Evaluation of
Auto-Test-Generation Strategies and Platforms

R.Gerlich[(1)], R.Gerlich[(1)(2)], Th.Boll[(1)], J.Mayer[(2)]

[(1)]BSSE [(2)]University of Ulm

DASIA'07

29.05. - 01-06.2007, Naples, Italy

Dr. Rainer Gerlich
Auf dem Ruhbühl 181
88090 Immenstaad
Germany

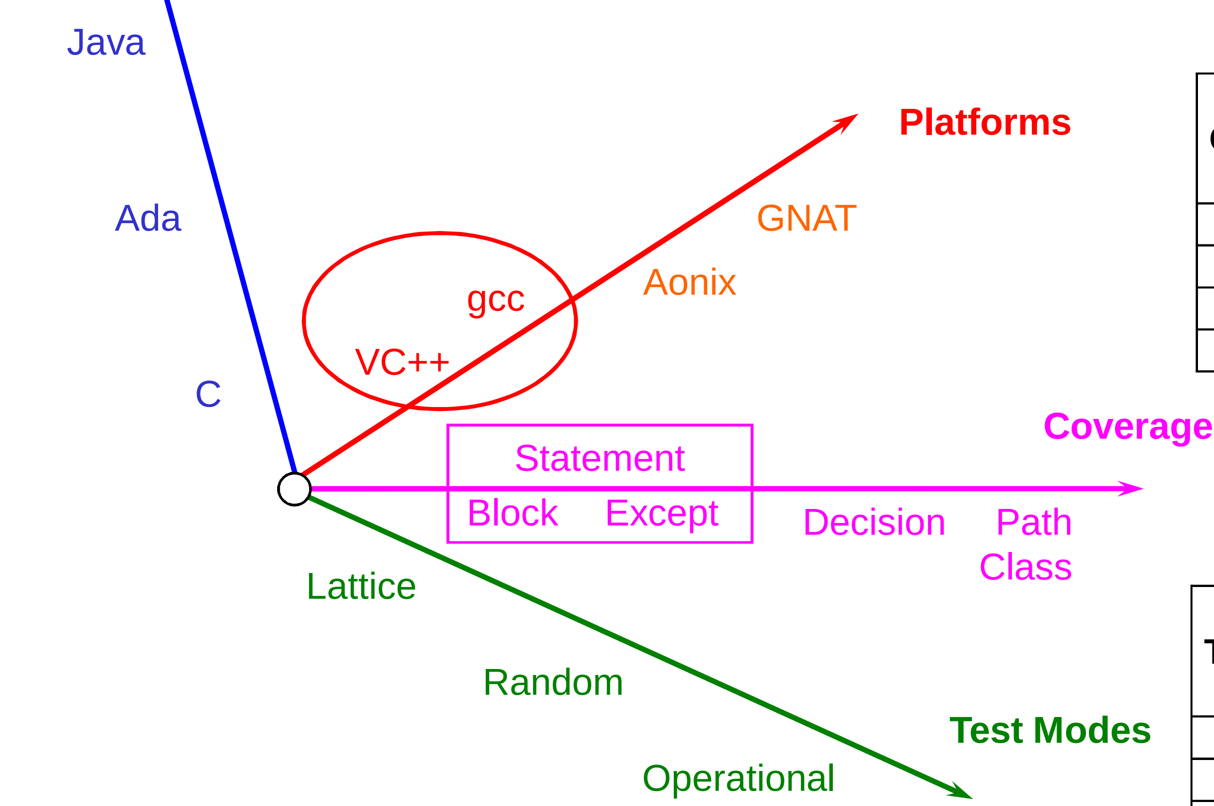
Tel. +49/7545/91.12.58
Fax +49/7545/91.12.40
Mobil +49/171/80.20.659
email Rainer.Gerlich@bsse.biz

Overview

- Test Strategies
- Platform Dependencies
- Auto-Testing Results
- Conclusions

Dimensions of Auto-Testing

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



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

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

Coverage

- **Block coverage**
  - record when a block is accessed
  - 1 .. n samples in a “basket”
  - n user-defined, usually 1 “*sufficient*”, but more needed
  - figures presented are based on n=1
- **Exception coverage**
  - record when an exception occurs
  - take each exception type in any case
    (exception code, location)
- **Statement Coverage**
  - identical with block coverage, if no exception occurs
  - equivalent to combination of block + exception recording
- **Decision Coverage**
  - record all items impacting branches (if, switch, for, while)
  - short circuit code, MC/DC
- **Path Set Coverage**
  - identify paths to a block
  - ☞ much more combinations than for block and statement coverage
  - ☞ but more reliable test coverage
  - 1 .. n samples in a “basket” per path set

Path Set Coverage

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

- **Path sets constructed by transformation of code**
  - equivalence transformation (e.g. loop-unrolling, unfolding, ...)
  - insertion of constraints to enforce decisions (e.g. `<loopcond>=true`)
- **constraint-based test data generation** (starting point: Gotlieb et al, 2001)
  - extended to path set coverage using transformed code (statement coverage)
- **numbers lead to combined strategy**
  - first random/lattice: fast (~3000/s), but often incomplete coverage
  - then constraint-based: slow, but complementary in coverage
- **future optimisations**
  - optimise constraint solver for inconsistency detection (proof by refutation)
  - path-look-ahead based on control-flow-graph properties

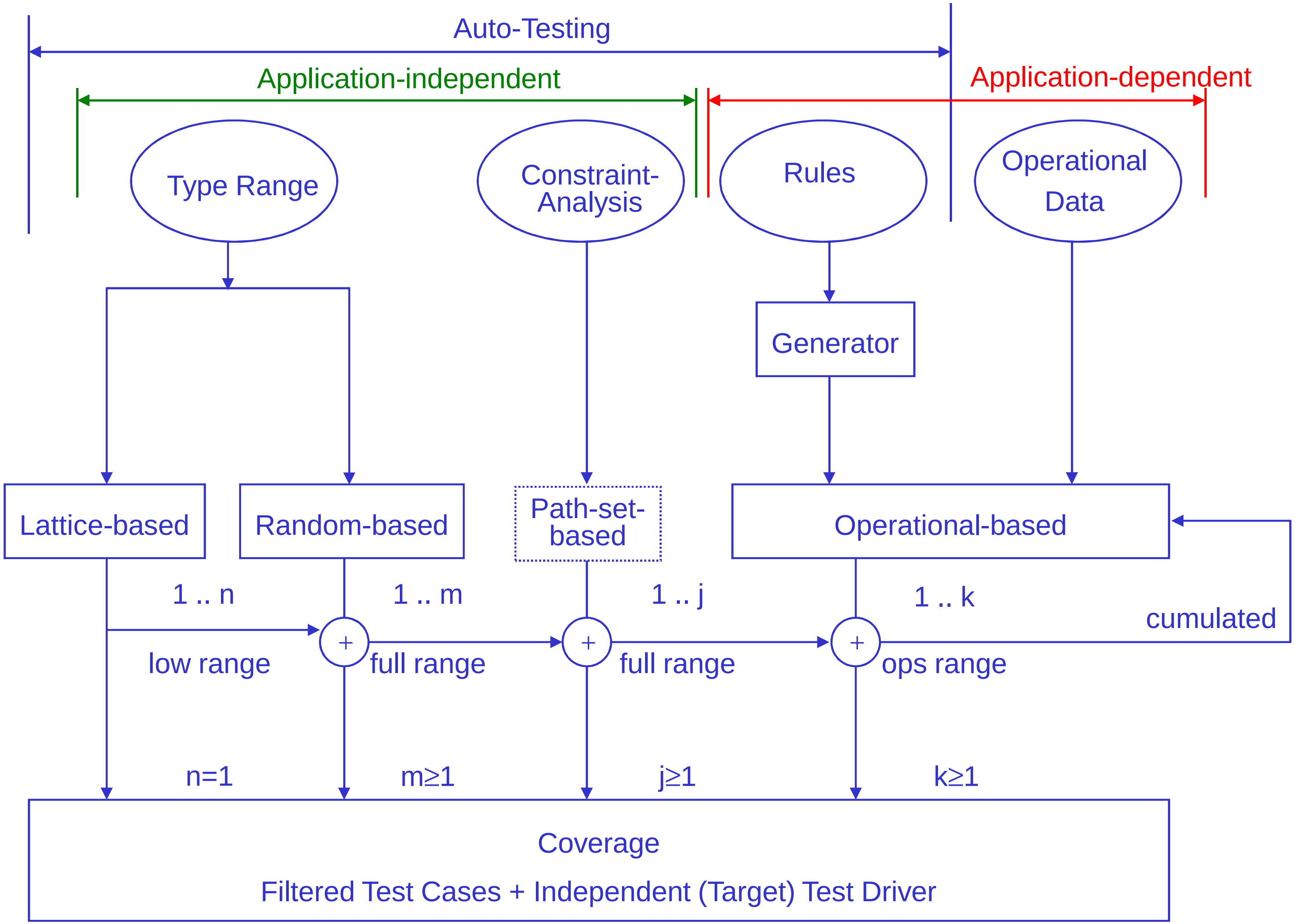
Test Modes (1/2)

- **Lattice (black-box) (subprogram parameters)**
  - type range is divided into n intervals
  - position of samples may be driven by a weight profile
    more samples around a user-defined center
  - full coverage from type’first .. type’last
  - ☞ good results for out-of-range-conditions at lower and upper limit
  - coverage filter: lower values are preferred
- **Random (black-box) (subprogram parameters)**
  - (pseudo) random choice over type’first .. type’last
  - currently no weights
  - coverage filter: random distribution
- **Extension: information from code analysis (white-box)**
  - additional test cases (lattice + random)
  - constants found in source code

Test Modes (2/2)

- **Operational Mode**
  - running a program in normal operation
  - collection of coverage for all subprograms simultaneously
  - case-to-case: input generation according to specification
  - flex: applications-specific generator according to parsing rules
  - test cases are complementary to lattice + random modes
- **Future extensions: path set + global data + stack data**
  - outcome from path class coverage activities
  - identify criteria to enter a branch
  - ☞ based on constraint-solving techniques
  - “simple” conditions are covered by “normal” lattice- and random based test generation
  - “complex” conditions are identified by constraint-solving techniques matter of CPU time consumption
  - also consider global and stack data
  - ☞ auto-testing should come close to 100% coverage

Auto-Test Strategies

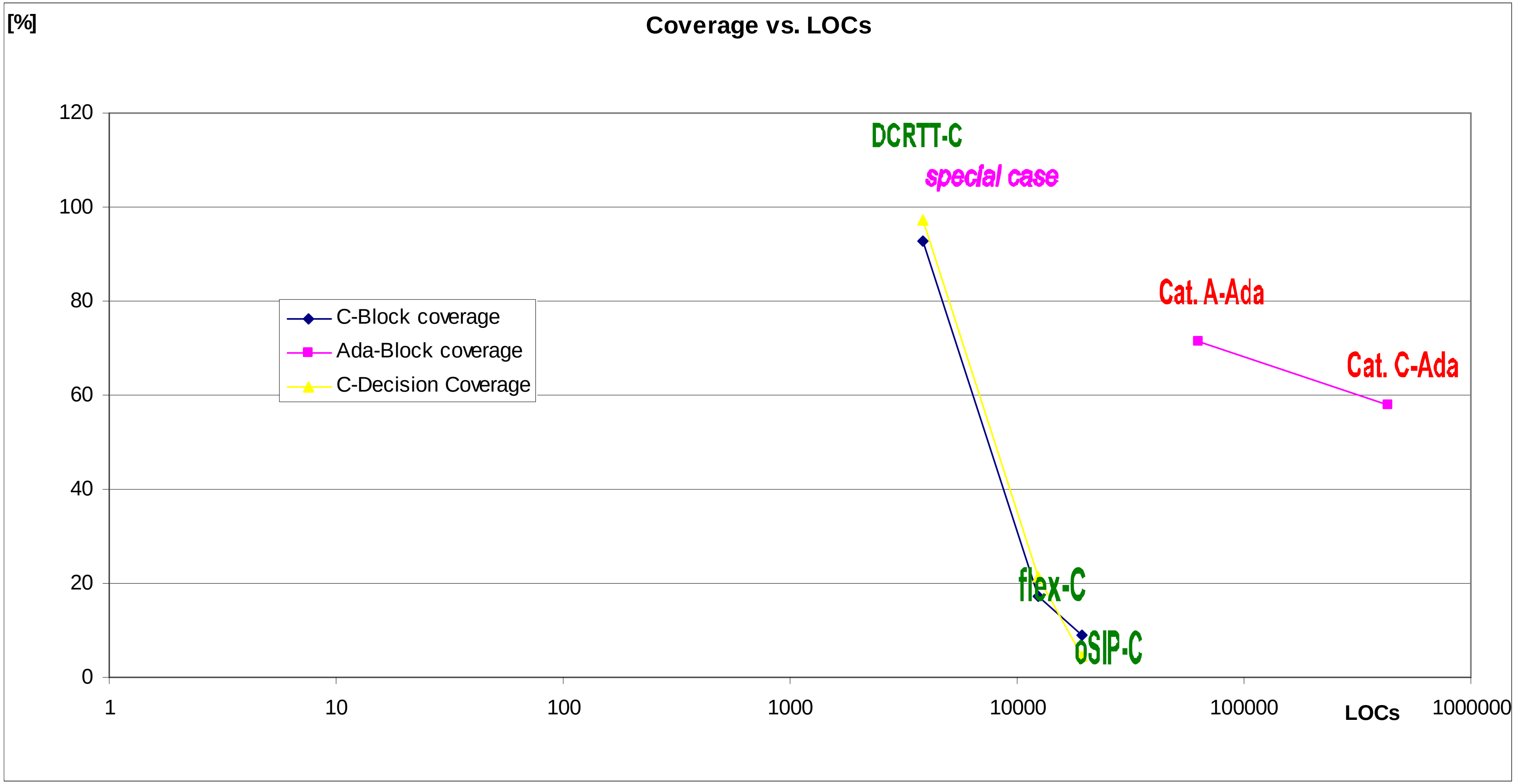


Systems-under-Test

- **DCRTT Test Suite**
  - test cases for critical issues of auto-testing
  - nature of code leads to high coverage
  - demonstration of non-reachable code: total coverage < 100%
  - demonstration of exception capture: significant part of exceptions
- **Open Source Packages**
  - open to everybody to re-run tests
  - comparison of results from different tools (oSIP⇔DART)
- **GNU oSIP**
  - open software for the Session Initiation Protocol (SIP)
- **flex, Berkeley University**
  - parser, code generator

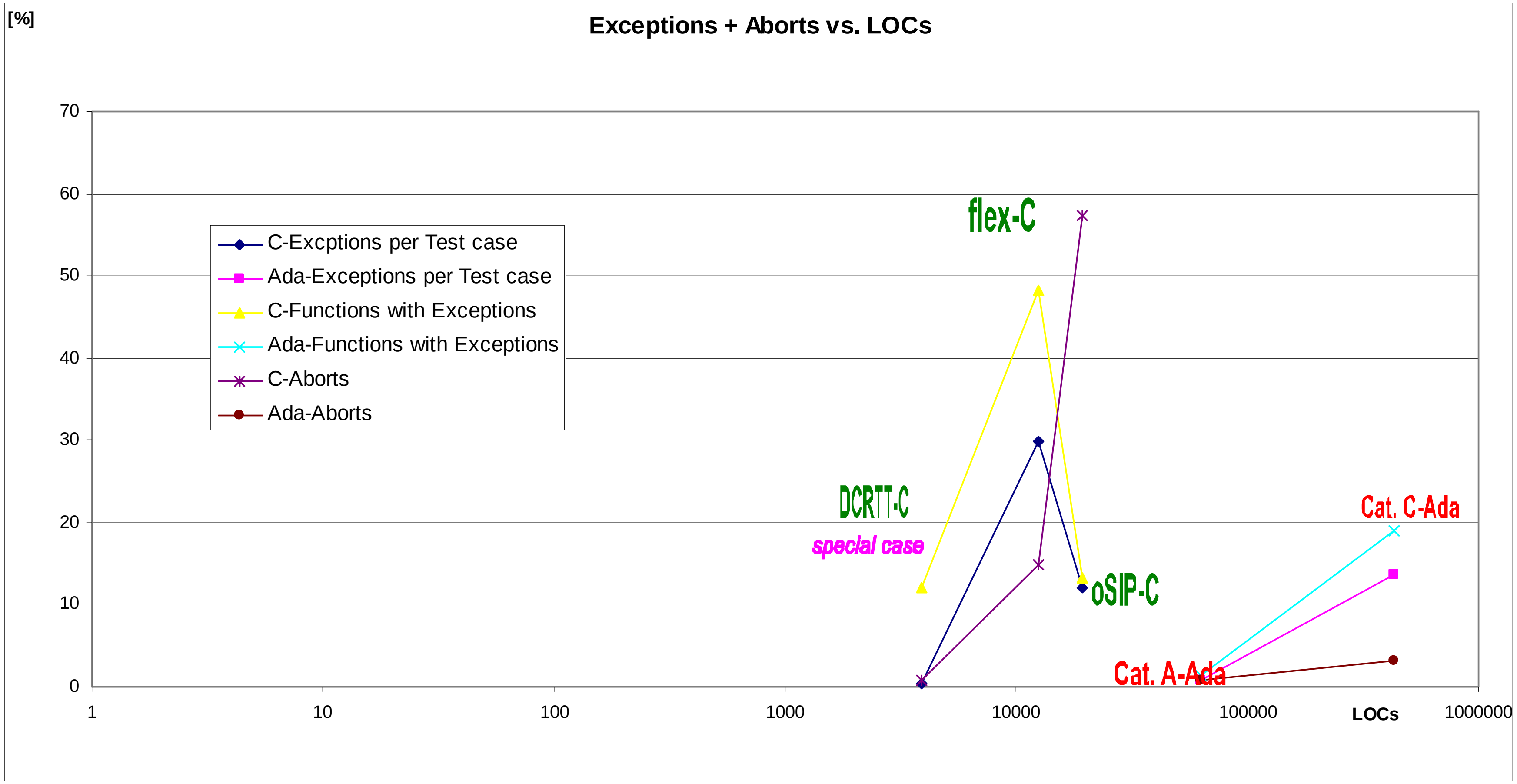
	Functions	LOC	Blocks	Decisions
DCRTT	142	3862	865	938
flex	189	12452	2397	2871
oSIP	655	19368	3402	5227

Overview on Coverage



- The more defensive the programming style ⇒ the higher the coverage
- The more information on type ranges ⇒ the higher the coverage
- Ada better than C
- DCRTT test suite is a special case: adherent to defensive programming style

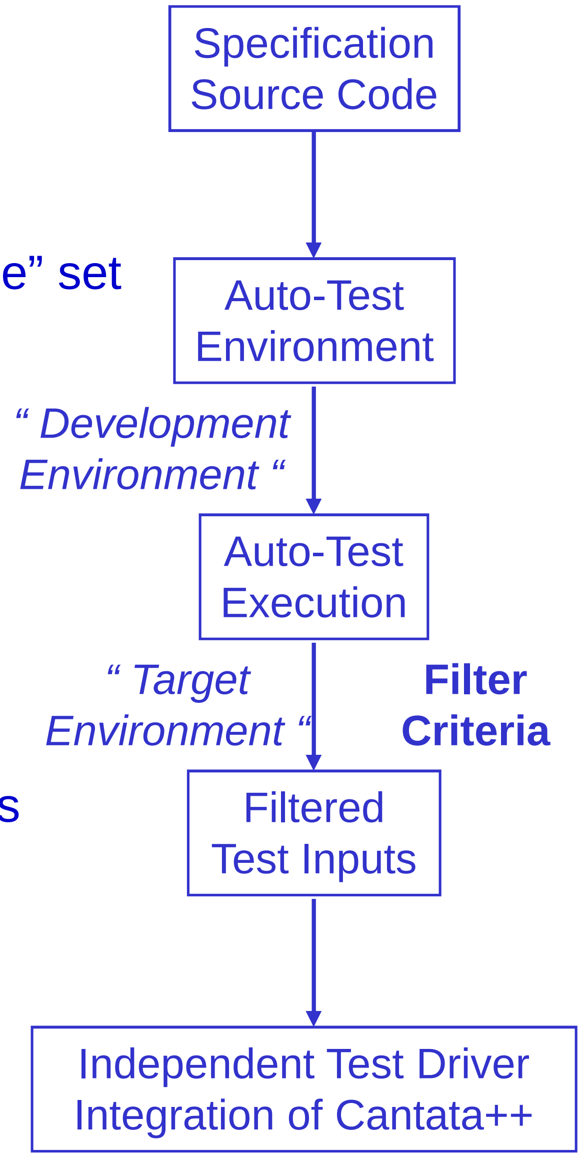
Overview on Locks and Aborts



- The more defensive the programming style ⇒ the less anomalies
- The more context information ⇒ the less anomalies
- Ada code: developed according to standards
- DCRTT test suite is a special case: intended generation of exceptions, locks, aborts

Test Case Filtering: Approach

- **Filtering**
  - ❖ identify test inputs of interest
  - ☞ auto-testing produces a high number of test inputs
    take coverage criterion to reduce this set to a "feasible" set
      - n samples for each block and decision item
      - each exception type
  - ❖ reduced set can be evaluated manually (should be)
- **Test driver generation**
  - ❖ auto-generate independent test driver
  - ❖ auto-feed in recorded inputs
  - ❖ auto-check output against previously observed outputs
  - ❖ run test driver on target or another platform
  - ❖ integrate test driver with another (test) tool to benefit from complementary capabilities
  - ☞ integration with Cantata++



Test Case Filtering: Results

Test Cases DCRTT	VC++		gcc	
	lattice	random	lattice	random
Total Samples	552339	428318	552342	428318
Filtered	769	626	736	600
Non-compliances	3	3	0	0

Test Cases flex	VC++		gcc	
	lattice	random	lattice	random
Total Samples	525660	492489	533070	487122
Filtered	359	328	365	313
Non-compliances	101	101	47	39

- **Platform aspects**
  - diversification brings more filtered test cases
  - a priori: unknown which one is the best ...

- **Test re-execution**
  - execution of filtered test inputs by independent test driver
  - re-evaluation by independent tool
  - non-compliances indicate computational non-determinisms, exception type and location
  - varying test conditions:

    memory, exception sensitivity, numerics

Platform Dependencies: Exceptions, Locks and Aborts

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

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

- **Exceptions**
  - activation compiler-dependent
  - numerics
  - differences indicate numerical weakness + instability

- **Locks + Aborts**
  - identify dormant problems
  - context / status dependency
  - differences indicate weakness + instability

Locks + Aborts	VC++		gcc	
	lattice	random	lattice	random
DCRTT	intended	intended	intended	intended
#	(1)	(1)	(1)	(1)
%	-	-	-	-
flex	28+10	17+12	14+14	12+16
#	38	29	28	28
%	20.12	15.35	14.81	14.81
oSIP			15+326	
#			341	
%			52.06	

Platform Dependencies: Coverage

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

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

oSIP 655 functions	Coverage / %			
	Lattice		Random	
Coverage Type	VC++	gcc	VC++	gcc
Block		8.94 ??		
Decision		4.36 ??		
true		34.65 ??		
false		76.75 ??		

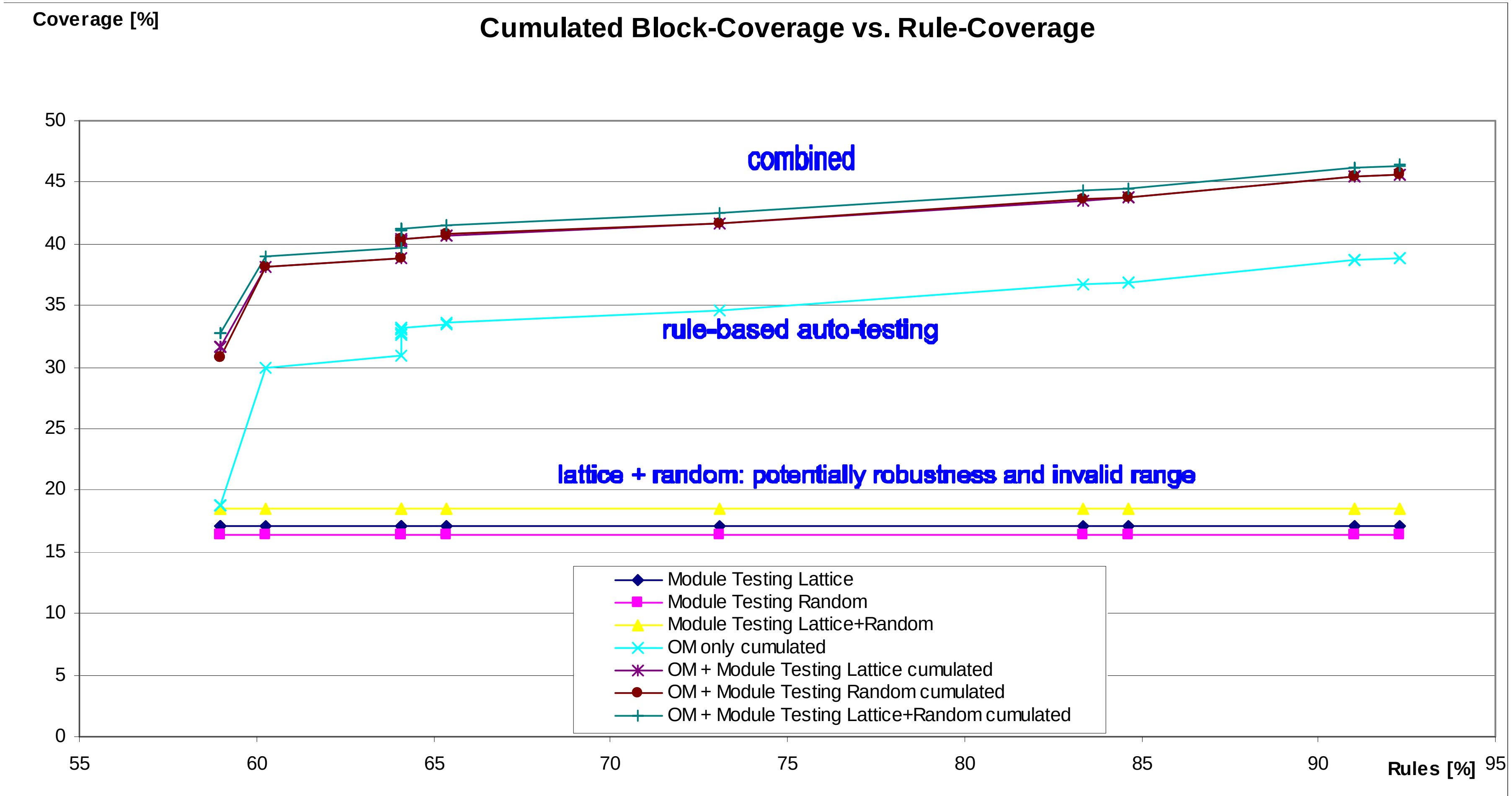
- **General considerations**
  - the more information about valid operation conditions, the higher the coverage
  - impact by exceptions
  - gcc: higher coverage, less exceptions
- **flex**
  - poor context information ⇒ low coverage + high exception rate
  - can be improved by adherence to coding standards
- **oSIP**
  - further evaluation dropped due to high abort rate
  - results may be corrupted due to crashes FUT, re-run required, ~18 h

Test Strategy vs. Coverage

flex gcc Rule coverage max. = 92.31%	Coverage / %	
Test Mode	Block	Decision
Lattice	17.2	21.3
Random	16.5	19.3
Lattice + Rnd	18.6	22.8
Operational Mode (OM) max.	29.58	42.95
Latt + OM max.	37.46	49.43
Rnd + OM max.	37.55	49.32
Latt + Rnd + OM max.	38.42	49.57
OM cumulated	38.82	49.84
Latt + OM cumulated	45.64	55.59
Rnd + OM cumulated	45.72	55.77
Latt + Rnd + OM cumulated	46.43	55.73

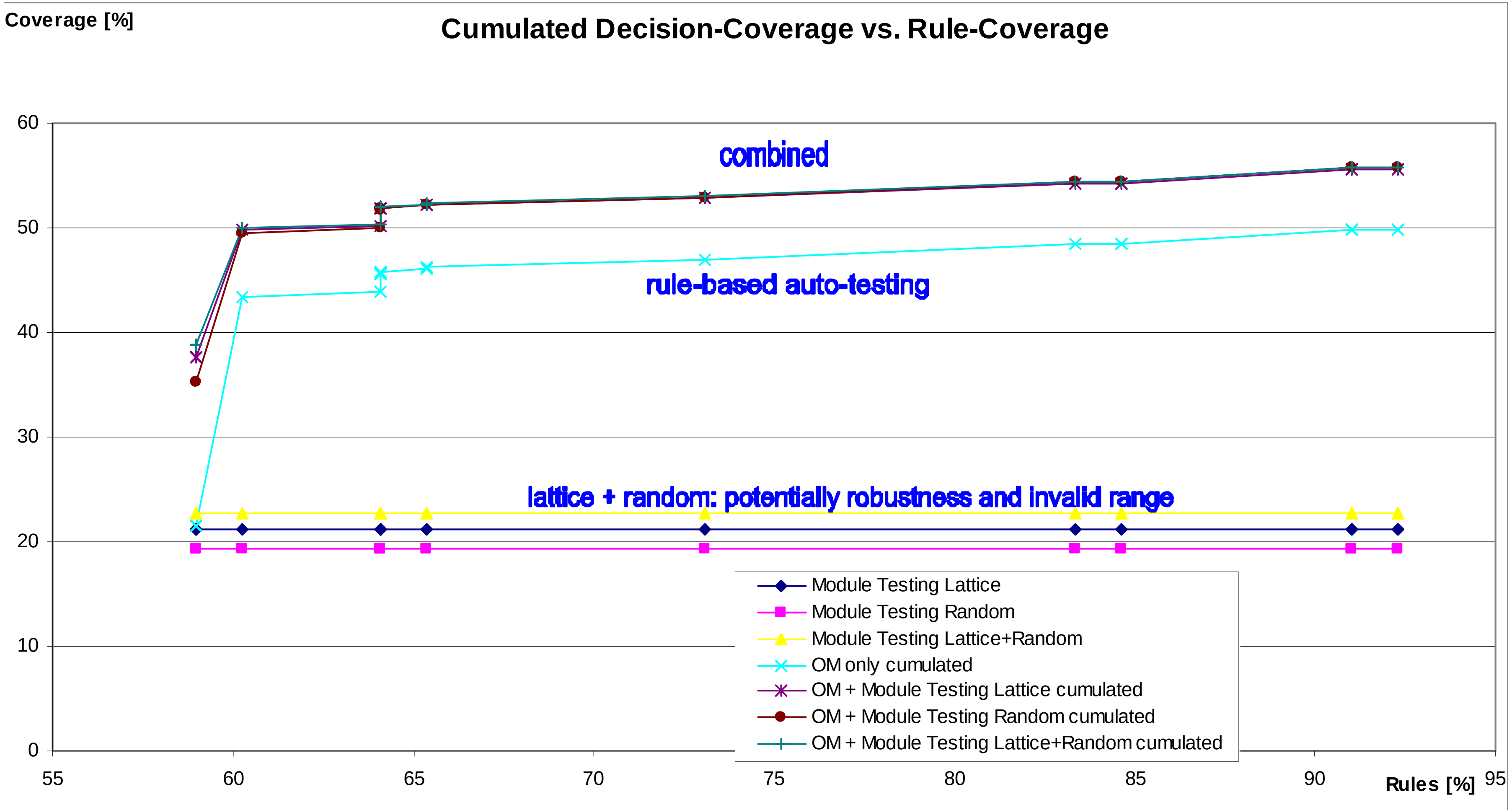
- **Block coverage**
  - lattice, random + OM test cases: complementary, significant part
  - lattice and random: coverage figure nearly equivalent, but structurally different
- **Decision coverage**
  - lattice, random + OM test cases: complementary, small part
  - lattice and random: nearly equivalent
- **flex**
  - poor context information
  - lattice and random: robustness testing, fault injection
  - the higher the lattice, random or operational coverage, the more overlap in coverage

Rule-based + Lattice + Random Testing: Block Coverage



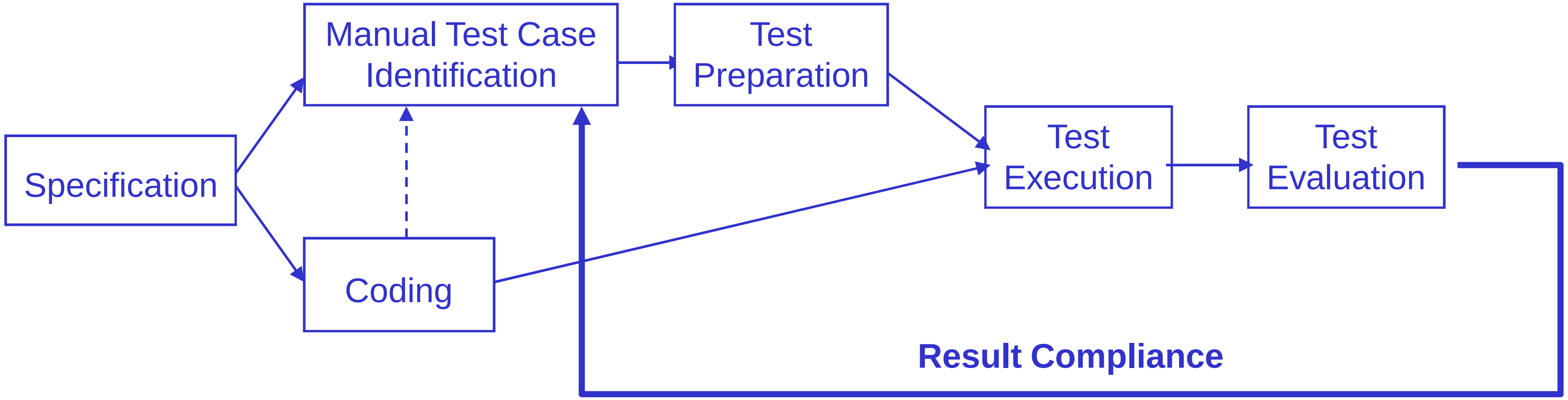
- **flex rules**
  - 76 rules to simplify expressions
  - 29 rule files generated, for 7 flex did not terminate
  - up to 2000 rules per file, up to 3000 bytes per rule (line)

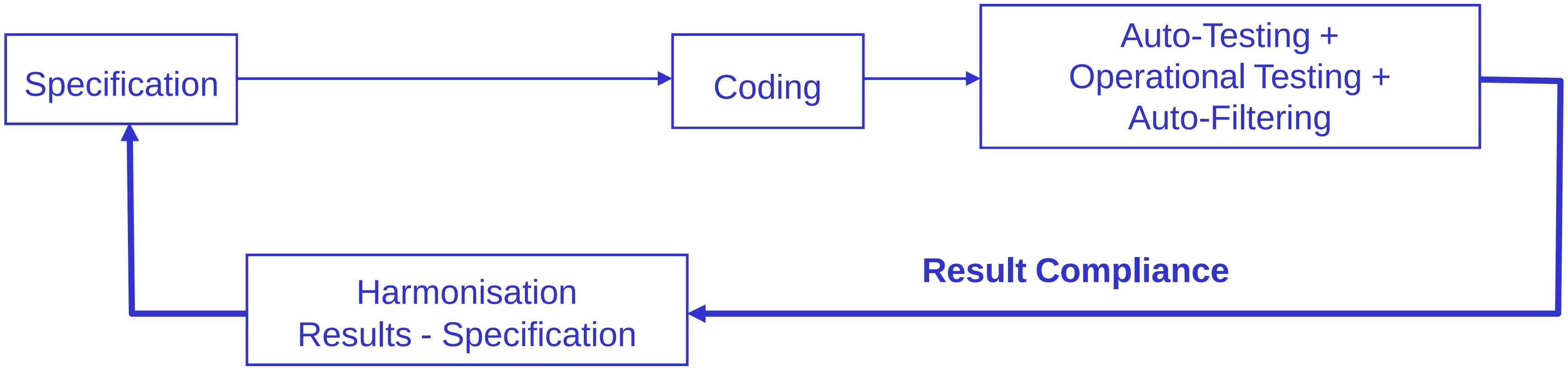
Rule-based + Lattice + Random Testing: Decision Coverage



- **flex** (not adherent to defensive programming style)
  - the lower the coverage, the more disjoined are lattice, random, operational
  - ideal case: all figures would be identical
  - 6 rules not yet covered

Modified Test Evaluation for (Full) Auto-Testing





General Conclusions (1/3)

- **Coverage**
  - good programming style ⇒ high coverage
  - poor information about valid conditions ⇒ low coverage
  - the more defensive the programming style, the higher the coverage

  - auto-testing cannot compensate poor context information

  ☞ auto-testing strongly supports well-formed code

  ☞ low coverage indicates weakness in code and potential problems

  ☞ the more information on type ranges, the higher the coverage

  ⇒ Ada better than C

General Conclusions (2/3)

❖ Efficiency

- ❖ the better the programming style, the more efficient is auto-testing
- ❖ the better the programming style, the higher the cost savings by auto-testing
- ☞ the lower the coverage, the higher is the manual effort for testing, verification, validation
- ☞ the lower the coverage, the less context information is provided
  $\Rightarrow$ recurring effort during maintenance

❖ Result production flex

- ❖ ~ 7 hours for all test modes + combinations + cumulation
- ☞ immediate feedback on code status
- ❖ one script only needs to be started
- ❖ most time needed for result presentation in Excel
- ❖ script can be easily adapted to other programs

General Conclusions (3/3)

- **Platform Diversification**
  - potential to identify more filtered test cases
  - potential to identify more exceptions
  - potential to identify more weakness

- **Test Strategies**
  - complementary in test generation
  - significant non-overlapping part for “flex-type” code
  - “rule-based” test generation complements “type-range” approach
  - deeper analysis needed on non-covered parts
  - indication for dead code (hypothesis to be checked):

    (too) low code coverage at high coverage of input domain

    in case all test modes are combined