Testing without requirements?

REFSQ 2018, Utrecht, The Netherlands

Tanja E.J. Vos

Testing at the GUI Level

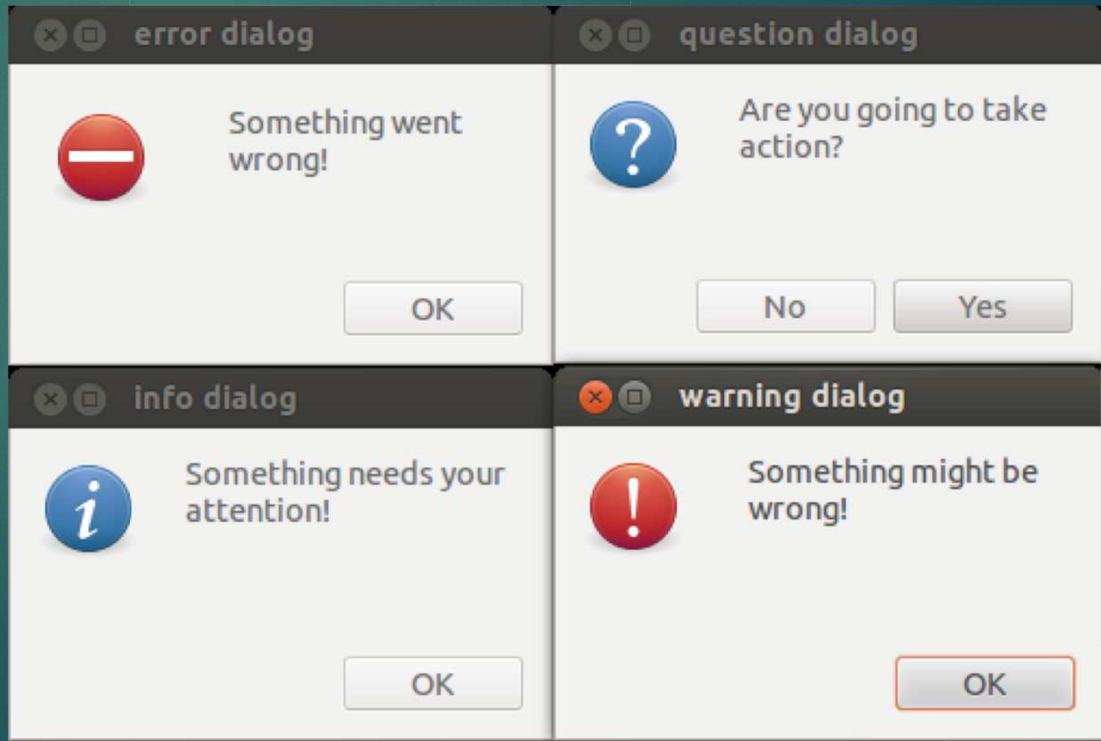
- GUI is where all functionality comes together
  - Interacts with the underlying code
  - The whole system can be executed by means of the GUI
  - Integration / System Testing

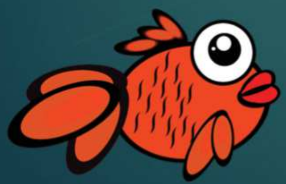
Testing at the GUI Level

- Most applications have GUIs
  - Computers, tablets, smartphones….
  - Even safety critical applications

Testing at the GUI Level

- Faults that arise at UI level are important
  - These are what your client finds
  - GUI tests from their perspective!

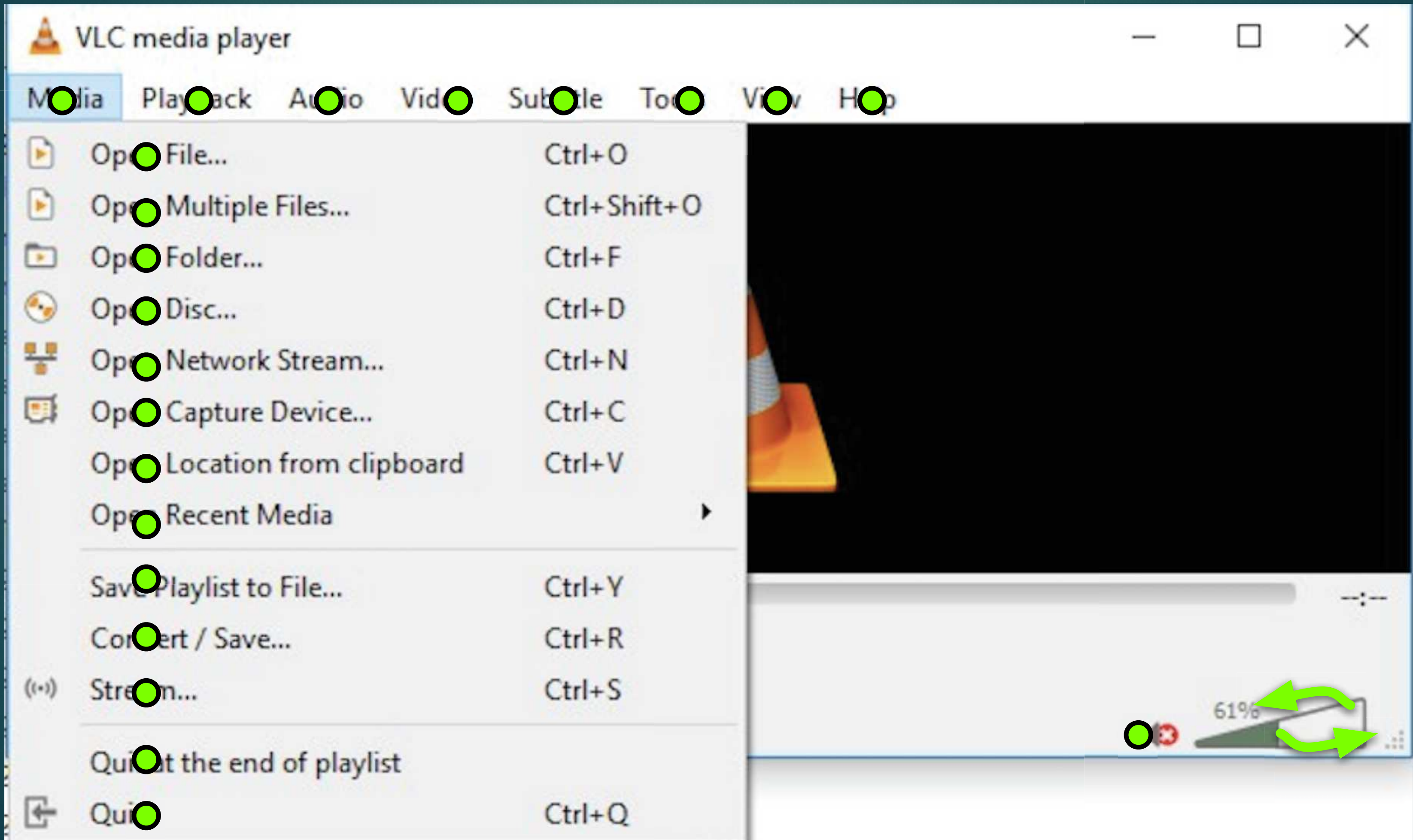




What is a GUI?

Contains graphical objects w, called **widgets**

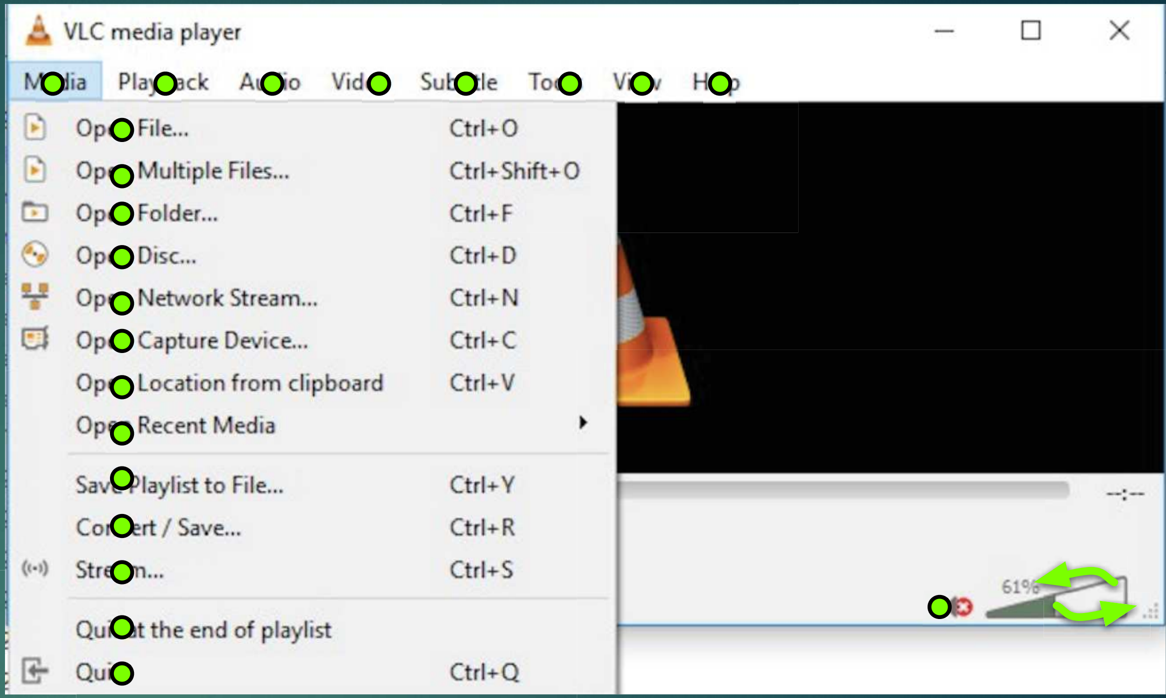
- Menus, textboxes, buttons, scrollbars

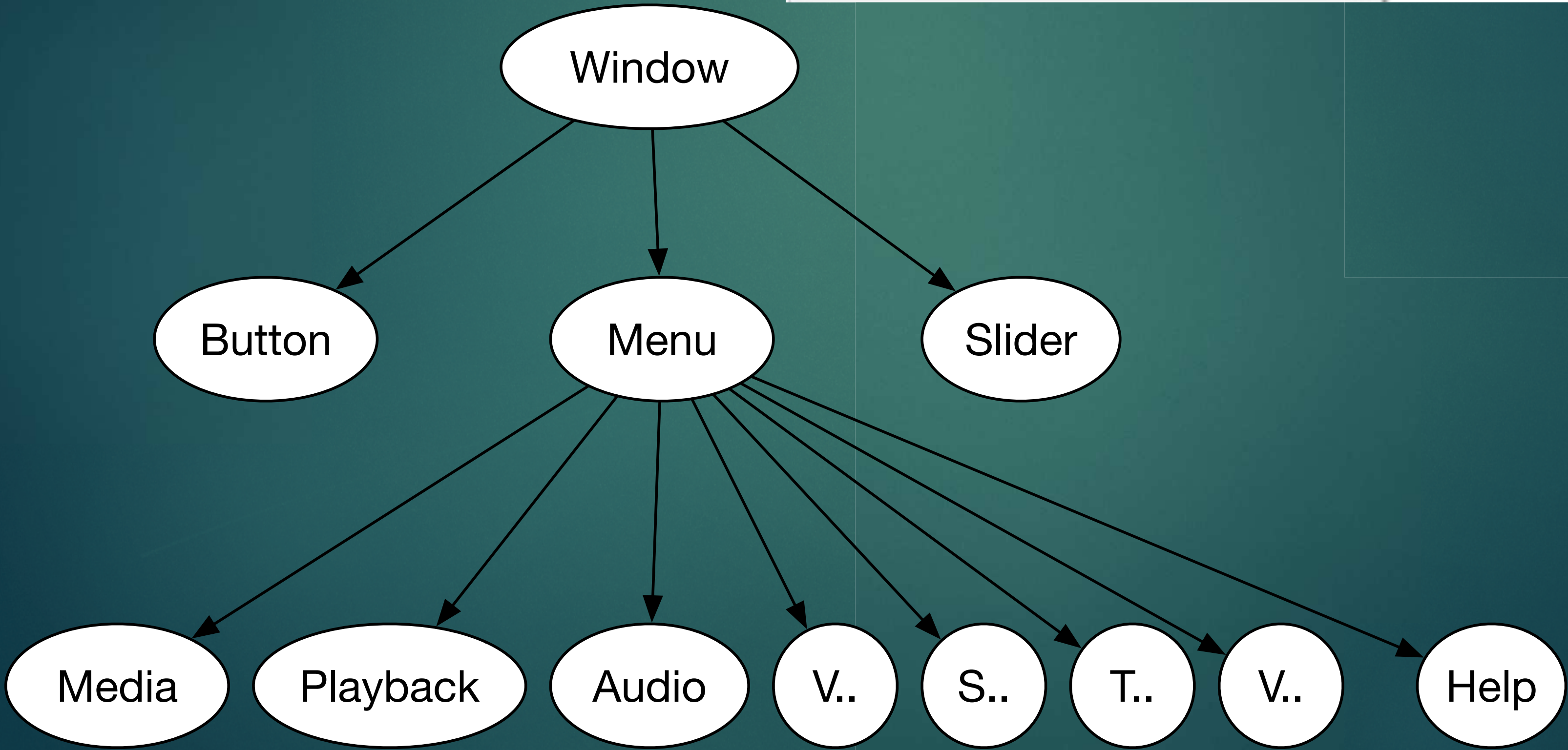


Widgets

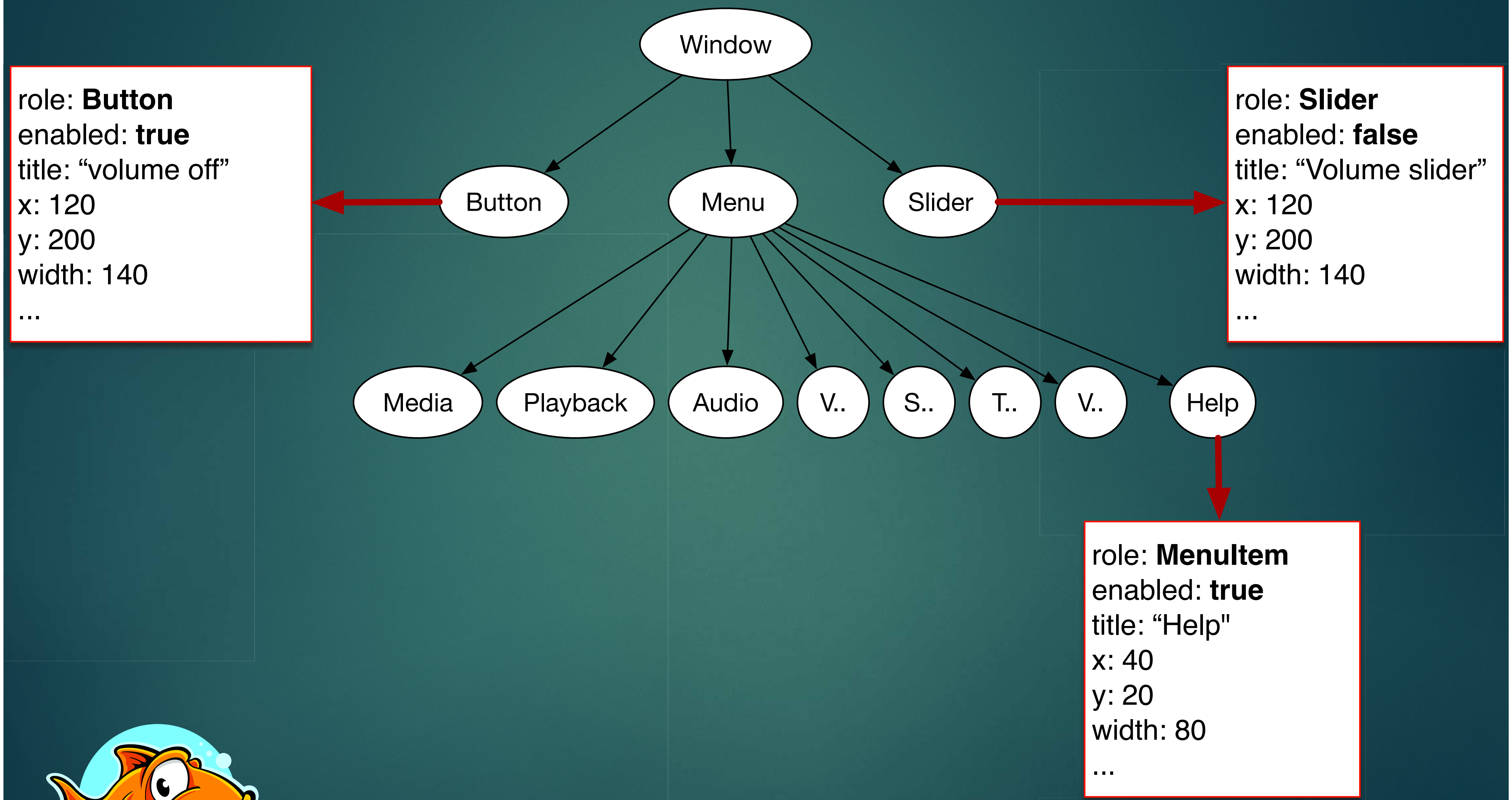
form a hierarchy

the **widget-tree**





Widgets have **properties** *p* which have values *v* at run-time.

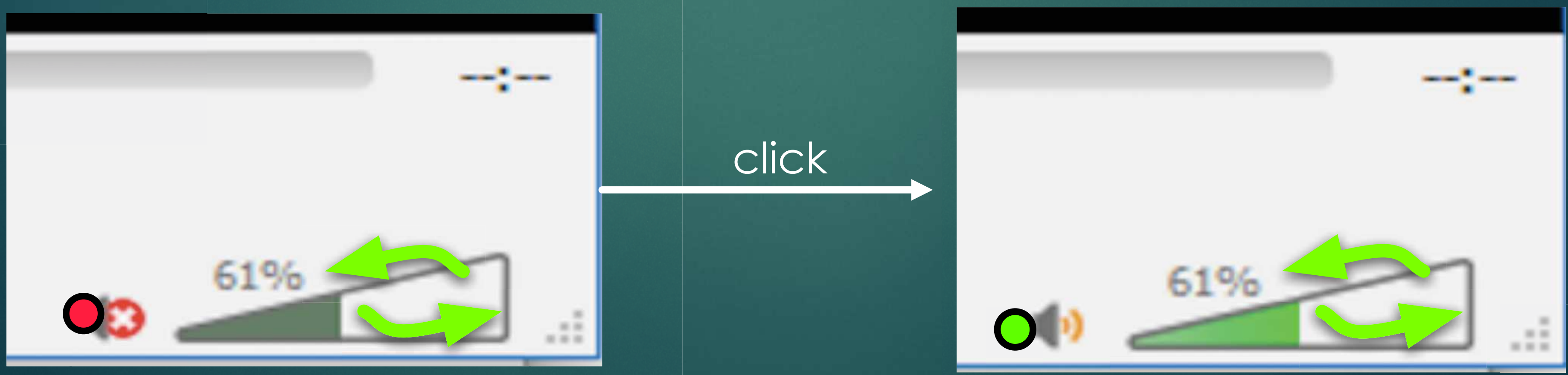


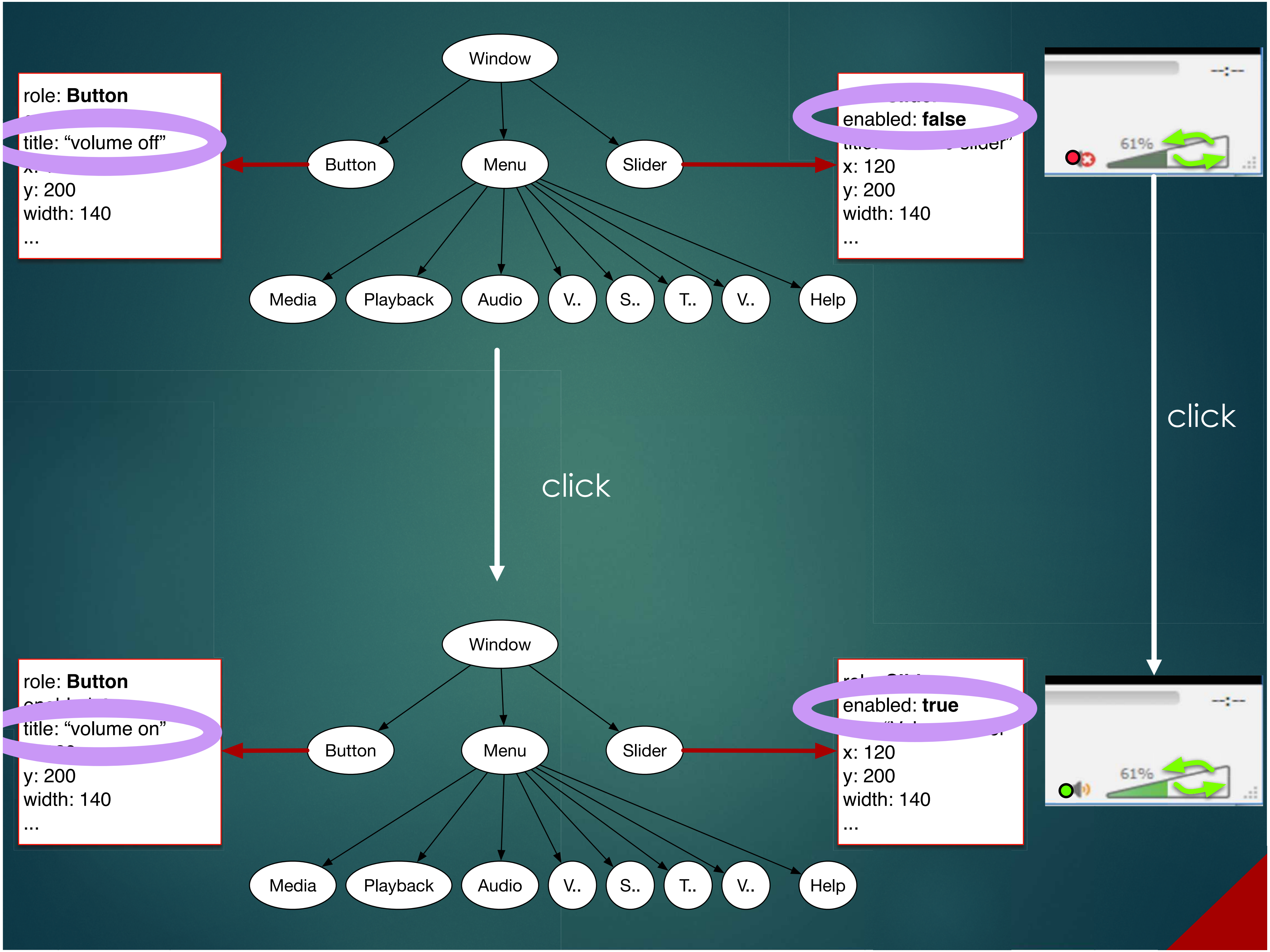
GUI state

- The widget tree
- + the values of the properties of each widget

GUI action

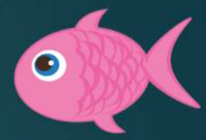
- Users can exercise actions (click, type, drag, drop,…)
- These cause a state change

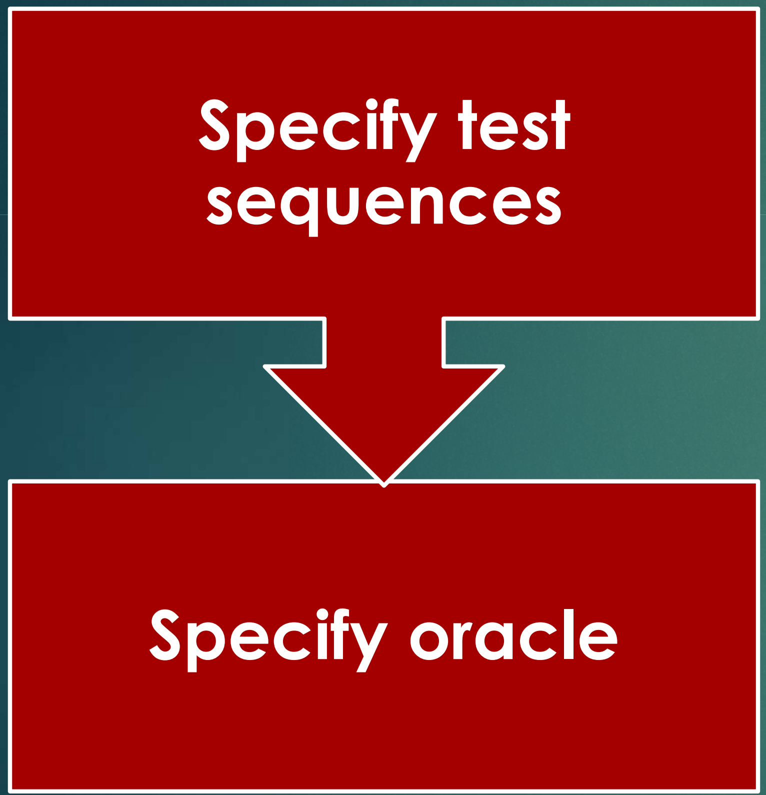




Window
Button
Menu
Slider
Media
Playback
Audio
V..
S..
T..
V..
Help
role: Button
title: "volume off"
y: 200
width: 140
...
enabled: false
x: 120
y: 200
width: 140
...
61%
click
click
Window
Button
Menu
Slider
Media
Playback
Audio
V..
S..
T..
V..
Help
role: Button
title: "volume on"
y: 200
width: 140
...
enabled: true
x: 120
y: 200
width: 140
...
61%

What is GUI testing

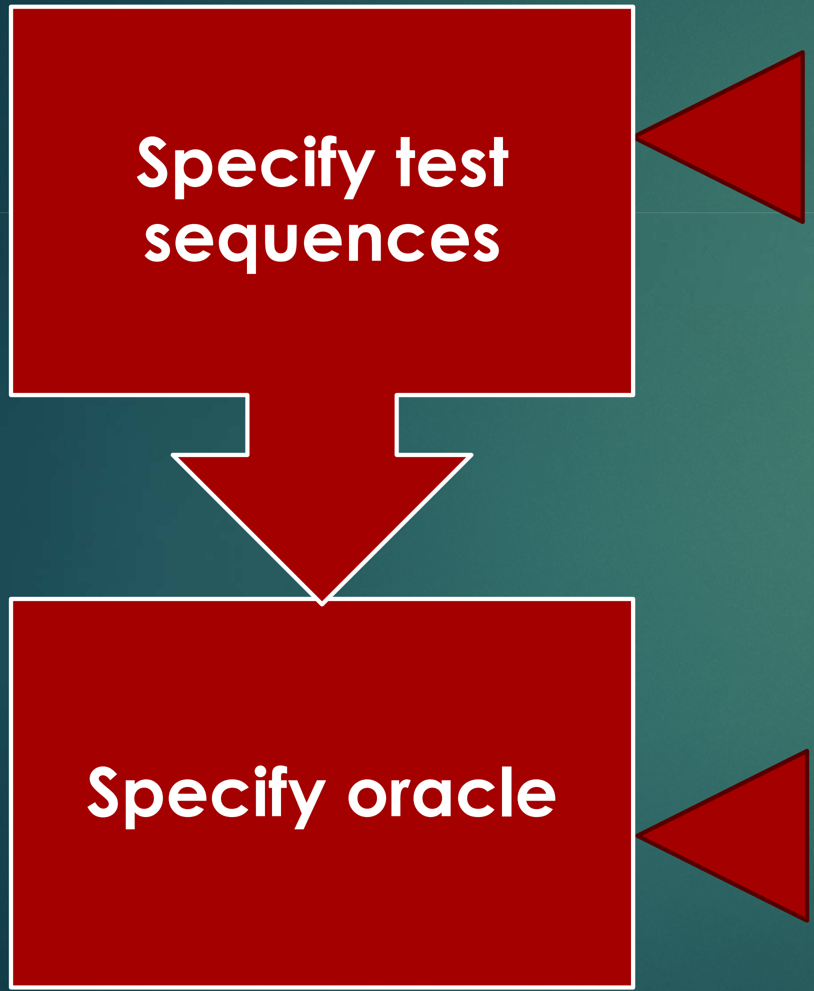




- Sequences of GUI actions
  - Click, drag, drop, type
  - Provide inputs where needed (e.g., filling text fields)
- The test oracle
  - The correct states after execution of each action

Together they test a requirement

What is GUI testing



Step 1
Open MS Word

Step2
Click on menu View

Step 3
Click on Media Browser

Step 4
Select a picture and drag into the document

After each step:
- No failure has occurred
- No error message has popped-up

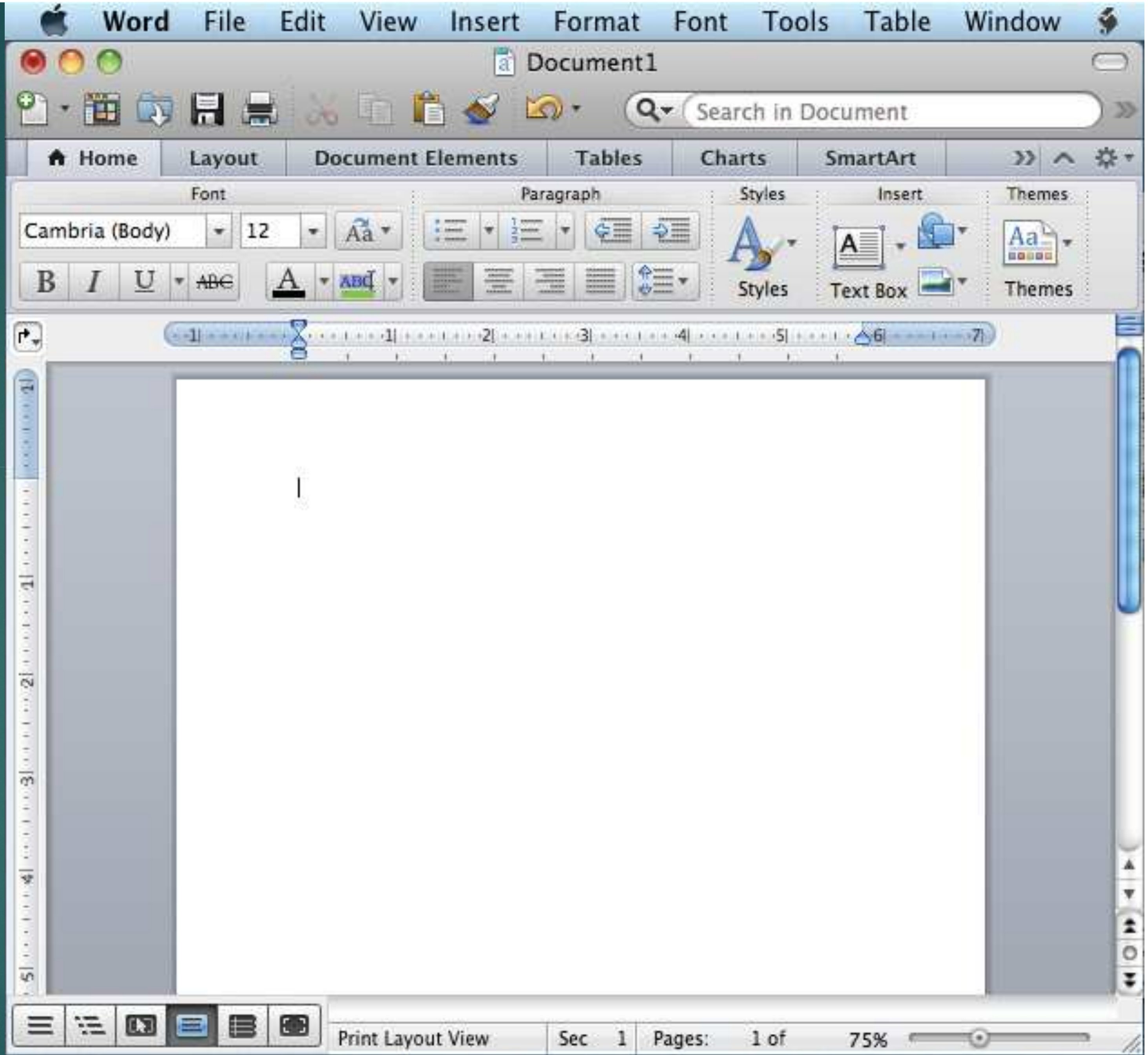
After last step:
- The picture is in the doc

Step 1
Open MS Word

Step2
Click on menu View

Step 3
Click on Media Browser

Step 4
Select a picture and drag into the document

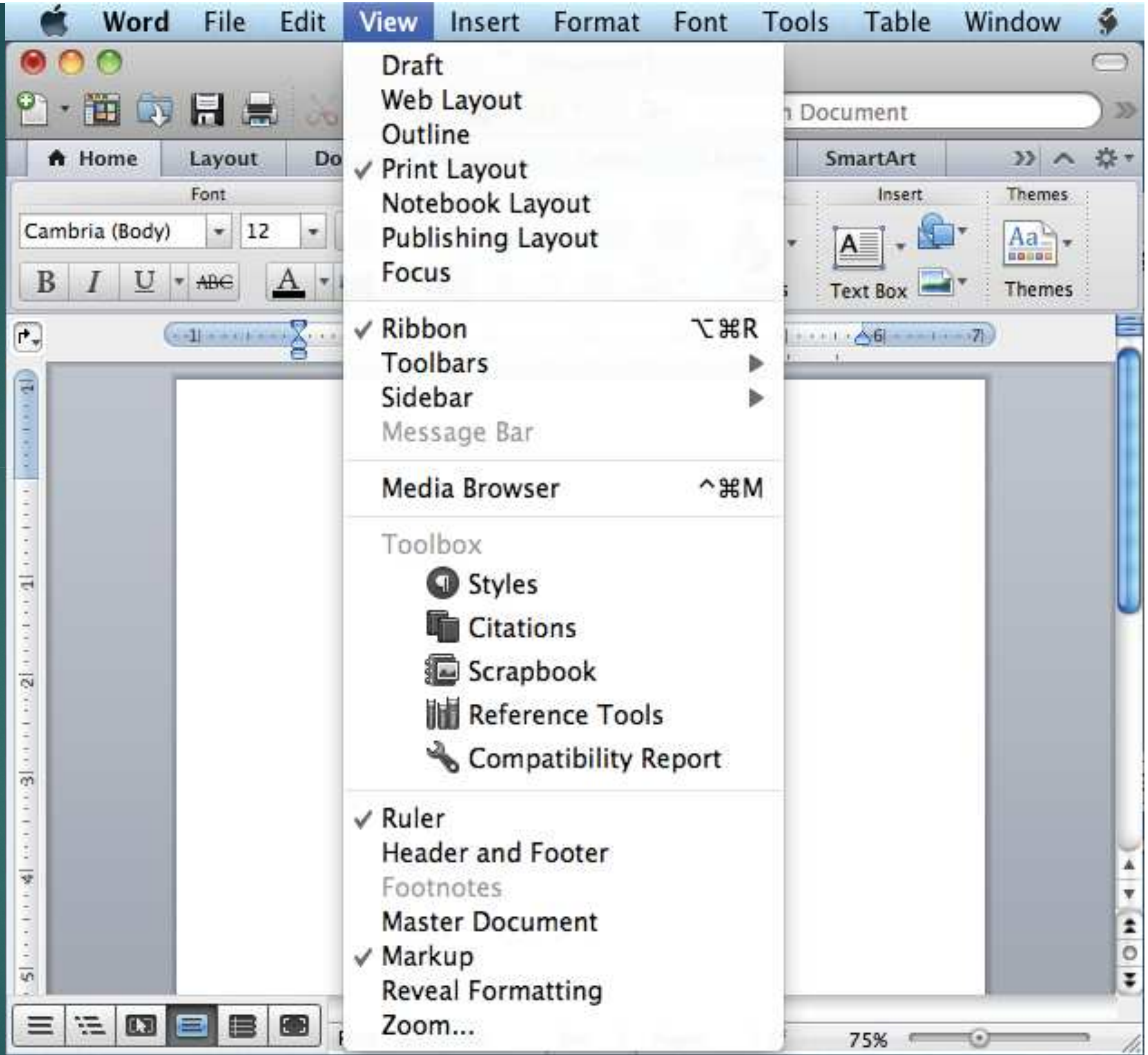


Step 1
Open MS Word

Step2
Click on menu View

Step 3
Click on Media Browser

Step 4
Select a picture and drag into the document

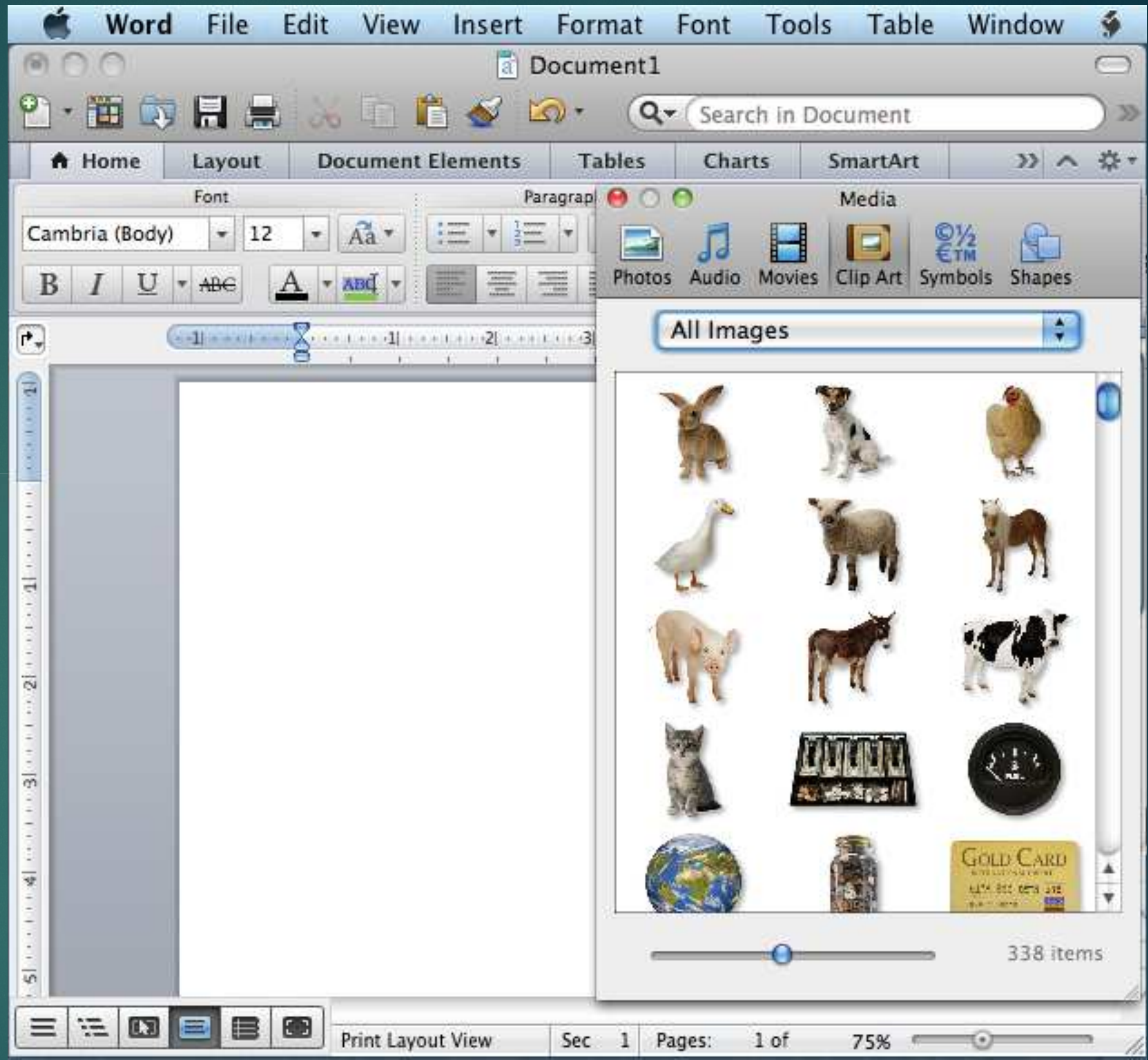


Step 1
Open MS Word

Step2
Click on menu View

Step 3 
Click on Media Browser

Step 4
Select a picture and drag into the document

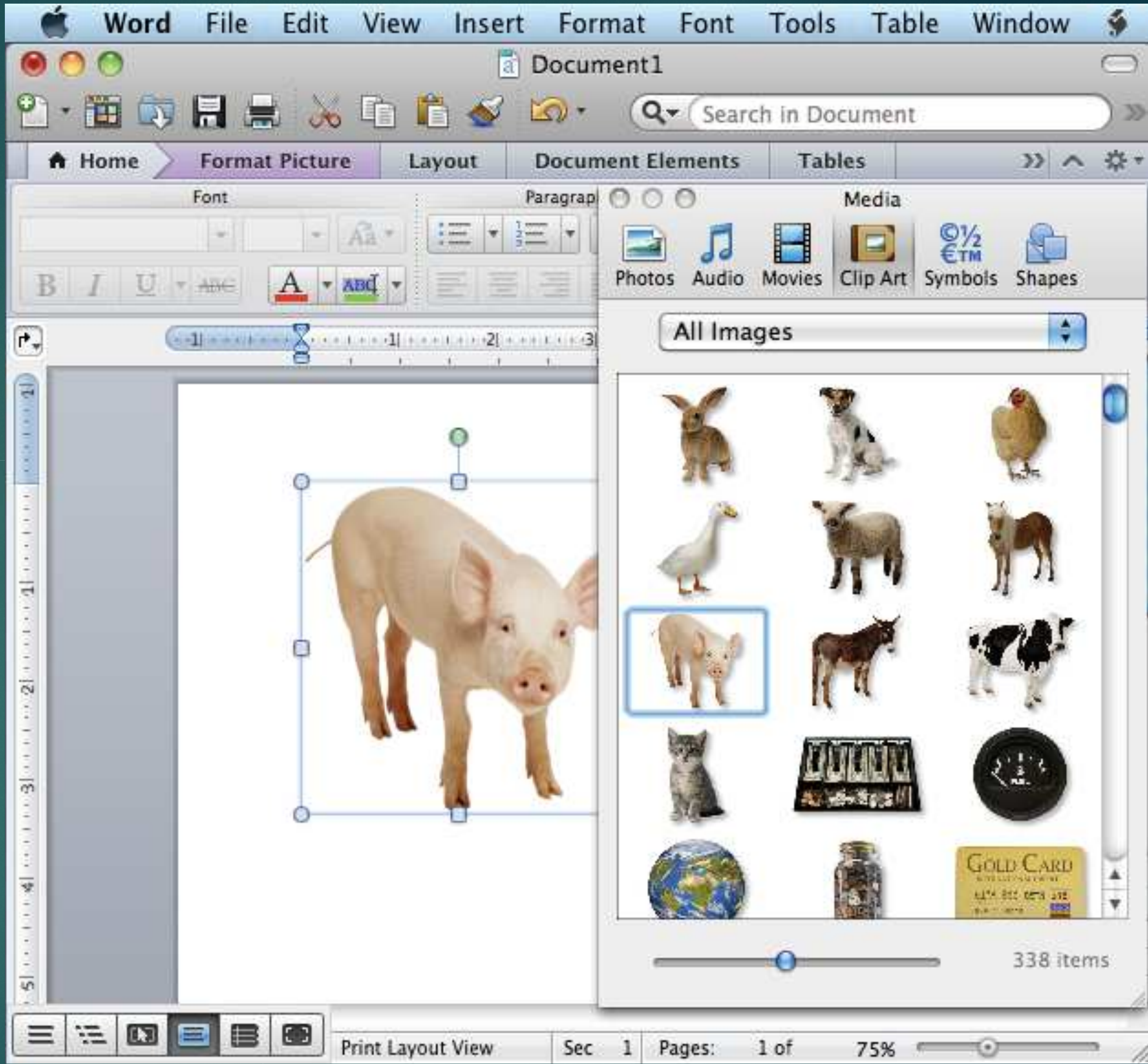


Step 1
Open MS Word

Step2
Click on menu View

Step 3
Click on Media Browser

Step 4 
Select a picture and drag into the document

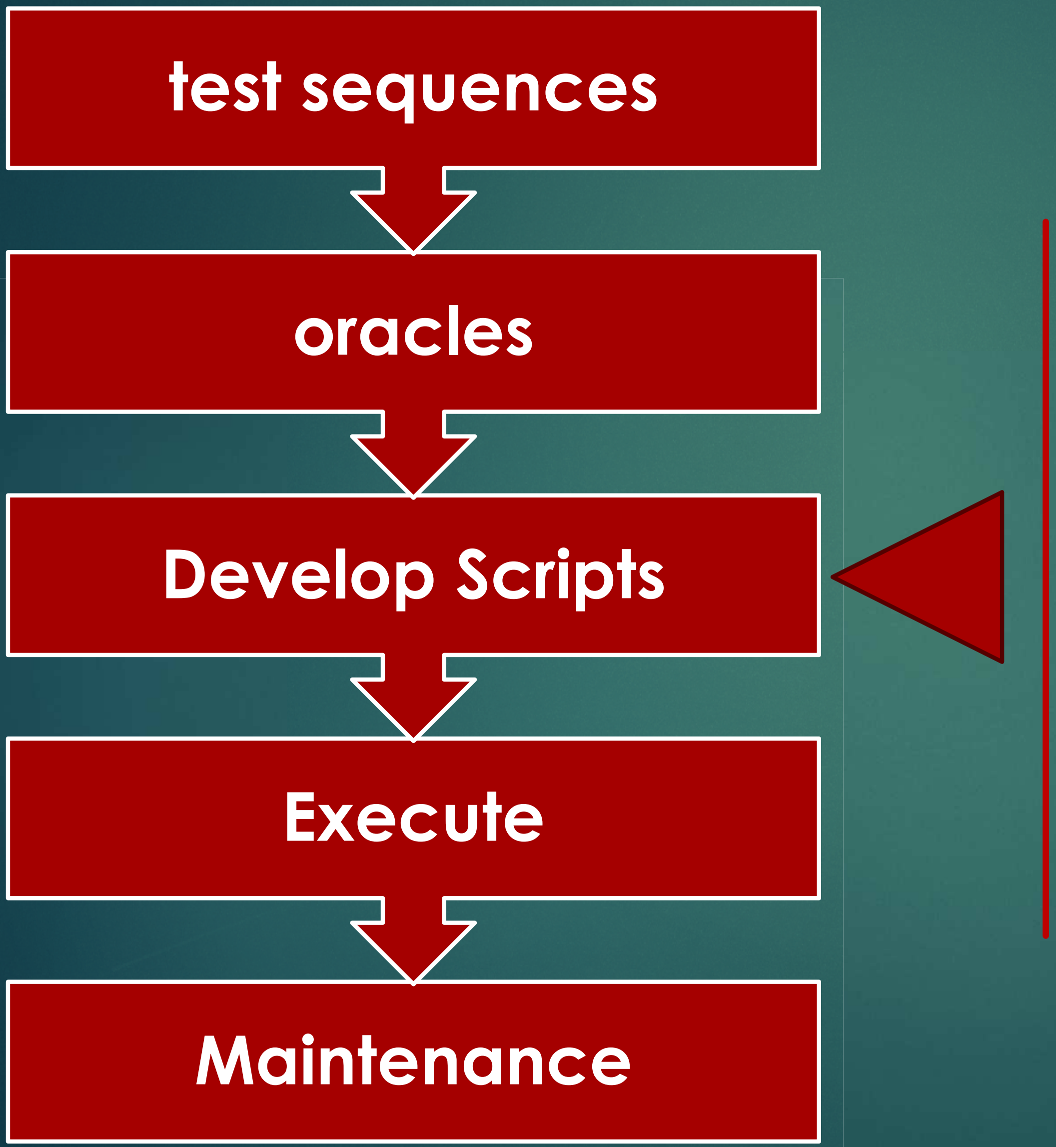


Manual testing

- Tedious
  - Executing the same clicks over and over again
- Tiresome and boring
  - Rerunning the same tests after changes to the SUT
  - Filling the same forms over and over again
  - Regression testing
- Error prone
- Costly


Need to automate

State of practice: make scripts



Capture & Replay

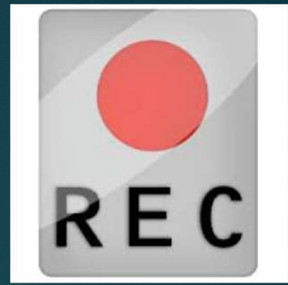
Visual testing

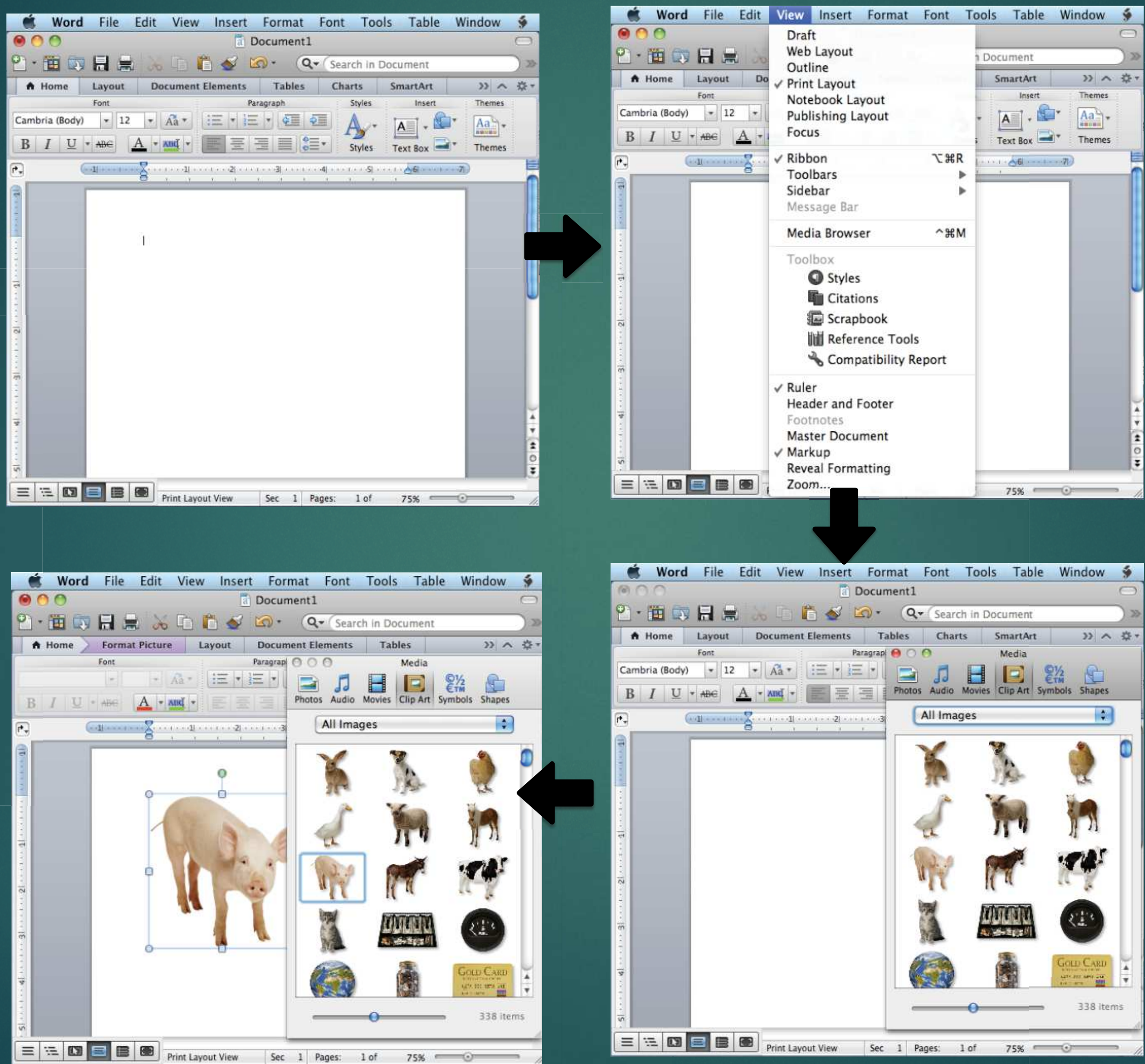
Capture & Replay

- Tools **Captures** user interaction with the UI
- **Records** a script
- That can be automatically **Replayed**

- Examples
  - Open source
    - Selenium
    - Abbot
    - ….
  - Commercial
    - QF-Test
    - Rational Functional /Robot Tester (IBM)
    - ….

Capture & Replay



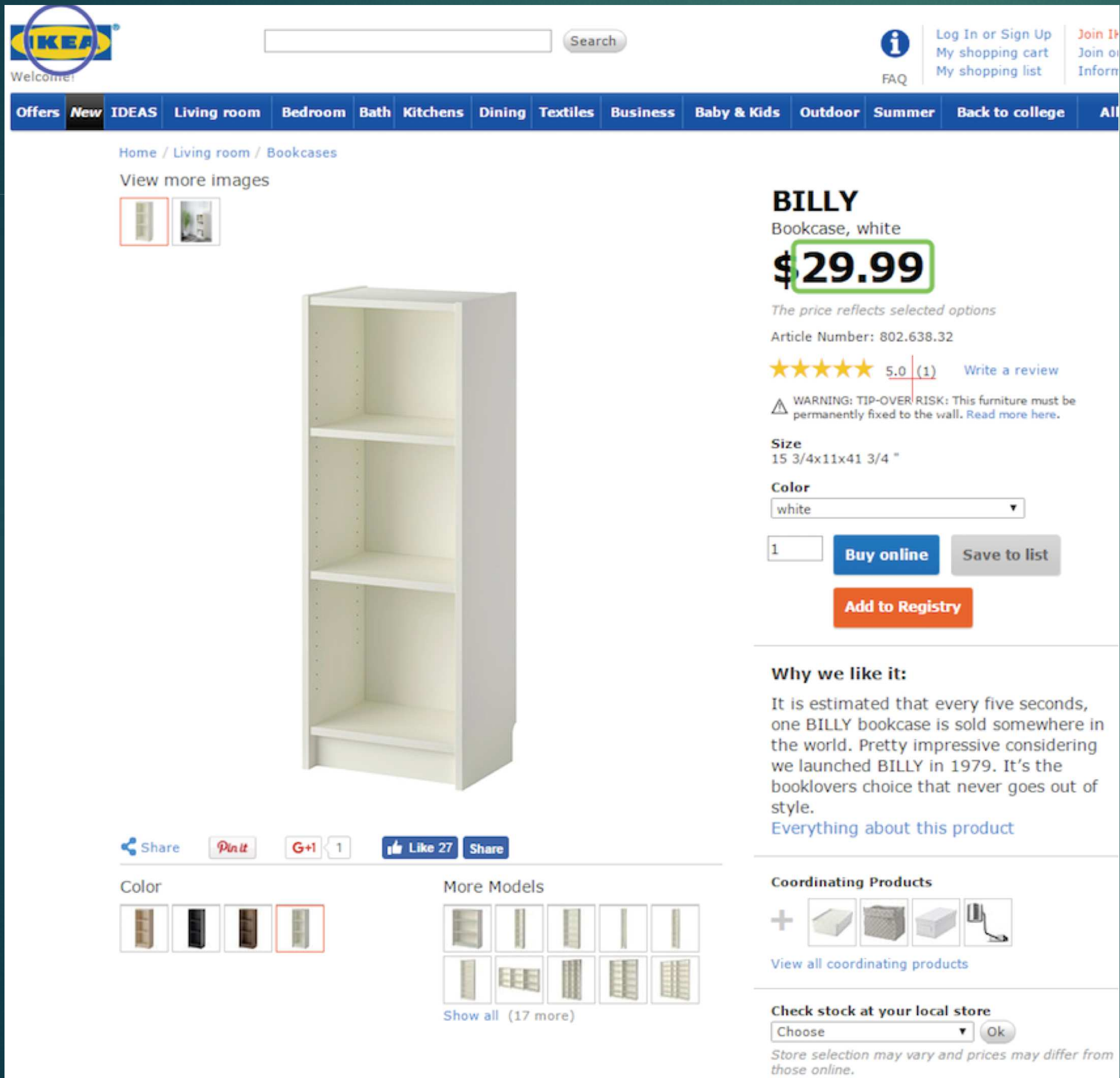


Capture & Replay

- Advantages
  - Simple and easy
- Disadvantages
  - Scripts break as GUI changes
  - Maintenance problem
- These are huge problems
  - GUIs change all the time
  - Requirements too!

Visual testing (VGT)

- Based on image recognition



Begin "Search for a Billy bookcase"

StartWeb "http://www.ikea.com/us/en"

Click 

Write "billy[ENTER]"

End

Begin "View the details"

Click 

End

Begin "Check the price"

Check 

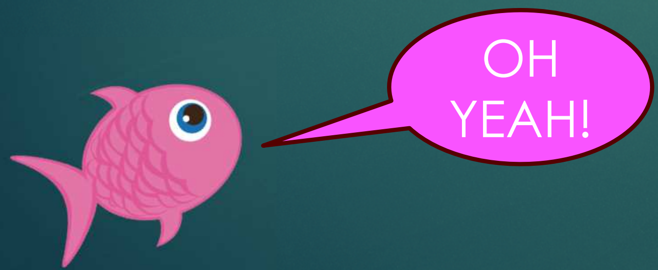
End

Visual testing

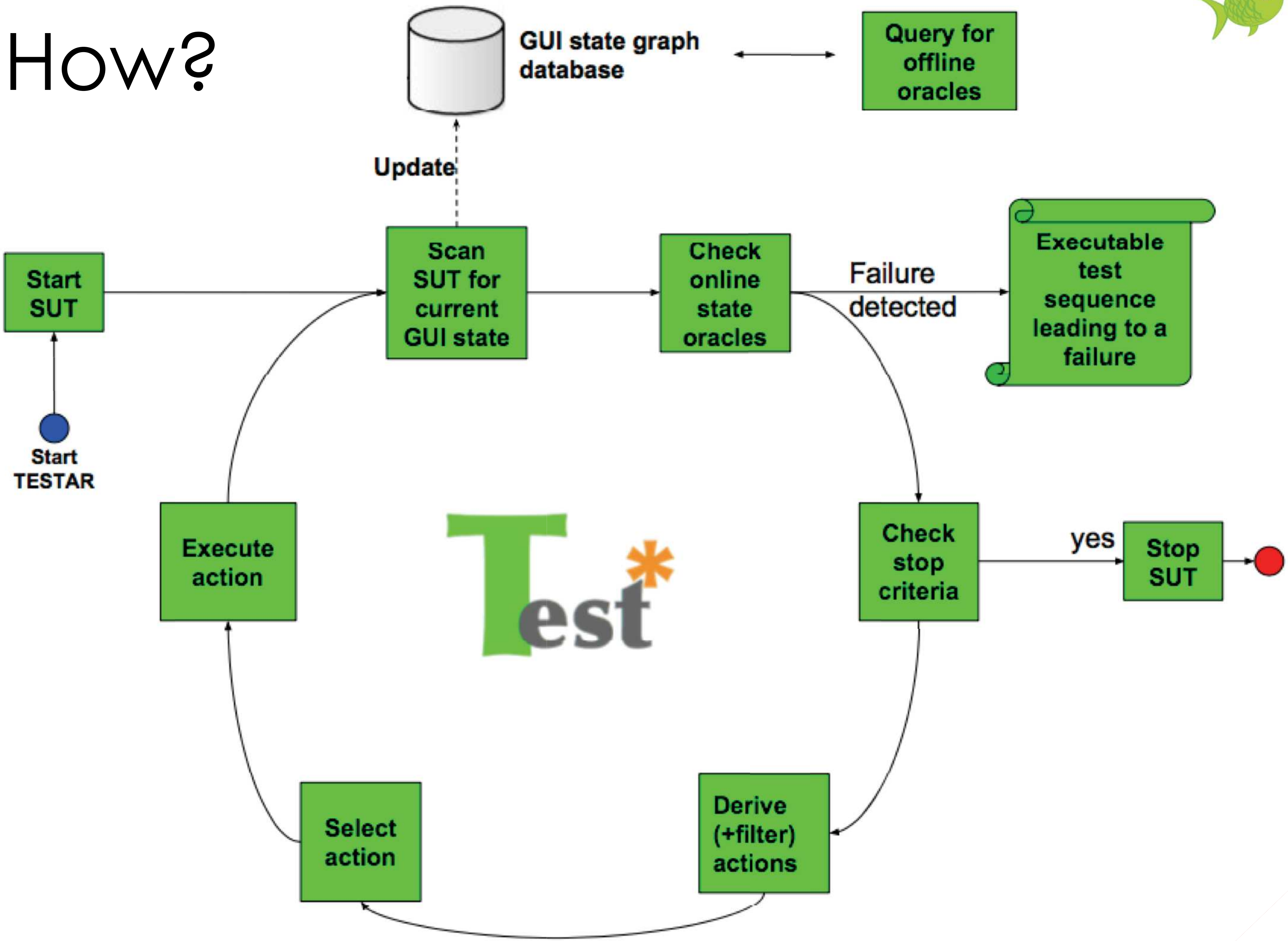
- Easy to understand
- Hardly no programming skills needed
- Solves part of maintenance problem
  - Robust against some changes
  - But not all
    - Move Media Browser within same menu: YES
    - Move Media Browser to another menu: NO
    - Change the icon: NO
- Studies show maintenance still an issue

Our contribution: Test*

- **Scriptless**
  - What is not there does not need to be maintained
- **Departs from random testing**
  - Immediately start testing without requirements

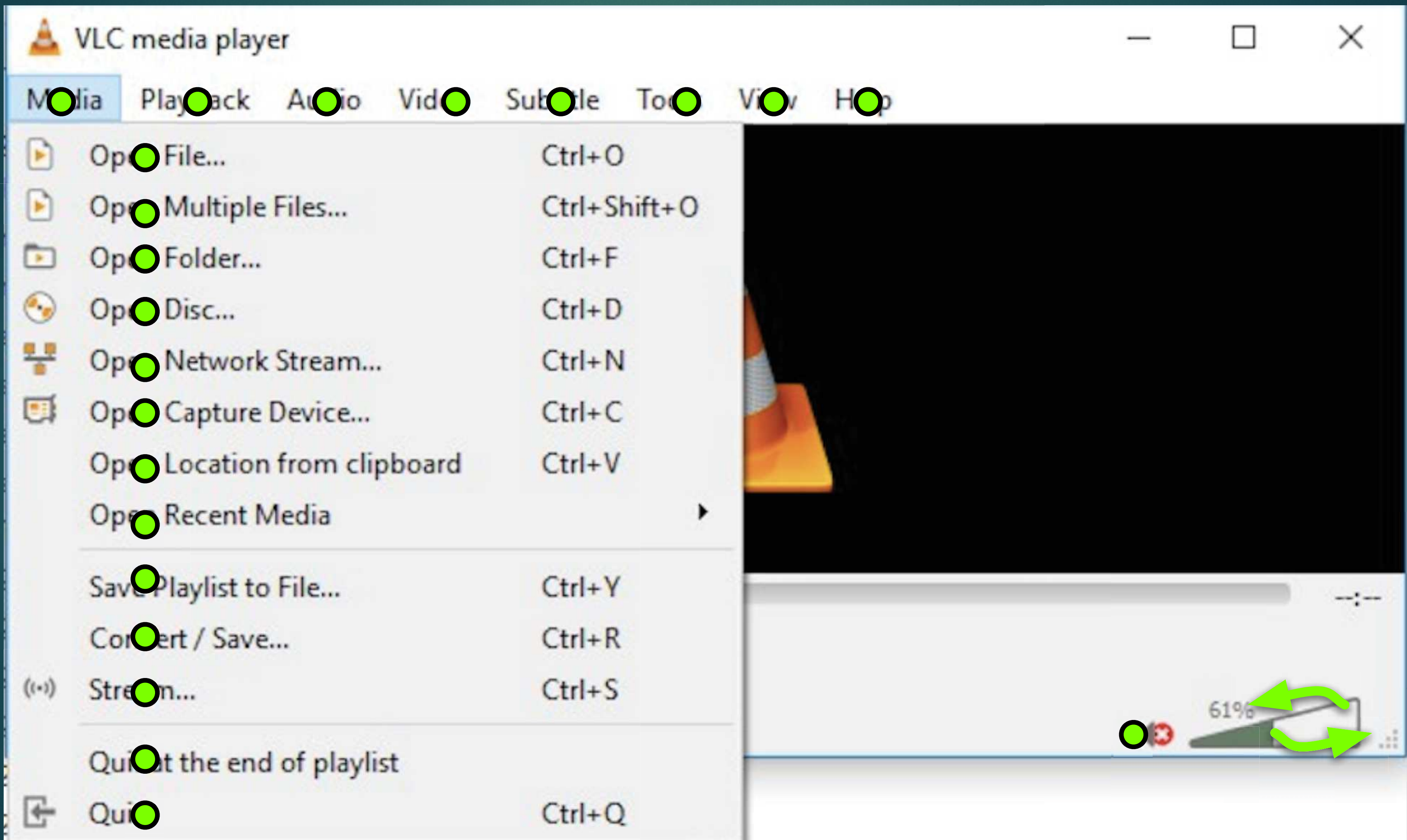


How?

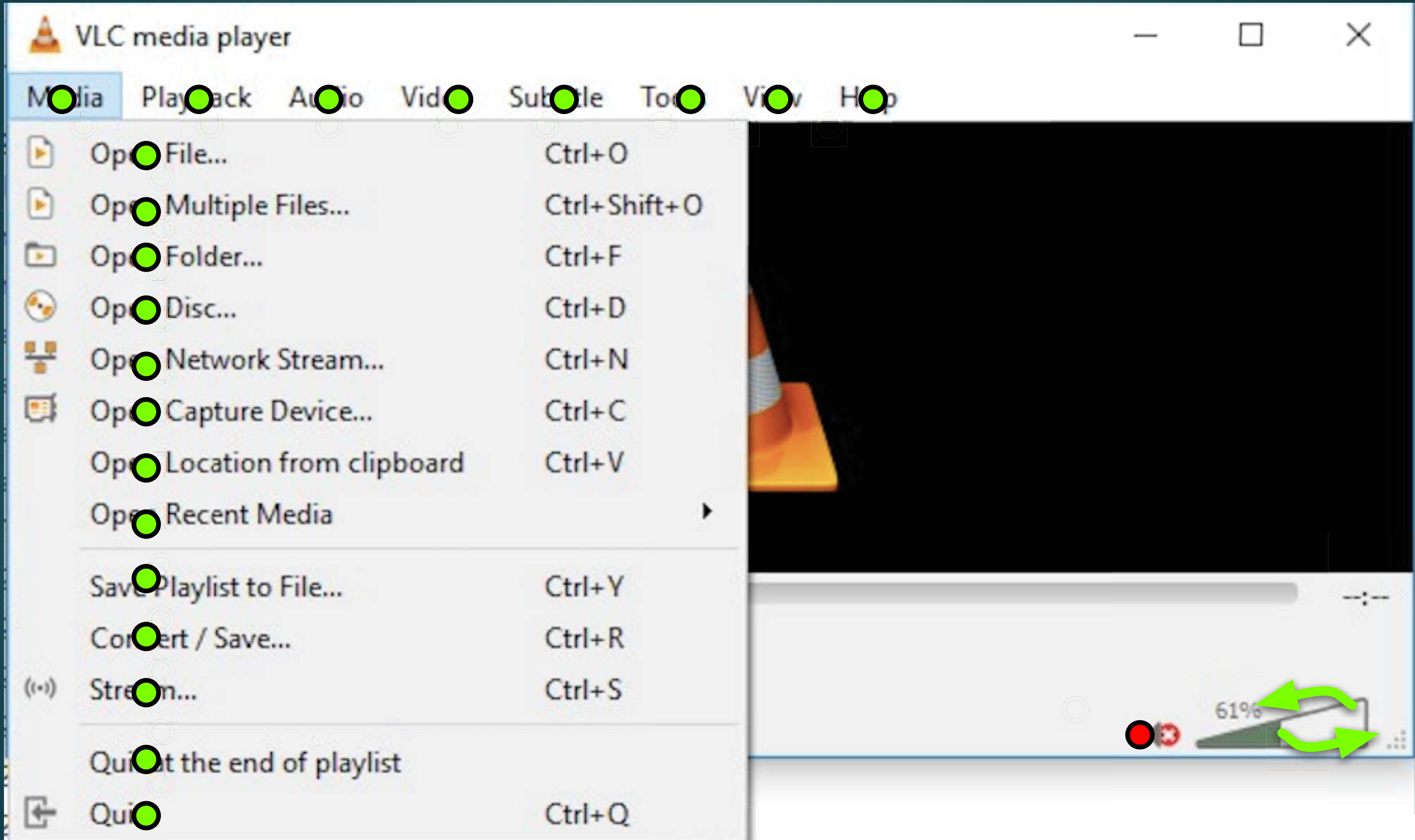


GUI state graph database
Query for offline oracles
Update
Start SUT
Scan SUT for current GUI state
Check online state oracles
Failure detected
Executable test sequence leading to a failure
Start TESTAR
Execute action
Check stop criteria
yes
Stop SUT
Select action
Derive (+filter) actions

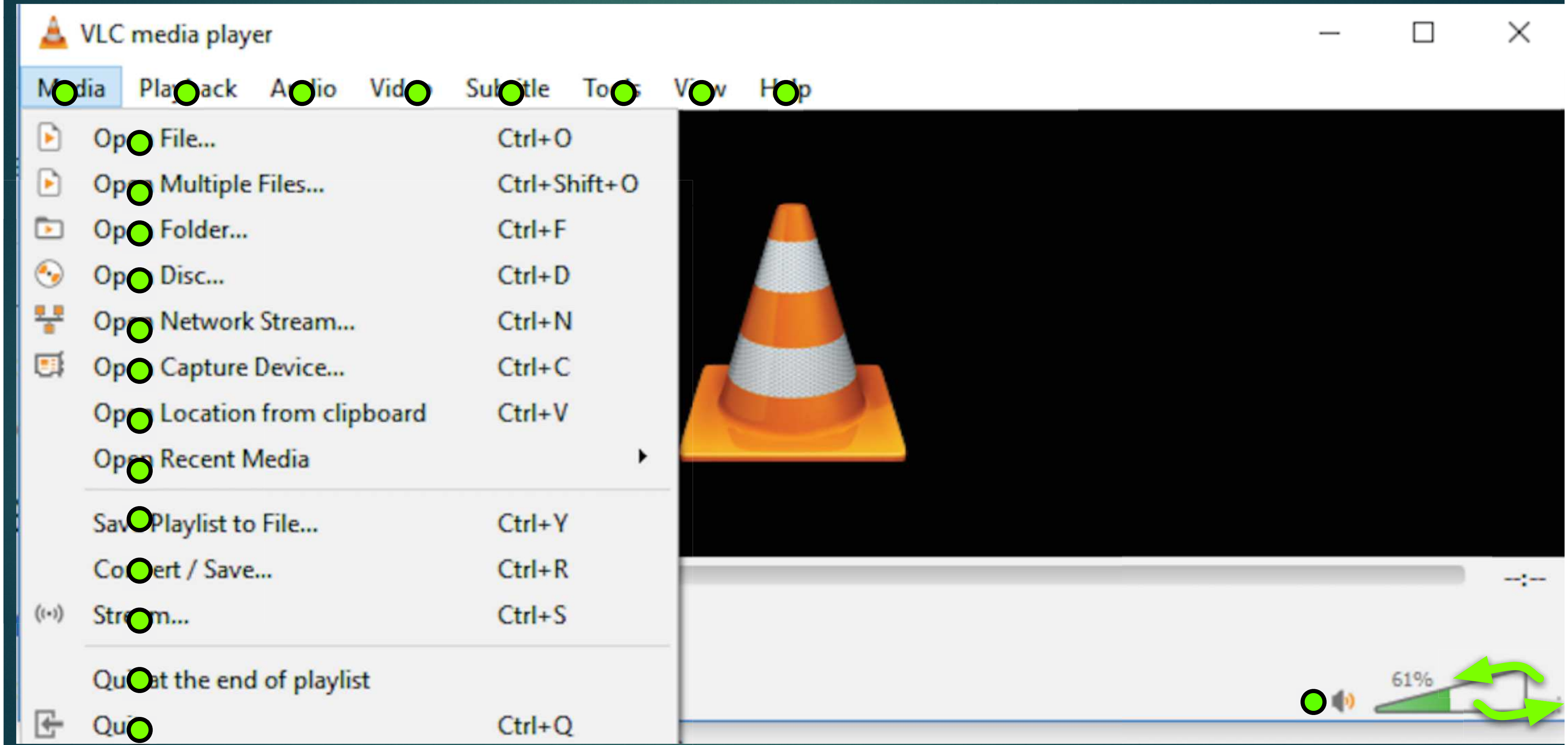
Current state and actions



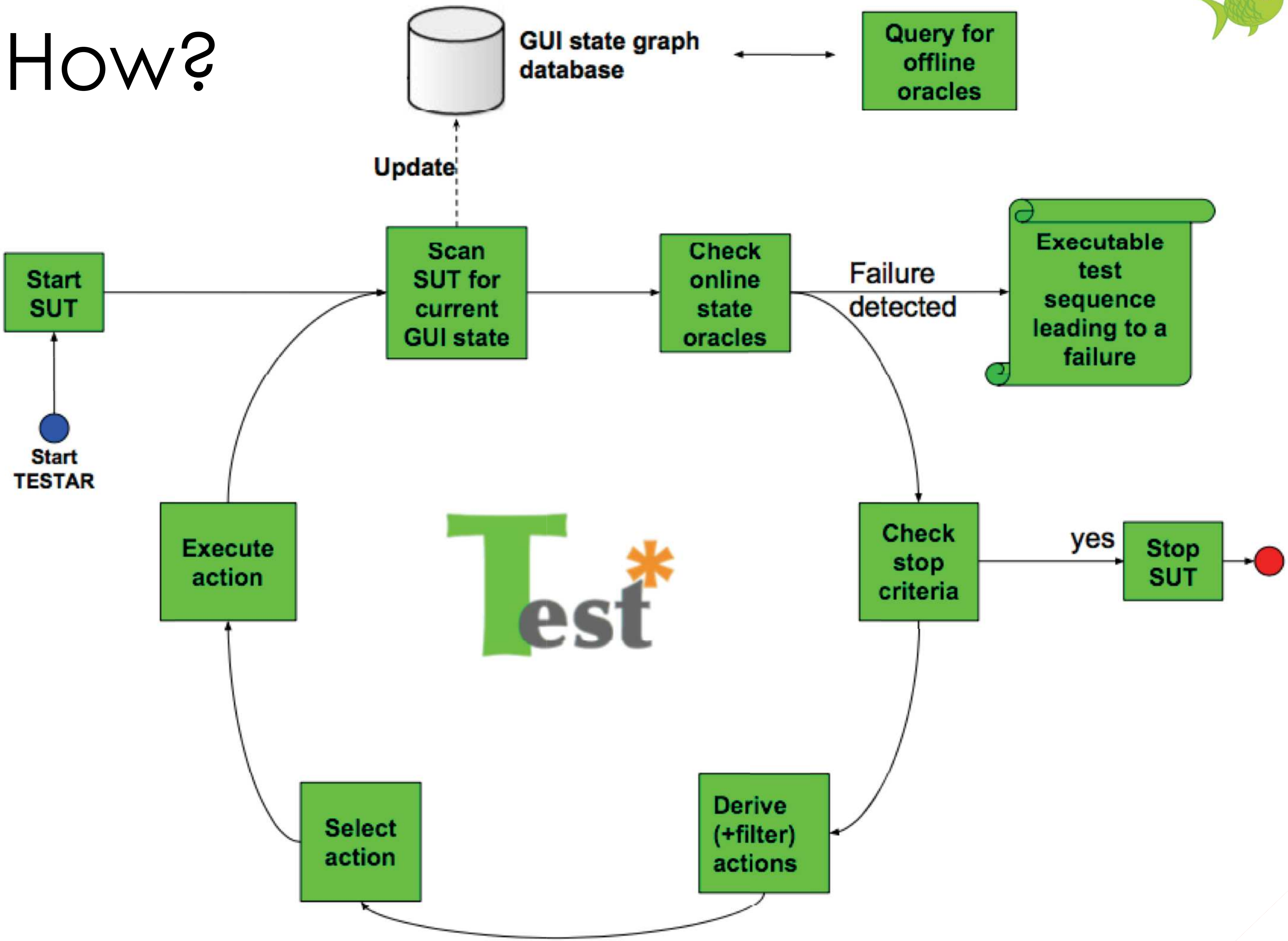
Select action



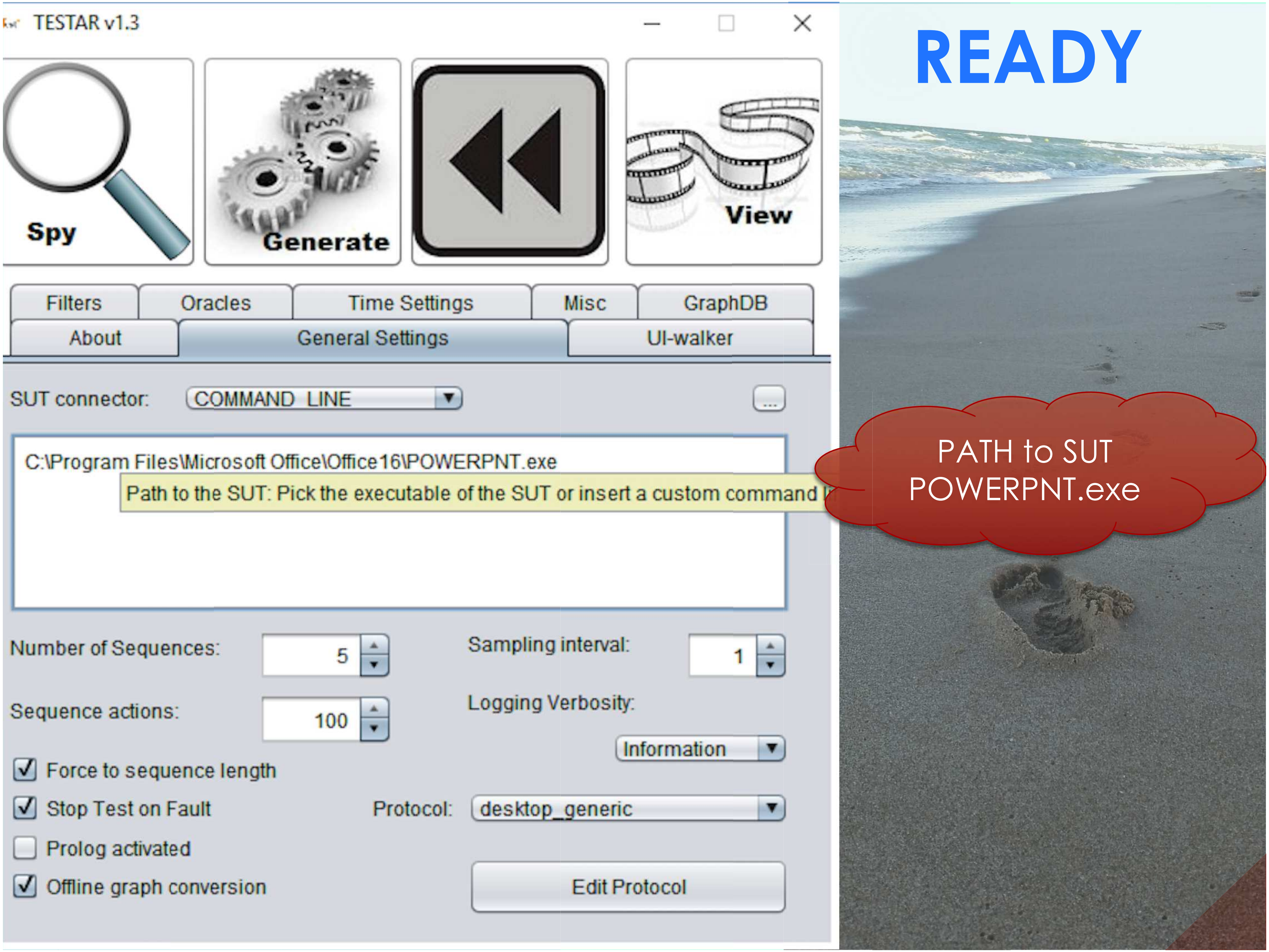
Execute to go to new state



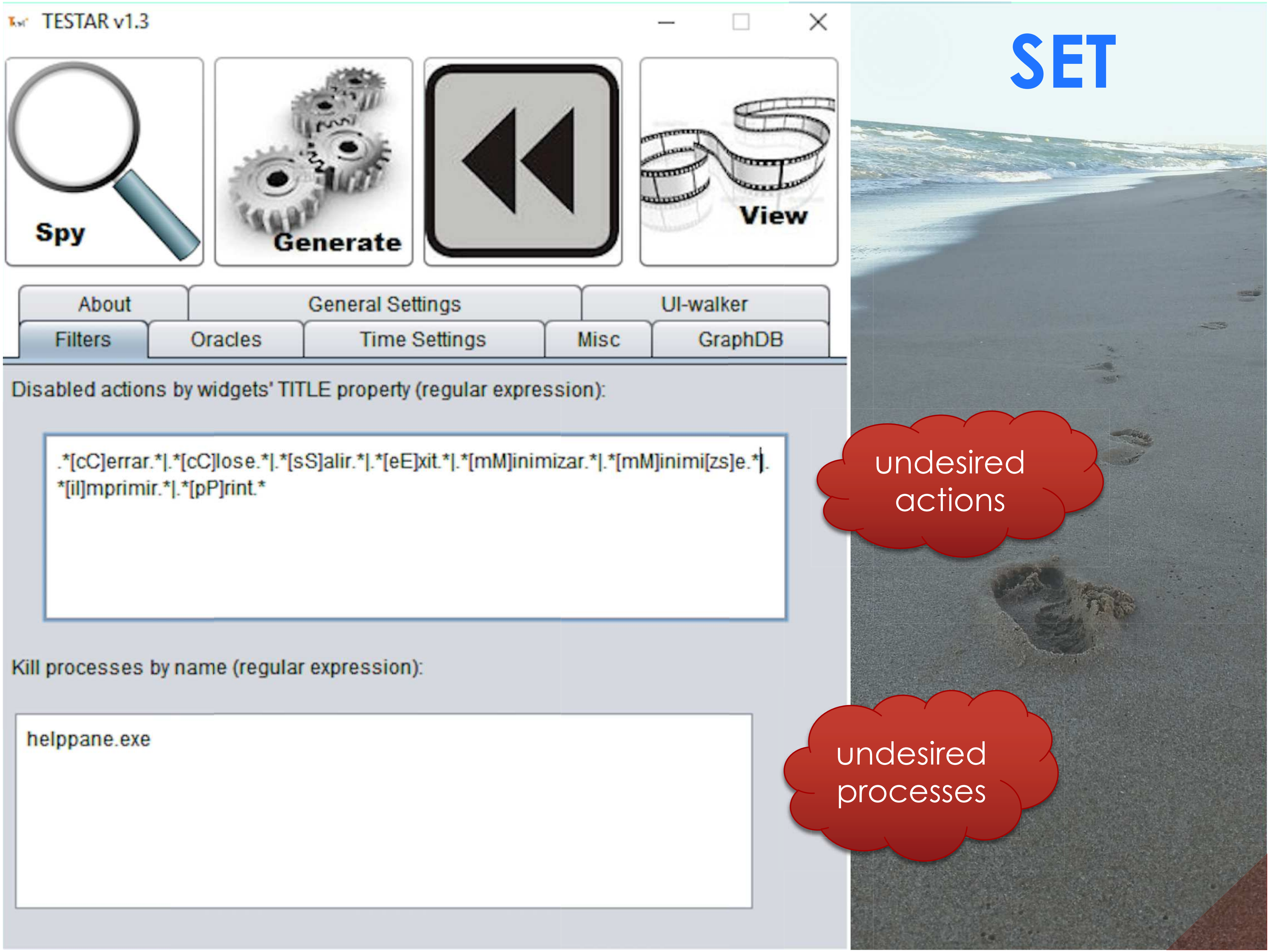
How?



GUI state graph database
Query for offline oracles
Update
Start SUT
Start TESTAR
Scan SUT for current GUI state
Check online state oracles
Failure detected
Executable test sequence leading to a failure
Check stop criteria
yes
Stop SUT
Execute action
Select action
Derive (+filter) actions
Test*



TESTAR v1.3
Spy
Generate
View
Filters
Oracles
Time Settings
Misc
GraphDB
About
General Settings
UI-walker
SUT connector:
COMMAND_LINE
C:\Program Files\Microsoft Office\Office16\POWERPNT.exe
Path to the SUT: Pick the executable of the SUT or insert a custom command
Number of Sequences:
5
Sampling interval:
1
Sequence actions:
100
Logging Verbosity:
Information
Force to sequence length
Stop Test on Fault
Protocol:
desktop_generic
Prolog activated
Offline graph conversion
Edit Protocol
READY
PATH to SUT
POWERPNT.exe


TESTAR v1.3
SET
Spy
Generate
View
About
General Settings
UI-walker
Filters
Oracles
Time Settings
Misc
GraphDB
Disabled actions by widgets' TITLE property (regular expression):
.*[cC]errar.*|.*[cC]lose.*|.*[sS]alir.*|.*[eE]xit.*|.*[mM]inimizar.*|.*[mM]inimi[zs]e.*|.
[iI]mprimir.|.*[pP]rint.*
undesired actions
Kill processes by name (regular expression):
helppane.exe
undesired processes

TEST

We can start automated testing

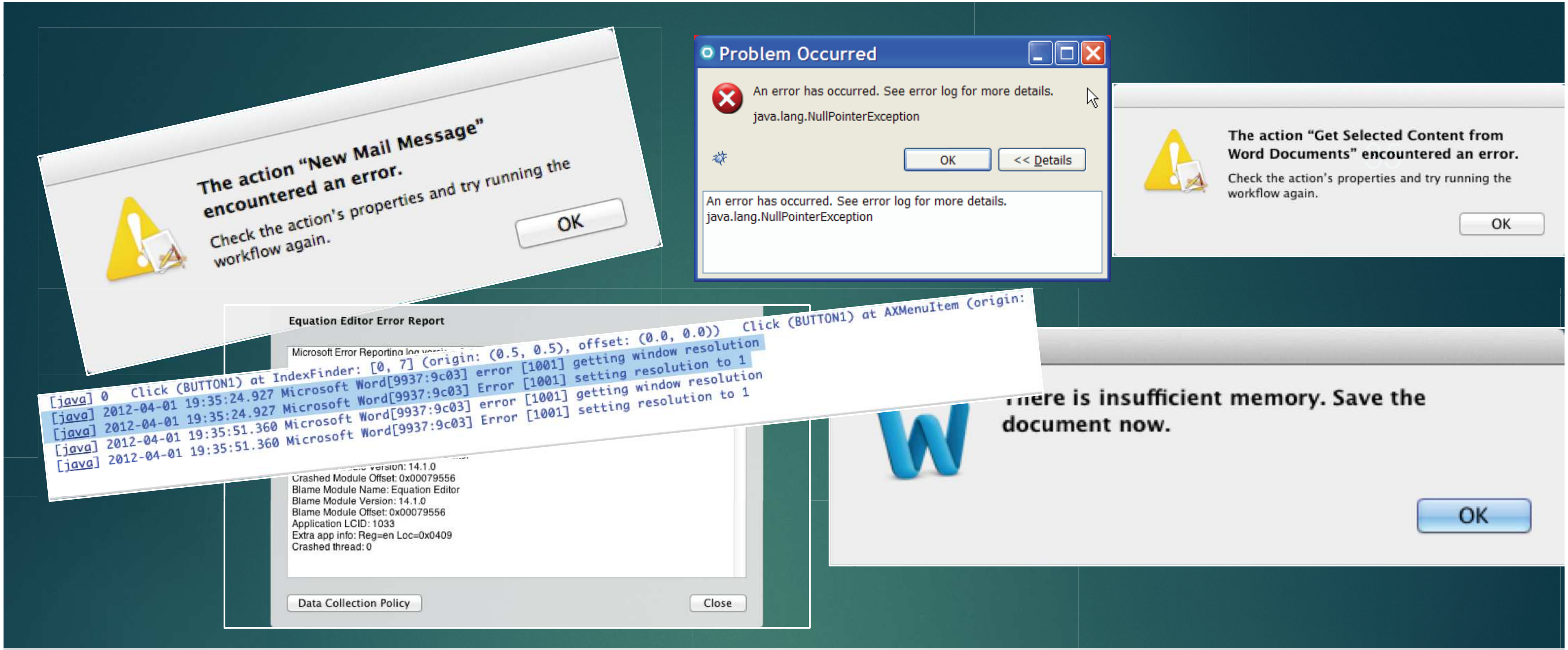
- Immediately (minimal set-up)
- No scripts
- No maintenance here
  - The widget tree is extracted in each new state
  - If the state is different, so is the widget tree

100% Automated online oracles

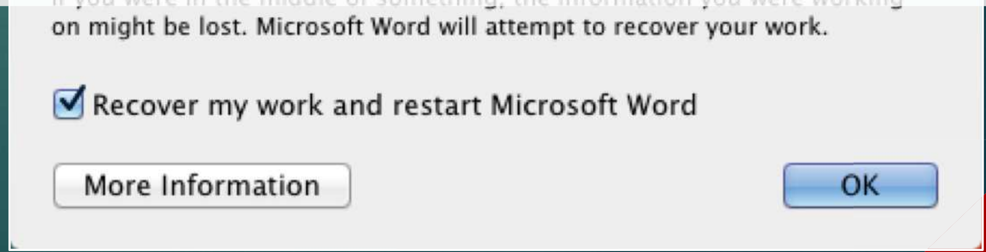
```
Verdict oracle_Crash (State state){
    if(!state.get(IsRunning,false))
        return new Verdict("System crashed!");
}
```

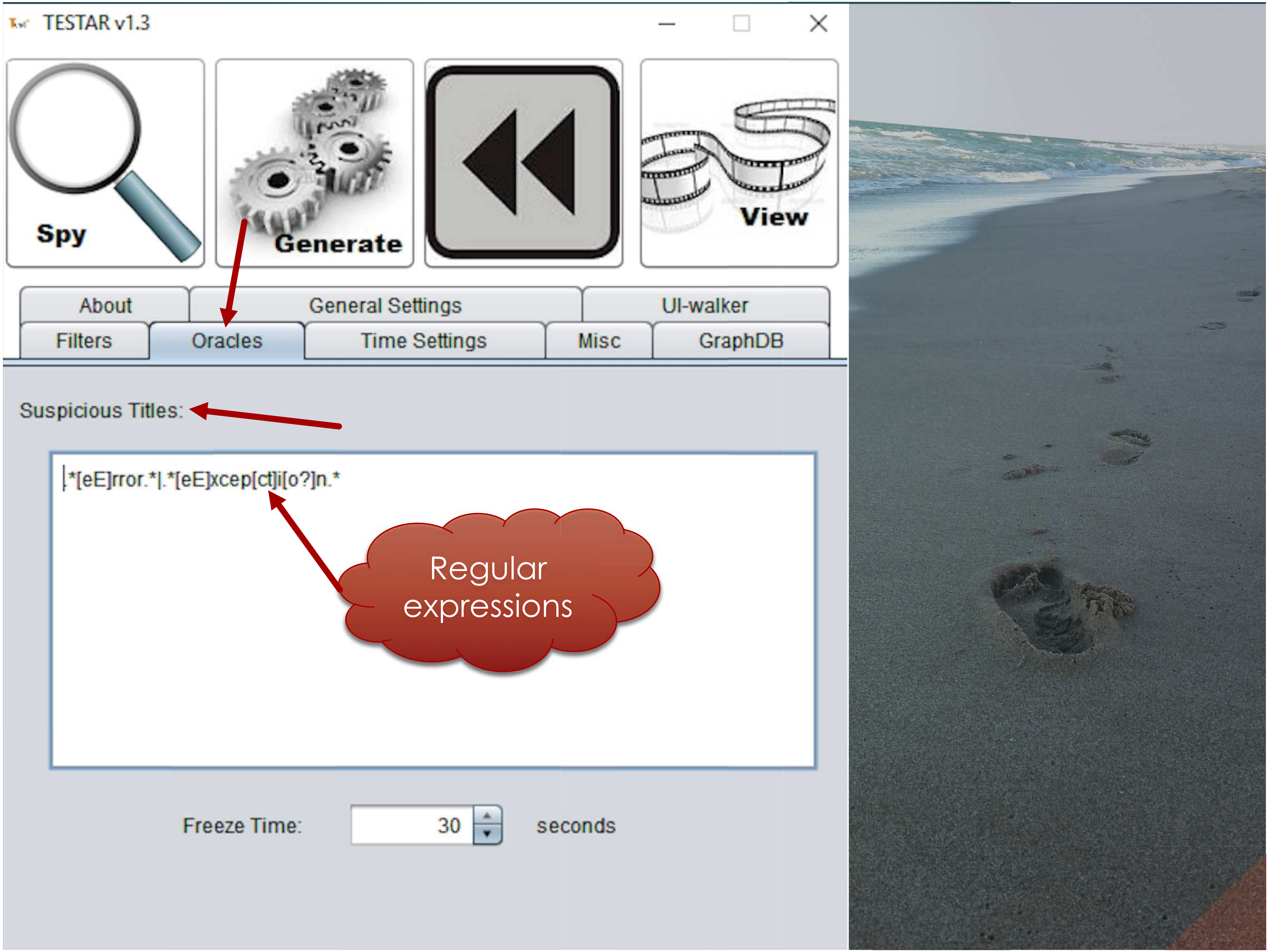
```
Verdict oracle_Responsiveness (State state){
    if(state.get(NotResponding, true))
        return new Verdict("System not responding!");
}
```

- Crashes
- Hangs



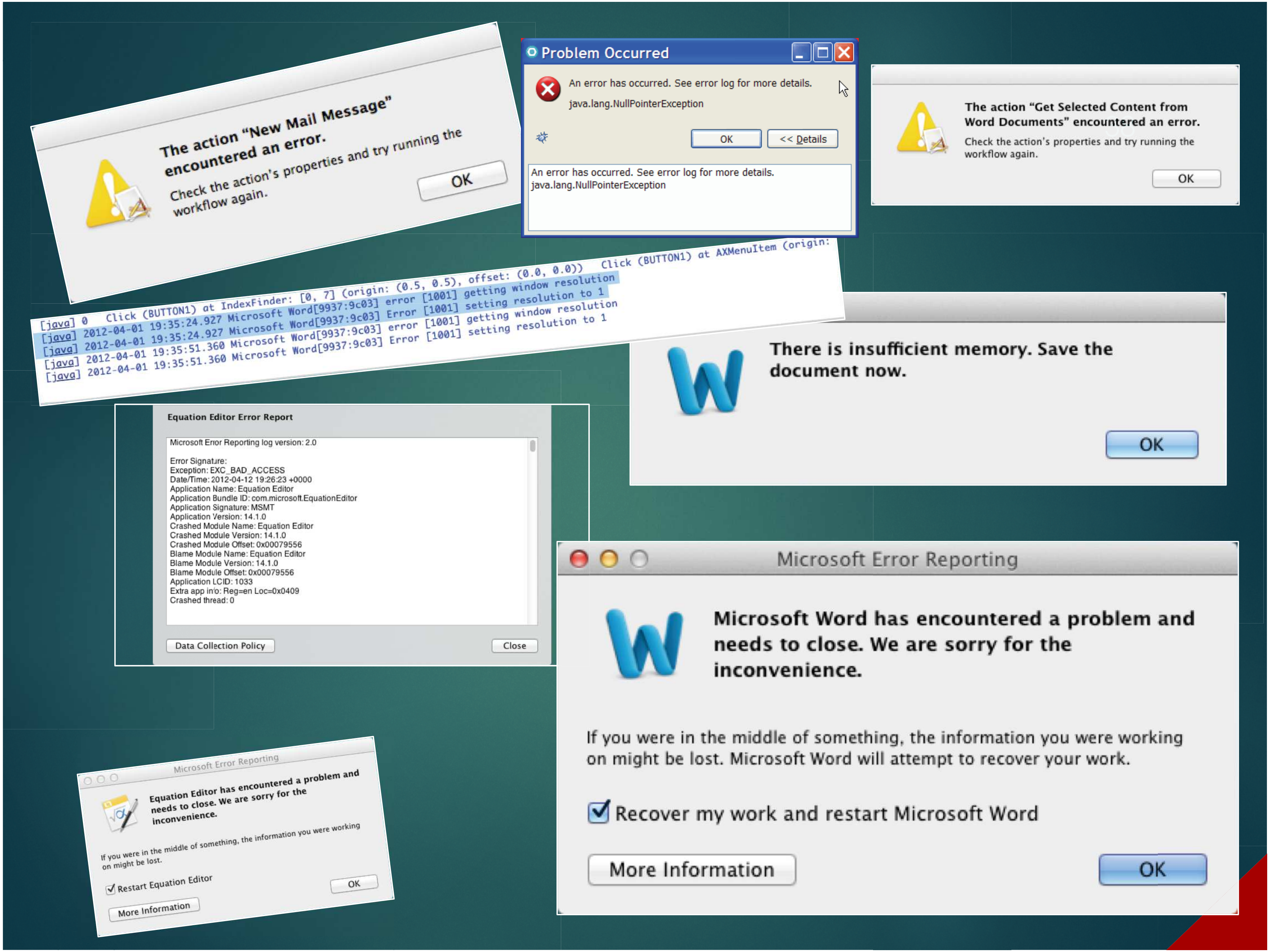
- Online oracles for suspicious titles and outputs
- Specify them with a regular expressión




TESTAR v1.3
Spy
Generate
View
About
General Settings
UI-walker
Filters
Oracles
Time Settings
Misc
GraphDB
Suspicious Titles:
.*[eE]rror.*|.*[eE]xcep[ct]i[o?]n.*
Regular expressions
Freeze Time:
30
seconds

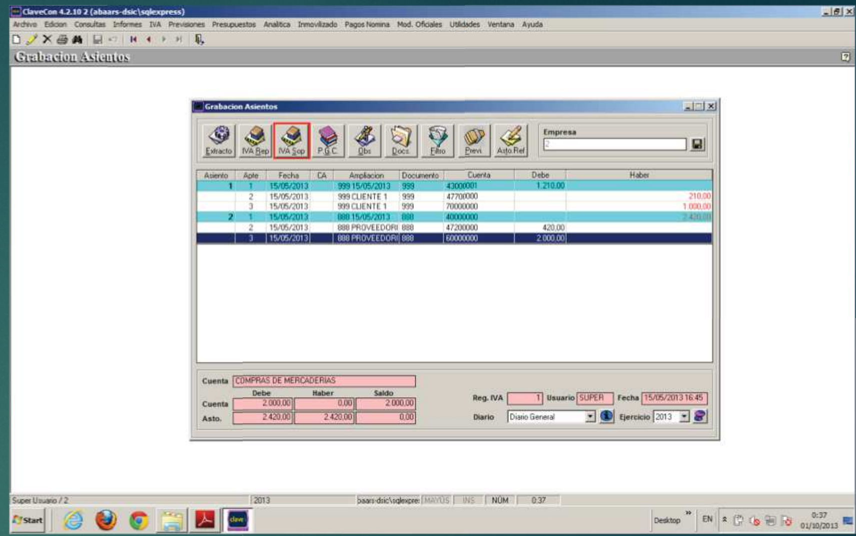
```
Verdicts oracle_SuspiciousTitles(State state){
   verdicts = new Verdicts():
   String regEx = settings().get(SuspiciousTitles);

   // search all widgets for suspicious titles
   for(Widget w : state){
      String title = w.get(Title, "");
      if(title.matches(regEx)){
      verdicts.add(new Verdict(“suspicious title..”);
   }
return verdicts;
}
```

Oracle – Suspicious titles (under the hood)


The action "New Mail Message" encountered an error.
Check the action's properties and try running the workflow again.
OK
Problem Occurred
An error has occurred. See error log for more details.
java.lang.NullPointerException
OK
<< Details
An error has occurred. See error log for more details.
java.lang.NullPointerException
The action "Get Selected Content from Word Documents" encountered an error.
Check the action's properties and try running the workflow again.
OK
[java] 0 Click (BUTTON1) at IndexFinder: [0, 7] (origin: (0.5, 0.5), offset: (0.0, 0.0)) Click (BUTTON1) at AXMenuItem (origin:
[java] 2012-04-01 19:35:24.927 Microsoft Word[9937:9c03] error [1001] getting window resolution
[java] 2012-04-01 19:35:24.927 Microsoft Word[9937:9c03] Error [1001] setting resolution to 1
[java] 2012-04-01 19:35:51.360 Microsoft Word[9937:9c03] error [1001] getting window resolution
[java] 2012-04-01 19:35:51.360 Microsoft Word[9937:9c03] Error [1001] setting resolution to 1
There is insufficient memory. Save the document now.
OK
Equation Editor Error Report
Microsoft Error Reporting log version: 2.0
Error Signature:
Exception: EXC_BAD_ACCESS
Date/Time: 2012-04-12 19:26:23 +0000
Application Name: Equation Editor
Application Bundle ID: com.microsoft.EquationEditor
Application Signature: MSMT
Application Version: 14.1.0
Crashed Module Name: Equation Editor
Crashed Module Version: 14.1.0
Crashed Module Offset: 0x00079556
Blame Module Name: Equation Editor
Blame Module Version: 14.1.0
Blame Module Offset: 0x00079556
Application LCID: 1033
Extra app info: Reg=en Loc=0x0409
Crashed thread: 0
Data Collection Policy
Close
Microsoft Error Reporting
Microsoft Word has encountered a problem and needs to close. We are sorry for the inconvenience.
If you were in the middle of something, the information you were working on might be lost. Microsoft Word will attempt to recover your work.
Recover my work and restart Microsoft Word
More Information
OK
Microsoft Error Reporting
Equation Editor has encountered a problem and needs to close. We are sorry for the inconvenience.
If you were in the middle of something, the information you were working on might be lost.
Restart Equation Editor
OK
More Information

ClaveiCon

- Spanish SME
- ERP system
- Written in Visual Basic
- Microsoft SQL Server 2008 database
- Targets the Windows operating systems.



	TESTAR
Preparation	26 hour
Testing	91 hour
Post testing	1,5 hour
Critical faults	10

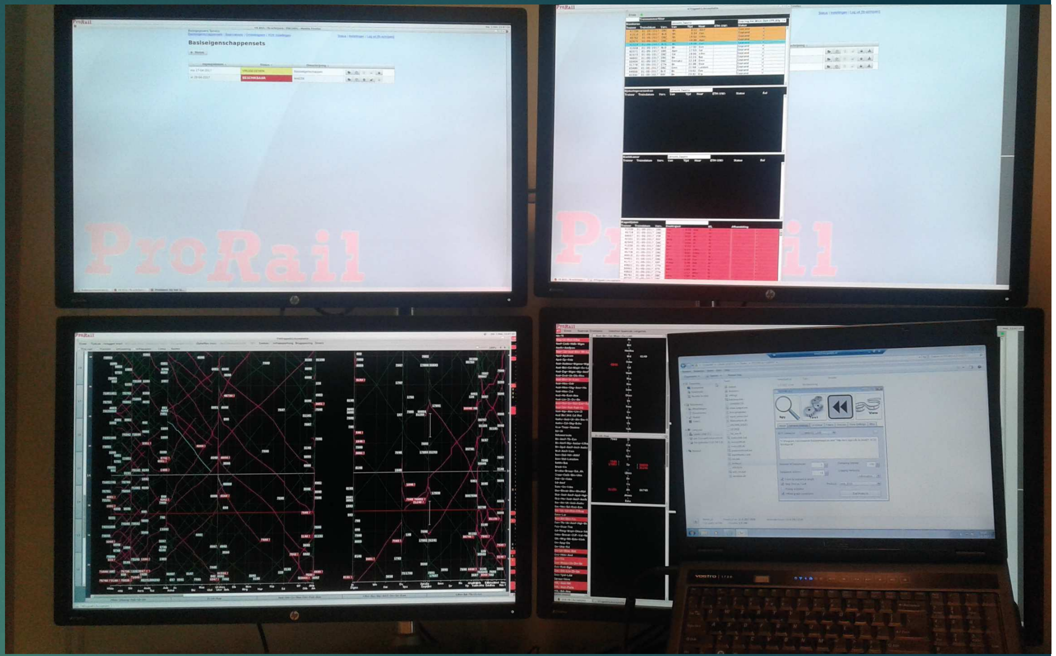
SOFTEAM

- French and large company
- Backend system for virtualization
- Web GUI
- We could re-inject existing faults

$$\text{FDR} = \frac{\text{num of Faults found}}{\text{num of injected Faults}} \times 100\%$$

	TESTAR	Manual
Preparation	40 hour	36 hour
Testing	77 hour	1 hour
Post testing	3,5 hour	2 hour
FDR	61%	83%
Code coverage	70%	86%

Cap Gemini/ ProRail



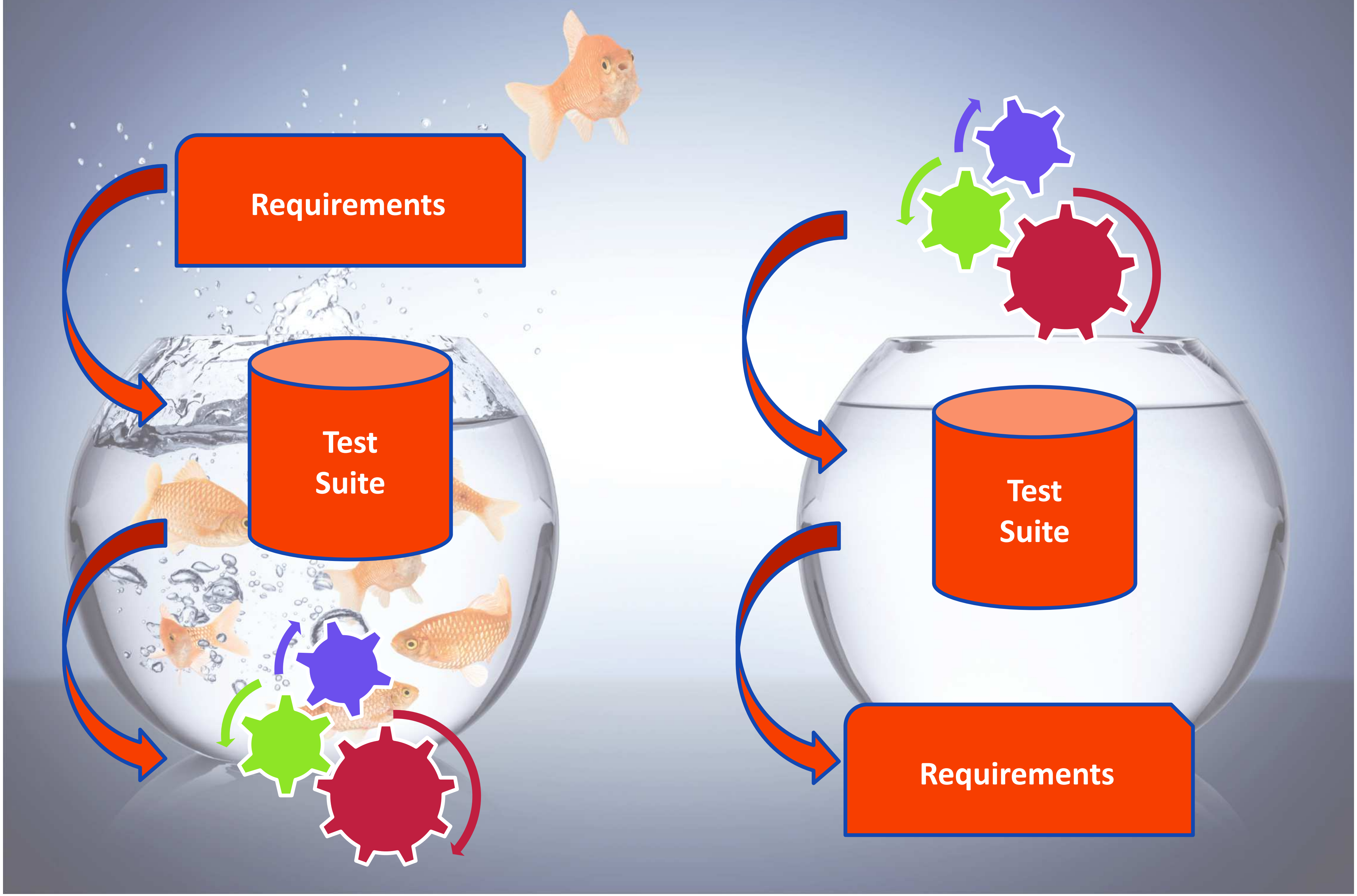
- Dutch cooperation
- Web GUI
- System for managing the assignment of train platforms

	TESTAR	Manual
Preparation	44 hour	43 hour
Testing	51 hour	6 hour
Post testing	5 hour	2 hour
Critical faults	4	0
Functional coverage	80%	73%

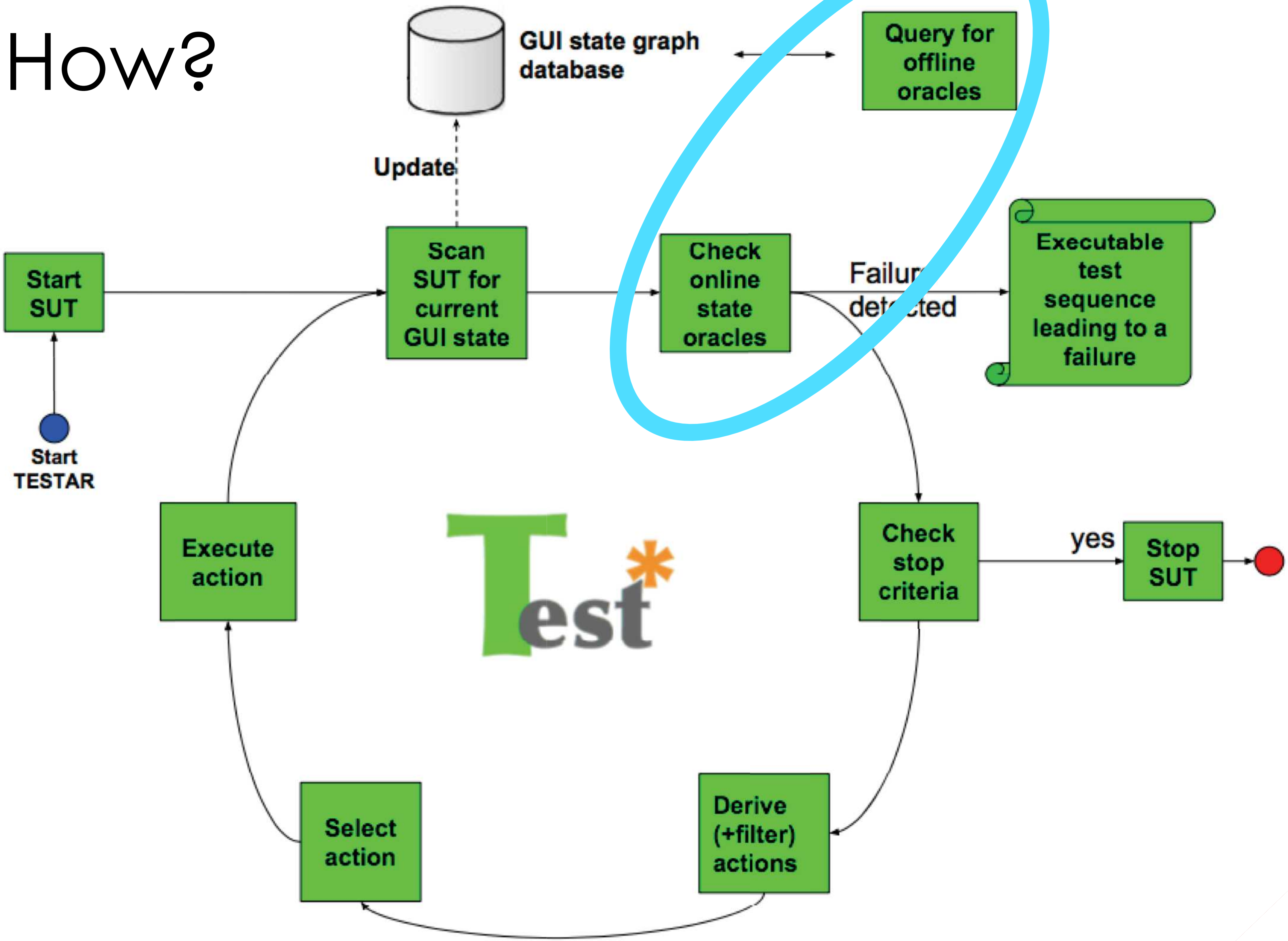
Beside these Test*

- Microsoft office suite
- Bitrix 24
- Test the test tool TESTONA (eclise based)
- Over 10 web applications of Spanish companies
- 12 students currently working on it
- Several companies doing proof of concepts

How does it change testing?



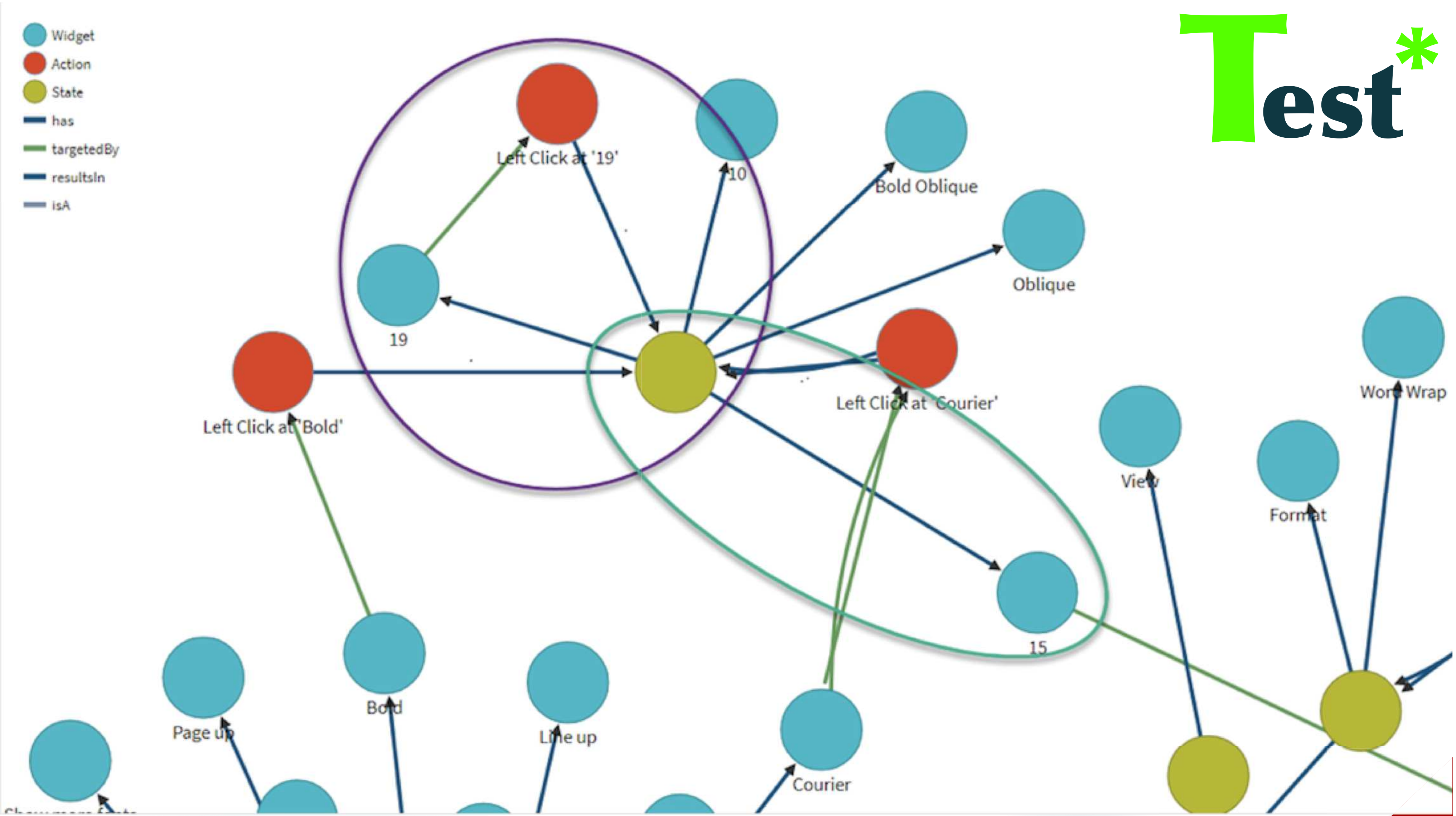
How?


GUI state graph database
Query for offline oracles
Update
Start SUT
Scan SUT for current GUI state
Check online state oracles
Failure detected
Executable test sequence leading to a failure
Start TESTAR
Execute action
Check stop criteria
yes
Stop SUT
Select action
Derive (+filter) actions
Test*

```
Verdicts oracle_ImagesWAI(State state) {
   verdicts = new Verdicts():
   for(Widget w : state){
     Role role = w.get(Tags.Role);
     if (role.equals("UIAImage") && title.isEmpty())
      verdicts.add(new Verdict("Not all images have an
                     alternate textual description");
   }
   return verdicts;
}
```

Online state oracle

Offline oracles: Query the graph database



Application/Domain specific oracles

Test*ar.org

Need to be programmed/specified

We cannot avoid making oracles manually

COMPLEXITY

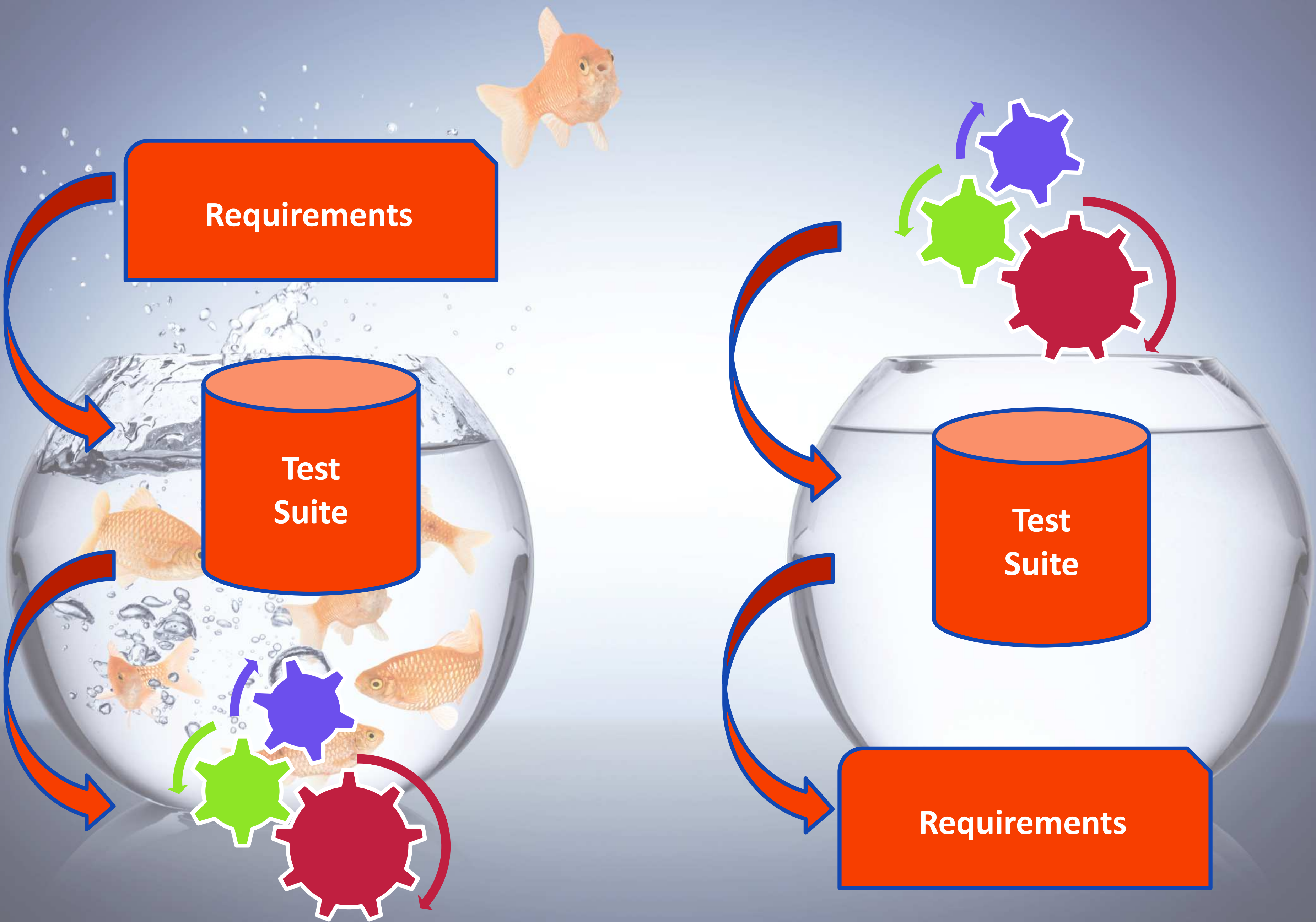
vs

EFFECTIVITY

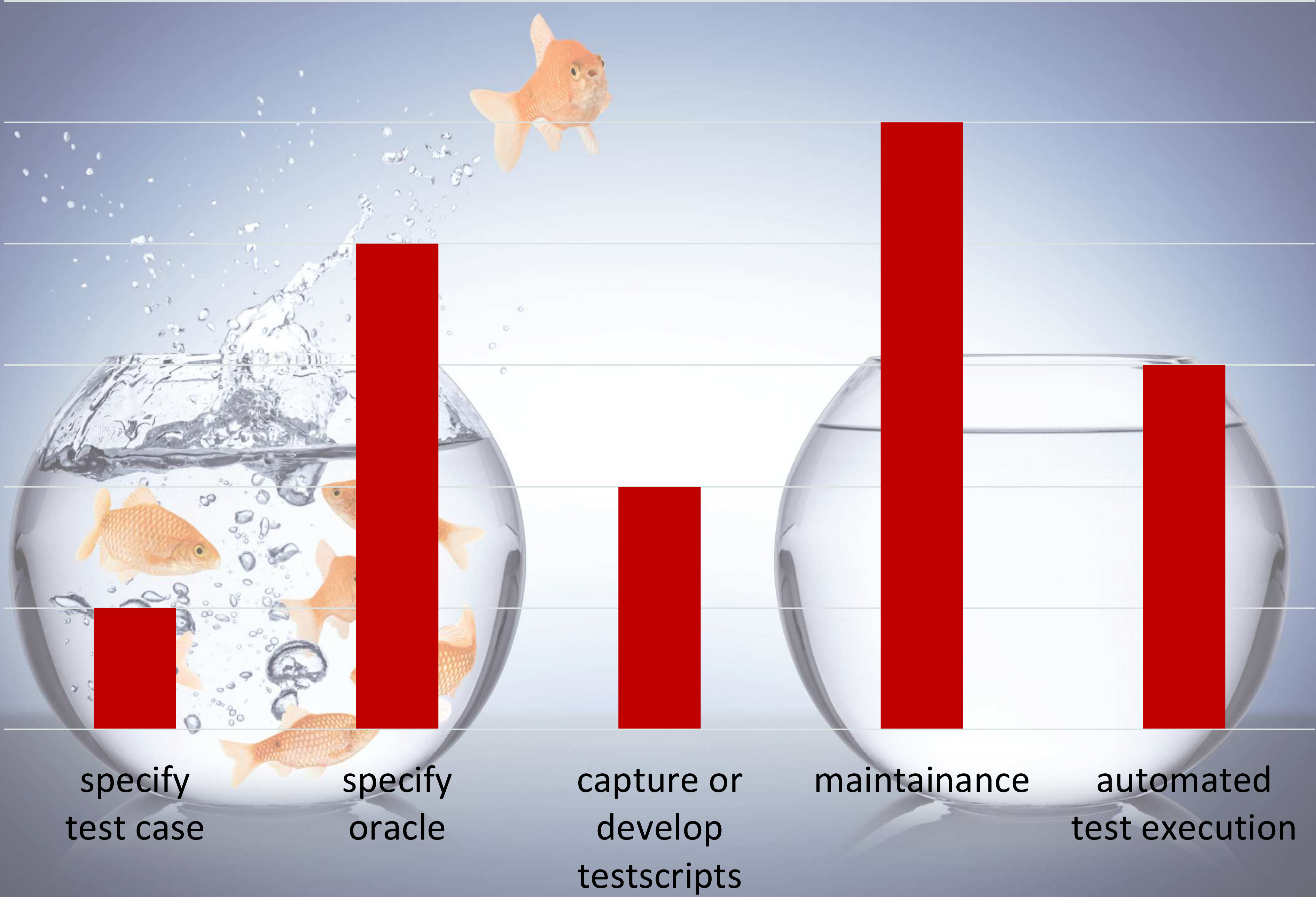
TESTAR shares this problem with ***ALL*** automated approaches

Oracle problem

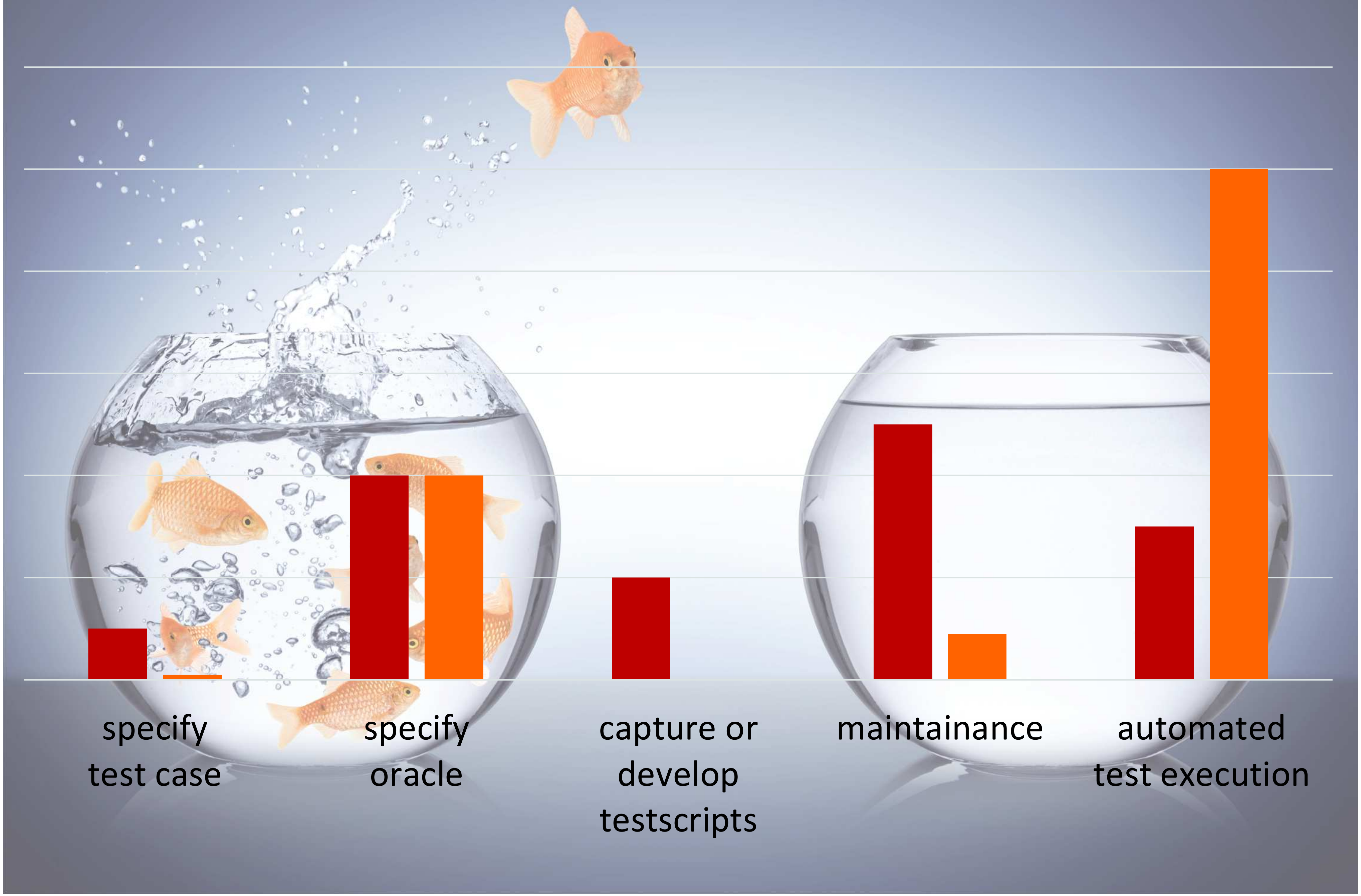
How does it change testing?



How is the test effort distributed



How can Test* change testing?



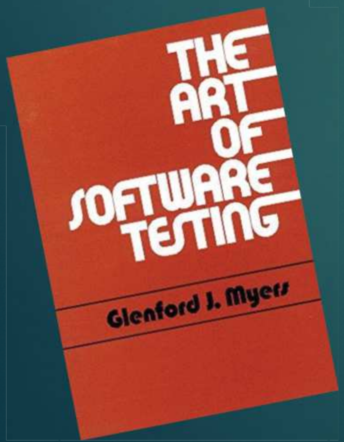
Random testing

"Valuable test case generation scheme"

E. Girard and J.C. Rault, A Programming Technique for Software Reliability, IEEE Symposium on Computer Software Reliability, 1973

"Necesary final step in the testing activities"

T. A. Thayer, M. Lipow, and E. C. Nelson. Software Reliability. North Holland, Amsterdam, 1978.

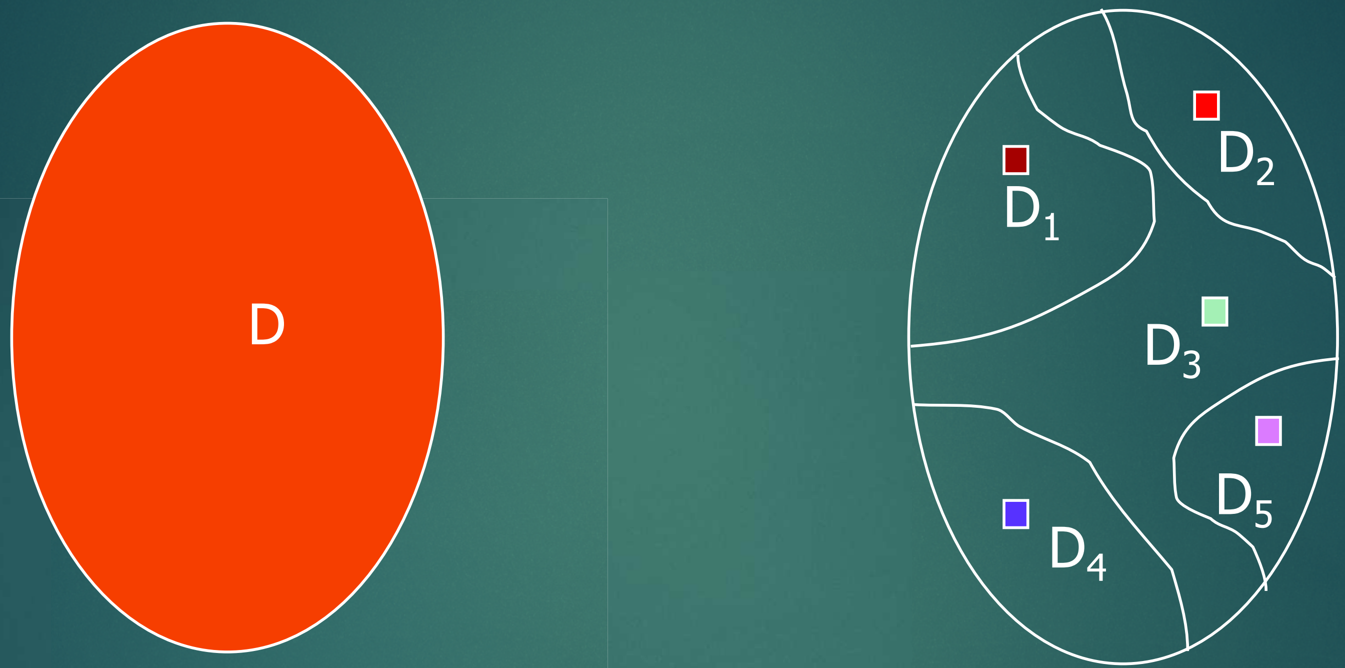


"Probably the poorest testing method"

Glenford Myers, The Art of Software Testing. New York: Wiley, 1979.

Use partition testing

In the 70s



Use domain knowledge of the SUT to partition
Group together similar test cases
Choose one

Random testing

In the 80s

438 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-10, NO. 4, JULY 1984

An Evaluation of Random Testing

JOE W. DURAN, MEMBER, IEEE, AND SIMEON C. NTAFOS, MEMBER, IEEE

Abstract—Random testing of programs has usually (but not always) been viewed as a worst case of program testing. Testing strategies that take into account the program structure are generally preferred. Path testing is an often proposed ideal for structural testing. Path testing is treated here as an instance of partition testing, where by partition testing is meant any testing scheme which forces execution of at least one test case from each subset of a partition of the input domain. Simulation results are presented which suggest that random testing may often be more cost effective than partition testing schemes. Also, results of actual random testing experiments are presented which confirm the viability of random testing as a useful validation tool.

Index Terms—Partition testing, path testing, random testing, software testing experiments.

I. Introduction

In [10], Howden remarks that "... much has been written about structural testing, but little about black box testing, the method it was supposed to replace and over which it was supposed to be an improvement." He then reports on a successful refinement of black box functional testing, in which a detailed design "structure" is used to develop test cases. Another black box testing method which has not yet been adequately studied is random testing. There has been strong disagreement about its value. Myers [13] says "probably the poorest [test case design] methodology is random input testing . . . ." On the other hand, Thayer *et al.* [14] recommend that the final testing of a program should be done with input cases chosen at random from its operational profile—the expected run time distribution of inputs to the program. Additionally, they showed how quantitative estimates of a program's operational reliability can be inferred from random testing. Girard and Rault [5] have also proposed random testing as a valuable test case generation scheme. Further, recent results [3] show that path testing, a popular paradigm for structural testing, can lead to less satisfactory reliability estimates than a corresponding number of random test case executions.

In this paper we present some simulation and experimental studies of random testing in order to investigate its effectiveness. In Section III we report on some simulation comparisons of random versus partition testing. In Section IV we investigate the error detection ability of random testing by applying it to a set of programs with known errors. Finally, in Section V we evaluate how well a set of programs is tested by random testing by using a variety of coverage measures such as the proportion of segments and branches executed by randomly generated sets of test cases.

II. Estimates of Error Finding Ability

Let θ be the probability that a program will fail to execute correctly on an input case chosen from a given input distribution. If the program is used for a long period of time with input from a particular operational profile (input distribution), then the failure rate actually experienced will converge toward θ. We thus refer to θ as the failure rate, which is a valuable measure of the operational reliability of the program.

Suppose the input domain D is partitioned into k subsets, $D_1, D_2, \cdots, D_k$, and that a randomly chosen input case has probability p_i of being chosen from D_i. Then $\theta = \sum_{i=1}^{k} p_i\theta_i$, where θ_i is the failure rate for D_i. The term "partition testing" refers to any test data generation method which partitions the input domain and forces at least one test case to come from each subset. Path testing is a special case of partition testing. D could also be partitioned according to alternative functions a program is supposed to perform. For example, a checking account program performs alternative functions depending upon whether the transaction being processed is a deposit, a withdrawal with sufficient funds, or a withdrawal with insufficient funds. Thus, partition testing contains instances of both black box and structural testing.

Thayer *et al.* [14] showed that if n random tests are carried out and x program failures are discovered, then θ^*, the $1 - \alpha$ upper confidence bound on θ, is the largest θ value such that

$$\sum_{i=1}^{x} \binom{n}{i} \theta^i (1-\theta)^{n-i} > \alpha.$$

(That is, there is a probability of $1 - \alpha$ that the calculated bound θ^* exceeds the actual θ value of the program. The θ^* value is, of course, valid only with respect to the input distribution used for test data selection.)

If $x = 0$, then $\theta^* = 1 - \alpha^{1/n}$. It has been shown [3] that if n test runs are made according to a partition testing scheme, and no failures are discovered, then, if there is no prior knowledge about the distribution of θ_i values, the resulting value of θ^* is bounded below by $1 - \alpha^{1/n}$. It is this result that led us to investigate the error finding ability of random testing experimentally.

Ideally, one might wish to characterize this error finding ability in terms of errors found per dollar, perhaps weighted by error severity. Being at present unable to characterize errors and their effects well enough to prepare any formal models for making "errors found per dollar" calculations, we pose two related questions.

Manuscript received October 1, 1982; revised September 30, 1983. This work was supported in part by the National Science Foundation under Grant MCS-8003322.

J. W. Duran was with the Southwest Research Institute, San Antonio, TX 78284. He is now with the Division of Mathematics, Computer Science, and Systems Design, University of Texas, San Antonio, TX 78285.

S. C. Ntafos is with the Computer Science Program, University of Texas at Dallas, Richardson, TX 75080.

PARTITION TESTING DOES NOT INSPIRE CONFIDENCE

Dick Hamlet
Department of Computer Science and Engineering
Oregon Graduate Center
19600 NW von Neumann Drive
Beaverton, OR 97006

Ross Taylor
Tektronix, Inc.
PO Box 1000, M/S 63/356
Wilsonville, OR 97070-1000

Abstract

Partition testing, in which a program's input domain is divided according to some rule and tests conducted within the subdomains, enjoys a good reputation. However, comparison between testing that observes partition boundaries and random sampling that ignores the partitions gives the counterintuitive result that partitions are of little value. In this paper we improve the negative results published about partition testing, and try to reconcile them with its intuitive value. Partition testing is shown to be more valuable than random testing only when the partitions are narrowly based on expected faults and there is a good chance of failure. For gaining confidence from successful tests, partition testing as usually practiced has little value.

1. Partition Testing

Partitions are the natural solution to the two fundamental testing problems of systematic method and test volume. By dividing a program's input domain into classes whose points are somehow "the same," it is sufficient to try one representative from each class; the problem of systematic testing is reduced to a proper definition of the classes. Partitions can be defined using all the information about a program. They can be based on requirements or specifications ("blackbox" testing), on features of the code ("structural" testing), even on the process by which the software was developed, or on the suspicions and fears of a programmer. Any such information divides the input domain, for example, into inputs required to invoke one of several software features F_1, F_2, ...; or, inputs that do (do not) make use of a suspect data structure; etc. The partition testing method forms the intersection of these input classes—e.g., one class formed from the above would be those inputs requiring feature F_2 *and* making use of the suspect data structure. The goal is to make the resulting classes so finely divided that each aspect of the program, of the specification, of development, each programmer concern, etc., is separated into one partition. Goodenough and Gerhart [1] expressed this method using "significant predicates" from both specification and program to divide inputs into classes, and the intersection by considering all combinations of predicate values. Richardson and Clarke [2] describe the special case of intersecting specification classes with those defined by program path predicates. Although "partition testing" usually carries a connotation of functional testing involving specifications, here we use the term in the general sense of any input-space division. Thus our results apply to most testing schemes that have been proposed, including all variants of path coverage, mutation, etc.

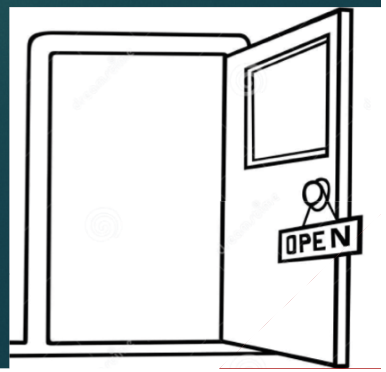
The strength of partition testing is its ability to use any and all information, and to examine information in combinations that may not have been thought of during development. Intuitively, the source of program bugs is some unlikely combination of requirements, design, and programmer inattention. By including these factors in the partition definition, it seems that nothing has been missed in testing. Good partitions are defined and refined throughout development as information arises.

Partition testing can be no better than the information that defines its classes. For the method to work perfectly, all inputs in one class must be interchangeable—if one causes a failure, any other must do the same. (Goodenough and Gerhart called this property "reliability", but

Duran and Ntafos (1984): simulation and experiments showing random better than systematic partition testing.

Hamlet and Taylor (1988): more experiments showing the same

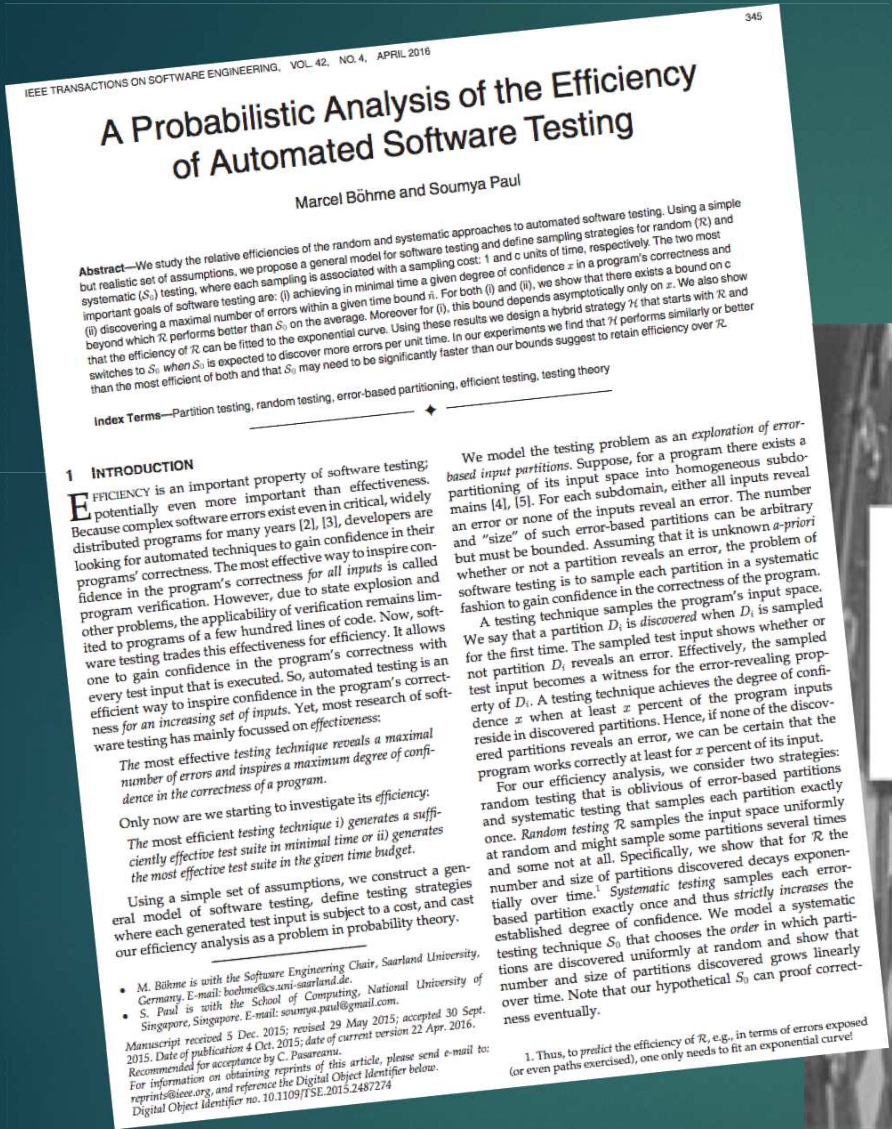
Counterintuitive →



Random testing

Counterintuitive

- Why do random testing and systematic testing seem to be almost on par?
- What are the properties of random testing?
- When is random testing more effective than partitioning and the other way around?



IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 42, NO. 4, APRIL 2016

A Probabilistic Analysis of the Efficiency of Automated Software Testing

Marcel Böhme and Soumya Paul

Abstract—We study the relative efficiencies of the random and systematic approaches to automated software testing. Using a simple but realistic set of assumptions, we propose a general model for software testing and define sampling strategies for random ($\mathcal{R}$) and systematic ($\mathcal{S}_0$) testing, where each sampling is associated with a sampling cost: 1 and c units of time, respectively. The two most important goals of software testing are: (i) achieving in minimal time a given degree of confidence x in a program's correctness and (ii) discovering a maximal number of errors within a given time bound $\hat{n}$. For both (i) and (ii), we show that there exists a bound on c beyond which $\mathcal{R}$ performs better than $\mathcal{S}_0$ on the average. Moreover for (i), this bound depends asymptotically only on x. We also show that the efficiency of $\mathcal{R}$ can be fitted to the exponential curve. Using these results we design a hybrid strategy $\mathcal{H}$ that starts with $\mathcal{R}$ and switches to $\mathcal{S}_0$ when $\mathcal{S}_0$ is expected to discover more errors per unit time. In our experiments we find that $\mathcal{H}$ performs similarly or better than the most efficient of both and that $\mathcal{S}_0$ may need to be significantly faster than our bounds suggest to retain efficiency over $\mathcal{R}$.

Index Terms—Partition testing, random testing, error-based partitioning, efficient testing, testing theory

1 Introduction

Efficiency is an important property of software testing; potentially even more important than effectiveness. Because complex software errors exist even in critical, widely distributed programs for many years [2], [3], developers are looking for automated techniques to gain confidence in their programs' correctness. The most effective way to inspire confidence in the program's correctness *for all inputs* is called program verification. However, due to state explosion and other problems, the applicability of verification remains limited to programs of a few hundred lines of code. Now, software testing trades this effectiveness for efficiency. It allows one to gain confidence in the program's correctness with every test input that is executed. So, automated testing is an efficient way to inspire confidence in the program's correctness *for an increasing set of inputs*. Yet, most research of software testing has mainly focussed on *effectiveness*:

The most effective testing technique reveals a maximal number of errors and inspires a maximum degree of confidence in the correctness of a program.

Only now are we starting to investigate its *efficiency*:

The most efficient testing technique i) generates a sufficiently effective test suite in minimal time or ii) generates the most effective test suite in the given time budget.

Using a simple set of assumptions, we construct a general model of software testing, define testing strategies where each generated test input is subject to a cost, and cast our efficiency analysis as a problem in probability theory.

We model the testing problem as an *exploration of error-based input partitions*. Suppose, for a program there exists a partitioning of its input space into homogeneous subdomains [4], [5]. For each subdomain, either all inputs reveal an error or none of the inputs reveal an error. The number and "size" of such error-based partitions can be arbitrary but must be bounded. Assuming that it is unknown *a-priori* whether or not a partition reveals an error, the problem of software testing is to sample each partition in a systematic fashion to gain confidence in the correctness of the program.

A testing technique samples the program's input space. We say that a partition D_i is *discovered* when D_i is sampled for the first time. The sampled test input shows whether or not partition D_i reveals an error. Effectively, the sampled test input becomes a witness for the error-revealing property of D_i. A testing technique achieves the degree of confidence x when at least x percent of the program inputs reside in discovered partitions. Hence, if none of the discovered partitions reveals an error, we can be certain that the program works correctly at least for x percent of its input.

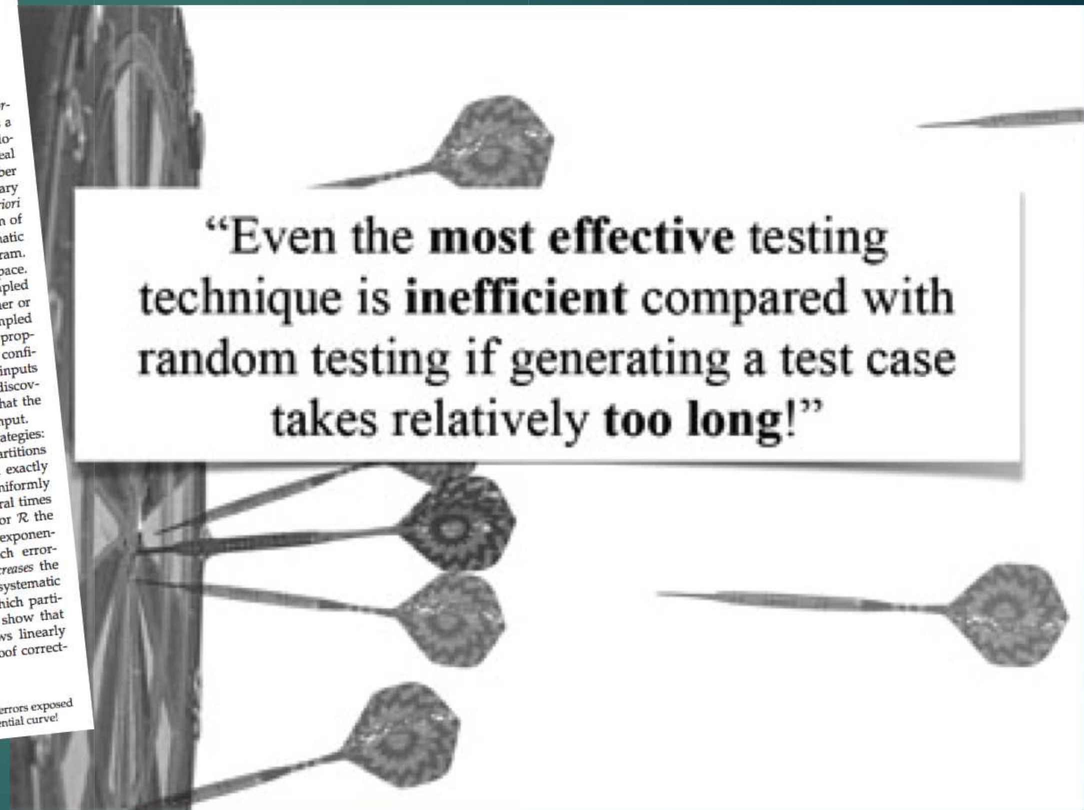
For our efficiency analysis, we consider two strategies: random testing that is oblivious of error-based partitions and systematic testing that samples each partition exactly once. *Random testing* $\mathcal{R}$ samples the input space uniformly at random and might sample some partitions several times and some not at all. Specifically, we show that for $\mathcal{R}$ the number and size of partitions discovered decays exponentially over time.[1] *Systematic testing* samples each error-based partition exactly once and thus *strictly increases* the established degree of confidence. We model a systematic testing technique $\mathcal{S}_0$ that chooses the *order* in which partitions are discovered uniformly at random and show that number and size of partitions discovered grows linearly over time. Note that our hypothetical $\mathcal{S}_0$ can proof correctness eventually.

• M. Böhme is with the Software Engineering Chair, Saarland University, Germany. E-mail: boehme@cs.uni-saarland.de.
• S. Paul is with the School of Computing, National University of Singapore, Singapore. E-mail: soumya.paul@gmail.com.

Manuscript received 5 Dec. 2015; revised 29 May 2015; accepted 30 Sept. 2015. Date of publication 4 Oct. 2015; date of current version 22 Apr. 2016.
Recommended for acceptance by C. Pasareanu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2487274

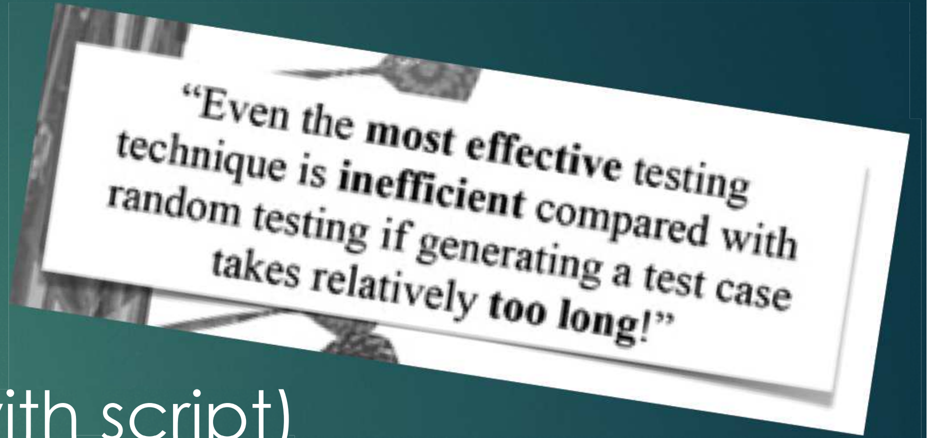
1. Thus, to predict the efficiency of $\mathcal{R}$, e.g., in terms of errors exposed (or even paths exercised), one only needs to fit an exponential curve!

Böhme and S. Paul (2016)



For automated GUI testing.....

- Generating test case is:
  - Specification
  - Capture (or automate with script)
  - Maintenance!!

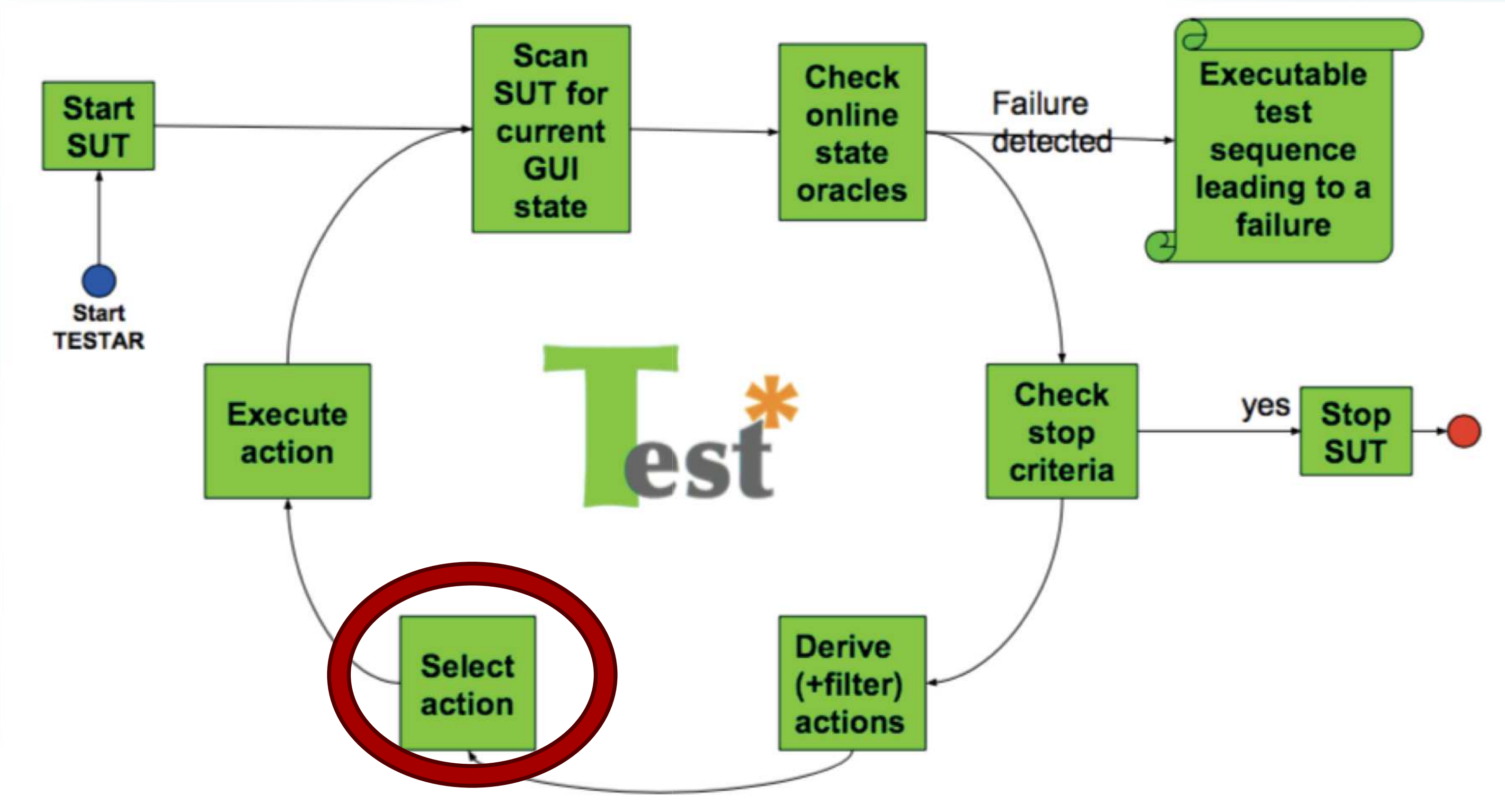


- And random selection gave us quite good results on the software we tested …….

- Can we do better?

How can we find more faults?

- Some test cases might be more likely to reveal faults
- Don´t pick at random, but try to optimize criteria!
- What criteria?



Where can we find faults?

- Surrogate measures
- We cannot measure %of faults found
- We measure something we believe, hope or have shown to be correlated to that attribute.
- Coverage
- Diversity
- Novelty

Let the testing tool learn by itself how to test better!!

Surrogate measures

- as many **different actions** as possible? **Q-learning**
- make **large call trees**? **Ant colonies**
- visit as **many different states** as possible? **Evolutionary algorithms**
- make **long sequences**? **Evolutionary algorithms**
- find **novel states**?
- We need to investigate many more

Machine Learning (Q-learning)

- sets S of possible states
- sets A of possible actions
- description T of the effect of action in a state
    - $T: S \times A \longrightarrow S$
    - state s then select an action from a ∊ A that causes a transition to a next state s'
- reward function $R: S \times A \longrightarrow \mathbb{R}$

find a policy π which maximizes the reward by selecting an appropriate action in each state

Rewards

- Set S of possible states the SUT can be in
- For all $s \in S$, we have sets $A_s \subseteq A$ of actions
- We focus is on exploration of the GUI
- We reward actions a with low execution count ec

$$\forall s \in S, a \in As:\ R(s,a) = \begin{cases} R_{max}\ , & ec\ (a) = 0 \\ \frac{1}{ec(a)}, & \textbf{otherwise} \end{cases}$$

$\mathcal{Q}$-learning algorithm

Require: $R_{max} > 0$ /* *reward for unexecuted actions* */
Require: $0 < \gamma < 1$ /* *discount factor* */

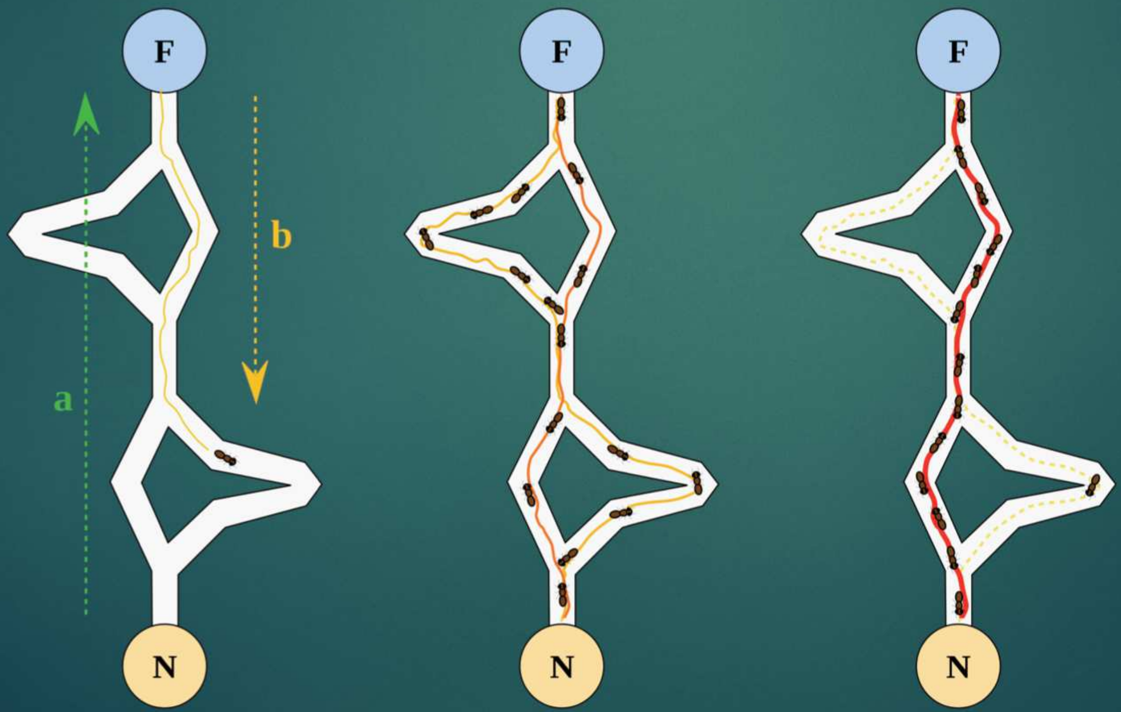
begin
  start SUT
  $\forall (s,a) \in S \times A :\ Q(s,a) \leftarrow R_{max}$
  initialize s and available action A_s
  repeat
    $a^* \leftarrow max_a\{Q(s,a) | a \in A_s\}$
    execute a^*
    obtain state s' and available actions $A_{s'}$
    $Q(s,a^*) \leftarrow R(s,a^*) + \gamma \cdot max_{a \in A_{s'}} Q(s',a)$
    $ec(a^*)$ ++
    $s \leftarrow s'$
  until stopping criteria met
  stop SUT
end

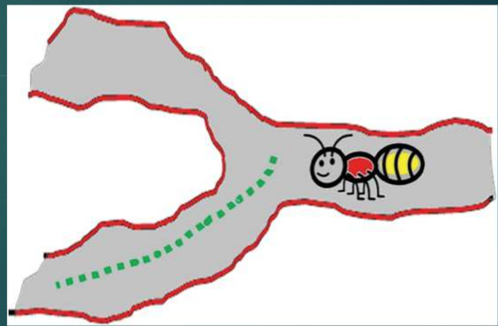
Learn Q

Use Q for selection

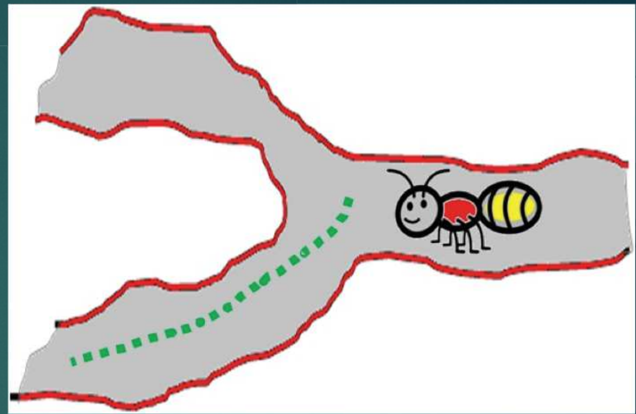
Ant Colony Optimization

- Collectively ants can solve complex tasks
- Ants communicate using pheromones
  - They lay this on their path
  - Pheromone trail strength accumulates when multiple ants use a path
  - Other ants go where there is good pheromone strength

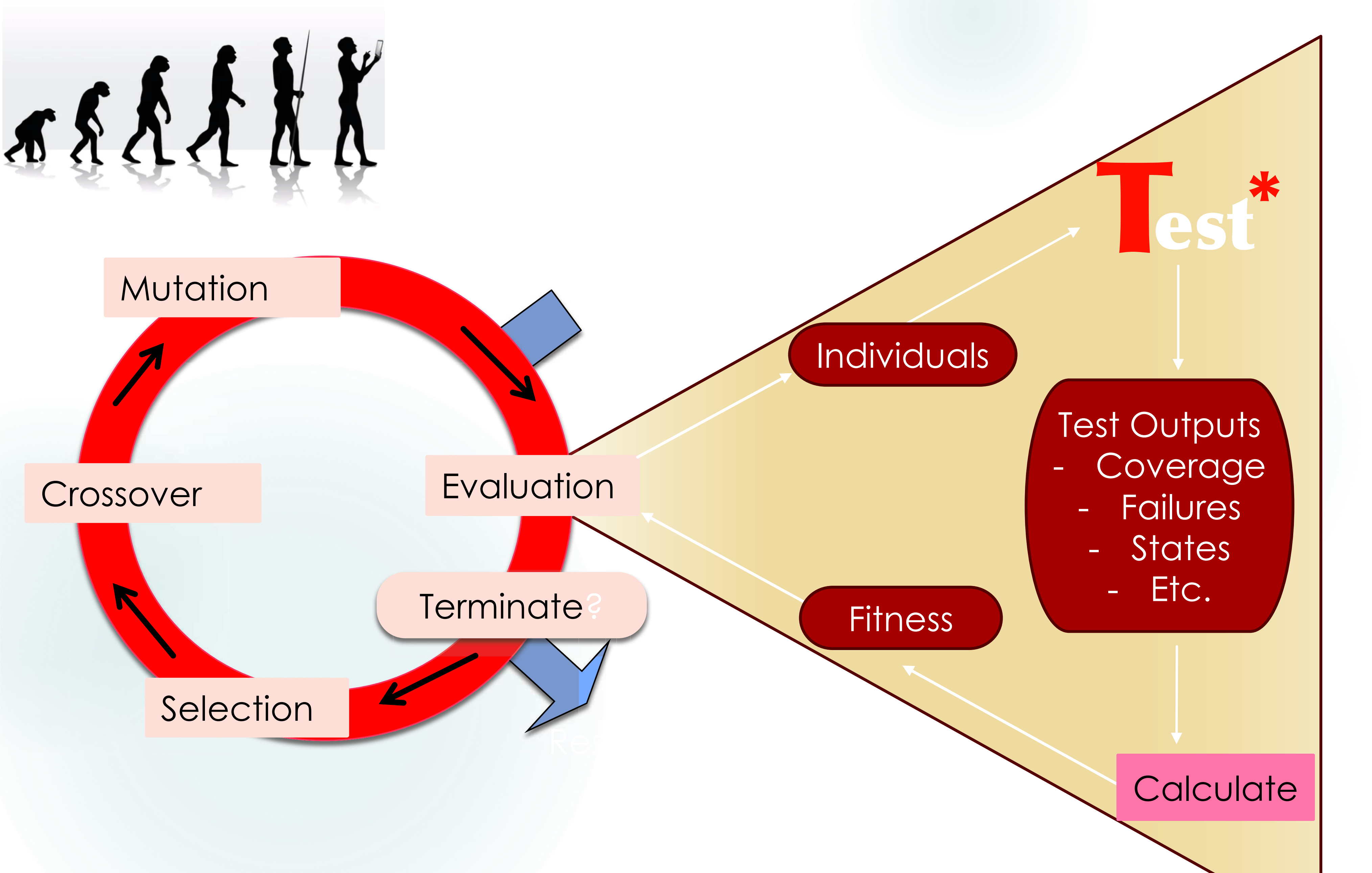




Ant Colony Optimization

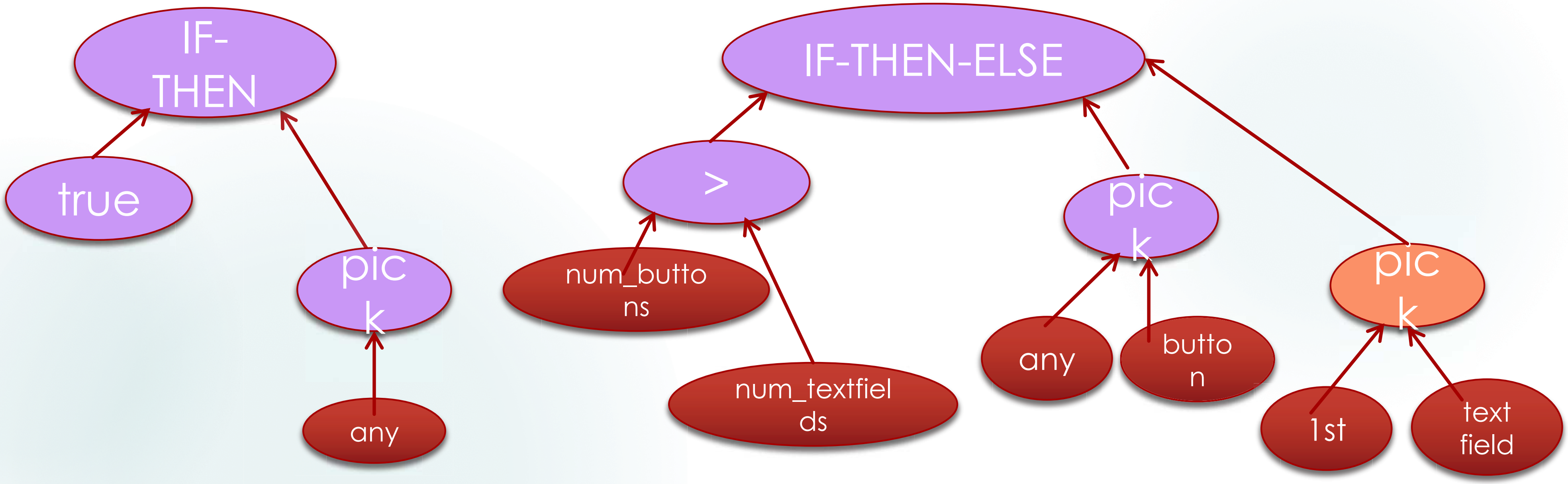


- We have a population of ants
- Set of choices C **(= actions)**
- The ants generate trails **(= test sequences)**
- By choosing c_i according to pheromone values p_i **(= selection criteria)**
- Choices **(= actions)** that appear in "good" trails **(= max call tree)** accumulate pheromones

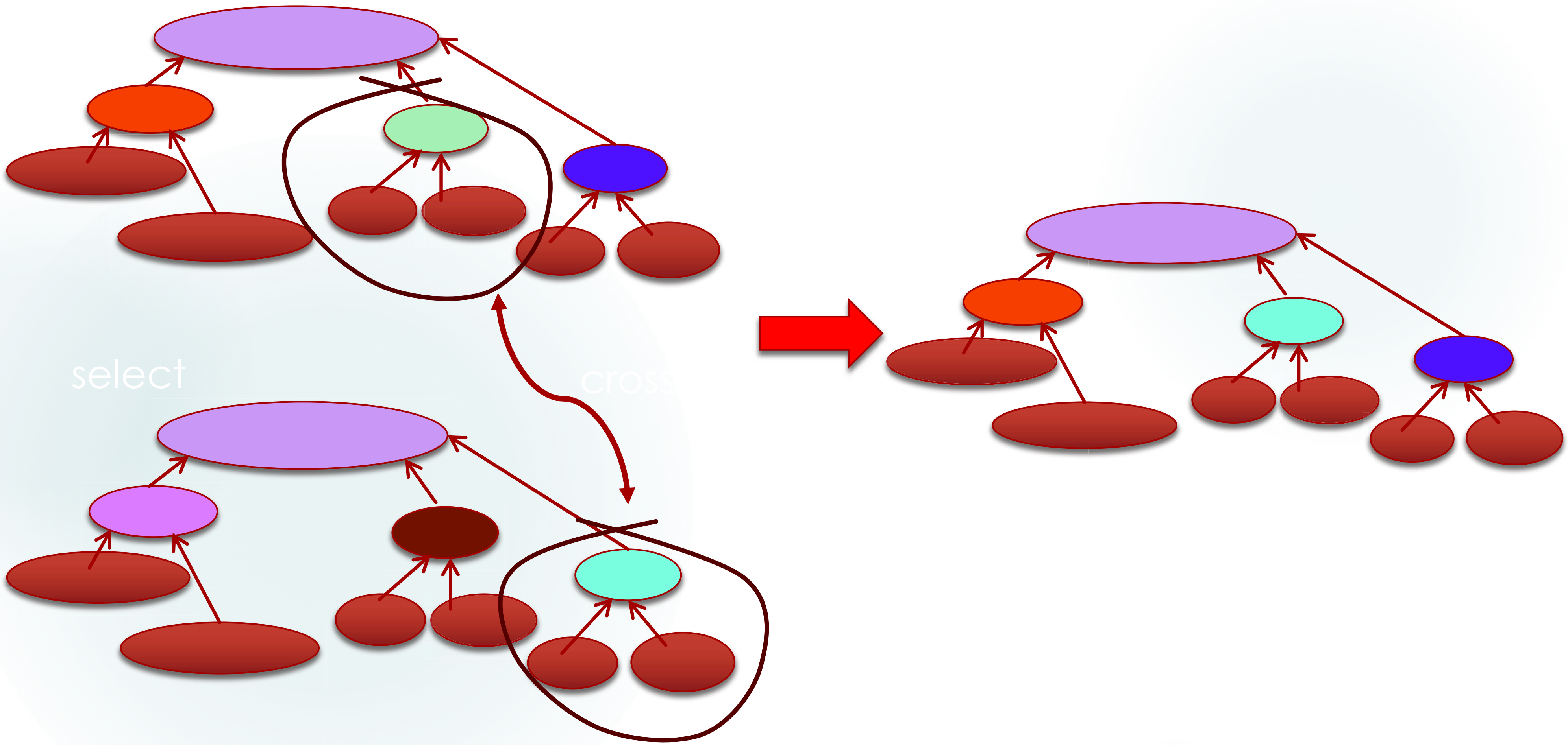


Evolve action selection rules

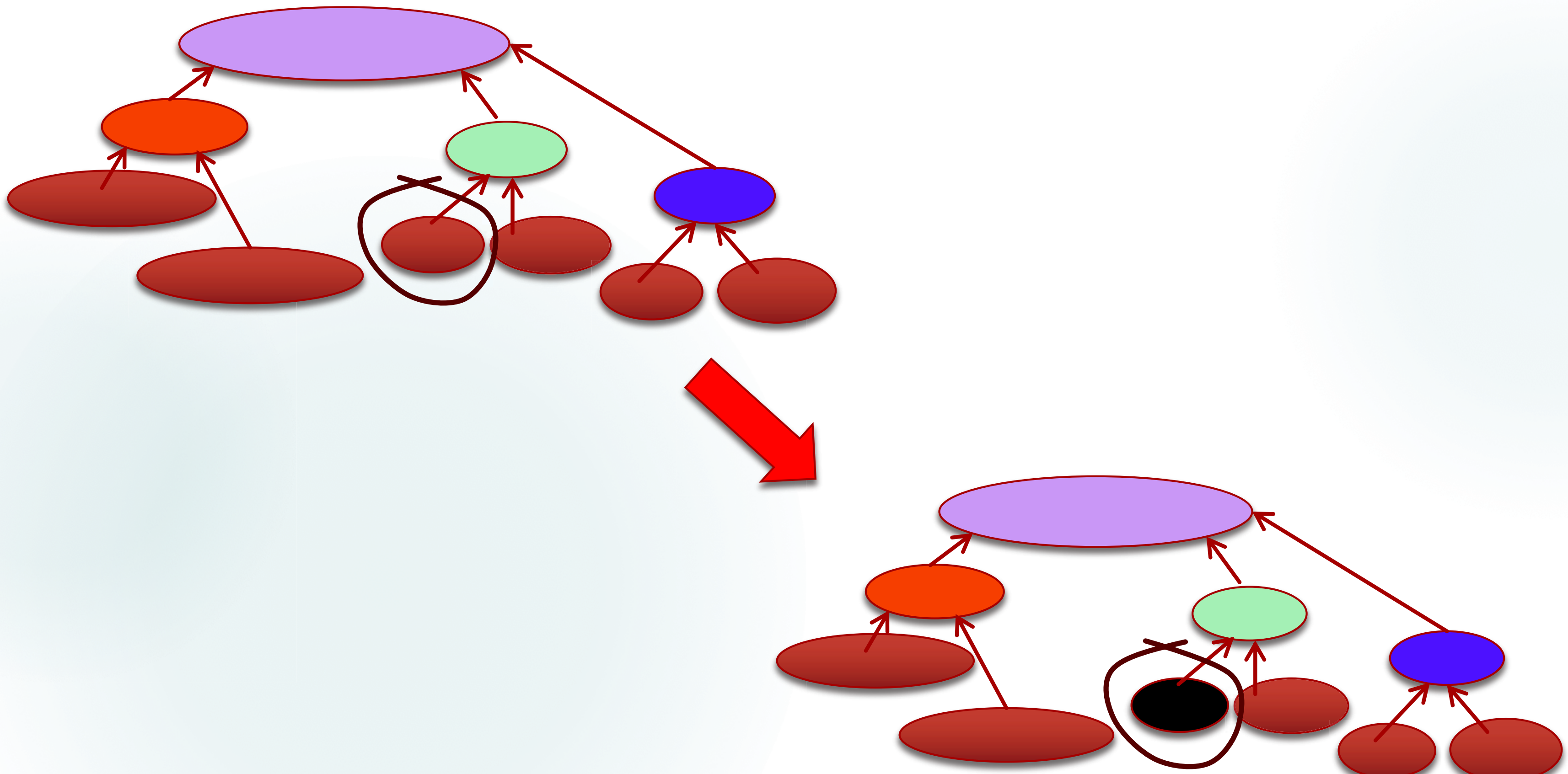
Action selection rules

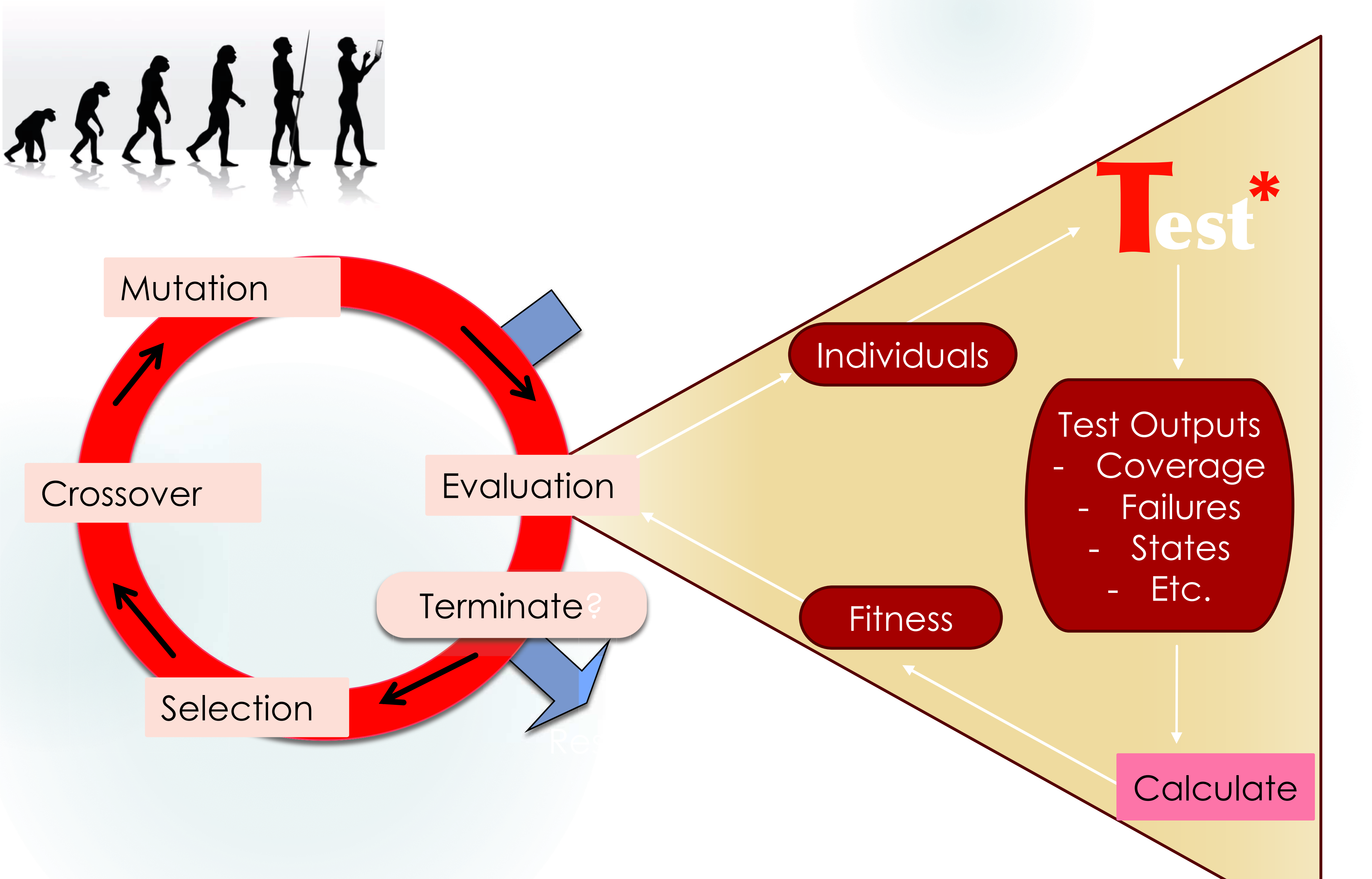


Crossover



Mutation





Evolve action selection rules

TESTAR towards 2025

- Let the testing tool learn itself how to test!
  - Use different machine learning algorithms (action selection/oracles)
  - Define more surrogate measures
- Learn from what the tool tests
  - Show that surrogate measures work
  - Relate them to (type of) failures
  - Extract models to aid exploratory testing
  - Improve visualisation
- More formal testing theory
  - Know better whether we have done well
- Reduce the human oracle cost:
  - Automate as much as posible all other test tasks
  - Make it as easy as possible for the tester

TESTAR Training @ TNO

- 15 and 16th of May 2018
- TNO in Groningen
- Training, hands-on and helpdesk!
- Interested?
- Send me an email.

axini

TESTWERK ICT

TNO innovation for life

TESTOMAT PROJECT The Next Level of Test Automation

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

TESTOMAT PROJECT
The Next Level of Test Automation

Test*

- **email**: info@testar.org
- **web**: http://www.testar.org/
- LIKE IT on : **http://facebook.com/tool.testar**
- **telephone/whatsapp**: +34 690 917 971

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

PROS
Centro de Investigación en Métodos de Producción de Software