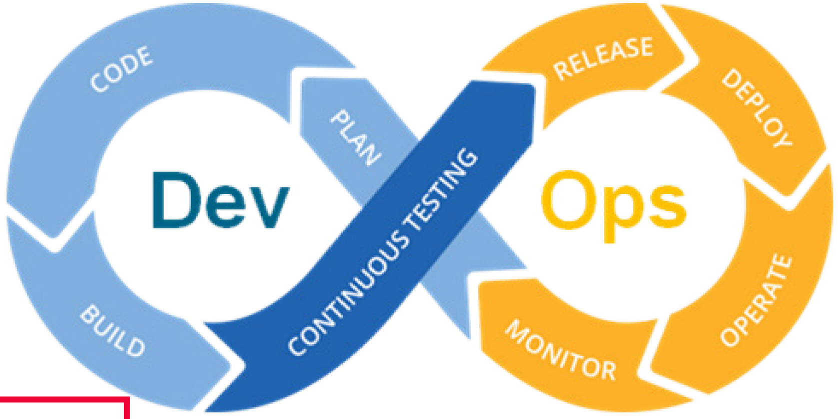
Automated Exploratory Testing with Defined Coverage

Andreas Ulrich
Siemens Technology, Munich, Germany
andreas.ulrich@siemens.com

Challenges in continuous testing



- Lack of testability support in products
  - Required to enable automated testing
- Lack of standard tools
  - In-house test automation still common practice
- Lack of faster feedback loops **FOCUS**
  - Need to gather feedback from tests in real-time; short test execution time
- Lack of testing infrastructure
  - Test environments need to run 24/7 and under repeatable conditions
- Test data management
  - Centrally managed test data to get consistent test results
- Scalability issues
  - Due to complex systems and the need for large concurrent test sessions

Source: Richa Agarwal, Common Challenges in Continuous Testing, August 24, 2020

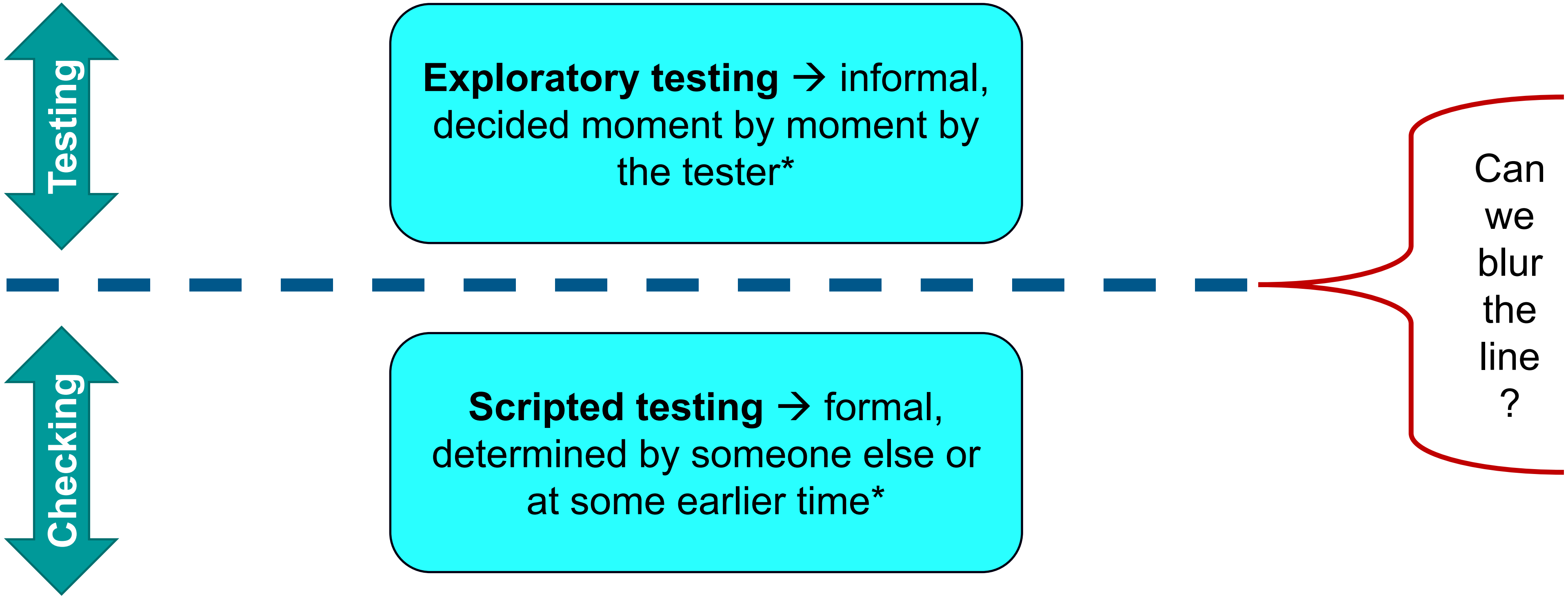
Exploratory testing

James Bach: “Testing is an exploratory process. It’s not just sometimes exploratory; it is inherently exploratory.”

Exploratory Testing is defined as simultaneous

- Learning,
- Test design and
- Test execution.

Testing vs Checking



Testing

Exploratory testing → informal, decided moment by moment by the tester*

Checking

Scripted testing → formal, determined by someone else or at some earlier time*

Can we blur the line ?

* James Bach, https://www.satisfice.com/exploratory-testing

Adaptive testing to automate exploratory tests

- Testing of a self-contained software component
- Only the component API is accessed → black-box test
- Scenario tests
  - Structured sequence of predefined test commands
- Tester reacts to SUT responses at runtime
  - Selection of next test command according to an overall test goal + randomization

```
Given: Set of test commands

var tcmd = Reset();
while(true)
{
   tcmd.Invoke();
   tcmd = SelectNext();
}
```

Test commands

A test command = SUT interaction + local state update; it follows the “4-As” pattern (extension of “3-As” as known in unit testing)

- **Arrange**: prepare input parameters for SUT call
- **Act**: perform SUT call
- **Assert**: validate correctness of returned data
- **Adapt**: update local state in tester

A test command is conditioned

- A condition describes the state that enables the test command
- Finding the right condition and state representation is a creative, exploratory act
- Test commands form a guarded command language

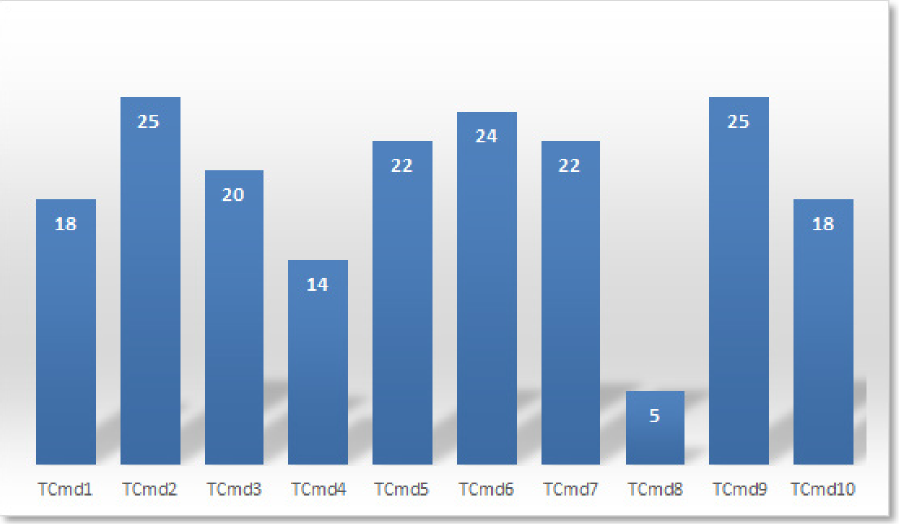
Example: Stack

```
[Condition(true)] TPush()
{ sut.Push(x);
  assert(sut.Length, i+1); i++; }

[Condition(i > 0)] TPeek()
{ sut.Peek();
  assert(sut.Length, i) }

[Condition(i > 0)] TPop()
{ x = sut.Pop();
  assert(sut.Length, i-1); i--; }
```

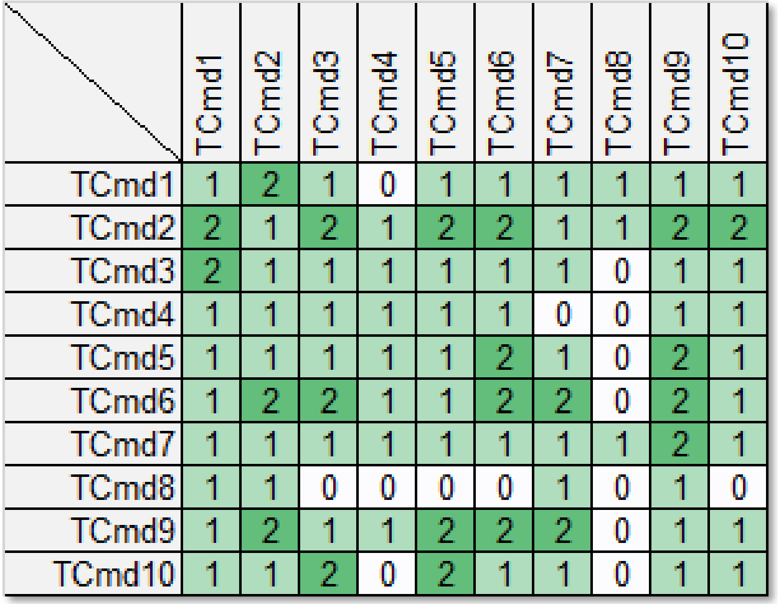
Adaptive testing applied in practice



Test goal: coverage over test commands

When is the set of test commands covered?

- Syntactical coverage over test command definitions
  → Counting test commands, pairs of test commands etc.
- Semantical coverage over states reached in test exec.
  → Counting states



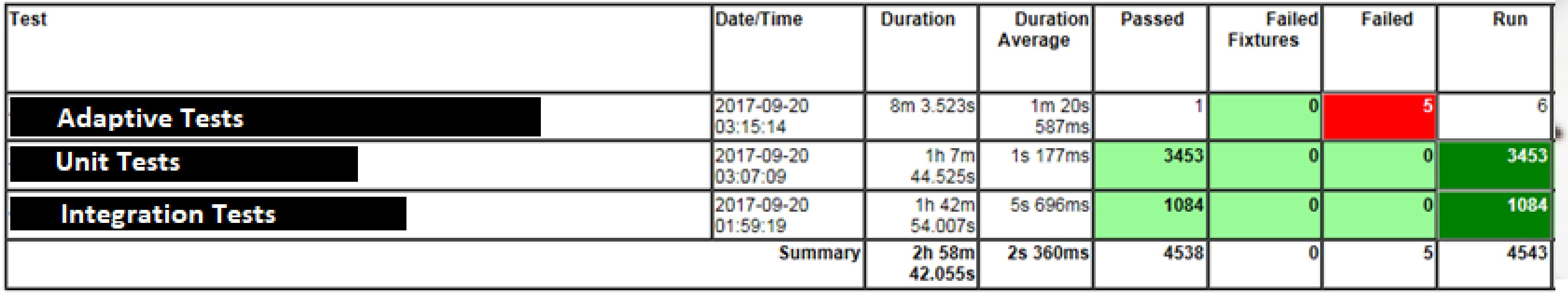
	TCmd1	TCmd2	TCmd3	TCmd4	TCmd5	TCmd6	TCmd7	TCmd8	TCmd9	TCmd10
TCmd1	1	2	1	0	1	1	1	1	1	1
TCmd2	2	1	2	1	2	2	1	1	2	2
TCmd3	2	1	1	1	1	1	1	0	1	1
TCmd4	1	1	1	1	1	1	0	0	1	1
TCmd5	1	1	1	1	1	2	1	0	2	1
TCmd6	1	2	2	1	1	2	2	0	2	1
TCmd7	1	1	1	1	1	1	1	1	2	1
TCmd8	1	1	0	0	0	0	1	0	1	0
TCmd9	1	2	1	1	2	2	2	0	1	1
TCmd10	1	1	2	0	2	1	1	0	1	1

In practise, different goals are chosen to configure about 5 test runs lasting 10 min or less

- Each test run is different due to randomization
- Able to detect deep state failures
- Remain effective over the entire DevOps cycle

What difference does it make – Example 1: Deep state failures

- 5 adaptive tests detected a failure that 3453 unit tests and 1084 hard-coded integration tests were unable to find!
- Adaptive tests remain effective in finding failures over the entire development cycle



Test	Date/Time	Duration	Duration Average	Passed	Failed Fixtures	Failed	Run
Adaptive Tests	2017-09-20 03:15:14	8m 3.523s	1m 20s 587ms	1	0	5	6
Unit Tests	2017-09-20 03:07:09	1h 7m 44.525s	1s 177ms	3453	0	0	3453
Integration Tests	2017-09-20 01:59:19	1h 42m 54.007s	5s 696ms	1084	0	0	1084
	Summary	2h 58m 42.055s	2s 360ms	4538	0	5	4543

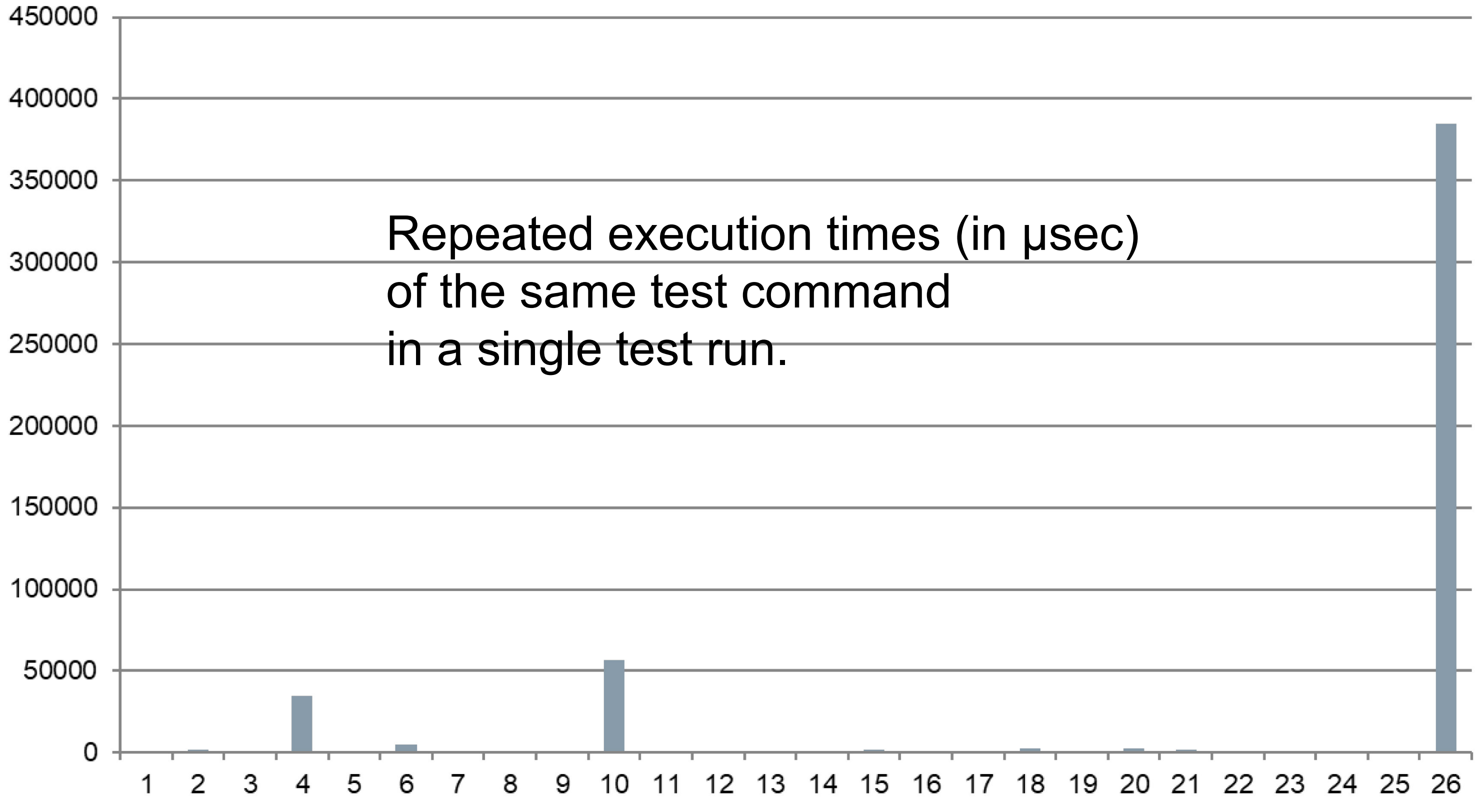
What difference does it make – Example 2: Performance degradation

Tracking of the execution time of test commands

- Within the same test run
  → Reliability tests
- Over multiple test runs over time
  → Regression tests

Expectation on execution times

- Constant
- Proportional
- Learned



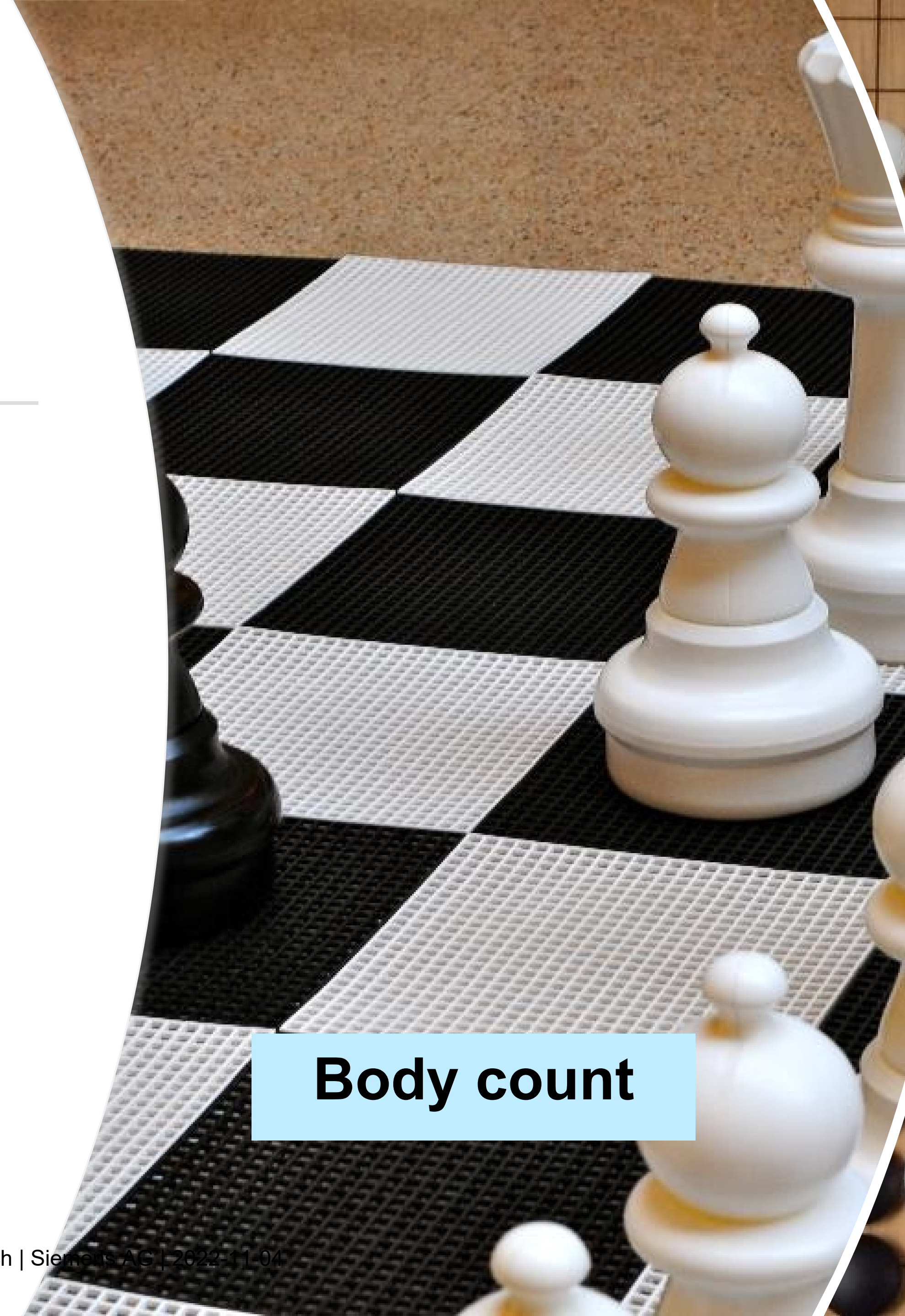
Repeated execution times (in μsec)
of the same test command
in a single test run.

Common strategies in testing

Detect failures until resources spent

Vs

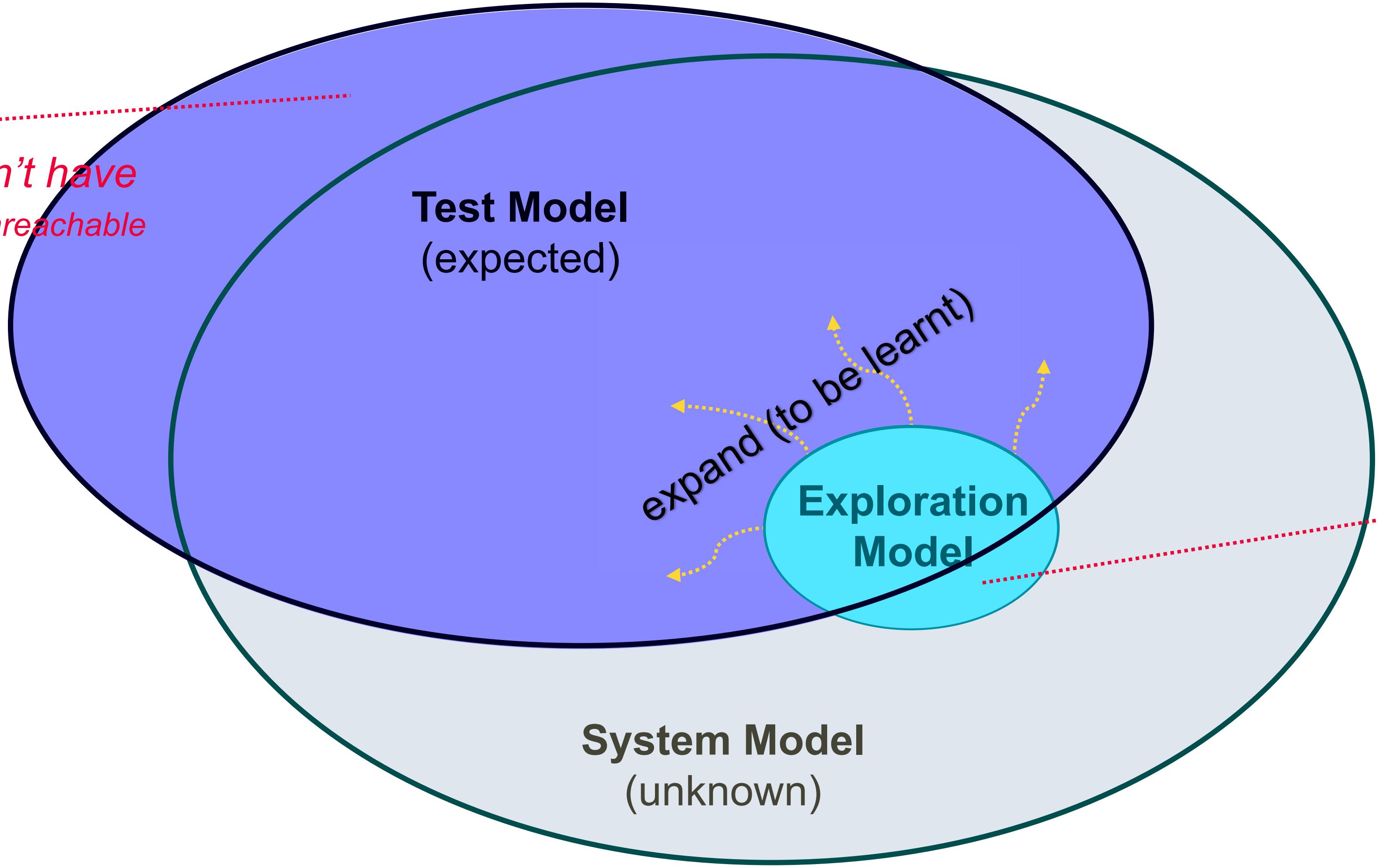
Gain confidence through covered behaviour



Body count

Gain territory

Test Model, System Model, and Exploration Model



Property that the system doesn't have

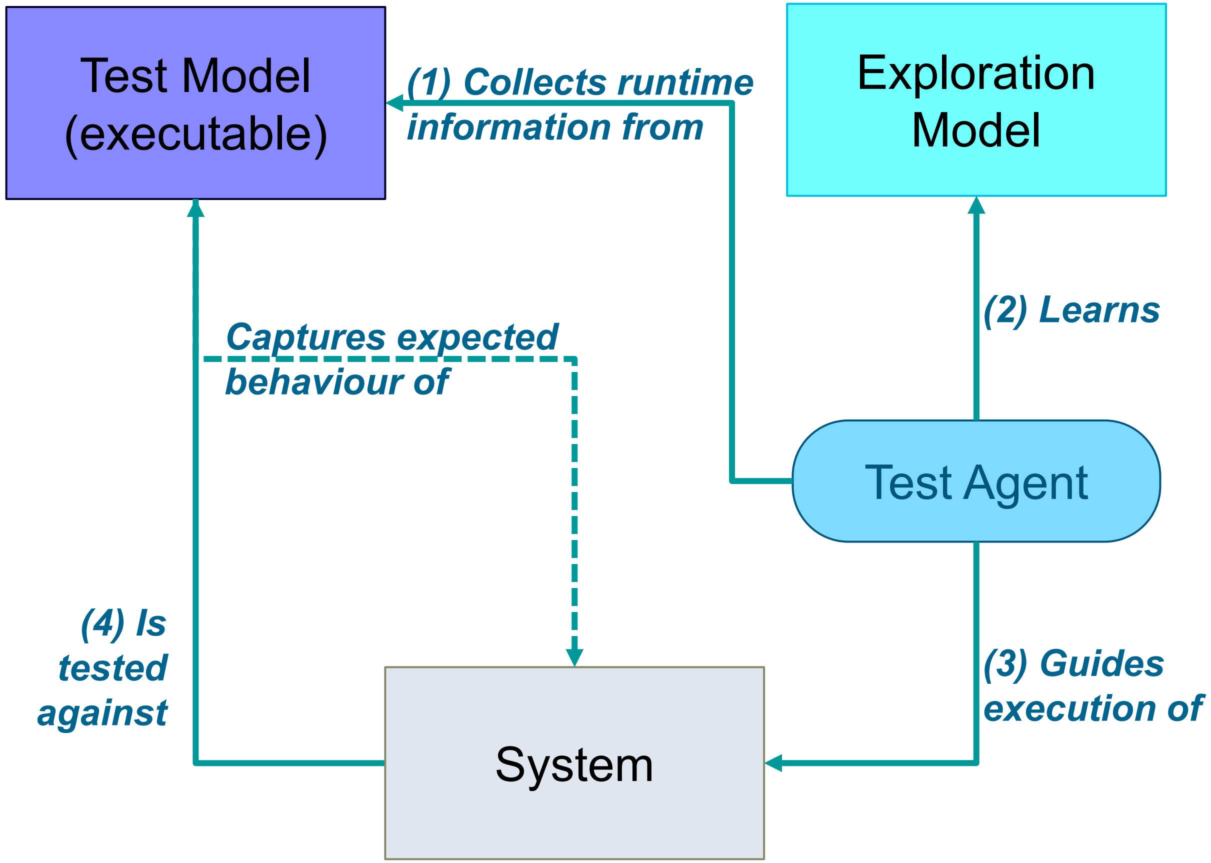
Cannot be verified, unreachable

Derivation from expected behaviour

(1) System failure

(2) Invalid property, i.e. wrong model

Adaptive testing with learning



Model concepts

- Test model → set of test commands capturing system properties
- Exploration model → is learnt from executions

Test goals

- Provide evidence that properties are satisfied
- Maximise coverage of exploration model

Test execution

1. Collect information about test commands
2. Learn model from executed test commands
3. Decide at runtime about next test commands
4. Run test commands against system

Learning and inference of state machines from test execution traces

Trace: Execution sequence of inputs and outputs

- I/O trace: sequence of input/output pairs
- Positive/negative samples: input sequence with evaluation of reached output

- Set of traces form a Prefix Tree

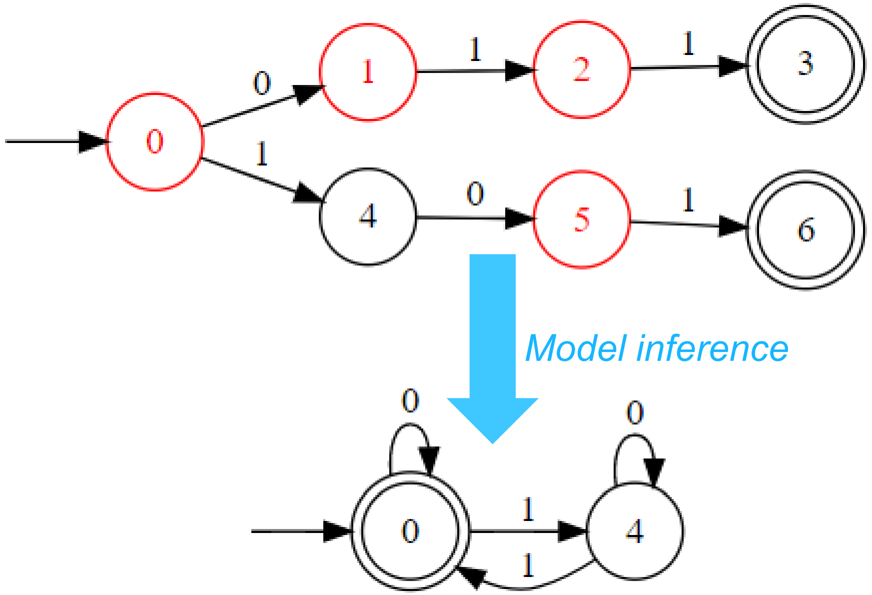
Inferred machine reproduces all traces from the training set with fewer states

- I/O traces → Moore machine
- Pos/neg. samples → DFA

Learning during the DevOps cycle, continuously

- Passive → Collect example traces randomly
- Active → Collect traces and derive queries

Example:
Input alphabet I = {0, 1}
Sample sets $\mathbf{S}_+$ = {011, 101}, $\mathbf{S}_-$ = {1}



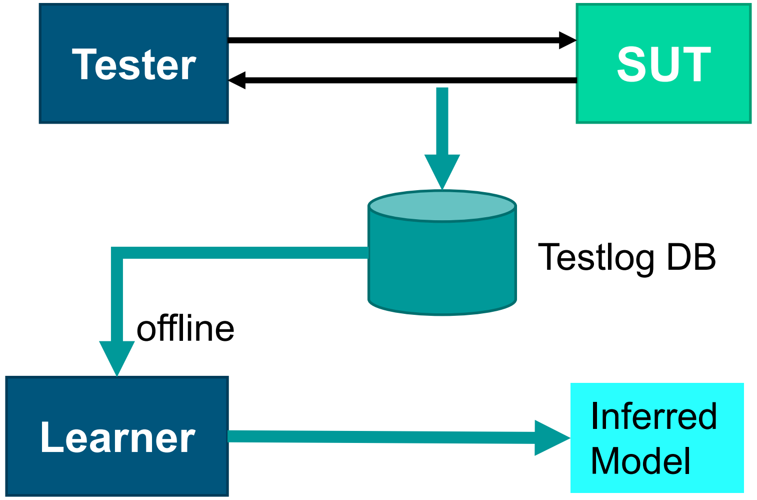
Passive learning and machine inference

Approach

- Database to store test logs from test execution runs
- Test logs interpreted as I/O traces to construct a prefix tree
- Moore machine is inferred from the prefix tree

Adequacy criterion for machine inference

- Preferred: "Identification in the limit"
  All traces <u>longer than *n*</u> can be produced from equivalent machines
- Realistic:
  All traces <u>up to length *n*</u> can be produced from equivalent machines
- Approximations to the minimum solution might be good enough

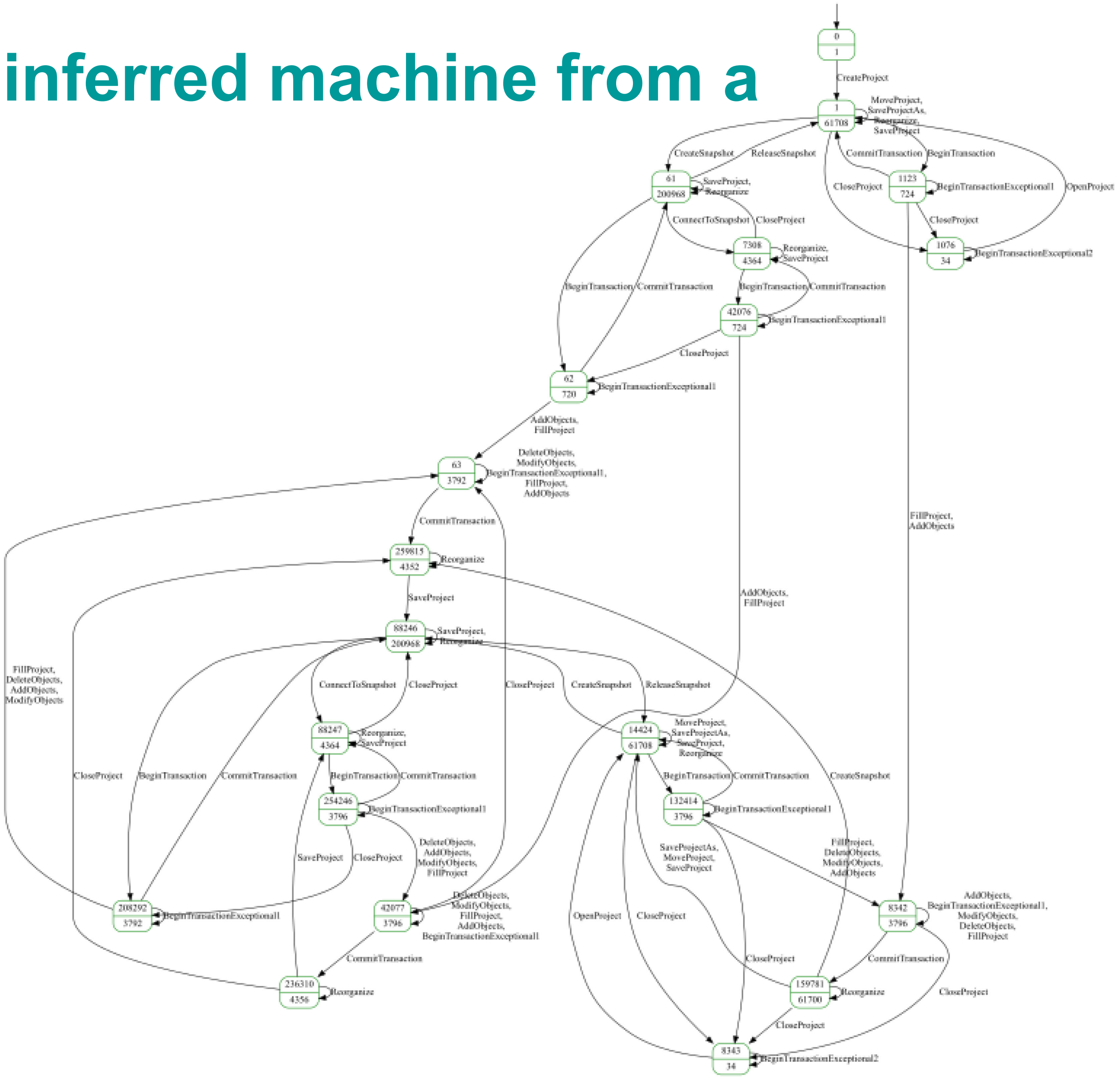


Passive Learning

Practical example of an inferred machine from a SW component test

- Inference from 544 test runs
- Building prefix tree in 0.754 sec
- Size of prefix tree is 275,881 states

- Inferring Moore machine in 11.312 sec
- Using 682.2 MB of heap memory
- Size of inferred machine is 21 states



Conclusions

Exploratory testing doesn’t need to stop at scripted tests

Adaptive testing preserves the exploration capability during runtime

Combining adaptive testing with learning to reach a defined level of state coverage

Learning helps find optimal solutions to a given coverage goal utilising:

- Exploration of new SUT behaviour
- Exploitation of learned knowledge