

Fault Identification Strategies

R.Gerlich, R.Gerlich (BSSE)

C. Dietrich (DLR)

DASIA'09

26.-29.05.2009, Istanbul, Turkey

Dr. Rainer Gerlich
Auf dem Ruhbühl 181
88090 Immenstaad
Germany

Tel. +49/7545/91.12.58
Fax +49/7545/91.12.40
Mobil +49/171/80.20.659
email Rainer.Gerlich@bsse.biz

Overview

- Fault Identification
- Fault Activation
- Identification Assessments
- Conclusions

Goals

Static + dynamic analysis

- Which methods should be applied?
- What is the optimum strategy to minimise the effort and the remaining faults, incl. dormant faults?

Sensitivity of methods and tools

- fault types
- anticipated and non-anticipated faults
- activation conditions of run-time faults
data, events, resources, platform, context
- theory vs. practice

Goals

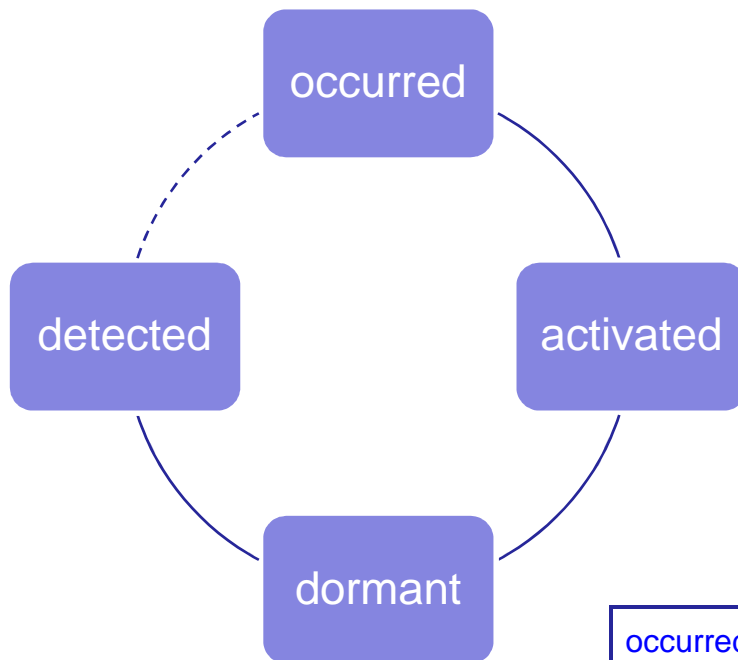
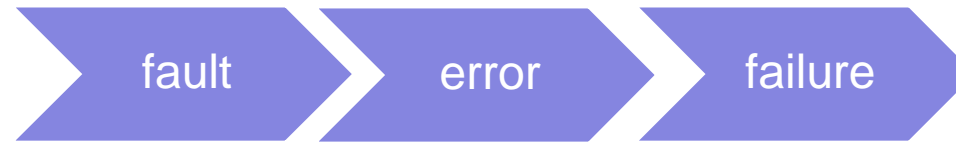
Contribution from fully automated auto-testing (starting with stimulation)

- broad stimulation
- robustness evaluation
- fault injection
- effectiveness of symptom-based fault monitoring
- applied to software after completion of required tests, V&V

Practical results

- activation conditions (Ada, C)
- fault types vs. methods and tools (C)
- fault coverage vs. tools (C)
- statistics on stimuli and test cases (C)

Fault Terminology



occurred	array[j]	index j is out-of -range
activated	array [j] \neq specified value	result is different from what is expected
dormant	array [j] $=$ specified value	result does comply with specified value
detected	array [j] \neq specified value	non-compliance is observed

Risk Assessment

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

no risk at all
provided index < array size

no risk within overall scope
but may change by maintenance
recurring verification effort!

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

no risk
but data corruption possible (i,j)

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

unknown risk
possibly fault propagation

```
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);  
}
```


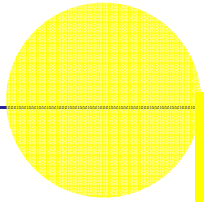

```
E3.1:  
#define ERROR -1  
int myFunc(int i, int j)  
{  
    int k,arr[500];  
    k=i+j;  
    if (k<0 || k>499)  
        k=ERROR;  
    else  
        k=arr[i+j];  
    return k;  
}
```

no risk (?)
depends on fault handling
but fault detection possible
if $arr[m] \geq 0 \forall m \in [0,499]$

Activation Conditions

Activation Conditions				
Depend.	Abbr	Example	Comment	Identif. Strategy
Input	IDF	<pre>int arr[500]; int myFunc(int i, int j) { return arr[i+j]; }</pre>	depends on what comes in	range checking exception monitoring
Platform	PDF	<pre>register short a=20000,b=2000,c; c=a * b; if (c>10){ } else { }</pre>	on a RISC architecture it will not be activated: register size is 32 bit on 16 bit: c<0, possibly overflow	platform diversif. coverage analysis symbolic execution
Context	CDF	<pre>const char cstr[]="123"; typedef enum {false,true} Boolean; void func() { char var[3]; Boolean bool; strcpy(var,cstr); }</pre>	activation may depend on platform / PDF (little-big endian)	platform diversification context change size check
Resource	RDF	<pre>char *str; str =malloc(100); *str=0;</pre>	malloc will return NULL when it lacks memory	fault injection symbolic execution ?
Event	EDF	<pre>void def_handler(int err_code) { switch(err_code) { default: err_handler(err_code); } } void err_handler(int err_code) { switch(err_code) { default: def_handler(err_code); } }</pre>	when a fault occurs with unknown id, recursion will occur.	fault injection symbolic execution
Combinat.			combinations of dependencies	
Independ.	-	<pre>errCode=SUCCESS; // TEST !!!! if (errCode==ERROR)</pre>	will always occur	coverage analysis symbolic execution dataflow analysis

Fault Identification Strategies

Fault Identification Strategy		Fault Manifestation
Syntactic Analysis		 error or warning message, compilation abort
Semantic Analysis		
Dataflow Analysis		
-----		-----
Symbolic Execution		error or warning message
Stimulation	parameter + data	 exception, abort, lock
Stimulation + Fault Injection	parameter + data	
	return values	
Range Checks		 DCRRTT msg.
Checks on memory corruption		
Platform diversification		exceptions, compiler messages, DCRRTT msg.
Coverage		coverage figures < 100% and red-coloured parts in graphics (DCRRTT), branching ratios / statistics

anticipated faults only

message-based
no platform-impact

symptom-based
platform impact

anticipated
+
non-anticipated faults




Pre-Run-Time Detection

Fault Ident.	Activation or Detection	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 invalid types in assignment missing variable declaration inconsistent interfaces inconsistent declarations types too small/big for used range
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 <u>absence</u> 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

Run-Time and Post-Run-Time Manifestation

Fault Ident.	Activation or Detection	Fault Manifestation	Source of Fault (non-exhaustive list)
Stimulation	variation of parameter and 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 und 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 specific DCRTT support	DCRTT msg.	out-of-range
Checks on memory corruption	Check on corruption of mallocated memory <i>specific DCRTT support</i>	DCRTT msg.	change of data outside the portion of allocated memory
Platform diversification	variation of OS, processor, compiler or memory allocation <i>specific DCRTT support</i>	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 <i>specific DCRTT support</i>	coverage figures<100% and red-coloured parts in graphics (DCRTT)	dead code faults in <i>logical expressions</i> , undetected by pre-run-time tools

Method and Tool Assessment

Example	Identification Strategy	Scope	Identification Reliability	
			theory	observ.
<pre>#define FILE_PATH "disk:/dir/" if (FILE_PATH == NULL) { } else { }</pre>	Coverage Analysis	FUT	sure	yes
	Semantic Analysis / Constant Propagation	unit	sure	no 
<pre>ret_value=SUCCESS; //TEST!! if(ret_value == ERROR) { // then-branch } else { // else-branch }</pre>	Coverage Analysis	FUT	sure	yes
	Dataflow Analysis / Constant Propagation	unit	sure	no 
	Symbolic Execution	calls	sure	?
<pre>long size,*getSize = NULL; if (readBuf(fd,offset,(void*)getSize, sizeof(long))== ERROR) { ... } else{ size=*getSize;}</pre>	RT anomaly + Fault Inj.	FUT	high / CDF + PDF	yes
	Dataflow Analysis / Constant Propagation	unit	sure	no 
	Symbolic Execution	calls	sure	?

Source Code and Analysis Tools

Figures represent snapshot,
but qualitative conclusions
remain valid in general

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

← results presented in detail

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

Source Code and Analysis Tools

Figures represent snapshot, but qualitative conclusions remain valid in general

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

← results presented in detail

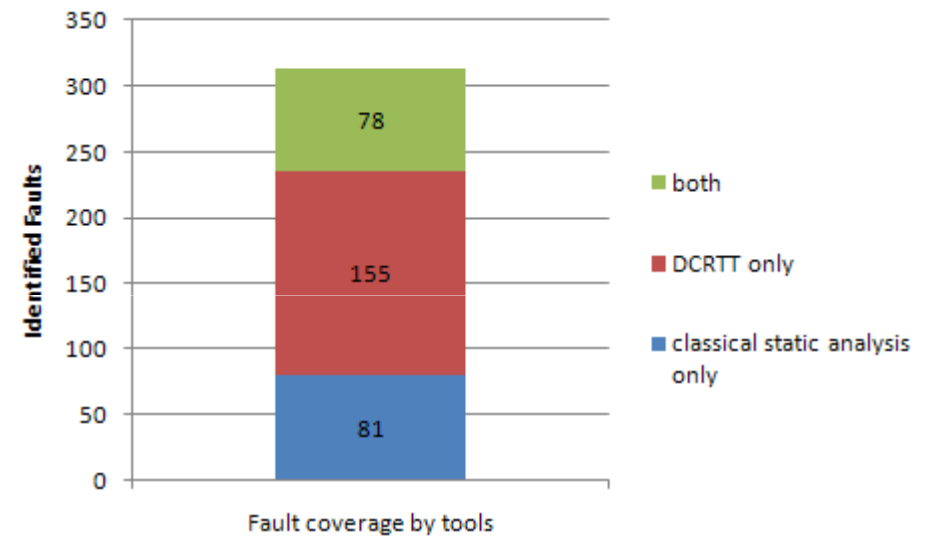
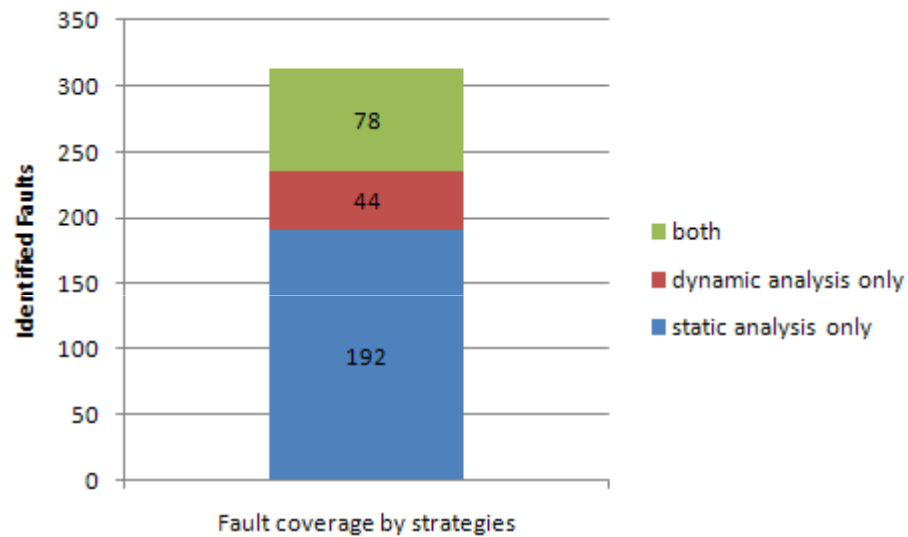
Tool		Method					
		Static Analysis				Dynamic Analysis Auto-Testing	
		syntax	semantic	dataflow	symbolic execution	anomaly monitoring	coverage evaluation
static	gcc compiler	x	x	x			
	gcc linker		x				
	Cantata++		x	x			
	theoretically				x		
dynamic / test auto-testing	DCRTT		x			x	x

Analysis Method	Faults Covered, abs.	Faults Covered, %
static analysis incl. DCRTT, only	192	61,2
dynamic analysis, only	44	14,0
both	78	24,8
total	314	100

Analysis Method	Faults Covered, abs.	Faults Covered, %
classical static analysis, only	81	25,8
DCRTT, only	155	49,4
both	78	24,8
total	314	100

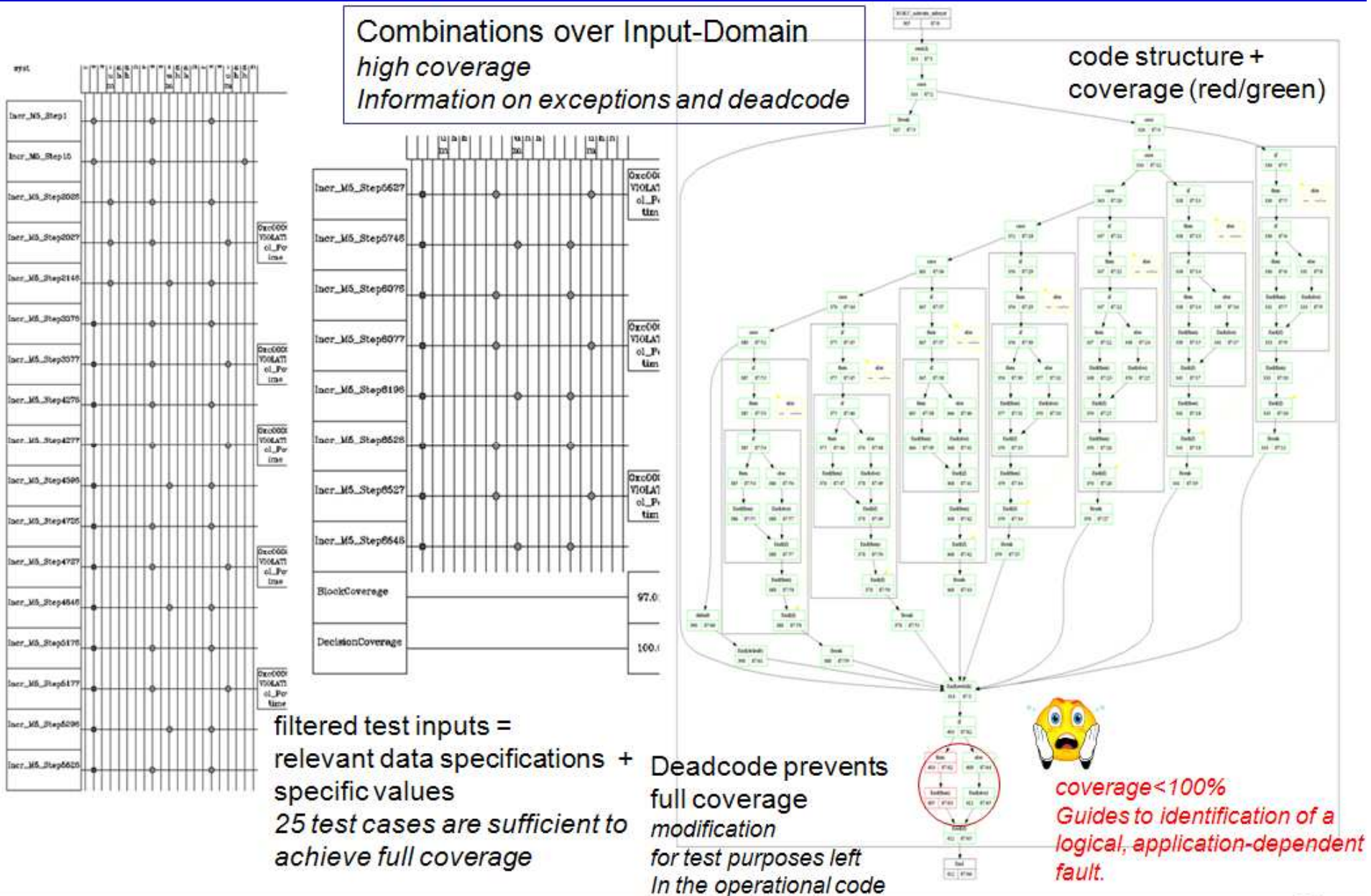
Source Code and Analysis Tools

Figures represent snapshot, but qualitative conclusions remain valid in general



Faults were identified after operation of software in space

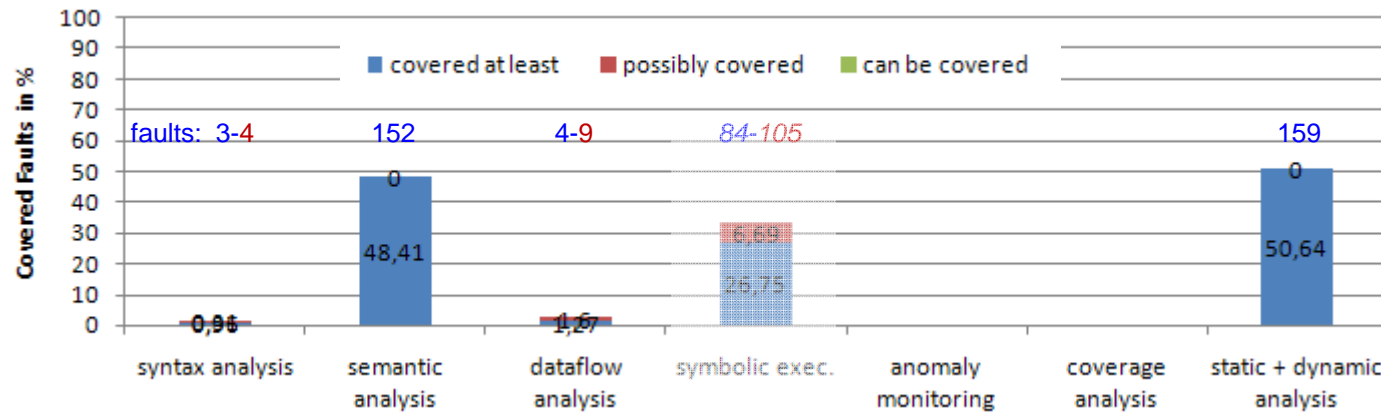
Test Coverage and Filtering



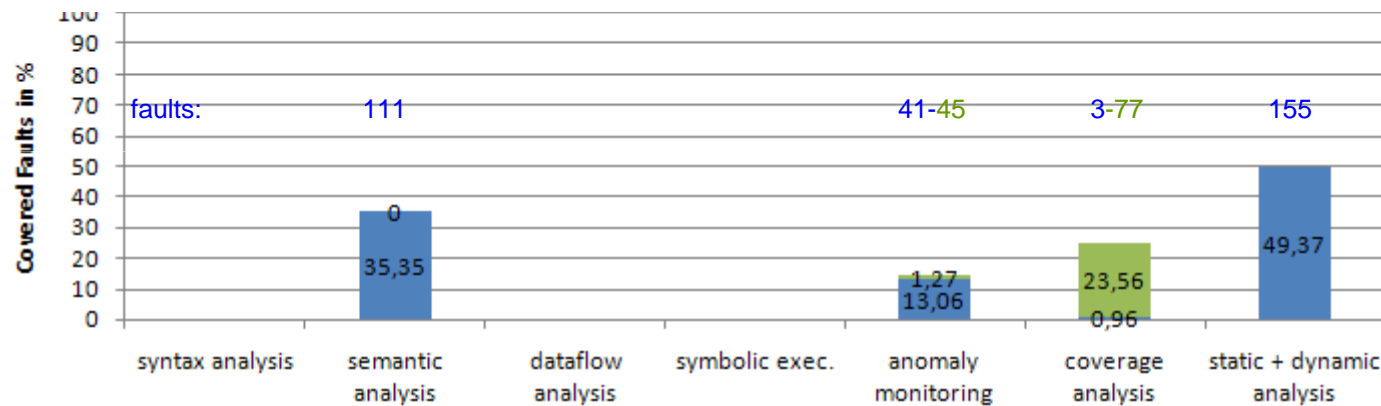
Figures represent snapshot, but qualitative conclusions remain valid in general

Fault Coverage vs. Methods and Tools

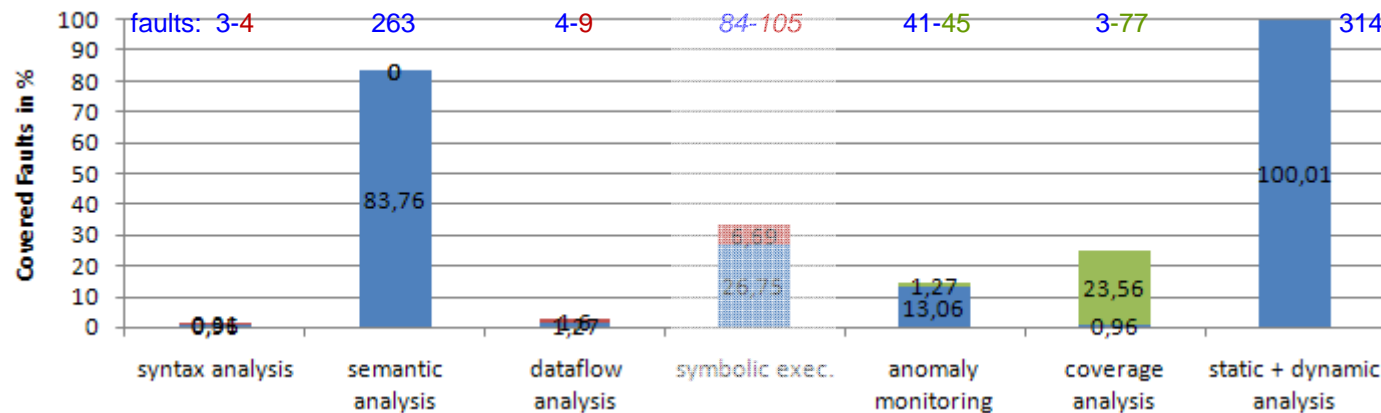
contribution from classical static analysis excl. symbolic execution



contribution from DCRTT



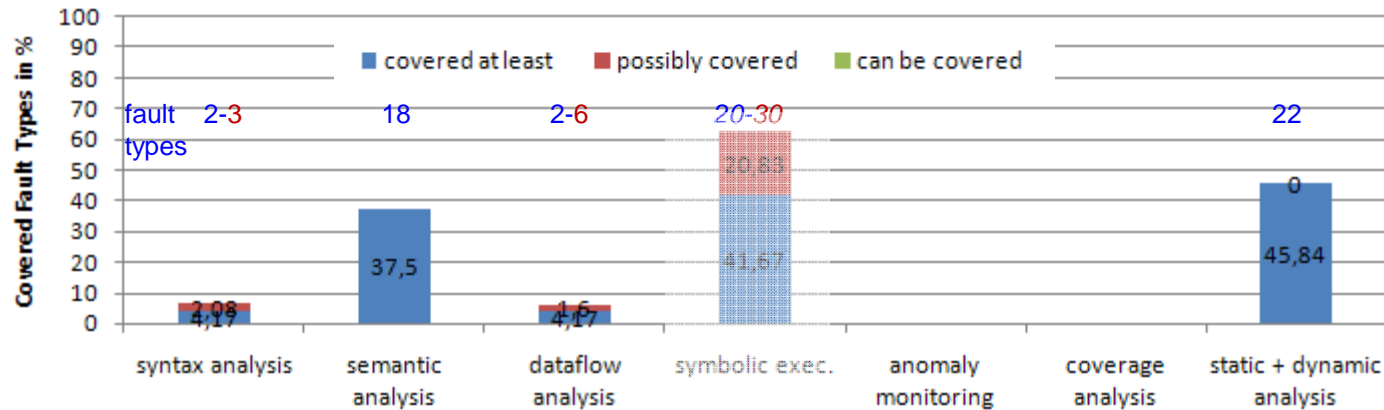
contribution from all strategies excl. symbolic execution



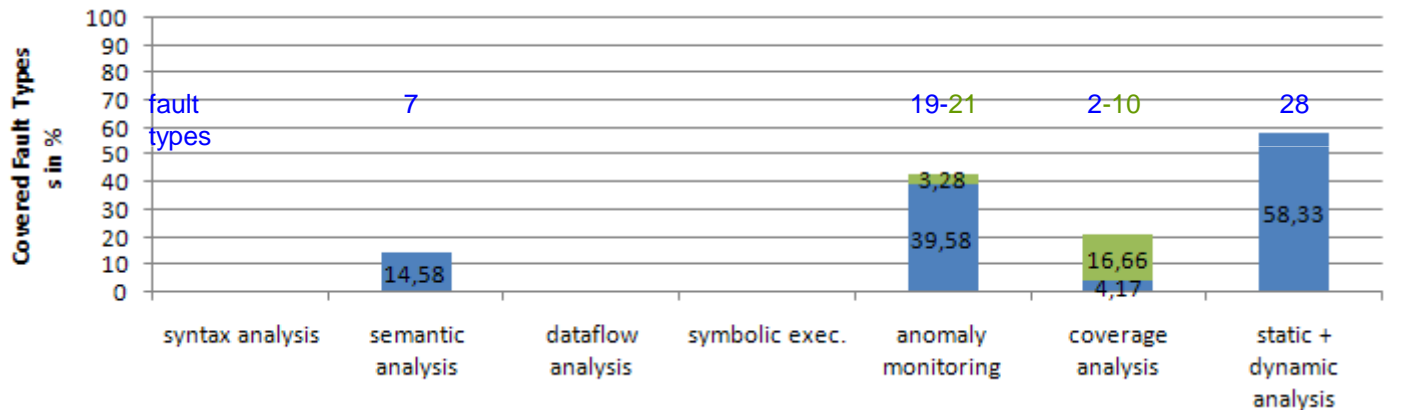
Figures represent snapshot, but qualitative conclusions remain valid in general

Fault Type Coverage vs. Methods and Tools

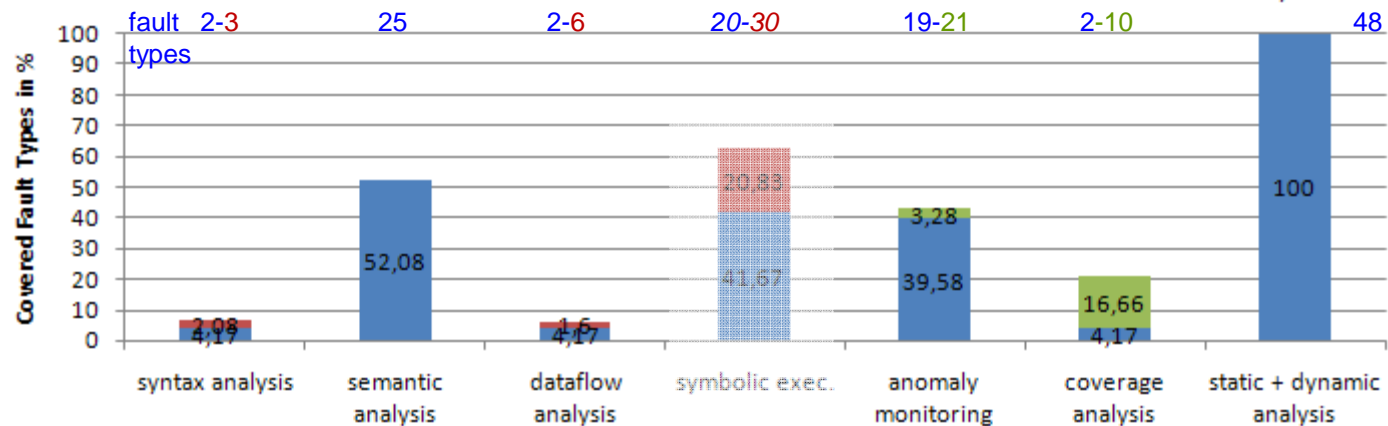
contribution from classical static analysis



contribution from DCRTT



contribution from all strategies w/o symbolic execution



Figures represent snapshot, but qualitative conclusions remain valid in general

Fault Coverage vs. Methods and Tools

Total faults	314	Static Analysis				Dynamic Analysis			
		Syntax	Semantic	Dataflow	Symbolic Execution	Detection Method		Stimulation Method	
						Anomaly	Coverage	Data	Platform
faults covered by classical static analysis methods	abs, min	3	152	4	84				
	abs, max	4	152	9	105				
	%, min	0,96	48,41	1,27	26,75				
	%, max	1,27	48,41	2,87	33,44				
faults covered by DCRTT	abs, min		111			41	3	27	1
	abs, max		111			45	77	27	1
	%, min		35,35			13,06	0,96		
	%, max		35,35			14,33	24,52		
faults covered in total	abs, min	3	263	4	84	41	3	27	1
	abs, max	4	263	9	105	45	77	27	1
	%, min	0,96	83,76	1,27	26,75	13,06	0,96		
	%, max	1,27	83,76	2,87	33,44	14,33	24,52		

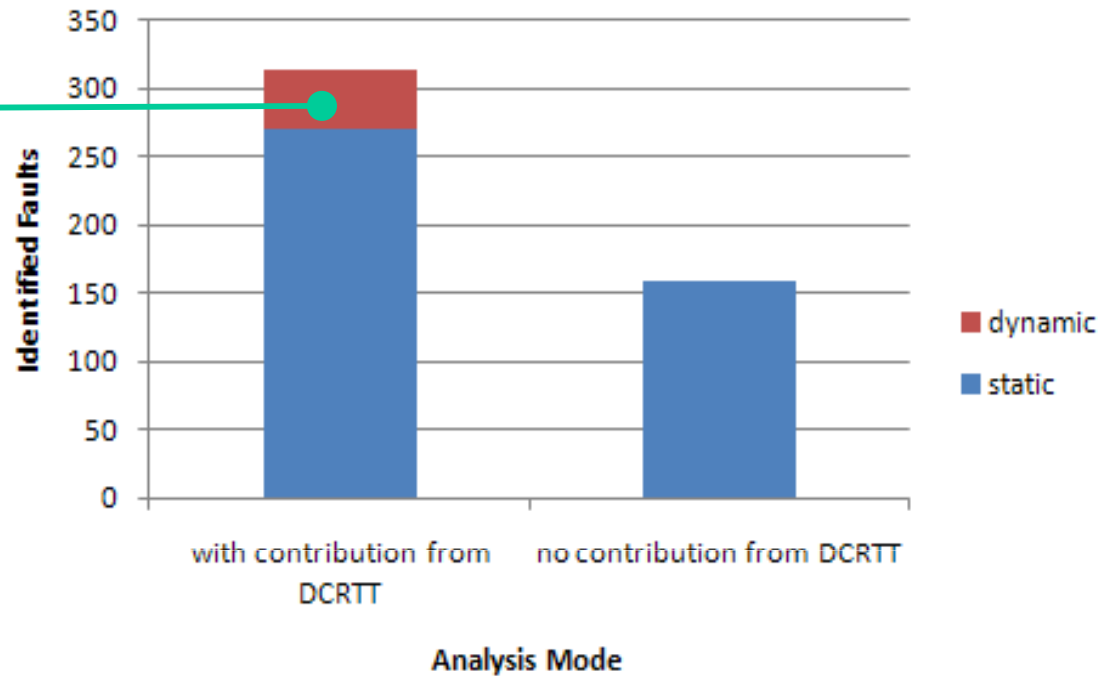
Total fault types	48	Static Analysis				Dynamic Analysis			
		Syntax	Semantic	Dataflow	Symbolic Execution	Detection Method		Stimulation Method	
						Anomaly	Coverage	Data	Platform
fault types covered classical static analysis methods	abs, min	2	18	2	20				
	abs, max	3	18	6	30				
	%, min	4,17	37,5	4,17	41,67				
	%, max	6,25	37,5	12,5	62,5				
fault types covered by DCRTT	abs, min		7			19	2	9	1
	abs, max		7			21	10	9	1
	%, min		14,58			39,58	4,17		
	%, max		14,58			42,86	20,83		
fault types covered in total	abs, min	2	25	2	20	19	2	9	1
	abs, max	3	25	6	30	21	10	9	1
	%, min	4,17	52,08	4,17	41,67	39,58	4,17		
	%, max	6,25	51,02	12,5	62,5	42,86	20,83		

Fault Coverage Summary

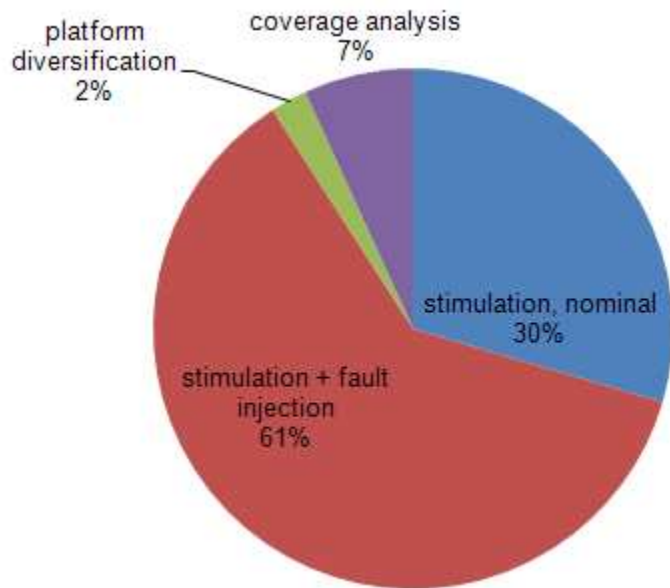
These 44 faults identified by auto-testing cannot be covered by other strategies at all.

But auto-testing could cover 122 faults. The difference of 78 faults is not allocated here, as static analysis strategies may be more efficient to identify the sources due to messages pointing directly to the location in the source code.

122 faults.



Fault Coverage vs. DCRTT Identification Strategies



Fault Coverage Summary

Faults Detected with DCRTT Dynamic Analysis Only			
Symptom-based Fault Identification Method	Applied at	# Faults	%
Recording of exceptions, aborts, deadlocks, livelocks and specific DCRTT run-time checks	run-time	13	29,55
As above + fault injection	run-time	27	61,36
As above + platform diversification	run-time	1	2,27
Coverage analysis	post-run-time	3	6,82
Total		44	100

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

Statistical Figures

Some Statistics		
Test case filtering	1 : 1000	this ratio is representative for the mapping of samples in the input domain onto equivalence classes
Test cases per function	7	average, estimated for full coverage
LOC per test case	8	average, estimated for full coverage
Overall number of test cases	5,000	to achieve full coverage FMECA may still be applied to select test cases which need to be verified against the technical or functional specification, then the benefit still is that all code has been executed and checked on robustness
Estimated manual effort per test case <i>(very conservative)</i>	1 man-hour	for identification of test case, test preparation and execution, verification of output vs. input and specification is not included, seems to underestimate the real effort
Estimated savings due to automation from stimulation to evaluation and filtering	2 man-years 300 k€	conservative estimate: per 40 KB of C code and assumption of 1 LOC/man-hour \Rightarrow 40,000 man-hours \Rightarrow 25 man-years \Rightarrow 10% savings of total costs (2 instead of 25), savings probably higher

Conclusions

Auto-testing

- raises significantly the identification probability of
 - faults in fault handlers due to fault injection
 - sporadic faults due to broad stimulation addressing exotic cases
- auto-identification of test cases by criteria, generation of test drivers
- symptom-based analysis covers non-anticipated faults
- application-independent criteria to identify application-dependent faults without oracle coverage, statistics on code execution

Sensitivity of Methods and Tools

- symptom-based identification turns out as feasible and efficient
- coverage of a fault type may depend on practical implementation and limitations
- anticipated faults + high tool complexity vs. non-anticipated faults + low complexity
- potential difference between theory and practice regarding fault coverages

Conclusions

Fault Identification Strategy

- none of the current methods and tools can cover all fault types
- give preference to static analysis, then apply dynamic analysis / testing
- do vary conditions (platform, context, fault injection, broad stimulation)
- diversification strongly recommended, especially for critical software

Recommendation

- for proof of correctness (per test case), the intended or an equivalent platform is needed
- for identification of faults everything is allowed what helps to activate and detect a fault

Conclusions

Sensitivity of Fault Identification

- each strategy allows only specific conclusions with limited scope regarding fault found and not found
- faults expected to be found theoretically, may not be found in practice
- practical evaluation needed on the sensitivity of methods and tools