Advanced Software Engineering

10. Test-Driven Software Experimentation

Fall Semester 2024

UNIVERSITY OF MANNHEIM
School of Business Informatics and Mathematics

Marcus Kessel marcus.kessel@uni-mannheim.de, Colin Atkinson colin.atkinson@uni-mannheim.de

Agenda

- Program Analysis
  - Static vs Dynamic
- Test-Driven Software Experimentation – Software testing is an empirical, continuous activity
- Stimulus Response Matrices (SRMs)
- Differential Testing
  - Test oracles
  - Regression testing
  - (Program-based) Mutation testing
  - N-version testing
- Stimulus Response Hypercubes (SRHs)
- Measuring Code Coverage

Static vs Dynamic Program Analysis

Static Analysis

- Analyzes code <u>without</u> execution
  - reasons over abstract code representations (e.g., AST, PDGs)
- No run-time activity required
- Analyzes all possible paths (if reachable)
  - → typically more complete and overapproximates
- Used for code understanding, static code analyzers (e.g., linting, code metrics …), symbolic execution, model checking etc.

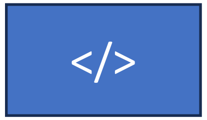
Dynamic Analysis (Software Testing)

- Analyzes code behaviour <u>at run-time</u>
  - executes the code on a set of inputs, observing behaviour, performance etc.
- Requires run-time activity
- Only observes actual paths taken
  - → typically more precise and underestimates
- Testing activities (unit, integration, system etc.)

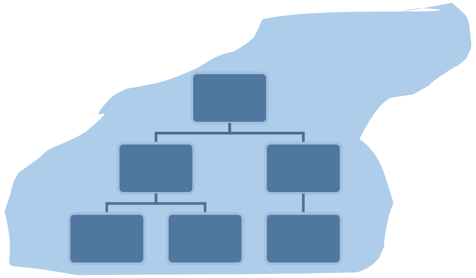
BUT: Rice's theorem states that *for any non-trivial property P of the set of partial functions (or programs), it is undecidable to determine whether a given program satisfies this property* (cf. halting problem)

→ Approaches complement each other, a combination of both is often used in practice

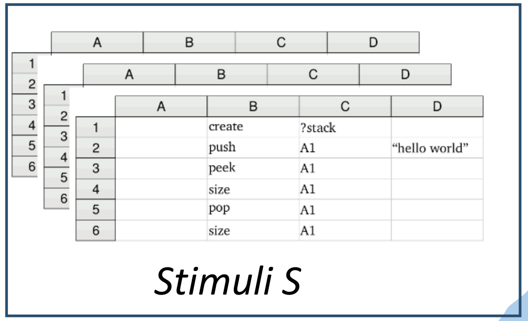
Static vs Dynamic Program Analysis – Example



Program
(code unit)

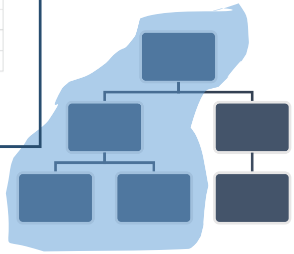


Static Call Graph
(all reachable paths)
→ no information about run-time data

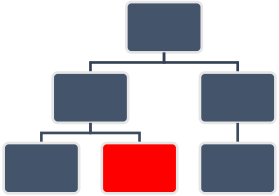


execute

missed (not exercised)



Dynamic Call Graph
(paths exercised by a set of tests)
→ monitors run-time behavior



Note: sometimes some subpaths
are not reachable (e.g., dead code)

Software Testing – An Empirical Activity

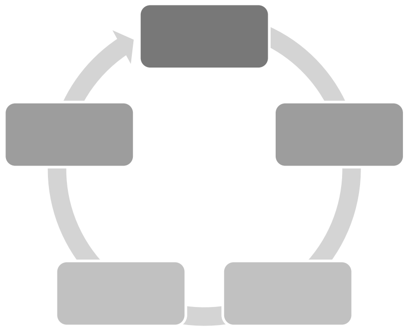
- Uncertainty and unpredictability – software is inherently complex and dynamic (i.e., hard to predict all possible outcomes and scenarios)
  - uncertainty requires empirical investigation through experimentation (i.e., software testing)
- Exploratory nature – testing is a process of exploration
  - we attempt to uncover previously unknown issues or defects (cf. Dijkstra)

> *"Program testing can be used to show the presence of bugs, but never to show their absence!"*

- Data-driven decision-making – testing generates vast amounts of data (e.g., test results, traces, metrics etc.) to make informed decisions about the system's behaviour and quality
  - testers heavily rely on this data to inform their strategies and decisions
- Experimentation and iteration – testing involves repeated experimentation with different (or equal) inputs, scenarios or configurations to gather evidence about the software's behaviour
  - iterative process allows testers to refine their understanding of the system and identify potential issues
- Human involvement – human-centric activity that requires expertise, experience and creativity
  - next to coverage criteria, testers use their empirical knowledge and understanding of software development, programming languages, and testing techniques to design and execute tests

Software Testing – Continuous Activity

- Software testing is an empirical, continuous activity …
  - Based on observation and experimentation
  - Focused on gathering evidence and data
  - Able to adapt to changing requirements or new information
  - Influenced by human judgment and experience



- This perspective emphasizes the importance of continuous learning, iteration, and improvement in software testing
  - aided by automated testing (cf. agile coding practices)
    - unit testing frameworks, continuous integration

➡ Test-Driven Software Experimentation

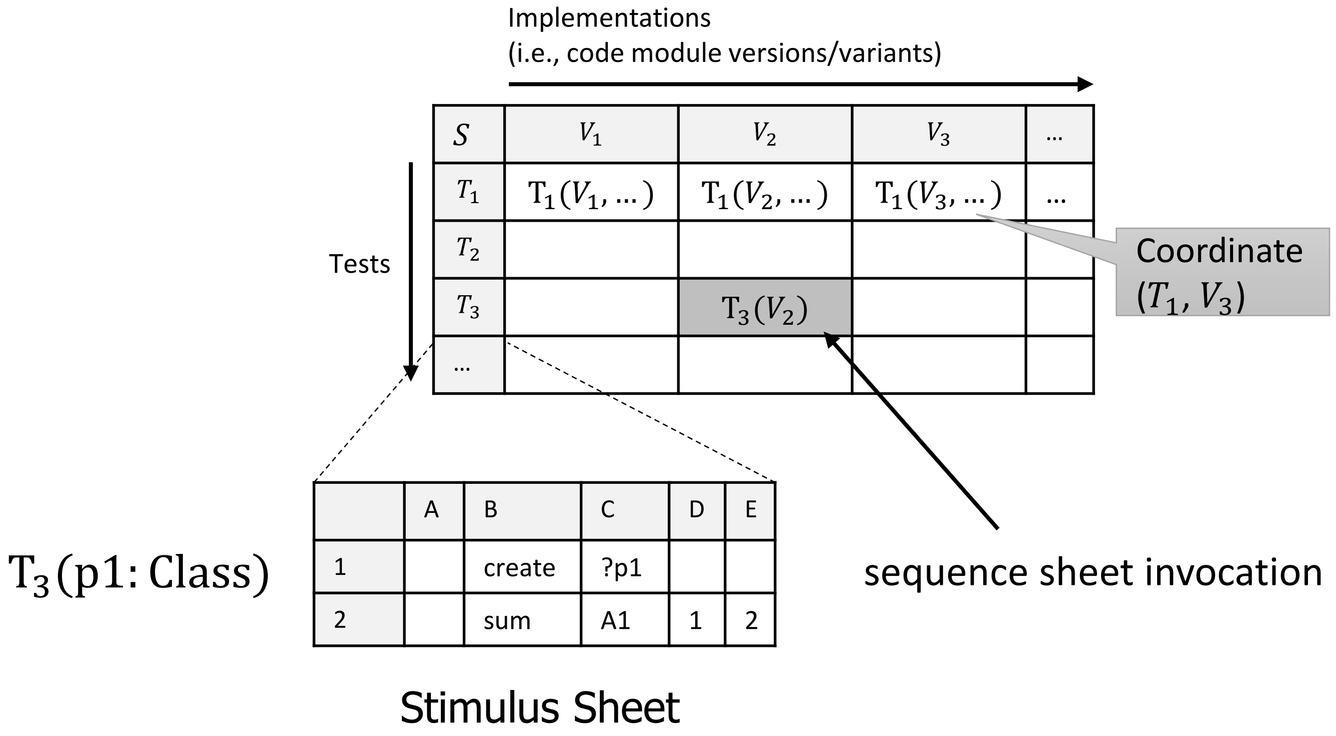
Test-Driven Software Experimentation

- A Test-Driven Software Experiment (TDSE) involves the execution of software subjects and the observation and analysis of their “de facto” run-time behaviour (i.e., run-time semantics)
  - *“Test-Driven Software Experiments (TDSEs) are experiments that involve <u>controlled</u> testing of software subjects (i.e., code modules) under various conditions, revealing important properties of the code’s run-time behavior that cannot be predicted solely through static analysis”*
  - Controlled testing: Repeatable observations for (deterministic) software in a controlled environment
    - different environments may cause different observations
- Practitioners
  - Evaluate code and tools, making informed decisions for adoption and integration (e.g., code recommendation)
- Researchers
  - Empirically validate tools and techniques involving the execution of software subjects (e.g., benchmarking tools like test generators)
- Educators
  - Teaching activities (e.g., test-driven exercises)

→ Supporting data structures needed

Stimulus Matrices (SMs) – Input

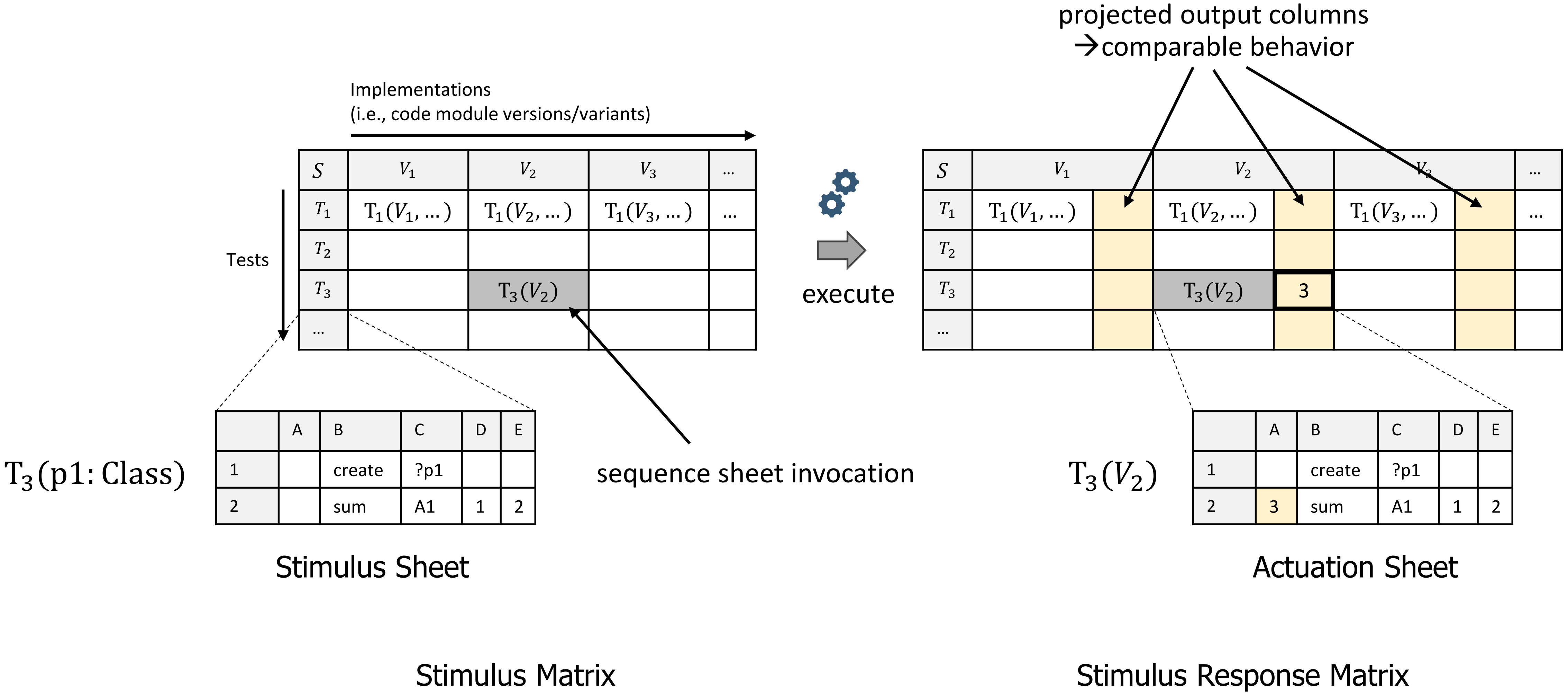
- Typically for one functional abstraction S
  - i.e., functionality under consideration
    - given by its interface
    - desired behaviour in terms of set of input/output mappings (e.g., sequence sheets)
- Configuration of test/code implementations pairs for execution
  - Columns: One or more Implementations V_i
  - Rows: One or more Tests T_j (sequence sheets)
    - Typically sequence sheets (Normal, parameterized)



Stimulus Response Matrices (SRMs) – Output

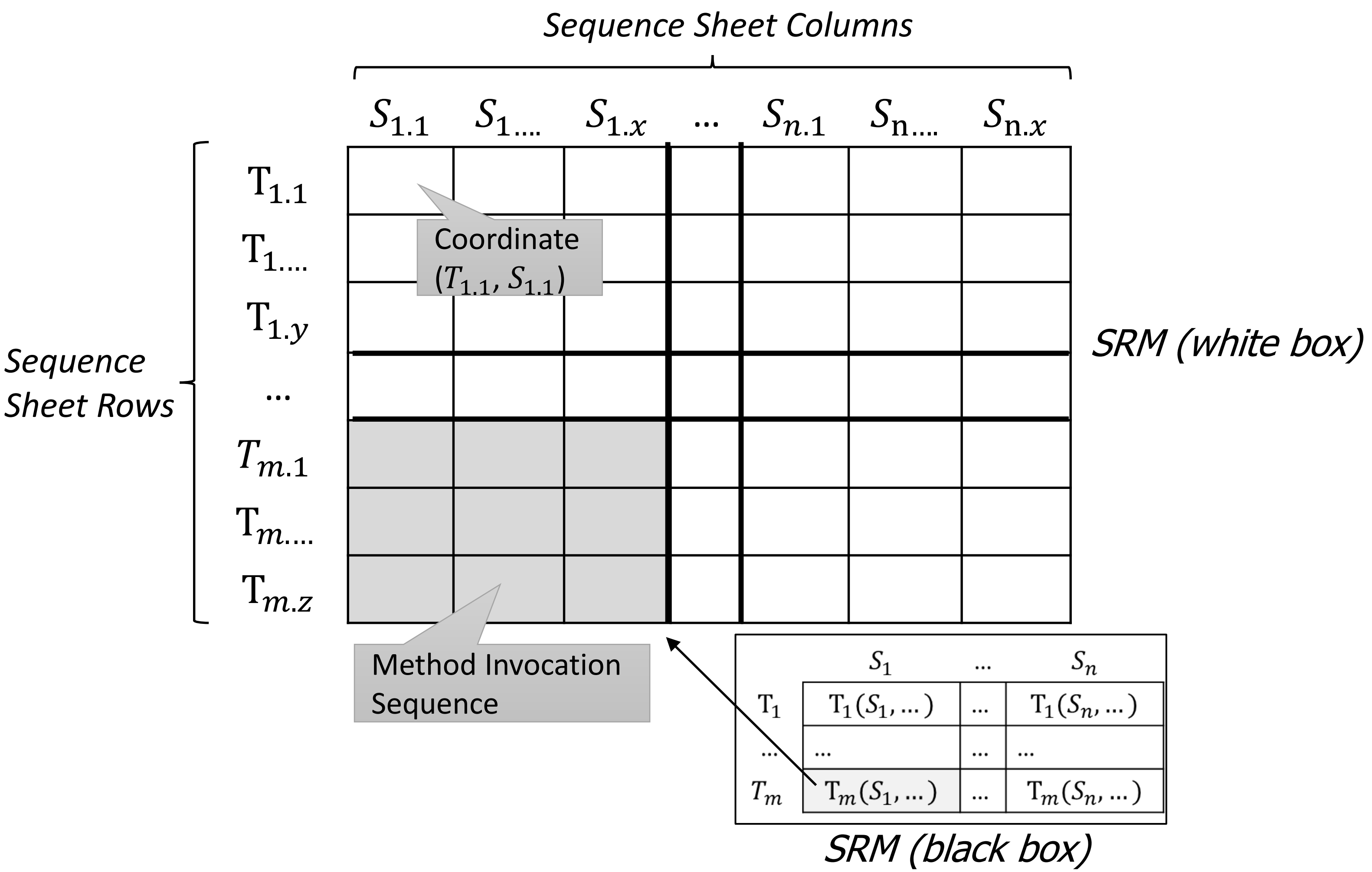
- Stimulus Response Matrices store run-time responses

Note: Assume that we ignore the first row



Stimulus Response Matrices (SRMs) – Records

- In general, SRMs store all cells of sequence sheets using two views
  - *black box*: sequence sheet body hidden
  - *white box*: sequence sheet body visible
    - can navigate SRM and select sequence sheet rows/columns/cells of interest
- In addition to functional records (output column), SRMs can also store non-functional information



SRM Example – Greatest Common Divisor

Problem: GCD (Greatest Common Divisor)
Interface: *gcd(int, int)->int*

```
test(p1:Class, p2:int, p3:int)
```

	A	B	C	D	E
1		create	?p1		
2		gcd	A1	?p2	?p3

$\underbrace{\text{A}}_{\textit{output}}$ $\underbrace{\text{B}}_{\textit{operation}}$ $\underbrace{\text{C D E}}_{\textit{inputs}}$

N versions (code module implementations) →

Tests ↓

	V1	V2	V3	…	V_n
test1	*test(V1,03,07)*	*test(V2, 03,07)*	*test(V3, 03,07)*		*test(V_n, 03,07)*
test2	*test(V1,10,15)*	*test(V2,10,15)*	*test(V3,10,15)*		*test(V_n,10,15)*
test3	*test(V1,49,14)*	*test(V2,49,14)*	*test(V3,49,14)*		*test(V_n,49,14)*
…					
$test_m$	*$test_m$(V1, x, y)*	*$test_m$(V2, x, y)*	*$test_m$(V3, x, y)*		*$test_m$(V_n, x, y)*

execute

	A	B	C	D	E
1		create	**V1**		
2	1	gcd	A1	3	7

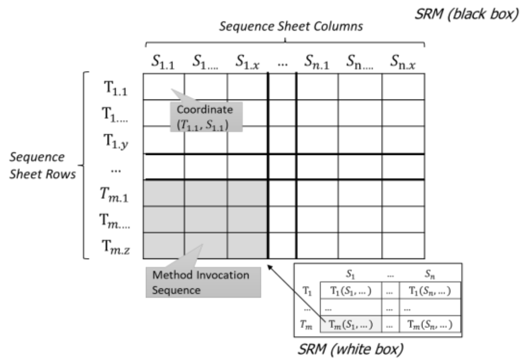
	V1		V2		V3		…	V_n	
test1	*test(V1,03,07)*	*1*	*test(V2,03,07)*	*7*	*test(V3,03,07)*	*-7*		*test(V_n,03,07)*	…
test2	*test(V1,10,15)*	*5*	*test(V2,10,15)*	*15*	*test(V3,10,15)*	*-15*		*test(V_n,10,15)*	…
test3	*test(V1,49,14)*	*7*	*test(V2,49,14)*	*14*	*test(V3,49,14)*	*28*		*test(V_n,49,14)*	…
…									
$test_m$	*$test_m$(V1, x, y)*	…	*$test_m$(V2, x, y)*	…	*$test_m$(V3, x, y)*	…		*$test_m$(V_n, x, y)*	…

comparison of observed outputs

Note: Assume that we ignore the first row

Queue Example – Complex Sequences

- Assume the functional abstraction of a queue data abstraction
  - SRMs store a de-collapsed view on rows of sheets (e.g., output column)



Output | Operation | Inputs (incl. service)

	A	B	C	D
1	<instance>	create	`MyQueue	
2	TRUE	enqueue	A1	1
3	TRUE	enqueue	A1	2
4	1	peek	A1	
5	2	size	A1	
6	1	dequeue	A1	
7	1	size	A1	

Actuation Sheet

test1(MyQueue)

De-collapsed sheet rows

	MyQueue		…	V_n	
test1	*test1(MyQueue)*	<instance>			…
…2		TRUE			…
…3		TRUE			…
…4		1			…
…5		2			…
…6		1			…
…7		1			…
test2	*test2(MyQueue)*				…
…	…				

Stimulus Response Matrix

Differential Testing & SRMs

- In general, Differential testing is a software testing technique that involves comparing the behaviour of two or more versions of a program, typically to detect differences in their functionality
- In practice, many testing activities can be related to differential testing
  - Test oracles – compare expected behaviour to actual behaviour of unit under test
    - Specified oracles (input/output mappings, pre-/post conditions, specification lang. …)
    - Cross-checking oracles – using different implementations of the same functionality
  - Regression testing – compare old version of code unit with new version (i.e., verify that code modifications do not change run-time semantics)
  - Mutation testing – compare original version of code unit to mutated versions
  - N-version testing – compare behaviour of two or more versions of a program and settle (agree) on common behaviour (e.g., critical, dependable software)

→ SRMs support and facilitate differential testing

Test Oracles – Specified Oracle

- A test oracle uses expected outputs ("ground truth") to decide whether a test passed or failed
  - represented as a virtual implementation version in SRMs
  - taken from output column of sheets

	A	B	C	D	E
1		create	**V1**		
2	1	gcd	A1	3	7

	A	B	C	D	E
1		create	**V1**		
2	5	gcd	A1	10	15

	A	B	C	D	E
1		create	**V1**		
2	7	gcd	A1	49	14

Expected outputs

Stimulus Sheets

	V1
test1	*test(V1,03,07)*
test2	*test(V1,10,15)*
test3	*test(V1,49,14)*

Stimulus Matrix

Virtual Impl.
(not executed)

execute

	Oracle		V1	
test1	*test(V1,03,07)*	*1*	*test(V1,03,07)*	*1*
test2	*test(V1,10,15)*	*5*	*test(V1,10,15)*	*5*
test3	*test(V1,49,14)*	*7*	*test(V1,49,14)*	*11*

Stimulus Response Matrix

Compare expected outputs with actual, observed outputs

Actual (observed) outputs

	A	B	C	D	E
1		create	**V1**		
2	1	gcd	A1	3	7

	A	B	C	D	E
1		create	**V1**		
2	5	gcd	A1	10	15

	A	B	C	D	E
1		create	**V1**		
2	11	gcd	A1	49	14

Actuation Sheets

Test Oracles – Cross-Checking Oracle

- Reference system *Ref* serves as cross-checking oracle in SRM
  - available reference implementation (e.g., different sort algorithms should return the same outputs for inputs)

	A	B	C	D	E
1		create	**V1**		
2		gcd	A1	3	7

	A	B	C	D	E
1		create	**V1**		
2		gcd	A1	10	15

	A	B	C	D	E
1		create	**V1**		
2		gcd	A1	49	14

Stimulus Sheets

	Ref	V1
test1	*test(V1,03,07)*	*test(V1,03,07)*
test2	*test(V1,10,15)*	*test(V1,10,15)*
test3	*test(V1,49,14)*	*test(V1,49,14)*

Stimulus Matrix

Reference System

execute

	Ref		V1	
test1	*test(V1,03,07)*	*1*	*test(V1,03,07)*	*1*
test2	*test(V1,10,15)*	*5*	*test(V1,10,15)*	*5*
test3	*test(V1,49,14)*	*7*	*test(V1,49,14)*	*11*

Stimulus Response Matrix

Compare observed outputs

Actual (observed) outputs

	A	B	C	D	E
1		create	**V1**		
2	1	gcd	A1	3	7

	A	B	C	D	E
1		create	**V1**		
2	5	gcd	A1	10	15

	A	B	C	D	E
1		create	**V1**		
2	11	gcd	A1	49	14

Actuation Sheets

Regression Testing

- Regression testing verifies that the software still behaves as expected (i.e., functions correctly) after updates
  - added features
  - fixed faults
  - refactoring, etc.
- Typical realizations
  - re-run existing tests on new version
  - old version may become your (cross-checking) oracle

	V1 (**old** version)	V2 (**new** version)
test1	*test(V1,03,07)*	*test(V1,03,07)*
test2	*test(V1,10,15)*	*test(V1,10,15)*
test3	*test(V1,49,14)*	*test(V1,49,14)*

Stimulus Matrix

execute

alternatives

	Oracle		V1 (**old** version)		V2 (**new** version)	
test1	*test(V1,03,07)*	*1*	*test(V1,03,07)*	*1*	*test(V1,03,07)*	*1*
test2	*test(V1,10,15)*	*5*	*test(V1,10,15)*	*5*	*test(V1,10,15)*	*5*
test3	*test(V1,49,14)*	*7*	*test(V1,49,14)*	*7*	*test(V1,49,14)*	*11*

Stimulus Response Matrix

Compare

e.g., introduced a fault

Mutation Testing – Program-based Grammars

- Recall syntax-based testing of grammars
- The original and most widely known application of syntax-based testing is to modify programs
- Operators modify a ground string (program under test) to create mutant programs
  - Mimic typical programmer mistakes (e.g., incorrect variable name)
  - Encourage common test heuristics (e.g., cause expressions to be 0)
- Mutant programs must compile correctly (valid strings)
- Mutants are not tests, but used **to find tests** → assessing test set quality
  - Fundamental Premise: In practice, if the software contains a fault, there will usually be a set of mutants that can be killed only by a test case that also detects that fault
- Once mutants are defined, tests must be found to cause mutants to fail when executed
- This is called “killing mutants”

Mutation Testing – Operators Example

1. ABS — Absolute Value Insertion
2. AOR — Arithmetic Operator Replacement
3. ROR — Relational Operator Replacement
4. COR — Conditional Operator Replacement
5. SOR — Shift Operator Replacement
6. LOR — Logical Operator Replacement
7. ASR — Assignment Operator Replacement
8. UOI — Unary Operator Insertion
9. UOD — Unary Operator Deletion
10. SVR — Scalar Variable Replacement
11. BSR — Bomb Statement Replacement

2. AOR — Arithmetic Operator Replacement:
Each occurrence of one of the arithmetic operators +, −, *, /, and % is replaced by each of the other operators.

Examples:

```
     a = m * (o + p);
Δ1   a = m + (o + p);
Δ2   a = m * (o * p);
Δ3   a = m / (o + p);
```

Remember: Mutated strings (here code) should be close to the original …

→ Researchers select appropriate operators through empirical assessments

Mutation Testing – Killing Mutants

Given a mutant $m \in M$ for a ground string program P and a test t, t is said to kill m if and only if the output of t on P is different from the output of t on m.

- If mutation operators are designed well, the resulting tests will be very powerful
- Different operators must be defined for different programming languages and different goals
- Testers can keep adding tests until all mutants have been killed
  - Dead mutant: A test has killed it
  - Stillborn mutant: Syntactically illegal
  - Trivial mutant: Almost every test can kill it
  - Equivalent mutant: No test can kill it (same behavior as original)

Mutation Testing – Test Requirements

<u>Mutation Coverage (MC)</u> : For each $m \in M$, TR contains exactly one requirement, to kill m.

- Recall the conditions from lecture *L2* for a software failure to be observed -
  - Reachability : The test causes the faulty statement to be reached (in mutation – the mutated statement)
  - Infection : The test causes the faulty statement to result in an incorrect state
  - Propagation : The incorrect state propagates to incorrect output
  - + Revealability (observability) : The tester must observe part of the incorrect output

  → Together, these are also know as the RIPR model
- This leads to two variants of mutation coverage …

Mutation Testing – Strong vs Weak Mutation Coverage

- Strongly killing mutants – satisfies all conditions (assumed in the remainder …)

> Given a mutant $m \in M$ for a program P and a test t, t is said to *strongly kill* m if and only if the output of t on P is different from the output of t on m

- Weakly killing mutants

> Given a mutant $m \in M$ that modifies a location l in a program P, and a test t, t is said to *weakly kill* m if and only if the state of the execution of P on t is different from the state of the execution of m on t immediately after l

  - Weakly killing satisfies reachability and infection, but not propagation

Mutation Testing – SRMs

- Basic SRM setting for strong mutation testing

Generated mutant implementations

	V1 (**Original**)	M1	M2	…	M_n
test1					…
test2					…
test3					…
…					
$test_m$					…

Identify discrepancies in behaviour (differences in output)

Mutation Testing – Min Example

Original Method

```
int min (int A, int B)
{
      int minVal;
      minVal = A;
      if (B < A)
      {
          minVal = B;
      }
      return (minVal);
} // end Min
```

6 mutants

Each represents a separate program

With Embedded Mutants

```
int min (int A, int B)
{
      int minVal;
      minVal = A;
∆ 1   minVal = B;
      if (B < A)
∆ 2   if (B > A)
∆ 3   if (B < minVal)
      {
          minVal = B;
∆ 4       Bomb ();
∆ 5       minVal = A;
∆ 6       minVal = failOnZero (B);
      }
      return (minVal);
} // end Min
```

Replace one variable with another

Replaces operator

Immediate runtime failure … if reached

Immediate runtime failure if B==0, else does nothing

Mutation Testing – SRM Min Example

- Setting for Min Example
- Assume there are 3 tests, and 6 mutants (see slide before)
  - a mutant is killed if there is at least one test with at least one disagreement in the expected outputs

	A	B	C	D	E
1		create	**?c**		
2	1	min	A1	2	1

	A	B	C	D	E
1		create	**?c**		
2	1	min	A1	1	2

	A	B	C	D	E
1		create	**?c**		
2	1	min	A1	1	1

Expected outputs

Stimulus Sheets

Generated mutant implementations

	V1 (Original)		M1		M2		M3		M4		M5		M6	
test1	*test1(V1)*	*1*		*1*		*2*		*1*		*E*		*2*		*1*
test2	*test2(V1)*	*1*		*2*		*2*		*1*		*1*		*1*		*1*
test3	*test3(V1)*	*1*		*1*		*1*		*1*		*1*		*1*		*1*

Killed
4 Mutants

Mutation Testing – Equivalent Mutants

- Mutant 3 (M3) in the *min()* example is behaviourally equivalent to the original implementation:

```
      minVal = A;
      if (B < A)
△ 3  if (B < minVal)
```

- The infection condition is "(B < A) != (B < minVal)"
- However, the previous statement was "minVal = A"
- Substituting, we get: "(B < A) != (B < A)"
  - This is a logical contradiction !
- Thus no input can kill this mutant

Mutation Score & SRMs – Example

- We can work out the mutation score directly from SRMs
- Mutation Score
  - Number mutants generated for killing (= 6)
    - i.e., set of test requirements
  - Number of killed mutants (= 4)
  - Score: <u>4/6 → ~66%</u>
- Important: that we can't reach 100% because of equivalent mutant M3
- Generally, only disagreements to explicit expected values are considered (i.e., marked in blue) – if missing, no comparison is made
  - similar to unit testing assumptions (i.e., missing assertion checks)

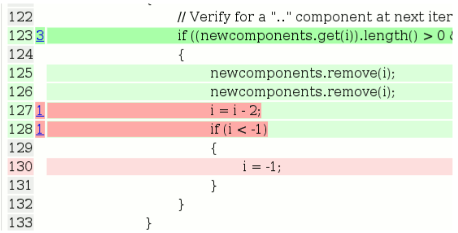
	V1 (Original)		M1		M2		M3		M4		M5		M6	
test1	*test1(V1)*	*1*		*1*		*2*		*1*		*E*		*2*		*1*
test2	*test2(V1)*	*1*		*2*		*2*		*1*		*1*		*1*		*1*
test3	*test3(V1)*	*1*		*1*		*1*		*1*		*1*		*1*		*1*

one output discrepancy in test2 (column)
→ M1 killed

two output discrepancies in test1, test2 (column)
→ M2 killed

Mutation Testing – Final Words

- Mutation is widely considered the strongest test criterion (“gold standard”)
  - And most expensive ! (run-time requirements in larger code bases …)
  - By far the most test requirements (each mutant)
  - Usually the most tests
- Mutation subsumes other criteria by including specific mutation operators
- Tool examples
  - Java: *PIT* https://pitest.org/ , Python: *mutpy*, *mutmut* …
  - Provide reports (may report other criteria as well) etc. → lookup their suggested default set of mutation operators
  - Can be integrated into automated testing workflows (i.e., continuous integration)
    - Typically compatible to existing unit testing frameworks/drivers



PIT reports
(green line coverage, red mutation coverage)

N-version Testing

- compare behaviour of two or more versions of a program and settle (agree) on common behaviour (i.e., responses in terms of outputs)
  - Traditional hypothesis: multiple versions have been developed by independent developers for the same behavior specification
  - assumes that different developers make different mistakes
- Proposed for critical, dependable software (e.g., avionics)
  - correct behavior is identified by majority vote at run-time (for instance)
  - other voting mechanisms possible
- can also be used for practical assessment of multiple implementation candidates where one final candidate is to be selected
  - i.e., fit-for-purpose evaluations in code recommenders
  - however, may require adaptation of the code candidates interfaces

SRM

	V1	V2	V3	V4	V5
test1	*1*	*7*	*-7*	*1*	*1*
test2	*5*	*15*	*-15*	*5*	*5*
test3	*7*	*14*	*28*	*7*	*7*
test4	*12*	*60*	*60*	*12*	*12*
tgen1	*1*	*1*	*1*	*2158*	*2158*
tgen2	*1*	*1*	*1*	*503*	*503*
tgen3	*-15*	*1*	*-15*	*-15*	*-15*

cluster size = 2

Oracle

| 1|= 3
| 7|= 1
|-7|= 1

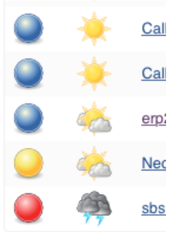
Test-based Voting	Cluster-based Voting
1	*1*
5	*5*
7	*7*
12	*12*
1	*2158*
503	*503*
-15	*-15*

Software Code Evolution – Continuous Integration

- Test-driven development methods work best when the current version of the software can be run against all tests at any time

A *continuous integration server* rebuilds the system, returns, and reverifies tests whenever *any* update is checked into the repository

- Mistakes are caught earlier
- Other developers are aware of changes early
- The rebuild and reverify must happen as soon as possible
- Thus, tests need to execute quickly



e.g., Jenkins weather report

A *continuous integration server* doesn't just run tests, it decides if a modified system is still correct

Software Code Evolution – Continuous Lifecycle

- Over time, as the production code evolves, the set of tests for the code implementation evolves as well
  - tests may be added (e.g., “bug-fixing” tests), updated or removed to maintain/improve reliability
- if changes are made to the implementation
  - code revisions (new versions) are added from time to time and are automatically verified for correctness
    - e.g., git push + automated testing

Code Revisions (by time)

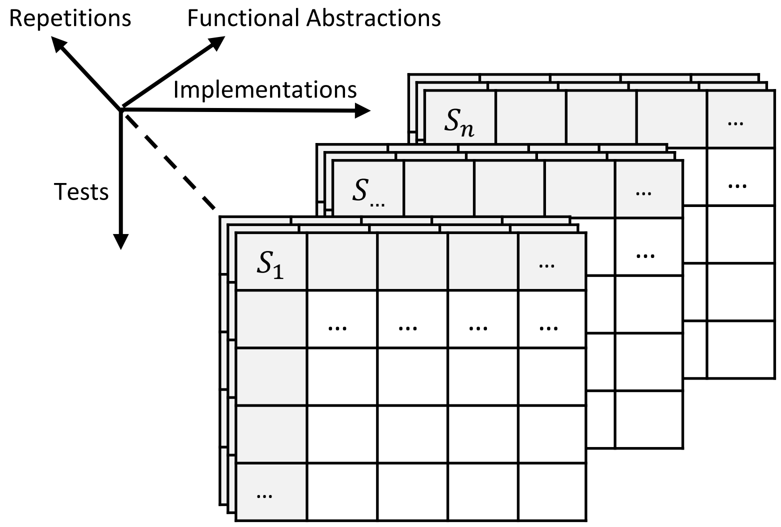
Tests

S	R_1	R_2	R_3	…
T_1	$\mathrm{T}_1(V_1, \dots)$	$\mathrm{T}_1(V_2, \dots)$	$\mathrm{T}_1(V_3, \dots)$	…
T_2				
T_3		$\mathrm{T}_3(V_2)$		
…				

Automated Testing (e.g., Continuous Integration)

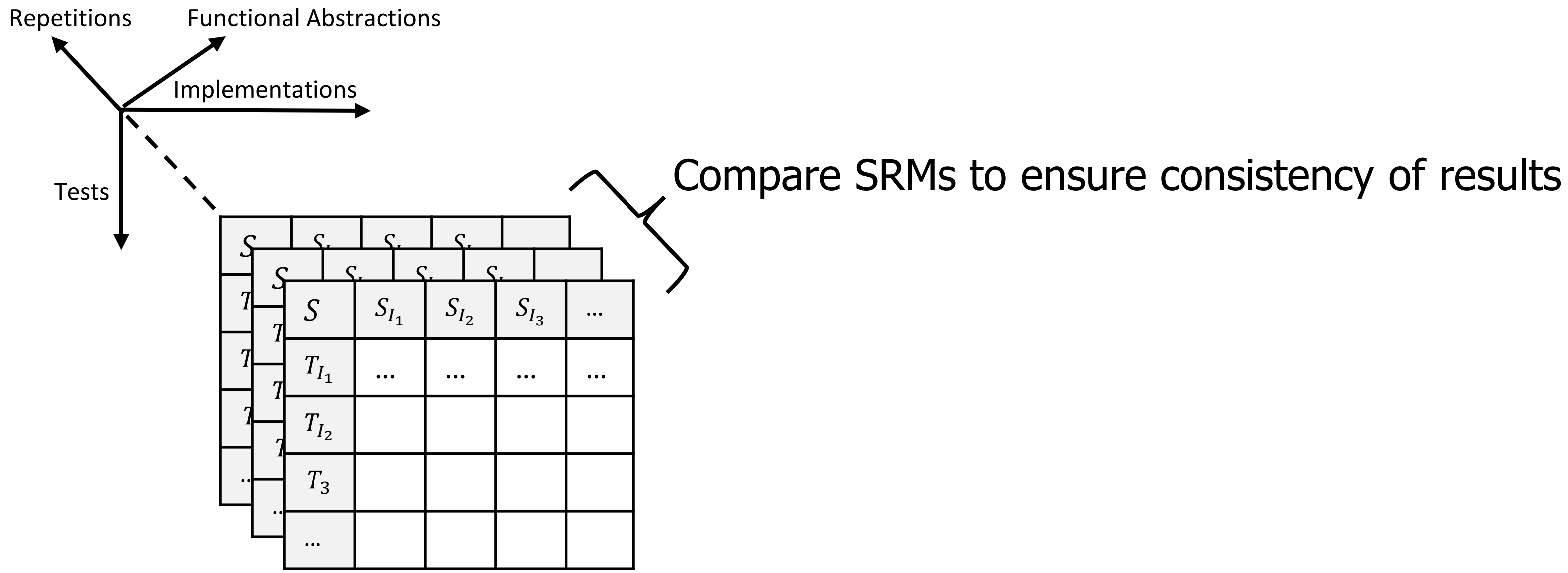
Stimulus Response Hypercubes – SRHs

- Aggregate multiple SRMs into a cube structure (cf. data warehousing applications)
- Navigate across multiple dimensions to inform decision-making and test reporting
  - Functional abstractions S_i
    - Tests
    - Implementations
  - Repetitions
  - …
- Enables software analytics
  - Reason over data (e.g., metrics etc.)



Repeatability – Checking for Test Flakiness

- Test flakiness refers to the phenomenon where automated tests occasionally fail or behave unexpectedly, even when there are no actual issues with the code being tested
  - Random failures
  - Inconsistent behaviour
  - False positives/negatives
- Causes, for instance
  - Source of randomness
  - Concurrency issues
  - External services (e.g., DBs)
- Consequences, for instance
  - Maintenance costs
  - Confidence erodes
  - May mask actual issues with the code



Advanced Topics – An Example

- Test set minimization of a test set T
  - Problem: Imagine you code base as well as the test sets evolve over time → Scalability?
  - Goal: Identify a minimal set of tests to improve efficiency, thus minimize run-time costs (time, money etc.) – also see *test set prioritization* (only running those tests affected by a change)
  - Idea: if a pair of tests satisfy the same test requirement, we can drop either of these, while still satisfy the same test requirements
    - E.g., identifying redundant tests based on the killing of the same mutants for a pair of tests → drop one redundant test
    - Similar to mutation matrices

	V1 (Original)	M1	M2	M3	M4
test1	*1*	*1*	*1*	*7*	*1*
test2	*5*	*0*	*5*	*5*	*5*
test3	*7*	*7*	*7*	*1*	*7*

assume that test1 and test2 kill the same mutant M3 → drop test1 or test3

Measuring Code Coverage in Practice

- a way to quantify how much of a program's code is exercised during automated tests at run-time
  - provides insight into the thoroughness of the test suite by measuring the percentage of code that has been exercised
  - can be used to identify non-exercised code elements
- Typical types of code coverage
  - Line coverage (or statement coverage)
  - Branch coverage
  - Method/function or class coverage
- Typical metrics reported (note: should be carefully interpreted!)
  - Percentage of code covered/missed etc.
- Tool examples
  - Java: *JaCoCo*, (Open)*Clover*, …
  - Python: *Coverage.py*, …

JaCoCo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.jacoco.core		97%		91%	143	1,537	125	3,631	19	746	2	147
org.jacoco.examples		58%		64%	24	53	97	193	19	38	6	12
org.jacoco.agent.rt		75%		83%	32	130	75	344	21	80	7	22
jacoco-maven-plugin		90%		82%	35	193	49	465	8	116	1	23
org.jacoco.cli		97%		100%	4	109	10	275	4	74	0	20
org.jacoco.report		99%		99%	4	572	2	1,345	1	371	0	64
org.jacoco.ant		98%		99%	4	162	8	428	3	110	0	19
org.jacoco.agent		86%		75%	2	10	3	27	0	6	0	1
Total	1,438 of 28,925	95%	183 of 2,386	92%	248	2,766	369	6,708	75	1,541	16	308

Conclusion

- Software testing is an empirical activity in general → Test-Driven Software Experimentation
  - Testing is done in a continuous manner (tests co-evolve with the units under test)
- Differential testing with SRMs
  - Test oracles (specified and cross-checking oracles)
  - Regression testing
  - Mutation testing / score
  - N-version testing
- SRHs for continuous testing
  - Repeatability and test flakiness
- Advanced topics – e.g., test set minimization
- Measuring Code Coverage

Literature

- Software Testing
  - Ammann, Paul, and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2017.
  - Barr, Earl T., et al. "The oracle problem in software testing: A survey." *IEEE transactions on software engineering* 41.5 (2014): 507-525.
  - Chen, Liming, and Algirdas Avizienis. "N-version programming: A fault-tolerance approach to reliability of software operation." *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*. Vol. 1. 1978.
- Data Structures
  - Publications
    - Marcus Kessel, Colin Atkinson, Promoting open science in test-driven software experiments, Journal of Systems and Software, Volume 212, 2024, 111971, ISSN 0164-1212, https://doi.org/10.1016/j.jss.2024.111971
    - Marcus Kessel , LASSO – an observatorium for the dynamic selection, analysis and comparison of software, Dissertation, 2023 - https://madoc.bib.uni-mannheim.de/64107/
  - LASSO Platform
    - https://softwareobservatorium.github.io/