Fault Identification Strategies

Rainer Gerlich[1], Ralf Gerlich[1], Carsten Dietrich[2]

[1] *BSSE System and Software Engineering, Auf dem Ruhbuehl 181,*
88090 Immenstaad, Germany, Phone +49/7545/91.12.58, Mobile +49/171/80.20.659, +49/178/76.06.129
Fax +49/7545/91.12.40, e-mail:Rainer.Gerlich@bsse.biz, Ralf.Gerlich@bsse.biz URL: http://www.bsse.biz
[2] *DLR, Deutsches Zentrum fuer Luft- und Raumfahrt, Space Agency, D-53227 Bonn, Königswinterer Str. 522-524*

ABSTRACT:

Various strategies for fault identification exist – e.g. based on formal analysis of code or on testing – of which each focuses on certain identification aspects and fault types. This paper characterises the strengths and weaknesses of methods – in theory and practice – focusing on application-independent identification strategies, and it suggests strategies to maximise the number of detected faults while minimising the related effort. Fault activation conditions are discussed in detail, resulting in an extended scope on stimulation needs. In particular, the contribution of automation in raising the activation probabilities is investigated. Various examples of fault activation mechanisms and statistics on fault types vs. identification methods are provided as observed in practice. An interesting result is the identification of application-dependent test cases by application-independent test strategies.

1 INTRODUCTION

The ultimate goal of fault identification as understood in this paper is to maximise the number of observable faults at minimum effort. This intention implies identification of faults even though they may be dormant under some conditions or in the current operational context of a software system (depending on a platform, mission configuration, etc.). In this respect, this goal is fully in-line with the goal of ISVV, Independent Software Verification and Validation.

To succeed for this ultimate goal the mechanisms of fault activation and fault hiding must be known, so that measures can be taken to minimise the number of dormant faults. Once the required mechanisms are known, the efficiency and reliability of identification strategies have to be considered. A number of examples is provided to explain the identified mechanisms of fault hiding and how they are addressed by automation. Finally, statistics on observed fault types, their activation conditions and the efficiency of identification (fault presence vs. identification in practice) are presented.

In Chapter 2 we introduce a terminology of faults and fault activation to get a baseline for the following discussions. In Chapter 3 we introduce the considered fault identification strategies and analyse for which fault types they are sensitive. In Chapter 4 we present representative examples for the various fault types and evaluate the sensitivity of a strategy. Finally, conclusions are provided in Chapter 5.

Larger tables have been moved to the appendix (Chapter 8).

2 CLARIFICATION OF TERMS

For understanding of the strategies a clarification is needed for the terms fault, error and failure as used in the context of software. Various definitions of these terms exist, e.g. by ISO/IEC [1], IEEE [2], DO178B [3], which are using the same term for different things or in a different interpretation.

2.1 Application-Independent Fault Identification Strategies

First of all, we introduce the term "application-independent fault identification strategy". In our understanding such a strategy allows to identify a fault without requiring specific knowledge on the application like a result of a calculation. Consequently, we are looking on strategies which are based on violation of syntactic, semantic or other rules or on occurrence of symptoms like exceptions raised when a fault is activated.

2.2 Fault, Error, Failure

While DO178B uses the sequence "error → fault → failure" to describe the source of an anomaly and its consequences, ISO/IEC and IEEE consider instead "fault → error → failure" to describe the same effects.

The term *anomaly* is used here whenever we do not want to distinguish between fault, error and failure.

In this respect our definition of these terms is:

- a fault is related to violation of a certain rule, which *may* have an impact on the quality-of-service (QoS),
  example: *potential* for index out-of-range
- an error is the manifestation of a fault, i.e. when a violation of the rule *does occur,* which *may* have an impact on the quality of service,
  example: the index *actually is* out-of-range
- a failure is the manifestation of a reduction of *QoS*.
  example: an exception caused by the error (like "access violation") prevents execution of the service.
  example: out-of-range and result is *really faulty*.

An anticipated fault is a fault for which the possibility of its occurrence is known in advance. A non-anticipated fault is a fault which is raised unexpectedly.

In order to understand the mechanisms by which faults may hide, more terms related to occurrence of faults need to be considered.

2.3 Occurring vs. Detected

The terms *occurring* and *detected* address different points of fault identification. A fault may *occur* during execution of the code (*at run-time*), while a fault may be *detected* during execution at run-time *or* by analysis *at pre-run-time or* at *post-run-time*. Detection implies presence of a fault, and occurrence is a *necessary*, but not a *sufficient* condition for detection.

2.4 Fix it or Forget It?

We are taking the term *occurring* where usually *detected* is used. When a fault occurs, i.e. the violation of a rule actually occurs, it might not be *detected*, especially if it does not manifest as a failure. Even a failure might *occur* without being *detected*, because the QoS is not or cannot be checked in this situation.

The essential question is whether we should only care about *detected* faults, errors or failures or about *occurring* ones and even such which do not occur at all under the actual conditions, but still have the potential to occur. Consequently, shall we adopt the following argument? "If nobody is able to detect an anomaly in terms of a reduced QoS, it is not a violation of a contract at all".

As we will see later, the decision, whether such a fault can be ignored, cannot be made before the activation condition of the fault is really known. Therefore deeper knowledge about the fault is required. However, from a rigorous point of view, *any* chance needs to be taken to identify faults.

In consequence, our understanding is, that all shapes of a fault, *"non-occurring*, but *having a potential to occur"*, *"occurring"* and *"detected"* faults must be considered for fault identification. Therefore we will discuss in detail how faults can be activated and/or detected, which

- may occur *only*, i.e. the violation of a rule happens, but they are not detected or detectable, or
- even may or can *not occur* under typical conditions, but are present in the code and have a potential for activation.

As an example, a quality check may be done, but the reduced quality may be not recognised because the feedback to the test engineer is too complex to be properly interpreted.

The answer to above question heavily impacts the strategies to be applied for fault identification. Our position is: an anomaly which may occur must be subject of an identification strategy aiming to fix it.

Therefore it is not sufficient to look for errors or even failures only, but for faults as such. As detection of a fault is strongly related to fault activation, we will detail this discussion in the following section about activated and dormant faults.

2.5 Dormant and Activated

Apart from the definition of fault, error, failure another issue is related to the terms "dormant" and "activated" when being associated with fault and error. The use of these terms becomes even more complicated when the whole chain from higher abstraction level (model, programming language) down to execution of the binary code on a processor is considered.

The common understanding is that

- the terms *activated* and *dormant* are complementary to each other in the sense: a fault, which is not activated, is dormant.
- a fault is activated when it manifests to an error.

This is not very precise regarding the term "activation". Activation requires a condition for activation, the "activation condition". Following above understanding on dormant and activated, it remains unclear what an activation condition really is. If an index is actually out-of-range, is this understood as the activation condition, even if there is no impact on QoS, at all? Therefore a deeper investigation on activation conditions is needed.

We have observed curious situations like

1. a fault in the source code is masked by a compiler or by the processor architecture (examples 2 and 3 in Tab. 8-1 below),
2. a fault is introduced by the transformation process (in case of models), a compiler, the processor's architecture and/or resource utilisation (examples 4 and 5, Tab. 8-1),
3. a fault is masked in the execution context depending on memory or stack allocation or the status of the execution environment, in general (examples 6 – 8, Tab. 8-1).

Depending on the scope considered for the anomaly, it may be concluded that

- an anomaly is present or not, in the sense there is fault potential or not,
- it is dormant (it cannot occur under given circumstances) or activated (then there is possibly a chance to detect it) depending on conditions unknown in most cases before the anomaly is detected.

In cases 1 and 2 above we are talking about a "platform-dependent anomaly" where platform is a synonym for *model transformer*, *compiler*, *operating system*, *test conditions* and/or *hardware architecture*, and in case 3 about a "context-dependent anomaly" where *context* stands for any item impacting the activation conditions (at run-time) on a given platform.

The term *dependent* indicates that the *basic* activation condition of a fault may be biased by another condition. The basic activation condition for an out-of-range fault is that the value *is actually* out-of-range. However – as we can see in example 6 of Tab. 8-1 – this is a *necessary* condition for fault activation, but may be not *sufficient* – because the QoS may be not compromised.

When considering the source code only

- in Case 1, ignoring the full scope down to execution of the binary format, a fault would be considered as present, while in the binary format or during execution it has vanished: no fault potential anymore. Such a fault we call *dormant w.r.t. the platform* or a *platform-dependent fault* (PDF), because the activation condition depends on the platform.
- in Case 2, a fault is not present in the source code, but in the binary format and the service of the system could be affected. In consequence, such a fault cannot be detected by source code analysis only. It is a *PDF, too*: a fault is added by a platform and the engineer does not know about it.
- in Case 3, a fault would be present, but would not compromise the service during any operational or test condition, unless the memory structure or the execution environment are modified e.g. during maintenance. Such a fault we call *dormant w.r.t. the context* or a *context-dependent fault* (CDF). In this sense missing a deadline would be classified as an CDF, i.e. the activation condition depends on the context "CPU-power" or "CPU-time consumption".

These two activation dependencies (PDF and CDF) have to be considered together with more basic activation conditions which we call – following the terminology of dependencies:

- *input-dependency*
  i.e. a fault is related to incoming data,
  and the related fault type is an *input-dependent fault* (IDF)
  example: index out-of-range
- *resource-dependency*
  i.e. a fault is related to lack of resources like memory, stack, CPU-power,
  and the related fault type is a *resource-dependent fault* (RDF),
- *event dependency*
  i.e. a fault depends on external events like a hardware fault or it is raised by software, e.g. by a fault handling part or fault propagation,
  and the related fault type is *event-dependent* or an *event-dependent fault* (EDF).

We take the term *input* here instead of *data* to distinguish between data which are essential to obtain a result, e.g. for execution of an algorithm – the *input data*, and data of the context, e.g. data which are not used during the calculation, but which may impact the activation condition.

The following examples E1, E2, E3 and E4 discuss the fault potential of an IDF w.r.t. the information available to prove absence of an IDF, i.e. the scope considered for a proof:

```
E1:
int arr[500];
k=arr[5+7];
```

```
E2:
const int i=5,j=7;
int arr[500];
k=arr[i+j];
```

```
E3:
int myFunc(int i, int j) {
 int arr[500];
 return arr[i+j];        }
```

```
E4:
int callee(int i, int j) {
 int arr[500];
 return arr[i+j];        }

void caller()            {
 int i=5,j=7,k;
 k=callee(i,j);          }
```

In E1 very clearly there is no fault in the code at all, regarding data corruption. The index is – usually – stored in a register, no way to activate the fault, except we do take into account a processor, compiler or assembler fault.

In E2 the situation is similar – at first glance. However, from a rigorous point of view there is a fault potential by software: the contents of i and j could be destroyed at run-time. E.g. we have observed that even a base pointer – stored outside the scope of a function – may be corrupted, manifesting as a Heisenbug.

In E3 – very clearly – there is fault potential, because it cannot be proven – in the scope of function myFunc – that the sum of i and j will really be in the range 0 .. 499.

The following cases are possible when the fault is activated by the basic condition (IDF) for E3:

- an access violation occurs, the fault occurs and may be detected by a raised exception,

- no address violation exception occurs, the address of arr[index out-of-range] may point to a value which is different from what is expected: the fault occurs and may be detected,
- no address violation exception occurs, the address of arr[index out-of-range] may point to a value which is identical with what is expected: the fault remains undetected regarding QoS, no way to detect it due to a reduced QoS. However, it may be detected by dataflow analysis, considering a larger scope or by a changed data context.

E4 is similar to E2 – in principle. Whether there is a fault potential in E4 depends on the scope. Considering the scope of the caller and the callee, there is no fault potential – apart from potential impacts by the context or the platform as discussed for E1 and E2. However, this – positive – conclusion is just a snapshot, because the proof is context-dependent: if by maintenance the situation in the caller changes, the proof may no longer hold. Therefore – from a rigorous point of view – there is a fault potential for callee – even when for the moment it can be ruled out. But it does exist regarding the full lifecycle of the software, and this cannot / must not be ignored.

The essential fact is that the assumptions which a proof depends on are usually not known. A formal analyser, e.g., can detect an out-of-range condition as being impossible. However, it does not list the assumptions on which this decision is based, and therefore it remains unknown whether after maintenance the proof is still valid. Consequently, the analysis must be repeated after any – even the smallest – modification. This conclusion is also true for reuse.

From a rigorous point of view, a proof on function-level by stimulation that an out-of-range will not cause a failure, is much safer and more rigorous than a proof on system level considering operational values, only.

The behaviour of the code at run-time in E3 could be made deterministic, i.e. detectable, as shown in E3.1 – provided an error handler is implemented on the level of the caller.

```
E3.1:
#define ERROR  -1
int myFunc(int i, int j) {
int k,arr[500];
k=i+j;
if (k<0 || k>499)
  k=ERROR;
else
  k=arr[i+j];
return k;                }
```

While for E3 it is unknown which value is returned when the fault is activated and – probably unknown – whether it is in the valid range or not, in E3.1 there is a convention to flag the error, so that it can be detected easily before it becomes critical (assuming that -1 does not overlap with any contents of arr). The activation condition is very clearly fixed, there is no platform- or context-dependency yielding a dormant fault or a non-detectable error.

Exceptions can be a better alternative in languages supporting them – such as Java, C++ or Ada – as they clearly occur outside of the normal function interface, while the error value might be ignored by a caller.

3 FAULT IDENTIFICATION STRATEGIES

To understand the required fault identification strategies we need knowledge about

- the activation conditions of faults,
- the principal strategies and their potential to detect faults – in theory and practice, and
- the fault-detection efficiency of a strategy.

These three aspects will be considered in this chapter.

The selected examples have been collected from three projects (Tab. 3-1):

- two Ada projects of Cat. A and C, after completion of normal tests and ISVV,
- one C project of Cat. C, after operation.

Lang.	Cat.	K Lines	KLOC	Functions
Ada	A	71	18	808
Ada	C	900	430	5500
C	C	48	40	765

Tab. 3-1: Analysed Projects

The examples are representative for the fault types only, the original code is not shown.

The analysis and test of the code as reported in this paper were executed after all usual fault identification methods (analyses, test, reviews) had been performed.

For the Ada projects examples on complex activation conditions only are given in Tab. 8-1 and Tab. 8-2, while more details will be presented for the C project in Chapter 4.

3.1 Fault Activation Conditions

Tab. 8-1 gives a number of examples for IDF, CDF and PDF activation conditions and their combinations collected from the three projects. The representative source code is provided as far as needed for understanding of the activation condition. The activation dependency is explained in the last column.

3.2 Fault Identification Strategies

Tab. 3-3 lists the considered strategies for fault identification and describes how a fault can be identified by a certain strategy. For each strategy examples of

faults are provided which can be – theoretically – identified by it.

The following strategies have been considered (for details see Ch. 3.3):

- strategies based on static analysis
- strategies based on symptoms and dynamic analysis.

To increase the probability of fault detection, testing was combined with fault injection and platform diversification.

For static analysis of the C code the gcc compiler 3.2.3 and Cantata++ (cantpp) [4] were used. i.e. syntactic, semantic and dataflow analysis, but not symbolic execution which is supported e.g. by the PolySpace tool [5]. For dynamic analysis (auto-testing) the DARTT (Ada) [6] and the DCRTT (C) [7] tools were applied. Tab. 3-2 gives the mapping between the considered methods and applied tools.

Detailed results are presented in Chapter 4 for the C project.

Tool		**Method**					
		Static Analysis				**Dynamic Analysis Auto-Testing**	
		syntax	semantic	dataflow	symbolic execution	anomaly monitoring	coverage evaluation
static	gcc compiler	×	×	×			
	gcc linker		×				
	Cantata++		×	×			
	theoretically				×		
dynamic / auto-testing	DCRTT		×			×	×

Tab. 3-2: Coverage of Methods by Tools

3.3 Assessment of Strategies

Tab. 8-2 discusses the efficiency of each of the strategies for the collected examples and the observed fault types:

- static analysis
  - o syntactic analysis
  - o semantic analysis
  - o dataflow analysis
  - o symbolic execution
- dynamic, symptom-based analysis
  - o analysis of run-time anomalies
    testing, possibly extended by fault injection and platform diversification
  - o coverage analysis

Symbolic execution was not applied, theoretical conclusions are made only.

If a strategy is not listed for a certain fault type, the fault type cannot be detected by it.

The following criteria are considered for the assessments:

- the reliability to detect a fault, theoretically and in practice,
- the manual and computational effort needed to detect and localize a fault.

The two principal categories of fault identification as discussed in Ch. 3.2 are

- execution-independent (incl. symbolic execution), mainly based on code analysis, and
- execution-dependent, i.e. testing, fault injection, platform and context diversification incl. evaluation of results at post-run-time.

Clearly, testing is not a method by which all faults can be identified [8].

Execution-independent methods are based on formal rules, an aspect which seems to make them superior to testing. However, this is the theoretical point of view. In practice, they may be less reliable than expected depending on the fault type in question for the following reasons:

- the analysis cannot be completed due to lack of computation time and/or disk and/or memory resources,
- a clear decision in the sense of green (no fault) and red (clearly a fault) cannot be derived,
- the tool itself may be subject to faults to some degree and suggests a wrong decision.

The first two reasons depend on the fault type and the complexity of the context: when information is needed across compilation units (files in C, packages in Ada) it may be difficult to derive a useful result in practice. To compensate the possibility of a fault in the tool, an independent tool should be used (tool diversification).

This may also provide a clear or clearer hint on a “fault / no fault” suggestion. The probability that a fault in such a tool suggests a wrong conclusion increases with the complexity of the context it has to consider.

Specific consideration is needed for automatic detection of a fault by testing. Testing can be divided into two principal test purposes: evaluation of test results based on

- application-dependent information,
- information for fault identification which is valid for every application.

In the second case automated detection of an activated fault is possible when its manifestation by *symptoms* can be observed automatically at run-time or evaluated at post-run-time, like exceptions, aborts, deadlocks, livelocks or specific messages and insufficient coverage.

Detection of faults by coverage analysis requires fixing of all other faults before being efficient. Otherwise, curious coverage figures may be a matter of exceptions, aborts etc.

3.3.1 Required Manual Effort

From the perspective of identification an essential point is the effort needed to find the critical location in the code and the raising condition. Execution-independent methods like semantic and dataflow analysis often immediately point to such a location and report the reason very clearly due to the direct relationship between code and fault. When symptoms are observed the location of fault occurrence can be identified in most cases, but not the reason directly.

Finding the reason requires usually manual analysis of the context. Symbolic execution also reports more on symptoms of a fault (range exceeded, division-by-zero) rather than on the source of a fault (where an extended range comes from). Therefore the identification effort of symbolic execution tends to be higher than for the other static analysis strategies and is comparable with identification effort of dynamic analysis strategies.

Though some faults can be detected by both, execution-dependent and execution-independent, methods, above conclusions suggest that it may be more efficient to start finding by execution-independent methods – as far as possible.

Consequently, execution-independent method should be applied prior to execution-dependent methods (dynamic analysis, symbolic execution). This will decrease the manual effort required for fault fixing.

3.3.2 Non-Anticipated Faults

All static analysis methods apply rules to identify faults, which implies they only can detect anticipated fault types. Identification methods based on symptoms like exceptions do not need to know a certain fault type prior to its manifestation, i.e. they do cover non-anticipated faults. They are just looking for the consequences – provided they can be observed. Therefore there is a good chance to detect such faults by symptoms. Last, but not least, symptom-based strategies are the only ones which can identify PDF and CDF.

The evaluation figures as provided in Chapter 4 do support these considerations.

3.3.3 Enforcing Fault Activation

Certain strategies – like fault injection, platform diversity and variation of the context – can increase the probability of fault activation which is only a matter of execution-dependent identification methods.

When aiming to demonstrate correctness (per test case), a representative (execution) environment is required. However, such an environment is not a pre-condition for fault identification. In latter case everything is allowed which helps to catch a fault. Consequently, all strategies useful to activate a fault are allowed and strongly recommended: fault injection, platform diversification, variation of the context. Several fault identification strategies may be combined like fault injection with platform diversification. We have experienced that porting the code to another platform and running it under changed conditions like emulation of hardware or another operating system is rather useful to detect – specifically non-anticipated – faults which cannot be detected at all on the original platform.

Automation – either to support platform diversification by auto-porting of the code or to stimulate the software-under-test over a – possibly huge – valid and invalid input domain – turned out as a pre-condition to efficiently raise the activation probabilities to a level which make the symptoms observable within a reasonable period of (execution / test) time.

3.3.4 Automatic Identification of Test Cases

The combination of automatic stimulation and automatic evaluation of test coverage (block and decision coverage/ MC/DC) allows the identification of application-dependent test cases by application-independent test and analysis methods.

Fault Ident. Strat.	Activity for Detection or Activation	Fault Manifestation	Source of Fault (non-exhaustive list)
Syntactic Analysis	Code analysis based on syntactic rules. Rules may extend beyond normal language syntax scope.	error or warning, compilation abort	syntax error, multiple data declaration = instead of == in condition, which usually is not a syntax error
Semantic Analysis	Code analysis based on local semantic consistency rules. Rules may extend beyond normal language semantic scope.	error or warning message, compilation abort	assignment to constant field (warning or error) invalid types in assignment (warning or error) missing variable declaration (error) inconsistent interfaces (error) inconsistent declarations (error) types too small/big for used range (warning)
Dataflow Analysis	Code analysis detecting relations between definitions of data items and their reached uses. Can be combined with constant value propagation.	warning message	unused assignment missing initialisation/assignment use of wrong source/target variables
Symbolic Execution	State transition equations are constructed based on control flow. Presence and/or absence of some types of faults can be deduced for some or all possible states.	error or warning message	out-of-range dead code critical casts de-referenced NULL pointer numerical exceptions memory access outside allocated range memory leak
Stimulation	variation of parameter und heap data within valid range only	exception, abort, lock	uninitialized data deadlocks and livelocks out-of-range critical casts de-referenced NULL pointer numerical exceptions
Stimulation + Fault Injection	variation of Parameter and Heap-Data within valid and invalid range	exception, abort, lock	missing protection against invalid data (out-of-range) faults in fault handling code
	corruption of return values	exception, abort, lock	missing protection against invalid data (out-of-range) missing check on returned NULL-pointer critical casts out-of-range faults in fault handling code missing protection against fault propagation
Range Checks	type range monitoring (DCRTT support)	DCRTT msg.	out-of-range
Checks on memory corruption	Check on corruption of mallocated memory	DCRTT msg.	change of data outside the portion of allocated memory
Platform diversification	variation of OS, processor, compiler or memory allocation	exceptions, compiler messages, DCRTT msg.	unused variables uninitialized data data corruption without raising an exception unsupported exceptions (like suppressed FPE)
Coverage	Analysis of identified functions with coverage<100% and manual analysis of function code	coverage figures<100% and red-coloured parts in graphics (DCRTT)	dead code faults in *logical expressions*, undetected by pre-run-time tools

Tab. 3-3: Fault Identification Strategies vs. Fault Types

The mechanism applied by DCRTT (called "Test Case Filtering") is straightforward once the capabilities of automatic stimulation and coverage evaluation at run-time are available:

1. define an upper limit on the number of executions after which an item is considered as "covered" (usually 1),
2. whenever a non-covered item (block or decision item) is entered in accordance with (1) above, record the corresponding inputs, outputs and other relevant data,
3. generate test drivers from such data for later regressions tests or tests on a target with resource constraints, which
    a. stimulate the function-under-test with test inputs,
    b. compare the recorded against the actual outputs.
4. confirm manually the correspondence between filtered inputs and outputs (proof of correctness) and thereby upgrade the automatically recorded inputs to test cases. If an output is faulty, correct the code and repeat the automatic stimulation until the outputs are correct.

Usually, test cases are derived from the specification. Therefore the approach described above seems to be non-compliant with standards – on the first glance. However, upgrading from test inputs to test cases implies a check against the specification. Consequently, the verification procedure is changed, while the verification result is the same:

- while usually the correct result (together with test inputs) is derived from the specification and is considered as reference for the correctness of the test output,
- in case of test case filtering the recorded input comes first, and the derivation of the correct output for the given input from the specification comes after, including checking of the coverage of the specification by the automatically identified test cases.

Another advantage of test case filtering is that it already considers the code, and provides test cases which never can be explicitly derived from a specification, because they are a matter of non-functional requirements on quality, safety, reliability etc. Such requirements otherwise would have to be applied to each piece of the code manually, then leading to additional code and test cases e.g. to check on proper fault handling. While such manual identification of test cases is rather tedious and error-prone, it is straightforward for automatic stimulation and does not require any manual effort.

Moreover, the filtered test cases are application-dependent, but were detected by an application-independent identification strategy.

3.3.5 Explanation of Assessment Terms

The following terms are used to characterise the identification capability of a strategy for a certain fault type in Tab. 8-2.

(as symbolic execution was not applied, only theoretical considerations are possible in this case):

- Scope of analysis
    - CU
      compilation unit + interface files
    - SC
      required source code down to given level of call hierarchy
    - FS
      required (full) scope over all levels + execution environment, from function-under-test incl. all further callees
- identification reliability
    - theoretical
      assessment based on theoretical considerations
        - medium, high, sure and capability to identify a CDF or PDF.

          CDF and/or PDF require support for fault activation. A strategy not providing this support will most probably not identify this fault.
        - medium
          the fault may be detected
        - high
          good chance to detect the fault
        - sure
          the fault will be detected in any case
    - observed
      assessment based on practical observations
        - n/a
          not applied in practice
        - yes
          the strategy identified such a fault type once, at least
        - no
          the strategy did not identify such a fault type although present and identification was expected
- identification effort
    - manual
      the effort needed to identify the source of the fault
        - low
          the issued message directly points to the source of the fault
        - medium

the message points to a location in the source code, and the context has to be analysed manually

- high
  the message points to a location in the source code. Detailed manual analysis is required, possibly across boundaries of functions and compilation units

- computational

  the computational time needed until a tool reports a fault or a symptom
  - very low
    range of seconds or lower
  - low
    range of minutes
  - medium
    range of hours
  - high
    range of dozens of hours or days.

3.3.6 Assessment Conclusions

The following compile options were used for the static analysis tools:

- gcc
  -Wall -Wunreachable-code -Wmissing-noreturn -Wfloat-equal -Wpointer-arith -Wsign-compare
  Remark: -Wall does **not** imply:
  "switch on all warnings"
- Cantata++
  --comp x86-Win32-gcc3.2.3 --ci:-ids --keepmod --no_link --parse:--c <path>\gcc.exe -c

All static analysis strategies do not consider the real execution context and its potential impact either by fault masking or platform dependent fault generation. Such conditions can only be covered with dynamic analysis / testing.

Version 3.2.3 (and higher) of gcc shows a significantly increased capability to detect semantic faults. Many faults which previously could only be detected by specific static analysis tools can be found today by a compiler like gcc.

With DARTT the faults as listed in Tab. 8-1, examples 1-5 (non-exhaustive list) were found. With DCRTT the examples for C in Tab. 8-1 (examples 6-11, non-exhaustive list) and Tab. 8-2 were found.

The following conclusions consider the practical results of syntactic, semantic, dataflow analysis and of dynamic analysis, but not of symbolic execution, because this strategy was not applied.

The examples 4,6,7,14,20 as given in Tab. 8-2 confirm that not every fault, which should be detected for sure by static analysis tools, will be detected in practice. For these cases dynamic analysis was the only strategy which detected the fault.

Observation of run-time anomalies was the only strategy which identified the faults of examples 9,10,11,12,13,15,16,17, and in nearly all cases – except for 16 – the fault was detected due to fault injection.

Coverage analysis was the only strategy by which the faults of examples 1,5,6 could be detected.

Consequently, pure static analysis – syntactic, semantic and dataflow analysis – is not sufficient to detect all faults, as the presented results demonstrate. Whether symbolic execution can also cover faults detected with observation of run-time anomalies and coverage is an open issue, in practice. However, the theoretical conclusions in Tab. 8-3 suggest, that symbolic execution cannot cover the observed fault types 2-6, 8, 18-22 from a principal point of view (see also Ch. 4.2). For fault types 1, 11, and 13 it strongly depends on the context whether they could be covered in practice.

Fault types 5, 6, 20 can only be covered by dynamic analysis, and it is likely that this conclusion is also valid for fault types 1, 11, 13.

Semantic analysis was the only strategy which can identify an inconsistency as described in example 22.

Regarding computation time, syntactic, semantic and dataflow analysis are the fastest strategies (very low – low) due to the limited context of a compilation unit. For these strategies the manual effort to identify the source of a fault can be expected low to medium and – in general – to be lower than for the other strategies due to the strong correlation of analysis and messages flagging the fault. In the other cases a message must be correlated manually with the source of the fault in the code.

3.3.7 Tool Operation

The installation procedures are straight forward for all tools and complete within about 15 minutes or even less time.

Specific preparation of the tools (gcc, Cantata++, DCRTT) for analysis and test execution was not required. The execution of gcc and Cantata++ was an integral part of DCRTT test automation. DCRTT itself only requires provision of the set of source files in a directory and configuration of test conditions like fault injection, scope of stimulation etc.. The only pre-condition to be fulfilled for DCRTT (and DARTT) is that the source files should be compilable and linkable free of errors.

All relevant diagnostic messages from compiler, Cantata++ and DCRTT are filtered and collected in separate files by the DCRTT run-time support. In case of Cantata++ only the features of static analysis were

used, but not the features for test set up, because this is inherently supported by the DCRTT feature for test driver generation (see also Ch. 4.6, test case filtering).

DCRTT filters test cases according to coverage criteria when scanning a function-under-test over the input domain (parameters and static/global data including fault injection). Once the input-output relationship has been confirmed as being correct, such test cases will serve as further reference, e.g. when re-executing tests. Such test drivers can be re-executed on the development and target environment, even under memory limitations such as 64KB, e.g. together with a lean operating system like KEIL.

If desired, DCRTT will also generate these test drivers in a format compatible with Cantata++, so that manual definition of test cases is no longer required.

4 EVALUATION OF RESULTS

In this chapter evaluation figures for static and dynamic strategies will be presented for the C software (40 KLOC) already mentioned in chapter 3.

During the development of this software a gcc 2.x version was used to detect faults by static analysis. Therefore many warnings were shown now by gcc 3.2.3 and Cantata++, indicating the progress made for static analysis.

Only such faults will be discussed here which have a reasonable and obvious fault potential.

Violation of rules such as standards on readability of source code, errors in source code which will cause compilation errors, or messages on unused data are not considered as a "fault" in the context of this paper. Unused data were detected, but not tracked as their amount would have compromised the figures for the more "serious" faults.

Compilation errors are of relevance here because of the long history of the source code under inspection. Due to evolution of compilers some violation of syntactic rules will be recognised as an "error" today, while it was not flagged as an error in an earlier compiler version.

The number of all these faults is estimated as high based on observation of corresponding compiler messages and analysed source code, probably being in a range of the number of recorded faults, but possibly up to multiples of this amount.

Also, anomalies due to lack of robustness (non-defensive programming style, missing protection against invalid data) are not considered as faults here. For a discussion on robustness issues see Ch. 4.3. The number of reported anomalies related to robustness issues is in the range of 200.

4.1 Detected Faults

Tab. 4-1 gives the number of faults detected by testing. All faults which were detected by static analysis are listed in the related column. As DCRTT also contributed to static analysis (see Tab. 8-3) the contribution from "classical" static analysis methods is shown in separate gray-shadowed lines.

Item		Identification Strategy			Total
		static	dynamic		
			min	max	
faults abs.	with DCRTT	270	44	122	314
	without	159	0	0	159
faults %	with	86.0	14.0	74.2	100.0
	without	50.6			
faults/ KLOC	with	6.8	1.1	5.8	7.9
	without	4.0	0.0	0.0	4.0

Tab. 4-1: Identified Faults

For dynamic analysis a minimum and a maximum value are provided:

- minimum
  faults are counted, only (see Ch. 3.3.1), which could or cannot identified at pre-run-time by static analysis, because e.g.
  - the fault is platform- or context-dependent,
  - the identification at pre-run-time is too complex, impossible or the tool did not identify it though it should have been possible, in principle.

  This figure is complementary to the one of static analysis regarding the total number of detected faults.
- maximum
  all identified faults are counted which could have been detected by symptoms or dynamic fault analysis, also such ones which are covered by "classical" static analysis.

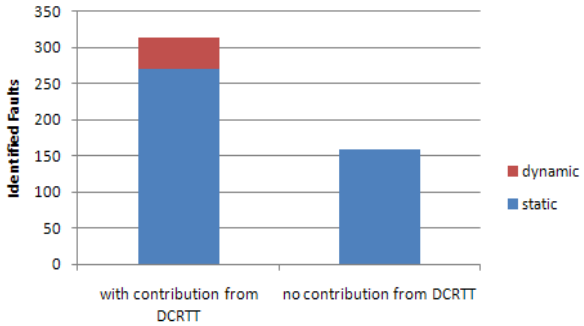


Fig. 4-1: Fault Coverage vs. Analysis Mode

Fig. 4-1 visualises the fault identification potential regarding the impact by auto-testing / DCRTT.

Tab. 4-2 correlates the faults detected by symptoms to the identification methods and when the method was applied: at run-time or post-run-time. Fig. 4-2 shows the graphical equivalent of the percentage figures.

As an important result, the figures of Tab. 4-1 indicate that neither static nor dynamic analysis strategies can fully cover the spectrum of (observed) faults. Though this conclusion is related to the observed faults of the reference application, it is valid in general.

Another major result derived from Tab. 4-2 is that 27 (fault injection) + 1 (platform diversification) =28 out of 44 faults (63.6%) (rows 2 and 3) could only be detected by enforcing activation conditions raising the probability for fault occurrence, e.g. due to

- flagging lack of memory (heap, stack) by a modified return code (NULL),
- modified return codes indicating a fault like -1 or NULL in other cases,
- invalid input data (out-of-range),
- faults activating the fault handling parts,
- conditions activating platform-dependent faults.

Only about 30% of faults (row 1 of Tab. 4-2 were detected by stimulation with valid data.

Consequently, (automated) fault injection is a "must" to maximise fault identification. A minor, though non-negligible part of one event only is related to platform diversification, which identified a serious, but dormant fault related to non-activated data corruption on its intended platform.

Finally, three faults were detected by coverage analysis in parts where the coverage figures were sufficiently high and not corrupted by occurrence of exceptions. Possibly, more such faults could have been detected if the sources of exceptions would have been removed.

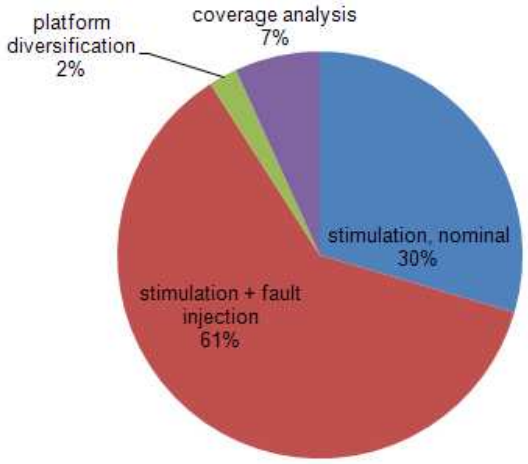


Fig. 4-2: Fault Coverage vs. DCRTT Identification Strategies

Symptom-based Fault Identification Method	**Applied at**	**# Faults**	**%**	**Comment**
Recording of exceptions, aborts, deadlocks, livelocks and *more run-time checks* during stimulation under nominal conditions and run-time anomalies	run-time	13	29,55	The phrase *"more run-time checks"* means: these are specific checks to identify malloc- and file-usage and corruption of mallocated memory supported by the test environment.
As above + fault injection	run-time	27	61,36	Stimulation under nominal and non-nominal conditions including enforced faults for return values. If types are not properly defined (e.g. int instead of enum) or the range is not checked, valid values, i.e. values in the specified type range, will be invalid, in fact, because they are not in the intended range.
As above + platform diversification	run-time	1	2,27	This specific fault was detected by corruption of mallocated memory: the test environment allocated function parameters by malloc, while in the operational environment they were allocated on the stack. This allowed the automated detection at run-time immediately after memory corruption.
Coverage analysis	post-run-time	3	6,82	Coverage analysis requires stimulation under nominal and non-nominal conditions to reach a maximum of branches incl. such ones for fault handling. To be efficient a high coverage figure should be achieved which requires fixing of all faults which cause exceptions and aborts.
Total		**44**	100	

Tab. 4-2: Results of Symptom-based Fault Detection

4.2 Fault Coverage and Potential of Strategies

Details on observed fault types, the most efficient identification strategy and the distribution of observed faults and fault types are shown in Tab. 8-3 and in Fig. 8-1 and Fig. 8-2, respectively. *Tab. 4-3* explains the acronyms as used in Tab. 8-3.

Acronym	Description
B	Block Coverage
b	detection possible by coverage analysis
C	Compiler
D	DCRTT specific add-on's
E	Exception
I	invalid Input (in parameter, static data)
i	invalid input is in valid / specified range
K	Lock (deadlock or livelock)
L	Linker
M	Decision Coverage (MC/DC)
m	detection possible by MC/DC
O	invalid Output (return value, out parameter)
P	detected due to platform diversity
p	possibly depending on platform
R	Run-time message issued by DCRTT
s	feature could be covered by symbolic execution
T	Tool, Cantata++
t	could also be identified by a static analysis tool
x	feature could be covered by analysis method, but was not observed in practice
( )	could possibly be covered, but theoretically incomplete
small letters	theoretical assessment, not applied or observed in practice

Tab. 4-3: Acronyms as used in Tab. 8-3

A capital letter always (except for i and s) means that the referenced strategy is the most efficient strategy in terms of procurement costs and fault identification capability, if several ones may successfully be applied. Rounded brackets express a principal capability of a strategy to identify a fault type, but it is not sure if identification is really supported or even fully possible. An "x" indicates that identification should have been possible by a certain strategy, but was not observed. Further, an "s" indicates that symbolic execution should be capable to identify a fault type, without saying anything about practical results. Finally, an "i" indicates that invalid input was received though the value was in the valid / specified range. This is a matter of imprecise use of types, which is – in part – a consequence of the type concept of C. Therefore DCRTT offers an option to precisely specify a limited range. If more than one capital letter occurs in a row, no clear decision on the optimum strategy is possible.

The main parts of Tab. 8-3 are:

- the fault types, which are described in Col. 2 - 3,
- the strategies of static analysis in Col. 4 – 7,
- the strategies of dynamic analysis in Col. 8 and 9 followed by applied stimulation methods (data stimulation, platform diversification),
- the observed number of faults in Col. 12-14

The following conclusions depend on the specific fault distribution profile as given in Tab. 8-3, but are still valid in general – apart from the quantities. The discussion below refers to the bottom lines of Tab. 8-3, where summary figures are provided.

A detailed view is required for the figures for static analysis, which include contributions from "classical" static analysis tools and an additional contribution from DCRTT (dynamic analysis, testing), identified by specific analysis directly related to the preparation of the test environment. Therefore three sets of summary figures are provided: the first set counting the contribution from classical static analysis, the second set considering the contribution from dynamic analysis / DCRTT, and the third set showing all contributions.

While in total 25 fault types (51%) and 270 faults (86%) were covered by static analysis, "classical" tools without DCRTT did only cover 18 fault types (37%) and 159 faults (51%).

For dynamic analysis a minimum and maximum value is provided for the number of observed faults (cf. Tab. 4-1). The minimum refers to the faults which cannot be detected by static analysis at all, the maximum number to the amount which can be detected by dynamic analysis, at most.

The contribution of strategies to fault coverage is shown in Fig. 4-3. 192 faults could be covered by static analysis, 44 faults by dynamic analysis, and 78 faults could have been covered either by static or dynamic analysis, where preference should be given to static analysis as discussed in Ch. 3.3.1.

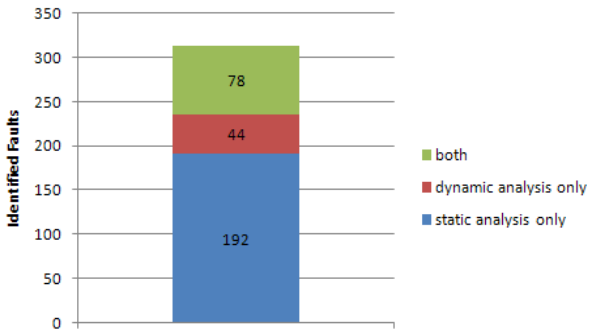


Fig. 4-3: Fault Coverage by Strategies

Dynamic analysis did cover 21 fault types (43%) and 44 faults (14%) at least, i.e. what was not covered by classical static analysis, and could cover 31 fault types (64%) and 122 faults (39%) at most. To these figures the related contribution from DCRTT static analysis should be considered, in addition: 7 fault types (14%)

and 111 faults (35%), which yields in total for the minimum 57% fault types and 111+44=155 faults (49%).

Regarding the comparison classical analysis vs. DCRTT the figures are (*Fig. 4-4*): 81 by classical analysis, 155 by DCRTT and 78 by both.

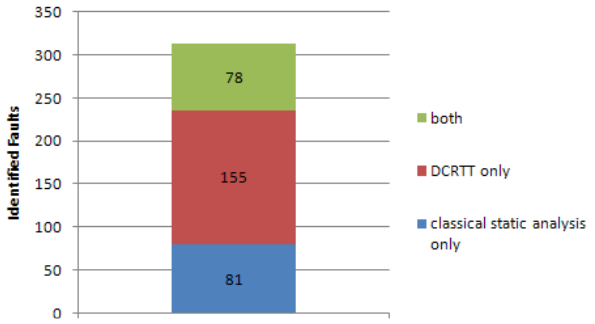


Fig. 4-4: Fault Coverage by Tools

Today, compilers (together with linker) can already detect a lot of fault types as indicated by 'C' in column "semantic analysis": 9 out of 49 (~18%). When combining all four static analysis strategies (incl. symbolic execution) and considering their maximum potential for fault identification, only 3 out of 49 fault types would be not covered. Taking a more realistic view, eight fault types may not be covered.

The largest contribution in the area of static analysis comes from semantic analysis supporting detection of about 51% of these fault types.

Symbolic execution may cover 40-63% of the fault types and 27-39% of the faults. Unfortunately, no practical results could be derived due to lack of a tool. The practical aspect is whether the full potential will really be available in practice due to a potentially high effort and/or high number of false alarms for which no clear decision on fault occurrence can be derived.

Anomaly monitoring supports detection of 39-43% of fault types and 13-50% of faults. Actually, coverage analysis contributed with about 4%, and has a potential for about 21% for identification of observed fault types, and 1-25% in case of faults.

4.3 Robustness

One of the challenging issues of testing based on function prototypes/specifications – as DARTT and DCRTT do – is the compliance between a prototype and its (function) body. It is common practice to rely on valid data in the body, without checking on the valid range of data coming in through the interface (see remarks for Example E3.1 in sect. 3.1).

Run-time anomalies (exceptions, aborts, locks) caused by such discrepancies between prototype / specification and a body are not included in the figures of Tab. 4-1. As in all such cases the specification allows a broader range than allowed, provision of data in the valid range (according to the specification) implies fault injection.

The following number of such anomalies were found during testing for the three projects of Tab. 3-1:

- Ada, Cat. A

anomalies were found during module testing. By detailed manual analysis it was proven that the conditions raising the anomalies cannot occur in the overall system context – actually.

- Ada, Cat. C
  - About 700 anomalies were observed. Most of them were related to the fact that operands of arithmetic operations like +,-,*,/ and the result cannot have the same type, which is usually a problem if the range of the type is rather small: i:=i+1; will always lead to an exception when on the right side the full range of i is applied (even for the full range of type Integer).
  - Apart from these – more or less – "uncritical" anomalies, a few cases were found – after completion of usual test and ISVV activities – which were critical.
- C

  About 200 anomalies were reported. Most of them were related to out-of-range conditions, e.g. for an index into an array because the index is of type int while the array size is of limited range and no range check was implemented.

Consequently, by dynamic testing, especially when fault injection is applied in addition, an immediate decision is possible – as confirmed in practice – on whether a defensive programming style was applied or not. A non-defensive style has consequences on detection probability based on code execution, because such anomalies will create an undesired change of control flow due to exceptions, aborts, deadlocks, livelocks. Therefore

- at run-time
  more faults may be hidden,
- at post-run-time
  faults in logical expressions (bad branching, deadcode) are more difficult to detect because the coverage figures are becoming too low, so that critical cases with a small contribution to coverage cannot be distinguished easily from those resulting from an undesired interruption of the control flow.

Consequently, the maximum number of faults can only be detected by dynamic analysis when all "obstacles" raising exceptions are immediately removed when detected.

4.4 Variation of Test Conditions

To achieve maximum coverage figures and fault identification, stimulation conditions should be varied. This may require a number of runs of DCRTT/DARTT with different configuration options.

The following principal configuration options and the described sequence of execution and result evaluation turned out as rather useful:

1. stimulation of function parameters in valid range only
   fix all anomalies before proceeding to next step
2. stimulation of parameters in valid and invalid range
   fix all anomalies before proceeding to next step
3. extend stimulation to global/static data as performed in steps 1 and 2
   fix all anomalies before proceeding to next step
4. activate fault injection and repeat steps 1-3
   fix all anomalies before proceeding to next step
5. evaluate coverage figures and identify faults related to erroneous conditions (wrong conditions, missing brackets after conditional expressions) or other dead code.

It strongly depends on the quality of the application whether all steps have to be executed sequentially as described above – possibly several times – or a shortened sequence is possible. However, as each step only requires to change a few configuration parameters and the tests are executed in background without requiring any human intervention, the manual effort reduces to evaluation of test results, only.

Consequently, this effort is a matter of the quality of the software under test: the better, the less.

4.5 Test Duration

The test duration of automated stimulation strongly depends on the complexity of the application, the size of the input domain and the execution time of a function-under-test.

Many symbols in an application will increase the time needed for linking. If linking is in the range of minutes and the number of functions is in the range of thousands, this may already require dozens of hours or days (1 minute x 1000 ≈ 16:30 hours).

A huge input domain (incl. invalid data) may require a high number of stimulation steps to achieve a sufficient high density of test samples. However, experience shows that a few thousands of test samples are sufficient to achieve maximum coverage. The number of auto-generated samples was about 6,500 on the average for the 765 functions tested, when suggesting 3,000 by test configuration, yielding a total of about $5x10^6$ test samples for all functions.

Deadlocks and livelocks also may significantly increase the test duration. This is a matter of the upper limit on execution time per function. If it is 15 minutes and 50 functions will run into a lock, this will waste about half a day of test time.

Consequently, a high test duration – though not requiring human resources, but impacting the turn-around time – may be an indication for the poor quality of the software under test, especially of its poor testability. As a rule of thumb: the higher it is, the poorer is the quality.

Finally, there is no need to test all functions together. A sub-set always can be tested, based on what is included in the provided files. This may speed up the overall duration in case tests need to be repeated.

4.6 Test Input Filtering

The identification of "interesting test inputs" by coverage criteria (extended by inputs causing anomalies) reduces the huge number (like $5x10^6$ test inputs) to an amount which can be evaluated manually and – when the results have been confirmed to be correct – be upgraded to reference test cases for regression testing and testing on the target.

Roughly, about 5,000 "interesting" test cases have to be considered for full coverage which yields the following average figures to achieve full coverage (executing each block or logical decision once at least):

- ~1 test case per 8 LOC
- ~7 test cases per function
- ~1 test case per 1,000 stimulation inputs.

Though the high reduction ratio of about 1,000 seems to indicate a rather inefficient stimulation approach, it actually visualizes the broad coverage of the samples in the input domain which is mapped onto a much smaller number of equivalence classes representative for the code-under-test.

4.7 Lifecycle Impacts

When anomalies are detected after completion of the coding phase, more effort is required for fixing. Therefore, it is highly recommended to apply auto-testing as soon as first pieces of code are compilable and linkable.

4.8 Identification Strategies and FMECA

In the past, testing consumed much manual time and effort for test preparation, execution and evaluation of results. Therefore FMECA (Failure Mode, Effects and Criticality Analysis) was considered as an instrument for reduction of analysis and test effort by guiding the engineers towards the most relevant/critical parts of a system. This procedure implied to neglect other parts

which were identified as non-critical, but which still could be faulty.

Today, due to full automation of testing as supported by DCRTT, manual effort is only needed for test evaluation and configuration of test modes (see Ch. 4.4). Due to automated stimulation over the input domain, all parts of the software can be covered at little expenses. The budget required for auto-testing is mainly driven by the quality of the software, as poor quality will increase the size of automatically generated reports, the related effort for manual evaluation, and the degree of re-execution of tests as described in Ch. 4.4.

Consequently, concentrating the effort on parts, to which FMECA guides to, is no longer indispensable. Due to these extended capabilities, quality analysis is no longer limited by budget constraints as in the past.

5 CONCLUSIONS

In this paper principal considerations were drawn on how application-independent faults can be identified and how sensitive a certain strategy is w.r.t. to certain fault types. As an unexpected benefit of the practical exercises it was found that also application-dependent test cases can be identified by a generic, application-independent strategy.

A major conclusion is that platforms and context may impact the activation conditions of faults and even may add or remove faults. In most cases these faults are non-anticipated and cannot be detected by static analysis methods, but only by dynamic analysis based on monitoring of symptoms.

The detailed conclusions on the analyses presented in this paper are divided into the following topics:

- principal problems of fault identification,
- suggested strategies for fault identification,
- aspects of project and lifecycle management,
- the role of automation.

5.1 Principal Problems

Above discussion yields that faults may be hidden for the following reasons:

- during testing
  - fault activation is masked by context- or platform-dependent conditions,
  - a fault is not activated,
  - a fault is not recorded due to a fault in the tool,
- during static analysis or symbolic execution of the code
  - the fault cannot be identified as its activation depends on the platform
  - the fault cannot be identified due to practical resource limitations, unsupported features or faults in a tool.

In some cases for which fault identification by static analysis was theoretically expected, one counter example at least was found that the fault was not detected for the software-under-test. As a consequence, even supported anticipated faults may remain hidden, while unsupported anticipated faults will remain hidden for this strategy, for sure.

Some examples were given for cases where faults can be added or removed silently during transformation of the source code into executable code.

Finally, only anticipated faults can be identified by analysis strategies including requirements-based testing, symbolic execution and FMECA, while automated dynamic analysis in general has the potential to identify non-anticipated faults as well.

5.2 Suggested Strategy

Above considerations strongly suggest diversification of tools in general and a combination of possibly *orthogonal* or independent tools regarding principal fault coverage of a strategy, its tool implementation and its complexity:

- to apply static analysis at pre-run-time for detection of anticipated faults with medium to high complexity of tool implementation, and
- dynamic, symptom-based analysis for detection of anticipated and non-anticipated faults derived from test results incl. coverage analysis, requiring low complexity of the strategy and little to medium complexity of tool implementation.

High complexity of a strategy implies increased probability of a fault in the tool, and the same is true for definition of the rules which form the base of static analysis. For these reasons rule-based static analysis cannot be considered as perfect, while it was considered as perfect in the past. Of course, testing is also not a perfect strategy. However, a combination of such – possibly non-perfect – independent methods and tool implementations gives a higher chance to detect faults. If not being combined, not all types of faults can be detected as the contents of Tab. 4-1 and Tab. 8-3 does prove.

As far as a fault is not identified, its potential impact and hazard potential remains unknown. Therefore test effort must not be reduced at the cost of fault detection, and projects should put forward the ambition of detecting as many faults as possible, whether anticipated or non-anticipated, having a potential to be activated or not under given conditions. Symptom-based analysis requires removal of faults as soon as they are detected.

Otherwise faults may remain undetected due to code unreachable by the raised anomalies.

The results presented in Chapter 4 prove that static analysis is not sufficient to detect all faults, even when combining all strategies of this fault identification domain. Vice versa, this is also true for dynamic analysis.

The analyses as given in this paper demonstrate that the potential of fault identification needs to be considered for each of the strategies to be sure that all types of faults can really be caught. Consequently, this leads to an assessment of the "quality of the fault identification potential" of a strategy, similarly to mutation testing which evaluates the "quality of a test set".

5.3 Management Aspects

According to the discussion in Ch. 5.1 and 5.2 principal decisions need to be made on fault identification strategies in order to achieve a high coverage of fault types by the chosen fault identification strategies.

Testing was already considered as a key strategy in the past complementing static analysis. However, due to the rather high effort needed to prepare the test environment and to evaluate the results, testing was limited to parts of the code – more or less – aiming an optimum mix between costs and fault identification probability.

Full automation of testing – only requiring delivery of the source code – significantly helps to reduce the effort while widening the scope to stimulation and monitoring of the whole code, thereby overcoming constraints imposed by budget and schedule. As described in Ch. 4.6 relevant test cases can be automatically derived in a rather comfortable way just by provision of the source code.

However, this way of full auto-testing requires to fix faults immediately when they are identified. Therefore auto-testing should be applied as soon as pieces of code are ready for linking.

From this perspective late start of auto-testing unnecessarily increases costs and effort.

5.4 Role of Automation

Automation turned out as a pre-condition to identify faults which may be hidden or occur very rarely, otherwise.

Automatic porting of the code, i.e. automatic adaptation of platform-dependent code to make it executable on another platform, and automatic stimulation over the full valid input domain extended to invalid inputs and fault injection, significantly increases the chance to meet the conditions of fault activation, which otherwise could not be raised due to limited budget and schedule.

6 ACKNOWLEDGEMENT

The C testing activities as referenced in this paper were part of a contract executed for DLR Space Agency (Deutsches Zentrum fuer Luft- und Raumfahrt) funded by BMWi (German Federal Ministry of Economics and Technology).

7 REFERENCES

[1] IEC 65A 122, ISO/IEC 10746

[2] IEEE TC FTD/IFIP WG10.4 definitions on Dependable Computing

[3] RTCA/DO-178B/ED-12B, Software Considerations in Airborne Systems and Equipment Certification

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

[5] PolySpace Products, www.mathworks.com/products/polyspace

[6] DARTT, Dynamic Ada Random Test Tool, http://www.bsse.biz → Products → DARTT

[7] DCRTT, Dynamic C Random Test Tool, http://www.bsse.biz → Products → DCRTT

[8] Dahl, O.-J., Dijkstra, E.W. and Hoare, C.A.R. :Structured Programming, Academic Press, London, England, 1972

8 APPENDIX

Tab. 8-1: Examples on Activation Conditions

Id	Dep.	Lang	Example	Comment

1	IDF + PDF	Ada	`var1:unsigned_byte;`
			`var2:unsigned_byte;`
			`var3:Integer;`
			
			`-- Assignment for`
			`-- var1, var2, var3`
			
			`var3:=Integer(var1*var2);`
				There is a fault in line 4 because the result of the product is also of type *unsigned_byte*, but it may exceed 255. An error does not occur if the result is less than 256. It is considered as activated for a result >255, because then an exception should be thrown.
				Observation 1: On an Intel-CISC-Architecture the result is handled modulo 256, but no exception is thrown. Moreover, the final result as assigned to var3 may be negative.
				Explanation 1: The CISC-Processor allows byte-operations and therefore the result is modulo 256. Obviously, the status of the overflow flag was not evaluated. This leads to an error.
				Observation 2: On a Sparc-RISC-Architecture the correct result is obtained, but no exception is thrown.
				Explanation 2: The RISC-Architecture always performs 32-bit operations, therefore no overflow will occur. The constrained range must be checked on software level, but obviously such a check is not implemented. The correct result of the product is delivered, but due to the cast it remains unknown whether this is an error or not. *From this perspective, a cast always should be considered as a potential fault.*

2	PDF	Ada	`type T_INT8 is`
			`range -128 .. 127;`
			`for T_INT8'SIZE use 8;`
			`type T_INT16 is`
			`range -32768 ..32767;`
			`for T_INT16'SIZE use 16;`
			`DIM1 :constant:= 80;`
			`DIM2 :constant:= 10;`
			`subtype T_NB is T_INT8 range 0..DIM2-1;`
			`type T_DESC is record`
			`  NB : T_NB;`
			`end record;`
			`M_DESC : T_DESC;`
			`M_DESC.NB:=0;`
			`  T_NB` ***`(M_DESC.NB * DIM1)`***
			`M_DESC.NB:=9;`
			`  T_NB` ***`(M_DESC.NB * DIM1)`***
				The essential lines are the 4 last lines. The result of the product M_DESC.NB * C_SUBFRAMES is platform-dependent. On an ERC32/Sparc RISC architecture and Aonix compiler the results are fully correct in both cases, while on an Intel CISC architecture and GNAT V3.15p compiler the second result is -48, i.e. 720 mod 256 as signed number: 720=0x2d0, -48=0xd0 as T_INT8. The critical point is the universal integer in the constant definition. According to the standard a universal integer will be converted to the type of the other operand, which is T_INT8. Therefore the result should also be T_INT8.
				On the Intel processor no exception was raised. Therefore either the multiplication operated on type Integer with later masking of the result to one byte, only, or it was performed as byte-operation, but the overflow bit was ignored.
				As the Sparc always applies 32-bit operations, no overflow did occur. The required handling on types would have to be done on software level. Whether this was done remains open. As a matter of fact, the source code was not improved to make it clear.

3	PDF	Ada	`same example as above`	In case of the Intel-processor and GNAT compiler the cast to T_NB of *M_DESC.NB* * DIM1 raised an out-of-range exception when the result is -48, while the previous overflow was not flagged.
4	PDF	Ada	T'size	delivers different values depending on compiler: as occupied or needed w.r.t. to memory alignment, Ada (83,95) standard allows different understanding
5	PDF	Ada	`Alignment clause` ***`at .. range`*** `in a record`	The Ada standard does not require to make this feature independent of the hardware architecture, this will cause problems when porting between Little/Big Endian architectures.

Id	Dep.	Lang	Example	Comment

6	CDF + PDF	C	`const char cstr[]=”123”;`
			`typedef enum {false,true}`
			`Boolean;`
			`void func() {`
			`char var[3];`
			`Boolean bool;`
			`strcpy(vstr,cstr); }`
				Obviously, in the declaration of vstr the terminating 0 of cstr was not considered. Therefore the terminating 0 overrides the following byte of bool. As bool only takes the values 0 or 1 (true, false) (as defined by the context), in case of Big-Endian 0 would be replaced by 0. Consequently, no error ever will occur on this platform. If false would be changed to “! True” = 0xffffffff, than an error would occur. Similarly, it is when porting the code to a Little-Endian platform. Then the leading byte of bool would be corrupted in case of true or if false==0xffffffff (as vstr is of length 4, bool would directly follow without insertion of alignment bytes).

7	CDF or PDF	C	`char *str;`
			`str=malloc(100);`
			`str=0;//should be *str=0;`
			`... strcat(str,”xyz”);`
				The pointer is initialised instead of the area it is pointing to. Although it seems to be fully independent in the sense that it will lead to a failure in any case, it will be dormant if no exception is raised. This has been observed for VxWorks 5.3 / gcc 2.6 on an Intel X86 platform, while it was raised for the same versions on the ERC32 target. Apart from that there is another fault: missing checking of the pointer returned from malloc.
				It may be context-dependent for the following conditions: a value could be successfully stored and retrieved from address zero, then it would be dormant. If this value would be destroyed before reading by another illegal access to NULL, it could be activated.

8	PDF	C	`register short a=20000,b=2000,c;`
			`c=a * b;`
			`if (c>10){ } else { }`
				The result depends on the platform. On a Sparc (32-bit) it will be correct, on an Intel it may incorrect depending on the assembler instructions generated by the compiler.

9	IDF + CDF	C	`#define SIZE 500`
			`int i,j,arr[SIZE],result;`
			`for (i=0;i<SIZE;i++)`
			`for (j=0;j<SIZE;j++)`
			`result=arr[i+j];`
				In this case it is obvious that an out-of-range condition will occur. However, whether the fault is really activated still depends on what is returned for arr[out-of-range]. Firstly, an access violation exception may occur, leading to an error and a failure due to a branch in the control flow. Secondly, the expected value could be returned if the context provides such a value, then the fault is not activated. Thirdly, an unexpected value could be returned, which may manifest as an error and a failure.

10	IDF	C	`int func(int i, int j) {` `return i+j;} //should be *`	A wrong operator is applied: ‘+’ instead of ‘*’. In case of i=j=2 the fault is dormant, in all other cases it is activated.

11	independent	C	`errCode=SUCCESS;`
			`// TEST !!!!`
			`if (errCode==ERROR)`
				A statement inserted for test purposes was not removed. This prevents that the error handling branch is entered.

Tab. 8-2: Assessment of Fault Identification Strategies

Id	Example	Identification Strategy	Scope	Identification Reliability		Identification Effort		Description
				theory	observed	manual	computation	
1	`if (myfunc(para)==0) ...` `else ...`	Cov. Anal.	FS	sure	yes	Med. / high	low	myfunc(para) may return an invariant value, leading to either the then- or the else-branch never being executed.
		Symb. Exec.	SC	depends on myfunc	n/a	depends on myfunc	high	
2	`if (myfunc==0) ...` `else ...`	Cov. Anal.	FS	sure	yes	low	low	myfunc is the address of a function and therefore constant and not NULL, leading to the then-branch never being executed.
		Sem. Anal.	CU	sure	yes	low	very low	
3	`if (x=1 || x=2) ...` `else ...`	Cov. Anal.	FS	sure	yes	medium	low	x is always assigned 1 leading to the else-branch never being executed. The second part of the disjunction is never evaluated.
		Synt. Anal.	CU	sure	yes	low	very low	

4	`#define FILE_PATH "disk:/dir/"`
	`if (FILE_PATH == NULL) { }`
	`else { }`
		Cov. Anal.	FS	sure	yes	low	low	FILE_PATH is a constant pointer, but analysis tools did not detect this fault: else-branch is never reached (deadcode). It seems that this a matter of pointers, because a comparison of constant scalars is detected.

		Sem. Anal. / Constant Propagation	CU	sure	no	low	very low	

5	`char *fn;`
	`void myfunc() {`
	`if ((fd = fopen(fn,"a")) == NULL)`
	`{} }`
		Cov. Anal.	FS	high / PDF + CDF	yes	medium	low	Missing initialization of fn did not cause an exception, this became only visible by missing coverage.

6	`ret_value=SUCCESS; //TEST!!`
	`If(ret_value == ERROR)`
	`{ // then-branch }`
	`else`
	`{ // else-branch }`
		Cov. Anal.	FS	sure	yes	medium	low	A test statement is not removed and prevents branching.

		Dataflow Anal. / Const. Prop.	CU	sure	no	medium	very low	
		Symb. Exec.	SC	sure	n/a	medium	high	

7	`char *str=malloc(100);`
	`str=0;//should be *str=0;`
	`strcat(str,"xyz");`
		Dataflow Anal.	CU	sure	cantpp n/a
					Gcc no
						medium	very low	strcat() copies into a string at address 0 (null-pointer), potentially leading to a memory access fault. The allocated memory is leaked. A run-time exception related to access of address 0 is platform-dependent.

		Symb. Exec.	SC	sure	n/a	medium	high	
		RT anomaly	FS	sure / PDF	yes	medium	low	

8	`UINT32 itemUsed[..],cc;`
	`INT32 fc;`
	`int setItem(UINT32 cc);`
	`fc = setItem(cc);`
	`itemUsed[cc] = fc;`
	`for(i=1;i<itemUsed[cc];i++)`
		RT anomaly + Fault Inj.	FS	depends on setItem	yes	medium	low	Critical cast between signed and unsigned if setItem returns -1, if e.g. the value of cc is invalid. Then the upper limit of the loop is 0xffffffff instead of -1 which generates a quasi-endless loop. Can be detected by fault injection either for cc or the return value, increasing the probability for a negative value.

		Sem. Anal. / Type Checking	CU	sure	yes	medium	very low	

Id	Example	Identification Strategy	Scope	Identification Reliability		Identification Effort		Description
				theory	observed	manual	computation	

9	*File ctrlInit.c*
	`int ctrlArr[4]={1,1,1,0};`
	
	File ctrlVal.c
	`funcCreatePtr`
	`(char **ptr, int ind) {`
	`  if (ctrlArr [ind])`
	`  { *ptr=getStrPtr();`
	`  return SUCCESS;`
	`  } else return SUCCESS;  }`
	
	File ctrlExec.c
	`char *ptr=NULL;`
	`if (ERR==funcCreatePtr(&ptr,3))`
	`  exit(99);`
	`else strcpy(ptr,"xxx");`
		RT anomaly + Fault Inj.	FS	sure / CDF + PDF	yes	medium	low	Several faults exist in this example which are difficult detect because the dependencies spawn across file boundaries.
								
								Firstly, always SUCCESS is returned from funcCreatePtr preventing branching in file ctrlExec.c.
								
								Secondly, if the value 3 is passed, ptr remains undefined causing an exception in ctrlExec.c: conditional initialization across file boundaries.

		Cov. Anal.	FS	high	yes	medium	low	
		Symb. Exec	SC	low / CDF	n/a	medium	high	

10	`void flash(int id) {`
	`char str[100],msg[1000];`
	`switch (id) {`
	`case XXX: strcpy(str,"defined");`
	`default: // nothing }`
	`20nitia(msg,"msg=%s",str); }`
		RT anomaly + Fault Inj.	FS	high / CDF + PDF	yes	medium	low	If the value of id is invalid, str is undefined: conditional initialization within a function.

		Symb. Exec.	SC	high	n/a	medium	high	

11	`void resetTime(int sub,`
	`int  status, int activity) {`
	`  switch (sub) {`
	`  case XXX:execTime[activity]`
	`.tv_sec = 0;}}`
		RT anomaly + Fault Inj.	FS	medium / CDF	yes	medium	low	An invalid value of activity leads to an out-of-range condition. A check is missing.

		Symb. Exec.	SC	depends on decl. / def. of execTime	n/a	medium	high	

12	`void fullName(char *path,`
	`char *fn, char *ffn){`
	`if (path == NULL)`
	`  20nitia(ffn,"%s",fn);`
	`else`
	`  20nitia(ffn,"%s/%s",path,fn);}`
		RT anomaly + Fault Inj.	FS	high / CDF + PDF	yes	medium	low	The case fn==NULL is not considered, though it is not guaranteed.

		Symb. Exec.	SC	high	n/a	medium	depends on callers	
13	`void loadDB(int tbl, int id) {` `elem1[id]=elem2[tbl];}`	RT anomaly + Fault Inj.	FS	high / CDF + PDF	yes	medium	low	An invalid value of id may cause memory corruption. An invalid value of id may cause an access violation exception.
		Symb. Exec.	SC	depends on decl. / def. of elem1	n/a	medium	depends on decl. / def. of elem1 and	

Id	Example	Identification Strategy	Scope	Identification Reliability		Identification Effort		Description
				theory and elem2	observed	manual	computation elem2	

14	`long size,*getSize = NULL;`
	`if (readBuf(fd,offset,`
	`(void*)getSize,sizeof(long))`
	`  == ERROR)  { … }`
	`else{ size=*getSize;}`
		RT anomaly + Fault Inj.	FS	high / CDF + PDF	yes	medium	low	getSize is initialized with NULL, this value is passed to readBuf, but it cannot be changed. If ERROR is not returned, the else-branch will crash.

		Dataflow Anal. / Const. Prop.	CU	sure	no	medium	very low	
		Symb. Exec.	SC	sure	n/a	medium	medium	

15	`int Day(time_t zeit_t)`
	`{ struct tm *zeit;`
	`  zeit = (struct tm *)localtime`
	`((const time_t *)(&zeit_t));`
	`  if(zeit==NULL)`
	`  printf("no conversion\n");`
	`  return zeit->tm_mday; }`
		RT anomaly + Fault Inj.	FS	high / CDF + PDF	yes	medium	low	If localtime returns NULL in case of lack of memory or an invalid input, a crash will occur when executing the return. This can be enforced by fault injection.

		Symb. Exec.	SC	sure	n/a	medium	medium	

16	*File fault.h*
	`void verifyFault(int ind);`
	`void handleFault(int ind);`
	
	File verify.c
	`#include <fault.h>`
	`void verifyFault(int ind) {`
	`  switch (ind) { …….`
	`  Default: handleFault(ind);} }`
	
	File verify.c
	`#include <fault.h>`
	`void handleFault(int ind) {`
	`  switch (ind) { …….`
	`  Default: verifyFault(ind);}`
		RT anomaly + Fault Inj.	FS	sure	yes	medium	low	Recursive call of handleFault across file boundaries. Rarely execution of fault handling part. Enforced execution by fault injection for ind as parameter of handleFault.

		Symb. Exec.	SC	high	n/a	medium	high	

17	`typedef struct TyMsg {`
	`  int  msgLen;`
	`  char msgData[100]; }`
	`TyMsg Msg; char buf[100];`
	`getMsg(&Msg);`
	`memcpy(buf,Msg.&msgData[1],`
	`Msg.msgData[0]);`
		RT anomaly + Fault Inj.	FS	high / CDF + PDF	yes	medium	low	Implicit correlation of contents of msgData[0] and the actual length of msgData assuming that both are compliant,but may not. Enforcing non-compliance by fault injection.

		Symb. Exec.	SC	medium	n/a	medium	high	

18	`int i,j;`
	`printf("Bit:%d\n",DB[i].Id[j].Bit)`
	`;`
		Dataflow. Anal.	CU	sure	yes	medium	very low	Used before defined
								Variable undefined when being used, possibly initialized to 0

		RT anomaly	FS	high / CDF + PDF	yes	medium	low	

Id	Example	Identification Strategy	Scope	Identification Reliability		Identification Effort		Description
				theory	observed	manual	computation	

19	`typedef enum {EMPTY,FULL}STATE;`
	`int create(int flag, STATE state);`
	`int id;`
	`id=create(EMPTY,0xff00ff00);`
		Sem. Anal. / Type Checking	CU	sure	yes	low	low	enumerated type mixed with another type
								args turned around
								Wrong literal

20	*File myFile1.c* `int multiDecl;` *File myFile2.c* `int multiDecl;`	Linker Checks	FS	sure	no	high	low	Link error: multiple symbols The C standard tolerates multiple declaration of data when they are not initialized. In this case all symbols are mapped onto the same address. When initialization is combined with the declaration at one location at least, the compiler flags an error. In case of uninitialized declarations a missing static key word – though intended – is difficult to detect. This may cause undesired interference of presumably independent data and unpredictable results. Such a potential fault is flagged during automatic generation of the test environment by DCRTT.
		DCRTT analysis			yes			

21	*File myFile1.c*
	`int multiDecl=0;`
	File myFile2.c
	`int multiDecl;`
		Linker Checks			yes			

		DCRTT analysis			yes			
22	`int actl[4] = {0,1,0,0,1,1};`	Sem. Anal.	CU	sure	yes	low	very low	Inconsistency between data declaration and its 22nitialization.

Tab. 8-3: Fault Types vs. Optimum Strategy and Faults Found

#	Fault Type	Fault Sub-Type	Static Analysis				Dynamic Analysis				Observed Faults		
							Detection Method		Stimulation		static	dynamic	
			Syntax Analysis	Semantic Analysis	Dataflow Analysis	Symbolic Execution	Anomaly Monitoring	Coverage Analysis	Data	Other		min	max
1.	Uninitialized data	Index			T	s	e				3		3
2.		Branch-dependent initialization				s	E					3	3
3.		Missing initialization of strings + pointer			x	s	E					1	1
4.		Missing initialization of static data				(s)	E					1	1
5.	Range exceeded	String (terminating 0 not counted for length)				(s)	R			P		1	1
6.		Index/pointer out-of-range after loop				s	E					1	1
7.		Index out-of-range				s	E		i			13	13
8.	Malformed logical expressions	Assignment instead of comparison		(C) T		(s)		b m			9		9
9.		Logical expression of scalars is constant			T	s		b m			1		1
10.		Logical expression of pointers is constant			x	s		B M				2	2
11.	De-referenced illegal ptr	NULL-ptr, ptr->elem after function returning NULL				(s)	E		O			6	6
12.		NULL assignment and dereference in branches			x	s	E					1	1

#	Fault Type	Fault Sub-Type	Static Analysis				Dynamic Analysis				Observed Faults		
							Detection Method		Stimulation		static	dynamic	
			Syntax Analysis	Semantic Analysis	Dataflow Analysis	Symbolic Execution	Anomaly Monitoring	Coverage Analysis	Data	Other		min	max
13.		Corrupted or non-initialized pointer (!=NULL)				(s)	E					1	1
14.		NULL dereference				(s)	E					1	1
15.	Pointer arithmetic	arithmetic on void or function pointer		T							29		
16.		Cast between ptr and int over a number of stages			(x)	(s)	E					1	1
17.	Data declarations	Inconsistency between declaration and extern		D							2		
18.	Data consistency	Missing range check				s	E		I			1	1
19.		Missing type in data declaration	C								2		
20.	Mixed type expression	Wrong enum literal out of different set		T							3		
21.		Mixing signed and unsigned by a cast		T		s					14		
22.		Implicit mixed signed and unsigned expression		D			E					3	3
23.	Test code not removed							B M				1	1
24.	Macro	One stmt expanded to two stmt (if, while, for)	(C)					b			1		1
25.	Unclear semantics, checks on correctness impossible	Missing correlation between array and array length				s	E		i			3	3
26.		Unchecked conversion of data				s	E		i			1	1
27.		Missing correlation of data				s	E		i			1	1
28.	Unreachable Code	Code following endless loop		T		s	K	b			10		10
29.		Combinations of return and break following each other		T		s		b			47		47
30.		code following return (uncond.)		T		s		b			2		2
31.		code following break		T		s		b			2		2
32.		return following exit		C t		s		b			2		2
33.		; following }		T							1		
34.	Missing return for non-void			C t							18		
35.	Non-terminating recursive calls	Function calls itself				(s)	K		I			1	1
36.		Recursion after a number of function calls				(s)	K		I			1	1
37.	Format	%s and int		C t		(s)	e				1		1
38.		Too many arguments		C t		s					1		
39.		Too few arguments		C t		s					1		
40.	Duplicate symbol	Data declaration in h-file		(L) D							47		
41.		Multiple data declaration in c-file		(L) D							3		
42.	Incompatible types in assignment	Inconsistent function pointer		T							3		
43.	Incompatible parameter lists and function declaration	Function return type		C t							6		
44.		Parameter type mix (declaration and call)		C t							2		
45.		Mismatch of parameter count		D							1		

#	Fault Type	Fault Sub-Type	Static Analysis				Dynamic Analysis				Observed Faults		
							Detection Method		Stimulation		static	dynamic	
			Syntax Analysis	Semantic Analysis	Dataflow Analysis	Symbolic Execution	Anomaly Monitoring	Coverage Analysis	Data	Other		min	max
46.	Missing includes	Missing declarations	C	C t							1		
47.	Name overloading	Struct and data name		D							57		
48.	Decisions	Too large list of decisions		D							1		
49.	**Total fault types**	**48**											
50.	**Total faults, observed**	**314**									**270**	**44**	**122**
51.	**without** DCRTT contribution to static analysis										**159**	**44**	**122**
52.	**Total fault types covered** contribution from "classical" static analysis tools, only	**abs, min**	2	18	2	20							
53.		**abs, max**	3	18	6	30							
54.		**%, min**	4,17	37,5	4,17	41,67							
55.		**%, max**	6,25	37,5	12,5	62,5							
56.	**Total fault types covered** contribution from DCRTT, only	**abs, min**		7			19	2	9	1			
57.		**abs, max**		7			21	10	9	1			
58.		**%, min**		14,58			39,58	4,17					
59.		**%, max**		14,58			42,86	20,83					
60.	**Total fault types covered** all contributions	**abs, min**	2	25	2	20	19	2	9	1			
61.		**abs, max**	3	25	6	30	21	10	9	1			
62.		**%, min**	4,17	52,08	4,17	41,67	39,58	4,17					
63.		**%, max**	6,25	51,02	12,5	62,5	42,86	20,83					
64.	**Total faults covered** contribution from "classical" static analysis tools, only	**abs, min**	3	152	4	85							
65.		**abs, max**	4	152	9	121							
66.		**%, min**	0,96	48,41	1,27	27,07							
67.		**%, max**	1,27	48,41	2,87	38,54							
68.	**Total faults covered** contribution from DCRTT, only	**abs, min**		111			41	3	27	1			
69.		**abs, max**		111			45	77	27	1			
70.		**%, min**		35,35			13,06	0,96					
71.		**%, max**		35,35			14,33	24,52					
72.	**Total faults covered** all contributions	**abs, min**	3	263	4	85	41	3	27	1			
73.		**abs, max**	4	263	9	121	45	77	27	1			
74.		**%, min**	0,96	83,76	1,27	27,07	13,06	0,96					
75.		**%, max**	1,27	83,76	2,87	38,54	14,33	24,52					

Gray cells in col. "min" above: this fault type was only covered by dynamic analysis / testing

Gray numbers represent values which are identical with values in previous lines.

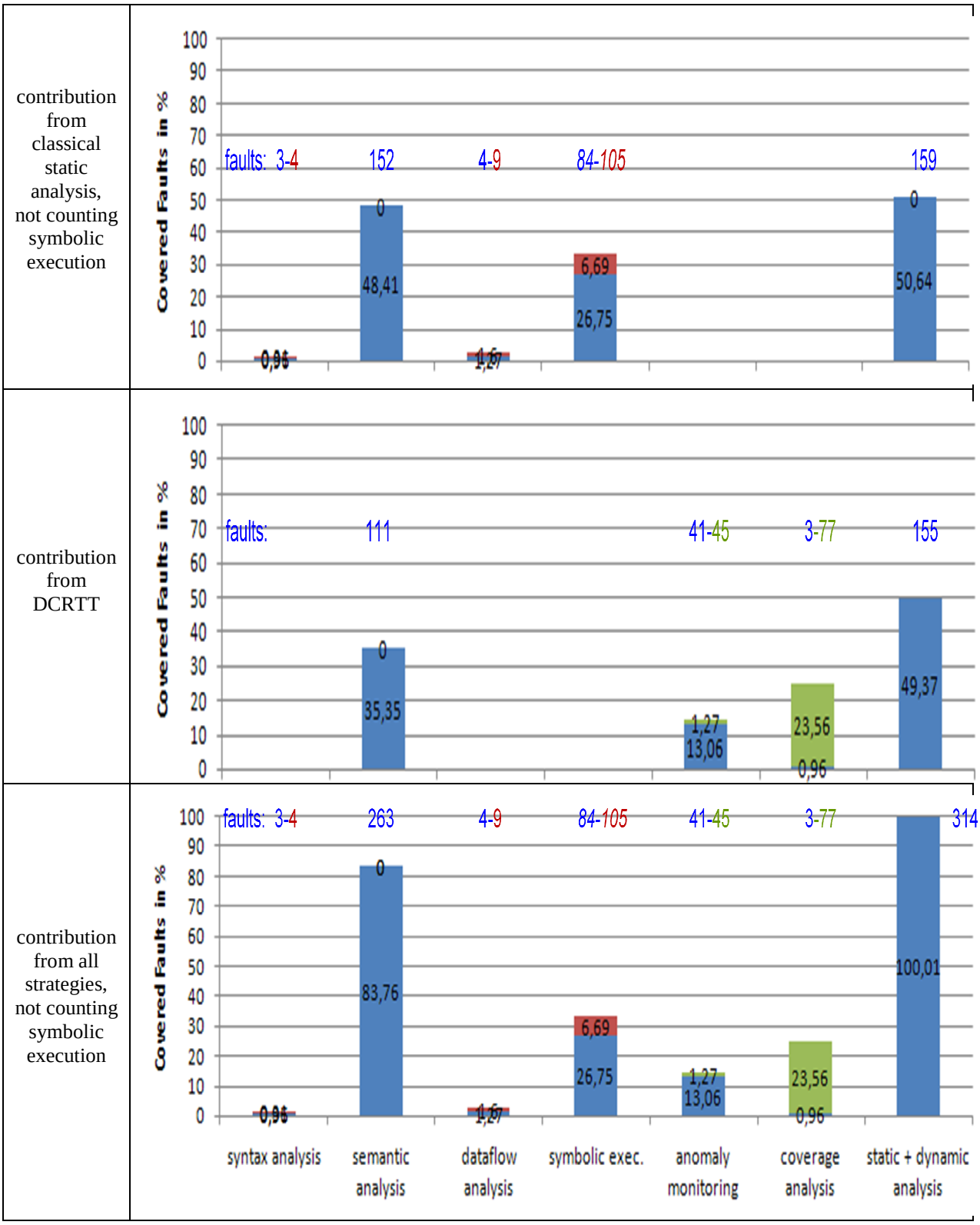


Fig. 8-1: Fault Coverage vs. Methods and Tools

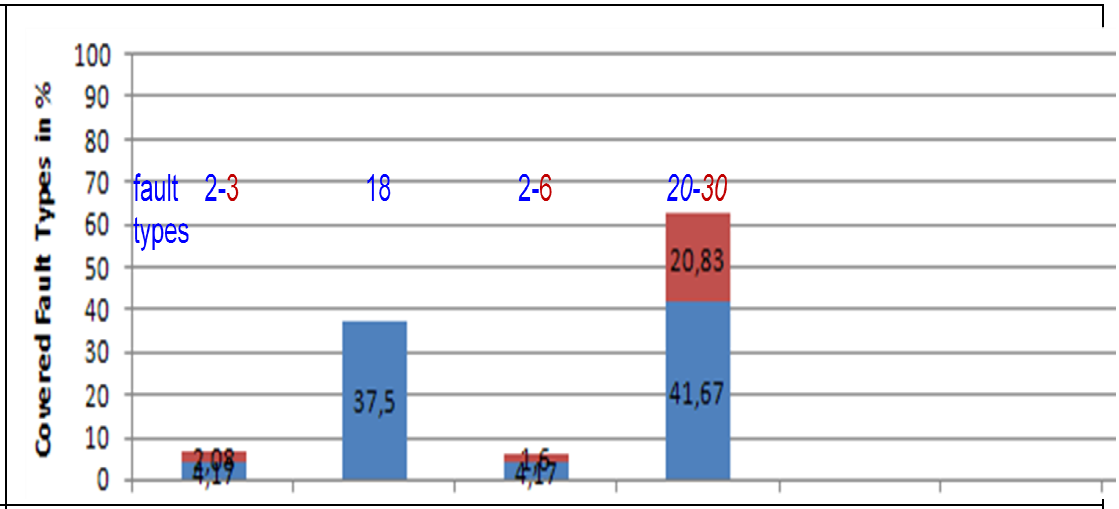
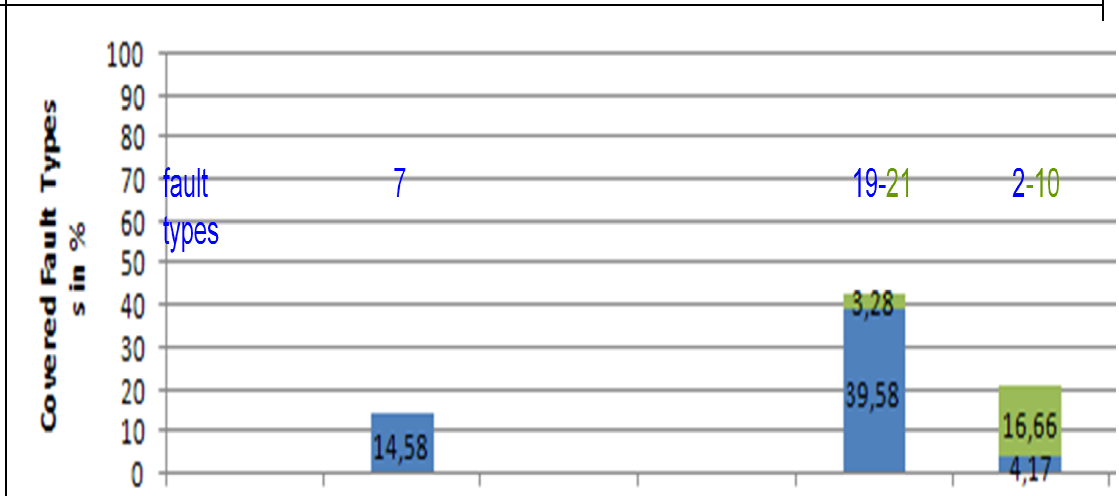
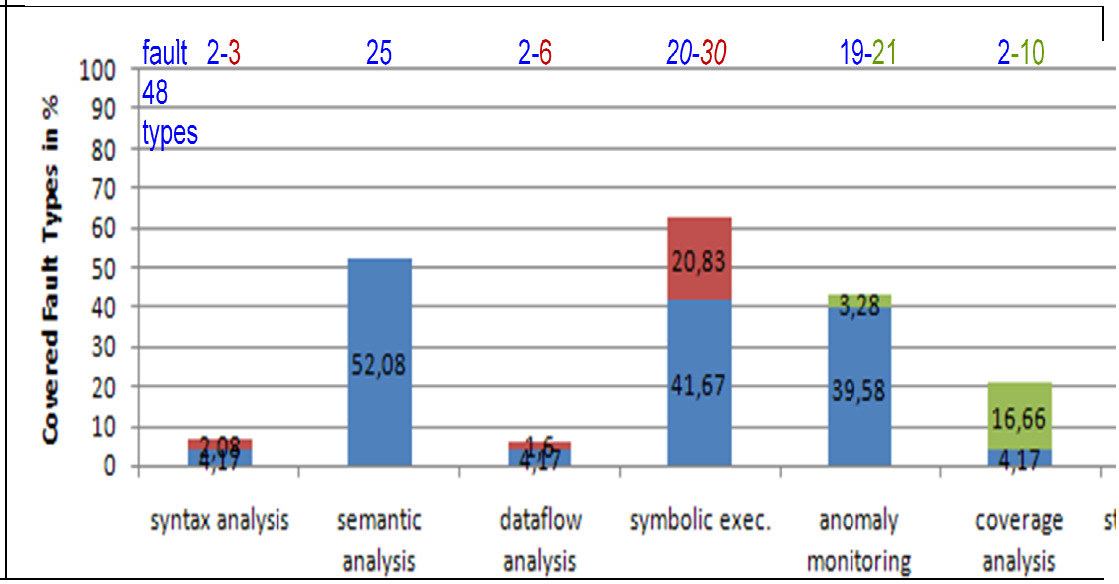
contribution from classical static analysis, not counting symbolic execution	
contribution from DCRTT	
contribution from all strategies, not counting symbolic execution	

Fig. 8-2: Fault Type Coverage vs. Methods and Tools