statistical testing, as the test cases are not generated according to the criterion. Instead they are generated without explicit bias. Only after test execution the test case set is evaluated for the fulfilment of the criterion, thereby providing a much larger set of test cases due to automatic test generation. This allows for quantitative statistical analysis, but also bears a high chance of finding better test case sets than by simple fulfilment of coverage criteria.

In addition to test cases derived from functional test goals (e.g. expected x==10), non-functional properties may also drive test generation as well. Such a property may be a structural criterion like “coverage” requiring that a certain coverage criterion, e.g. statement coverage, will be met by the set of generated test cases.

From a rigorous point of view all above test criteria should be addressed. However, due to limitations in budget and effort and schedule constraints, and deterministic testing based on manual testing, not all of the test goals can be met.

From a principal point of view the final decision on correctness of properties has to be made by an engineer on a case-to-case basis, even when full test automation is applied. Consequently, the number of test cases for manual verification should be reduced.

This reduction takes place by two separate processes. Firstly, from the total set of generated test cases a subset is selected according to the given test criteria, such as coverage (in a general sense), robustness criteria (e.g, "no exception may be raised" or "no timeout may occur") and plausibility or correctness criteria in the form of verification functions. Test cases flagged as "interesting" or "important" by these criteria are to be kept in the test case subset for manual inspection. Specifically the verification functions need not necessarily be correct, i.e. they can flag correct results as incorrect. As long as the ratio of such incorrectly placed flags is low, the incorrectness of verification functions only leads to a slight increase in the size of the selected test case subset. Incomplete verification functions flagging incorrect results as correct, however, should be avoided, as it may keep test cases indicating faults from being raised to the test engineer.

Secondly, methods from deterministic test case generation are used to solve or diminish the "small target" problem. This way the required number of test cases required to allow a fulfilment of all test criteria can be reduced together with the time required to generate these test cases.

In order to preserve the important capability of statistical testing of raising problems not hit by deterministic testing, an improved automated test approach should be based on statistical testing, merely being extended by the means of deterministic testing.

Such an extended test approach should consider:

1. provision of test vectors by full test automation covering test case generation, execution and evaluation.

2. selection of a subset of the test cases which most likely represent faults in the software. This selection happens using extended criteria such as the assumption that an exception indicates a fault, or specific, user-defined plausibility criteria, both negative (ensured implausibility) and positive (ensured plausibility).

3. identification of an optimised set of test cases which is sufficient to identify critical and intolerable faults of the software on the target and to prove the correctness of the software up to the required reliability.

   An “optimised” set is a “minimum set” w.r.t. to the test criteria. Consequently, it is not the “absolute minimum” but an adequate minimum.

   By automated reduction of test cases the manual provision of results shall become feasible, while the quality of the test case set is improved. Then results for much fewer test cases have to be evaluated manually, while more functional and more and better non-functional criteria can be applied – as outlined in the next para.

   In contrast to deterministic method, removal of test cases from the full test case set shall take place after their execution so that the results of test case execution can be taken into account by the reduction algorithm. Test cases triggering exceptions or having their results flagged as implausible by plausibility checks shall be kept to a higher amount than those test cases performing normally according to the specified rules (“no exceptions”, “seems to be plausible”). Also such an algorithm shall try to fulfil test rules such as general purpose coverage by the selected test cases.

4. complement of the manual test approach by new capabilities which only can be provided by test automation.

   Robustness can be evaluated by execution of a large number of test cases, requiring only a quantitative analysis of test results based on non-functional test criteria. When applying intelligent automated evaluation methods, answers to questions can be given, which, while important for the validity and correctness of the system, would not have been asked otherwise (e.g., in case of classical formal verification methods).

Moreover, the scope of automation can be extended in the context of verification. As testing does not provide a proof on absence of fault in the SUT, it should be complemented by other, more formal means in the context of automation. As has been shown in [AISVV2005] comparison of messages from independent platforms (compilers, OS, processor type) can help to identify faults in the source code, or even in compilers. To benefit from different platforms porting of the source code may be needed which is only feasible by automation, especially for large applications (10 .. 1000 KLOC).

Today, manual testing implies "deterministic testing", i.e., the goal-oriented identification of test cases from a functional point of view and given a formally specified test goal. This approach can be considered as a "reduction" or "optimisation" of number of test cases from the whole large set of test cases based on available knowledge and requirements on the SUT ("manually defined test profile"), which possibly implies missing of important test cases related to critical issues.

The feedback gained from BSSE activities in statistical testing leads to the conclusion that a combination of deterministic and statistical testing will improve the efficiency and effectiveness of automated testing, and thereby the quality of test results.

The explanation of the approach starts with an overview on the state-of-the-art on deterministic and statistical testing in Chapter 2. In Chapter 3 the feedback obtained by the performed testing activities is presented. The synthesis of both worlds, which is currently being researched and implemented, is described in Chapter 4. Finally, conclusions are made in Chapter 5.

2 ISSUES OF TESTING

Testing aims to prove by an empirical approach that the SUT meets the expectations, either expressed as explicit requirements or provided as informal idea, of which the full extent is possibly unknown. This requires identification of test cases, their execution and evaluation. In practice, testing can never prove the correctness completely because this would require the enumeration of all possible test cases during test, a venture which could take infinite time and projects often not even have the finite time they would need for limited testing.

Therefore the goal needs to be rewritten to "proving the presence of faults". Murphy's[1] and the engineers' own experience tells us that there's always one fault we just did not detect yet. In the context of "Independent Software Verification and Validation" (ISVV) this goal is also known as "demonstration that faults are still present".

From this point of view the execution of a test case can only be deemed successful if a fault was detected. Depending on the level of previous testing this may require further refinement or extension of test cases. If in previous tests a considerable number of faults was found and corrected, the quality and quantity of the possibly remaining faults can be estimated and used as an criterion on whether to continue testing or not. Coverage alone, however, cannot constitute such an abortion criterion.

Manual identification usually results in too few test cases, automatic generation in many test cases, often more than can be executed even automatically within the given schedule.

The basic approach to test case reduction is the identification of so-called "equivalence classes". There are several different definitions for equivalence classes in testing, the most common defining an equivalence class as the set of inputs for which each input yields the same execution path as any other input from the set. This path-equivalence results in a possibly infinite set of equivalence classes in many cases and thus is not practical for test case reduction. In terms of the goal of testing the ideal definition would declare two test cases to be equivalent if they are equally expressive regarding the presence of a specific fault. As for the ideal test criterion, this equivalence criterion cannot be applied in general, but only for anticipated faults.

In practical applications it is therefore necessary to define equivalence in terms of the test criteria, such as coverage criteria or specific types of faults – in contrast to faults in general.

2.1 The Challenge of Fault Identification

The goal of verification is the identification of faults in an SUT w.r.t. to a specification. A specification might not allow to directly define evaluation criteria to decide on whether an observed property of an SUT is correct or not. Therefore the contents of a specification may have to be refined towards feasible evaluation criteria.

[1] It is said that the engineer Edward Murphy, jr. has made a statement to the following effect: "If there's more than one way to do a job, and one of those ways will result in disaster, then somebody will do it that way." It is most commonly formulated as "Anything that can go wrong will go wrong" and is known as "Murphy's law".

(Source: Wikipedia, http://en.wikipedia.org/w/index.php?title=Murphy%27s_law&oldid=46524418)

The goal of validation is the identification of faults in a specification. Consequently, considering a specification as the reference, it is still more difficult to identify faults in a specification, because there is no base upon which a conclusion on the SUT can be made. Faults in a specification can only be identified when the creator of the specification compares the properties of the SUT with the expectations, which should be reflected in the specification.

Testing contributes to verification and validation. “Verification testing” aims to identify faults in the SUT w.r.t. the specification. “Validation testing” forces the SUT to present its properties so that the creator of the specification can identify discrepancies, e.g. some missing functionality which was forgotten in the specification.

Anyway, for identification of faults a clear picture is needed on what is correct and how faults may affect observed outputs. If so, it is straightforward to identify faults or at least to flag possible faults. In this case a scheme will help to identify faults, a so-called fault model.

When having a reference to what is correct, e.g. some information which can be compared with the “answer” from the SUT, it is rather easy to identify a fault, even if a transformation is needed to make both entities comparable.

However, the problem is more complex because not everything in an SUT directly corresponds to an item of a specification, as the process of implementation of a specification also means substantiation of the elements of the specification and thus information is usually added. In this case reference to the specification cannot be made, at all. Therefore the fault model related to the specification has to be extended in a generic manner, based on the way the SUT is implemented. In case of array access, e.g. a rule may be introduced requesting a check of the array index. Then any illegal value of an index can be identified as fault, immediately. For fault removal it is important to signal such an event to the outside world, e.g. by an error message or a return code. Otherwise the check for fault identification would not be needed, at all, because it would be not visible.

Unfortunately, it is not usual at all to apply such fault models. Therefore in an SUT no checks may be found allowing early identification of faults and avoidance of fault propagation. In this case identification of a fault may be related to a message from the run-time system because it does some checks, possibly supported by hardware.

A pre-condition for raising an error message is the execution of the statement in doubt, i.e. of all statements of an SUT, because everything has to be doubted at the beginning. Consequently, a non-zero coverage must be achieved in order to have a chance to get an error message.

2.2 The Challenge of Coverage

A number of different structural coverage criteria are in wide use. Software standards – such as DO178B used for airborne software or ECSS Q80 for ESA space systems – define the criteria required to be fulfilled by software tests on products in their scope.

The most often used criteria are statement coverage, different forms of condition/decision coverage and several dataflow-based criteria. Path coverage is presented as extreme example of a criterion which typically cannot be fulfilled by finite test case sets. In addition, path set and path class coverage are introduced as more general forms of coverage criteria [SmartG].

Statement coverage requires each statement in the SUT to be traversed at least once during execution of the full test case set. It is equivalent to basic block coverage, where a basic block is a linear-sequential code block, i.e. traversal of a basic block respectively traversal of one statement of a basic block implies traversal of all statements in that basic block.

Statement coverage is weak as it does not properly represent the complexity of an SUT. In case of a loop, statement coverage only enforces at least a single iteration of the loop, thereby treating it like a conditional statement. Consequently, back edges in the control flow graph are ignored and the context of the execution of a statement is not considered.

Decision coverage enforces iteration of the different possible outcomes of a decision. Regarding loops it is equivalent to statement coverage as a single iteration implies the loop condition to be true at least once and false exactly once. However, conditional statements inside loops are clearly handled differently as two traversals of one conditional statement are required for full coverage, thereby requiring either one test case with two iterations of the enclosing loop or two test cases with one iteration each with different outcomes for the decision in the conditional statement.

Different variations of decision coverage require the leaf-level parts a decision is composed of – the so-called conditions – to take all their possible outcomes at least once. Modified Condition/Decision Coverage (MC/DC) specifically requires that each condition must independently determine the decision at least once, and thereby targets possible faults in each of the conditions.

Path coverage requires the traversal of each possible execution path at least once. The set of possible paths is guaranteed to be finite only as long as no back edges are present in the control flow graph. As soon as loops are introduced into the control flow, the total path set may be infinite. Therefore path coverage can only seldomly be fulfilled by a finite test case set.

For path class coverage the overall path set is partitioned into a finite set of pairwise disjoint subsets, the "path classes" [SmartG]. For each path class at least one test case is required yielding an execution path from the given path class. Path classes can be constructed by general and extensible rules. The total path set can, e.g., be partitioned based on the number of iterations in a given loop, so that there is at least one test case for traversing the loop once, more than once or skipping the loop.

Note that statement coverage and the different forms of decision coverage do not define path classes, as a single test case can cover more than one statement but only one path class.

Typically, the total path set is not directly known as specific paths may be infeasible, e.g., due to a loop being iterated at least twice for any input. Instead a superset of the total path set is constructed from the structure of the SUT and then partitioned according to the given rules. Infeasible path classes need to be sorted out by various means [SmartG].

The criteria presented above are all based on control flow. There are additional criteria based on data flow. A definition is an assignment of a value to a variable. Reading a variable constitutes a use of said variable. A definition of a variable reaches a use if during execution no further definition of that variable is encountered. The all-uses-criterion, e.g., requires each definition to reach each of its uses at least once during test execution.

Path set coverage is a generalised form of path class coverage, lacking the requirement for pairwise disjointness of the various subsets of the total path set. They are constructed similarly but do not require additional care for assigning duplicate paths to unique path classes. Instead an additional definition is required by the test engineer on whether a test case may count for coverage of multiple path sets if the executed path lies in the intersection of these path sets. If so, test cases for more specific path sets could also cover less specific path sets also containing the respective path. Path set coverage can be used to express any coverage criterion with a finite set of coverage elements, such as the criteria presented above except for path coverage. Moreover, path set coverage can be used for any combination of these criteria with additional custom criteria, e.g. based on a database collecting test experience.

All these criteria define a set of path sets each of which is to be covered by at least one test case. Some of these path sets may be related to very small, finite or even singular subsets of the input domain. Covering such "small target" path sets is difficult with pure random test case generation. In some cases the probability for hitting such small targets may even be zero, possibly requiring an infinite sequence of tries. Even when the probability is non-zero but very small, the number of tries required may exceed the available time budget of the respective project.

The extension of functional code by code for identification of faults has an impact on the coverage figures. If no fault occurs, such branches for fault identification are never entered, and the coverage of the SUT remains below 100%. Therefore fault injection has to be applied as well enforcing execution of all statements.

As the domain of faults is usually much larger than the nominal domain, fault injection is nearly never applied, due to lack of resources. Therefore automated testing shall help to tackle this problem.

In this respect the requirement of a high coverage may even be counterproductive. The pure request to reach 100% coverage may compromise the goal of fault identification. Engineers may omit means for fault identification as described above, forgetting that the coverage criterion is not the ultimative goal to ensure good quality of an SUT, but just an indirect goal implying that by execution of a statement a fault condition may be raised. However, it does not make sense at all to enforce execution without providing means for identification of a fault. E.g. when no branches are added in the SUT which allow to signal the fault to the outside world, the only, but possibly low chance is, that OS services will raise an exception. Hence, a high coverage figure may be reached, but the faults still cannot be identified.

Of course, sophisticated means like mutation testing may help to identify the weakness for fault detection, but as described above much simpler and more straightforward means can already help a lot.

2.3 The Challenge of Test Generation

In the deterministic approach test cases are selected according to existing knowledge on the software, based on the expected functionality. When focusing on the functionality, it is possible that not every branch in the software is covered during test as explained above. This problem is especially critical in manual test case selection, due to the limited ability of the human brain to appropriately understand and capture this mapping. Ironically, this human limitation is part of the reason why tests are required at all.

However, even when a branch is entered, it is not ensured that a possibly present fault will occur in a detectable manner under the given conditions. On the contrary, multiple test inputs may not contribute more than a single test case regarding the identification of a fault. This leads to equivalence classes for reduction of the number of test cases, requiring execution of one test case only from the set of test cases of a class. The following example shall illustrate why test case generation from a specification may not be sufficient, and how the number of test cases can be reduced by equivalence classes.

When a specification requires computation of y=1/x – free of faults – the following steps apply:

1. The specification:

   Provide the results for 1/x in the range
   $$-100 \leq x \leq 100, \quad x!=0$$

2. The implementation (which adds functionality (and faults) not expressed by the specification)

   $$y=(x-1) * (x-2) / (x * (x-1) * (x-2))$$

3. The resulting equivalence classes

   To test the correct use of y=1/x from the specification, two test cases – usually – would be sufficient: x==0 and x!=0. In fact we have two equivalence classes regarding the exclusion of division-by-zero.

   But the implementation is more complex – for whatever reason. Then we may observe an exception at run-time not only for x==0, but also for x==1 and x==2, even though all three test cases identify the same bug: the code is not properly guarded against division by zero a priori. With the goal being to properly test the division-by-zero shield, four equivalence classes are needed for the SUT instead of two when looking on the specification.

   As a side note: This is a simplistic yet evident example showing the dependency of test efficiency on the implementation. In fact, the actual implementation of floating point division either in software or in hardware is much more complex than the specification above may imply. The same is the case for most other software. The effect is even more critical for more complex code.

In the deterministic / manual approach, selection of test cases would be quite simple – in this case. When inspecting the code, one could easily identify the need for four test cases (we ignore here the fact, that most probably an inspection would request the change of the code to 1/x). When generating test data on a statistical basis automatically, depending on the probability distribution of the generator and the range of x the probability of hitting one of these interesting cases may be quite low. And all this considering only the case that x is a direct parameter. If x is, e.g., the height of a data tree passed to the function, the probability of generating a tree with one of the required heights 0, 1 or 2 may be even infinitely small in case of random testing.

For manual test case generation this would be a fairly simple example. Consider, however, large call hierarchies and more complex code scenarios. Even taking out multiple iterations for loops, the number of different alternatives to be considered raises exponentially with the maximum block nesting depth and the length in terms of basic blocks. Obviously the complexity limit for proper manual test case generation will be exceeded by far in realistic applications. Therefore any reduction of test cases here is a reduction in test case set quality.

Deterministic approaches perform a-priori test case reduction, relying purely on the validity and completeness of the test constraints imposed by the test goals and other additional information provided, e.g., by a model applied for code testing.

Statistical testing does not make any direct assumption on the SUT or the test criteria, and hence cannot be biased. As will be explained in chapter 3 below this is why statistical testing will provide answers to questions which where never asked, provided test evaluation do not suppress such answers.

The difference between statistical and random testing is that a uniform distribution is applied. Starting with a uniform distribution is straightforward when nothing is known about the implementation. As soon as more information is collected, the distribution could be better adapted to the profile of the SUT and the test criteria as to allow proper coverage even of small target values.

In case of [AISVV] this was exercised (to Ada code) for “case” where the specification used an integer type (“int”) which is converted to an enumeration type for the “case”. Obviously, a random distribution over the full 32-bit range would poorly cover the range actually used. When taking an adapted profile covering each integer between 0 and 200 (the highest observed “select”-value in a case) the overall coverage increased slightly by about 3% points for an SUT consisting of about 20,000 blocks.

However, this was not really a statistical approach, because between 0 and 200 there was no random choice possible for the first 200 test cases, they were selected in a deterministic manner. Only the remaining test cases were chosen randomly from the complementary range. This leads over to the question whether an adapted profile is meaningful at all. When knowing details already, why not directly choose the discrete values rather than select them randomly?

In case of statistical testing each point of the input domain has the same chance to be selected. However, a point may be chosen multiple times or even not at all. Therefore, taking the probability 0.005 for a set of 200 points, the probability that one of the 200 points is not hit after 200 tests is approximately

$$e^{-1} \approx 36\%$$

Even after 2400 tests the probability is $e^{-12} \approx 10^{-5}$. When combining the statistical and the deterministic set it is ensured that the interval 0 .. 200 is really covered for 100% even after 200 test cases, only. In this specific case the deterministic part of test case selection is specialised for the "case"-statement and the case that the argument to the switch is a direct input to the SUT. Similar adaptations are possible for other cases. However, there is no need to stop there. The main difference to the purely random approach is that more formal and informal coverage can be reached by the same or even a smaller number of test cases. This way, the generation of test cases benefits from a simple deterministic adaptation and still does not lose the general benefits of unbiased random testing. The adaptation can be understood as a goal-directed modification of the probability distribution.

Statements related to fault handling can never be reached when test cases in the nominal range are generated. Therefore a test generator should also support fault injection. When the nominal input domain is known, the non-nominal domain is known, too. However, if the nominal domain already spreads over the whole possible range, from a formal point of view no non-nominal range exists. This may happen if a type is declared as "int" (in C) although the effective range is only 0 .. 200. Consequently, for automated test generation more precise specifications are needed.

However, in some cases the derivation of invalid test cases may even be impossible, because there are no means to drive test generation into the right direction from specification level. An example is (in C):

```
fd=fopen(myFile,"w");
if (fd==NULL) { <error handling> }
```

In this case it is nearly impossible to ensure that the body of the "if" will be entered when the pattern for “myFile” is generated randomly. Therefore a statistical test tool needs to undertake specific actions to cover such branches as well.

2.4 The Challenge of Finding Non-Anticipated Faults

The success of test case execution may depend on conditions which are not identified by deterministic testing or cannot be identified because the complexity is too high or information is missing.

A sequence of two test cases, e.g., may yield correct results, while they will fail when executed in inverse order, because a fault condition is activated due some side-effect of the previous test case. The goal of executing both cases only in one sequence implies missing the fault condition.

By statistical testing the determinism of the test sequence is removed. Hence, there is a reasonable chance to detect a non-anticipated fault, i.e. a fault for which a test condition cannot be derived from the available information or verification requirements.

2.5 Test Approaches

In the following sections a brief survey on the deterministic and statistical test approaches is given.

2.5.1 Statistical Approaches

The basic statistical approach applies a random number generator to generate test inputs based on the input specification of an SUT. Typically, the random number generator follows a uniform distribution, but based on such a uniform random number generator any applicable and calculable distribution law $P(X=a)=f(a)$ can be used.

Adaptive Random Testing (ART, [Chen]) tries to minimise the chance of generating more than one test input detecting the same fault. The method considers different patterns which faults may produce in the input domain, such as stripe patterns, point patterns or grid patterns. The method targets a better distribution of “good, successful” test cases over the input domain (regarding fault identification).

When generating a high number of test cases with Chen's original ART algorithm, the set of generated test cases approximates a grid of equidistant points in the input set. This is similar to “incremental testing” where the range is covered from the minimum to the maximum value (type’first .. type’last) by monotonically increasing equidistant test cases. The strategy of ART is to look for areas where most probably other faults may be detected once a fault was detected by a certain test vector. Assuming that it is very unlikely to detect another fault close to the current fault, the distance to the next potential test vector has to exceed a minimum length.

Following this idea the optimum distance between two consecutive test cases in a set of n test cases, is – for a scalar type –

$$\Delta = (\text{type'last} - \text{type'first}) / n$$

For nested types this approach is applied to each leaf type, i.e. a scalar type. Compared to real random generation of test cases there are two major differences:

1. all test cases of an incremental test set are hit exactly once when moving in equidistant steps from the minimum to the maximum value. Compared to real random testing this is an advantage because then the probability that a certain value out of n is hit at least once after n cases is 100% compared to only ~64% in case of a uniform random profile.

   For an enumeration type with six literals ("dice") it is well known that after 6 trials not every value will have occurred. A much larger number of test cases is needed to be sure that every possible value will have occurred. After 72 trials the probability is 10^{-5} that every value will have occurred once at least. To achieve a coverage of 100% much more test cases are needed. Consequently, "incremental testing" is better for a small number of possible test vectors, because all cases can be met by a minimum number of trials.

2. some test cases of an incremental test are never hit, while in case of real random generation for finite types there is a finite, though possibly small probability, that every value out of a (large) range will be hit.

In this approach test case generation is no more statistical at all, but the "random character" of test case selection is still preserved to some degree, because there is no preference for a certain fault assuming that faults themselves are distributed randomly and their occurrence is not bound to one singular value, but to an ε-region around a centre. From this point of view the probability to exactly hit a fault condition or to come close to the related test vector with incremental testing is as good as with real random distribution, because the real location of faults is not known. Therefore any test case selection can be considered as random regarding a hidden fault. When covering systematically the input domain with a "deterministic", but still random approach is even better than by real random data implying smaller coverage of the whole domain for the same given number of trials.

As for statistical and random testing, incremental testing may also get a certain profile when reasonable, i.e. the equidistant steps are replaced by steps with varying distance.

All the approaches discussed above suffer from fault or test case patterns with extremely small hit probabilities and therefore tend to generate a huge number of test cases when trying to fulfil a given test criterion.

2.5.2 Deterministic Approaches

In deterministic test case generation test inputs are derived by mainly deterministic methods from the original test goal. They are often based on constraint solving techniques.

Gotlieb, et al. [Gotlieb00, Gotlieb01] proposed a Constraint Logic Programming (CLP) framework for generating test inputs based on statement or decision coverage. The method reduces the input set based on constraints derived from the code and the statement to be covered and then selects one test case from the remaining set. Several similar approaches exist, e.g., Durrieu, et al. for tests of Scade models and a structural coverage criterion related to influence of inputs to the outputs [Durrieu]. Similar approaches can be found in testing of hardware processors (e.g. [Bin02]).

Note that solving such constraint sets is not generally possible. Solvers may take exponential time, provide incomplete solutions or lose possible solution values in the process, which may lead to a falsely inconsistent constraint set. Also solvers may be unable to detect that a given constraint is not satisfiable, which is important in the context of entailment checking used by Gotlieb for detecting whether specific paths through the SUT must be traversed.

Furthermore the method proposed by Gotlieb requires the full information about the SUT and all called subprograms to be transformed into a constraint set problem which is to be solved. It is to be expected that with practical code sizes (like about half a million LOC for ATV FAS) application of the method is infeasible.

Nevertheless the above-mentioned methods seem to achieve their goal quite well for small problems. Unfortunately, their adaptation to other or more general coverage criteria is not necessarily straight-forward.

2.6 Synthesis

The aim of test generation is to produce results for verification of an SUT. In case of deterministic testing and manual testing test case generation is "goal-oriented", i.e. the test cases are derived from the goal to get a result for certain items of interest, related to the specification and the implementation. This procedure implies a reduction of test cases according to the identified goals. Simple coverage criteria like "statement coverage" can be applied only.

This is different for statistical testing and automated testing[2]. Here a large number of test cases is derived to get a reasonable coverage and more complex coverage criteria can be used. The test tool generates "test vectors" (inputs and outputs) itself, and the results of interest can be selected from this set. It should be the task of an automated test tool applying statistical testing to support the user in selection of such "interesting" test cases.

In order to reach 100% coverage (of the applied coverage criterion) statistical testing has to be tuned. For a black-box approach a uniform distribution is reasonable ("random testing"). When extending such an approach towards white-box testing and deterministic testing, a non-uniform statistical profile does not lead to optimum results as the considerations in section 2.3 shows. Therefore a tuned random testing approach should be based on a combination of random testing and deterministic test case identification, for such cases which can hardly be covered randomly, i.e. when the probability is too low that such cases will really occur during a practical test.

3 FEEDBACK FROM EXERCISES IN STATISTICAL TESTING AND AUTOMATION

In the past a number of activities with full test automation (from test generation to test evaluation) were executed at BSSE, based on tools automatically generating statistical / random test cases from a specification, in several languages (Ada [AISVV], C [DCRTT] and Java [SmartG]) and for different types of test goals. This experience provided essential hints on how to improve automated statistical testing.

In this context a "specification" is machine-readable and provides information on the inputs which an SUT shall accept. It may be a prototype of a subprogram defined by the types of its parameters and return type – if any, or a spreadsheet including equivalent information on an SUT's interface.

The goal of this survey is to demonstrate that

1. a number of opportunities exist for automated random testing, and
2. it is an inexpensive way to complement conventional testing due to automated extraction of test information from existing source code or data.

[2] Only a combination of statistical and automated testing is useful, because it is hard to believe that an engineer can generate a large number of test cases (justifying the term "statistical"), with uniform profile.

3.1.1 The Tools

The applied tools derive test cases from a specification, but the test goals and the type of the inputs depend on the type of the software. In principle, in this respect we can distinguish between "type-based", "state-based" and "database-based" test generation for the executed activities. To a larger part, the automated tests were executed after the software had undergone the usual tests and verification procedures including ISVV – when applicable.

The test evaluation criteria depend on the type of testing.

3.1.2 Type-Based Test Generation

In this case test data from the range of types are derived, more specifically of types of subprogram parameters, to stimulate the subprograms by provision of test vectors from the valid or invalid range. The test data are (1) selected randomly or (2) incrementally taking test samples in an ordered manner from the type range.

The assumption is that

1. when valid data are passed to a subprogram, no anomaly shall be observed ("anticipated fault").
2. invalid data shall be recognised and rejected by a subprogram ("non-anticipated fault"). Therefore, as in the previous case, no anomaly shall be observed.

The test issues are: the analysis of the robustness of the subprogram-under-test ("robustness testing"), documentation of observed anomalies and provision of test vectors for manual inspection.

These assumptions may raise a discussion regarding "design-by-contract". In this case a callee expects that a caller will not pass invalid data, and therefore a callee does not check for invalid conditions. Obviously, this will raise exceptions when a parameter is of type "int" but the callee only accepts data in the range 0 .. 100, when being stimulated over the full range of "int" according to the prototype definition.

However, during tests the "design-by-contract" must be doubted until it has been proven that the caller really complies with the contract. Therefore the callee must be capable to identify violations during testing thereby suppressing fault propagation. Hence, about assumptions should apply, and stimulation according to the prototype definition is a valid check whether violations of the contract can really be identified by the callee, at least during testing.

3.1.2.1Test of Ada Software

The DARTT tool [DARTT] has recently been applied to two major space software applications: the "Flight

Application Software" (FAS) of ATV (Automated Transfer Vehicle) [AISVV1] and the "Monitoring and Safety Unit" [ACG] of ATV.

The FAS amounts to about 900,000 lines of source code (~430,000 LOC), including about 5400 subprograms to test and about 4000 type definitions. It is category C software. The DARTT tests identified a number of inconsistencies between a subprogram specification and its body, a dormant fault, potentially unsafe code, and potential memory misalignment. For the real FAS target the identified weakness did not lead to a manifestation of the fault, as could be shown by code analysis, which however was not performed for all cases. A statement coverage of about 57% was achieved by random/incremental testing.

The MSU software amounts to about 70,000 lines of source code (~30,000 LOC), including about 818 subprograms to test and about 540 type definitions. It is category A software. The DARTT tests identified seven locations where a potential anomaly could not be excluded from the information available in the context of the subprogram. The DARTT test results were subject of the "Test Readiness Review", during which a deeper analysis was initiated showing that no anomaly can occur under operational conditions. A statement coverage of about 67% was achieved by random/incremental testing.

3.1.2.2Test of C Software

The DCRTT tool [DCRTT] is the equivalent of the DARTT tool for testing of C functions. These are the results from two applications:

1. a customer package of about 32,000 source lines (~ 13,400 LOC), including 54 functions and about 50 type definitions,
2. the specification of the MSU software expressed in C code which was automatically generated from a Scade [SCADE] model, amounting to about 7,200 source lines (~ 4500 LOC), including about 75 functions and about 210 type definitions.

For application 1 an unexpected termination of a test was observed 4 times. Analysis of the detailed report (automatically generated) showed that in two cases the SUT exited in a controlled manner by an exit statement ("anticipated fault"), due to unacceptable conditions which were correctly identified.

In case of application 2 no crashes and no exceptions were observed. One test required about 32 million test cases, resulting from 25 input parameters, another one 1 million due to 20 parameters, even though for each parameter the minimum of 2 cases was selected only. A later context analysis may yield that the 25 or 20 parameters may not be independent, so that the number of test cases could possibly be reduced. The coverage achieved by random testing was 60%.

The following figures 3-1 – 3-4 show some results as presented in the automatically generated document. Equivalent information is also provided by the DARTT tool. Fig. 3-1 gives an overview on observed exceptions per functions. This allows easy identification of critical functions. Fig. 3-2 provides an example of the presentation of input and output vectors. For each parameter and each mode (in, out, return) the variation of the parameter value is shown for each test case. Fig. 3-3 visualises the block coverage in %. In this case most of the functions reach a coverage of 100%. The gap for function 11 is caused by an intended timeout condition (for-ever-loop). In Fig. 3-4 all blocks of the functions are shown and the height of the bars indicated the number of block executions (logarithmic representation). For each branch of the control flow a truth table and the number of observed decision values are provided.

3.1.2.3Test of Java Software

The SmartG tool [SmartG] is (currently) a prototype developed during a diploma thesis aiming to improve coverage criteria. It is being upgraded to a commercial version.

The activities lead to definition of a new coverage criterion, the path class coverage (cf. Section 2.2).

The path class coverage criterion is defined based on a set of rules transforming an abstract syntax representation of an SUT to a number of path class descriptions. The rules aim to take into account typical implementation mistakes, such as not considering the special case of a loop not being executed at all.

The main goal of the basic concept is the establishment of a database for conservation of test know-how in the form of such rules. Instead of focusing on a minimum and uniform coverage of structural elements of the code, the focus is on the coverage of typical mistakes.

Research for extension beyond the simplifying limitations imposed by the time frame of a diploma thesis on the concept is currently in progress as PhD research project.

Test inputs are generated randomly. All test cases are taken into account in a general statistical evaluation. Based on the path class coverage criterion a subset of the total set of test cases is selected for manual inspection. The coverage rules thereby define what kind of test cases are interesting to the test engineers, without generally limiting the scope of the test.

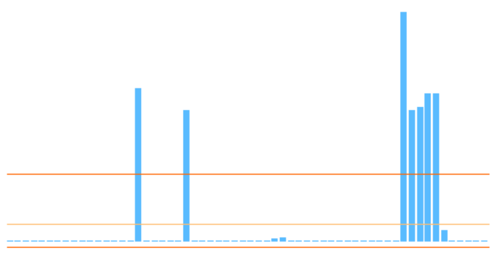
Fig. 3-1: Number of Exceptions vs. Functions

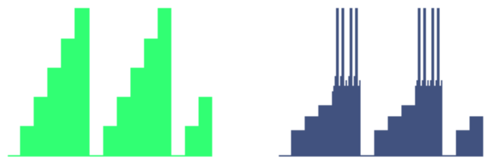
Fig. 3-2: Input-Output Vector

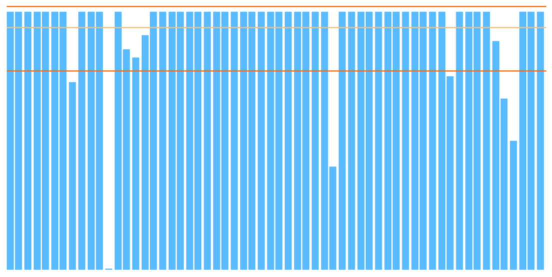
Fig. 3-3: Block Coverage vs. Functions

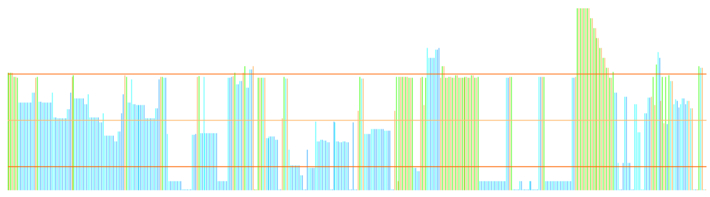
Fig. 3-4: Number of Block Executions vs. Function

Filename	**: whitebox.c**
Function	**: correctWB**
Line	**: 14**
Expression	**: if((a>1) || (b==0))**
	: if(A || B)

Control Flow Item	**False**	**True**
(a>1) || (b==0)	1890	1485
a>1	2025	1350
b==0	1890	135

A	**B**	**Res**
0	0	0
1	0	1
0	1	1
1	1	1

Fig. 3-5: Control Flow Decision Table

For each of the path classes derived by the rules the same number of test cases is required. This ensures that also special cases – which seldomly are considered by testers and implementers, if at all – are considered by the generated test case set. Actual test case generation follows a uniform profile and any faults detected according to the fault model (exceptions, timeouts, non-plausible or incorrect results) are highlighted in any case by the test case selection algorithm, independent of whether the test case represents a special or a more general case.

Surprisingly, logical faults could be identified just by the recorded coverage figures and the statistical evaluation. One such scenario included the detection of a divergence between the expected and the actual weights of the targeted kinds of test cases. The identification of a fault with the help of a coverage profile was really not expected. This demonstrates that not necessarily the precise result is needed, but more "fuzzy" information is sufficient.

A simple example shall help to clarify this result. Consider a typical algorithm used for calculating the greatest common divisor of two positive integers:

```
while (a!=b)
  if (a>b) a=a-b; else b=b-a;
return a;
```

One would expect that none of the alternatives inside the loop – a>b or a<=b – is favoured. If the statistics show that one alternative is heavily favoured over the other in the test cases, we can conclude that the algorithm is faulty. A specific look at the test cases for manual inspection and at the source code can then reveal the origin of this fault. If necessary, the set of selected test cases can be refined by addition of appropriate test rules.

3.1.3 State-Based Test Generation

State-based test generation aims to prove the correctness of a set of FSMs and to demonstrate that 100% coverage of the states and actions can be achieved. During testing the FSMs are stimulated by automatic generation of test cases out of the set of inputs expected by the FSMs.

The ISG tool [ISG] was applied for verification of the consistency of the Finite State Machines (FSM) included in the FAS Ada code. The related information was automatically extracted from the Ada code and converted into the modelling language ISGL of ISG by model transformation. Due to time and budget constraints only the information on high-level commands (HLC) could be extracted. Extraction of all commands would have required a detailed data flow analysis and exploitation of the mission database.

39 FSMs were constructed with more than 1000 HLC, nearly 400 states, and more than 10,000 atomic FSM actions in less than 7 seconds by analysis of the Ada code.

The static analysis of the FSMs by the ISG tool [FSMana] confirmed the completeness and consistency of the FSMs. However, the stimulation did not yield a full coverage of the states for most of the FSMs. A deeper analysis showed, that the set of states of a number of FSMs consists of subnets which cannot be reached from each other, when applying the HLCs only. The hypothesis is that full coverage could be achieved, when the full set of commands is injected, but this could not be proven due to lack of information.

The existence of such subnets, not reachable from each other, was a surprise because it was expected that already the set of HLC would be sufficient to achieve full state coverage. Therefore the detection of such subnets originally was not an explicit test goal, but was detected by the rather abstract coverage criterion, not by evaluation of data processing results.

Though it was not possible to fully prove the reachability of each state under all possible conditions, the result is still of interest: only by low-level commands (LLC) the SUT can escape from such a subnet, and this makes verification of LLCs a more critical issue.

Formal analysis can now be refined to ask also for the reachability of any state from any other state, while it might previously have only asked for the theoretical consistency of the FSMs (e.g., no invalid target states or commands). Here the results of statistical test methods have raised important questions which would not have been posed otherwise.

3.1.4 Database-Based Test Generation

This test type aims to check the consistency of the source code with the database which holds the data for commanding of the software.

The TCinjCT tool [TCinjCT] was applied to check the consistency between the definition of FAS telecommands (TC) as included in the ATV mission database and the reception of the commands as implemented in the FAS Ada software. The mission database (MDB) includes about 5600 commands, of which about 5500 were injected. 126 commands were not injected because they could not be received by the Ada software, but were directly transferred to the hardware.

5 errors were detected and confirmed: in one case the definition of a TC in the MDB was not compliant with the implementation in the Ada code, in one case the range of a TC parameter was not compliant with the corresponding Ada type definition, in three cases out-of-range conditions were observed in Ada code when processing a TC.

Moreover, a dormant fault[3] in Ada compilers was detected, and portability issues of Ada for record and length representation clauses were identified. These findings can be considered as answers for questions not being asked. Furthermore they highlight the fact that using Ada does not imply reliable and safe software by itself.

3.2 Lessons Learnt

When starting the above tests with the test automation tools, the basic expectation was just to get a huge set of test vectors and reports on observed anomalies. However, much more information on potential weakness of an SUT was identified. In many cases hints on existing or potential problems were observed which never would have expected, and therefore never would have been subject of deterministic test generation. In fact, the additional observations complement the usual test goals and increase the quality of an SUT.

In case of DARTT a dormant fault (write on logically protected memory, “storage error”) was identified, which could not be detected by usual tests from a principal point of view. In addition potential constraint errors and potential misalignments of data were detected. In case of manual and deterministic testing, such test cases would never have been identified to detect such faults.

As a β-version of DCRTT was available right now, no huge SUT was tested so far. For manually generated code similar weakness of the code like for DARTT was observed amongst which where overflow conditions and illegal pointers. A positive result was that the code automatically generated by Scade did not show any weakness when being stimulated with valid data. This may be not a big surprise, because the SCADE code generator is certified.

In case of SmartG logical program errors could easily be identified due to strange coverage figures. During the experiments not only deliberately introduced faults – such as singular mutations of conditions or arithmetic

[3] A dormant fault is a fault which is not activated during a number of tests, and therefore cannot be removed.

operations – in the example algorithms were detected but also inadvertent bugs in the implementation. Some of these faults concerned very special cases of the application. For example, in case of a binary search algorithm a fault resulted in an item not being found if it was the first item in the array. This was detected based on the specific analysis of exception-raising test cases. Furthermore coverage asymmetry in algorithms expected to be symmetric lead to the detection of faulty condition statements.

In case of ISG tests on the FSMs non-reachable subnets were detected though this was not a test goal.

Finally, in case of TCinjCT inconsistencies between the MDB and the Ada code were detected, and a dormant fault in Ada compilers and non-compliances between Ada compilers and the Ada standard (ambiguous interpretation of the Ada standard while neither of the compiler vendors was wrong).

Surprisingly, as could be observed in the experiments described above, very general test criteria ("fuzzy" w.r.t. to the specification) like achieved coverage or consistency checks are very strong for identification of faults or potential weakness, while not requiring precise results for the performed calculations. The essential point seems to be that statistical tests do not make any assumptions about test issues, thereby not constraining the scope of the tests. This helps greatly in identifying faults which most probably never would have been detected by manual and deterministic tests, of course including such faults which also could have been detected by such tests. A successful combination of deterministic and statistical testing therefore must make sure that the fuzziness of test case selection is kept intact and amended instead of being replaced by more directed test case selection methods.

Another result of these exercises is that “incremental testing” seems to be more suited than real random testing. This could be a consequence of the strategy to optimise the selection of test vectors w.r.t. to a given region or range. Especially, when many enumeration types are used, this strategy delivers much better results for coverage at a given number of test cases.

It was even observed that the fault identification probability is – surprisingly – rather high when only 2 test cases are selected, type’first and type’last. This simple solution was necessary when the input domain is spawned by many parameters, and the number of combinations has to be reduced by choosing a low number per parameter. Obviously, these boundaries are rather error-prone. While range checks can be made by compilers giving a reasonable probability to detect out-of-range conditions in case of strong-typing languages, this is not valid when data from outside are entering a system which by their nature do not have a “face” at all. Then even a compiler cannot identify inconsistencies, because inconsistencies between an external database and a compiled program are out of its scope. Indeed, most of the faults were found for this case.

3.3 Observed Weaknesses

In case of DARTT, DCRTT and SmartG test data may be generated or conditions may occur, which can never happen under normal operational conditions, and against which a subprogram does not protect itself. This causes false alarms. Typically, this happens when a sequence of subprograms is called where subprograms called earlier ensure correct parameter values e.g. for pointer parameters in C, while they are correctly defined in context of the full system.

The reason is that such tests are executed in a limited scope. To avoid such false alarms a broader scope needs to be considered. Such "false alarms" could be useful as well regarding the robustness of a SUT. But if such conditions can really be excluded under operational conditions, they may be confusing.

Moreover, when many parameters are passed to an SUT a huge number of test cases is derived, though only a very small part may be sufficient. In case of tests with DARTT and DCRTT subprograms with 20 .. 26 scalar parameters were observed. When testing with TCinjCT a TC with more than 200 scalar parameters was recognised. Finally, when testing the code generated by Scade with DCRTT one parameter was identified, a structure, having more than 100 elements on top level, expanding to more than 8000 scalar items in total. Even if the base type of all scalar elements would be “Boolean”, a huge number of test cases would have been to be exploited. This raises the need to identify constraints on test case generation.

The number of test cases still increases when not only considering subprogram parameters, but static data which impact the flow of execution during a test.

In case of SmartG the basic problem of statistical testing – the need for a high number of test cases for satisfaction of a given coverage criterion – re-occurred in another form: Some specific cases had such a small stake in the input set that the mean number of test case trials needed to get one test case for the given path class would have been too high to execute the required number of test cases in a reasonable time frame.

3.4 A Challenging Issue

Having entered the first step of full test automation for a number of languages and types of software, the following challenge has to be tackled: mastering of a huge amount of test cases identified for real applications, which even cannot be executed by powerful computers within a reasonable time.

Taking the example of 200 and 8000 scalar items which drive a test and assuming a boolean type (only two values TRUE and FALSE are possible) for all items, then we get $2^{200} \sim 10^{60}$ and $2^{8000} \sim 10^{2400}$ independent test cases. Even if 1,000,000 test cases per second can be executed, this would respectively require $\sim 10^{46}$ years and $\sim 10^{2386}$ years, taking roughly $\pi * 10^7$ seconds per year from which yields $\sim 10^{14}$ test cases per year.

Comparing these huge figures with the expected properties of a software system, it is quite clear, that a huge number of test cases is mapped on the same property. This can be explained by the example of 1/x.

In principle a limited set of test cases is sufficient (theoretically 2 samples) out of the huge set of floating point values resulting from the given granularity of $\sim 10^{-6}$ (float, 32 bits) or $\sim 10^{-15}$ (double, 64 bits).

Therefore the solution of the problem obviously is the identification of equivalence classes within a reasonable amount of computer time. Apparently, the problem cannot be completely solved regarding the theory, but the goal is to find a practical solution which allows to identify a large number of faults, especially the critical faults, by an acceptable number of test cases.

4 OPTIMISING AUTOMATED TESTING

A key issue of test case reduction in case of statistical testing is the understanding on how many test cases are needed for identification of a fault. When testing statistically, the probability of identifying a fault condition needs to be considered. Identification of a fault requires activation of the fault condition and recognition of such an event by the engineer.

From a principal point of view the probability of fault identification P depends on the following three sub-probabilities, when dividing an SUT in independent blocks (without branch) which can randomly be selected for execution:

$$\boxed{P = P_E \text{ x } P_F \text{ x } P_I}$$

where

P_E is the probability for execution of a block during the tests,

P_F is the probability that the fault is activated when the block is executed, and

P_I is the probability for identification of the fault (by an engineer) when it really occurs.

In case of a uniform distribution among blocks, P_E is the quotient of the number of covered blocks divided by the total number of blocks. E.g. if a SUT comprises 1000 independent blocks which can be entered independently, the probability is 1/1000 for one covered block per test case. For n trials (test cases), it is n/1000. Hence, P_E increases with the number of executed test cases.

P_F depends on the granularity of the data type and its range. Usually, the probability is very low that a fault really occurs when the statement is executed. In case of 1/x the probability is $\sim 2^{-56}$ or $\sim 10^{-16}$ if x is of type "double" (a mantissa of double comprises 56 bits). This is where deterministic methods will help to hit the critical values by identification of equivalence classes.

The goal of optimising automated testing therefore must be to increase both P_E and P_F as far as possible, ideally to 1. P_F clearly depends on the number of different contexts in which the faulty statement is executed. Simple statement coverage obviously does not require more than one such context. The coverage criterion is therefore critical for the improvement of P_F. Some types of faults may be anticipated and need to be targeted specifically by means of deterministic test case generation. This knowledge helps to increase P_F from in most cases close to 0 towards almost 1. Still non-anticipated faults can only be covered by statistical testing.

An example on how this can be achieved is: “if (x==10.123)”. If x is a subprogram parameter a code analyser can easily identify “10.123” as specific test case and can add it to the set of random (or incremental) test cases. Then even for “statistical testing” the probability is 100% that the “then”-branch will be entered.

Finally, P_I represents the capability of a human being to identify the fault when it really occurs. According to previous experience, it happened that a fault occurred, but it was not recognised by an engineer because the information on the fault was embedded in a lot of other information, which – in fact – was hiding the fault. Therefore, proper and automated evaluation of tests will increase P_I towards 1 and help to identify and to remove the fault.

4.1 Already Implemented Improvements

As an immediate feedback from the exercises described in chapter 3 a number of improvements were implemented:

- for reduction of false alarms

  Now execution of subprograms prior to execution of the SUT is possible. Initialisation subprograms to be called prior to execution of a SUT are identified by patterns and automatically called before a test is started.

- classification of faults or anomalies

  to improve identification and analysis of reported anomalies and faults,
  and

- optimisation of test case distribution

  deterministic coverage of integers in the range of e.g. -200 .. 200 to better cover enumeration types in the body of a subprogram while an integer value is used in the prototype.

- identification of simple singular cases like “if (x==3.141)” which are added to the statistical test set.

  Such test cases are executed when all other test vectors have been generated according to the selected test strategy (statistical, random or incremental testing).

By these improvements the block coverage could be increased. While for the FAS of ATV about 60% and for MSU Ada software of ATV about 70% could be reached without improvements, a recent test with the improved version of DCRTT applied to customer software yielded about 95% block coverage. Of course, the coverage figures strongly depend on the structure of the SUT, especially on the impact by subprogram parameters in the current version. A further improvement can be expected when test case generation will consider static variables in addition – as far as meaningful.

4.2 Planned Improvements

The next planned steps are to identify and to apply improved coverage criteria and test strategies. So far only "statement coverage" is the "criterion of choice", which – however – requires that P_F is 1, i.e., it is assumed that all faults can be detected immediately by a single execution of a statement, which is not really true. Therefore the much stronger "path class coverage" criterion – as discussed above – is being introduced in DARTT and DCRTT, considering different types of paths which the program can traverse.

In a near-term perspective constraints shall be considered by which the coverage of low-probability elements of the coverage/selection criterion can be improved, and the use of generic test rules ("test meta-information") shall be extended.

4.3 Synthesis of Statistical and Deterministic Testing

A synthesis of statistical and deterministic testing must merge the strengths of both worlds together with automation capabilities, considering automated statistical testing as root which is enhanced by the advantages of deterministic testing. The rationale for the use of statistical testing as the base for testing is that no specific information is required to generate tests cases.

It has been shown that 60 .. 70% (up to 95% by recent improvements) of block coverage can be reached by random testing. In case of path class coverage, lower but still acceptable figures have been reached. Therefore the first step in the generation process is statistical test case generation. Any missing coverage is to be provided by the second step, consisting of constraint-guided statistical test case generation. As the criteria typically do not constrain the input set to a single test case, multiple test cases can be selected statistically from the remaining constrained input set.

In the targeted approach statistical testing is associated with full automation of testing from test case identification to presentation of evaluated results. From deterministic testing the identification of rules is adopted which are related to information extracted from source code. So far the constraint-based approaches only support simple coverage criteria like statement / block coverage (C0). These methods need to be extended for allowing the application of generic test rule sets. A near-term goal is C3 coverage, the coverage of paths.

4.4 Test Generation

As the distribution of faults is unknown, any profile chosen for test generation may be good or wrong. A uniform profile may be wrong when faults are grouping around a certain region, it may be the right one when faults are distributed uniformly across the input domain. When defining a profile it is important not to exclude or to give less attention to sub-regions where faults can be found. Human analysis may guide the right way, but if some conclusions are wrong, a wrong profile will be selected. Therefore a better approach is to control test case selection by feedback from the software-under-test.

What are suitable criteria for such a feedback? The following two ones should be of relevance:

1. a uniform distribution for execution of statements / blocks and a maximum coverage of path classes,

2. the variation of the test output vector w.r.t to the variation of the test input vector.

A high number of executed statements / blocks for independent test cases increases the probability $P_E x P_F$ to activate a fault. A uniform profile may be sufficient, but if more information is available on P_F a non-uniform profile could be identified. When all paths are chosen by which a block can be reached, this increases the confidence in having generated independent test cases.

The variation of the output tells a test engineer something about the potential fault density. If the output is frequently changing it may depend on many parameters and complex logic. Therefore verification should put emphasis on such regions. Vice versa, if the output does not change much, only a few test cases out of this region may be sufficient for verification.

As a measure for the variation of the output, the distance between two arbitrary nested structures could be defined in the same way as the distance between n–dimensional vectors is computed. Then this distance is divided by the change of the input vector. Obviously, computation of such a measure only makes sense when the test input changes monotonically. Therefore the incremental test mode is well suited to derive such measures.

When controlling test generation this way, it is fully sufficient to start with a uniform, statistical distribution, because the feedback will drive the generation profile towards the direction as defined by the optimisation criteria, and will give advice for which test cases verification is important. The statistical distribution ensures that no region of interest is excluded.

In addition to statistical test case generation the "deterministic approach" to test generation can help to enter branches which will hardly be covered by random test cases due to very low probability to hit them. The state-of-the-art for deterministic testing is to identify only one test case by which a certain branch can be reached, i.e. to achieve statement coverage. However, the formula given at the beginning of this chapter indicates that a number of test cases will be needed to get sufficient confidence in detecting a fault. Therefore more than one test case should be identified, which requires more sophisticated methods for constraint solving.

Another issue may be the identification of an optimum test set allowing the minimisation of the effort for test repetition. In case of full test automation such an optimisation is not really needed, because all the tests can be repeated without much human effort. Also, such a test set does strongly depend on the SUT. Therefore a replay of a test scenario optimised for a given software structure may lead to erroneous results – faults may remain undetected – when being applied to a changed SUT.

However, if tests need to be repeated on another environment, e.g on the target system, for the same – unchanged – SUT, the test effort and duration can be reduced. An optimum test set can be identified in the development environment which provides better support for execution of a large number of test cases and to derive a feedback.

4.5 Test Execution

For test execution an environment has to be built (automatically, to be efficient enough) which allows to generate the test cases according to the chosen strategy, to stimulate the SUT, to derive and record the feedback and to capture anomalies like exceptions.

The environment for test execution forms the base for test evaluation. It relies on instrumentation of the SUT providing the capability for observation of properties. The degree of instrumentation must be chosen properly in order not to corrupt properties of the SUT resulting in wrong results. E.g. timing properties may be impacted by heavy instrumentation, therefore it should be minimised when looking on such properties.

A key point of instrumentation are the capabilities for parsing of source code and handling of user-defined types. User-defined types may be simple scalar types, but also complex nested structures. Operations for generation of test cases and monitoring of such types cannot be created automatically by means of the class concept of OOP (Object Oriented Programming) as an infinite variety of structures has to be covered, but only by automatic code generation.

4.6 Test Evaluation

As the amount of information derived from the automatically executed tests is very high, automatic evaluation and compression of this information, e.g., by statistical methods and graphical overviews, is a must.

Clearly, the probability P_I for recognition of a fault when it occurs is 0 when no information is provided at all. However, it is also close to 0 when too much information is presented, because then an engineer can hardly detect the relevant information within a large stream of mostly irrelevant information. These two extreme cases are the ones which are very easy to achieve. In case of no instrumentation there will be no information of interest, this is trivial. If the SUT can be properly instrumented, a lot of information will be produced, and then this capability is useless as well.

What is the real challenge is to allow an engineer at little effort to control what shall be provided. If an SUT comprises 5000 functions it is possible of course to ask an engineer to define for each of the 5000 cases the desired filtering of information. However, this will require a lot of effort and time, and is not very efficient, though much effort is already saved by automatic instrumentation.

Therefore the test environment must be intelligent enough to identify the “interesting” information itself. This capability will reduce the human effort to an absolute minimum and to speed up test evaluation while optimising the chance to detect a fault.

4.7 Improvement of Testability

A future challenge is to improve testability by getting better information on the test input domain. This is especially true for C and Java software, as they provide only a very limited type system and have no means for specific range limitations. In many cases this implies that the range of the specified type exceeds the actual valid and useful range by a large factor. This means that during test case generation a huge number of invalid inputs is generated and a high amount of test effort is required for testing an interface. Then guards need to make sure that invalid inputs do not cause the system to cease functioning or – even better – are not passed at all. In contrast to Java and C, Ada provides means for densely restricting the value set of a type. However, these means actually have to be used to improve testability: At any interface the most restrictive type regarding the valid values should be declared for parameters. As a side-effect this also allows the compiler to statically check whether the given type range is obeyed in calls or at least to insert dynamic range checks which are helpful in fault identification during test. Otherwise such checks would have to be done manually, increasing effort for development, testing and maintenance, or inserted by automatic instrumentation.

Also, the number of automatically generated test cases can be reduced when knowing more about the nature of a variable or subprogram parameter. When e.g. a variable or parameter is declared as constant, it is clear that it should not considered for test vector generation. Similarly, when a parameter is declared as OUT parameter no inputs need to be generated. This will reduce the input domain by one dimension in each such case. In principle, some information can already be derived from the parameter type or by code analysis. When e.g. a parameter of type “int” is passed (“by value”), it is evident that it is an IN parameter. Unfortunately, OUT parameters cannot be identified this way because passing of a pointer “int *” implies INOUT. Only by additional code analysis a conclusion is possible whether it is a pure OUT parameter. Such a conclusion may be very difficult when a call tree must be analysed.

To support these issues DCRTT allows definition of type ranges for user-defined scalar types in a separate file, and identification of IN, OUT and INOUT function parameters. This information is considered for test generation.

The higher the constraints are , e.g. on type ranges, the higher is the probability to detect faults by automatic evaluation of properties and test results in case of automated testing. By (automatic) instrumentation the values of data can be automatically monitored, and any out-of-range condition can easily be identified this way.

Consequently, the strength of test automation can still be increased, when already during the development of a SUT more is done for its testability.

5 CONCLUSIONS

Recent experience with statistical testing demonstrated the benefits of this approach, but also provided hints on possible improvements. Analysis of recent deterministic test approaches yielded a number of interesting features which can complement and enhance pure statistical testing. However, the results obtained from statistical testing clearly show that future improvements in testing must be based on statistical methods. Firstly, the amount of test cases needed to rigorously test an application can only be mastered by automated statistical test approaches, though the help of deterministic methods is needed to reduce the number of test cases required. Secondly, the unbiased identification of test cases is only possible by automation combined with statistical test case generation. Especially, the provision of answers to questions which never would be asked in case of a deterministic approach is a very strong point of the statistical approach.

The proposed synthesis of statistical and deterministic methods is considered as a next step of improvement, but it is not the final one, for sure. Hints for further improvements can be expected when having implemented this next step, especially regarding mastering of highly complex software systems, huge test case sets, required testing rules etc.

6 ACKNOWLEDGEMENTS

Ralf Gerlich would like to thank the University of Ulm, Germany, for the support by a Ph.D. scholarship, under which parts of this ongoing work were carried out.

7 REFERENCES

[ACG] DARTT Test Results ACG-MSU, ACG-TR1-BSSE, Nov. 2005
Automatic Code Generation (ACG), ESTEC contract no.18670/05/NL/GLC

[AISVV1] DARTT Test Results AISVV-FAS, AISVV-TN2-BSSE, 2005, Automated ISVV, ESTEC contract no.18056//04/NL/JA
R.Gerlich, R.Gerlich, Th.Boll, K.Ludwig, Ph.Chevalley, N.Langmead: "Software Diversity by Automation", DASIA'05 "Data Systems in Aerospace", 30 May – 2 June, 2005, Edingburgh, Scotland

[AISVV2] Report on AutoPorting and DARTT Module Tests, AISVV-TN5-BSSE, 2005,
Summary Report, AISVV-TN4-BSSE, 2005,
Automated ISVV, ESTEC contract no.18056//04/NL/JA

[Bin02] E. Bin, R. Emek, G. Shurek, A. Ziv: "Using a constraint satisfaction formulation and solution techniques for random test program generation", IBM Systems Journal, Vol. 41, No. 3, 2002

[Chen] T.Y. Chen, R. Merkel, G. Eddy, P.K. Wong: Adaptive Random Testing Through Dynamic Partitioning, Fourth International Conference on Quality Software (QSIC'04), 2004, pp. 79-86

[Durrieu] G. Durrieu, O. Laurent, C. Seguin, and V. Wiels: Automatic Test Case Generation for Critical Embedded Systems, Proceedings of Data Systems In Aerospace (DASIA 2004), Nice, France, June 2004.

[DARTT] Dynamic Ada Random Test Tool, http://www.bsse.biz → Products → DARTT
DARTT User's Manual, BSSE, 2005

[DCRTT] Dynamic C Random Test Tool, http://www.bsse.biz → Products → DCRTT
DCRTT User's Manual, BSSE, 2006

[FSMana] see TCinjCT, the information is included in the same report

[Gotlieb00] A. Gotlieb, B. Botella, M. Rueher, A CLP Framework for Computing Structural Test Data, Lecture Notes in Computer Science, Volume 1861, Jan 2000, Page 399

[Gotlieb01] A. Gotlieb: InKa: An Automatic Software Test Data Generator, Proceedings of Data Systems In Aerospace (DASIA 2001), Nice, France, May 2001.

[TCinjCT] TCinjector, Statistical generation of telecommands, "Report on System Tests", Automated ISVV, ESTEC contract no.18670/05/NL/GLC, AISVV-TN3-BSSE, Nov. 2005

TCinjCT User's Manual, BSSE, 2005

[ISG] Instantaneous System and SoftwareGeneration (incl. test generation),
http://www.bsse.biz → Products → ISG
ISG User's Manual, BSSE, 2000

[Mayer] Mayer, J. 2005. Lattice-based adaptive random testing, Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering (Long Beach, CA, USA, November 07 - 11, 2005). ASE '05. ACM Press, New York, NY, 333-336.

[SCADE] SCADE tool, ESTEREL Technologies, Toulouse, France

[SmartG] Ralf Gerlich, diploma thesis: Size-optimising Automatic Random Testcase Set Generation for Verification and Validation, University of Ulm, July 2005