BSSE System and Software Engineering

Evaluation of Auto-Test Generation Strategies and Platforms

Ralf Gerlich [(1)], **Rainer Gerlich** [(2)], **Thomas Boll** [(2)], **Johannes Mayer** [(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)] Ulm University, *Inst. of Applied Information Processing, Helmholtzstr. 18, 89069 Ulm , Germany,*
Phone +49/731/50-23573, Fax +49/731/50-23975, e-mail: johannes.mayer@uni-ulm.de
URL: http://www.mathematik.uni-ulm.de/sai/mayer/

ABSTRACT:

As the test effort takes a significant part of the software development lifecycle, efficient test strategies are a pre-condition for reduction of development costs and time. In this respect two main issues exist: firstly, the tuning of the test track from test case identification to evaluation, secondly, the reduction of number of test cases to be processed and evaluated. Both aspects were considered in the work presented in this paper.

For reduction of the effort related to the test track two test automation tools have been applied: DCRTT for C and SmartG for Java. While DCRTT is ready for industrial use at high degree of automation of all test steps, SmartG is a prototype exploiting the identifcation of path sets by random testing. DCRTT only requires provision of the source files and then delivers test drivers, a filtered and reduced set of test cases, related test results and detailed information on data ranges and observed exceptions. The manual effort is reduced to test evaluation.

DCRTT identifies a significantly reduced set of test cases left for manual evaluation. The selection criteria are based on block coverage, decision coverage and occurrence of exceptions when generating inputs from the valid and invalid data range. One or more test cases may be collected for each basket of such a criterion and element of a function's structure.

It is of high importance how well such automated strategies do work. Therefore a number of investigations have been performed, evaluating the achieved coverage and number of reported exceptions. Three test modes have been considered: random and lattice-based test generation for module testing and operational testing imposing the complete main program to representative operational conditions.

The practical findings are evaluated against theoretical considerations.

1 INTRODUCTION

The manual selection of proper test cases for detection of faults is a challenging task. The input domain is unstructured and usually very large. By the structure of the source code it is divided into sub-domains reflecting the program's logic, forming "equivalence classes" regarding the test goals.

Therefore a subset of the full set of test cases ("optimised reduced test set") is sufficient for fault identification. However, the reduction of the full set to a reduced set remains unknown to the developer or tester due to the complexity of the mapping. This may lead to inefficient test case selection and an insufficient number of test cases.

Test case definition based on a specification ("black box testing") does not consider a program's structure and extensions needed for implementation. A limited number of test cases derived from a program's structure may not consider all what is addressed by a specification.

The considered approach is based on the following conclusions: when applying a representative set of test cases, i.e. a reduced but sufficient test set, derived from the implementation, "the code", all the properties as required by the specification should be made visible, but also the non-compliances.

For realisation of this concept automated test generation is applied in order to identify a proper subset, taking the implemented software as information source. In practice, this means to consider prototypes (declarations) of subprograms (like C functions) and related type definitions. This is sufficient to build a complete test environment for each of the functions included in a set of source files without manual intervention.

The automated test generation does reduce the manual effort needed to identify, run and evaluate the test cases, when identifying test cases of interest out of a large input domain. Such auto-generation and selection of test cases is based on a chosen strategy. But what is the best strategy? There are a number of questions regarding the efficiency of hitting faults: what is the most efficient strategy, do different strategies complement each other or does one exist which is the best, and which manual effort remains at the end?

Besides the algorithms applied to identification of test cases of interest, more parameters may impact the decision whether to select a test case or not – a result which was found during recent exercises on auto-testing.

Surprisingly, it turned out that a compiler may generate code which does not allow identification of critical cases. Of course, this is not a specific disadvantage of auto-testing, it is a matter of test execution in general, but recognised when running the auto-testing tool on different platforms. This observation leads to the recommendation to exploit test results on different compilers in order not to miss such a test case. If the test environments are built by a test tool, no manual effort is required – except for result evaluation.

Coverage in its various forms is considered as an essential criterion for acceptance of tests, as it is a metric on which part of the software has ever been executed.

Not surprisingly, because frequently applied, is compromising test representativity by selecting one test case only to fulfil each element of a coverage criterion, e.g. a single statement in case of statement coverage. One test case is not sufficient in most cases. When a block contains a polynomial of degree n, n+1 samples ≡ test cases are needed, at least, to be sure that the implemented algorithm meets the specification. The reduction to one test case per block is not adequate at all and seems to be imposed by the limited human resources. In case of auto-test generation this problem does no longer exist.

Similarly, the restriction of coverage analysis to block or statement coverage, but not to decision coverage compromises the representativity of tests regarding later operations.

Exceptions are indicators for critical code sections. No exception should occur at all for high quality software. When stimulated by valid data[1], obviously no exception should be propagated to operating system (OS) level causing program abortion. Being stimulated by invalid data, also no exception should be propagated to the OS level, as quality software should protect itself against inadvertent data.

On module testing level, focusing on a Function-Under-Test (FUT) only, expected (anticipated) exceptions, such as “file not found”, may propagate outside an FUT, because they may be handled on a higher level. In this case, it needs to be checked whether the reported occurrence of an exception is anticipated and handled or not.

Consequently, an exception observed during the auto-tests may indicate a problem. Unfortunately, experience shows that many unhandled exceptions are raised when software is exposed to auto-testing, only being tested manually before. This has also been observed in [1,14].

Sufficient coverage is a pre-condition for fault identification. If coverage of a certain part of source code is zero, there is no chance to detect a fault in this area. However, it is not sufficient to know that coverage is zero, but information is needed how to move to sufficient coverage.

By sampling the code by a high number of test cases and by applying filtering criteria, auto-testing can guide to find the test cases needed for sufficient coverage and result evaluation.

The paper is organized as follows: Section 2 gives an overview of the approach and its employed methods. Related work is also discussed. Section 3 is dedicated to a description of the tools implementing the approach. The results of case studies are thereafter presented and discussed in Section 4. Conclusions are given in Section 5.

2 STRATEGIES FOR TEST GENERATION

2.1 Theoretical Foundations

Random testing is a well-known and often used strategy [15]. Thereby, test data is randomly selected from the input domain. Based on this approach and the assumption that failure-causing input appear somewhat clustered within the input domain, Chen et al. [16] have

[1] E.g. for the sqrt-function for real (float) data, negative values are considered as invalid. In general, the valid range depends on the applied algorithms. In Ada a valid range can be more precisely defined by introduction of user-defined types with adequate ranges. In C or Java this is not possible. Then the algorithms must be protected by explicit range checks (“guards”) against invalid data.

proposed "Adaptive Random Testing" (ART) which evenly spreads the test cases within the input domain. In many situations, ART is more effective than pure random testing. A special implementation of the idea of ART is provided by Latticed-Based Adaptive Random Testing (LART) [17]. LART selects test data from a dynamically refined lattice structure imposed on the input domain. In order to retain the advantage of randomness, the lattice points are selected in random order and their position is randomly translated by a small amount. LART has been shown to select only about half as much test cases as D-ART [16] does (in order to detect the first failure). This is quite close to the theoretical optimum of 50% of the number of test cases of random testing [18]. Besides those ART approaches, it has been shown that a partition testing strategy that randomly selects test cases from partitions such that the number of test cases related to the number of elements is equal for all partitions can outperform random testing in every case [19].

2.2 Similar Approaches

Recently (in 2006 and 2007), two papers have been published [1, 2] also discussing strategies for better fault identification and suggesting or applying auto-testing. DART [1] is based on a combination of symbolic reasoning and random testing. In [2], information gained through the execution of randomly generated test data is used in order to guide the selection of further test cases. Both approaches enhance random test data generation in order to be more effective. Another approach that starts with random test data and uses feedback gained through executions is search-based testing [20], which has e.g. successfully been applied in order to automatically generate test data fulfilling certain coverage criteria.

2.3 Applied Strategies

The issues of (auto-)testing can be divided into four categories as shown in Fig. 2-1:

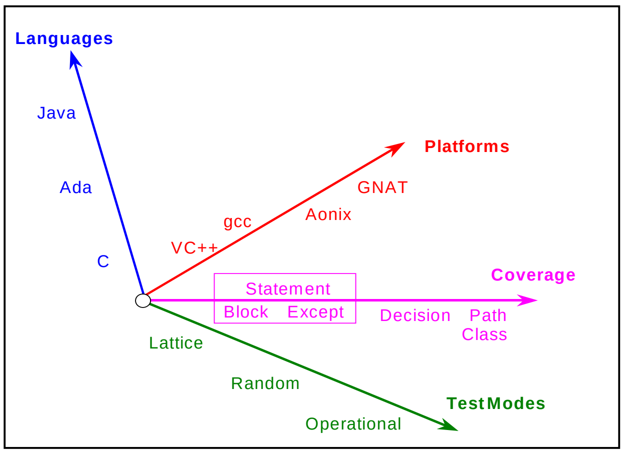


Fig. 2-1: Issues of Auto-Testing

- Languages
- Test platforms (compiler)
- Test evaluation and acceptance criteria
- Test generation modes.

2.3.1 Languages

C and Java are considered as supported languages by the applied tools DCRTT and SmartG (Tab. 2-1). Some figures from previous Ada auto-testing with the DARTT tool [6,7,8,9] are also presented.

	Languages		
	C	**Ada**	**Java**
Tool	DCRTT	DARTT	SmartG

Tab. 2-1: Tools vs. Supported Languages

2.3.2 Platforms

The VC++ (V7, .NET) and gcc (V3.2.3) have been used on an MS-Windows XP-platform to compile and link the generated test environments.

2.3.3 Coverage

During test execution the impact on the code is measured, for which up to four properties are recorded:

- Block coverage
- Decision coverage
- Path set coverage
- Occurrence of exceptions.

Tab. 2-1 gives an overview on which property is supported for which language.

Coverage	**Languages**		
	C	**Ada**	**Java**
Block	+	+	
Exception	+	+	
Decision	+		
Path Set			+

Tab. 2-2: Languages vs. Coverage Support

Block coverage is identical with statement coverage if no exceptions occur in the block and no unconditional branching statement (exit, return, goto) is executed.

A path set consists of a subset of paths which can be traversed while executing the code under test. A path set is said to be covered by a test case if a path from the path set is traversed when stimulating the code under test with the inputs from the test case.

A path set can be described by annotated code constructed from the original code by use of equivalence transformations and constraint annotations. The constraint annotations limit the possible state of the program – i.e. the values which may occur in its variables – and thereby also the choices possible at decision points in the program.

Thereby the set of allowed inputs is constrained. As each input tuple is associated with exactly one path through the program – not vice versa, as traversal of a single path can result from different input tuples – the constraints can be used to also limit the set of possible paths.

2.3.4 Test Generation Modes

A number of test modes have been applied to generation of test cases. Coverage criteria were used to filter test cases of interest for manual evaluation out of the huge number of auto-generated test cases.

The overall strategy to find test cases of interest is divided into two sub-strategies

- a strategy for auto-generation of (a large number of) test cases, and
- a strategy for identification and filtering of test cases of interest, intended to re-run in a simplified environment, on the development or target system, for auto-checking of previous and current results in case of regression testing and for manual evaluation.

A number of strategies were exploited for test generation:

- random-based testing
- systematic testing
  - lattice-based testing
  - code-analysis-based testing
  - constraint-based testing
- operations-based testing including rule-based testing.

Test Mode	**Languages**		
	C	**Ada**	**Java**
Random	+	+	+
Lattice	+	+	
Operational	+		

Tab. 2-3: Languages vs. Supported Test Modes

Tab. 2-3 shows which test mode is (currently) supported for which language.

Random-based and systematic testing address the level “module testing”. In the auto-testing approach these modes directly stimulate subprograms (or functions) based on the prototype definition. Operations-based testing means normal operation of the program as it happens during integration and system level testing or later user operations. Rule-based testing is a mix between auto-testing and operational testing: based on a (formal) specification of the expected input to the main program, test data are randomly generated.

The intention of “operations-based testing” is to catch test cases which were missed during auto-testing on module level.

In case of random testing (pseudo) random numbers are generated a given range.

In case of lattice-based testing the given range is divided into equidistant intervals by a user-defined number of sampling points. For large ranges such as in C long int, long long int, float, double, long double, it is unlikely to hit a reasonable small “value” in a range below 10,000 or 100,000 (except for 0). Therefore the equidistant distribution is slightly modified for such large ranges. For a user-defined portion of the range, equidistant steps are defined based on a linear scale, while above this limit a logarithmic scale is used. This allows to hit reasonable small values, but also to cover the borders of a type's range (type'first and type'last).

For composite types like structures or arrays, the derivation of values is applied to each component.

While for random-based test cases conditions like “a==b” are difficult to hit when values are randomly generated for a and b, latticed-based test generation inherently meets such conditions if a and b of same type (which should always be true from a principal point of view).

While in Ada a user-defined type can be restricted in range, in C, C++ or Java a new type cannot get an own range. Therefore, it is likely that in C always the full range of the base type applies. In order to provide a similar feature like in Ada, a limitation of ranges for user-defined types is supported by DCRTT. The restricted ranges have to be defined separately.

Code-analysis-based testing is an approach derived from practical observations aiming to increase coverage by values used in the code. The source code is analysed for literals (constants, numerics, strings, enumeration values) which are used in conditional expressions. When fed in the related condition is hit and a block can be entered which may not reached by random- or lattice-based test cases.

For such literals not only the identified value is taken, but another value which yields the opposite result of condition evaluation. E.g. in case of "a==100" two values are added: "100" yielding TRUE and another value fulfilling the condition "a!=100". In addition, for strings and pointers NULL is always added to the set of test cases, and an empty string for string types.

Constraint-based testing aims to apply general-purpose constraint solving mechanisms to select test inputs. In contrast to code-analysis-based testing more computationally complex analysis and deduction is applied to find test inputs required for a given coverage criterion. It is to be used as a last-ditch effort when the other, less complex methods (lattice-, random- and operations-based testing) do not find test inputs to reach non-covered blocks.

To optimise the results of test generation, the different test generation modes for "module testing" (random-, lattice-, code-analysis- and constraint-based) may be combined amongst each other, and random- and lattice-based module testing may be combined with operational testing.

Subprogram parameters and global data which impact branching are subject of stimulation during "module testing", while in operational mode the user "stimulates" the program.

In case a limited type range is defined, test cases may be selected from the valid range only, or out of the valid and invalid range. For usual C type ranges, no visible "invalid type range" exists, in fact. The invalid range is implicitly defined by the internals of a function. If not properly protected, exceptions may be raised when exceeding the non-visible valid range.

2.3.5 Test Evaluation Criteria

Automated test generation and execution produces a huge number of test vectors: input and output data ("test vectors") and exceptions. In addition, from the point of view of test execution deadlocks, livelocks and aborts provide further information on unsuccessful test execution. While these anomalies are easy to capture in an automatic manner, the test vectors still need to be analysed manually for fault detection or for proof of correctness.

Therefore the applied strategy aims to reduce the huge number of auto-generated test cases to such ones of "interest", ending up with a significantly reduced number of test cases.

As coverage of a block or branch is a pre-condition of fault-detection, use of coverage criteria seems to be a proper strategy to identify test cases suited for manual evaluation.

Following coverage criteria guide the selection of "interesting test cases":

- Block coverage
- Decision coverage (both boolean conditions)
- Occurrence of exceptions.

Later, the path set coverage criterion [3] also shall be considered.

For all three criteria (block and decision coverage, exceptions) test cases are collected in a basket. A basket may contain a number of test cases up to a user-defined limit. It is common practice to use 1 for this limit, which however is considered as insufficient. Whenever a block is entered or a condition is met a test case is put into the basket unless the given limit is reached. In addition, there is not only a basket for a positive condition, but also for the negative / opposite condition. In case of an "if" or "else if " there are two baskets for "TRUE" and "FALSE", for a "switch" there are baskets for each "case" and "default" (even if "default" does not occur in the code).

Whenever a block is entered or a decision evaluates to true or false while the related basket is not full, a test case is considered as "interesting". Any test case causing an exception is also considered as "interesting".

At the end of the test a test driver is generated: for each such case a function is defined which executes one of these test cases. The complete set is called from the main program.

The benefit of this procuedure is that the test cases are minimised according to a function's internal structure, while black-box testing may lead to test cases leading to an unbalanced coverage, thereby wasting manual effort.

Each function sets the input variables prior to the test step, calls the FUT, compares the output with the expected output as observed during the previous auto-test run, checks whether the expected or an unexpected exception occurred. Any non-compliance is reported, including composite data.

The test driver has a simple code structure, but may be large, depending on the identified test cases of interest. It can be executed at whole or in part on the intended target system, where the CPU and memory resources may be very limited.

Once the test vectors have been verified by manual inspection, this test driver may be used for regression testing, then automatically identifying potential changes after maintenance or compliance with previous results.

By a DCRTT option the test driver will be instrumented for integration with Cantata++ [11] from IPL.

3 THE TEST ENVIRONMENT

3.1 The Test Tools

The used tool DCRTT [4] is based on activities in auto-testing started in 1993 [5] for testing of Ada source code resulting in the development of the DARTT tool [6]. Based on experience-in-the-large with DARTT [7,8,9] DCRTT, was developed for C. Due to the previous experience and user requests, DCRTT provides much more features now like code-analysis-based testing, operational testing, test case filtering, support for MCDC evaluation (Modified Condition Decision Coverage), classification of filtered test cases like known from the test classification method (CTM, Classification Tree Method) [10].

If a commercial issue comes up, DARTT will also be raised to this support level.

SMARTG, a prototype tool combining random and constraint-based test case selection for generic path set coverage for Java, is a result of efforts on the definition and analysis of the path set coverage criterion. It supports stimulation of static Java methods with primitive types and arrays thereof as parameter types.

The generic coverage criterion used is based on path sets.

This criterion allows the use of a large scope of different coverage rules. Such rules can target typical programming errors, but also parts of the specification can be rewritten as construction rules or even specific path sets. Taking as an example a condition which according to the specification must not occur during execution, an interesting path set could be constructed by constraining the state of the system at some point of execution so that the condition must occur. Non-coverage of this path set would then indicate that the system correctly implements this part of the specification as no inputs can be found which trigger the forbidden condition.

The experiments on auto-testing mainly concentrate on DCRTT, applied to more complex software, while SmartG as a prototype was only applied to some typical examples to demonstrate the feasibilty of this approach.

3.2 Implementation

3.2.1 DCRTT

DCRTT implements the strategies as described in 2.3.4 and 2.3.5 above. Generation of the needed test environment, instrumentation of the code, recording of test vectors and auto-documentation, generation of the test drivers for filtered test cases, and generation of batch-files for control of execution can be handled for any C (ANSI) source code including arbitrary user-defined types and functions.

In most cases, it is sufficient just to provide the set of source files, to start a batch-file (without parameters) and to wait until completion.

The test driver for the filtered test cases is established as independent C program. For each test case a function is built which defines the input values (for every user-defined type), calls the Function-Under-Test, evaluates the test results by comparing the data values by the values obtained in the run for filtering and the observed exceptions for exception type and location (file and line number).

Without instrumentation options an execution time profile can be derived in addition.

The test driver can be compiled with an option for integration with Cantata++ [11]. In this case the Cantata++ run-time environment is activated and all features for test analysis are available, including the features for checking of results and coverage analysis and other reporting features.

Each file and each function respectively is instrumented automatically for recording of block and decision coverage, exception capture, and – optionally - monitoring of data values at each location a variable is used. If a limited type range is defined, exceeding of the range is reported.

In case of operational testing all functions or as desired are instrumented, and the observed properties are recorded in parallel.

3.2.2 SmartG

In the SMARTG tool, random test case generation is applied in the first step, filling coverage gaps using constraint-based techniques. This strategy aims to reduce the cases where the computationally expensive constraint solving mechanisms have to be applied.

The SMARTG tool applies the novel concept of coverage named "path sets". The same “generate-first, filter last”-approach is applied as with DARTT/DCRTT, using the same bucket concept. Test case buckets are defined for each path set, for a specifiable exception classification and for execution timeouts.

3.3 Managing Auto-Testing

For test evaluation a feedback from the code is needed, for recording of test vectors, coverage, exceptions and data values. Of course, this impacts execution time and real-time properties.

If real-time execution is of relevance, a staggered approach needs to be applied minimising the disturbance. This is a matter mostly of operational testing, but not of module testing.

For testability reasons always a time slot should be reserved for overhead induced by test instrumentation, otherwise the software would not be testable. Then it should be possible to activate slicewise the instrumentation, for one function only or a subset of functions as compliant with the execution time margin.

If full instrumentation is needed, a capture-replay approach is proposed: recording of input data under operational conditions in a first step, having activated the instrumentation only for recording of these data. Then in a second step these data are replayed and the system is executed with full instrumentation, but time-triggered by the events of data provision.

This approach is feasible for "synchronous systems" which are broadly implemented in the embedded domain. In this case a clock drives execution, but the results do not really depend on the period (provided it is longer than the "worst-case execution time"). Even, it the clock is stopped and then running again, there is no impact. It is even possible, to run different execution chains and to model time jitter between the clocks.

The feature of capture-replay is currently in preparation for DCRTT.

3.4 Test Strategies

It is expected that interesting test cases cannot be collected by one of the auto-testing modes only. As will be discussed later, missing context information on the valid data range may not allow to optimise test generation towards high coverage in lattice- and/or random-based mode. Therefore a combination of several test strategies is suggested.

In case of few context information lattice- and random-based mode may only address robustness of the software under test, when many test cases outside the intended data range are produced. In this case these test modes can efficiently be complemented by operational testing.

In this test mode all the features for recording of coverage and exceptions are kept like in the other test modes, and the overall coverage can be cumulated from one test mode to the next one.

Fig. 2-1 explains this strategy.

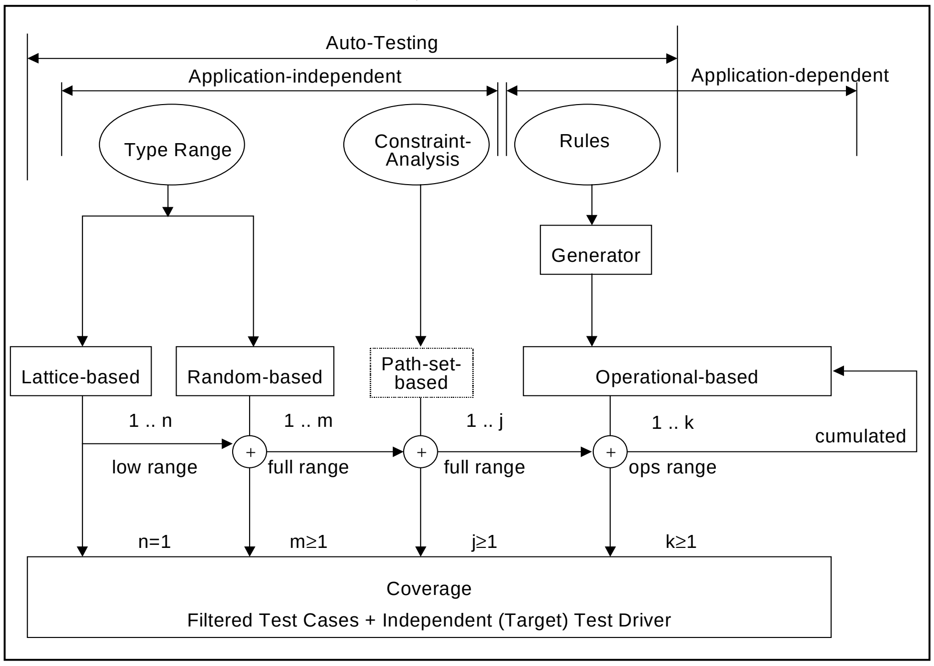


Fig. 3-1: Suggested Test Strategy

Between all these steps the contents of a basket can be inherited, so that only additional values are collected in a following step. However, if desired a basket may also be emptied prior to test execution, e.g. to investigate which type of test cases are identified in a certain test mode.

Firstly, lattice-based testing enriched by additional literals is performed, followed by random-based testing extended by literals, too. Finally, the software-under-test is executed under operational conditions and instrumentation of the code.

The reasons for this sequence are:

Lattice-based testing tends to prefer data from the lower range. E.g. if the maximum volume of the basket is one test case only, it is likely that it will lie in the lower range, when the first observed test case is chosen. As it is unknown whether another test case will follow at all, the first one cannot be dropped. If more than one test case is allowed, it is likely to get a cluster in the lower range. Therefore it is recommended to run lattice-based testing with a small basket size only. But it should be the "starter" because it is more likely that in this mode more test cases can be found (see the conclusions below).

Test cases are better distributed across the input domain for random-based testing. Therefore in this mode a larger basket is recommended, to capture more test cases over the full range.

This is also the case for testing under operational conditions, though the effective range may be smaller than in case of lattice- and random-based testing.

Constraint-based testing for path set coverage should be applied after lattice- and random-based test modes. It may even be moved after the end of operational testing. There is a simple reason for this sequence: constraint-solving is time consuming, therefore it should be limited to such code only, which cannot be reached by the other test modes. Current experience shows that a coverage of 70% percent can easily be achieved by these test modes. Consequently, the time consuming processing only needs to be applied to 1/3 of the software, or possibly less.

In addition, constraint-based testing may introduce an unwanted bias due to the direct specification of path sets brought in from the outside.

Operational testing is also of interest – even if a sufficient number of interesting test cases would have been collected already – to monitor coverage under such conditions – possibly during separate test execution – and the actual range of the variables.

3.5 Some Issues of Practical Auto-Testing

By analysis of the C base types and user-defined types it is rather straightforward (but still challenging) to generate input data, except for one case: passing of pointers.

If a pointer is passed to a function, concrete data have to be declared outside the function. However, in C any information is missing about the size of the pointer, unless an intelligent parsing of the code is done, which can be rather complex and even infeasible in practice in a general manner.

Therefore DCRTT needs to make an assumption on the size, based on user-provided information. DCRTT resolves this ambiguity by allocating an array to each such variable thereby removing one level of pointer or array from the type definition. This happens recursively for each such level. As DCRTT is fully controlling the test environment, all the code remains consistent.

In case of operational testing DCRTT cannot control creation of the environment, data even may be allocated dynamically with varying size. Therefore DCRTT makes the assumption that the size of the memory allocated to such a pointer only amounts to one element of the related type. This impacts only recording of data values.

4 SOFTWARE UNDER TEST

To exploit the test strategies DCRTT and SmartG have been applied to a set of software packages.

Any later conclusions on test results do only relate to the reference set of tested functions.

The tests have been executed on two compilers on an XP-platform: VC++ V7/.NET and gcc 3.2.3.

4.1 DCRTT

For exploitation of the test modes the following software packages have been used:

- A specific set of functions used for DCRTT testing, targeting general, specific and critical aspects of the C language w.r.t. generation of an auto-testing environment, in particular regarding the user-defined types. Currently, there is a set of about 150 functions which are provided with the DCRTT installation. When the test were executed, the number of test functions amounted to 142.
- Two program packages of “foreign” source code from the Open-Source domain (oSIP and flex), considered as sufficiently representative and complex software. In particular, the nature of this software required full scanning of a large set of included compiler h-files, including about 4000 function and type definitions, each, and up to about additional 100 KLOC.

  Open source software has been considered because everybody may access this software and rerun the tests.

4.2 SmartG

SMARTG comes also with its own set of benchmark functions which are specifically addressing path set coverage based on a typical rule set. Additionally a set of methods implementing algorithms from the domain of computational geometry were tested.

4.3 Overview on Tested Functions

The oSIP package (GNU open software for the Session Initiation Protocol) [12] has been selected because it was also used in [1].

Flex (Fast LEXical analyser generator) [13].has been selected because it is considered as a representative complex application, for which rule-based auto-test generation can be applied.

Tab. 4-1 shows the some figures characterising the size and the complexity of the test packages. "LOC" means "source lines without comment and blank lines", a block is either enclosed by "{ }" or a sequencing of statements following a branching condition like "if", "switch", "case", loop constructs. A "decision item" is a simple expression evaluating to a boolean value in a branching condition.

	Functions	LOC	Blocks	Decisions
DCRTT	142	3862	865	938
flex	189	12452	2397	2871
oSIP	655	19368	3402	5227

Tab. 4-1: Test Package Characteristics

A summary feedback from the DCRTT tests is given in Fig. 4-1 and Fig. 4-2.

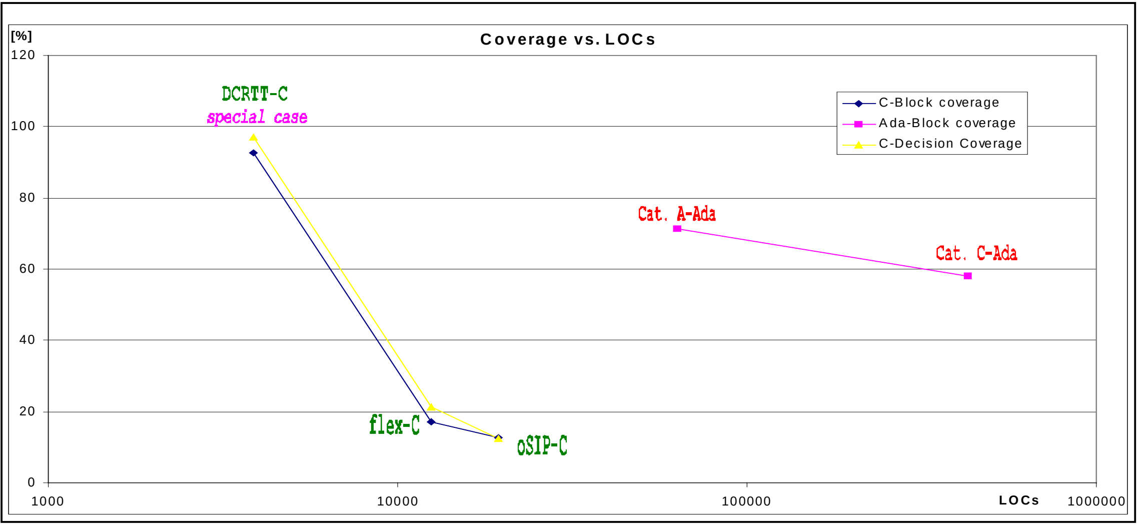


Fig. 4-1: Coverage vs. Software Categories

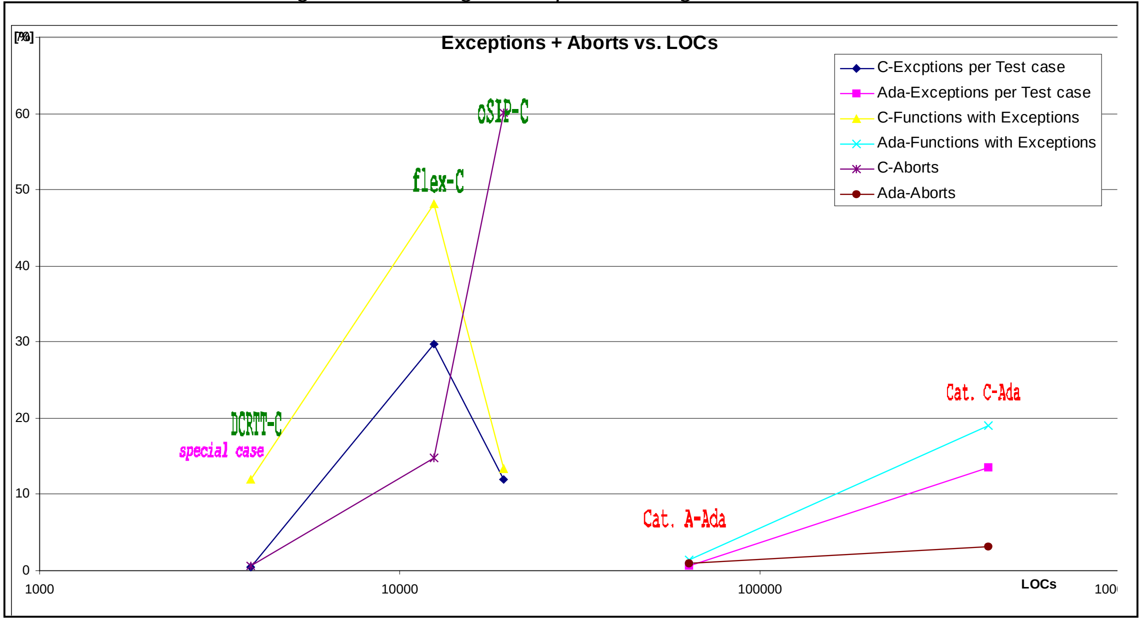


Fig. 4-2: Locks, Aborts and Exceptions vs. Software Categories

For comparison, these figures also show results obtained during previous DARTT tests [8,9]. Fig. 4-1 shows for C and Ada software the coverage achieved by lattice-based testing vs. LOC.

The results from DCRTT test suite yielded high coverage. These tests may be considered as a singular case because they specifically address the aspects of auto-testing. The reason why 100% coverage was not achieved is simply that functions are included which address unreachable code and exceptions.

For open source software only low coverage was achieved, because context information could not be found in the code which can guide auto-testing to find the test cases of interest. It is a weakness of C that it does not enforce provision of such information like on type ranges.

This is different for Ada and consequently higher coverage was achieved for two different packages. It is also obvious that for high integrity software as is "Cat. A" coverage is higher than for software of lower category "Cat. C". The more defensive character of Cat. A software and the more rigorous standards seem to enforce provision of information needed to tune auto-test generation.

Fig. 4-2 shows the number of observed deadlocks, livelocks and aborts during lattice-based testing. It is obvious that the more context information on valid data ranges can be found in the code, the less anomalies were observed. In case of oSIP more than 50% of the functions were aborted due to such anomalies. Therefore it was decided not to evaluate this software in more detail.

The decrease of number of functions with exceptions and of the percentage of exceptions is a consequence of the high abort rate, and does not imply any conclusion on a better programming style preventing exceptions.

5 DCRTT TEST RESULTS

The test results have been analysed for coverage figures, observed exceptions and platform dependencies causing differences in test results,

5.1 Coverage

5.1.1 Module Testing

5.1.1.1DCRTT Test Functions

The set of functions consists of 865 blocks and 938 decisions (see Tab. 4-1). The following table Tab. 5-1 shows the results of lattice- and random-based testing extended by code-analysis-based testing.

For decision coverage two further rows provide information on the coverage of the “true” and “false” branches of a logical item. A value less than 100% indicates that not in all cases the “THEN” or “ELSE” branch was reached.

Latticed-based testing gives slightly better results than random-based testing. It can be expected that worse results will be achieved when the literal test cases are omitted found by code analysis.

DCRTT Test Suite 142 functions	**Coverage / %**			
	Lattice		Random	
Coverage Type	VC++	gcc	VC++	gcc
Block	91.10	92.6	85.20	85.20
Decision	96.70	97.10	91.90	91.90
true	90.74	93.20	83.53	83.53
false	96.14	96.16	94.90	94.90

Tab. 5-1: Coverage Figures for DCRTT Test Suite

It was not expected that the figures for random testing nearly reach such of lattice-based testing. An analysis showed that due to use of literals (found from code analysis) also the cases “a==b” were hit which is very unlikely for random testing. As “0” is always considered as a specific test case, it helped to meet “a==b”. Therefore additional conditions were added like “a==b && a!=0” in order not to make it too simple to fulfill such conditions.

The “decision”-row shows the percentage that a block following a decision could be entered. The “TRUE” and “FALSE” rows give the figures on how often a condition evaluated to TRUE or FALSE. As a decision may be based on the evaluation of several conditions and due to “short circuit code” evaluation of conditions may be dropped when the result will not be changed, the coverage of all conditions is lower than the coverage of the “parent expressions”.

Full MC/DC (Modified Conditional Decision Coverage) is achieved when for TRUE and FALSE 100% are reached.

It needs to be mentioned that a number of blocks are included which never can be reached because such cases were added for demonstration purposes.

Also, exceptions – when occurring systematically – prevent execution of the following blocks, thereby decreasing coverage. Consequently, comparison of coverage figures requires absence of exceptions.

For this set of test functions it can be assumed that 100% coverage of the blocks and decisions which can be reached has been achieved. In consequence, the fact that 100% could not be reached indicates (built-in) problems in the code, either dead code or exceptions.

5.1.1.2 oSIP

The results obtained for the oSIP software are shown in Tab. 5-2. Tests have only be executed for the gcc compiler and for lattice-based testing. After evaluation of these first results and recognition of the high number of anomalies further evaluation was dropped.

As already discussed the figures may not be considered as reliable due to potential corruption of coverage information and high abort rate, because in case of aborts no coverage information can be collected.

oSIP 655 functions	Coverage / %			
	Lattice		Random	
Coverage Type	VC++	gcc	VC++	gcc
Block		12.64 ??		
Decision		12.42 ??		
true		34.52 ??		
false		85.98 ??		

Tab. 5-2: Coverage Figures for oSIP

5.1.1.3flex

The results for tests of the flex software are shown in Tab. 5-3.

In this table the dependencies of coverage figures on compilers are evaluated for each test mode. An explanation of the higher values for gcc is, that less exceptions were produced (see sect. 0), which resulted in more executed blocks and decisions because the control flow was not interrupted. As will be discussed later, this does not imply an advantage for gcc, because the results of execution may be wrong.

Also, it has to be mentioned that the crashes of a FUT may cause corruption of coverage figures, maintained in memory. This was recognised by values which seemd to be too low. Further investigation showed that the run-time system may be corrupted by the FUT such that the contents cannot be saved to file. This causes loss of cumulated coverage up to the time of corruption.

Measures were undertaken to prevent loss of such information, but the more protections were added, the more complex remaining conditions were identified. This process of improvement of self-protection against corruption of the run-time system by the FUT due to invalid operations is not yet finished. This is the reason why for the combination of lattice- and random-based testing no valid figures can be presented for VC++.

flex 189 functions	Coverage / %			
	Lattice		Random	
Coverage Type	VC++	gcc	VC++	gcc
Block	15.28	17.15	15.20	16.44
Decision	16.75	21.25	18.01	19.33
true	56.97	58.85	54.16	55.86
false	86.49	84.26	87.23	87.57

Tab. 5-3: Coverage Figures for flex/ Compiler View

This is not the only point where much effort was and still is required to run the tests while the FUT does behave unpredictable. Similar problems arise when faults are injected by DCRTT, e.g. NULL pointers. Then the test environment must not crash itself, e.g. when collecting and storing test results implying invalid data. However, compared to the problem described this issue is rather easy to solve from a principal point of view.

Tab. 5-4 shows the same results as Tab. 5-3, but reorders the results so that a direct comparison is possible between test modes.

Moreover, results are shown for a combination of lattice- and random-based testing: Firstly, coverage figures were derived for lattice-based testing.

Then these figures were taken as a starting point for random-based testing, adding now only blocks which were not already covered by lattice-based testing.

Test Mode	**Corrupted Coverage Files VC++**		
	lattice	random	combined
block	1	1	0
decision	1	71	29

Unfortunately, for VC++ a number of corruptions of the coverage files occurred yielding a low decision coverage for random case in one test run.

flex 189 functions	Coverage / %					
	VC++			gcc		
Coverage Type	Lattice	Random	Lattice + Random	Lattice	Random	Lattice + Random
Block	15.32	15.20	16.08	17.15	16.44	18.57
Decision	16.82	(10.55) 18.01	18.15	21.25	19.33	22.78
true	57.76	(45.88) 54.16	60.08	58.85	55.86	60.40
false	85.71	(86.80) 87.23	85.99	84.26	87.57	84.86

Tab. 5-4: Coverage Figures for flex / Test Mode View

The last column "Lattice + Random" shows for gcc that by this combination of test modes the coverage increases by about 10% of the value achieved in a single test mode. This result also implies that in this case both test modes are nearly equivalent, though not fully. But even the small difference justifies a combination of both test modes in order to gain a higher coverage and more filtered test cases at the end.

5.1.1.4Conclusions on Module Testing

Compared to the coverage figures of the DCRTT test suite the corresponding figures for flex are rather low. A potential explanation was already given above.

In fact, if context information needed to guide auto-testing cannot be retrieved from the source code automatically, it is most likely that auto-testing will mainly perform robustness testing and fault injection.

In order to benefit from auto-testing a high coverage is needed, however, to derive a high number of filtered test cases and generation of the test driver for test repetition.

At this point the purpose of auto-testing has to be reconsidered and a strategy has to be identified which still allows to meets this goal.

Most of the time for getting a test environment for test repetition and regression testing is needed to define test cases of interest and to build the test driver.

This goal can still be reached by operational testing though the test cases are not generated automatically, in general. However, the concept of identification of interesting test cases and auto-generation of the test driver can still be applied to operational testing and its combination with module testing. Also, in this case the test driver can be generated with little effort.

For what is still not covered, constraint analysis shall provide the needed inputs.

5.1.2 Operational and Rule-Based Testing

For module testing only one function out of a set of functions is considered for testing. Usually, this implies that certain values may not have been initialised, which leads to poor conditions for proper operation.

The better a function protects itself against invalid conditions ("defensive programming style"), the better are the results which can be obtained during module testing by auto-test generation.

For operational testing this potential problem does not exist, because the main function itself should care about proper initialisation. Therefore an increased coverage can be expected from operational testing, if the functions provide poor context information for lattice- and random-based test modes.

For flex operational testing was performed in addition to module testing in order to evaluate the contribution from this test mode to coverage. A formal specification of inputs is available for flex which lead to the decision to generate the inputs for operational testing automatically. If no such precise syntax and semantics is available, the program under test has to be stimulated manually in the way the program expects inputs, possibly supported by a test management tool.

The input can be built from 75 rules driving the generation of a C program by which a source file can be parsed by the provided rules. The parser generator (yacc) adds another internal rule, so that in total 76 rules exist.

A test generator was built which provides a number of application rules composed of the pre-defined 75 basic flex rules. The generation of these rules is based on pre-defined configuration values, limits and probabilities which may be varied to get a variety of rule sets for stimulation (Tab. 5-5).

```
30 max. number of rules per file
 2 max. number of sequences with {}
10 max. number of elements (operators, constants)
10 max. string length
10 max. length of a character class
10 % to generate an empty string
 8 % to generate a subsection in section 2
10 % to insert an end-of-file rule
20 % to insert a start-of-line marker into a rule
40 % to generate a context dependent rule
10 % to generate an end-of-line rule ($)
40 % to generate a closure (*,+,?) rule
-1  maximum depth for recursion
```

Tab. 5-5: Input Parameters for Rule Generation

30 input files were generated from 8 sets of configuration parameters. For seven of these files flex did not terminate. The remaining 23 files were used to stimulate flex. The coverage of the flex basic rules and of the C code was measured, and the coverage figures were cumulated from run to run.

For four different combinations of test modes the cumulated coverage was measured:

- operational testing only, starting with zero coverage,
- operational testing starting with coverage of lattice-based testing,
- operational testing starting with coverage of random-based testing,
- operational testing starting with coverage of lattice-based testing, followeded by random-based testing.

After the 23 flex runs a total rule coverage of 94.74% was reached. Only 4 of the 76 rules were not covered:

```
rule  1: "$accept : goal"
rule 27: "flexrule : error"
rule 30: "scon : '<' '*' '>'"
rule 34: "namelist2 : error"
```

Rule 1 is the internally added, organisational rule which cannot be reached from outside. Two error conditions were not covered and one rule related to start conditions, which was not, but could be covered by the generator. Fig. 5-1 shows the number of executions of the basic rules

Tab. 5-6 gives an overview on the achieved coverage figures for two sets of functions. The reason to present the figures for 189 and 161 functions is: the module tests were executed on the set of files which come with the flex download package. During operational testing it was recognised that three files are not needed for linking at all: scan.c (redundant with initscan.c), libmain.c and libyywrap. As the blocks and decisions items were counted for the 189 functions, 28 functions could never be executed. This lead to a significant lower coverage (as presented during the conference). Therefore the tests have been repeated with the minimum set of 161 functions.

To give a complete overview and to allow comparison with the results from pure module testing also these values are added.

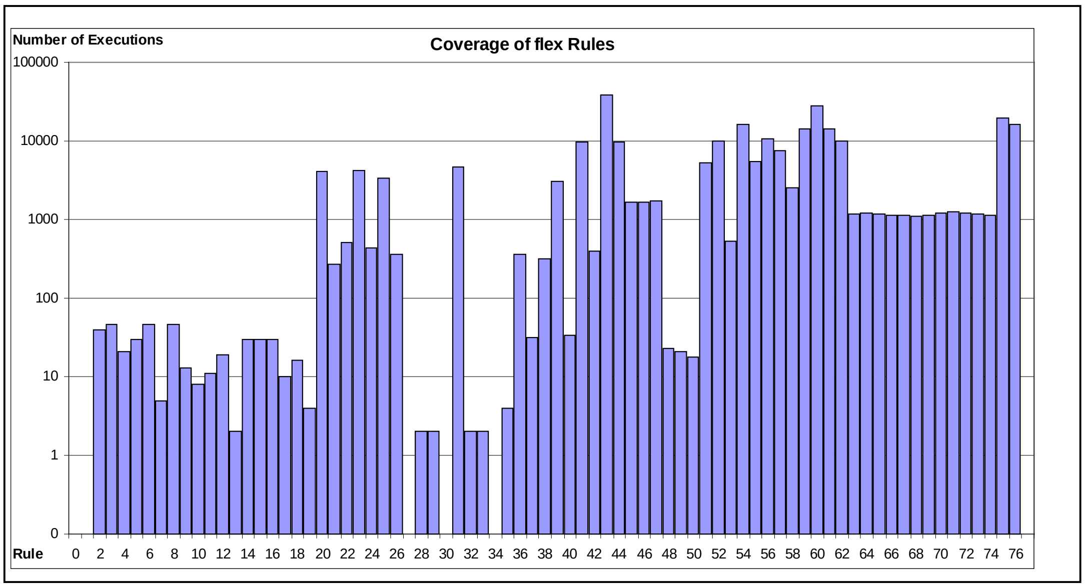


Fig. 5-1: Execution Profile of flex Rules

flex gcc Rule coverage max. = 94.74%	**Coverage / %** 189 functions		**Coverage / %** 161 functions	
Test Mode	Block	Decision	Block	Decision
Lattice	17.2	21.3	19.3	23.1
Random	16.5	19.3	19.5	24.5
Lattice + Rnd	18.6	22.8	21.1	24.8
Operational Mode (OM) max.	29.58	42.95	40.8	55.2
Latt + OM max.	37.46	49.43	48.0	60.7
Rnd + OM max.	37.55	49.32	48.3	60.4
Latt + Rnd + OM max.	38.42	49.57	49.4	60.7
OM cumulated	38.82	49.84	50.9	62.7
Latt + OM cumulated	45.64	55.59	56.8	67.6
Rnd + OM cumulated	45.72	55.77	57.1	67.4
Latt + Rnd + OM cumulated	46.43	55.73	58.0	67.8

Tab. 5-6: Coverage Figures for Test Mode Combination

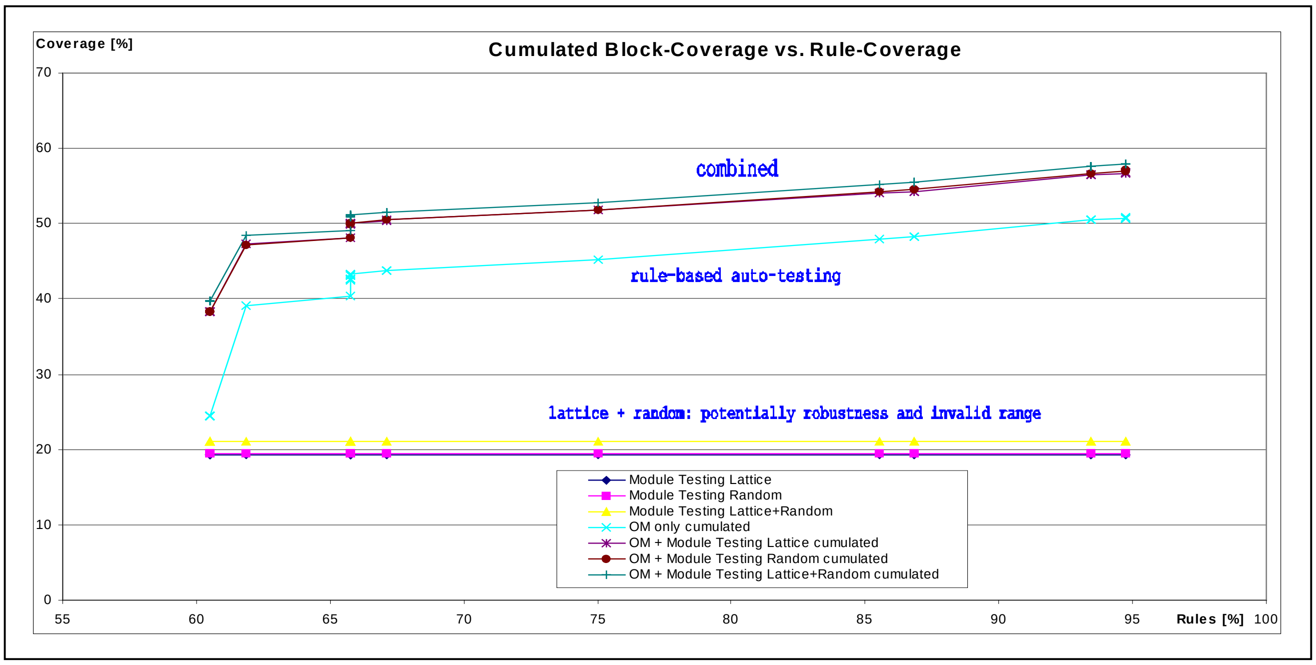


Fig. 5-2: flex Code Coverage vs. Rule Coverage

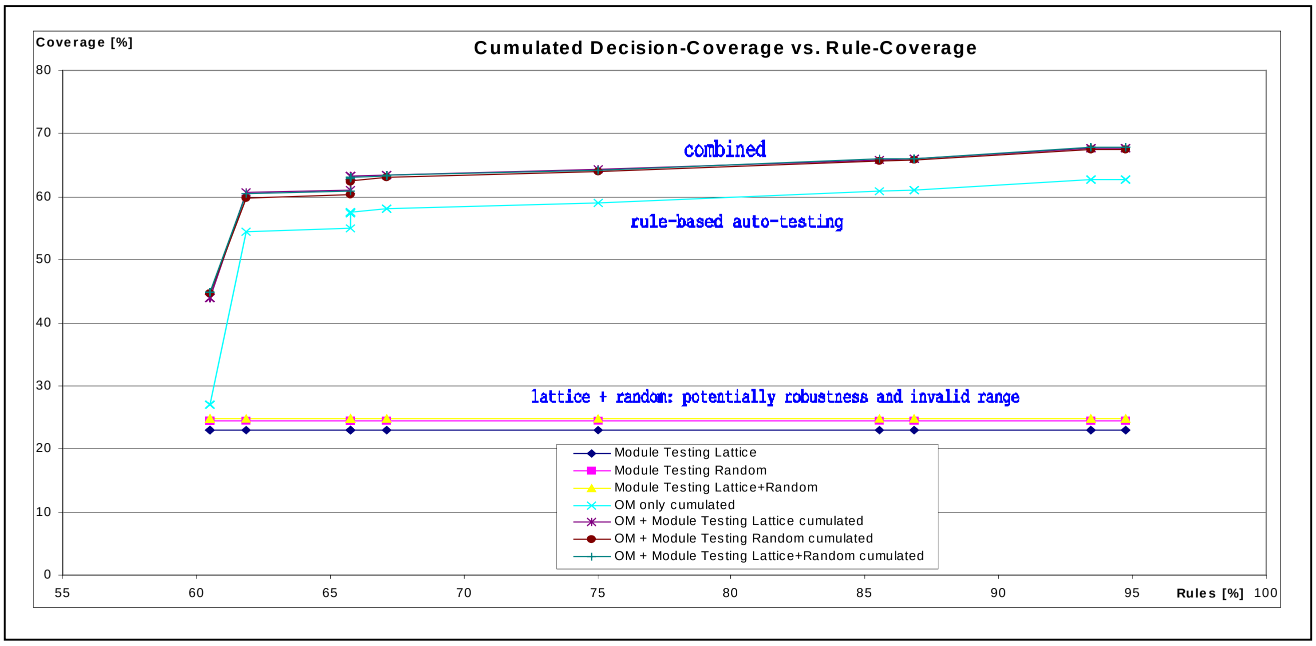


Fig. 5-3: flex Decision Coverage vs. Rule Coverage

Then for each combination the maximum coverage values are given, observed when executing flex for one of the 23 input files. Finally, the cumulated coverage figures are shown for all combinations.

Fig. 5-2and Fig. 5-3 show the code and decision coverage vs. for rule coverage.

From the graphical figures, it is obvious that the coverage figures of module testing are rather complementary with operational testing. This supports the hypothesis that for less defensive programming style module testing focuses on robustness testing, which is complementary to normal operations. For higher code and decision coverage (and higher rule coverage) the overlap in covered blocks and decisions increases, which is just a consequence of the higher coverage. The higher the coverage of two test modes, the higher is the probability to overlap. This conclusion also implies that at higher rule coverage the error handling – or the area of robustness testing – is better covered.

Though the rule coverage is close to 100%, the code coverage is not: about 1/3 of the code is still not covered. An explanation for this result cannot be given at short hand, more detailed evaluation is needed.

The following Tab. 5-7 gives an overview on the percentage of functions for which 100% block or decision coverage or both was achieved. In addition to block coverage the coverage of checkpoints was also recorded. A check point is either a statement where a branch of the control flow may occur (if, switch, loops, break, exit, return, goto) or the end of a block. Checkpoint coverage less than 100% at block coverage of 100% implies that this block was left prior to its end, either by an exception or e.g. a return. Checkpoint coverage of 100% implies 100% statement coverage. In addition, the percentage of functions which have 0% coverage for all criteria is given, i.e. for functions never called. In fact, only one function, yy_try_NUL_trans of file initscanc (or scan.c), was never called.

Full statement and MC/DC coverage was achieved for nearly 50% of functions. Most of the functions fulfilling the 100% criterion are of lower complexity and have (roughly) 10 blocks or decision items at most. However, for yyparse, possibly the most complex function with 285 blocks and 571 decision items still reaches about 75% for all coverage criteria.

flex was operated in default mode, i.e. no additional command line parameters were provided other than the rule input file. Using other parameters probably will raise the coverage. However, the deadline for the conference and finalisation of this paper did not allow further evaluation of these options.

Coverage / %				**Functions / %**
Block	Stmt	Decision	MC/DC	
0	0	0	0	0.62
100				63.35
100	100			52.80
		100		67.70
		100	100	53.42
100		100		57.14
100	100	100	100	47.20

Tab. 5-7: Functions with 100% or Zero Coverage

5.2 Exceptions

Raising and capture of exceptions in case of invalid data is very efficient to identify critical code, because such weakness can be easily reported without requiring any manual evaluation effort. Exceptions may either be raised by an application itself or by the platform (compiler, OS; processor) in order to flag a problem. Consequently, if an exception is foreseen or is not raised a problem will be not recognised.

In the report files on exceptions no information on user defined exceptions was found. The only exceptions raised were initiated by the compiler based on hardware support (e.g. access violation, numerical problem).

Surprisingly, the number of raised exceptions is platform/compiler-dependent.

5.2.1 Platform Dependencies

The number of exceptions raised during the tests have been recorded together with locations in the source code. As two different compilers were used (on top of the same OS and processor type) differences between compilers were detected. The gcc does not flag most of the Floating-Point Exceptions (FPE), only "division by zero" is raised. The reason could not be identified so far. VC++ does raise all floating point exceptions. Tab. 5-8 shows the dependencies for the DCRTT test suite.

DCRTT Test Suite 142 functions	**VC++**		**gcc**	
	lattice	random	lattice	random
Exceptions				
expected	79	60	30	32
occurred	79	60	30	32
non-compl.	3	3	0	0
Functions with Exceptions	27	27	17	17
Filtered Tests	769	626	736	600

Tab. 5-8: Platform Dependency of Exceptions for DCRTT Test Suite

As the test input vectors differ for lattice- and random-based testing, a different number of exceptions was observed for both test modes. A comparison of the observed exceptions yielded that gcc does not support FPEs, which explains the big difference between VC++ and gcc. A further conclusion is that this also impacts the number of exceptions in lattice- and random-based test modes: lattice-based testing systematically covers values close to boundaries of a type range, leading to a higher probability of FPEs.

For each case (compiler, test mode) three figures are shown:

- expected exceptions
  These are exceptions recorded during auto-testing, enforcing generation of a "filtered test case".
- occurred exceptions
  These are exceptions observed during execution of the test driver, i.e. of the filtered test cases.
- non-compliances

These are cases where the equivalent of an expected exception was not observed during execution of the test driver. Either it was not raised, another exception type was raised, or it was raised at another location. Some non-determinism may occur for "access violation" exceptions, because its occurrence depends on the actual memory configuration or pointer contents.

Such non-compliances are a strong indication for non-deterministic results and weakness of code. Their occurrence also depends on the compiler. The gcc is less accurate from this point of view.

Clearly, occurrence of exceptions also impacts identification of filtered test cases, though the differences between the test modes cannot be explained only by this dependency.

Tab. 5-9 gives the results for flex software. Though no deep manual code analysis was performed on flex source code, it is likely that it does not use many floating-point operations. This explains the small difference between the two test modes. The higher number in case of lattice-based testing may be caused – as a hypothesis – by a higher coverage of invalid data, causing e.g. an index-out-of-range condition and an access violation.

flex 189 functions	**VC++**		**gcc**	
	lattice	random	lattice	random
Exceptions				
expected	179	154	177	146
occurred	124	110	135	121
non-compl.	101	101	47	39
Functions with Exceptions	101	191	91	93
Filtered Tests	359	328	365	313

Tab. 5-9: Platform Dependency of Exceptions for flex

Tab. 5-10 shows the observations regarding anomalous program termination, categorised by locks (deadlocks, livelocks) and aborts. In case of an abort DCRTT cannot take control after such a termination condition. An abort is something different from an exception and program-enforced exit, because DCRTT captures exceptions and exits during the tests, so that the test can be continued.

A lock cannot be detected by the DCRTT test program itself. It is recognised by a monitoring program which launches the test program and enforces termination when a lock is detected. Therefore data of the test program cannot be saved at termination. Similarly, it happens for an abort. As reason for such an abort stack overflow has been observed, but not all such aborts were deeply investigated.

The occurrences of anomalous termination are also an indicator for the quality of the software: the less the figure, the more defensive is the programming style.

For the DCRTT test suite there is one intended live lock (a for-ever-loop).

For flex a livelock was observed for function add_action in file misc.c. A global variable was not in the correct range (due to missing initialisation) and caused the algorithm to fail in an endless loop.

Locks + Aborts	**VC++**		**gcc**	
	lattice	random	lattice	random
DCRTT	intended	intended	intended	intended
#	(1)	(1)	(1)	(1)
%	-	-	-	-
flex	28+10	17+12	14+14	12+16
#	38	29	28	28
%	20.12	15.35	14.81	14.81
oSIP			25+369	
#			394	
%			60.14	

Tab. 5-10: Platform Dependencies of Anomalies

VC++ seems to be more sensitive for anomalies, but a general conclusion is not possible, because the occurrence is non-deterministic. While the summary figures are close together (except the one for VC++/lattice), the dedicated figures differ significantly, possibly statistically – more or less, depending on the actual configuration of the run-time environment.

As a high number of anomalies (60% of module tests) was observed, this package was dropped for further test runs and evaluation.

5.2.2 Conclusions on Exception and Anomaly Occurrence

Having executed a large number of tests in Ada and C, the following conclusions from an organisational and software-engineering point of view can be drawn on occurrence of exceptions:

(a) the lower the number of non-anticipated exceptions, the more defensive is the programming style,

(b) the higher the number of exceptions, the higher is the manual effort for verification

(a) is a consequence of a discrepancy between a subprogram's specification ("prototype") and its body, the implementation. E.g. if the prototype defines a parameter x of full range of "int", and the parameter is used in an expression like "1/x" in the body, then a "division by zero" exception will be raised, for sure during auto-testing (as "zero" is an inherent special test case). A "defensive programming style" would restrict the type range to "positive" which is rather easy to do in Ada.

In case of C it is more complex because the compiler does not support consistency checks like in Ada and accurate definition of type ranges . In order to increase the density of test cases in the valid range a hint on the intended valid range is required, which is supported by DCRTT. However, in order to identify potential inconsistencies, "robustness testing" needs to consider the full range. Consequently, this requires two steps of testing: a detailed exploration of the valid range, and a check over the full range for assessment of robustness.

(b) is based on the following consideration: if a non-anticipated exception occurs during auto-testing, the software engineers have to prove that this cannot happen under operational conditions. However, such a proof requires a – recurring - detailed manual analysis of the whole system to ensure that it really cannot happen, while it would be easily a matter of an Ada compiler to confirm compliance of the parameter's type through the whole system at zero human effort.

In C the only indication for such a compliance is the absence of exceptions during testing of the given full type range, which is not fully deterministic, however. But in a defensive programming style also in C invalid data can be identified easily when checks are added (manually), which can also be logically removed by the pre-processor, if really needed.

Consequently, auto-testing strongly helps to identify potential weakness of an implementation and to save money, if applied early enough in the lifecycle.

5.3 Test Case Reduction and Filtering

Test case filtering aims to find the "test cases of interest" by coverage criteria. From a large set of test data stimulating the FUT during module testing and the system-under-test during operational testing a reduced set of test cases is identified. For each such tests case a function is created in a test driver which prepares the test environment for each of the "interesting cases" (*Fig. 5-4*). This driver can be executed completely independent from the previous auto-testing environment. An option exist, to automatically configure the test driver for integration with Cantata++. Tab. 5-11 shows the results of test case filtering for the DCRTT test suite.

The total number of samples were executed ("test data generation and filtering") within about 1 hour on a laptop 1.6 GHz mobile and a desktop 3.1 GHz.

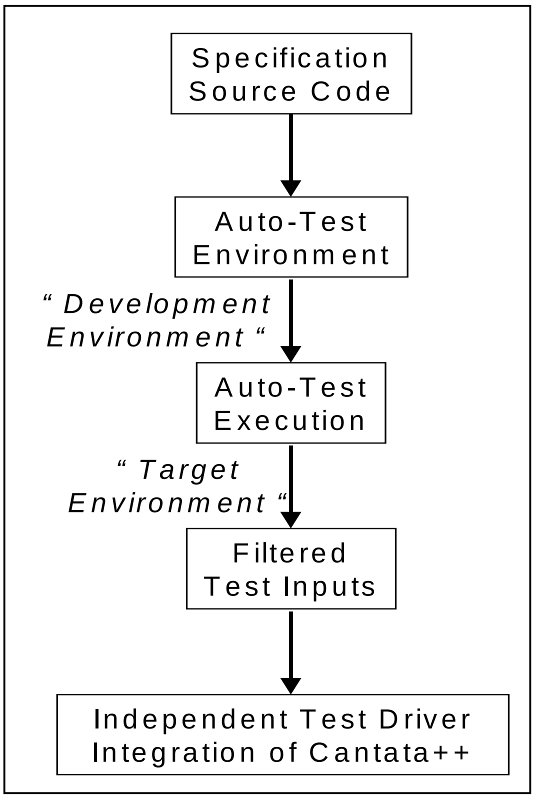

Specification Source Code
Auto-Test Environment
" Development Environment "
Auto-Test Execution
" Target Environment "
Filtered Test Inputs
Independent Test Driver Integration of Cantata++

Fig. 5-4: Generation of Independent Test Driver

The difference in total number of samples between lattice- and random-based testing occurs because for random testing exactly the user-defined number of samples can be taken, while for lattice-based testing an integer number for each stimulated test parameter must be derived by logarithmic calculation from a given number, which together spawn a product of an integer number, which usually is larger than the planned number.

Roughly, a reduction of test cases by a factor of 700 is achieved. This is still a high number for manual evaluation. However, this software packages consists of nearly 900 blocks and decision items each, which makes it reasonable to end up with an optimised / minimised set of about 700 test cases.

The lower number of filtered test cases for random-based testing directly relates to the lower coverage which was achieved for this test mode (see Tab. 5-1). From this point of view lattice-based testing must be considered as the better choice, at least for this software package. A similar conclusion can be made for flex. However; for a general conclusion more data need to be collected.

In principle, the conclusion could also be that random testing would generate more efficiently "interesting test cases", so that the same coverage could be achieved with less filtered test cases. Although currently no specific evaluation was done supporting or rejecting this hypothesis, it seems that the lower number of filtered test cases is a matter of the lower coverage.

As a future issue is the optimisation of filtered test cases, i.e. to investigate if some of these test cases are already covered by others.

Execution of the generated test driver with Cantata++ and evaluation of coverage by its capabilities yielded the same coverage results as before during "test data generation and filtering", getting the advantage of additional evaluation capabilities by Cantata++.

In fact, DCRTT test filtering acts as a "writer of Cantata++ test cases" which otherwise need to be identified manually.

Test Cases DCRTT	**VC++**		**gcc**	
	lattice	random	lattice	random
Total Samples	552339	428318	552342	428318
Filtered	769	626	736	600

Tab. 5-11: Test Case Reduction for DCRTT Test Suite

Tab. 5-12 shows the equivalent results for flex. It is important to mention that these are the test cases derived during execution of the module testing modes (lattice/random) corresponding to the coverage figures of Tab. 5-3and Tab. 5-4. The lower number of test cases is a consequence of the low coverage.

Test Cases flex	**VC++**		**gcc**	
	lattice	random	lattice	random
Total Samples	525660	492489	533070	487122
Filtered	359	328	365	313

Tab. 5-12: Test Case Reduction for DCRTT (MT only)

Fig. 6-1 shows the distribution of filtered test cases for function bldtbl in file tblcmp.c of flex software. On the vertical axis the identified test cases are identified on the left side, and anomalies are reported on the right side, if occurred. On the horizontal axis the values of the stimulated parameters are shown, divided into 7 classes: invalid low, very low, low, medium, high, very high, invalid high related to the type range. For enumeration types the literals are shown, if less than 6 literals are defined. For pointers (incl. strings) the case "NULL" is indicated in addition, and for strings an empty string. At the bottom the coverage figures are shown as achieved for the FUT.

6 SMARTG TEST RESULTS

6.1.1 Constraint-/Path-set-based Testing

SmartG is currently a prototype for constraint-based test data generation of Java code. The applied algorithms shall be later integrated into DCRTT, and possibly DARTT.

Path sets were derived for four example Java methods:

- calculation of the greatest common divisor of two positive integral numbers (GCD),
- determination of whether two rectangles intersect,
- determination of whether one rectangle is contained in another,
- determination of whether a point is contained in a rectangle.

While these functions may seem simple from a functional point-of-view, their implementations pose important challenges for random test data generation and constraint-based testing.

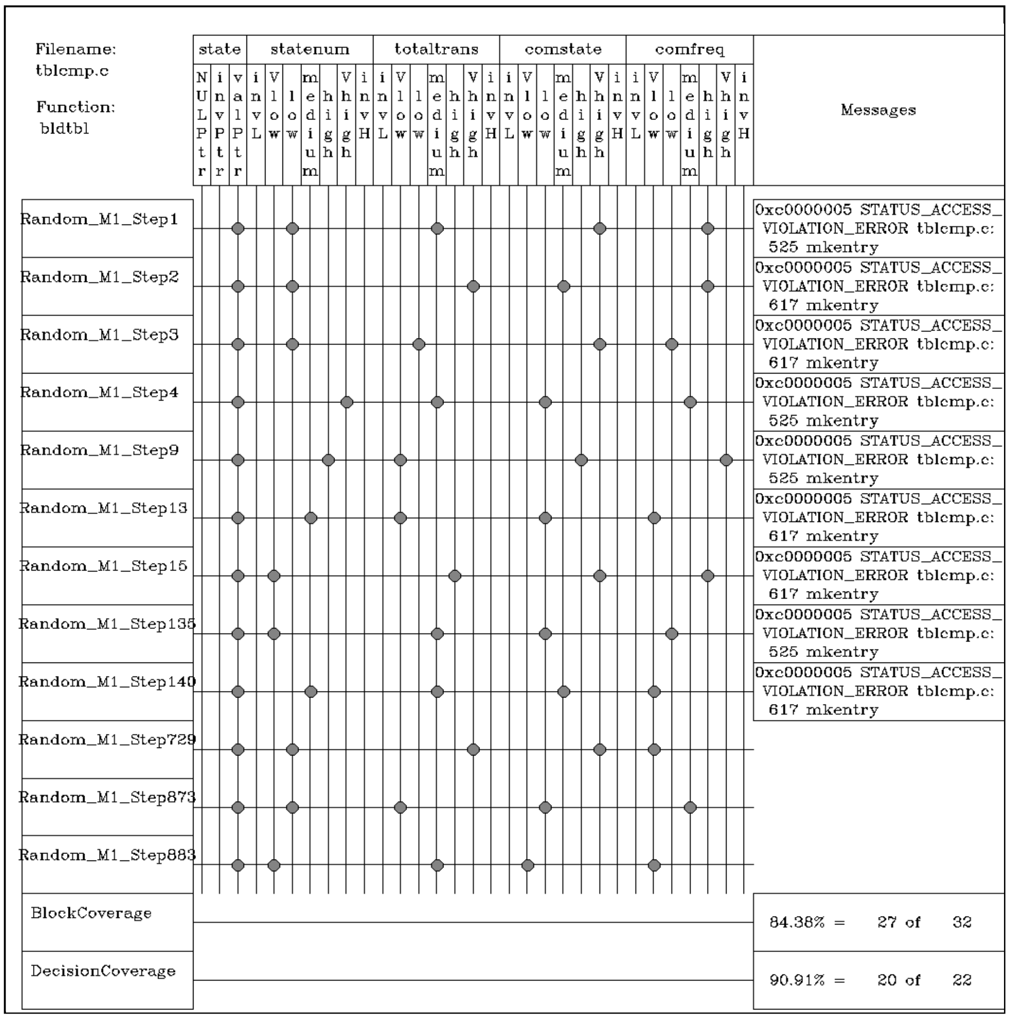
An implementation of Euclid's algorithm for determination of the greatest common divisor will typically consist of a loop iteratively subtracting one of the integers from the other until both are equal, based on the property `gcd(a,b)=gcd(a-b,b)`.

The case of a loop not being iterated at all is a very interesting case because many developers tend to forget this possibility when programming a loop.

Therefore this special case is constructed as a path set consisting of only those paths where the loop is skipped, i.e. the loop condition is fulfilled from the start.

Finding appropriate test inputs for this case by random test data generation is difficult and may be very time consuming. In two independent selections of two integers from a set of 2^{32} possible values for each a pair of equal integers is only selected in one out of 2^{32} cases.

Given a typical throughput of 3000 test input tuples per second for a function-under-test at this complexity level [3] this gives a mean run time of 50 days to acquire at least one such test input with a probability of 95%.



0xc0000005 STATUS_ACCESS_VIOLATION_ERROR tblcmp.c: 525 mkentry
0xc0000005 STATUS_ACCESS_VIOLATION_ERROR tblcmp.c: 617 mkentry
0xc0000005 STATUS_ACCESS_VIOLATION_ERROR tblcmp.c: 617 mkentry
0xc0000005 STATUS_ACCESS_VIOLATION_ERROR tblcmp.c: 525 mkentry
0xc0000005 STATUS_ACCESS_VIOLATION_ERROR tblcmp.c: 525 mkentry
0xc0000005 STATUS_ACCESS_VIOLATION_ERROR tblcmp.c: 617 mkentry
0xc0000005 STATUS_ACCESS_VIOLATION_ERROR tblcmp.c: 617 mkentry
0xc0000005 STATUS_ACCESS_VIOLATION_ERROR tblcmp.c: 525 mkentry
0xc0000005 STATUS_ACCESS_VIOLATION_ERROR tblcmp.c: 617 mkentry

84.38% = 27 of 32

90.91% = 20 of 22

Fig. 6-1: Distribution of Filtered Test Cases

In contrast, the constraint-based test data generator was able to provide test inputs for all 8 path sets constructed from the GCD source code within about 350 milliseconds (Tab. 6-1), including such for the special case of skipping the loop.

Looking at the example of determining the intersection of two rectangles, the experiment shows that in contrast to the GCD example the random test data generation approach is superior in this case from a performance point of view.

For the code determining whether one rectangle is contained in another, the path set construction provides 9 different path sets, consisting mainly of the different paths through the conditional statements determining whether the extensions in x- and y-direction of the rectangles do overlap. The constraint-based approach delivers one test input tuple for each path set within about 560ms, which corresponds to about 16 test input tuples per second.

However, the different path sets merely constrain the inputs by simple inequalities, such as requiring that the x-ordinate specifying the right limit of the contained rectangle is less than the x-ordinate specifying the right limit of the containing rectangle, or vice versa.

It can be easily derived that by randomly selecting values for the ordinates will provide appropriate inputs with a high probability of about 50% when only the pair of the right ordinates is considered. Considering all four pairs of ordinates (top, bottom, right, left) results in a reduction of about 1/16. The additional requirement that both rectangles must be non-empty (i.e. `left<right`, `top<bottom`) may additionally reduce this probability, but it will nowhere come near to the impressively small 1 in 2^{32} probability observed in the GCD example.

Therefore for this example the pure random generation concept will produce a much higher throughput of interesting and also differing test inputs than the constraint-based approach.

From this it follows that an efficient strategy must first try to provide the required coverage by a random generation approach. Only if the desired coverage is not reached after a certain amount of time has been spent, the constraint-based generation concept may be used to fill the gaps.

Example	**# path sets**	**Time / ms**	**Mean Throughput / s**
GCD	8	~350	~23
rectangle intersection	96	~3300	~29
rect-in-rect	9	~560	~16
point-in-rect	9	~55	~164

Tab. 6-1: Performance of Constraint-based Testing

The tool used in these experiments is still in a prototype phase. There are optimisations planned regarding the prediction of paths which are expected to make the approach feasible also for more complex constructs, including the consideration of subprogram calls.

The algorithm used to derive the test data once the path sets are determined is of such a general nature that, e.g., also machine code could be used as input, which might make the non-availability of source code for parts of a system under test less critical.

It also needs to be noted that the efficiency of the approach depends heavily on the efficiency of the underlying constraint solver. The solver must be optimised for fast detection of inconsistencies in the constraint set, i.e. the absence of a solution fulfilling all the constraints.

Unfortunately, most constraint solvers based on the finite-domain constraint solving strategy typically are not optimised for this kind of determination. In early phases of this research it was found that the generally available solvers of this kind need a huge number of propagation steps just to determine that `a<b` is inconsistent with `b<a`.

Such an optimised constraint solver was defined using K.U. Leuven JCHR [22], based on a combination of axiomatic propagation (e.g. $A \leq B \wedge B \leq A \Leftrightarrow A=B$) and finite domain constraint solving techniques. The solver has been shown to identify inconsistencies efficiently.

As the K.U. Leuven JCHR constraint compiler was lacking trailing capabilities to allow backtracking after detecting an inconsistent constraint store, such capabilities had to be implemented in addition, based on the Memento design pattern [23]. The trailing method is only loosely interfacing with the generated constraint handler code and the JCHR compiler, and only a few modifications were necessary to the latter. However, a more integrated implementation might be more optimised, but is expected to require considerable changes to the structure of the generated constraint handlers.

7 DISCUSSION ON TEST STRATEGY

7.1 Feedback on Test Strategies

The results presented in chapters 5 and 6 support globally the strategy as defined in Fig. 3-1. As an optimisation, the feedback from practice is to perform constraint-based testing after operational testing, especially when module testing leads to low coverage figures as observed for flex.

In this approach, for flex the time-consuming constraint-based testing only needs to be applied to about 1/3 of test cases based on selection by coverage criteria, hopefully ending up with very little manual effort required fortesting.

Similarly, without constraint-based testing the effort for manual testing is reduced to 1/3, when aiming to achieve the full 100% coverage by conventional (manual) testing.

The results on rule-based testing show, that it is still a challenge to achieve 100% code coverage, even if the syntax for the operational input is formally specified. In case of flex it seems that more cases and combinations can be generated than visible from the rule specification.

While the intention of auto-testing is to derive test cases for detailed testing and test evaluation at little manual effort, a major part of this issue can still be kept, even this basic approach leads to low coverage due to a programming style which is not well supporting auto-testing.

By capturing of test cases during operational testing a lot of more test cases can be identified automatically, so that human effort is only required for system operation.

7.2 Test Evaluation Strategy

Due to automatic capture of "test cases of interest" the test procedure for auto-testing looks quite different as for conventional testing.

In the upper part of *Fig. 7-1* the conventional procedure is shown. It starts with identification of test cases, firstly from the specification, later – possibly from code, too. Then the test environment has to be prepared, the tests have to be executed and evaluated.

In the lower part the modified test procedure in case of aut-.testing is shown. For the results produced by auto-test generation and auto-capturing during system operations only test evaluation remains as manual activity. However, the evaluation procedure differs from the one in case of manual testing. The auto-test environment provides test vectors and the compliance of these results have to be compared with the contents of the specification. This requires a transformation of the results in a form which can be compared with the specification.

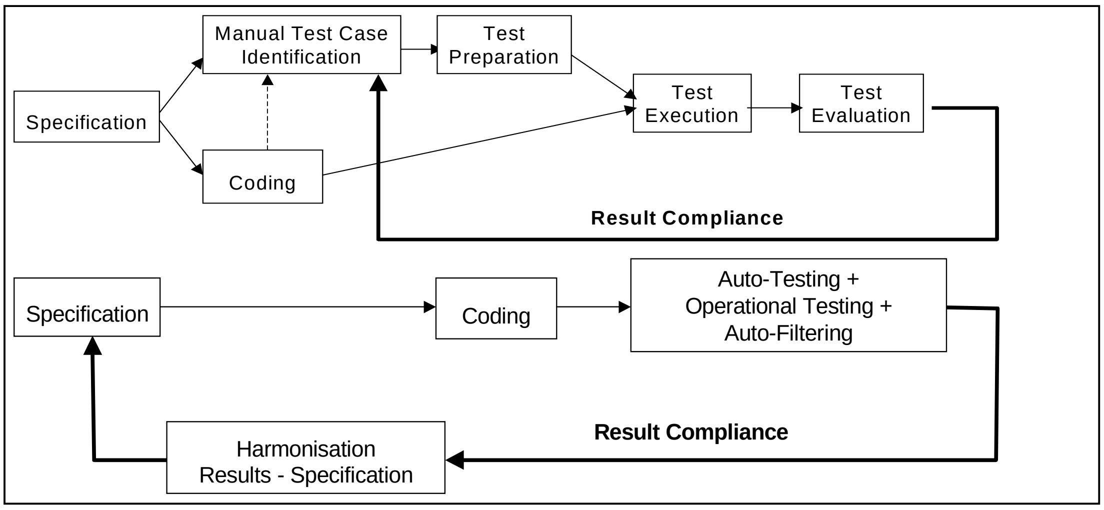

Manual Test Case Identification
Test Preparation
Specification
Test Execution
Test Evaluation
Coding
Result Compliance
Specification
Coding
Auto-Testing + Operational Testing + Auto-Filtering
Result Compliance
Harmonisation Results - Specification

Fig. 7-1: Test Evaluation Procedures

8 CONCLUSIONS

Specific conclusions were already drawn in sections 5.1.1.4 and 5.2.2 on module testing and occurrence of exceptions and anomalies. The final conclusions summarise the findings.

8.1 Coverage

The feedback on auto-testing based on information from specification and code is:

- the more defensive is the programming style, the higher is the coverage,
- considering a "defensive programming style" as a "good programming style" for high integrity software, it follows:
  the better the programming style, the higher the coverage,
- the more information on type ranges is available, the higher is the coverage. This explains why a higher coverage can be achieved for Ada software compared to C, if no measures are undertaken to compensate this weakness.

The recommendations of DO178x [21] suggest to take constants to express limits ("immediate operands" in assembler), e.g. for a loop range, to avoid potential corruption during execution. Applying this recommendation consequently towards a defensive programming style will help a lot to increase the coverage which can be achieved by automatic code analysis.

From this follows directly

- auto-testing cannot compensate poor context information in the code due to an "optimistic" programming style, based on the assumption of "inherent correctness" of code,
- auto-testing cannot compensate poor testability of code.

Vice versa, low coverage is a strong indication for weakness of code and potential problems during its execution.

8.2 Platform Dependencies

The results lead to the following conclusions:

- Test results depend on the properties and capabilities of compilers. One compiler may give more hints on potential weakness of the code than others.
- Test results depend on the actual configuration of the test environment formed by the OS, run-time system of the compiler, the processor type, the available memory and memory access, and on attached hardware.

As it is unknown a priori which is the best platform / compiler and configuration of the test environment regarding sensitivity to raise excpetions or to produce anomalies, the recommendation is to apply more than one platform or test driver.

Non-occurrence of exceptions and anomalies does not imply their absence, similarly to "non-occurrence of faults does not imply absence of faults".

8.3 Test Efficiency and Cost Savings

The exercises described above demonstrate that auto-testing and its combination with operational testing does save a lot of human effort, in the investigated cases up to

2/3 of test preparation, test execution, documentation of results and preparation of regression testing.

Auto-testing is the more efficient, the better is the programming style. For high integrity software like space software this condition should be true, leading to the conclusion that for space software significant cost savings are realistic. Moreover, investments in a good programming style will result in high cost savings, possibly in multiples of the basic investment.

A low coverage indicates missing context information in the code, immediately whenever auto-testing is applied. This requires

- firstly, manual test procedures implying high effort,
- secondly, a detailed analysis of all the source code to ensure that no improper operation can be induced at run-time by data, dynamically created and changed dynamically. This also implies high recurring effort, as this manual analysis always has to be repeated after every small change in the course of maintenance.

8.4 Software Engineering Issues

A challenging result is that coding style should better or at all consider the needs of auto-testing. At little effort appropriate standards can be established and applied which shall help to reduce the costs of testing.

It is a real challenge because nowadays such issues are poorly considered, either regarding programming style or language support.

9 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 on constraint-based testing were carried out.

10 REFERENCES

[1] P.Godefroid, N.Klarlund, K.Sen: DART: directed automated random testing, Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 213-223, Chicago, IL, USA, 2006, ACM Presse, ISBN:1-59593-056-6

[2] C.Pacheco, S.K.Lahiri, M.D.Ernst, Th. Ball: Feedback-directed random test generation, ICSE'07, Proceedings of the 29th International Conference on Software Engineering, Minneapolis, MN, USA, May 23-25, 2007

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

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

[5] R.Gerlich, G.Fercher: "A Random-Testing Environment for Ada Programs", Eurospace Symposium "Ada in Aerospace", Brussels, November 1993

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

[7] 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

[8] 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

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

[10] CTM, Classification Tree Method, see e.g. J.Wegener: Test Case Design by CTM and CTE, TACOS'04, Barcelona, Spain, March 2004, http://www.lta.disco.unimib.it/tacos/InvitedTalk/WegenerTACoS04.pdf

[11] Cantata++, IPL Ltd. Bath, UK, http://www.ipl.com

[12] The GNU oSIP Library, open software for the Session Initiation Protocol (SIP), http://www.gnu.org/software/osip/osip.html.

[13] Flex, Fast LEXical analyser generator, http:// www.gnu.org/software/flex/

[14] B.P. Miller, G. Cooksey and F. Moore, "An Empirical Study of the Robustness of MacOS Applications Using Random Testing", First International Workshop on Random Testing (RT'06), Portland, Maine, ACM Press, July 2006, pp. 46 – 54.

[15] R. Hamlet, "Random testing", In: J. Marciniak (ed.), Encyclopedia of Software Engineering, Wiley, 1994, p. 970 – 978.

[16] T. Y. Chen, H. Leung, I. K. Mak: "Adaptive Random Testing", Proceedings of the 9th Asian Computing Science Conference Advances in Computer Science (ASIAN 2004), LNCS 3321, Springer-Verlag, 2004, pp. 320–329, http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3321&spage=320

[17] J. Mayer, "Lattice-Based Adaptive Random Testing", Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), ACM Press, New York, NY, USA, 2005, pp. 333–336.

[18] R. Merkel, "Analysis and Enhancements of Adaptive Random Testing", PhD Thesis, Swinburne University of Technology, Australia, 2005.

[19] T. Y. Chen, T. H. Tse, Y.-T. Yu, "Proportional Sampling Strategy: A Compendium and Some Insights". Journal of Systems and Software 58(1), 2001, pp. 65-81.

[20] P. McMinn, "Search-Based Software Test Data Generation: A Survey", Software Testing, Verification and Reliability, 14(2), 2004, pp. 105-156.

[21] DO-178B, Software Considerations in Airborne Systems and Equipment Certification, Radio Technical Commission for Aeronautics (RTCA), http://www.rtca.org/

[22] P. Van Weert, T. Schrijvers, B. Demoen, "K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR System for Java", Proceedings of Second Workshop on Constraint Handling Rules, Sitges, Spain (Schrijvers, T. and Frühwirth, T., eds.), 2005, pp 47-62

[23] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995