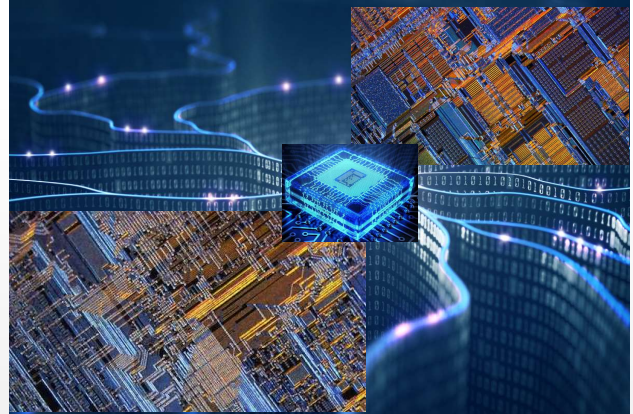


# ML/AI in HW Verification

Dr. Monica Farkash

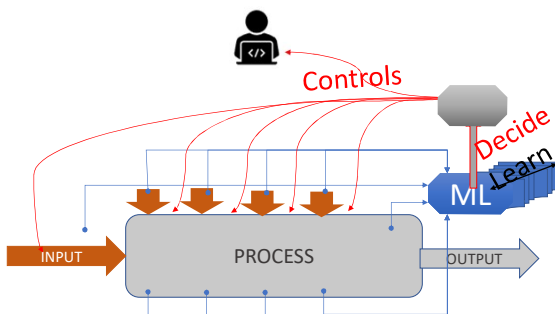


EE 382M-11, Verification of Digital Systems, Spring 2020  
 Department of Electrical and Computer Engineering  
 The University of Texas at Austin

1



. Use ML/AI to **IMPROVE** VERIFICATION



Phase 1 – Learn

- What's going on
- What control changes would be pertinent and available to achieve something useful

Phase 2 – Use (Verification Flow Integration)

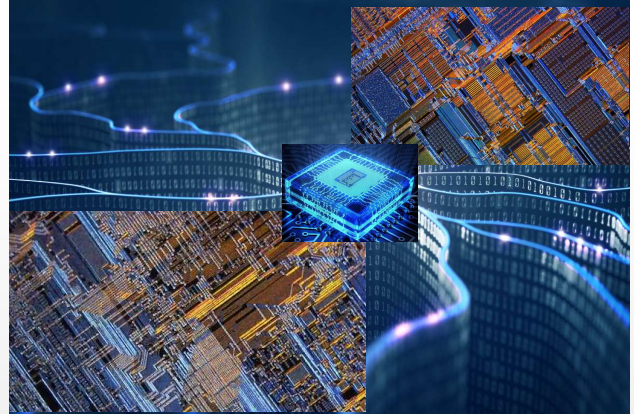
- Based on what you learned write a piece of SW that implements the decisions in real life ( production)

2

# ML/AI in HW Verification

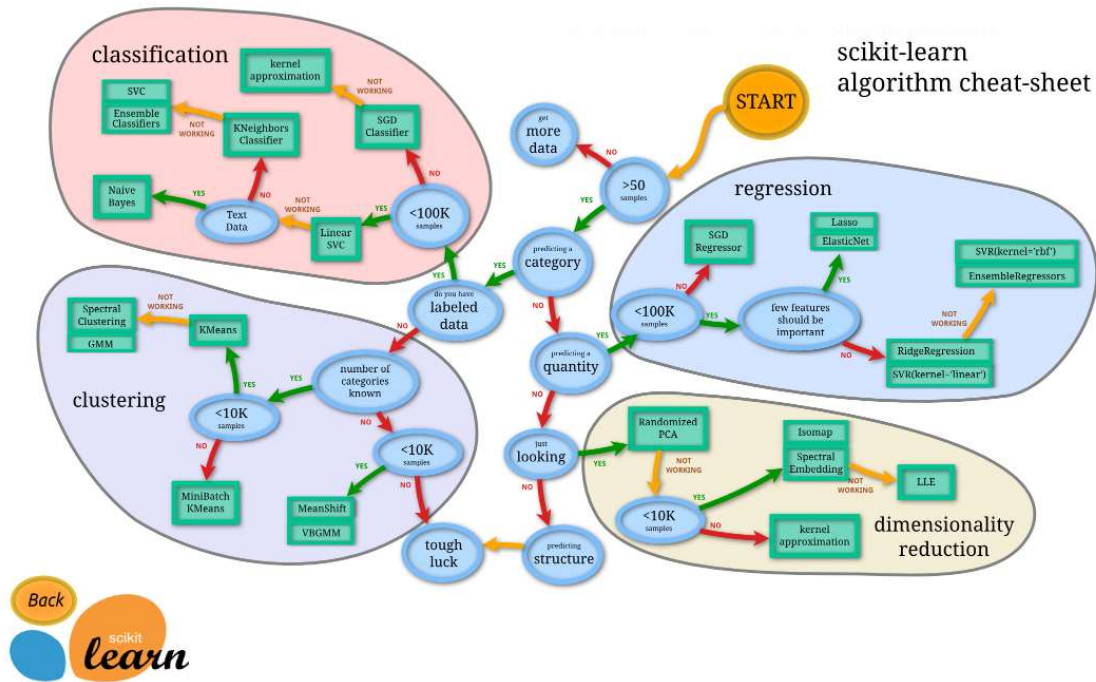
## CONTENTS

1. Practical ML info
2. An Example
3. General View
4. More Examples



EE 382M-11, Verification of Digital Systems, Spring 2020  
 Department of Electrical and Computer Engineering  
 The University of Texas at Austin

3

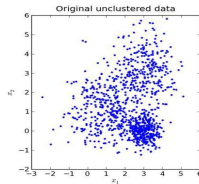
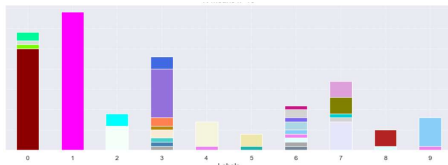
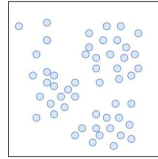
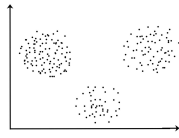


<https://peekaboo-vision.blogspot.com/2013/01/machine-learning-cheat-sheet-for-scikit.html>  
[https://scikit-learn.org/stable/tutorial/machine\\_learning\\_map/index.html#ml-map](https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html#ml-map)

4

# Clustering

- "Group" the records according to how "similar" they are
- **Unsupervised** ( no labels )
- "Guess" number of clusters (groups)



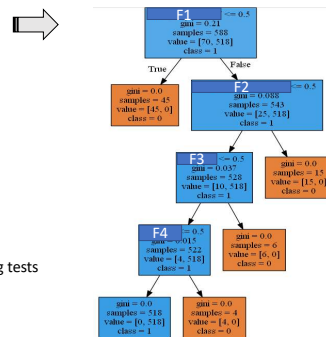
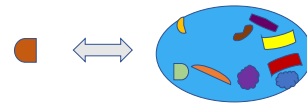
5

# Classification

**Supervised learning** : You "learn" from labeled examples

**Classification** :  
The problem of "guessing" to which class it belongs, based on features and past processed examples

**Example:**  
After failing tests are manually labeled as the bug that triggered the failure  
Use classification to learn what differentiates tests that fail due to Bug A from all the other failing tests  
Here Classification is used not for future identification but for feature ID.



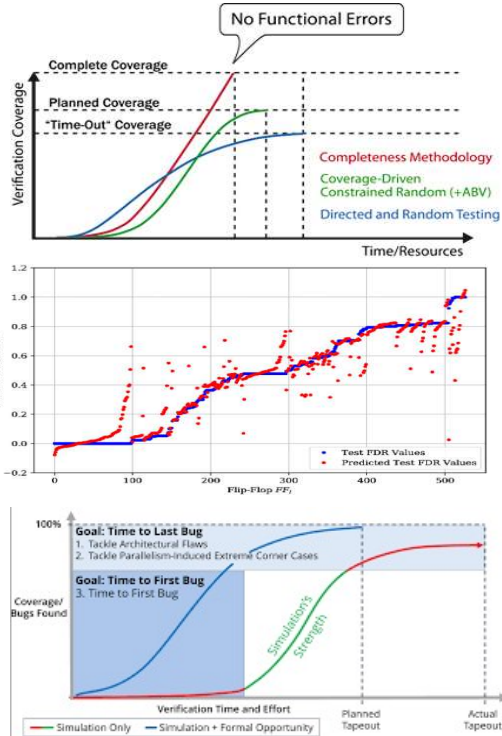
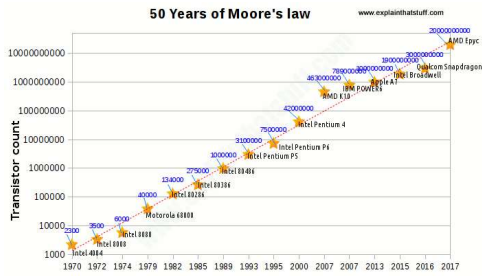
6

# Regression

- Find a *function* that explains the examples (values)
- use it to predict

### Example:

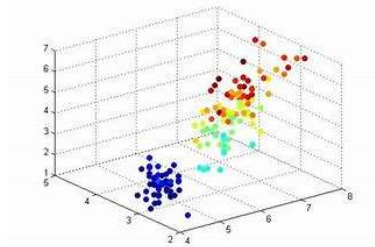
- Guess time needed for coverage



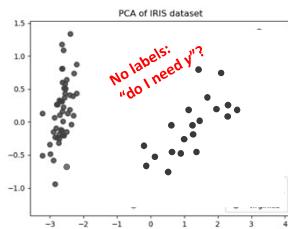
7

# Dimensionality Reduction

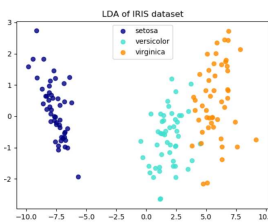
- Variables = dimensions ( 2D, 3D, nD spaces)
- Reduce # variables



Principal Component Analysis



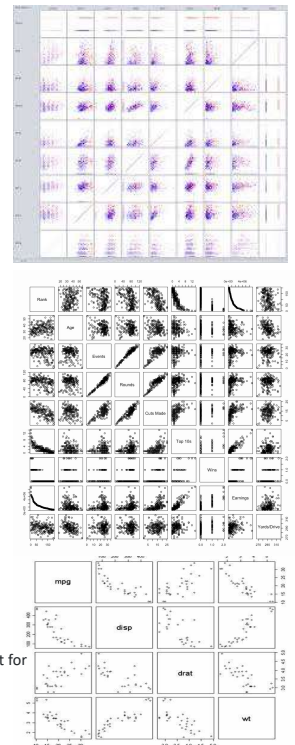
Linear Discriminative Component



PCA decompose a multivariate dataset in a set of successive orthogonal components that explain a maximum amount of the variance.

LDA tries to identify attributes that account for the most variance between classes.

<https://www.cs.waikato.ac.nz/ml/weka/>



8

# Neural Network

## “Black Magic Box”

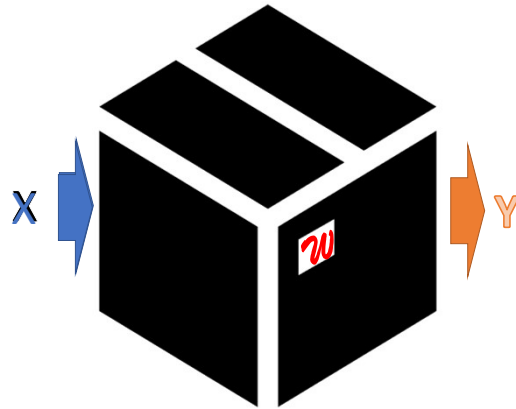
- Input vector  $X$
- Output vector  $Y$
- *guess* nonlinear function

## Learning:

- Provide  $(X,Y)$  and have the box learn internal variables such that for a given  $X$  it has as output the respective  $Y$

## Use:

- Provide new  $X$  and read the  $Y$



Keras: <https://keras.io/>

9

# ML/AI in HW Verification

## CONTENTS

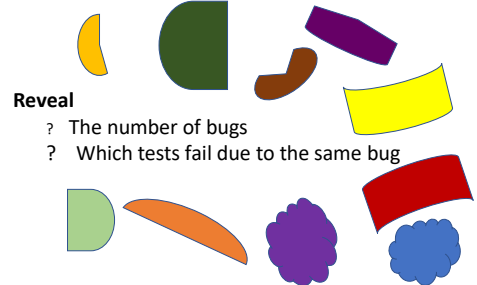
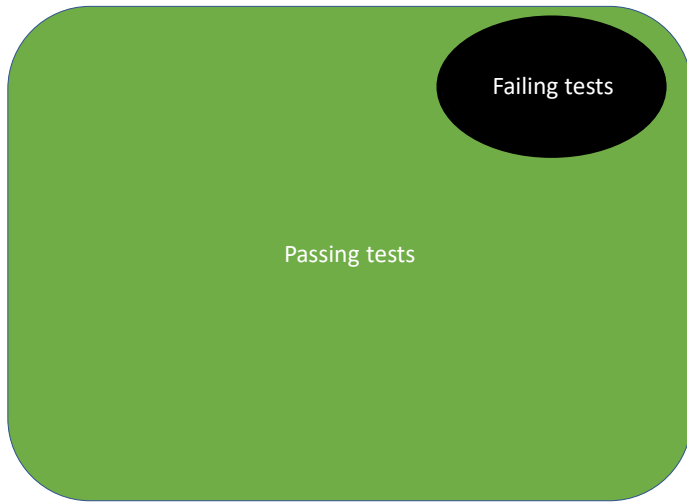
1. Practical ML info
2. An Example
3. General View
4. More Examples



EE 382M-11, Verification of Digital Systems, Spring 2020  
 Department of Electrical and Computer Engineering  
 The University of Texas at Austin

10

# Triage



**Reveal**

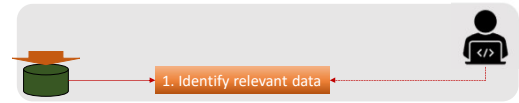
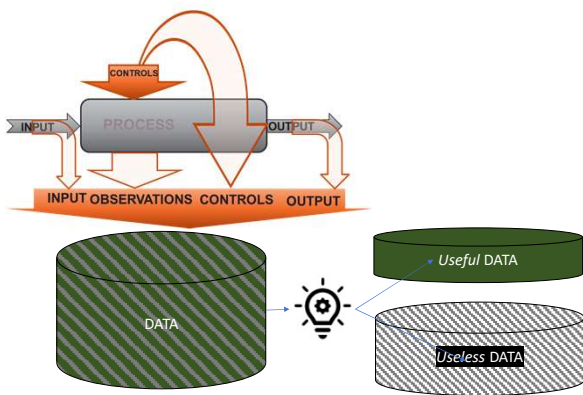
- ? The number of bugs
- ? Which tests fail due to the same bug

**Why?**

- => NO REDUNDANT EFFORT !
- => How much effort needed ? (#bugs)
- => Which test to debug first :
  - As in impactful bug
  - As in short test for that bug
- => Hints per bug ?
- => Who should debug what ? (skills)

11

# Triage: Potential Data



**Input data:**

- Test contents (if available)
- Test template (*scenario*)

**Observations:**

- Assertions /cover points
- Events (log files)

**Controls: (testbenches?)**

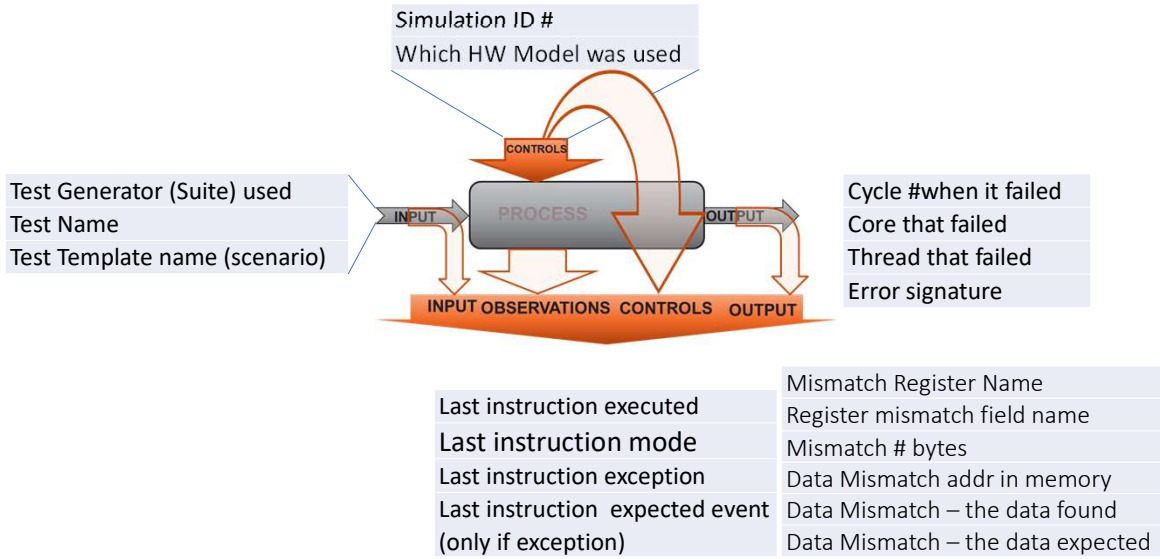
- Warm load, Config modes

**Output:**

- Pass/Fail
- Coverage ( toggle/line/fct. )

12

# Example



13

- Test Generator (Suite) used
- Test Name
- Test Template name (scenario)
- Simulation ID #
- Which HW Model was used
- Last instruction executed
- Last instruction mode
- Last instruction exception
- Last instruction expected event (only if exception)
- Cycle #when it failed
- Core that failed
- Thread that failed
- Error signature**
- Mismatch Register Name
- Register mismatch field name**
- Mismatch # bytes
- Data Mismatch addr in memory
- Data Mismatch – the data found
- Data Mismatch – the data expected

## Data Processing: Exclude & Change

- AD: OP: MOV [EAX],0x000C0045 → MOVE
- Mode before / Mode after ( same or different ) . ex. PmSVG64 , Cm32, ...
- Exception ID – hierarchical info

→  $x < th1; th1 < x < th2; x > th2$

(more than one register mismatch)

R1	R2	R3	R4	R5	R6
1	0	0	1	1	0

Mainly common sense and expert knowledge

14

## Similarity

test	F1	F2	F3	F4	F5	F6	F7
T1	A	X	R	P	W	L	9184610
T2	A	Y	R	P	U	0	14
T3	b	X	R	T	V	0	102

Group them according to how similar they are.

Simple idea: use *distance* function = # features same values

T1, T2 : F1, F3, F4

	F1	F2	F3	F4	F5	F6	F7
T1	A	X	R	P	W	L	9184610
T2	A	Y	R	P	U	0	14
T3	b	X	R	T	V	0	102

T1, T3: F2, F3

	F1	F2	F3	F4	F5	F6	F7
T1	A	X	R	P	W	L	9184610
T2	A	Y	R	P	U	0	14
T3	b	X	R	T	V	0	102

T2, T3: F3, F6, "F7" (binning)

	F1	F2	F3	F4	F5	F6	F7
T1	A	X	R	P	W	L	9184610
T2	A	Y	R	P	U	0	14
T3	b	X	R	T	V	0	102

15

## Why Data Pre-processing

Grouping according to similarity, data can *help*, *harm* or be just *useless*.

test	F1	F2	F3	F4	F5	F6	F7
T1	A	X	R	P	W	L	9184610
T2	A	Y	R	P	U	0	14
T3	b	X	R	T	V	0	102

*Useless:*

- F3: all tests have same value. F3 doesn't help and it doesn't hurt. It simply doesn't provide information (and clogs the system).
- F5: each test has a different value. F5 doesn't help and doesn't hurt. It simply doesn't provide information.

*Help:*

- F1: T1 and T2 "score" a similarity. T3 is not similar to T1, nor T2
  - F2: T2 and T3 "score" a similarity. T2 doesn't.
  - F4: Again, T1 and T2 "score" a similarity.
- ⇒ {T1, T2} score more similar features than {T1, T3} or {T2, T3} **therefore (!)** it is more likely for T1 and T2 to fail due to the same bug

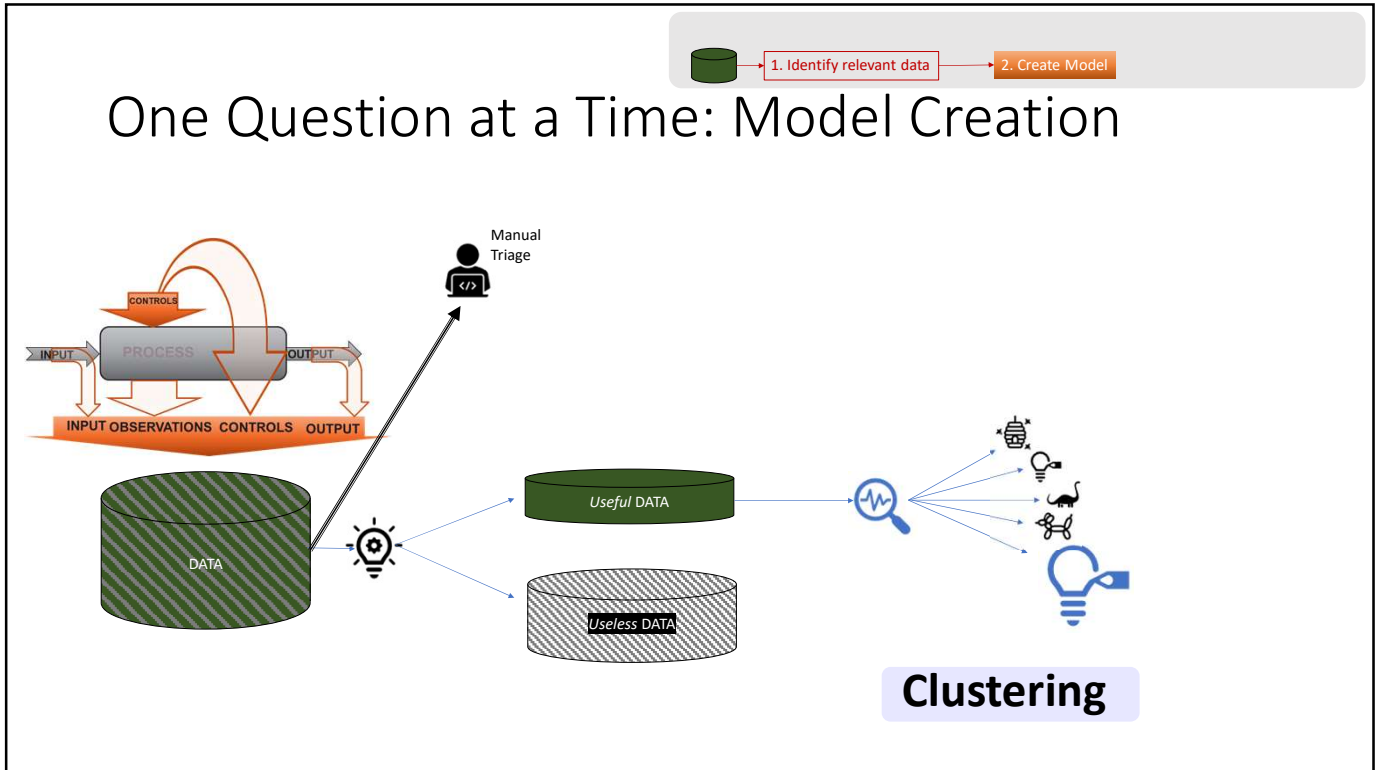
*Harm:*

- F6: the lack of information ( NaN ) is perceived as '0' here and it counts as if T2 and T3 'score' a similarity.
- F7: randomly generated data is generally binned and because T2 and T3 have small values, they will end up in the same "bin" and here is another wrongly perceived similarity between T2 and T3. (random is not uniformly distributed, and even if, binning would still create information from where there's none)

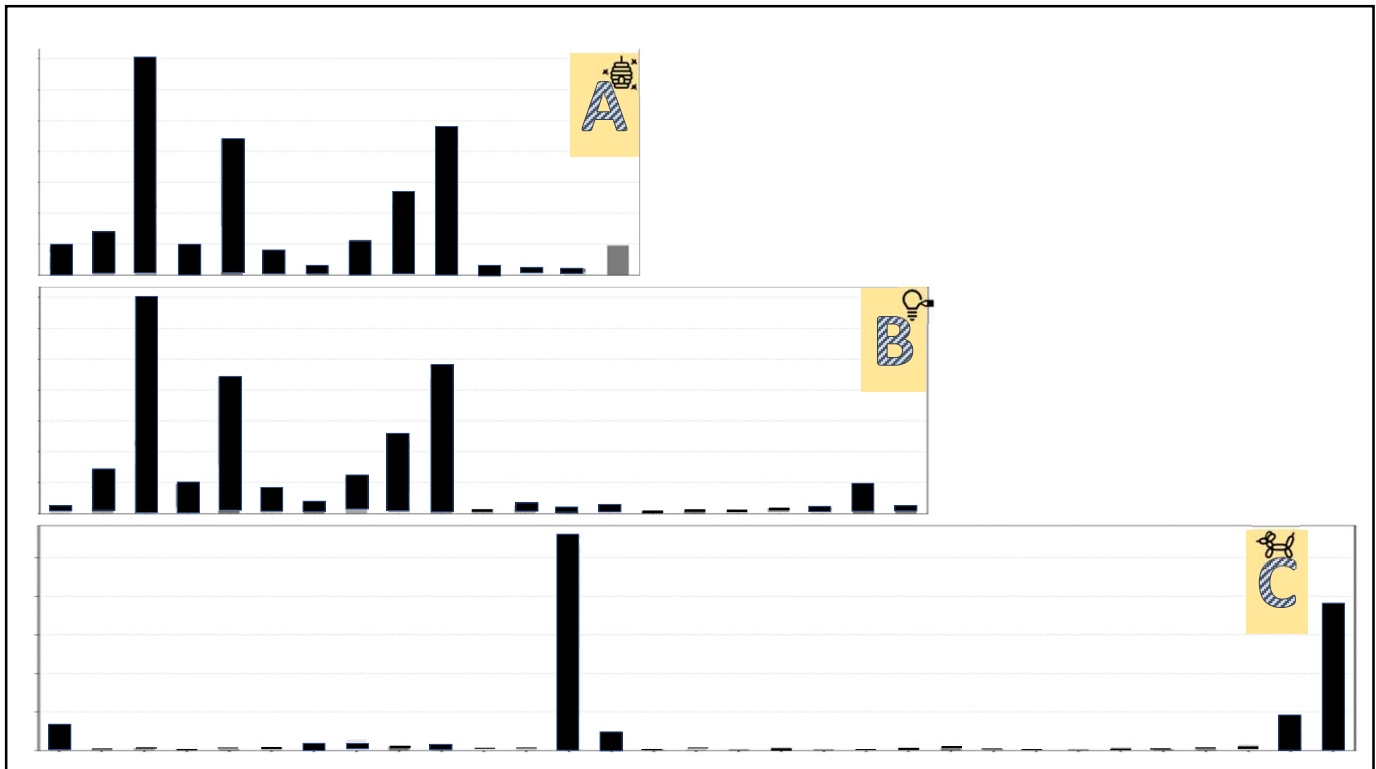
The two harmful => misinformation => {T2, T3} seem closer to each other ( wrong!)

16

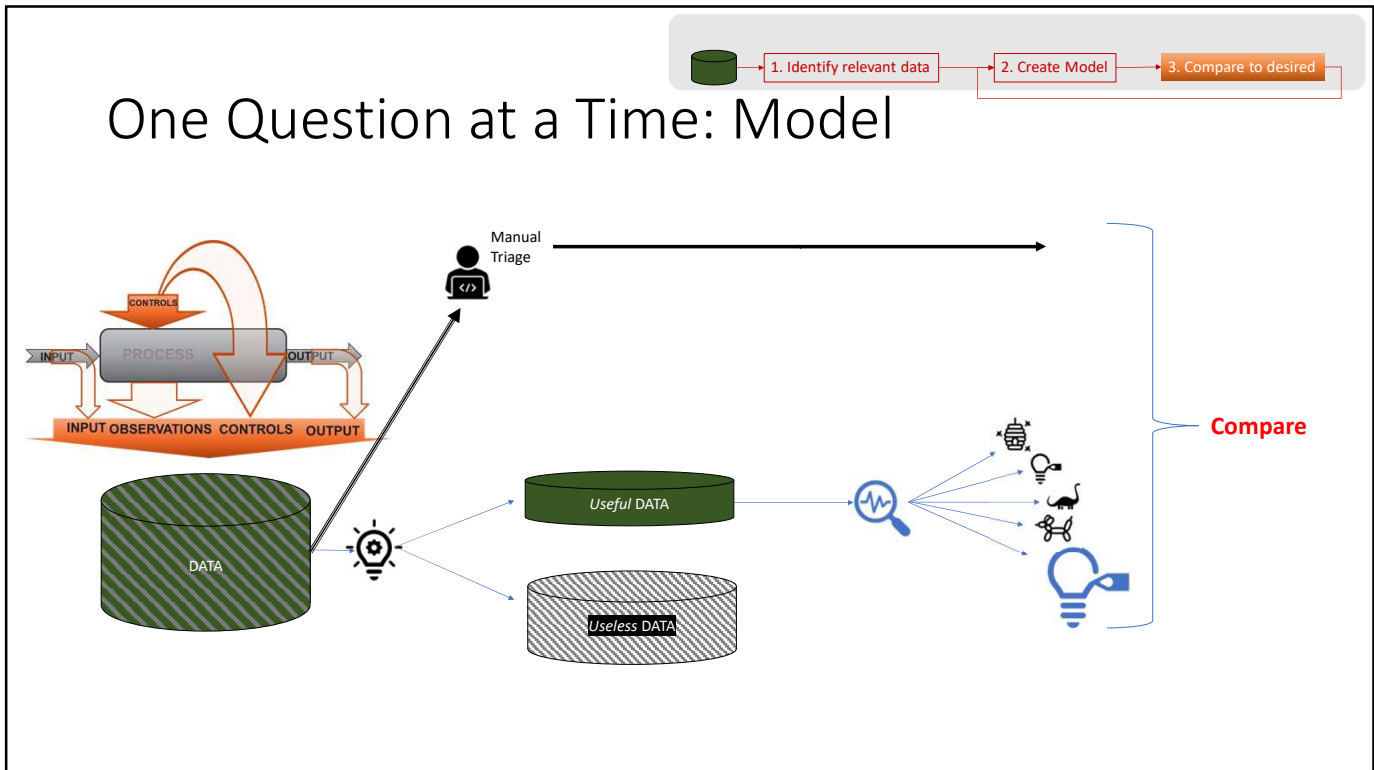




17



18



19

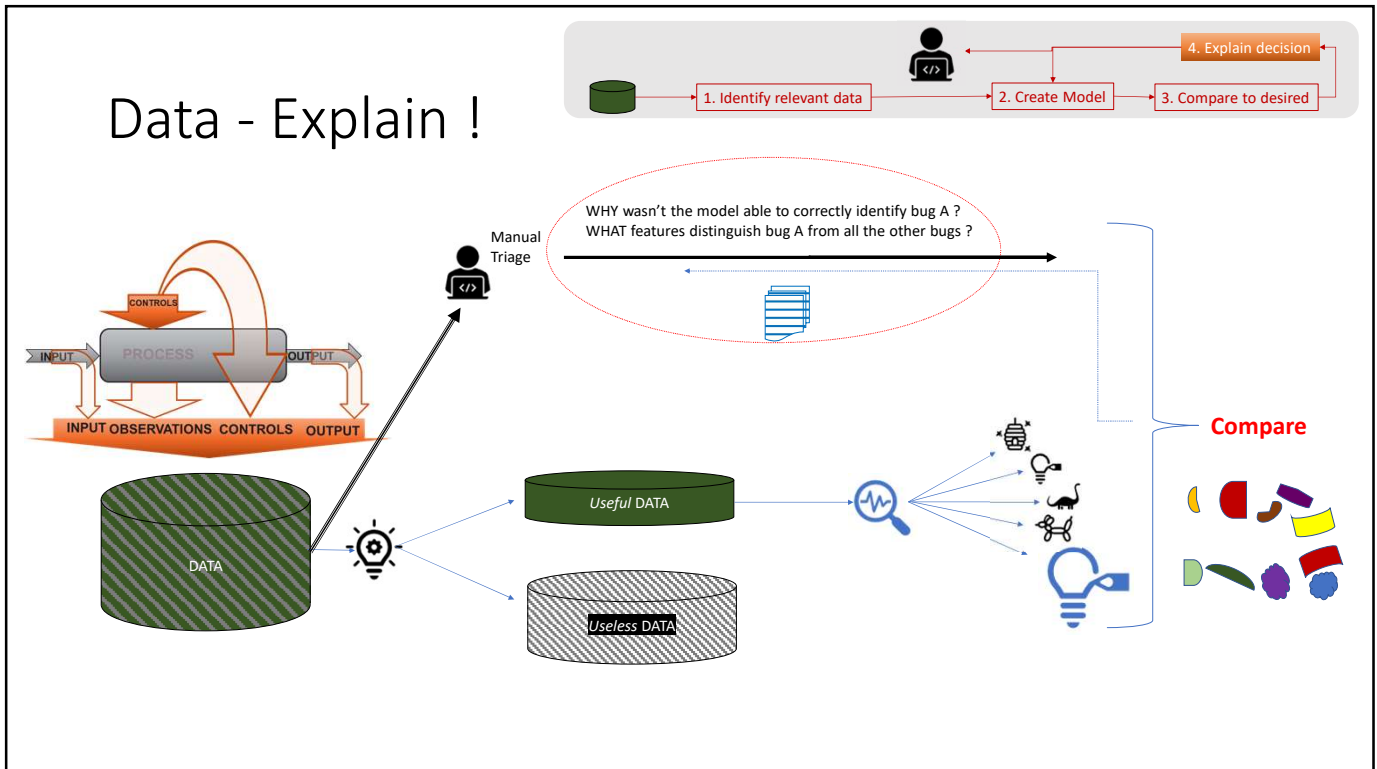
## Compare Criteria Definition

1. **Number groups = number real distinct bugs**  
 Allows for planning => SHIFT LEFT  
 Ideal # Groups = # Real Bugs
2. One group/engineer debugs one test/group => **distinct bugs**  
 Cost of Redundancy  
 more people – higher danger or redundancy
3. Identifying all bugs that are **main hitters** (lots of failures)  
 Cost of resources, time ...=> SHIFT LEFT  
 Less groups – higher danger of missing a big hitter

20



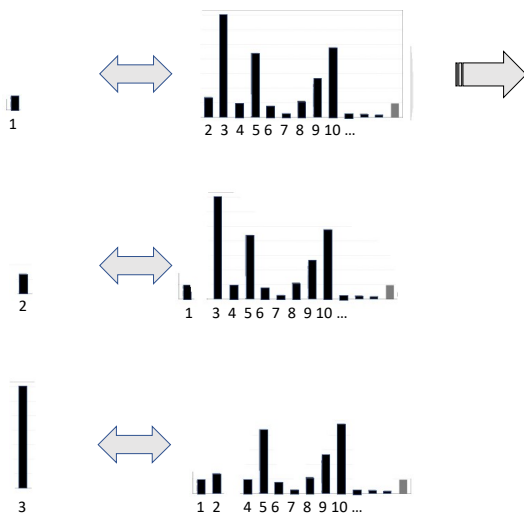
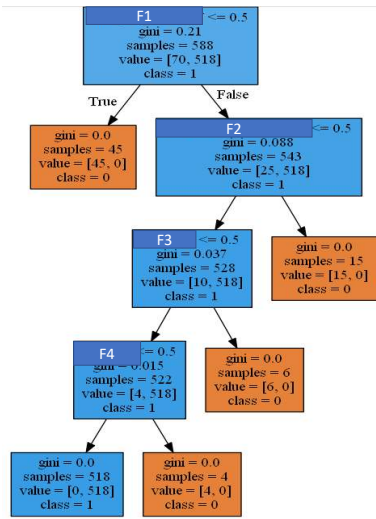
21



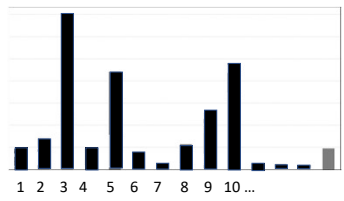
22

# Data - Explain !

WHY wasn't the model able to correctly identify bug A ?  
WHAT features distinguish bug A from all the other bugs ?

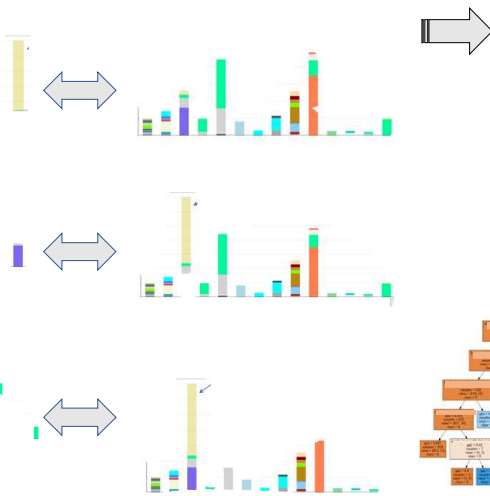
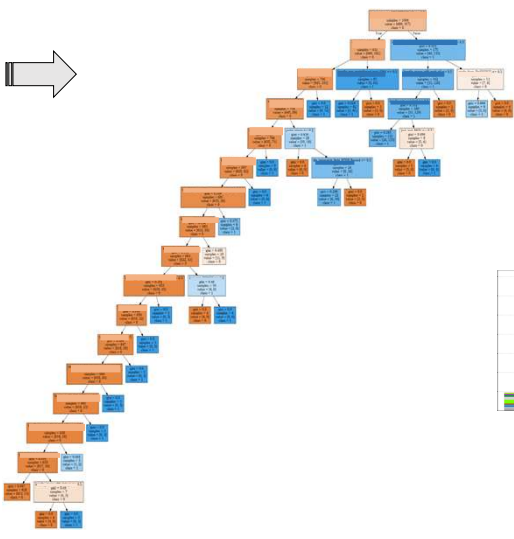
**Explain**



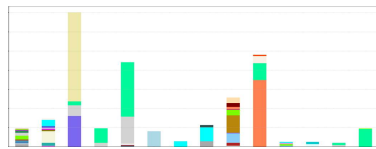
23

# Data - Explain !

WHY wasn't the model able to correctly identify bug A ?  
WHAT features distinguish bug A from all the other bugs ?

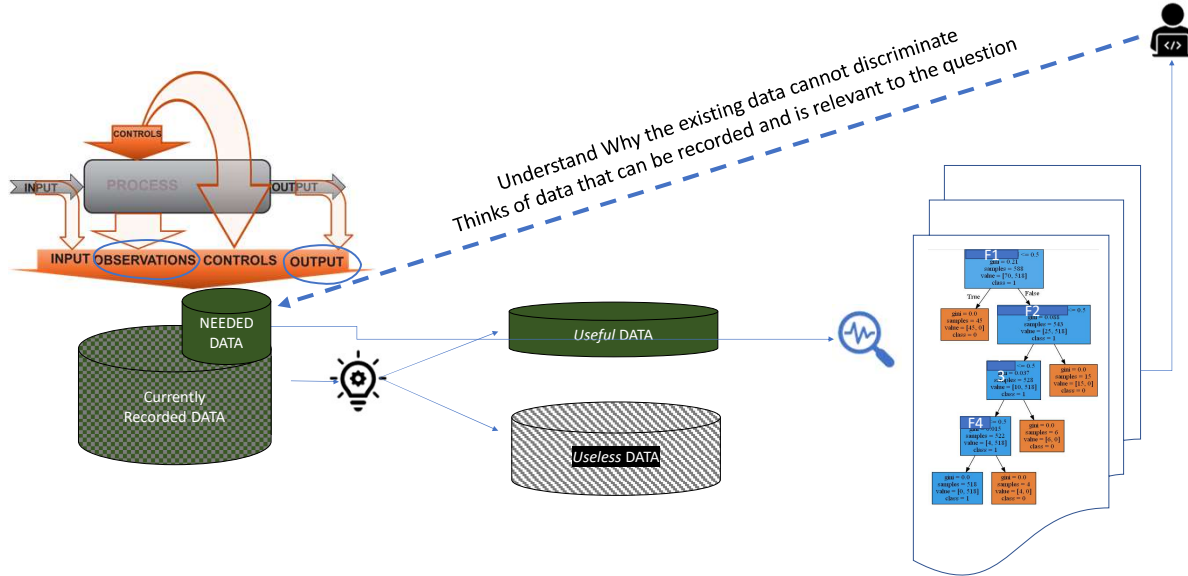



**Explain**



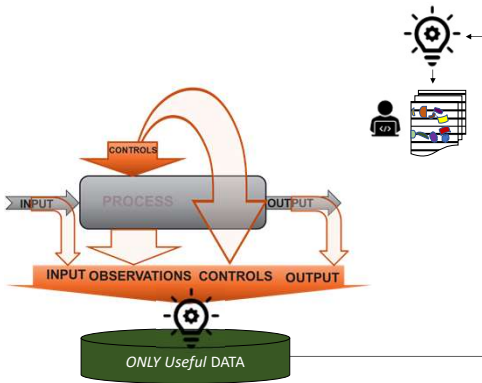
24

# Data - Explain !



25

# Flow Implementation



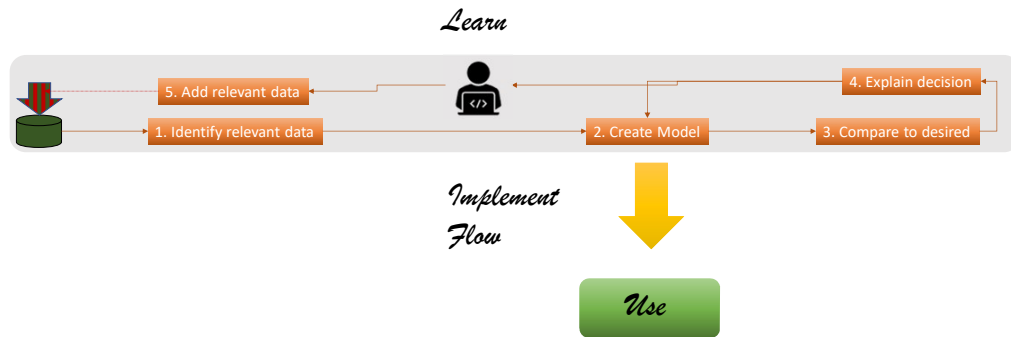
**Flow:**

- Integrated solution
- Automatic everything that's possible
- Triggered by events, time, ..
- Graphical view of what's going on
- Records history ( quality )

Finally you provided a solution that can be used

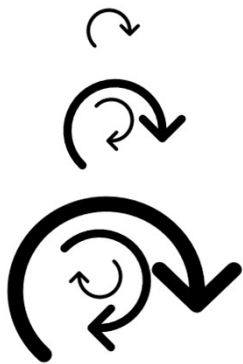
26

## Use



27

## Not Working – Now What ?



1. Model : Use something else, not Clustering

2. Data : Work with them to get more relevant data

3. Goal : Change the “Question” to something attainable and still useful

28

### 3. Goal

## Change it to something attainable

#### 1. Triage by initial definition

- Divide the failed tests into groups such that each group of tests failed due to the same **unique** bug.

#### 2. Min Overlap

- Divide the failing tests into as many groups as available verification engineers, with the condition that the likelihood of two of them debugging the same bug is minimized.

#### 3. RTL vs. non-RTL

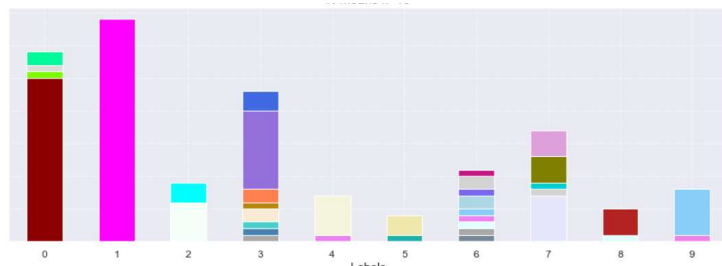
- **Learn** from the past what combination of features points to RTL respectively non-RTL bugs and use it in the next iteration. Update rules every time you get a new batch.

29

### 2.Data

## Improve its *quality*

Work on getting more relevant data



Explain to user

Use other means to show which data is relevant and why

Example: Decision tree to explain clustering

30

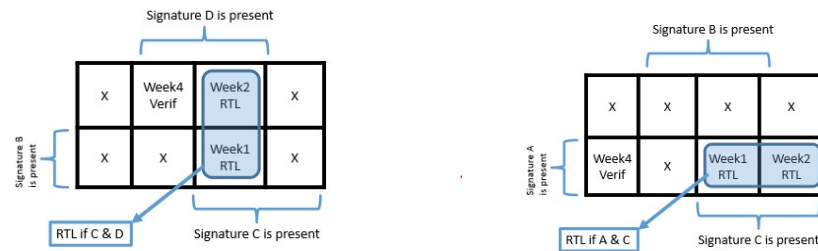
# 1. Model

## Use something else

### Clustering -> Rules Learning

- Try to learn the combination of features that explain type of bugs
- Unsupervised (Clustering) -> **Supervised**
- Reduced the Goal to ID RTL from non-RTL failures

### Karnaugh map



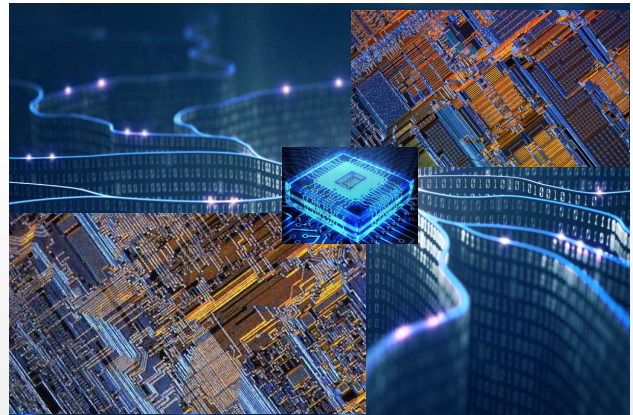
Expediting Design Bug Discovery in Regressions of x86 processors Using Machine Learning" Ahmed Wahba, Justin Hohnerlein, Farhan Rahman Advanced Micro Devices, Inc. (AMD)

31

## ML/AI in HW Verification

### CONTENTS

1. Practical ML info
2. An Example
3. General View
4. More Examples



EE 382M-11, Verification of Digital Systems, Spring 2020  
 Department of Electrical and Computer Engineering  
 The University of Texas at Austin

32



### What types of problems we can solve

Functional verification

- Which (existing) tests to run to achieve coverage faster
- What type of test to create to verify (given/all) area

Debugging

- Which tests failed due to the same bug
- How many distinct bugs are in there
- What characteristics do the tests failing (presumably same bug) have
- How is the behavior of one failing test different than the behavior of similar passing tests

Performance

- Which benchmark to run (as in keep/replace/remove)
- What configuration (as in modes) is optimal for performance

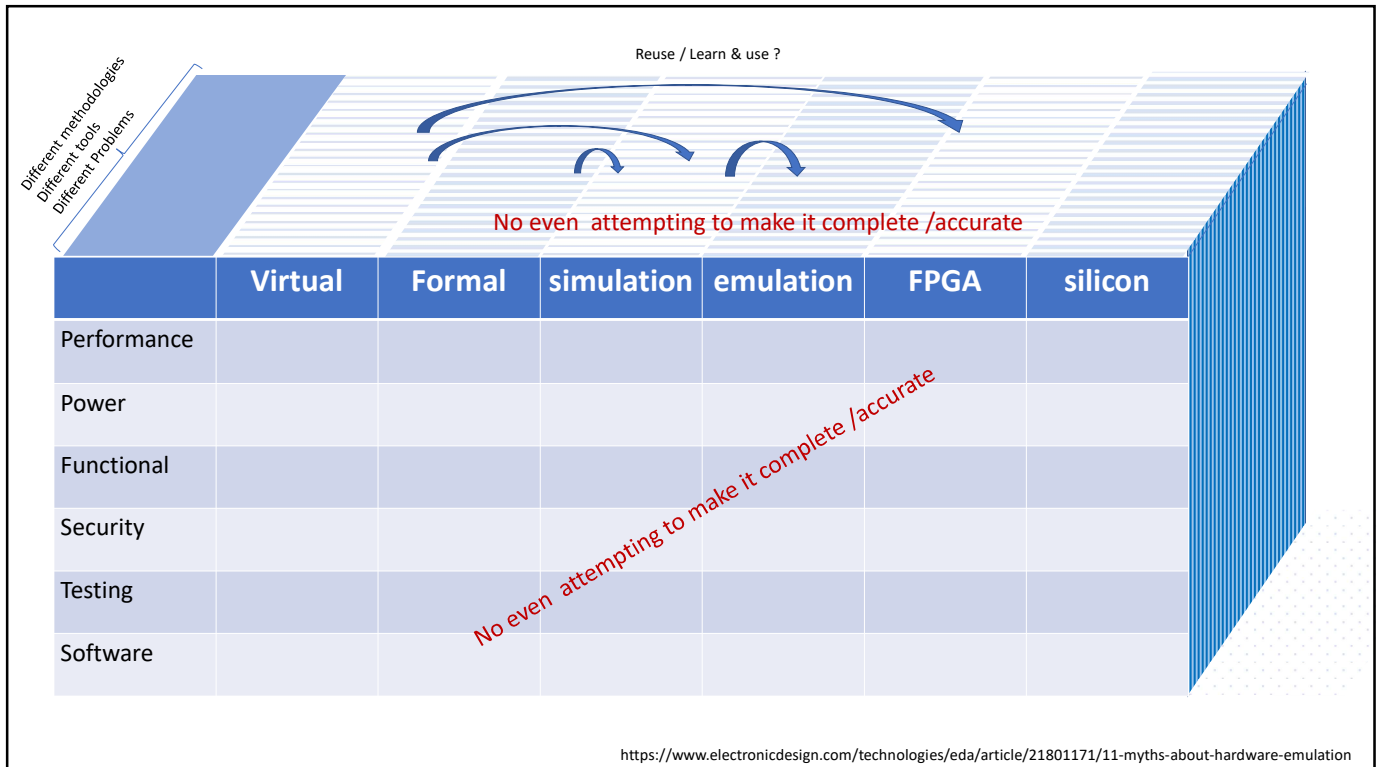
Management

- How close am I to finish ? (coverage curve)
- Who inserted the bug, in which version of which file
- Which files should be rewritten

Resources

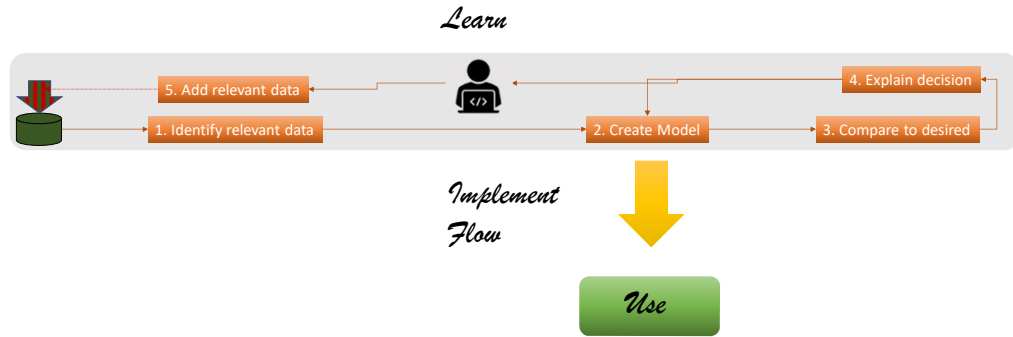
- What resource (processing power + memory) is needed by this run (simulation – run attributes)
- Which engine / heuristics is best (as in formal)

33



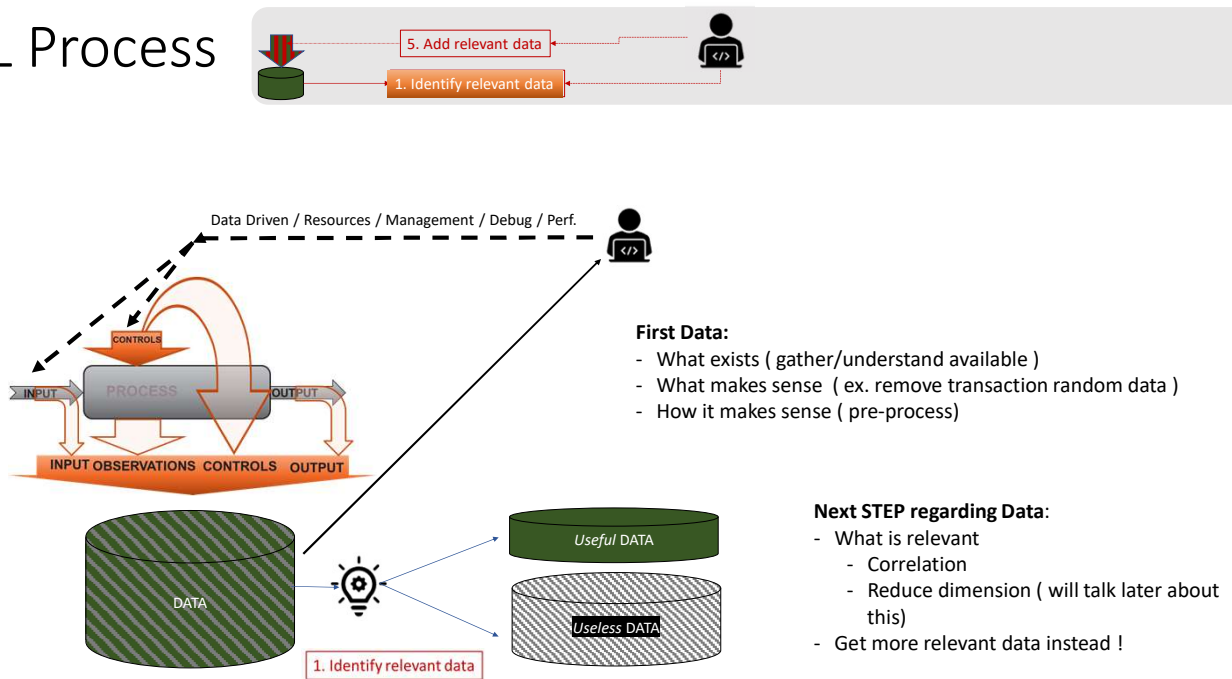
34

# ML Process



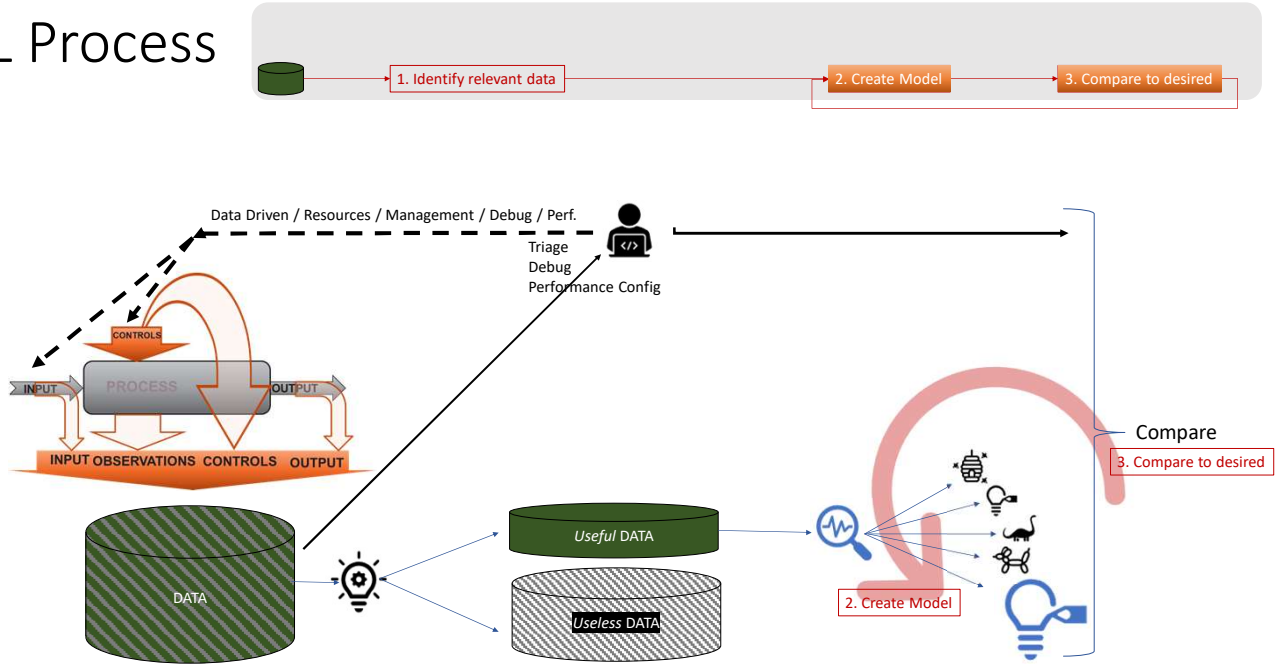
35

# ML Process



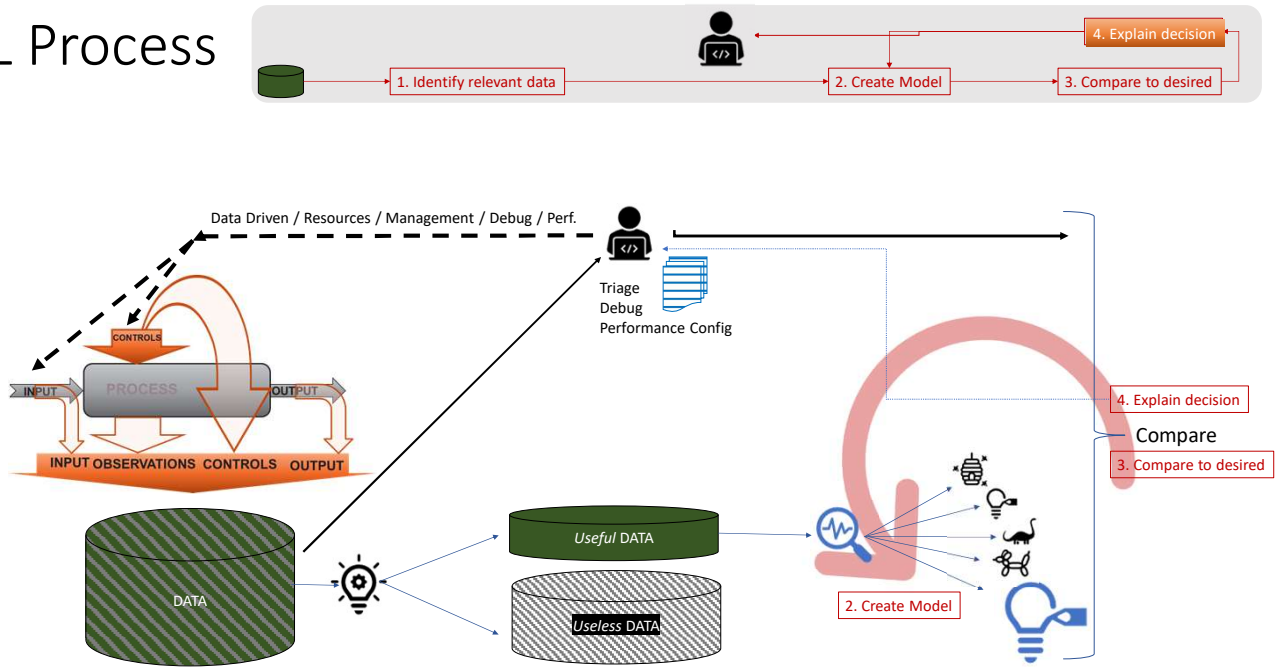
36

# ML Process



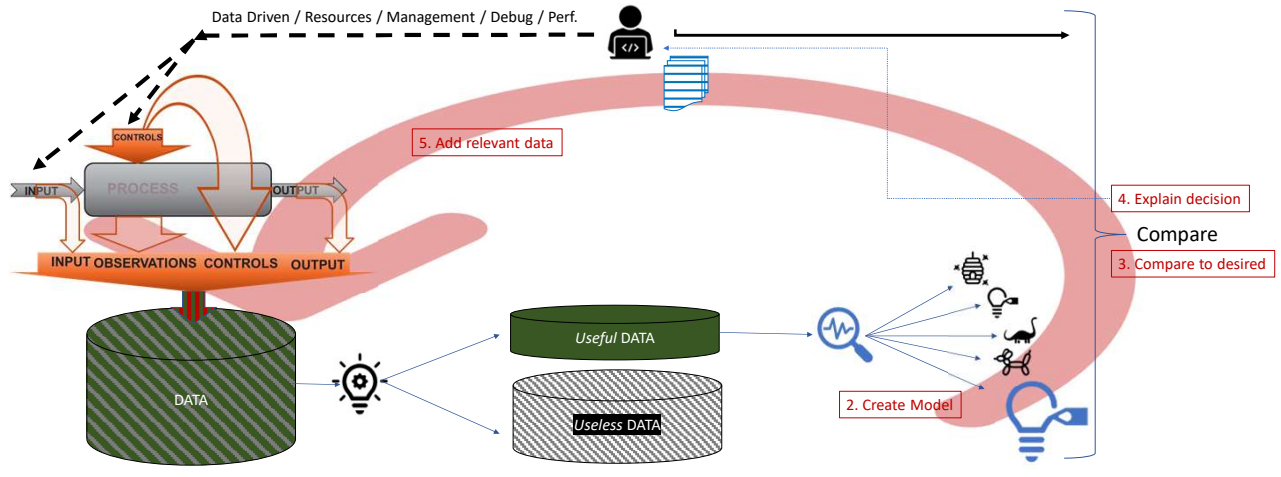
37

# ML Process



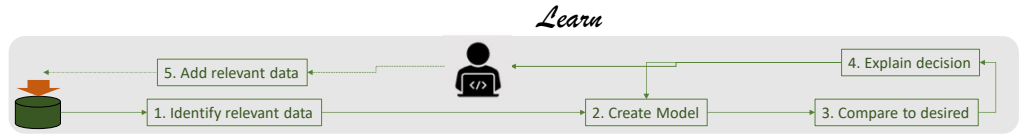
38

# ML Process

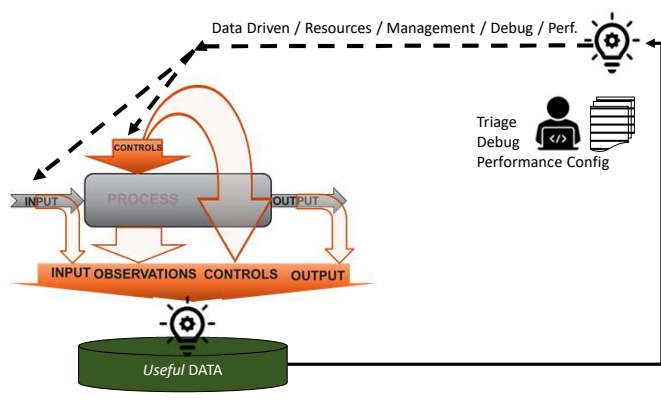


39

# ML Process



*Implement Flow*

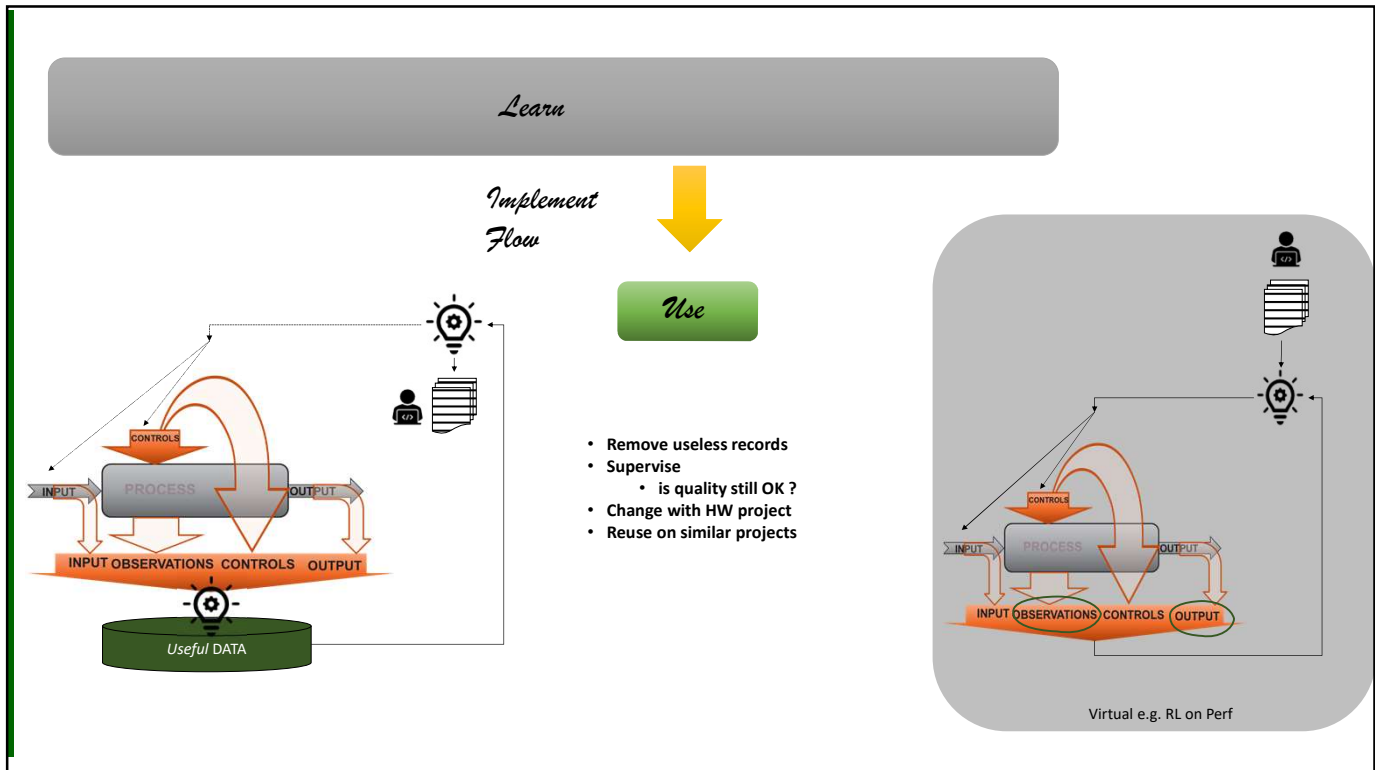


- Flow:**
- Integrated solution
  - Automatic everything that's possible
  - Triggered by events, time, ...
  - Graphical view of what's going on
  - Records history ( quality )

**We leave behind a FLOW !**

- Not a model
- Not an analysis
- Not a tool

40



41

## Common Mistakes

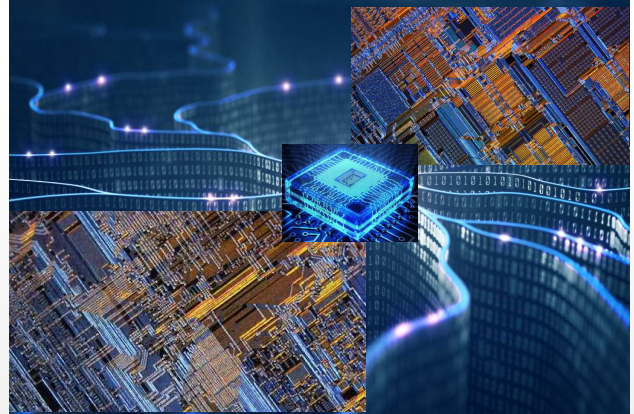
- **Data:**
  - Not pondering on data and deciding if useful or not !
  - Accept whatever data they give you as given instead of working with them to get the data that would be relevant !
- **Goal:**
  - Not challenging and changing (adapting) the goal if/when needed
- **Model:**
  - Not explaining the model (as in which features, why ... )
  - Choosing uselessly complex solutions ! Consider ROI !
  - Believing that the task stops with proving the model
- **Flow:**
  - Not designing a full solution ( as in where the data is stored, what triggers the activity, directory structure, resources, processing power, parallel runs ... )
  - Not providing *visuals* over the process ( graphics, web pages, ...) – high impact, useful, appreciated
  - Not implementing quality measurements & keeping records on usage

42

# ML/AI in HW Verification

## CONTENTS

1. Practical ML info
2. An Example
3. General View
4. More Examples

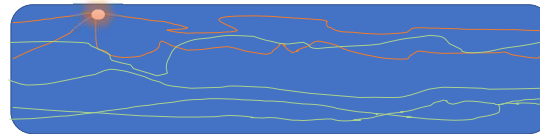


EE 382M-11, Verification of Digital Systems, Spring 2020  
 Department of Electrical and Computer Engineering  
 The University of Texas at Austin

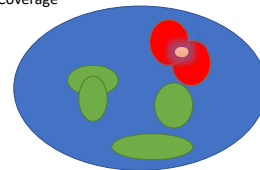
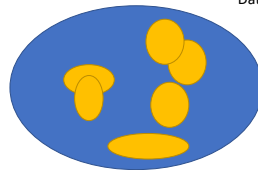
43

# Debug

Data: Paths

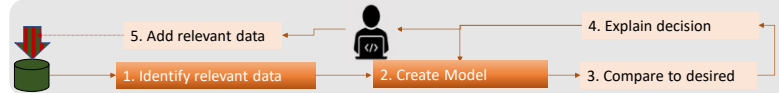


Data: Coverage



44

## HW Debug



- Type of data that is being used:
  - Functional coverage
  - Configuration (modes) coverage
  - Graph coverage ( scenario/events coverage )
- Main idea to extract debug hints:
  - set of failing tests compare to (what we learned from) set of passing tests
  - one failing test compare to (what we learned from) set of passing
  - one failing test compared to one passing that is *most similar*

45

## Functional Coverage

### Data:

- Functional Coverage of **Passing** & **Failing** tests

### Model:

- Identify what is prevalent in Failing compared to Passing
- Assume passing tests distribution as 'normal'

### Example:

#### Feature $F_k$

10% of all passing tests hit assertion  $F_k$

80% failing tests hit this assertion

=> might have something to do with this assertion !

**Q: Why not expect 100% of failing test to have that assertion on ?**

**NOT TRIAGE !**  
Triage "sees" failing tests only.

46

## Register Configurations (Modes)

Data:

- **Configuration (Modes)** of passing & failing tests

Model:

- Identify what is prevalent in Failing compared to Passing
- Assume passing tests distribution as 'normal'

Example: identify all the register configurations  $C_k$  that seem to be correlated with the failing tests

Feature  $C_k$

10% of all passing tests have configuration  $C_k$

80% failing tests have configuration  $C_k$

=> might have something to do with this configuration !

47

## Event(s) compare

Data: ( source can be log files )

- Learned (expected) behavior on a bus ( signal patterns / beh rules)
- This **particular** test's behavior compare to expected

Model:

- Identify the difference between **this behavior** and the expected
- The moment behaviors **diverge**

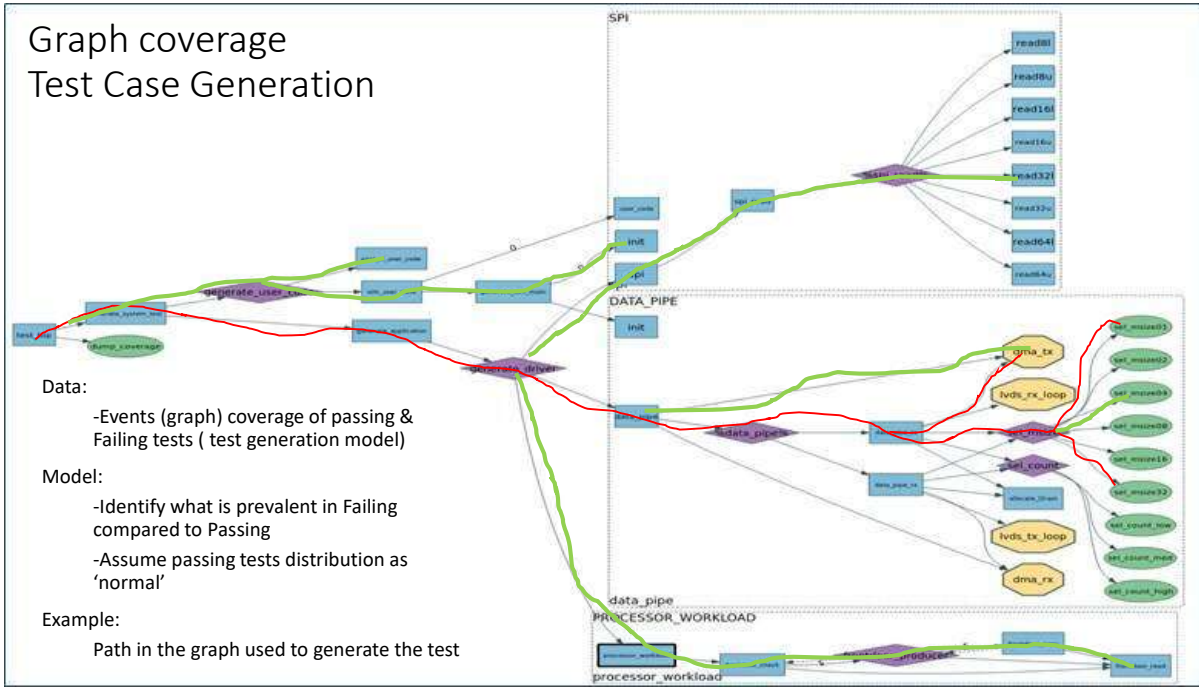
Example:

- All previously passing tests show a delay of at least 5 cycles, and this one, at point "x" has a delay of 4
- I try to rerun the same on a new model, and the behavior changes in a given point
- *Everybody who calls this SW library (API) first calls fct create() then set() then ... and this entity called set() before create()*

48



# Graph coverage Test Case Generation



- Data:**
  - Events (graph) coverage of passing & Failing tests ( test generation model)
- Model:**
  - Identify what is prevalent in Failing compared to Passing
  - Assume passing tests distribution as 'normal'
- Example:**
  - Path in the graph used to generate the test

Breker Systems <https://brekersystems.com/inside-portable-stimulus-concurrency-and-schedules/>

49

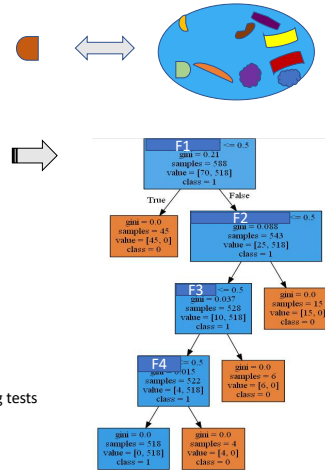


## Classification

**Supervised learning :** You "learn" from labeled examples

**Classification :**  
The problem of "guessing" to which class it belongs, based on features and past processed examples

**Example:**  
After failing tests are manually labeled as the bug that triggered the failure  
Use classification to learn what differentiates tests that fail due to Bug A from all the other failing tests  
Here Classification is used not for future identification but for feature ID.



50

# Metric Driven Verification

Coverage & Other - Driven Verification

51

## Coverage Driven Verification

### Goal:

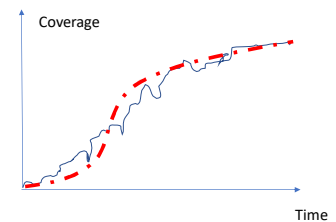
How to achieve same (or higher) coverage in less time with less resources

### Answer:

- Learn what is going on during the process
- Use that information during the process:

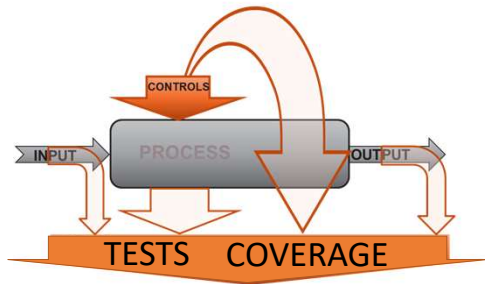
1. REGRESSION TESTS - Rank / prioritize / select

2. CREATING TESTS - Decide which tests to create



52

# 1. REGRESSION TESTS



**Data:** 1. Given data

Tests & Coverage

**Model:** 2. Create Model

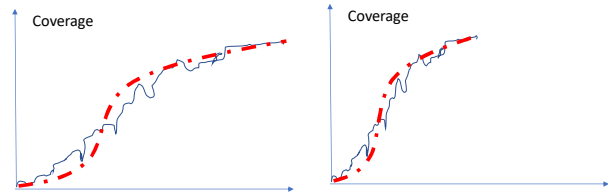
Learn impact of each test separately  $T_i \rightarrow C_i$

**Use:**

Select subset of tests that give the fastest coverage

**Learn:** Coverage is "a function" of tests

**Use:** Run only tests that impact coverage



53

# 1. REGRESSION TESTS

## Problems

### 1. Model

1. Quality depends on test "consistency" regarding coverage

**Testing conditions** are a function of Test, HW & Methodology

- Impossible to re-create *same testing conditions (testbench, random) even if we run the same test*
  - Even if possible: same test changes *path when HW changes*
2. Different chosen coverage metrics point to different views  
functional/ toggle/ line

54

# 1. REGRESSION TESTS

## Solution: Example 1

**Metric:** functional coverage

example: cache eviction achieved, buffer full achieved, lock on address challenged

**Tests:** functionally consistent

- test that seem to achieve similar functional coverage with small HW changes
- generally are tests that target given areas and allow for randomness
- Example: do cache eviction but different index/tag

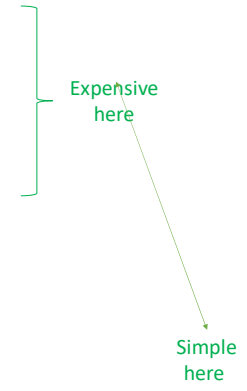
**Methodology:** Run regression with given set of tests

### SOLUTION Example

1. When you run – keep track of the best 10 tests to cover each cover point
2. Use the best tests first for all, then for those not covered go to the second bests, etc – and stop when you achieve coverage

**High Quality:**

The functionally consistent test and coverage abstracts the RTL changes made to implement it



55

# 1. REGRESSION TESTS

## Solution: Example 2

**Metric:** line coverage (how many times each RTL source code line is “ran” )

**Tests:** manually written

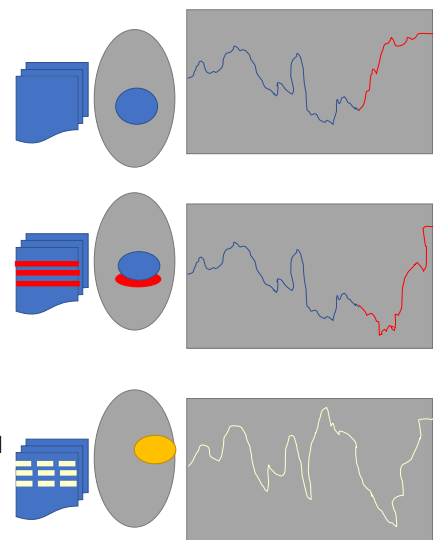
- test that were written by verification engineers to target their code.

**Methodology:** Run regression with given set of tests

### SOLUTION Example

1. When you run – keep track of line coverage per test
2. Use that information ( start with the least covered lines) and add tests until you achieve desired line coverage

RTL change & its IMPACT on what the test exercises



Simple here



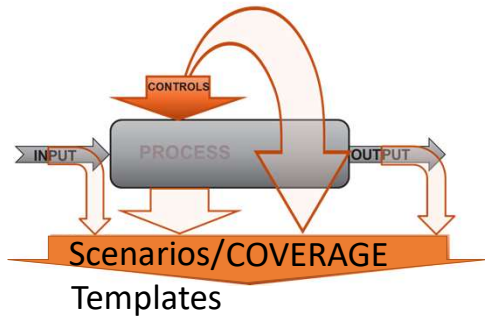
Expensive here

**Low Quality:**

The tests can get “off track” with RTL changes

56

## 2. CREATING TESTS



**Data:**

1. Given data

Scenarios / Templates & Coverage

**Model:**

2. Create Model

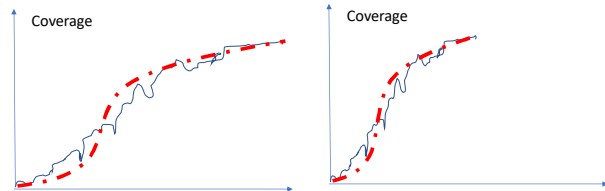
Learn impact of each Scenario /Template separately  $S_i \rightarrow C_i$

**Use:**

- Run several times the same Scenario & learn the coverage distribution it achieves  $S_i \rightarrow C_i$
- Select the subset of scenarios that give the fastest coverage

**Learn:** Coverage is "a function" of Scenarios

**Use:** Create only tests that impact coverage



57

## 2. CREATING TESTS

### Generate Tests for Coverage

Requires the capability of identifying the dependency between a coverage need and the input into the TCG that would cover it

- Learning coverage as function of Scenario, then use the inverse:

Input into TCG identified as a function of **Coverage**

- Use
  - Offline TCG – scenarios
  - Online TCG ( testbenches) – bias & online controls that guide the activity towards a given area ( behavior)
  - Success depends on the methodology:
    - ( functional coverage – scenario based TCG )

58

## 2. CREATING TESTS

### Solution Example 3

Simple here

**Metric:** line & toggle

Simple here

**TCG:** random seed

**Methodology:** Have sequences of random on top of random (with biases and constraints) to determine testing conditions

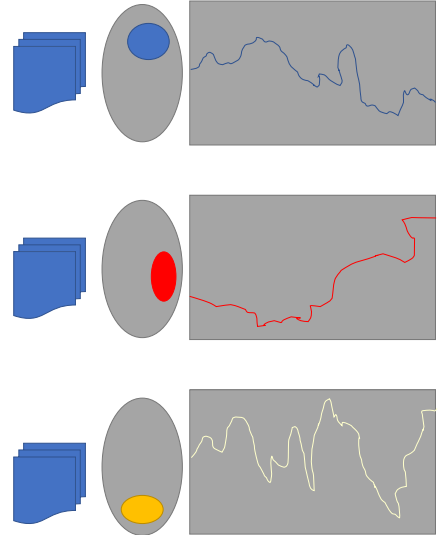
What about Here?

Difficult SOLUTION:

Lean the "bias" -> impose the bias as constraint

**These Solutions Require CONTROL over the process !!**

Same "test" – Same RTL – different area is being exercised



59

## Additional Data (examples)

### 1. HW code repository

Learn:

line coverage as a function of ( test )

(which test is best to cover a given line)

Use:

Run first the tests that cover the changed lines (4%, 6% to get same coverage)

### 2. Issue data base

Learn:

fail likelihood as a function of ( test )

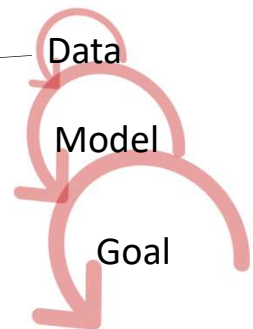
(which test/ scenario has the highest chance of finding a bug )

### 3. Documentation

Learn:

correlations between "areas"

(if bug found – likely to find another one in same or in a close "area" )



60

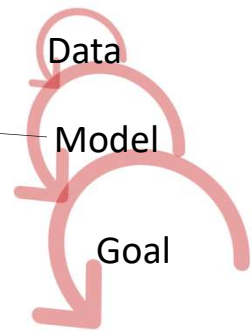
## Different Models

Simple model:

“record top tests per coverage need“

Clustering:

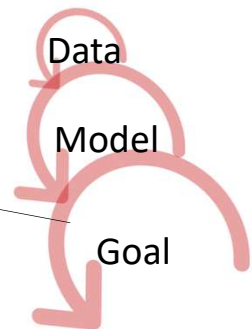
- Use a distance function to determine the similarities between tests (if they cover the same areas or not – requires hierarchical approach)
- Clustering on tests to ID groups, & choose from each group the best tests
- Useful also for test templates.



61

## Different Goals

- Identify the tests (input into TCG) with higher chances of finding a bug
  - DATA: documentation, issues DB, repository ( which RTL file contained bugs in the past), coverage, ...

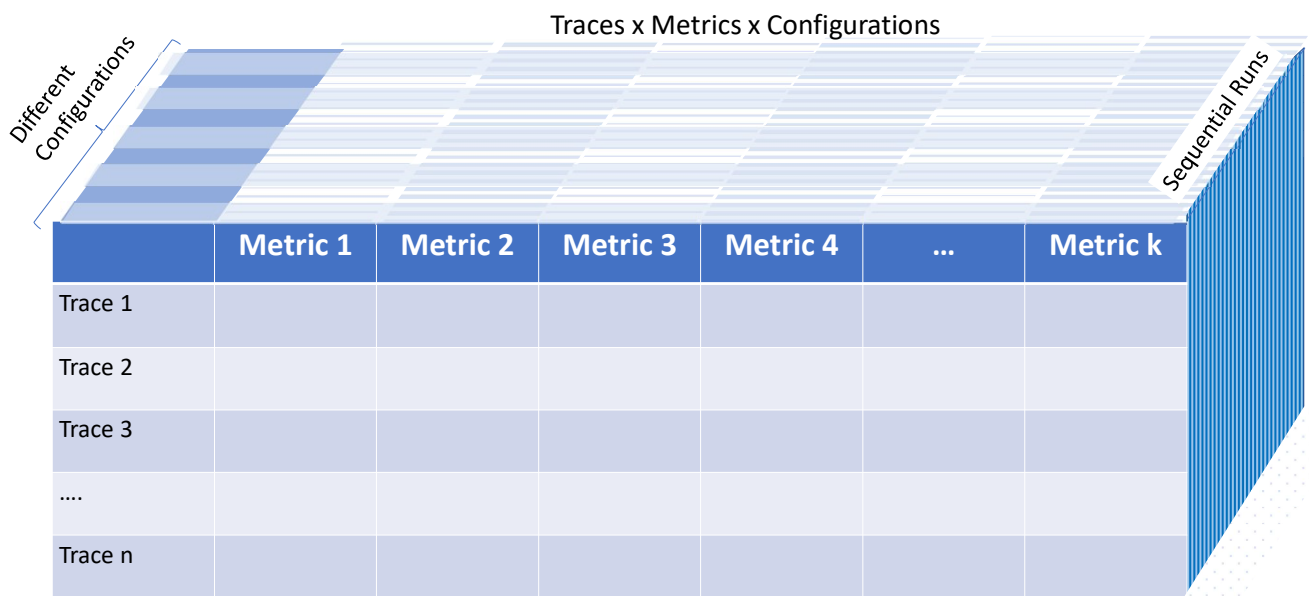


62

# Performance

63

## Data



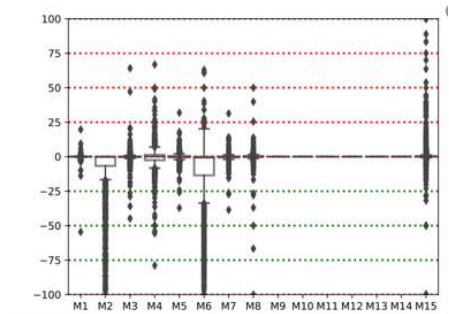
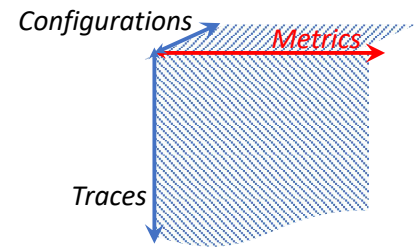
64



## Goals

### • **Metrics :**

- Identify how the metrics change from a configuration to another
- Is there a correlation to each other, or to the IPC
- Do all metrics put together *explain* the IPC ?
  - yes: the weight per metric for IPC
  - no: there are “*missing*” metrics to be identified

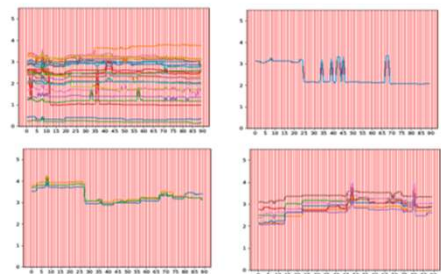
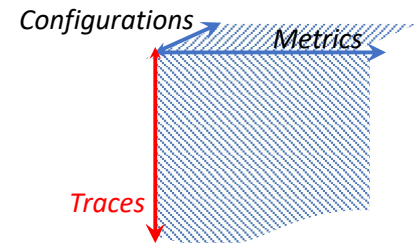


65

## Goals

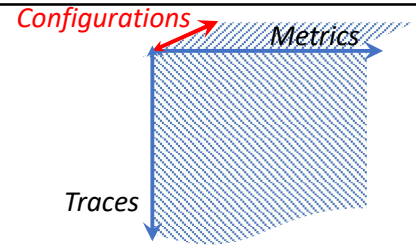
### **Traces** (benchmarks)

- Distance function that shows which benchmarks react in the same way to varying configurations
- Group benchmarks on how similar they are
- Identify the group representative(s) that shows the most impact of change
- Use that information to
  - Decrease # runs
  - Manage traces additions/exclusions



66

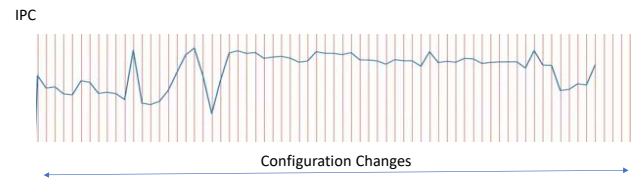
# Goals



## Configurations:

- Decide on the **optimal configuration**

*How would you do it ?*



67

# Resource Usage

68

## Formal

Goal: Which heuristics are best for a given type of problem

Data:

- problem (type)
- heuristic (used)
- resources (time/ processing power/ memory size)

Model:

- Learn resource = function (problem, heuristic)

Use:

- Heuristic = function ( problem)

69

## Processing Power

Goal: What resources to allocate per simulation request

Data:

- Project for which we run the simulation
- Model requested
- Team requesting it
- Test case name (family)
- Allocated resources
- Used resources

Model:

- Needed Resources = function ( project/model/team/testname)

Use:

- Allocate resources accordingly, with a likelihood of being wrong and still save

70

# Management

71

## Project Management

- Complex systems intended to control a project's management
- Use a variety of sources
- Have a variety of goals

72

<http://verifyter.com/technology/debug>

*Automated, Validated Identification of Faulty Code*

**Regression Test Failures**

- Random Tests
- Directed Tests
- Single Cause
- Multiple Cause
- Build Failures
- Test Failures
- Intermittent

**PinDown Patented Algo to find Bugs:**

- Smart Test Grouping
- Run Debug Tests
- Smart Pin-Pointing

**Test Result with Modified Code**

Pass → Assign Bug Report

Fail → Pin-Point

Revision Control System

Committing Fix (optional)

topping the tests at an optimal time

Fig 2. Stopping based on historical statistics

<http://verifyter.com/technology/vo>

<https://vimeo.com/verifyter>

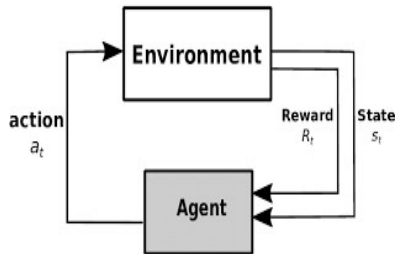
73

# Reinforcement Learning

For (really) *intelligent* tools

74

## Reinforcement Learning

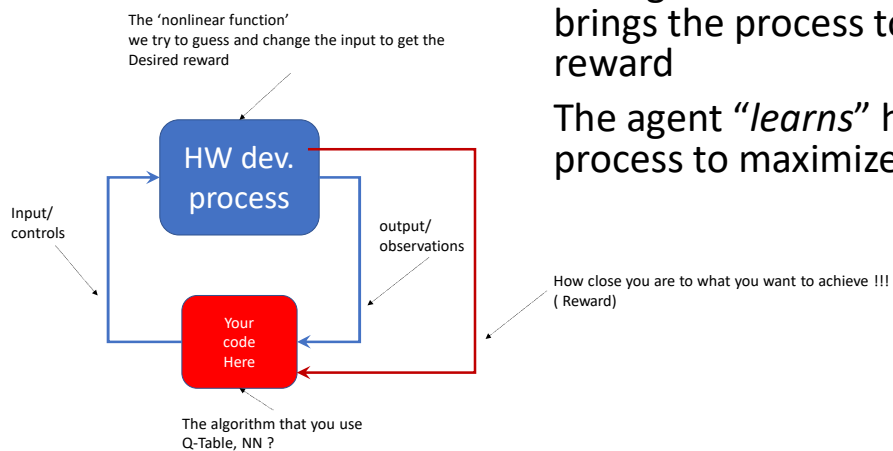


## Agent

- Gives input to the environment
- Gets information from it
- Decides which next input to give  
TO MAXIMIZE REWARD

75

## Reinforcement Learning



Write an agent that controls the process and monitors its state.

The agent “*sees*” how close each input brings the process to the desired reward

The agent “*learns*” how to get the process to maximize the reward

76

Your code Here

**NN**

- Keras
- Lots of doc. available

**Q-Table**

- Easy to implement and understand

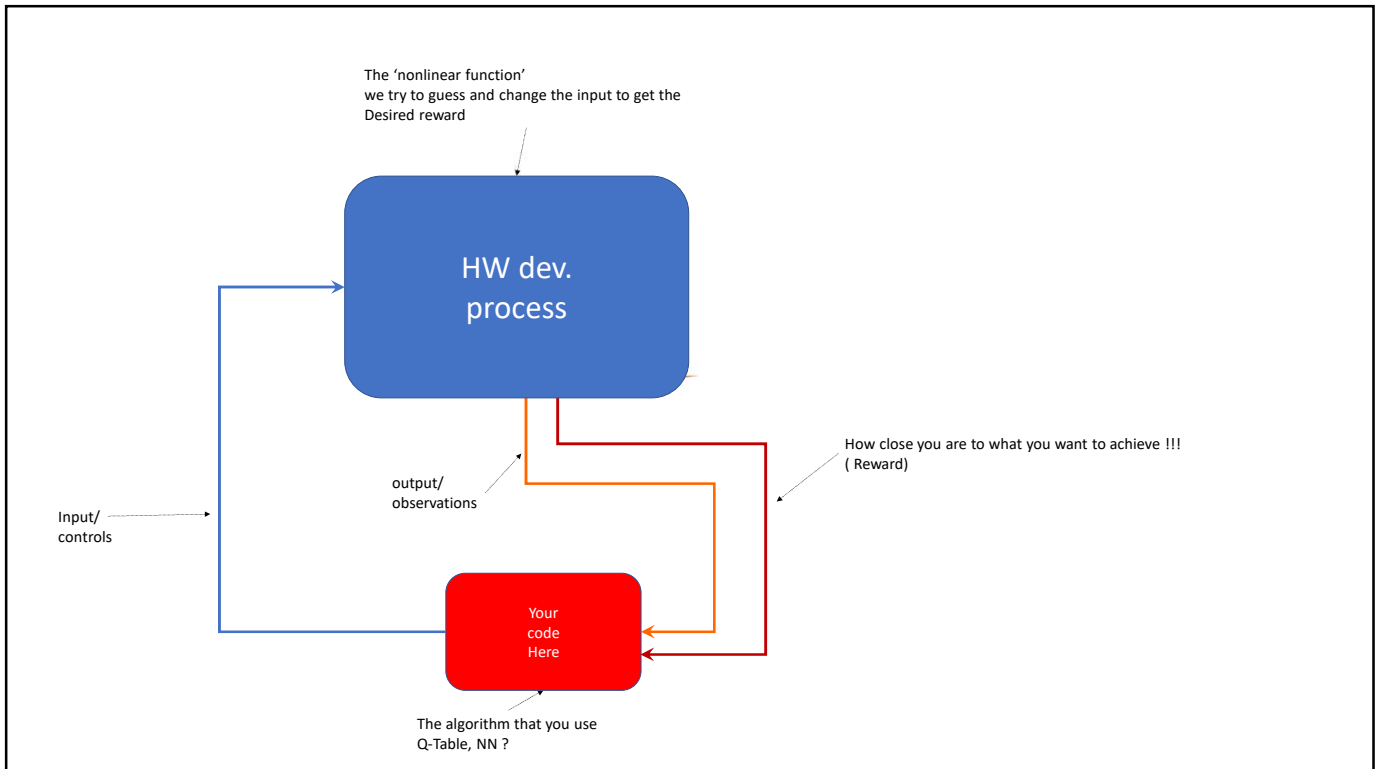
	Action 1	Action 2
State 1	0	5
State 2	0	5
State 3	0	5
State 4	20	0

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

Takes into account the reward the next state can potentially bring to you

<https://adventuresinmachinelearning.com/reinforcement-learning-tensorflow/>

77



78

## Conditions for success

- Fast process
  - Takes a lot of steps to learn the input needed to maximize reward
  - Not useful if each step takes a long time and is expensive
- Reward measurement
  - The reward is a value that shows how well the system did considering the input. Difficult to define and to measure. ( distance function from some desired state)
- Process Uncertainty
  - Works best if I/O without internal memory
  - Processes have internal memories – more difficult to learn