Software Testen VL (188.280)

Automating Test Automation

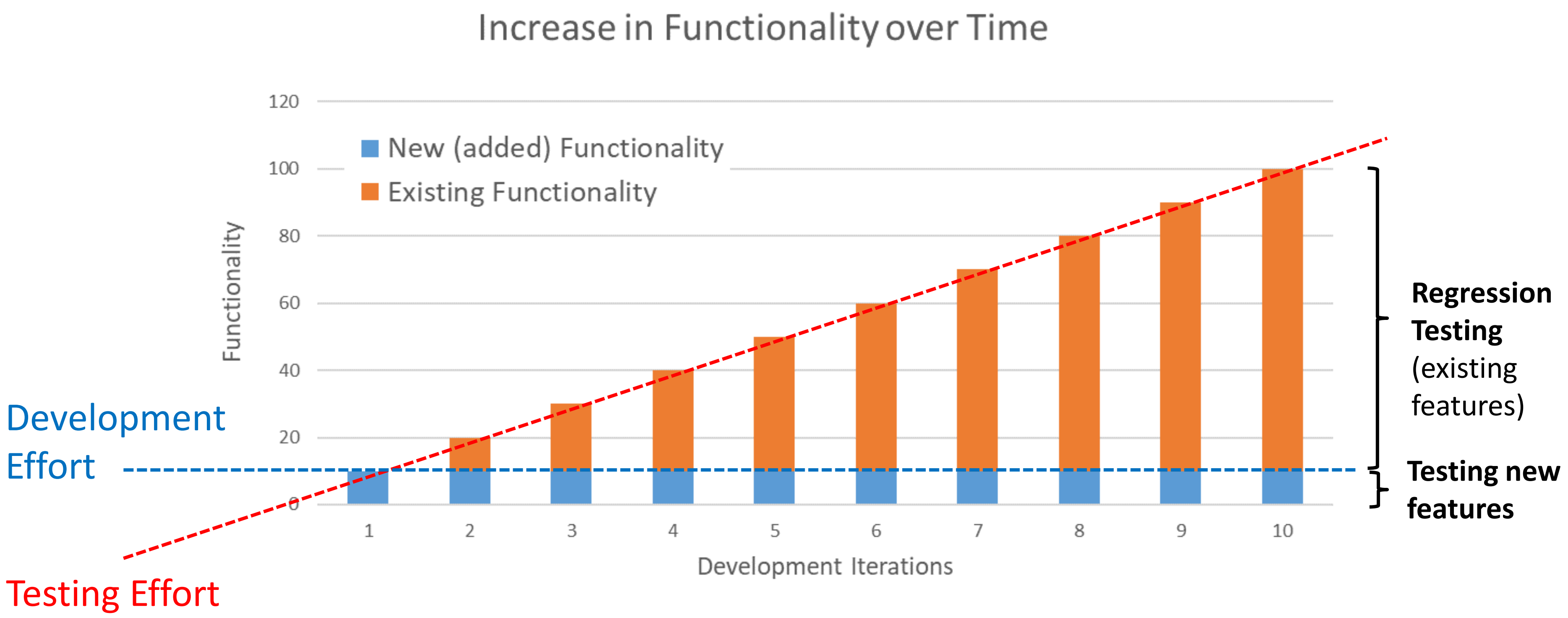
Rudolf Ramler[1] Dietmar Winkler[2]

[1] Software Competence Center Hagenberg

[2] Vienna University of Technology, Institute of Information Systems Engineering, Information And Software Engineering Group, Quality Software Engineering

rudolf.ramler@tuwien.ac.at
http://www.scch.at

Increasing Testing Effort



- Constantly increasing testing effort due to accumulating functionality over time
- Need for regression testing of existing features (= testing for side effects due to new features)

Transition to Test Automation

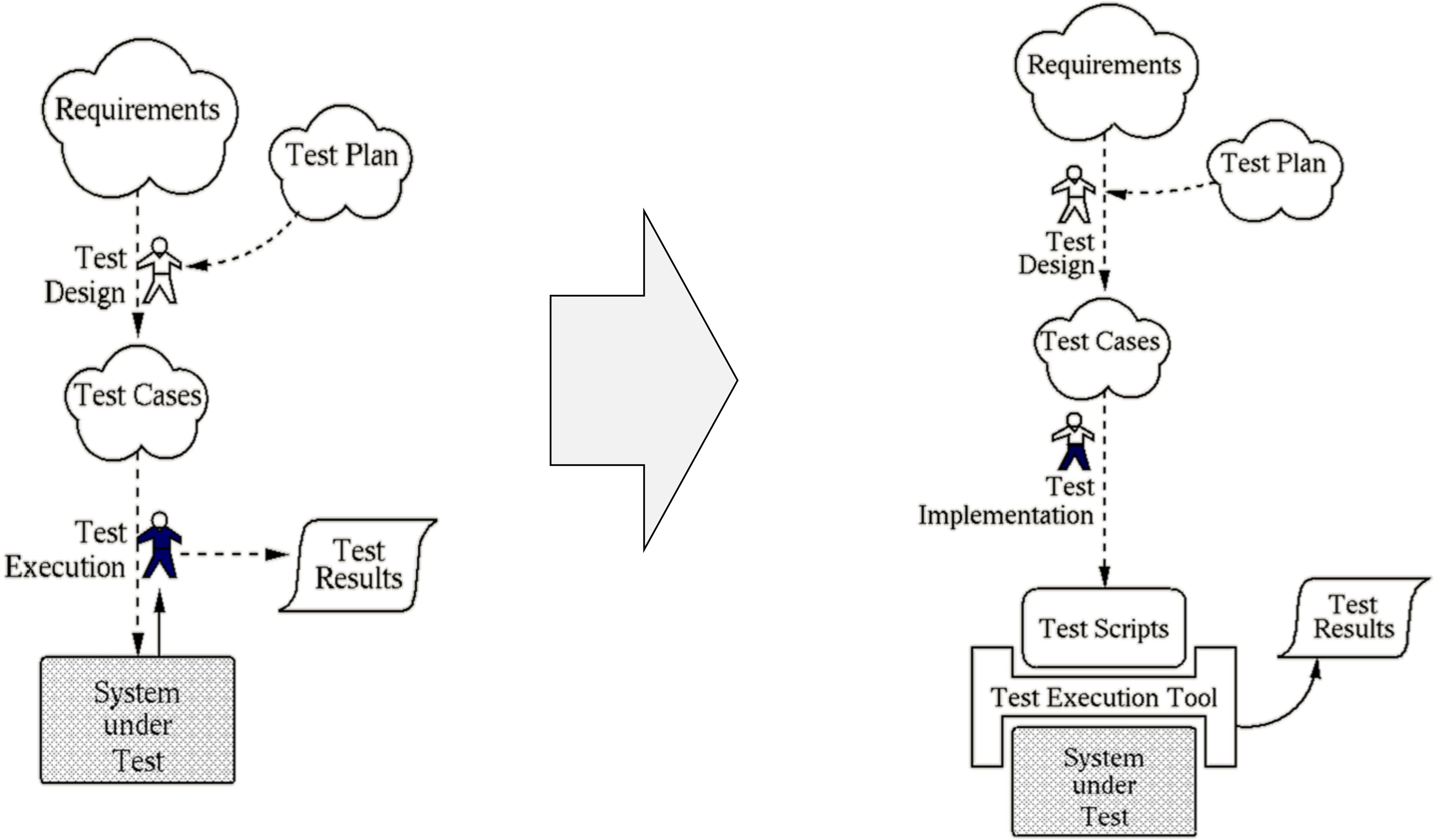
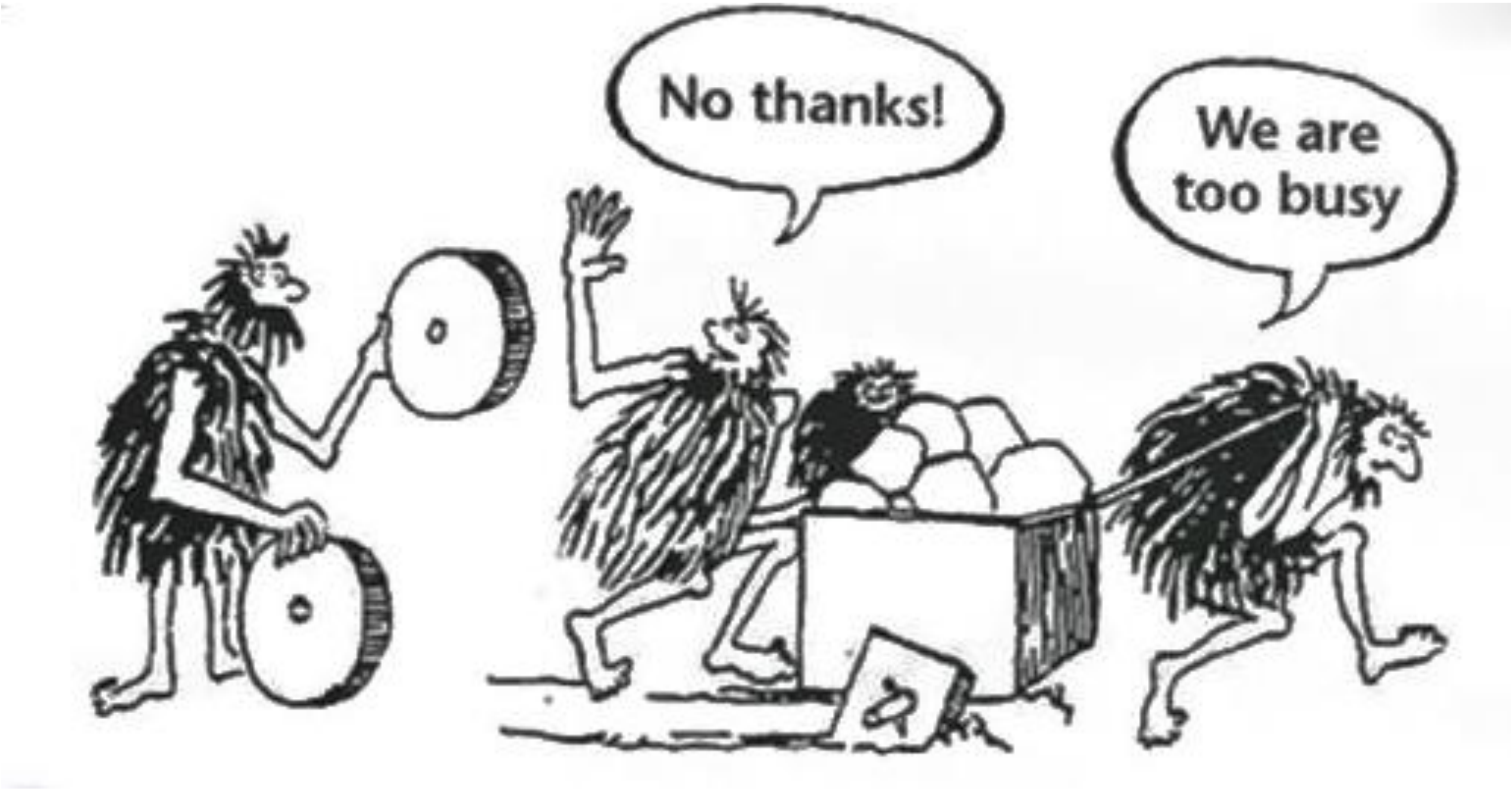
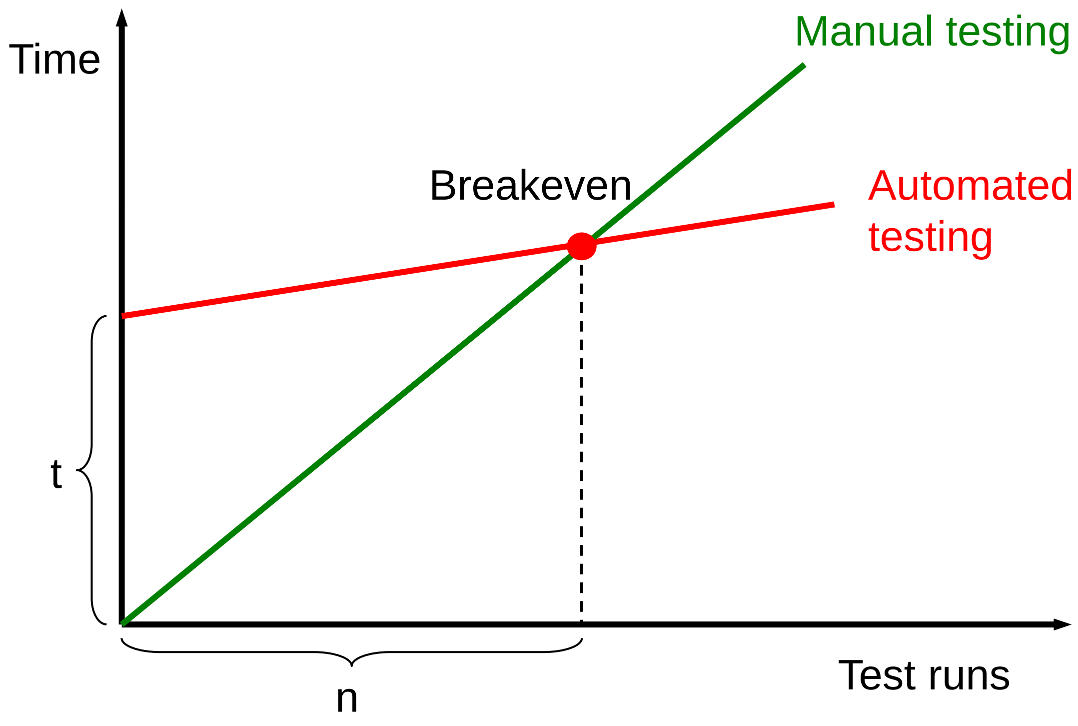


Figure from Utting & Legeard, Practical Model-Based Testing, 2006

Economics of Test Automation



Economics of Test Automation



Investment in automation (t) pays off after n repetitions

Caveats of this overly simplistic model

- Realistic estimates for number *n* test runs, until break even is reached?
- Upfront investment in automation per test case and for automation infrastructure in general (tools, stable test environment, reporting, ...)
- Test maintenance: Automated testing curve is not linear but increases over time
- Repeating the **exact same automated test** n-times vs. executing n **different manual tests** covering a wide variety of scenarios

Goal: Advanced Automation

Automated Test Design and Test Implementation

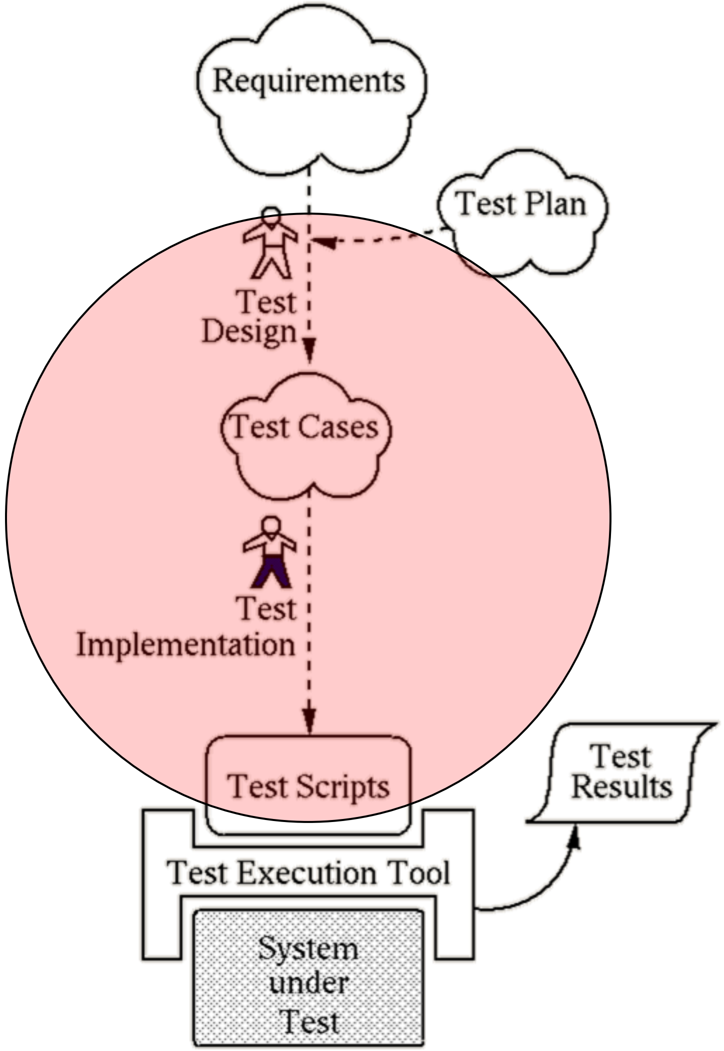


Figure from Utting & Legeard, Practical Model-Based Testing, 2006

Overview

- Random testing
- Feedback-directed random testing
- Assertions as test oracle
- Model-based testing
- Model-based testing by example
- ModelJUnit
- Closing thoughts

Recap

Structural (white box) test design

- Goal: Cover all statements, decisions, etc.
- Next best test case: one that increases coverage

```
int abs(int x) {
  if (x < 0) {
    return -x;
  } else {
    return x;
  }
}
```

Cover all statements, decisions, etc.

Recap

Specification (black box) test design

- Goal: Cover all input and/or output combinations
- Next best test case: value of an input class not used so far

Equiv. classes: [MAX_INT..0], [-1..MIN_INT]
Boundary values: MAX_INT, 1, 0, -1, MIN_INT

```
int abs(int x) {
  if (x < 0) {
    return -x;
  } else {
    return x;
  }
}
```

Random Testing

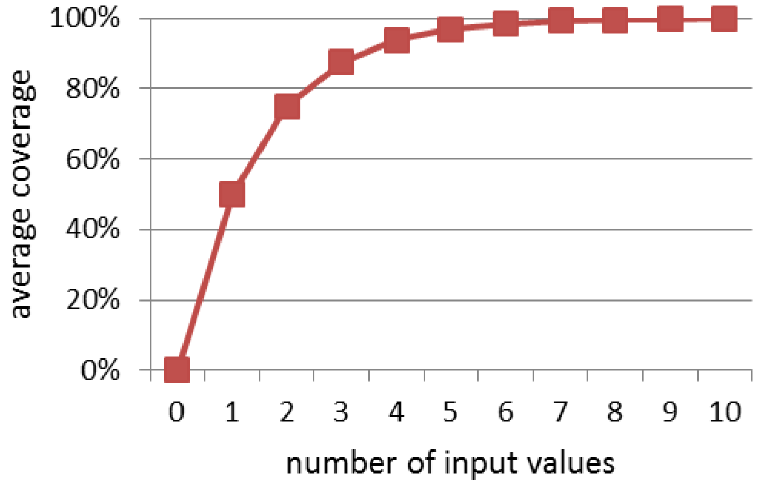
Random input values (black box test design)

- Works without knowledge of partitions and boundaries, based on assumptions about distributions
- Goal: Cover all input and/or output combinations
- Next best test case: new random value

Random.randInt()

```
int abs(int x) {
  if (x < 0) {
    return -x;
  } else {
    return x;
  }
}
```

Number of random input values and coverage:



Random Testing

random testing: A black box test design technique where test cases are selected, possibly using a pseudo-random generation algorithm, to match an operational profile.

- Random input: random in the mathematical sense
  → basis for statistically meaningful evaluation
- Will perform operations no sane human would ever perform
  “Nobody would ever do *that*!”
- Used for (additional) testing non-functional attributes such as reliability and performance

Random Testing

Zergling Rush

To overwhelm an opponent through the use of cheaply made units at the expense of any long-term strategy (taken from Blizzard's "StarCraft").

www.urbandictionary.com

Harry Robinson: How to Build Your Own Robot Army, STAR West 2006

Random Testing Approaches

- **monkey testing:** Testing by means of a random selection from a large range of inputs and by randomly pushing buttons, ignorant of how the product is being used.
  - Random testing applied to graphical user interfaces
  - Android UI/Application Exerciser Monkey

> The Monkey is a program that runs on your emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. You can use the Monkey to stress-test applications that you are developing, in a random yet repeatable manner

- **fuzz testing:** testing by means of providing invalid, unexpected, or random input data to the system under test.
  - Generating invalid input data by making small changes to valid data, e.g., corrupting an input file, malformed data formats, exceeding (size) boundaries
  - Commonly used for security testing

Problems of Random Testing

- **Generation problem**
  - How to (statistically) sufficiently cover a large input space?
  - Randomly generated data follows a homogeneous distribution; interactions may require special inputs
    - "Guessing" the correct username/password with random input?
- **Test minimization problem**
  - Large number of redundant test cases
  - Large number of generated tests contain lots of noise
    - Failing test cases may have a large number of irrelevant steps, which increases the effort for reproduction in debugging
  - False positives (e.g., tests fail although system is correct)
- **Oracle problem**
  - Generating random data is easy. But how do we determine the outcome (pass/fail) of a random test?

Test Oracle

- **test oracle:** A source **to determine expected results** for comparison with the actual result of the execution in order to decide pass or fail of the test

- Examples: specification, user manual, other systems (benchmark), previous version of the system, models, assertions in the code, human knowledge, ...

- **Differential testing:** comparing the output of two different systems
- **Delta testing:** comparing the output of two versions of the system

One of the grand challenges in software testing

Overview

- Random testing
- Feedback-directed random testing
- Assertions as test oracle
- Model-based testing
- Model-based testing by example
- ModelJUnit
- Closing thoughts

https://randoop.github.io/randoop/

Randoop

Automatic unit test generation for Java

Download the Latest Release | View the GitHub Project

What is Randoop?

Randoop is a unit test generator for Java. It automatically creates unit tests for your classes, in JUnit format.

The Randoop manual tells you how to install and run Randoop.

How does Randoop work?

Randoop generates unit tests using feedback-directed random test generation. This technique randomly, but smartly, generates sequences of method/constructor invocations for the classes under test. Randoop executes the sequences it creates, using the results of the execution to create assertions that capture the behavior of your program. Randoop creates tests from the code sequences and assertions.

Randoop can be used for two purposes: to find bugs in your program, and to create regression tests to warn you if you change your program's behavior in the future.

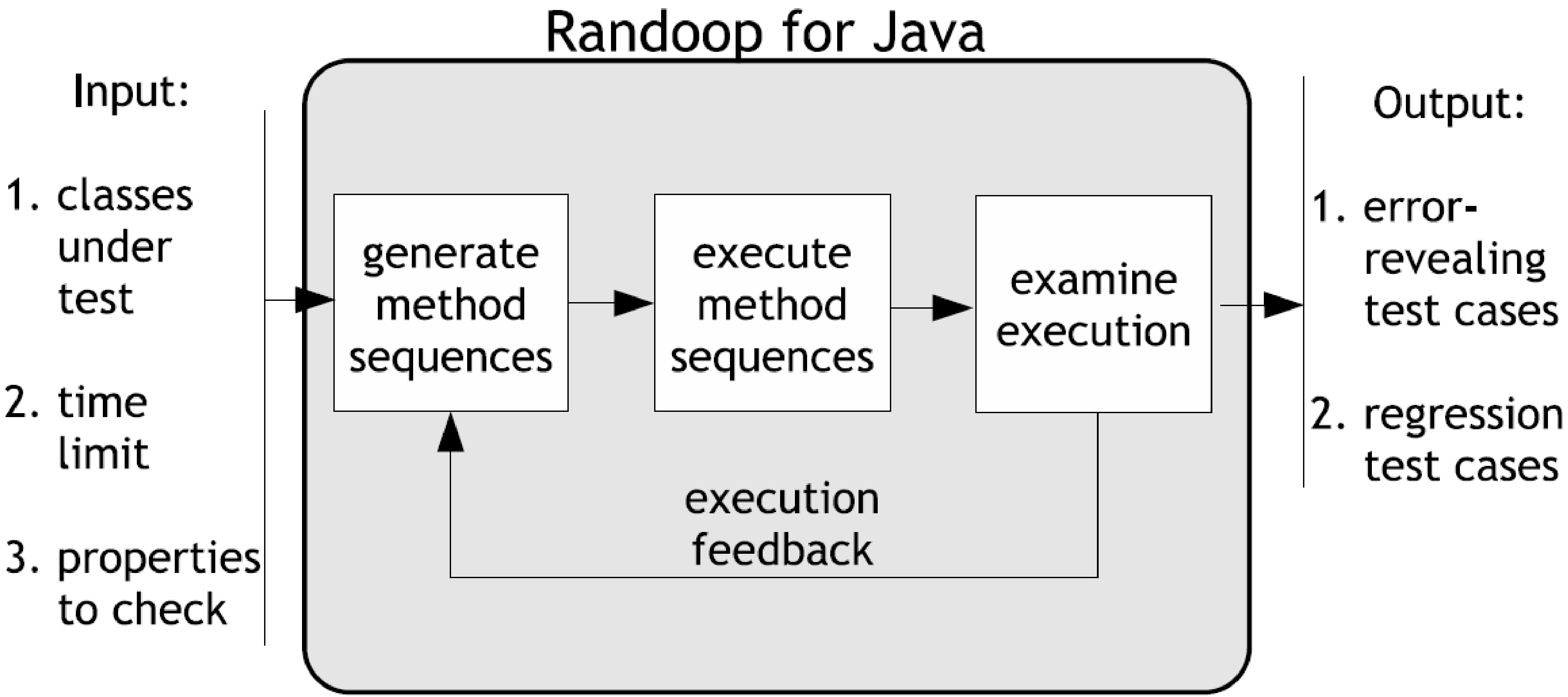
Randoop's combination of randomized test generation and test execution results in a highly effective test generation technique. Randoop has revealed previously-unknown errors even in widely-used libraries including Sun's and IBM's JDKs and a core .NET component. Randoop continues to be used in industry, for example at ABB corporation.

Randoop: Execution Sequence

Constructor Summary
`BitSet()`   Creates a new bit set.
`BitSet(int nbits)`   Creates a bit set whose initial size is large enough to explicitly represent bits with indices in the range `0` through `nbits-1`.

Method Summary	
`void`	`and(BitSet set)` Performs a logical **AND** of this target bit set with the argument bit set.
`void`	`andNot(BitSet set)` Clears all of the bits in this `BitSet` whose corresponding bit is set in the specified `BitSet`.
`int`	`cardinality()` Returns the number of bits set to `true` in this `BitSet`.
`void`	`clear()` Sets all of the bits in this BitSet to `false`.
`void`	`clear(int bitIndex)` Sets the bit specified by the index to `false`.
`void`	`clear(int fromIndex, int toIndex)` Sets the bits from the specified fromIndex(inclusive) to the specified toIndex(exclusive) to `false`.
`Object`	`clone()` Cloning this `BitSet` produces a new `BitSet` that is equal to it.
`boolean`	`equals(Object obj)` Compares this object against the specified object.
`void`	`flip(int bitIndex)` Sets the bit at the specified index to to the complement of its current value.
`void`	`flip(int fromIndex, int toIndex)` Sets each bit from the specified fromIndex(inclusive) to the specified toIndex(exclusive) to the complement of its current value.
`boolean`	`get(int bitIndex)` Returns the value of the bit with the specified index.
`BitSet`	`get(int fromIndex, int toIndex)` Returns a new `BitSet` composed of bits from this `BitSet` from `fromIndex`(inclusive) to `toIndex`(exclusive).
`int`	`hashCode()` Returns a hash code value for this bit set.
`boolean`	`intersects(BitSet set)` Returns true if the specified `BitSet` has any bits set to `true` that are also set to `true` in this `BitSet`.
`boolean`	`isEmpty()`

Feedback-directed Random Testing



C. Pacheco, M.D. Ernst: Randoop: Feedback-directed Random Testing for Java. OOPSLA '07

Randoop: Error-Revealing Tests

```
// Fails on Sun 1.5, 1.6.
public static void test1() {
   LinkedList l1 = new LinkedList();
   Object o1 = new Object();
   l1.addFirst(o1);
   TreeSet t1 = new TreeSet(l1);
   Set s1 = Collections.unmodifiableSet(t1);
   Assert.assertTrue(s1.equals(s1));
}
```

Randoop: Regression Tests

```
// Passes on Sun 1.5, fails on Sun 1.6 Beta 2.
public static void test2() {
   BitSet b = new BitSet();
   Assert.assertEquals(64, b.size());
   b.clone();
   Assert.assertEquals(64, b.size());
}
```

Overview

- Random testing
- Feedback-directed random testing
- Assertions as test oracle
- Model-based testing
- Model-based testing by example
- ModelJUnit
- Closing thoughts

Test Oracle: Assertions

- An *assertion* is a statement in the Java programming language that enables you to "test" your assumptions about your program in the form: **`assert expression;`**
  - Assertions contain a boolean *expression* that **you believe** will be true when it is executed
    - If **not true**, the system will throw an **`AssertionError`**
    - If **true**, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors
- Using assertions as test oracle
  - Command-line: **`-enableassertions`** or **`–ea`** (default: off)
  - Run test case generation, e.g., Randoop
    - If an *AssertionError* is caught, a failing test is generated

What are Assertions?

https://stackoverflow.com/questions/2758224/what-does-the-java-assert-keyword-do-and-when-should-it-be-used

Assertions (by way of the **assert** keyword) were added in Java 1.4. They are used to verify the correctness of an invariant in the code. They should never be triggered in production code, and are indicative of a bug or misuse of a code path. They can be activated at run-time by way of the `-ea` option on the `java` command, but are not turned on by default.

An example:

```
public Foo acquireFoo(int id) {
  Foo result = null;
  if (id > 50) {
    result = fooService.read(id);
  } else {
    result = new Foo(id);
  }
  assert result != null;

  return result;
}
```

share improve this answer

edited Sep 2 '15 at 16:25

answered May 3 '10 at 14:14

Ophidian

7,647 ● 2 ● 21 ● 24

Assertions vs. JUnit Asserts

```
public class Triangle {

public Triangle(int a, int b, int c) {..}

public int calcPerimiter() {
    assert (a>0 && b>0 && c>0) :
           "Not a valid triangle";
    ...
}
```

- **Part of the code**
- Knowledge of developer
- Checked in any execution
  (when VM argument *-ea* is set)
- Generic expression (assert for all instances)

```
public class TriangleTest {

@Test public void testIsValid() {

   assertTrue(Triangle.isValid(1,1,1));
   assertFalse(Triangle.isValid(0,1,1));
   assertFalse(Triangle.isValid(1,0,1));
   assertFalse(Triangle.isValid(1,1,0));
```

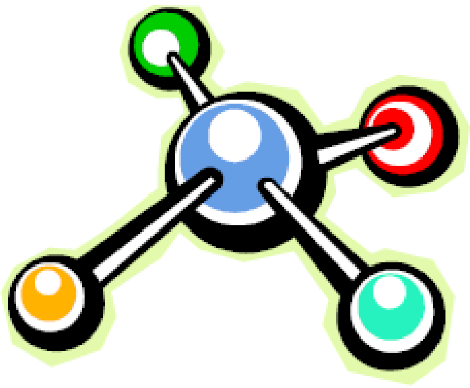
- **Part of tests**
- Knowledge of tester
- Checked in test run
- Test scenario (assert for specific instance)

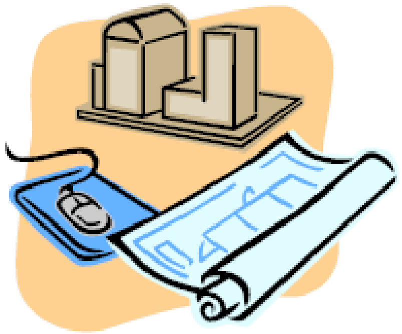
Overview

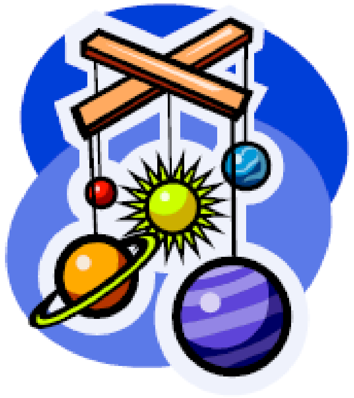
- Random testing
- Feedback-directed random testing
- Assertions as test oracle
- Model-based testing
- Model-based testing by example
- ModelJUnit
- Closing thoughts

What is a model?

- A model is a description of a system
- A model is an abstraction of the system it describes
- Models help to understand and predict the system’s behavior







What is a model?

An *orrery* is a mechanical model of the solar system that illustrates or predicts the relative positions and motions of the planets and moons

Model-based Testing

Model-based testing is defined as

automatable derivation of concrete test cases from abstract formal models, and their execution

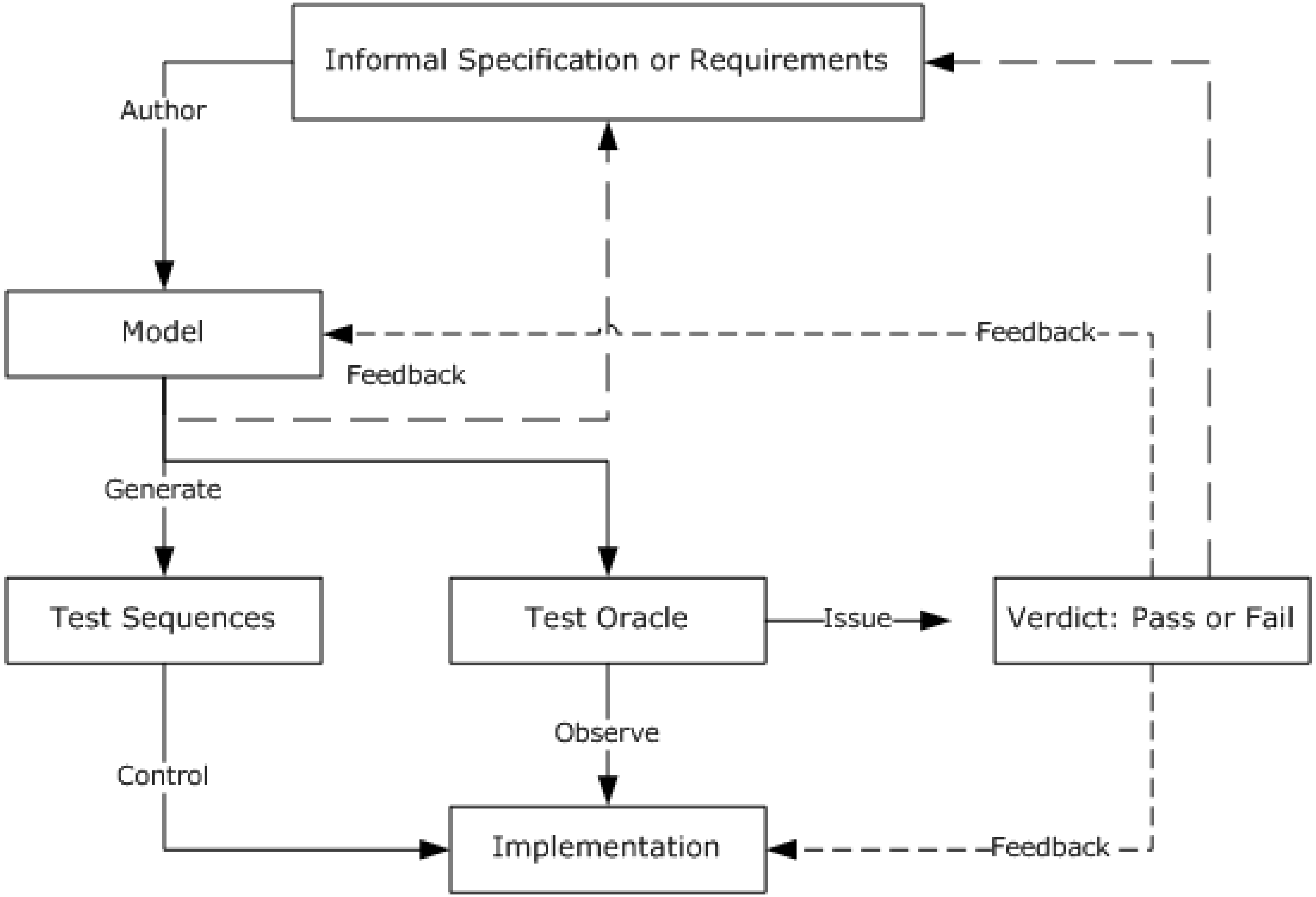
- The output of the model (expected behavior) is compared to the output of the system under test (actual behavior)

  → Supply the action and see if the system responds as expected

  → Create one model and derive many tests (at almost no additional cost)

- Note: The model must be simpler than the SUT, or at least easier to check, modify and maintain (Otherwise, the effort of modeling would equal the efforts of validating the SUT / implement the SUT)

Model-based Testing Workflow



https://msdn.microsoft.com/en-us/library/ee620469.aspx

Techniques

- Techniques for automated generation of test cases from a model
  – **Random generation** of tests, e.g. random walk
  – **Markov chains**, e.g., incorporating usage profiles
  – **Graph search** algorithms, e.g. node or arc coverage algorithms (including shortest path "Chinese Postman" algorithm)
  – **Model checking** or theorem proving: model checker yields paths that reach a certain state or transition
- On-line or off-line test generation
  – **Off-line**: Test cases (e.g. test scripts) are generated from a model → Test cases can be stored and maintained like conventional test cases, explicit test case selection
  – **On-line**: Test case generation and test execution at once → Testing non-deterministic systems (react to actual outputs)

Pros and Cons

Advantages

- Early bug detection
  - Modeling exposes problems in the specification and design
  - Modeling early in development
- Support for evolving requirements and test case maintenance
  - The model is easier to update than a suite of individual tests
  - The model can be re-used when the specifications change
- Reduced costs
  - Test cases are generated from the model → more test cases in less time
- Time to address advanced test issues
- Improved tester job satisfaction

Disadvantages

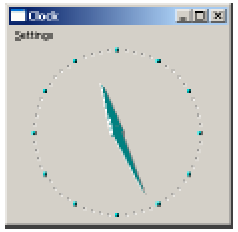
- Still high effort, shifted from testing to modeling
  - Initial effort for modeling + automation
  - State explosion and model complexity for large systems
  - Verification and validation of the model
- Advanced (modeling) skills of testers required
- Model is an abstraction of the system, deliberate omission of details may miss important details
- Scaling of model from simple to complex
  - Manual testing can start with a few complex test cases
- High effort for defining an appropriate oracle

Overview

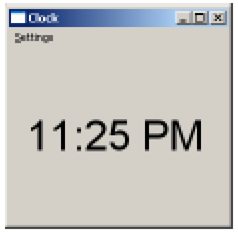
- Random testing
- Feedback-directed random testing
- Assertions as test oracle
- Model-based testing
- Model-based testing by example
- ModelJUnit
- Closing thoughts

Modeling Clock Actions

We also tracked how our actions change those values:



Digital

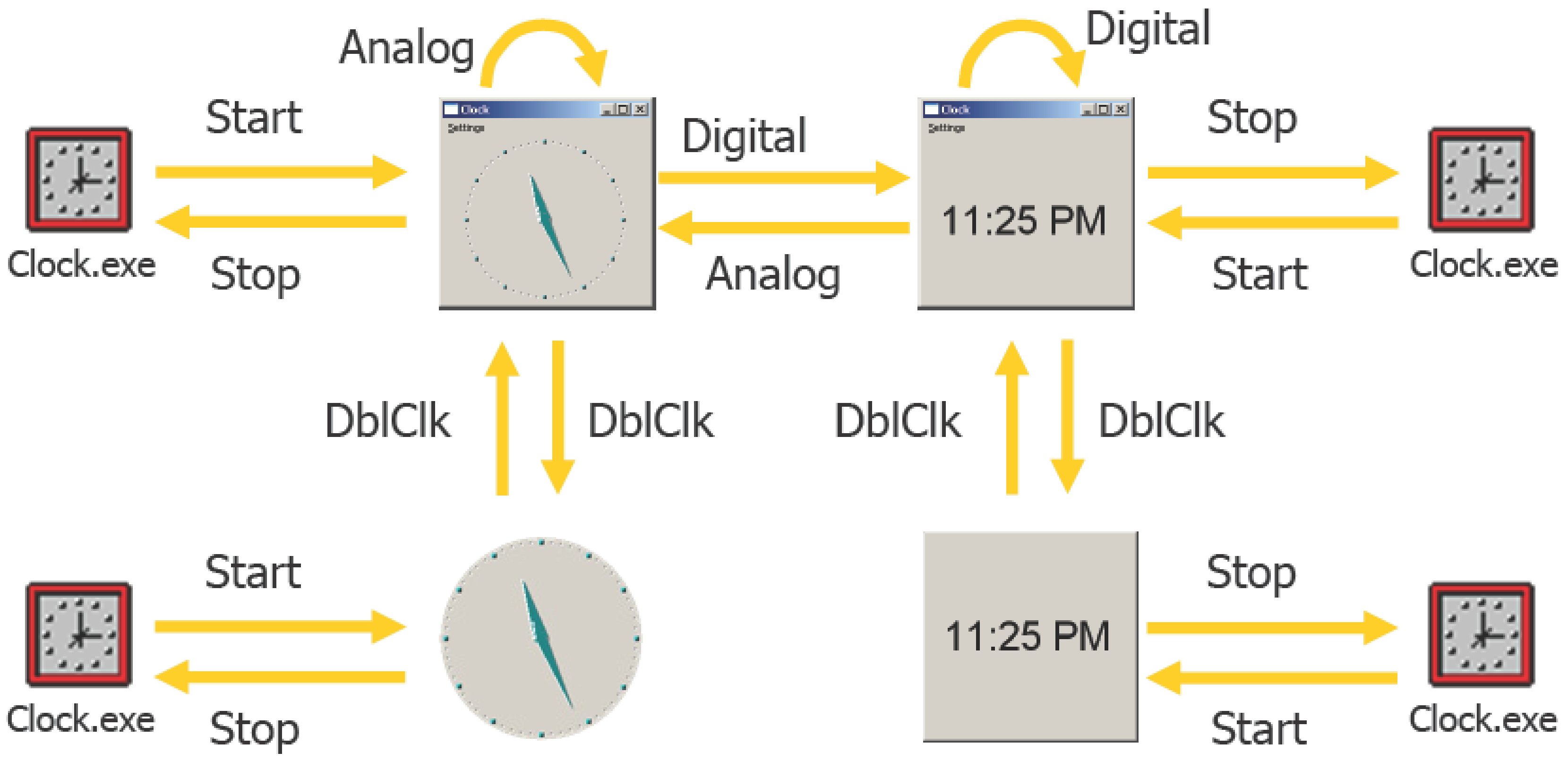


Running
Analog
Framed

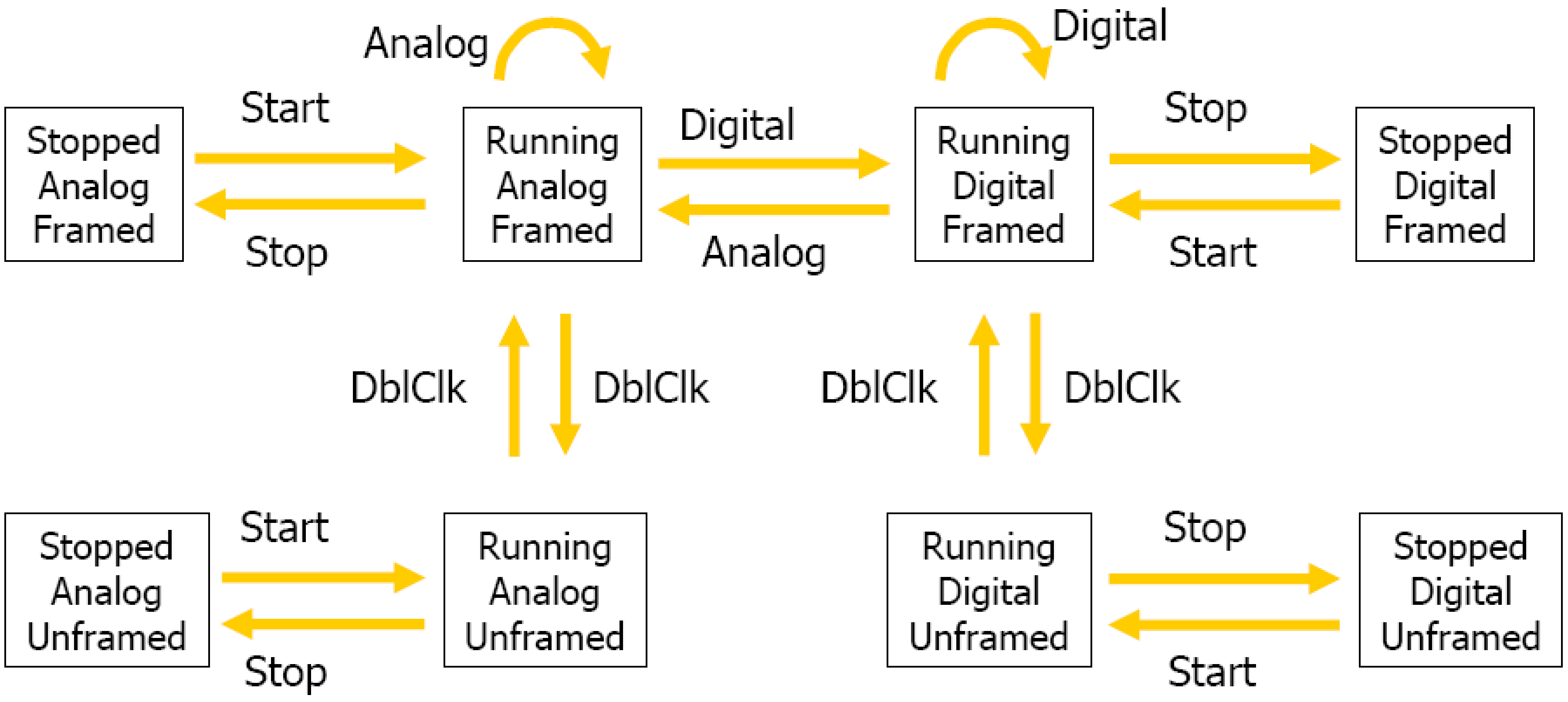
Digital

Running
Digital
Framed

So, we can replace this model …



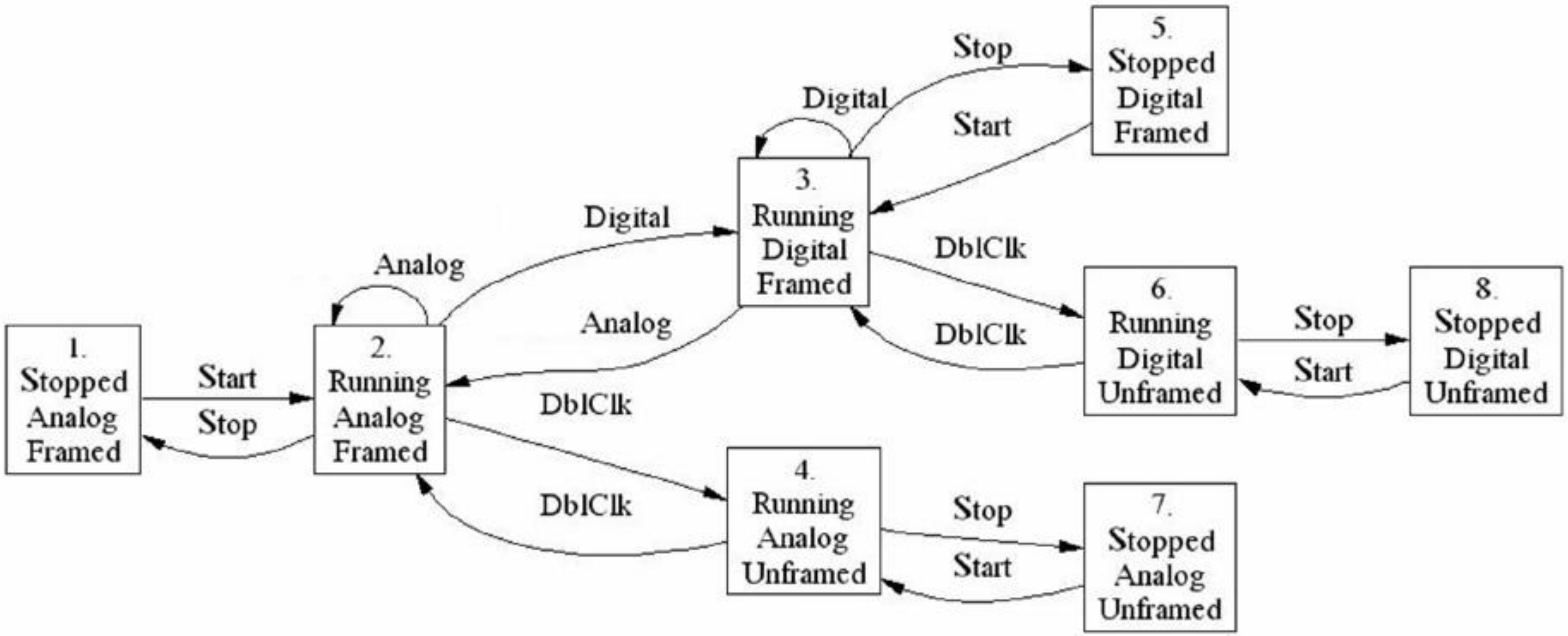
… with a state variable model



A generated state table!

STARTSTATE	ACTION	ENDSTATE
Stopped Analog Framed	Start	Running Analog Framed
Running Analog Framed	Stop	Stopped Analog Framed
Running Analog Framed	SelectAnalog	Running Analog Framed
Running Analog Framed	SelectDigital	Running Digital Framed
Running Analog Framed	DblClk	Running Analog Unframed
Running Digital Framed	Stop	Stopped Digital Framed
Running Digital Framed	SelectAnalog	Running Analog Framed
Running Digital Framed	SelectDigital	Running Digital Framed
Running Digital Framed	DblClk	Running Digital Unframed
Running Analog Unframed	Stop	Stopped Analog Unframed
Running Analog Unframed	DblClk	Running Analog Framed
Stopped Digital Framed	Start	Running Digital Framed
Running Digital Unframed	Stop	Stopped Digital Unframed
Running Digital Unframed	DblClk	Running Digital Framed
Stopped Analog Unframed	Start	Running Analog Unframed
Stopped Digital Unframed	Start	Running Digital Unframed

A state diagram

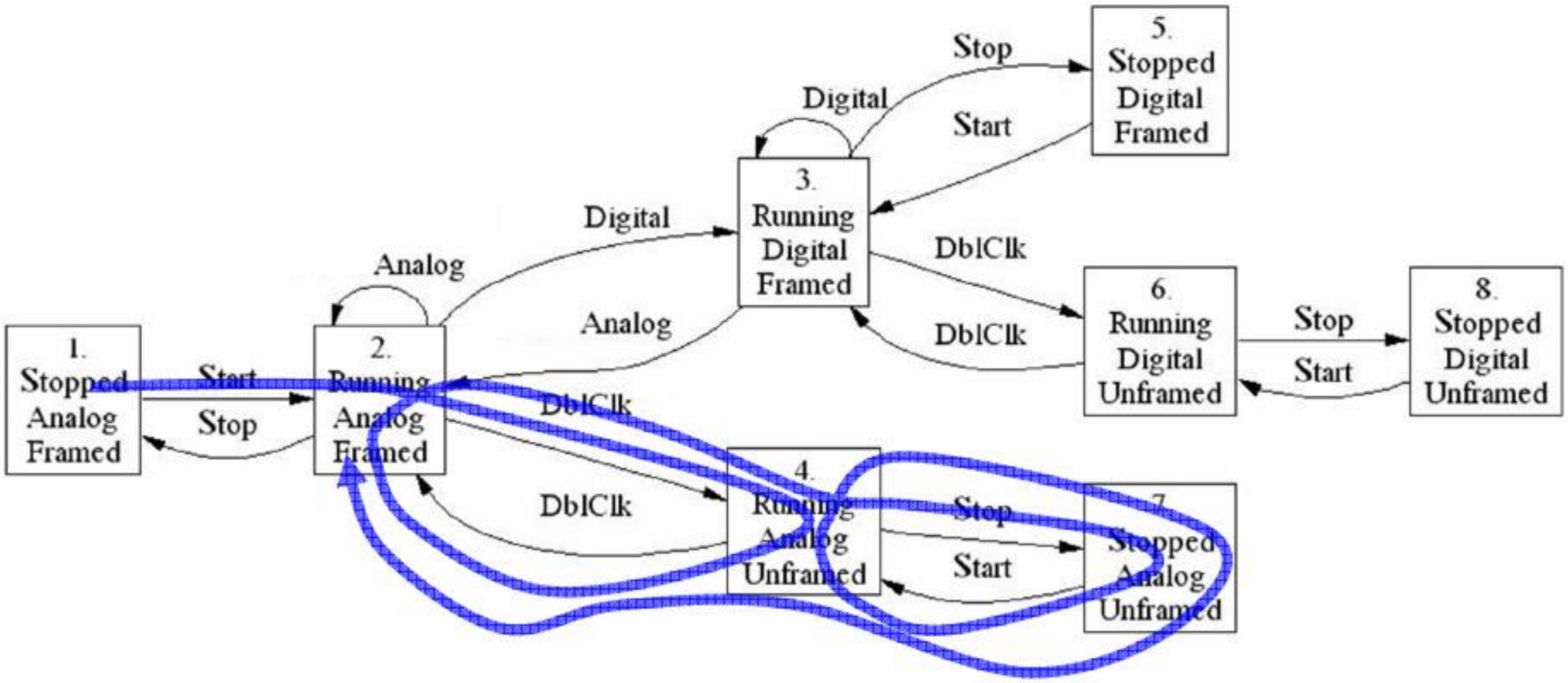


Generating Test Sequences

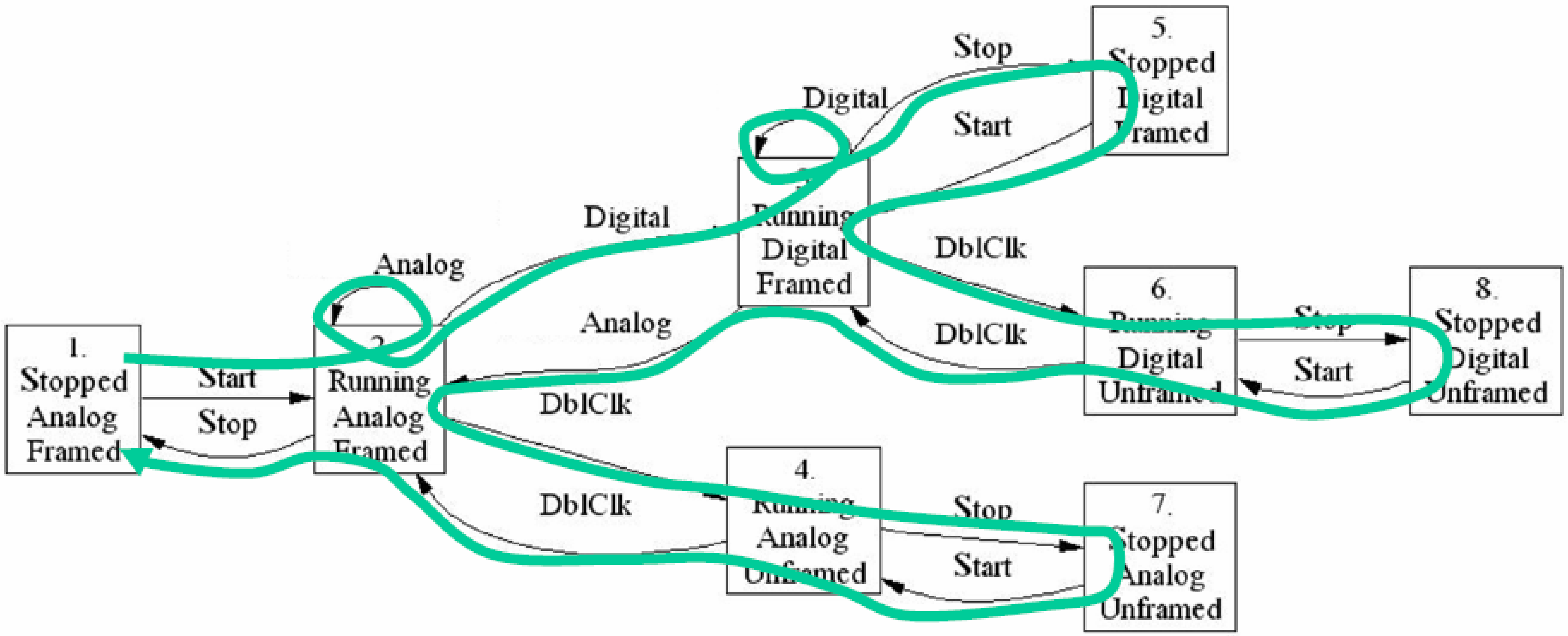
We can use the machine-readable model to create test sequences:

- Random walk
- All transitions
- Shortest paths first
- Most likely paths first

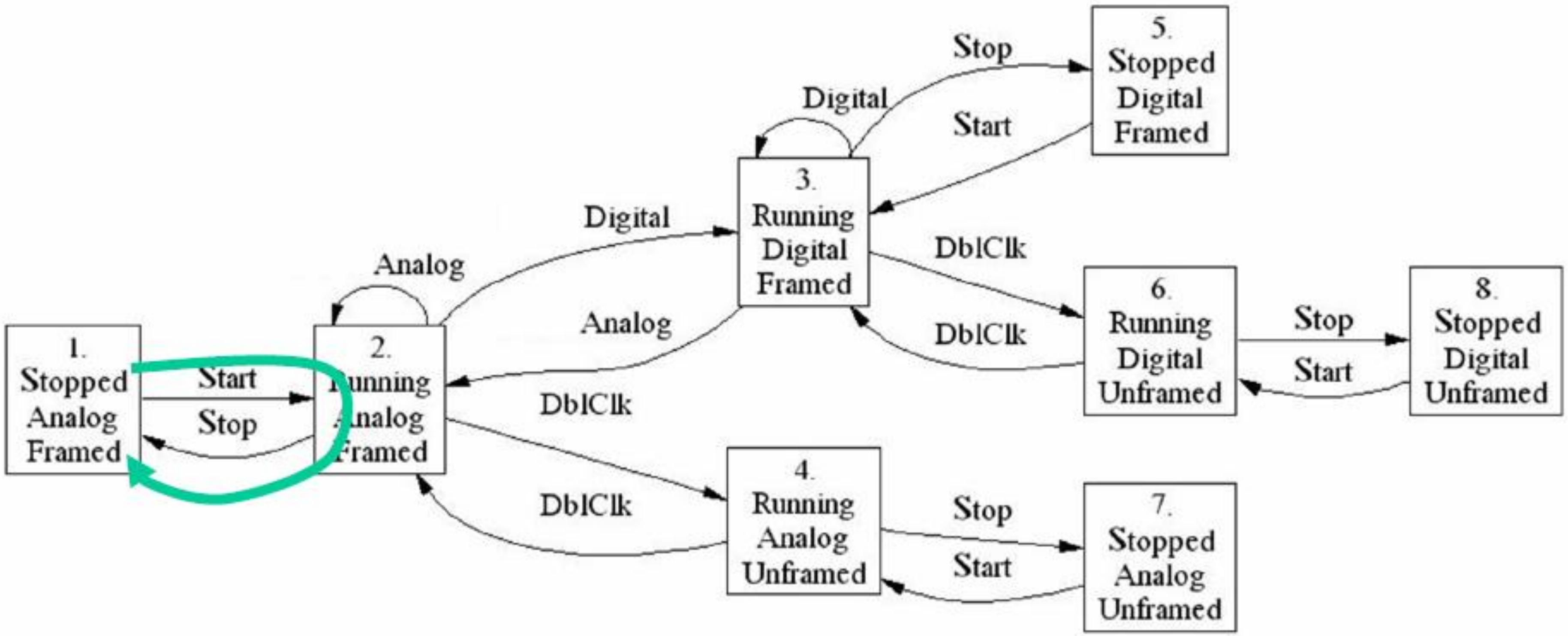
A Random Walk



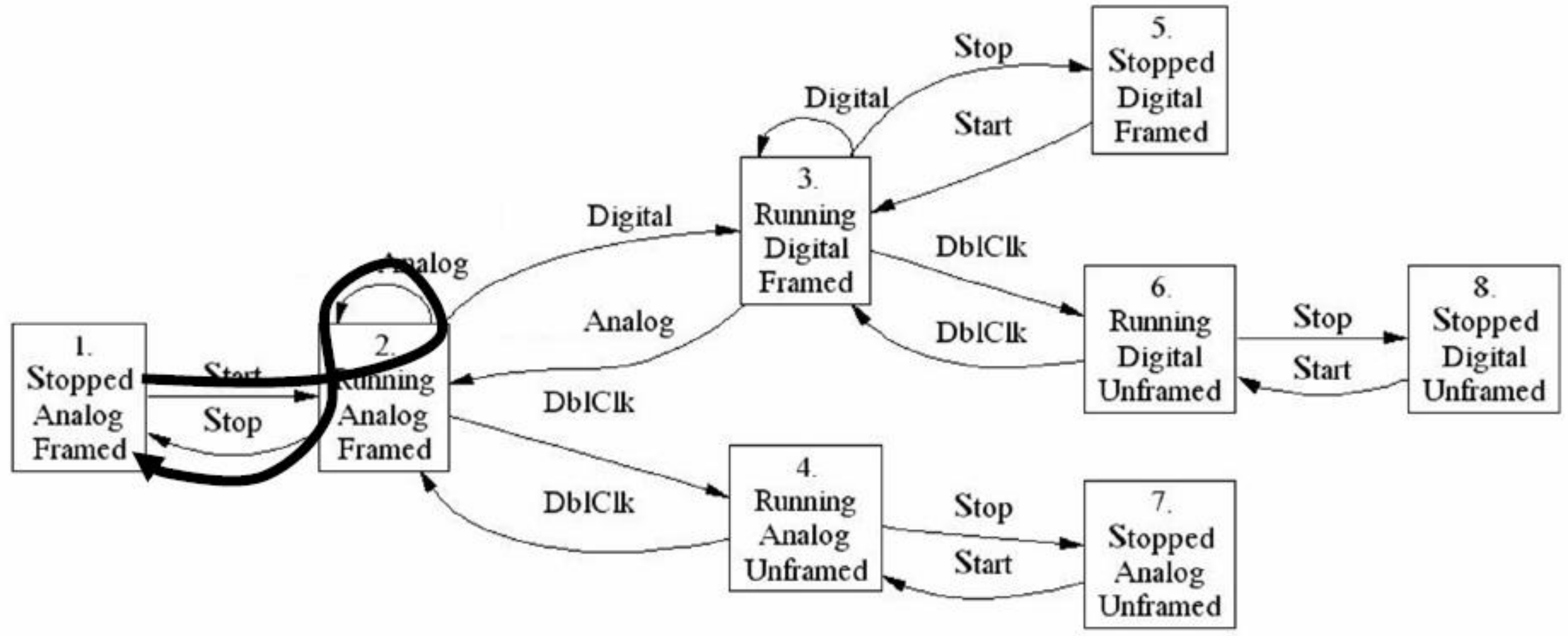
A sequence that hits all transitions



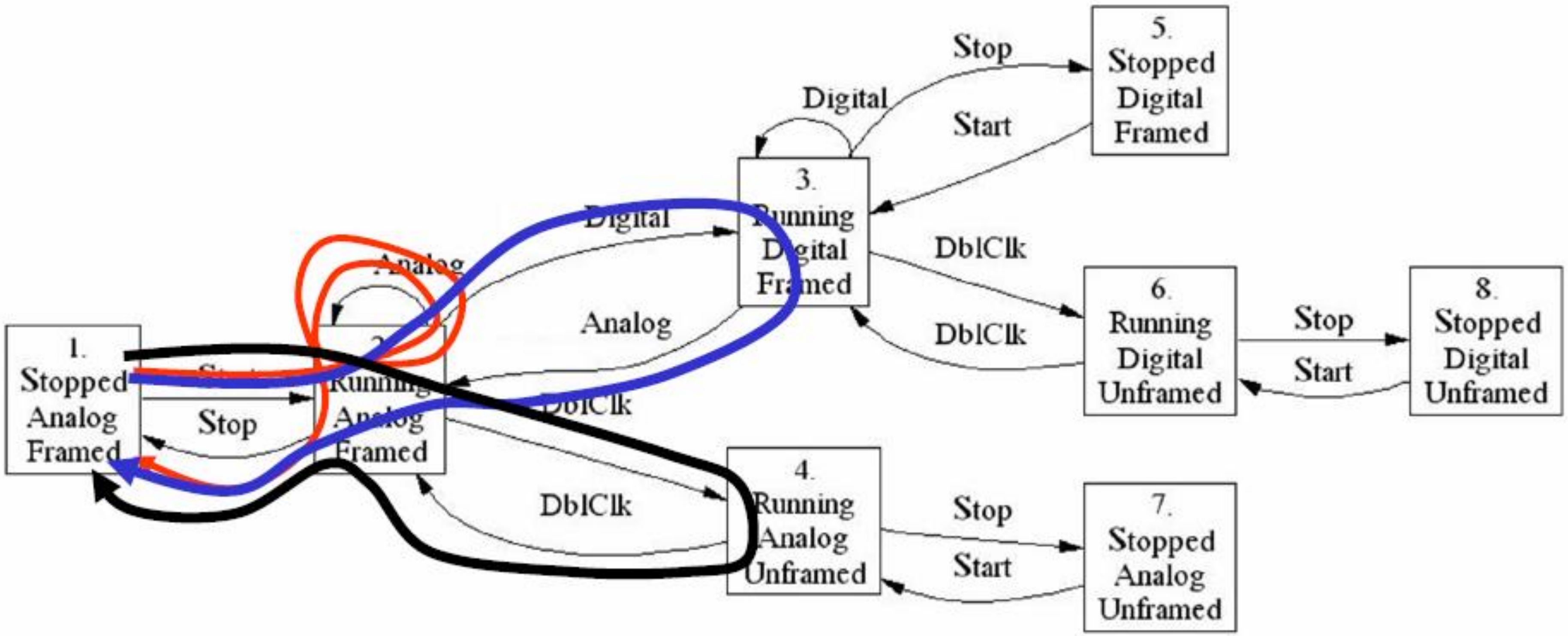
All paths of length 2



All paths of length 3

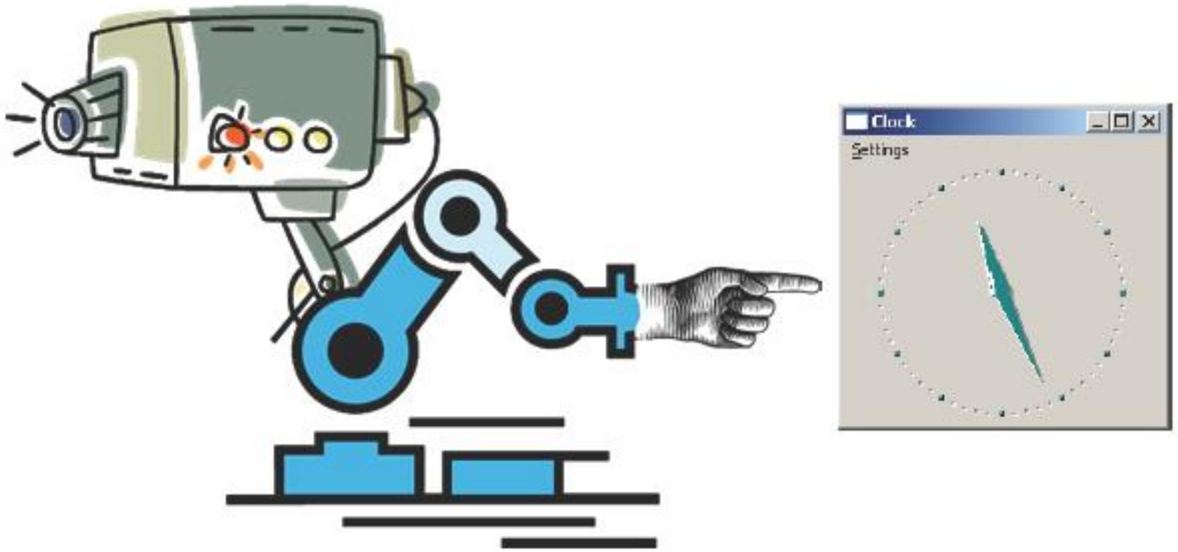


All paths of length 4



Executing the Test Actions

1. Start
2. Analog
3. Digital
4. Digital
5. Stop
6. Start
7. Double Click
8. Stop

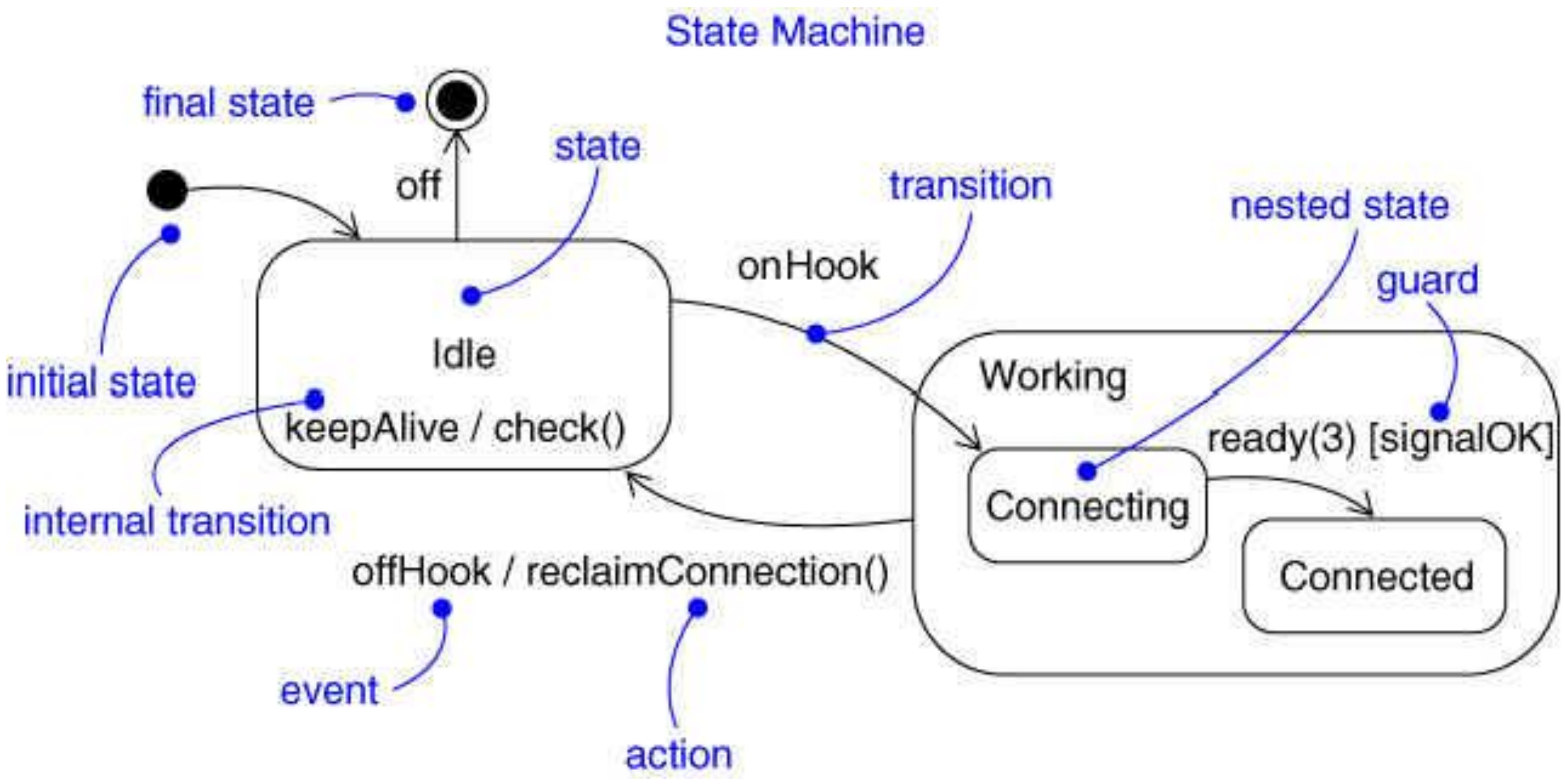


Overview

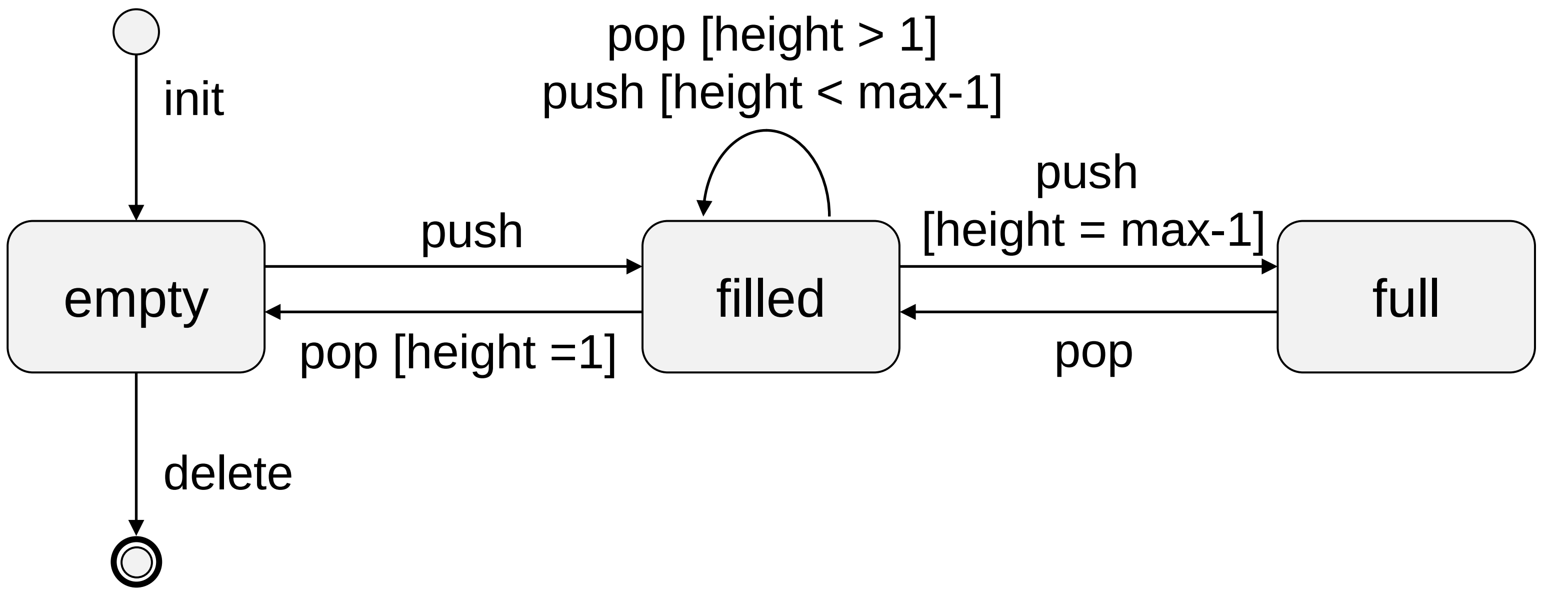
- Random testing
- Feedback-directed random testing
- Assertions as test oracle
- Model-based testing
- Model-based testing by example
- ModelJUnit
- Closing thoughts

What is a model?

- UML State Charts ...

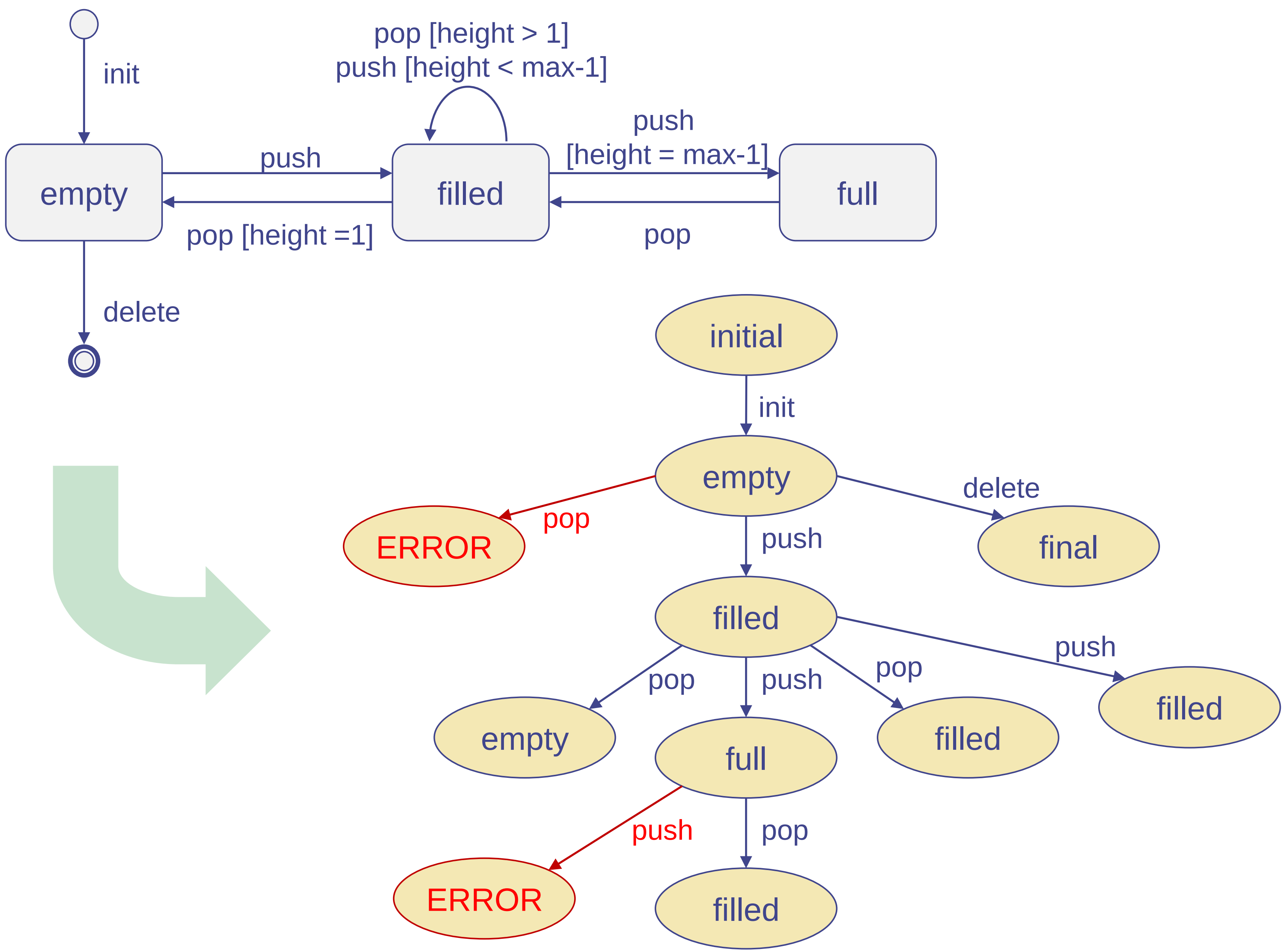


Example: Stack



states := {initial, empty, filled, full, final}

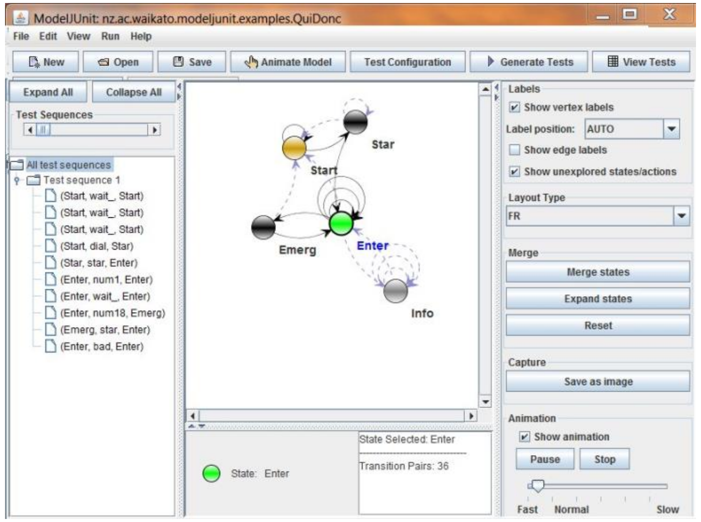
transitions := {initial→empty, empty→final, empty→filled, filled→empty, filled→full, full→filled, filled→filled}


init
pop [height > 1]
push [height < max-1]
empty
push
filled
push
[height = max-1]
full
pop [height =1]
pop
delete
initial
init
empty
pop
ERROR
delete
final
push
filled
pop
push
pop
push
empty
full
filled
filled
push
pop
ERROR
filled

Example: Stack - Derive test cases

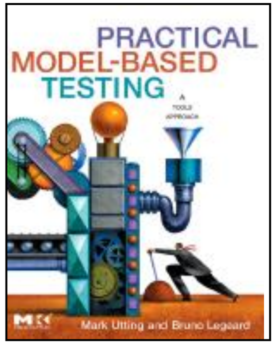
#	*Sequence*
1	init(2) **-empty-** delete() **-final**
2	init(2) **-empty-** push(a) **-filled-** pop() **-empty**
3	init(3) **-empty-** push(a) **-filled-** push(b) **-filled**
4	init(2) **-empty-** push(a) **-filled-** push(b) **-full-** pop() **-filled**
5	init(3) **-empty-** push(a) **-filled-** *push(b)* ***-filled-*** pop() **-filled**
6	init(2) **-empty-** push(a) **-filled-** push(b) **-full-** push(x)
7	init(2) **-empty-** pop()

ModelJUnit



- Supports online model-based testing, with models written as Java classes
- Test generation executed from
  - **JUnit** test: to integrate in build process, ...
  - **ModelJUnit GUI:** demonstration, exploring the model, ...

- References
  - https://sourceforge.net/projects/modeljunit
  - Utting, Mark, and Bruno Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.
  - Utting, Mark. "How to design extended finite state machine test models in Java." *Model-Based Testing for Embedded Systems* (2012): 147-169.



ModelJUnit: Model Elements

`Object getState()`: This method returns the current visible state of the EFSM. So this method defines an *abstraction function* that maps the internal state of the EFSM to the visible states of the EFSM graph. Typically, the result is a string, but it is possible to return any type of object.[9]

`void reset(boolean)`: This method resets the EFSM to its initial state. When online testing is being used, it should also reset the SUT or create a new instance of the SUT class. The boolean parameter can be ignored for most unit testing applications.[10]

`@Action void` *name*$_i$`()`: The EFSM must define several of these *action* methods, each marked with an `@Action` annotation. These action methods define the transitions of the EFSM. They can change the current state of the EFSM, and when online testing is being used, they also send test inputs to the SUT and check the correctness of its responses.

`boolean` *name*$_i$`Guard()`: Each action method can optionally have a *guard*, which is a boolean method with the same name as the action method but with "`Guard`" added to the end of the name. When the guard returns true, then the action is enabled (so may be called), and when the guard returns false, the action is disabled (so will not be called). Any action method that does not have a corresponding guard method is considered to have an implicit guard that is always true.

Example: Stack Model

```
public class StackModel implements FsmModel {
    private static final int CAPACITY = 2;
    private int size;

    @Override public Object getState() {
        if (size == 0) return "EMPTY";
        if (size == CAPACITY) return "FULL";
        return "FILLED";
    }

    @Action public void push() {  size++; }
    public boolean pushGuard() { return size < CAPACITY; }

    @Action public void pop() { size--; }
    public boolean popGuard() { return size > 0; }
```

model

state variable

state abstration function

actions

guard function

StackModelWithAdapter

```
public class StackModelWithAdapterDemo implements FsmModel {
    private static int CAPACITY = 3;
    private int size = 0;
    private Stack<String> stack = new Stack<String>(CAPACITY);

    public boolean pushGuard() { return size < CAPACITY; }
    @Action public void push() { stack.push("test #" + size); size++; }

    public boolean popGuard() { return size > 0; }
    @Action public void pop() {
        String data = stack.pop();
        size--;
        assertEquals("test #" + size, data);
    }

    public Object getState() {
        if (size == 0) return "EMPTY";
        if (size == CAPACITY) return "FULL";
        return "FILLED";
    }

    public void reset(boolean testing) {
        stack = new Stack<String>(CAPACITY); size = 0;
    }
```

system under test

call to SUT

assert result

Run StackModel with JUnit

```java
public class StackModelTest {

    @Test
    public void test() {
        Tester tester = new RandomTester(new StackModelWithAdapter());
        tester.buildGraph();
        CoverageMetric coverage = new StateCoverage();
        tester.addListener(coverage);
        tester.addListener(new VerboseListener());
        tester.addListener(new StopOnFailureListener());
        tester.generate(100);

        tester.getModel().printMessage(coverage.getName() + ": "
                + coverage.toString());
    }
}
```

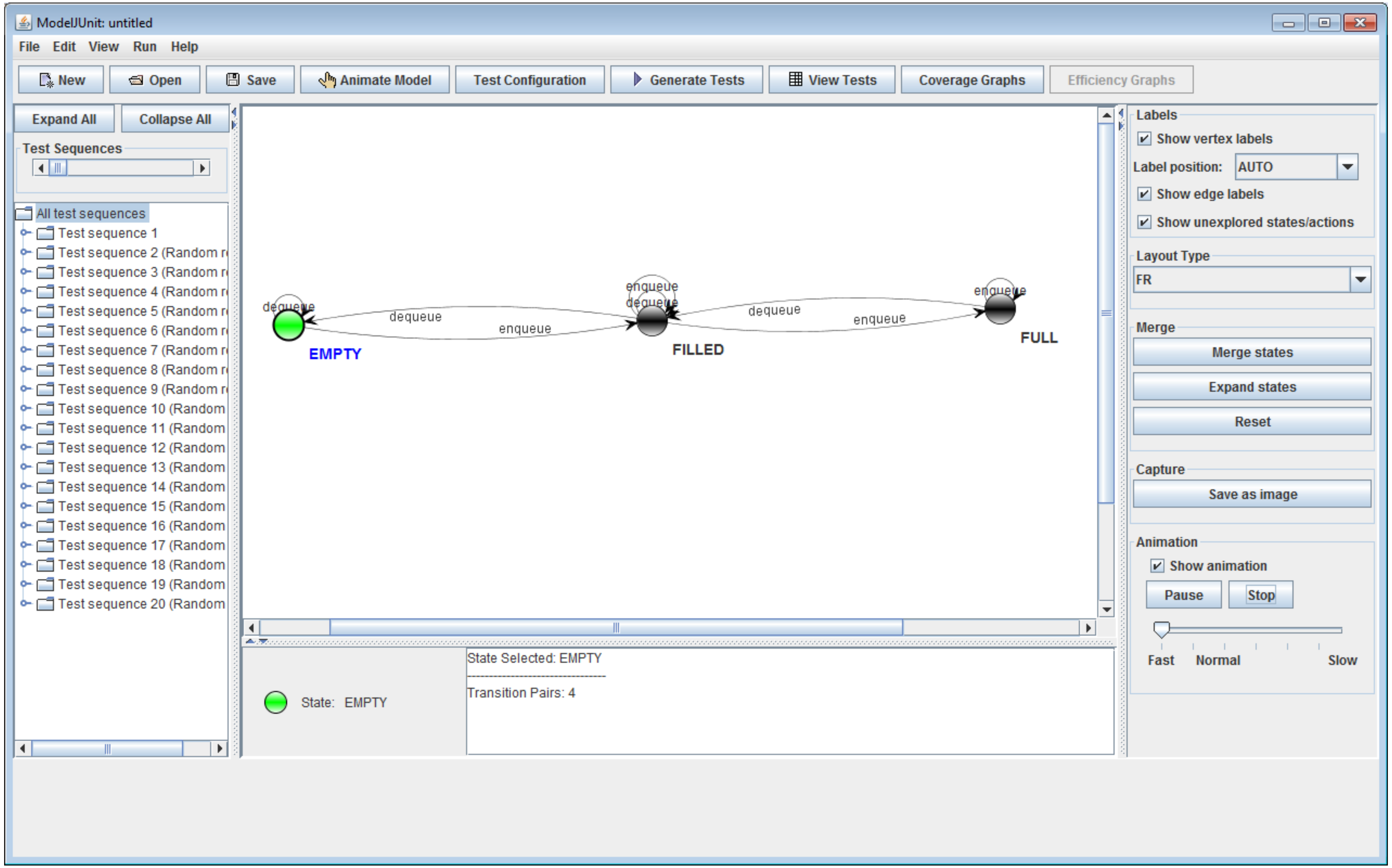
JUnit test

test sequence length

throws exception that will fail JUnit test run

ModelJUnit GUI

Run *nz.ac.waikato.modeljunit.gui.ModelJUnitGUI*



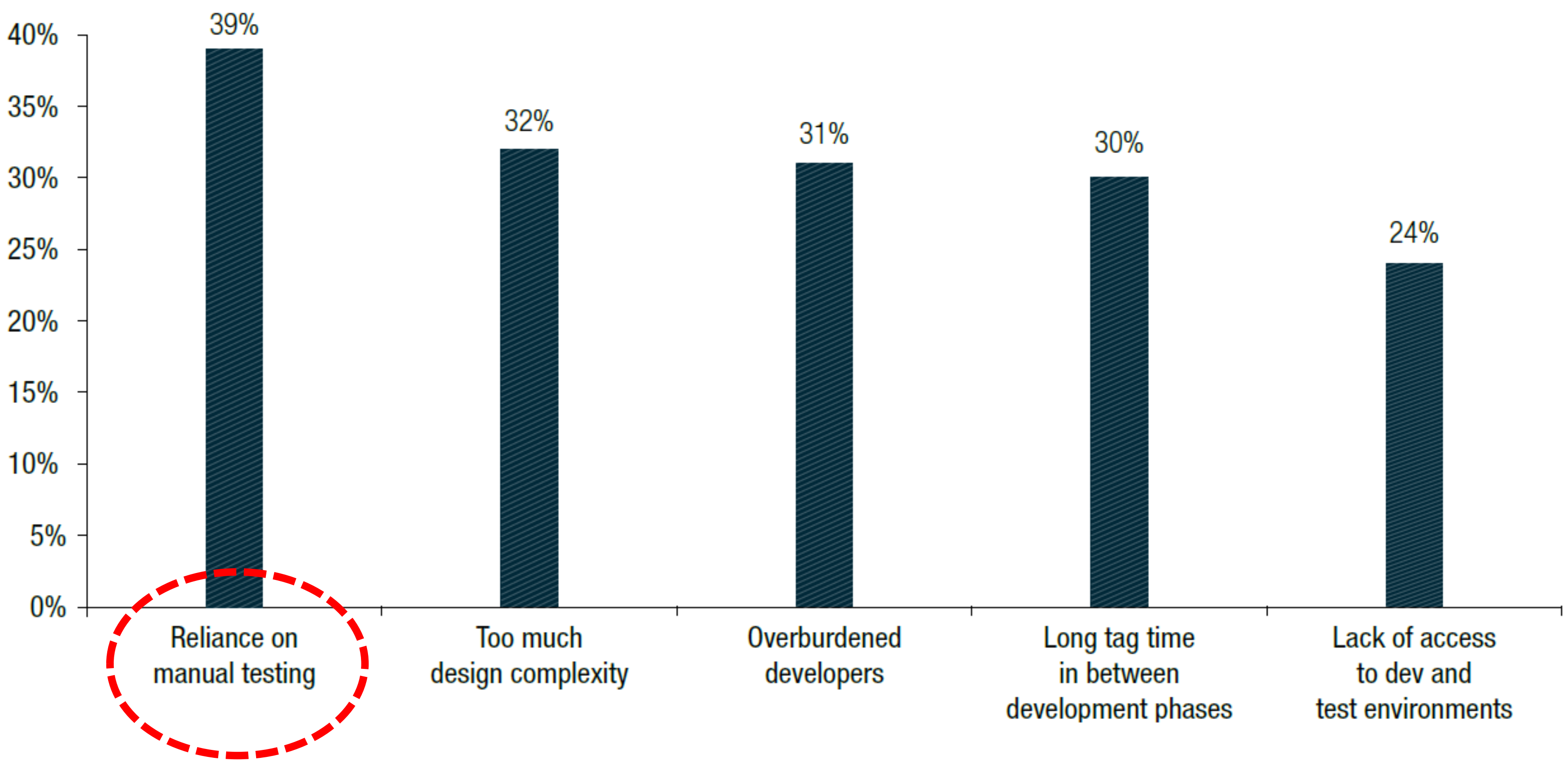
Overview

- Random testing
- Feedback-directed random testing
- Assertions as test oracle
- Model-based testing
- Model-based testing by example
- ModelJUnit
- Closing thoughts

World Quality Report

Top 5 technical challenges in Application development

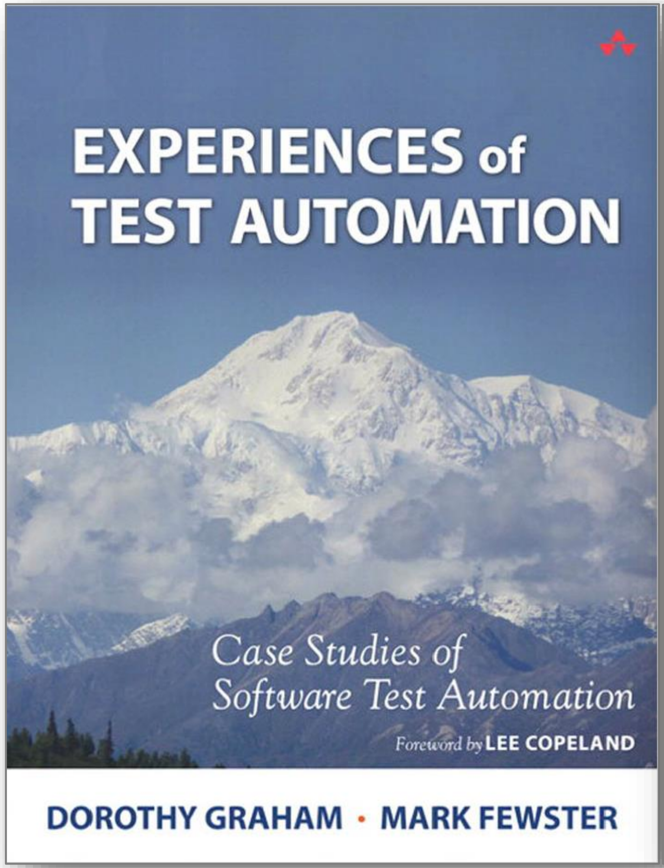
FIGURE 2

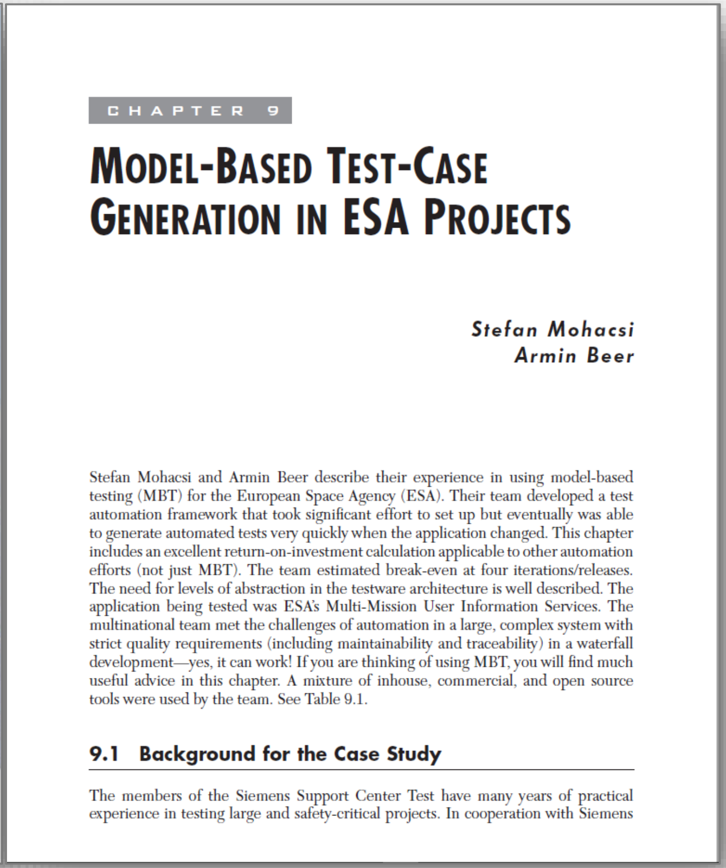


https://www.sogeti.com/explore/reports/world-quality-report-2015-2016/

Literature

Graham, D. and Fewster, M.: *Experiences of test automation: case studies of software test automation*. Addison-Wesley Professional, 2012





CHAPTER 9

MODEL-BASED TEST-CASE GENERATION IN ESA PROJECTS

Stefan Mohacsi
Armin Beer

Stefan Mohacsi and Armin Beer describe their experience in using model-based testing (MBT) for the European Space Agency (ESA). Their team developed a test automation framework that took significant effort to set up but eventually was able to generate automated tests very quickly when the application changed. This chapter includes an excellent return-on-investment calculation applicable to other automation efforts (not just MBT). The team estimated break-even at four iterations/releases. The need for levels of abstraction in the testware architecture is well described. The application being tested was ESA's Multi-Mission User Information Services. The multinational team met the challenges of automation in a large, complex system with strict quality requirements (including maintainability and traceability) in a waterfall development—yes, it can work! If you are thinking of using MBT, you will find much useful advice in this chapter. A mixture of inhouse, commercial, and open source tools were used by the team. See Table 9.1.

9.1 Background for the Case Study

The members of the Siemens Support Center Test have many years of practical experience in testing large and safety-critical projects. In cooperation with Siemens

References

- Binder R. V.: Testing Object-Oriented Systems: Models, Patterns and Tools. Addison-Wesley,1999
- Robinson H.: How to build your own robot army. StarWest, 2006
- Robinson H.: Intelligent Test Automation. STQE Magazine, Sept./Oct. 2000
- Utting, Mark, and Bruno Legeard. Practical model-based testing: a tools approach. Morgan Kaufmann, 2007.
- Utting, Mark. "How to design extended finite state machine test models in Java." Model-Based Testing for Embedded Systems (2012): 147-169.
- Utting M. et al.: A Taxonomy of Model-based Testing. Working paper series. University of Waikato, 2006