



**UNIVERSITÀ
DEGLI STUDI
DI BERGAMO**

Combinatorial testing

Angelo Gargantini - University of Bergamo –Italy
TAROT summer school – Avila, Spain - July 2022

About myself

angelo.gargantini@unibg.it

cs.unibg.it/gargantini

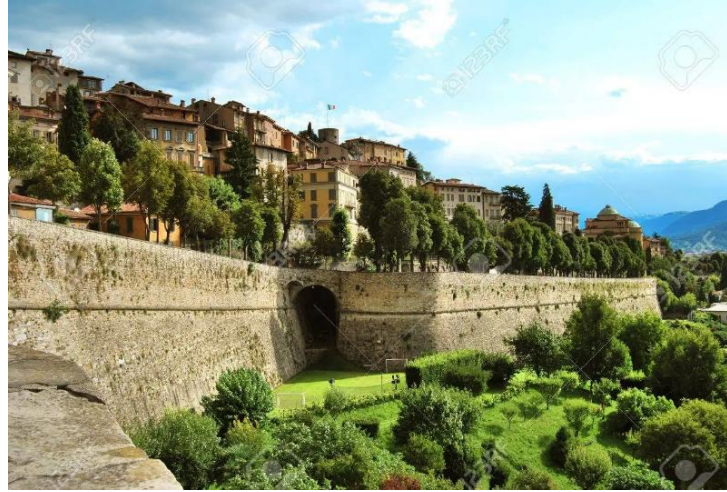
- PhD @Politecnico di Milan
- Worked @ Naval Research Lab – Washington DC
- At the University of Catania
- Now professor at the University of Bergamo



**UNIVERSITÀ
DEGLI STUDI
DI BERGAMO**



Bergamo and Avila have in common:



SE group @ Unibg

projects

- ▶ Formal methods
 - ▶ abstract state machines
- ▶ Testing
 - ▶ Combinatorial, Model-based testing
- ▶ (certified) Medical software
 - ▶ Mechanical Ventilator Milan during COVID



people

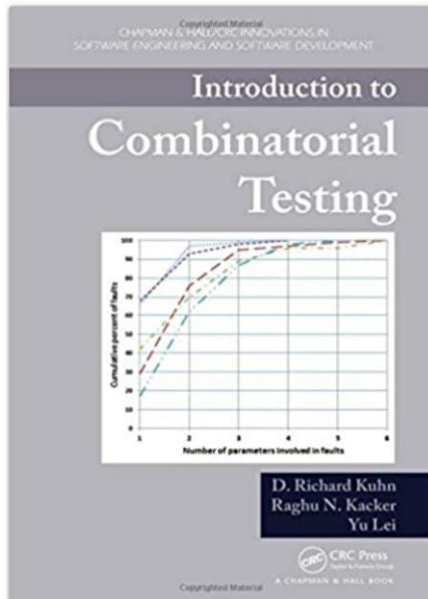
- ▶ Gargantini Angelo
- ▶ Scandurra Patrizia
- ▶ Silvia Bonfanti - PhD, now post doc
- ▶ Andrea Bombarda - PhD student

- ▶ Former member
 - ▶ Paolo Vavassori
 - ▶ Marco Radavelli
- ▶ Collaborators
 - ▶ Elvinia Riccobene @ University of Milan
 - ▶ Paolo Arcaini @ National Institute of Informatics, Tokyo

Outline

- ▶ What is combinatorial testing
 - ▶ **Efficiency:** It can detect faults
- ▶ Partition testing
 - ▶ A method to apply partition testing
 - ▶ How to choose variable values
- ▶ Combinatorial interaction of parameters
- ▶ Generation techniques
 - ▶ IPO, AETG, IPOS
- ▶ Adding constraints
 - ▶ Logic approach, using SAT/SMT solving
- ▶ New ideas

Credits / references



Ammann, Offutt
*Introduction to
Software Testing*



© Ammann & Offutt

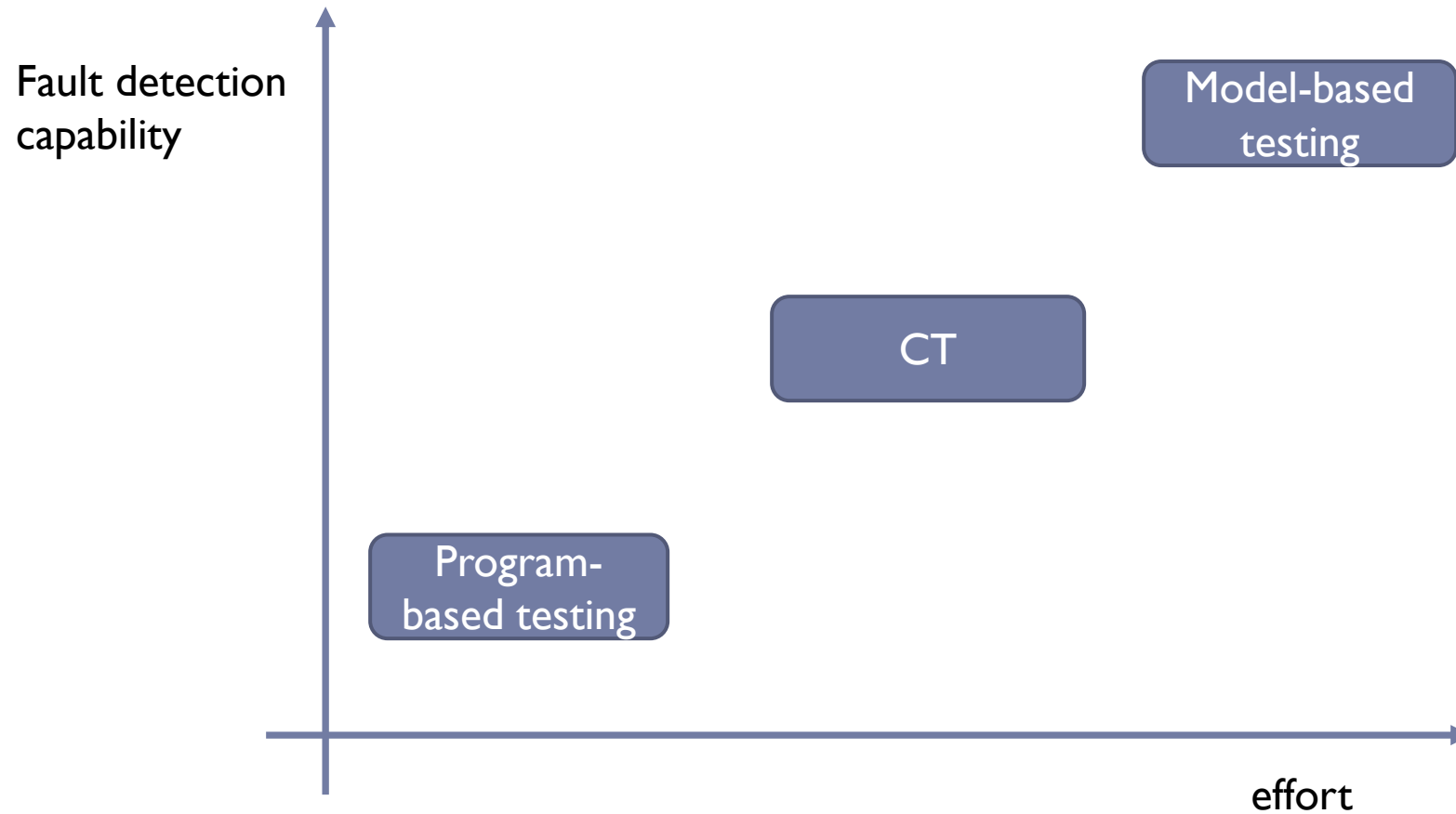
- ▶ <http://csrc.nist.gov/groups/SNS/acts/>
- ▶ www.pairwise.org

Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. ACM Comput. Surv. 43, 2, Article 11 (January 2011), 29 pages.
<https://doi.org/10.1145/1883612.1883618>

What is Combinatorial testing

- ▶ It can be classified “input space” testing or testing based on the **interfaces**
- ▶ No internal information about the system under test is considered, but only the information about the inputs
- ▶ It can be model based testing
 - ▶ Model of the inputs
- ▶ Program based testing
 - ▶ The program is analyzed to extract the parameters
 - ▶ E.g. the parameters of a method. ...

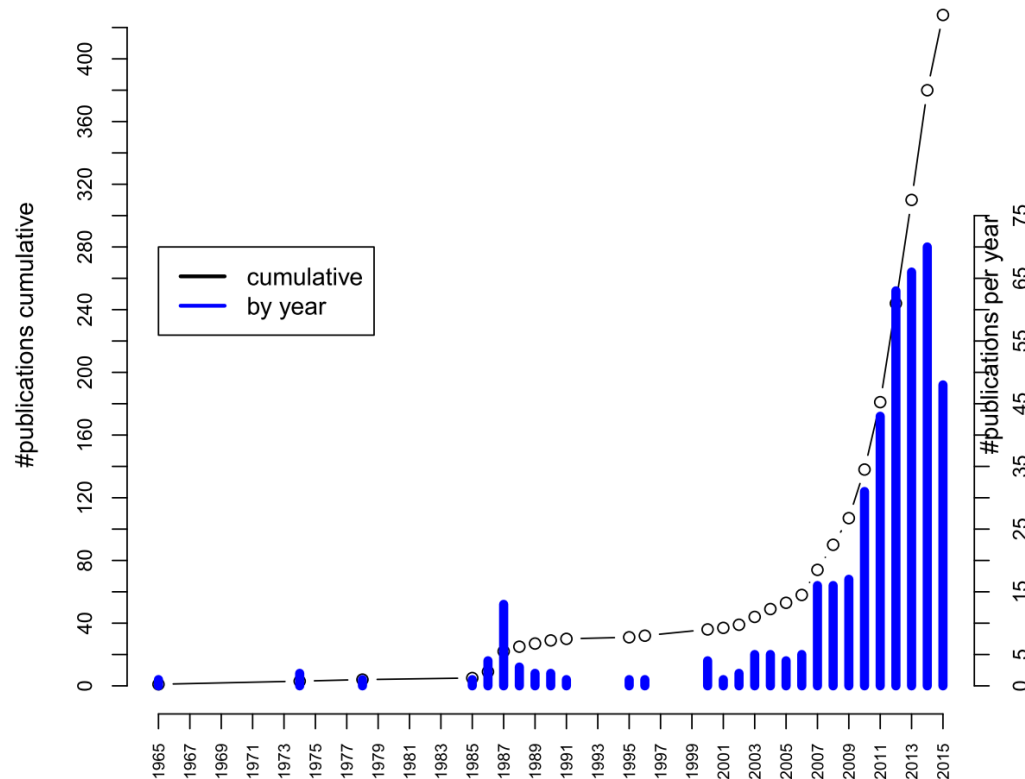
MBT vs program-based testing



Advantages of Input based testing

- ▶ Can be equally applied at several levels of testing
 - ▶ Unit
 - ▶ Integration
 - ▶ System
- ▶ Relatively easy to apply
 - ▶ Test generation is simple, simpler than structure based testing or fault based
- ▶ Easy to adjust the procedure to get more or fewer tests
- ▶ No implementation knowledge is needed
 - ▶ just the input space
 - ▶ Usable even if the complete code/model is not accessible

history and status of CT



- First paper: 1965
- Very old idea form “design of experiments” field
- Every year workshop IWCT at ICST
- COMPETITION

<https://fmse-lab.github.io/ct-competition/>



Combinatorial testing is effective

CIT effectiveness

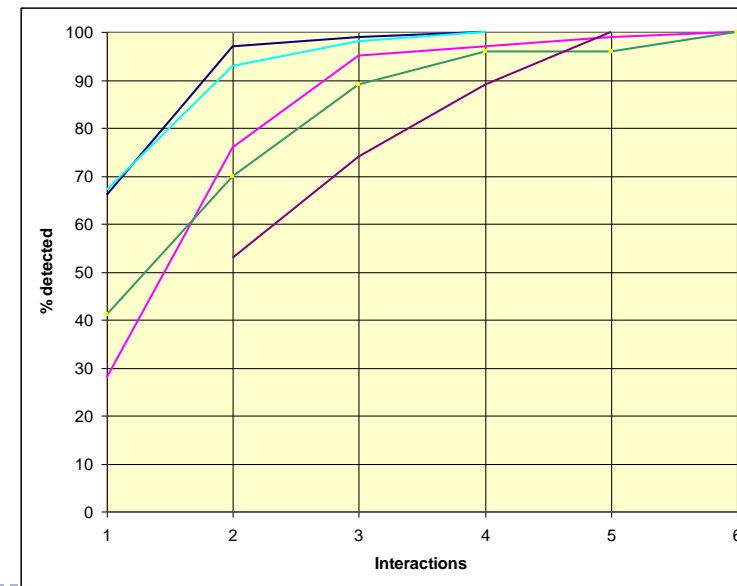
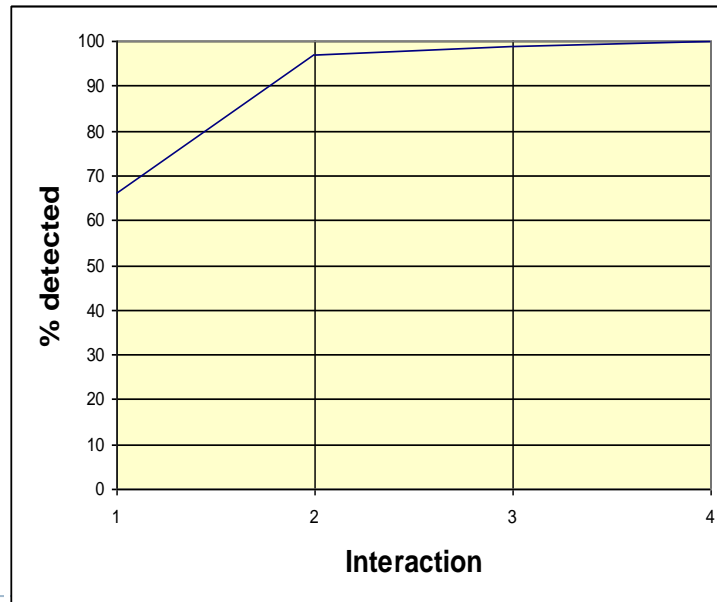
- ▶ Experiments show that CIT is
- ▶ effective
 - ▶ finds faults that traditional testing may be not able to find
- ▶ efficient
 - ▶ A low degree of interaction between inputs can already discover most faults
 - ▶ Pairwise is the most used
 - ▶ Never with interaction > 6

Effectiveness -1

- ▶ *Compared to a traditional company that would use the quasi-exhaustive strategy, the Combinatorial design method (CDM) strategy would reduce its system level test schedule by sixty-eight percent (68%) and save sixty-seven percent (67%) in labor costs associated with the testing.*
- ▶ *Reference: Raytheon (2000). Jerry Huller. Reducing Time to Market with Combinatorial Design Method Testing.*

Effectiveness -2 - Kuhn @ NIST

- ▶ Maximum interactions for fault triggering was 6
- ▶ Reasonable evidence that maximum interaction strength for fault triggering is relatively small
 - ▶ % errors (seeded or found) vs interaction strength for several application:



Effectiveness - 3

- ▶ More experiments are needed
- ▶ New experiments are welcome!

*Combinatorial testing is better
than structural testing ?*

*Combinatorial testing is better
than random testing ?*



Partition testing

Problems ...

- ▶ The input domain to a program contains all the possible inputs to that program
 - ▶ For even small programs, the input domain is so large that it might as well be infinite
- ▶ Testing is fundamentally about choosing finite sets of values from the input domain

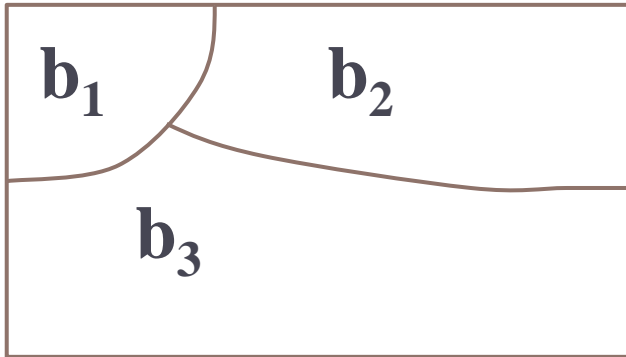
Solution: Input partitioning

- ▶ Domain for each input parameter is partitioned into regions
 - ▶ The domain is substituted by an enumeration



Partitioning Domains

- ▶ Domain D
- ▶ Partition scheme q of D
- ▶ The partition q defines a set of blocks, $Bq = b_1, b_2, \dots, b_Q$
- ▶ The partition must satisfy two properties :
 1. blocks must be pairwise disjoint (no overlap)
 2. together the blocks cover the domain D (complete)



$$b_i \cap b_j = \Phi, \forall i \neq j, b_i, b_j \in B_q$$

$$\bigcup_{b \in B_q} b = D$$

Using Partitions – Assumptions

- ▶ Choose a value from each partition
- ▶ Each value is assumed to be equally useful for testing
- ▶ Application to testing
 - ▶ Find characteristics in the inputs : parameters, semantic descriptions, ...
 - ▶ Partition each characteristics
 - ▶ Choose tests by combining values from characteristics
- ▶ Example Characteristics
 - ▶ Input X is null -> true or false
 - ▶ Order of the input file F -> sorted, inverse sorted, arbitrary
 - ▶ Min separation of two aircraft -> integer 0 ... 1000
 - ▶ Input device -> DVD, CD, VCR, computer

Choosing Partitions

- ▶ Choosing (or defining) partitions seems easy, but is easy to get wrong
- ▶ Consider a file the contains word in some “order”

**b_1 = sorted in ascending order
 b_2 = sorted in descending order
 b_3 = arbitrary order**

but ... something's fishy ...

What if the file is of length 1?

**The file will be in all three blocks ...
That is, disjointness is not satisfied**

Solution:

**Each characteristic should
address just one property
 b_1 and b_2**

File F sorted ascending

- b_1 = true
- b_2 = false

File F sorted descending

- b_1 = true
- b_2 = false

Properties of Partitions

- ▶ If the partitions are not **complete** or **disjoint**, that means the partitions have not been considered carefully enough
- ▶ They should be reviewed carefully, like any design attempt
- ▶ Different **alternatives** should be considered

Example for program based testing

▶ Java

```
enum Color { RED, GREEN, BLU}
```

```
Void foo(long x, Color c, boolean value)
```

Color and boolean domain already partitioned. What about long domain?

Example of partition, from **Boundary Value Analysis**

▶ **MAX_VALUE**

A constant holding the maximum value a long can have, $2^{63}-1$.

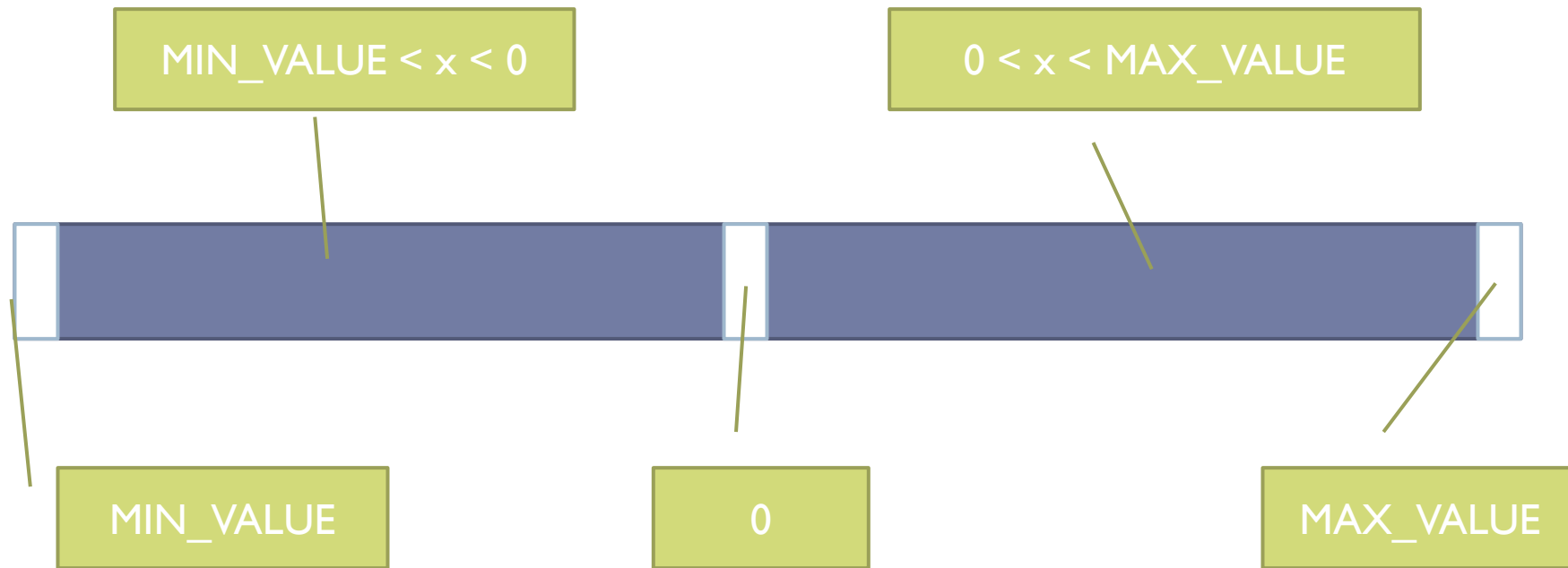
▶ **MIN_VALUE**

A constant holding the minimum value a long can have, -2^{63} .

▶ **BETWEEN MAX E MIN?**

Partition for long

Partitions in 5 subsets



What to do when a Domain is a product of domains

- ▶ When our domain $D = D1 \times D2 \times D3 \dots$
- ▶ example: `foo(int x, enum y, string s)`
 $D = \text{int} \times \text{enum} \times \text{string}$
- ▶ Three possible choices:
 - ▶ Partition D as a whole
 - ▶ Partition D_i and make the cartesian product
 - ▶ Partition D_i and apply combinatorial testing

Partition of cartesian product

- ▶ Given two domains D1 and D2
- ▶ Let P1 a partition for D1 and P2 a partition for D2
- ▶ Partitions can be multiplied to obtain again partitions
- ▶ $D1 \times D2$ can be partitioned by $P1 \times P2$
- ▶ $P1 \times P2$ will contain all the combinations of P1 and P2

D1	d11	(d11,d21)
	d12
	d13
		d21	d22	d23
	D2			

Product of partitions, application

- ▶ For more than one input:

```
/** given three sides return the type  
    of the triangle*/
```

```
TriType Triang(int Side1,int Side2,int Side3)
```

- ▶ If one splits every input in 5 subsets, the input is partitioned in $5 \times 5 \times 5 = 125$ subsets

Partition testing

- ▶ Several methods are based on partition testing [see books by Myers, and Beizer]:
 1. Equivalent Partition
 2. Domain Testing
 3. Boundary Value Analysis
 4. Category Partition [Ostrand Balcer 1988]:
 - ▶ Identify the parameters and variables and their choices
 - ▶ Generate all combinations (test frames)

Partition does not solve the problem!

- ▶ Category partition testing gave us
 - ▶ Systematic approach: Identify characteristics and values (the creative step),
 - ▶ generate combinations (the mechanical step)
- ▶ While equivalence partitioning offers a set of guidelines to design test cases, it suffers from two shortcomings:
 1. It raises the possibility of a large number of sub-domains in the partition.
 - ▶ Test suite size grows very rapidly with number of categories. Can we use a non-exhaustive approach?
 2. It lacks guidelines on how to select inputs from various sub-domains in the partition.





INPUT DOMAIN MODELLING

How to select the inputs and their values

A. Input Domain Modeling

▶ Step 1 : Identify testable functions

- ▶ Individual methods have one testable function
- ▶ Programs have more complicated characteristics—modeling documents such as UML use cases can be used to design characteristics
- ▶ Systems of integrated hardware and software components can use devices, operating systems, hardware platforms, browsers, etc

• Step 2 : Find all the parameters

- Often fairly straightforward, even mechanical
- Important to be complete
- Methods : Parameters and state (non-local) variables used
- Components : Parameters to methods and state variables
- System : All inputs, including files and databases

Modeling the Input Domain (cont)

- ▶ **Step 3 : Model the input domain**
 - ▶ The domain is scoped by the **parameters**
 - ▶ Each parameter refers to a characteristic of the system
 - ▶ Each characteristic is partitioned into sets of **blocks**
 - ▶ Each block **represents a set of values**
 - ▶ This is the most creative design step in applying ISP

Modeling the Input Domain

- ▶ Partitioning characteristics into blocks and values is a very creative engineering step
- ▶ More blocks means more tests
- ▶ The **partitioning** often flows directly from the **definition** of characteristics and both steps are sometimes done together
 - ▶ Should evaluate them separately – sometimes fewer characteristics can be used with more blocks and vice versa
- ▶ Strategies for identifying values :
 - ▶ Include valid, invalid and special values
 - ▶ Sub-partition some blocks
 - ▶ Explore boundaries of domains
 - ▶ Include values that represent “normal use”
 - ▶ Try to balance the number of blocks in each characteristic
 - ▶ Check for completeness and disjointness

Two Approaches to Input Domain Modeling (IDM)

1. Interface-based approach

- ▶ Develops characteristics directly from individual input parameters
- ▶ Simplest application
- ▶ Can be partially automated in some situations

2. Functionality-based approach

- ▶ Develops characteristics from a behavioral view of the program under test
- ▶ Harder to develop—requires more design effort
- ▶ May result in better tests, or fewer tests that are as effective

1. Interface-Based Approach

- ▶ Mechanically consider each parameter in isolation
- ▶ This is an easy modeling technique and relies mostly on syntax
- ▶ Some domain and semantic information won't be used
 - ▶ Could lead to an incomplete IDM
- ▶ Ignores relationships among parameters

Consider TriTyp

Three *int* parameters

IDM for each parameter is identical

Reasonable characteristic : *Relation of side with zero*

2. Functionality-Based Approach

- ▶ Identify characteristics that correspond to the intended functionality
- ▶ Requires more design effort from tester
- ▶ Can incorporate domain and semantic knowledge
- ▶ Can use relationships among parameters
- ▶ Modeling can be based on requirements, not implementation
- ▶ The same parameter may appear in multiple characteristics, so it's harder to translate values to test cases

Consider TriTyp again

The three parameters represent a *triangle*

IDM can combine all parameters

Reasonable characteristic : *Type of triangle*

Angelo Gargantini - TAROT 2022

Interface-Based IDM – TriTyp

- TriTyp, had one testable function and three integer inputs

First Characterization of TriTyp's Inputs

Characteristic	b_1	b_2	b_3
q_1 = "Relation of Side 1 to 0"	greater than 0	equal to 0	less than 0
q_2 = "Relation of Side 2 to 0"	greater than 0	equal to 0	less than 0
q_3 = "Relation of Side 3 to 0"	greater than 0	equal to 0	less than 0

- ▶ A maximum of $3*3*3 = 27$ tests
- ▶ Some triangles are valid, some are invalid
- ▶ Refining the characterization can lead to more tests ...

Second Characterization of TriTyp's Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Refinement of q_1 "	greater than 1	equal to 1	equal to 0	less than 0
q_2 = "Refinement of q_2 "	greater than 1	equal to 1	equal to 0	less than 0
q_3 = "Refinement of q_3 "	greater than 1	equal to 1	equal to 0	less than 0

- ▶ A maximum of $4 \times 4 \times 4 = 64$ tests
- ▶ This is only complete because the inputs are integers (0 .. 1)

Possible values for partition q_1

Characteristic	b_1	b_2	b_3	b_4
Side1	2	1	0	-1

Test boundary conditions

Functionality-Based IDM – TriTyp

- ▶ First two characterizations are based on syntax-parameters and their type
- ▶ A semantic level characterization could use the fact that the three integers represent a triangle

Geometric Characterization of TriTyp's Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles	equilateral	invalid

- Oops ... something's fishy ... equilateral is also isosceles !
- We need to refine the example to make characteristics valid

Correct Geometric Characterization of TriTyp's Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid

At the end of the first 3 steps

- ▶ Set of parameters
 - ▶ Each parameter has a list of possible values
 - ▶ Each value represent a block
 - ▶ Each value can represent a partition
- ▶ There may be constraints over the possible values

Combination Strategies criteria

- ▶ **Step 4 : Apply a test criterion to choose combinations of values**
 - ▶ A test input has a value for each parameter
 - ▶ One block for each characteristic
 - ▶ Choosing all combinations is usually infeasible
 - ▶ Coverage criteria allow subsets to be chosen
- ▶ **Step 5 : Refine combinations of blocks into test inputs**
 - ▶ Choose appropriate values from each block

Choosing Combinations of Values

Step 4 – Choosing Combinations of Values

- ▶ Once characteristics and partitions are defined, the next step is to choose test values
- ▶ We use criteria – to choose effective subsets
- ▶ The most obvious criterion is to choose all combinations ...

All Combinations (ACoC) : All combinations of blocks from all characteristics must be used.



Example

- ▶ Characteristic1: color : {Green, White, Red}
- ▶ Characteristic2: age [0...100] : blocks [0..6] [7..40][41..100]
- ▶ Characteristic3: isFemale : {true, false}

- ▶ ACoC all the combinations of blocks
 - ▶ (not the combinations of all the values)
- ▶ Combination1: Green, 5,true

Number of combinations

- Number of tests is the product of the number of blocks in each characteristic :

$$\prod_{i=1}^Q B_i$$

- The second characterization of TriTyp results in $4*4*4 = \underline{64 \text{ tests}}$ – too many ?

How do we test this?

- ▶ 34 switches = $2^{34} = 1.7 \times 10^{10}$ possible inputs = 1.7×10^{10} tests ????



ISP Criteria – Each Choice

- ▶ AoC tests are too many
- ▶ One criterion comes from the idea that we should try at least one value from each block

Each Choice (EC) : One value from each block for each characteristic must be used in at least one test case.

- Number of tests is the number of blocks in the largest characteristic

$$\max_{i=1}^Q \text{size}(B_i)$$

For TriTyp

- ▶ Three inputs side1, side2, side3
- ▶ Four values each 2, 1, 0, -1

- ▶ A test with 4 test is enough:

2, 2, 2

1, 1, 1

0, 0, 0

-1, -1, -1

Each Choice (EC)

- Characteristic1: color : {Green, White, Red}
- Characteristic2: age [0...100] : blocks [0..6] [7..40][41..100]
- Characteristic3: isFemale : {true, false}
- Each Choice (EC)
 - 3 Combination:
Green, 5,true
...
- Is there something in the middle?

AoC “Too many” ! - EC only 2

EC only 2:
All ON
All OFF



some assumptions

- ▶ What if we knew that **one** single switch always causes the fault?

2 tests would be enough to find if the system is correct:

- all off, all on

What if we knew that any failure involves **2 switches** in a particular configuration?



ISP Criteria – Pair-Wise

- ▶ Each choice yields few tests – cheap but perhaps ineffective
- ▶ Another approach asks values to be combined with other values

Pair-Wise (PW) : A value from each block for each characteristic must be combined with a value from every block for each other characteristic.

Pair wise – in other words

- ▶ Taken every pair of parameters:
 - ▶ Every combination among those parameters is tested

Example

Display Mode	Color	Screen size
full-graphics	Monochrome	Hand-held
Low resolution	16-bit	Laptop
text-only	True-color	Full-size

- ▶ 3 variables with 3 values each: $3^3 = 27$ possible combinations
- ▶ Combinatorial testing with much fewer tests

Test Suite - example

- ▶ pairwise testing can be achieved by only 9 tests

Test	Color	Display Mode	Screen Size
1	Monochrome	Full-graphics	Hand-held
2	16-bit	Text-only	Laptop
3	True-color	Full-graphics	Hand-held
4

One test covers many combinations:

e.g. Test 1 covers 3 pairs:
(Monochrome, Full-graphics)
(Monochrome, Hand-held)
(Full-graphics, Hand-held)

Other figures:

2^{100} combinations with 10 tests;

$10^{20} \rightarrow 200$ tests; ...

Combinatorial approach

- ▶ **Pairwise combination** instead of exhaustive
 - ▶ Generate combinations that efficiently cover all pairs of values
 - ▶ **Rationale**: most failures are triggered by single values or combinations of a few values. Covering pairs (triples,...) reduces the number of test cases, but reveals most faults
 - ▶ Extended by t-wise: test all the combinations of t values

ISP Criteria –T-Wise

- ▶ A natural extension is to require combinations of t values instead of 2

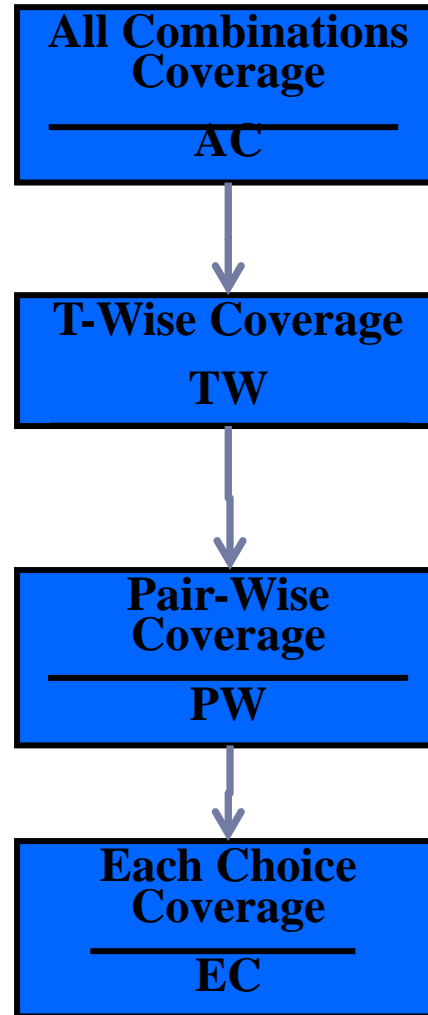
t-Wise (TW) : A value from each block for each group of t characteristics must be combined.

- Number of tests is at least the product of t largest characteristics
- If all characteristics are the same size, the formula is

$$(\text{Max}_{i=1}^Q (B_i))^t$$

- If t is the number of characteristics Q , then all combinations
- That is ... Q -wise = AC
- t -wise is expensive and benefits are not clear

ISP Coverage Criteria Subsumption



Exercise

- ▶ example system: component based application

CLIENT	WEB SERVER	PAYMENT	DATABASE
FIREFOX	WEB SPHERE	MASTER CARD	DB/2
IE	APACHE	VISA	ORACLE
OPERA	.NET	AMEX	ACCESS

One test is missing ...

- ▶ $3^4 = 81$ exhaustive test cases
- ▶ 9 test cases

CLIENT	WEB SERVER	PAYMENT	DATABASE
FIREFOX	WEB SPHERE	MASTER CARD	DB/2
FIREFOX	.NET	AMEX	ORACLE
FIREFOX	APACHE	VISA	ACCESS
IE	WEB SPHERE	AMEX	ACCESS
IE	APACHE	MASTER CARD	ORACLE
IE	.NET	VISA	DB/2
OPERA	WEB SPHERE	VISA	ORACLE
OPERA	.NET	MASTER CARD	ACCESS



Generation techniques

Generation techniques families

- ▶ **Algebraic**
 - ▶ Based on some mathematical/algebraic properties
- ▶ **Search based, greedy, based on heuristics**
 - ▶ Because the problem of generating a minimum test suit for combinatorial testing is NP-complete, most methods and tools use a greedy approach
- ▶ **Logic based**
 - ▶ Based on SAT/SMT solving and model checking

Classification

- ▶ **Algebraic methods** that are mainly developed by mathematicians
 - ▶ Latin squares, Orthogonal arrays, Covering arrays
 - ▶ Recursive Construction
- ▶ **Search-Based** methods that are mainly developed by computer scientists
 - ▶ AETG (from Telcordia), TCG (from JPL/NASA), DDA (from ASU), PairTest/Fireeye (from NIST)
 - ▶ Incremental construction

Search-Based vs Algebraic Methods

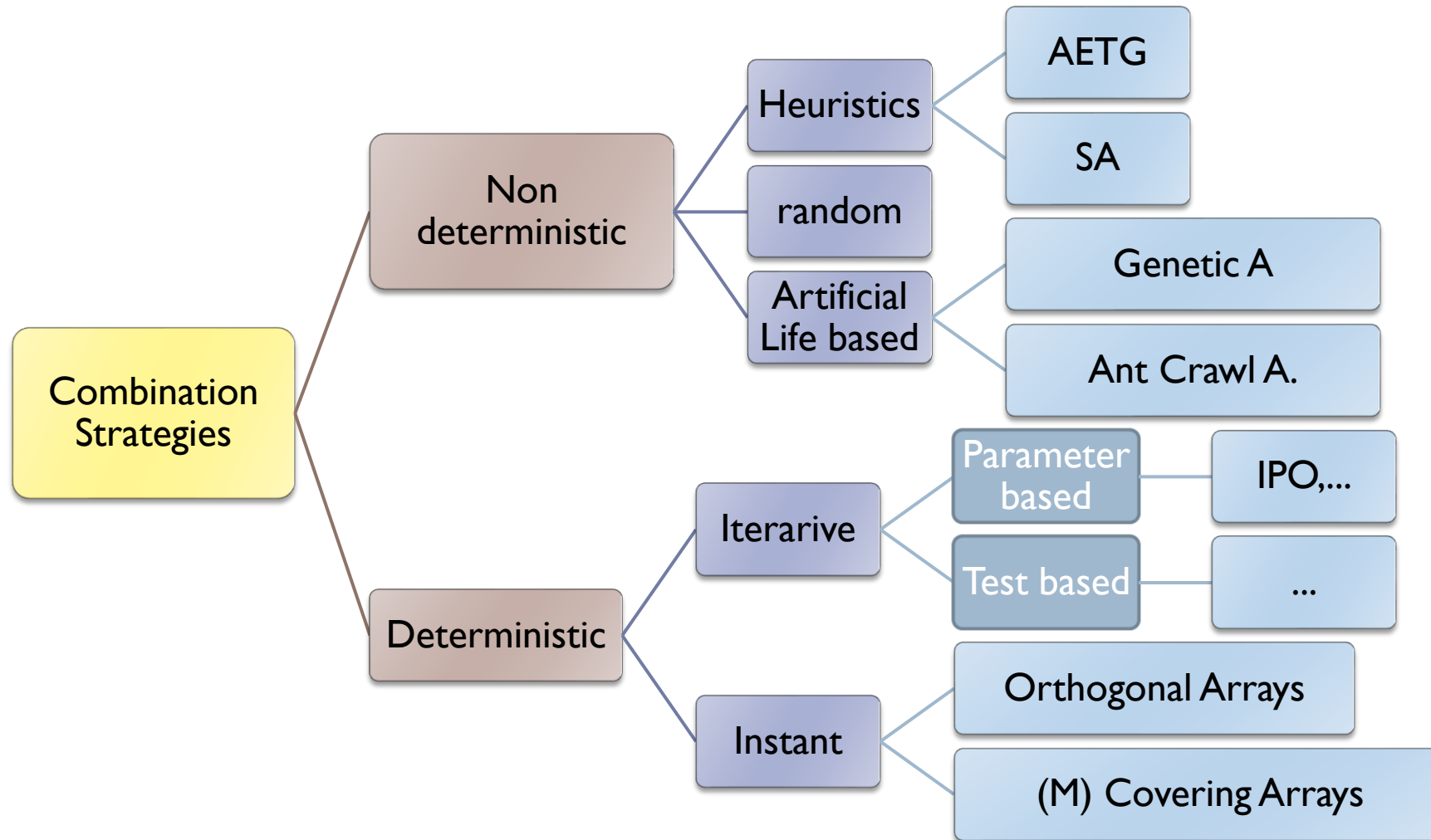
- ▶ **Algebraic methods:**

- ▶ Advantages: very fast, and often produces optimal results
- ▶ Disadvantages: limited applicability, difficult to support parameter relations and constraints
- ▶ E.g. most work only if all the parameters have the same domain size

- ▶ **Search-based methods:**

- ▶ Advantages: no restrictions on the input model, and very flexible, e.g., relatively easier to support parameter relations and constraints
- ▶ Disadvantages: explicit search takes time, the resulting test sets are not optimal

Complete Taxonomy (Grindal)





Algebraic methods

Latin Squares

- ▶ a **Latin square** is an $n \times n$ table filled with n different symbols in such a way that each symbol occurs exactly once in each row and exactly once in each column.

1	2	3
2	3	1
3	1	2

- ▶ Similar to Sudoku
- ▶ Every row has one instance of a character
- ▶ Every column has one instance of a character
- ▶ **Orthogonal** if, when combined, each combination appears once

Two orthogonal Latin Squares

L1: Latin square for 1,2,3,4

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

L2: Latin square for A,B,C,D

A	B	C	D
D	C	B	A
B	A	D	C
C	D	A	B

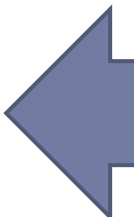
L1 and L2 are
orthogonal?



Combination of the Orthogonal Latin Squares

L1 + L2 combined

A1	B2	C3	D4
D2	C1	B4	A3
B3	A4	D1	C2
C4	D3	A2	B1



Every pair
appears
only once !



From LQ to tests

- ▶ Add to the matrix the index r for rows and c for columns
- ▶ The matrix can be described by the tuples $\langle r, c, z_i \rangle$ where z_i is at position r, c
- ▶ The set of all the tuples can be used as test suite for pairwise

Combination has Interesting Properties

- ▶ Each pair of symbols occurs precisely once
 - ▶ Avoid redundant pairs guarantees minimality
- ▶ An exponential relationship is turned into a quadratic relationship for number of tests required
- ▶ Can combine more orthogonal Latin Squares to increase the number of simultaneous factors
- ▶ Let K be the number of Variables and N be the number of possible values: $N^2 < N^K$
- ▶ Problems:
 - ▶ How many orthogonal square exists of size n ?
 - ▶ No orthogonal square to a 6x6 Latin Square exists!

Extensions

- ▶ Orthogonal arrays similar to Latin squares
- ▶ Covering arrays (CA)
 - ▶ can be built recursively
 - ▶ always exist
 - ▶ **BUT** All the parameter must have the same size
- ▶ Mixed covering arrays (MCA)

Greedy methods

Greedy methods

- ▶ **Parameter based**
 - ▶ One column at the time
 - ▶ IPO
 - ▶ IPOS – still room to improve
- ▶ **Test case based**
 - ▶ Add one test at the time
 - ▶ AETG

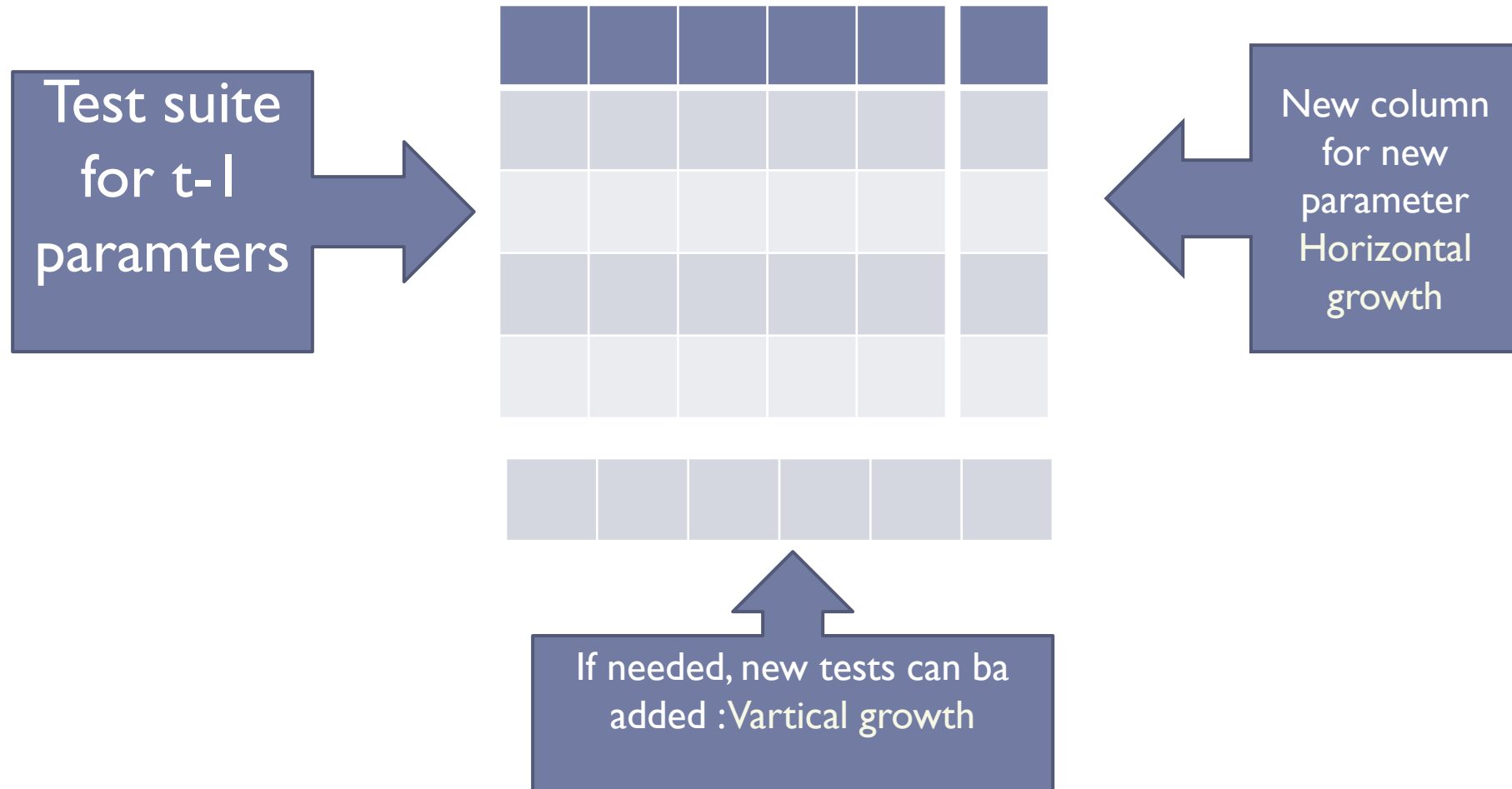
IPO: In-Parameter-Order

- ▶ Originally presented in:
- ▶ Yu Lei, K. C. Tai, "*In-Parameter-Order: A Test Generation Strategy for Pairwise Testing*," High-Assurance Systems Engineering, IEEE International Symposium on, p. 254, Third IEEE International High-Assurance Systems Engineering Symposium, 1998
- ▶ Several extensions
- ▶ NIST
<http://csrc.nist.gov/groups/SNS/acts/>
- ▶ TOOL: FireEye, now ACTS

IPO: In-Parameter-Order

- ▶ Builds a **t-way** test set in an incremental manner
 1. A **t-way** test set is first constructed for the first **t** parameters, simply considering their combinations
 2. Then, the test set is extended to generate a **t-way** test set for the first **t + 1** parameters
 3. The test set is repeatedly extended for each additional parameter.
- ▶ Two steps involved in each extension for a new parameter:
 - **Horizontal growth**: extends each existing test by adding one value of the new parameter
 - **Vertical growth**: adds new tests, if necessary

Adding parameters



Strategy In-Parameter-Order

```
/* step 1: for the first t parameters  $p_1, p_2, \dots, p_t$  */  
T := {(v1, v2, ..., vt) | v1, v2, ..., vt are values of p1, p2, ..., pt}  
if n = t then stop;  
/* step 2: for the remaining parameters */  
for parameter pi, i = t + 1, ..., n do  
  begin /* add parameter pi */  
    /* 2a: horizontal growth */  
    for each test (v1, v2, ..., vi-1) in T do  
      replace it with (v1, v2, ..., vi-1, vi), where vi is a value of pi  
    /* 2b: vertical growth */  
    while T does not cover all the interactions between pi and  
      each of p1, p2, ..., pi-1 do  
      add a new test for p1, p2, ..., pi to T;  
  end
```


Example

- ▶ Consider a system with the following parameters and values:
 - ▶ parameter **A** has values **WIN** and **LIN**
 - ▶ parameter **B** has values **1NT** and **AMD**, and
 - ▶ parameter **C** has values **IPV4**, **IPV6**
- ▶ Pairwise testing $t = 2$

Step 1: the first t parameters

- ▶ if a test suite wants to cover all the t-combinations of t parameters, it must contain all the possible combinations
- ▶ $t = 2$
- ▶ parameter A has values WIN and LIN
- ▶ parameter B has values INT and AMD
- ▶ Initial test suite (CA):

A	B
WIN	INT
WIN	AMD
LIN	INT
LIN	AMD

Step 2: adding a new parameter

- ▶ Add the tests to cover the $t+1$ th parameter.
- ▶ Add a column to the CA for the new parameter
- ▶ For the values of the new

Horizontal Growth

A	B
WIN	INT
WIN	AMD
LIN	INT
LIN	AMD



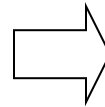
A	B	C
WIN	INT	IPV4
WIN	AMD	IPV6
LIN	INT	IPV4
LIN	AMD	IPV6

Step 2 b

- Check if all the tuples are covered, in case add new rows (vertical growth)

Vertical Growth

A	B	C
WIN	INT	IPV4
WIN	AMD	IPV6
LIN	INT	IPV4
LIN	AMD	IPV6



A	B	C
WIN	INT	IPV4
WIN	AMD	IPV6
LIN	INT	IPV4
LIN	AMD	IPV6
LIN	AMD	IPV4
LIN	INT	IPV6

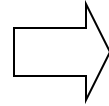
missing :
AMD, IPV4
INT, IPV6

Exercise

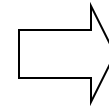
- ▶ parameter A has values A1 and A2
- ▶ parameter B has values B1 and B2, and
- ▶ parameter C has values C1, C2, and C3

Example (2)

<u>A</u>	<u>B</u>
A1	B1
A1	B2
A2	B1
A2	B2



<u>A</u>	<u>B</u>	<u>C</u>
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1



<u>A</u>	<u>B</u>	<u>C</u>
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1
A2	B1	C2
A1	B2	C3

Horizontal Growth

Vertical Growth

Open problems

- ▶ When adding a new column, how to choose the values?
- ▶ When adding a new row, how to choose the new row?

IPO-s

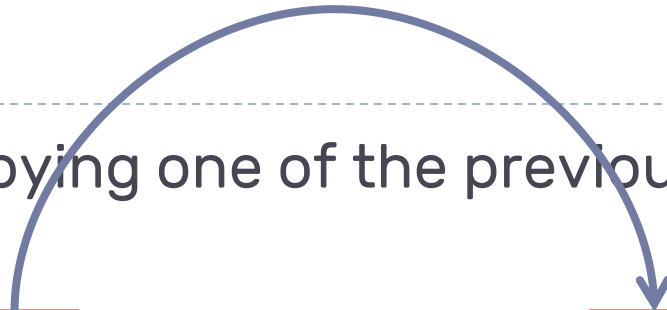
- ▶ Andrea Calvagna and Angelo Gargantini
IPO-s: incremental generation of combinatorial interaction test data based on symmetries of covering arrays
in 5th Workshop on Advances in Model Based Testing (A-MOST 2009), Proceedings (2009)
- ▶ IPO-s: in parameter order, symmetry based
 - ▶ pairwise combine first two parameters
 - ▶ add one more column (parameter)
 - ▶ make sure is paired it with all previous
 - ▶ repeat until all parameters paired.

Symmetry based

- ▶ When adding a new column, one can exploit some properties of the test suite if the parameters are ordered in their size
- ▶ If the test suite covers already the first $t - 1$ parameters
 $1 \dots t - 1$
- ▶ The last column $t - 1$ is already paired with all the previous parameters: it can be copied as t column
 - ▶ Only the pairs between t and $t-1$ will be missing

Example

- ▶ set new column copying one of the previous, e.g. the last one



P1	P2
0	0
0	1
0	2
1	0
1	1
1	2
2	0
2	1
2	2

P1	P2	P3
0	0	0
0	1	1
0	2	2
1	0	0
1	1	1
1	2	2
2	0	0
2	1	1
2	2	2

P2 is paired with P1

P3 is already paired
with P1 too

Only pairs between
P2 and P3 are missing

Next step: change
values of P3 to add
the missing pairs ...

AETG: Automatic Efficient Test Generator

- ▶ A commercial tool developed by Telcordia, and protected by a US patent
- ▶ As web service
- ▶ <http://aetgweb.argreenhouse.com>
- ▶ The algorithm is more or less known
 - ▶ greedy non deterministic



AETG algorithm

- ▶ Starts with an empty set and adds one (complete) test at a time
 - ▶ Keep adding a new (complete) test case until all the combinations are covered
- ▶ Each test to be added is **locally optimized** to cover the most number of missing pairs
 - Use a greedy algorithm to construct M tests (candidates)
 - Choose the test that covers most uncovered pairs
 - A “good” M is 50

Complete algorithm

- ▶ Assume that we have a system with k test parameters and that the i -th parameter has l_i different values.
- ▶ Assume that we have already selected r test cases. We select the $r+1$ by first generating M different candidate test cases and then choosing one that covers the most new pairs.
- ▶ Each candidate test case is selected by the following greedy algorithm.
 - ▶ Choose a parameter f and a value l for f such that that parameter value appears in the greatest number of uncovered pairs.
 - ▶ Let $f_1 = f$. Then choose a random order for the remaining parameters. Then we have an order for all k parameters f_1, \dots, f_k .
 - ▶ Assume that values have been selected for parameters f_1, \dots, f_j . For $1 \leq i \leq j$, let the selected value for f_i be called v_i . Then choose a value v_{j+1} for f_{j+1} as follows. For each possible value v for f_j , find the number of new pairs in the set of pairs $\{ f_{j+1} = v \text{ and } f_i = v_i \text{ for } 1 \leq i \leq j \}$. Then let v_{j+1} be one of the values that appeared in the greatest number of new pairs.
- ▶ Note that in this step, each parameter value is considered only once for inclusion in a candidate test case. Also, that when choosing a value for parameter f_{j+1} , the possible values are compared with only the j values already chosen for parameters f_1, \dots, f_j .

How to build the candidates

- ▶ Let UC be the set of all pairs still missing in the current test suite
 1. Select the variable and the values included in most pairs in UC
 2. Put the rest of the variables into random order
 3. For each variable in the sequence choose the values included in most pairs in UC

- ▶ Repeat steps 1-3 M times

Step 1: adding a new row: first value

- ▶ To add a new tests:
 - ▶ Consider all the tuples not covered (UC)
 - ▶ Choose a parameter f and a value l for f such that that parameter value appears in the greatest number of uncovered pairs.
 - ▶ Fix $f_1 = f$ and choose a random order for the remaining parameters.

<u>OS</u>	<u>CPU</u>	<u>IPV</u>
WIN	INT	IPV4
WIN	AMD	IPV6

<u>UC:MISSING</u>	
LIN	INT
LIN	AMD
INT	IPV6
AMD	IPV4

<u>F1</u>	<u>F2</u>	<u>F3</u>
OS	CPU	IPV
LIN		
OS	IPV	CPU
LIN		

Step 2:

- ▶ Assume that values have been selected for parameters f_1, \dots, f_j . For $1 \leq i \leq j$, let the selected value for f_i be called v_i . Then choose a value v_{j+1} for f_{j+1} as follows. For each possible value v for f_j , find the number of new pairs in the set of pairs $\{ f_{j+1} = v \text{ and } f_i = v_i \text{ for } 1 \leq i \leq j \}$. Then let v_{j+1} be one of the values that appeared in the greatest number of new pairs.

Step 2: in brief

1. Take the next unassigned parameter
2. Consider every value and choose the best
3. Return step 1

UC:MISSING

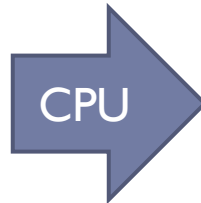
LIN INT

LIN AMD

INT IPV6

AMD IPV4

F1	F2	F3
OS	CPU	IPV
LIN		



F1	F2	F3
OS	CPU	IPV
LIN	INT	
LIN	AMD	

Both new
pairs
Take the
first



F1	F2	F3
OS	CPU	IPV
LIN	INT	IPV4
		IPV6

INT, IPV6: 1 new pair
With IPV4: 0 pairs

OS	CPU	IPV
LIN	INT	IPV4

Comparsion among strategies

- ▶ Several experiments can be performed to evaluate how a strategy performs
 - ▶ Sveral classical benchmarks
- ▶ **TIME**
 - ▶ Necessary to generate the test suite (Complexity)
 - ▶ Not so important but must be reasonable
- ▶ **SIZE**
 - ▶ Of the final test suite
 - ▶ Size matters – NO PICTURE HERE
 - ▶ Especially for parameters of different sizes (otherwise algebraic methods win)

Open problems

- ▶ Not only the size is important
 - ▶ PICTURE HERE ?
- ▶ Requirements for a better CIT technique
 - ▶ Constraints
 - ▶ Over the inputs – later
 - ▶ Seeds
 - ▶ Tests already generated or required to be included in the final test suite
 - ▶ Combination requirements defined by the user
 - ▶ Particula combinations required by the user
- ▶ Others
 - ▶ Fault detetcion capability ? We want the smallest test suite – what about faults
 - ▶ Debugging and CIT



Constraints

Constraints Among Characteristics

- ▶ Some combinations of blocks are infeasible
 - ▶ “less than zero” and “scalene” ... not possible at the same time
- ▶ These are represented as constraints among blocks
- ▶ Two particular types of constraints
 - ▶ A block from one characteristic cannot be combined with a specific block from another **forbidden combinations**
 - ▶ A block from one characteristic can ONLY BE combined with a specific block from another characteristic
- ▶ Infeasible combinations should be not included

Example of constraints

CLIENT	WEB SERVER	PAYMENT	DATABASE
FIREFOX	WEB SPHERE	MASTER CARD	DB/2
IE	APACHE	VISA	ORACLE
OPERA	.NET	AMEX	ACCESS

- ▶ If the webserver is .Net, than Database is Access
- ▶ With the webserver APACHE, the database can NOT be Oracle
- ▶ Most approaches support only **forbidden combinations**

Complex constraints

- ▶ Constraints must be translated to forbidden combinations
- ▶ Example
 - ▶ If $x = 1$ then $y > 10$
 - ▶ Forbidden (x,y) : $(1,0), (1,1) \dots (1,9)$
- ▶ As the number of input grows, the explicit list may explode and it may become practically infeasible to find for a user
- ▶ Moreover there may exist implicit constraints, whose consequences are not immediately detected by human inspection.
- ▶ Example
 - ▶ (1) $x > 6$, (2) $b \neq y$ (3) $x = 7$ implies $b = y$
 - ▶ (implicit) $x > 7$
- ▶ Goal: supporting expressive constraints
 - ▶ Easier to obtain from the requirements



CTwedge

CTWEDGE in brief

<https://github.com/fmselab/ctwedge>

Language for CIT problems

1. with a precise formal semantics and a grammar by XTEXT
2. A textual editor integrated in the eclipse IDE

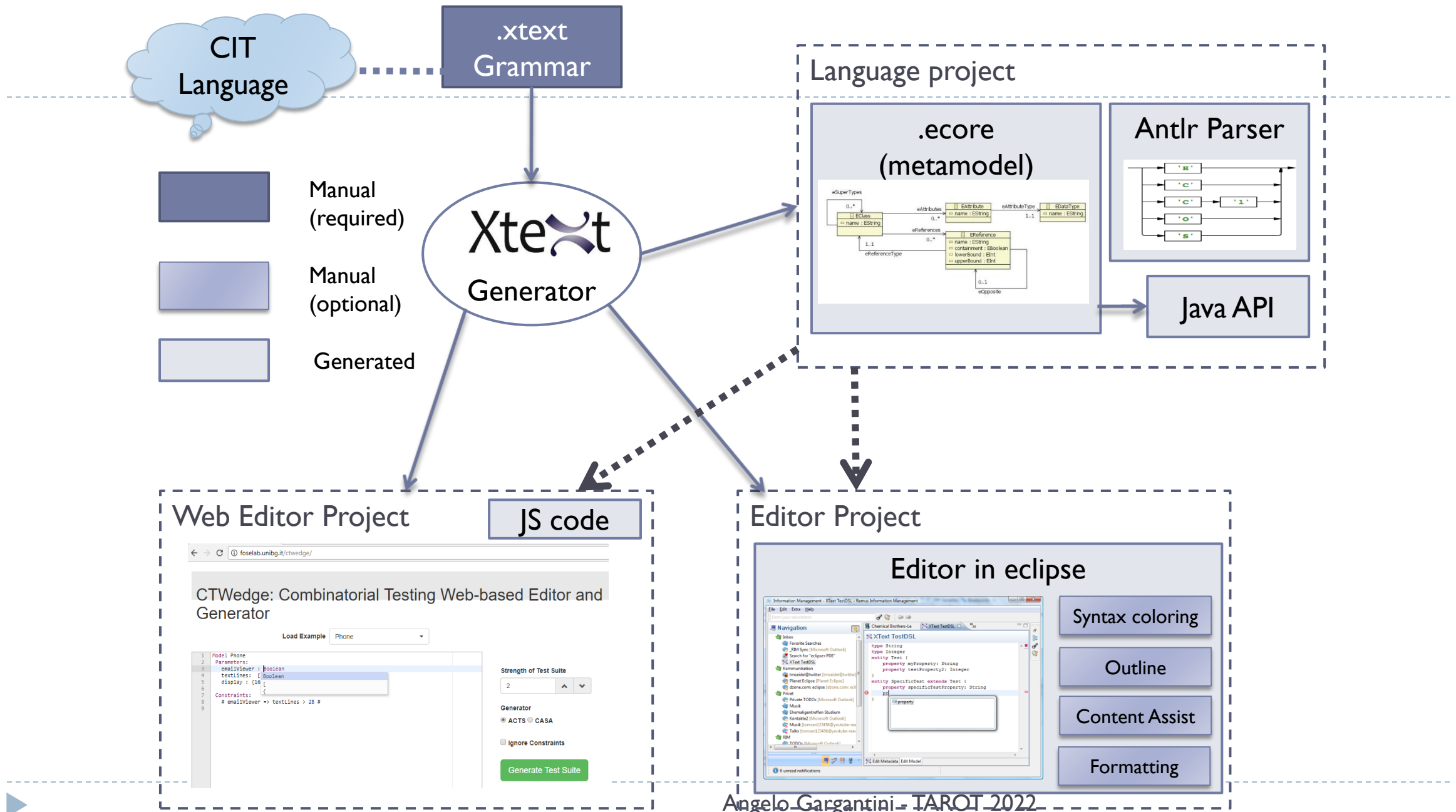
Xtext

Set of tools

3. for importing/exporting CIT problems
4. for generating test suites (by using external tools)

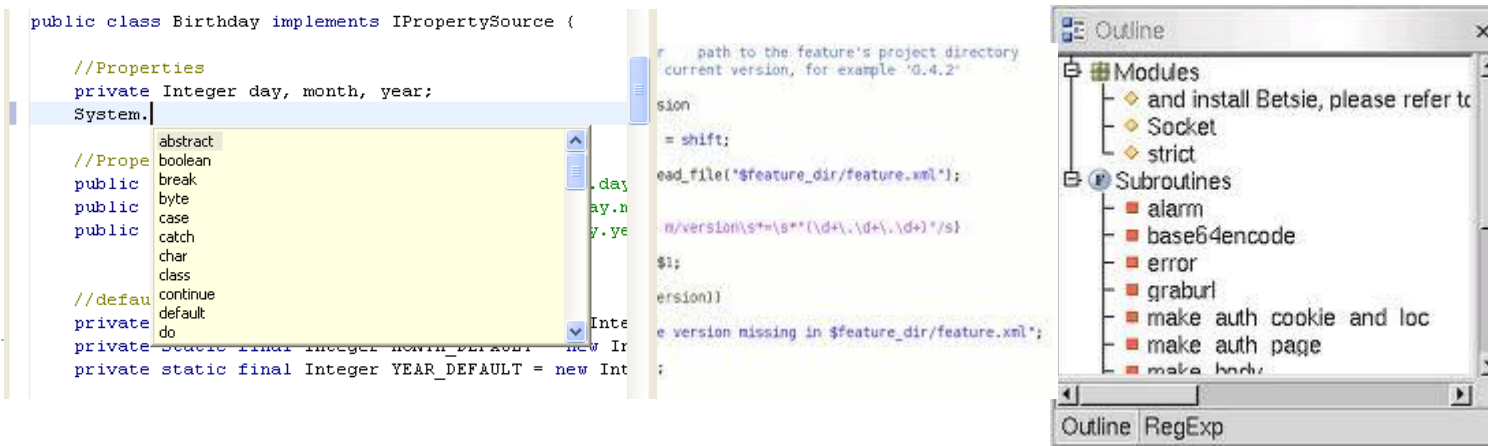
Framework

5. based on the Eclipse Modeling Framework (EMF), library to manipulate combinatorial problems in Java
6. A rich collection of Java utility classes and methods
7. A rich collection of benchmarks



Editor features

- ▶ **Syntax Coloring**
- ▶ Content Assist
- ▶ Template Proposals
- ▶ Rich Hover
- ▶ Rename Refactoring
- ▶ Quick Fixes
- ▶ Outline
- ▶ Folding
- ▶ Hyperlinks for all Cross References
- ▶ Find References
- ▶ Toggle Comment
- ▶ Mark Occurrences
- ▶ Formatting



How to use CTWedge

1. As a eclipse plugin
2. As a web service
 - ▶ <https://foselab.unibg.it/ctwedge/>

modelling Combinatorial Problems

Grammar

► Very similar to EBNF:

CitModel:

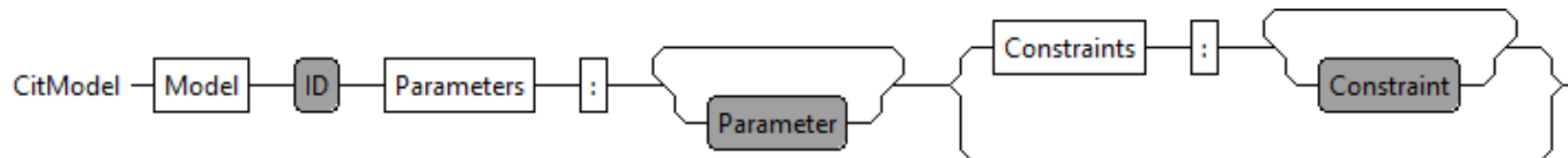
'Model' *name=ID*

//list of parameters

'Parameters' ':' (parameters+=Parameter)+

// constraints

(*'Constraints'* ':' (constraints+=Constraint)+)?;



- Translated to ANTLR



CTWEDGE Language in a glance

Model Model

Parameters:

...

end

Constraints:

...

end

Parameters



Constraints

Example:

*A family of phones,
that can have several
types of cameras,
display,...*

Example

```
/*  
 * This is an example model  
 */  
Model Phone  
  Parameters:  
    emailViewer : Boolean  
    textLines:  [ 25 .. 30 ]  
    display : {16MC, 8MC, BW}  
  
  Constraints:  
    # emailViewer => textLines > 28 #
```


Parameters and their types

- ▶ To describe a combinatorial problem would be sufficient to specify the number of variables and their cardinality.
- ▶ ctwedge language forces the designer to name parameters and to specify their types by listing all the values in their domain.
- ▶ **Choice:** explicit parameter names to facilitate the modeling of real systems and to ease the specification of constraints and seeds

Enumerative for parameters that can take a value in a set of symbolic constants.

the display of the cell phone can be colored (with 16 or 8 millions colors) or black and white,

```
display: { 16MC 8MC BW };
```


Parameters (2)

Boolean for parameters that can be either true or false.

the phone can have an email viewer

emailViewer: boolean;

Numerical values in a range for parameters that take any value in an integer range

Phones have a number of lines between 10 and 30, but only every 5 is valid

textLines: [10 .. 30] step 5;

A list of Numbers for parameters that take any value in a set of integers

The phone has been produced in 2012 and 2013

Year: {2012 2013};

Constraints

- ▶ In ctwedge, we adopt the language of propositional logic with equality and arithmetic to express constraints
- ▶ General Form (GF) constraints
 - ▶ propositional calculus and Boolean operators
 $a \text{ or } b \Rightarrow c \text{ and } d$
 - ▶ equality and inequality

If the phone has an email viewer then

 $\# \text{ emailViewer} == \text{true} \Rightarrow \text{textLines} \geq \text{threshold} \#$
 - ▶ arithmetic over the integers
 - ▶ relational and arithmetic operators for numeric terms
 $\# \text{ textLines} \geq \text{threshold} + 10 \#$
- ▶ A valid test must satisfy all the constraints

TEST GENERATION

Test generation

- ▶ CTWEDGE does not include in itself generators. Currently supports the following test generators, each defined as generator plugin:
 - ▶ AETG is a plugin developed by students following the pseudo code for the greedy algorithm of AETG.
 - ▶ IPO is a plugin developed by us following the pseudo code for IPO.
 - ▶ Random is a simple random algorithm that adds new randomly built tests until all the n-wise combinations are covered.
 - ▶ ACTS is an external test generator tool developed by the NIST.
 - ▶ CASA is an external tool for test generation based on simulated annealing by Myra Cohen and colleagues.
 - ▶ ATGT_SMT is an external tool combining heuristics and SMT solving.
- ▶ Some support constraints, seeds, ...



Logic based

Logic approach to CCIT

- ▶ Testing can be viewed as a particular logic problem
 - 1. **Formalize** the testing problem with a suitable logics
 - ▶ As a set of logical predicates “**test predicates**”
 - 2. **Solve** it by a suitable tool
 - ▶ Reuse techniques and tools used normally for verification
 - ▶ *TESTS AND PROOFS (TAP) manifesto*
- Testing and proving are complementary*

Formalize: Test Predicates

- ▶ Testing criterion defined by a set of *test predicates*
 - ▶ formulas over the state determining if a particular testing goal is reached
- ▶ A *test set T is adequate according to a criterion C* if each test predicate of C is true in at least one state of a test sequence of T
- ▶ Example

I want to test the case in which x is equal to 0

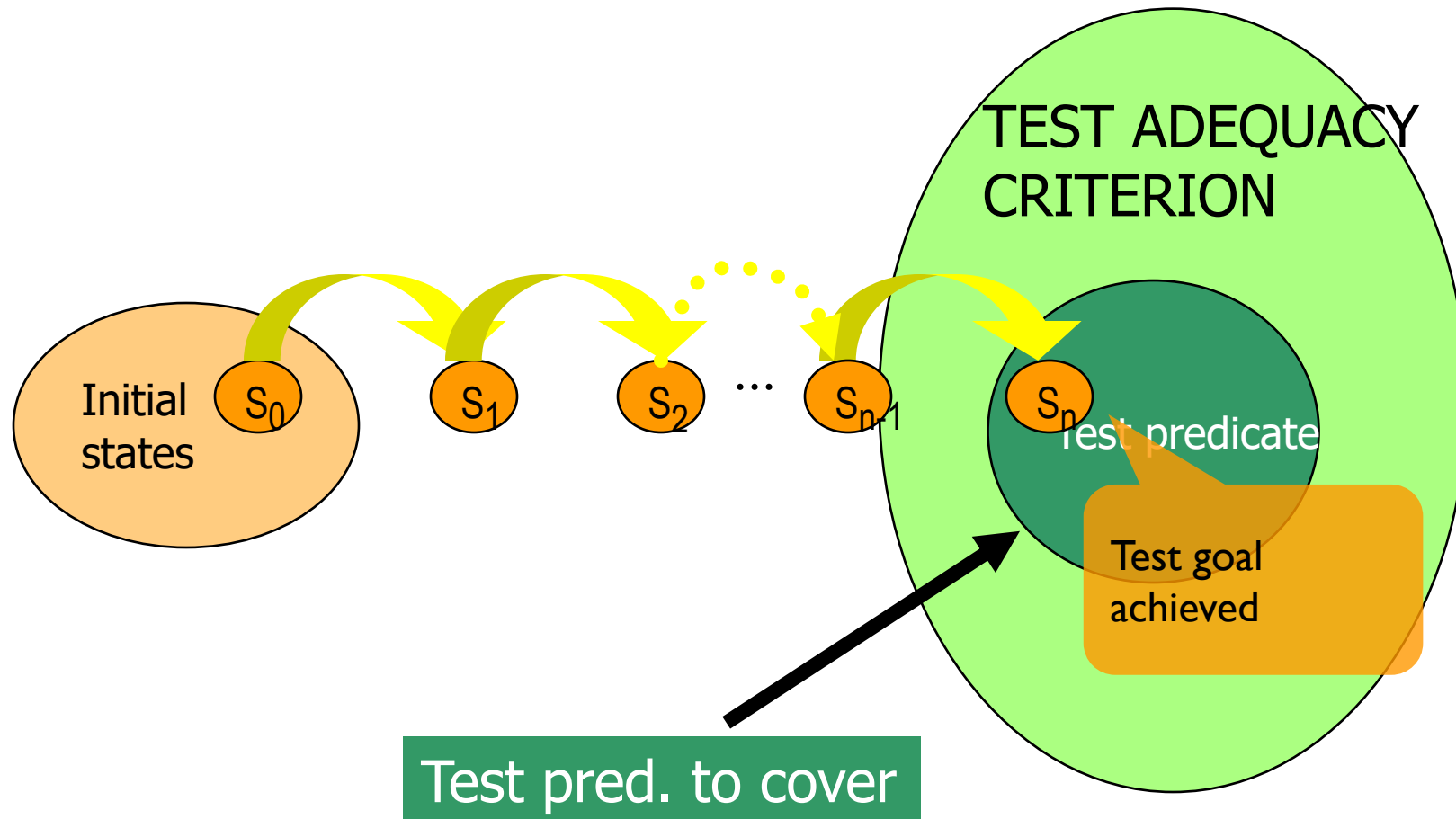
$$x = 0$$



Solve: find a test

- ▶ Find a behavior of the system (as modeled) that satisfies the **model and the test predicate**
- ▶ Behavior:
 - ▶ A combination of inputs for combinatorial testing
 - ▶ A **sequence** of inputs for reactive systems

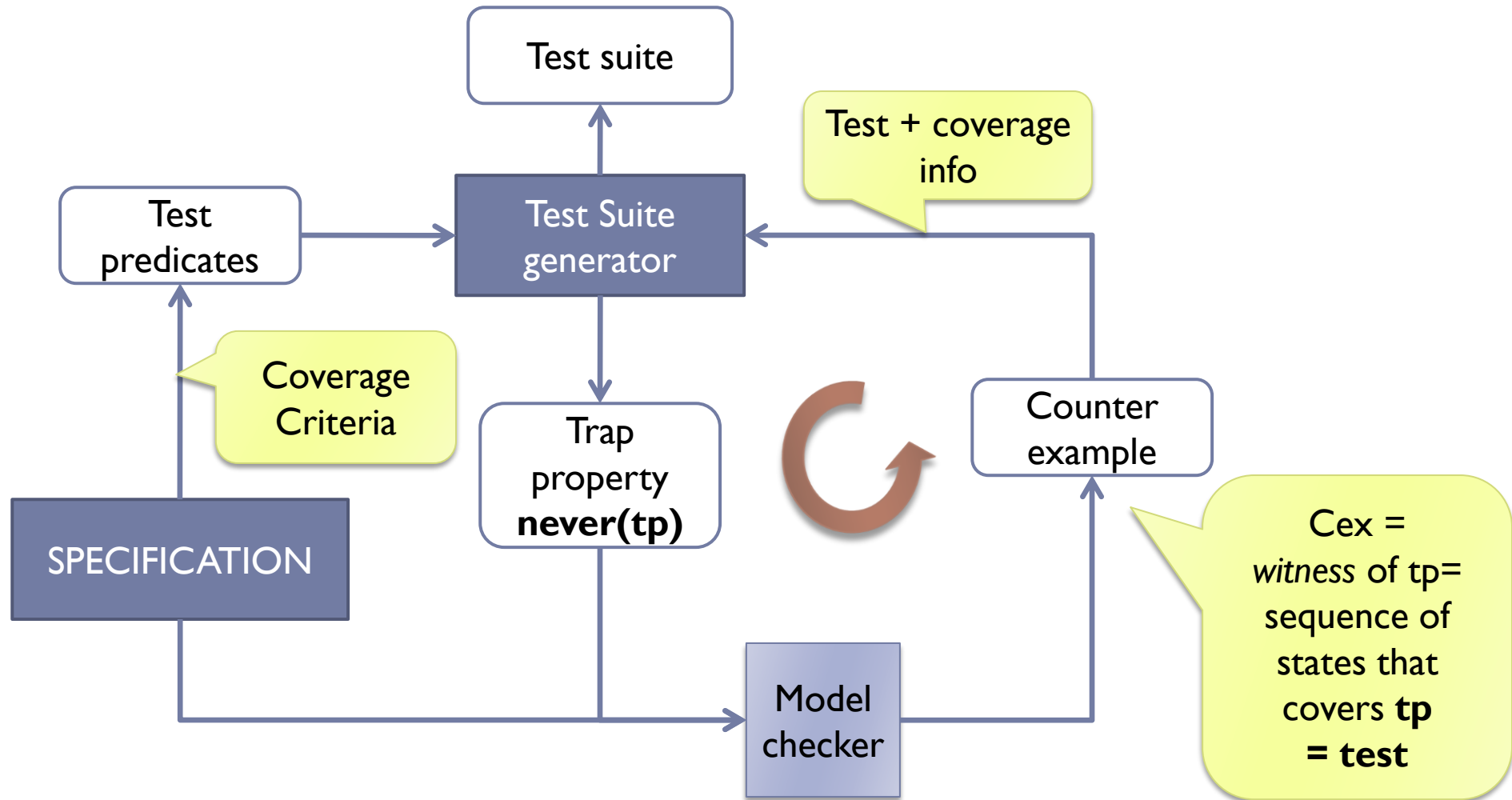
Test predicate \rightarrow Test sequence



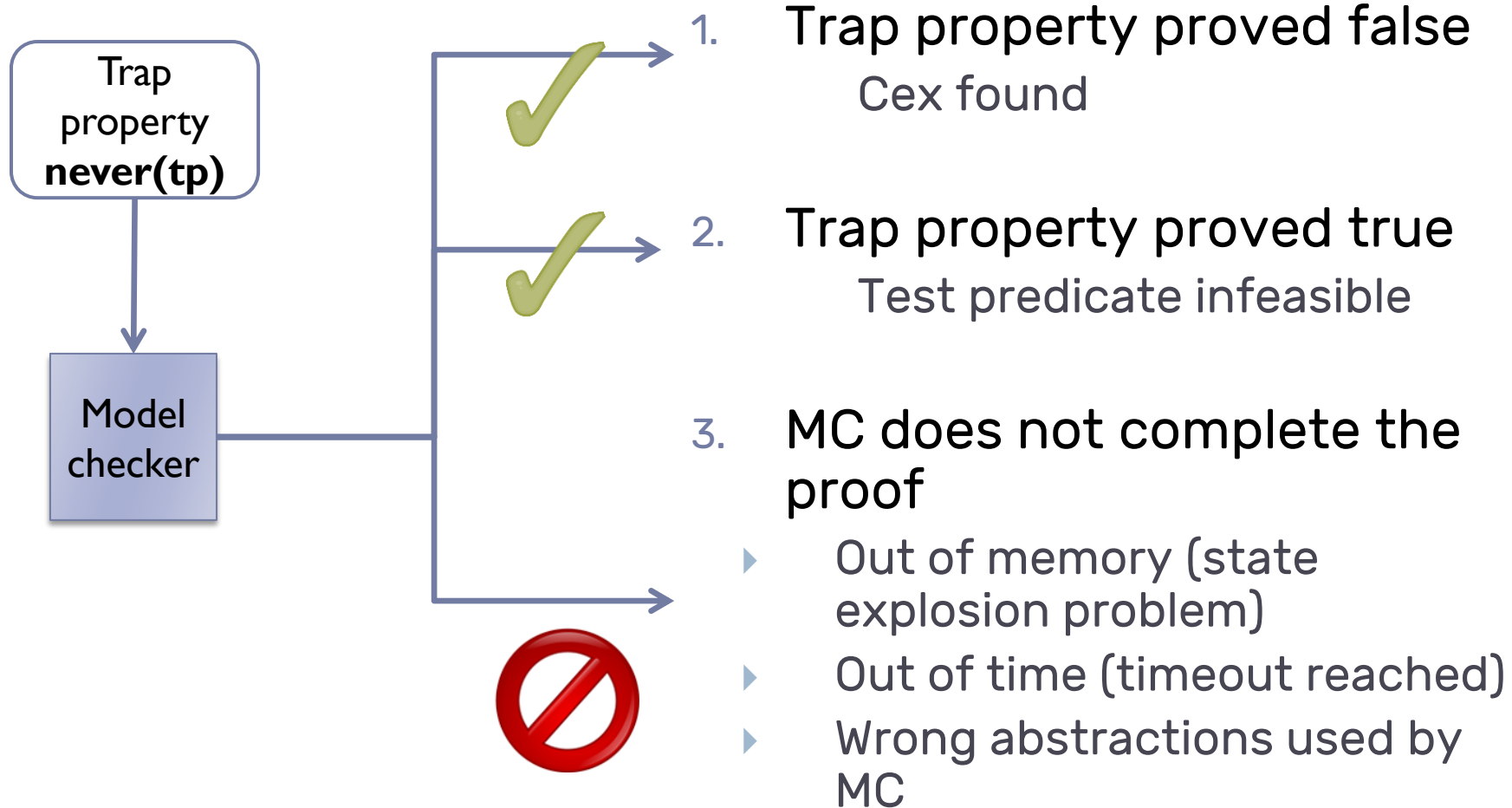
Test generation by model checking

- ▶ Model checking can be used for model-based tests generation
- ▶ several notations, systems, coverage criteria (data flow, structural, mutation, ...) and using several model checkers
 - ▶ [FAW09] Fraser, Ammann, and Wotawa. *Testing with Model Checkers: A Survey*. *Journal for Software Testing, Verification and Reliability*, 2009
 - ▶ [GH99] Gargantini, Heitmeyer. *Using model checking to generate tests from requirements specifications*. FSE/ESEC, 1999
 - ▶ [ABM98] Ammann, Black, Majurski. *Using model checking to generate tests from specifications*. *Formal Engineering Methods*, 1998

Mc for test generation



Some limits



Open issues

- ▶ Several questions remain unanswered about the **model checking** itself:
 - ▶ Which model checking technique is *best* suited for test case generation?
 - ▶ Which optimizations are useful in the context of test case generation?
 - ▶ What impact does the choice of model checking technique have on the test case generation?
- ▶ Can be applied to Constrained Combinatorial Interaction Testing (CCIT)?

1. Formalization of the problem

- ▶ formalize by means of test predicates
- ▶ pairwise testing:
- ▶ express each pair as a corresponding logical expression, a test predicate
 - ▶ $p1 = v1 \odot p2 = v2$
 - ▶ $p1$ and $p2$ are inputs, $v1$ and $v2$ their values
- ▶ Similarly, the t-wise coverage can be modeled by a set of test predicates, each of the type
 - ▶ $p1 = v1 \odot p2 = v2 \wedge \dots \wedge p_t = v_t$

Solve find a test = logical model

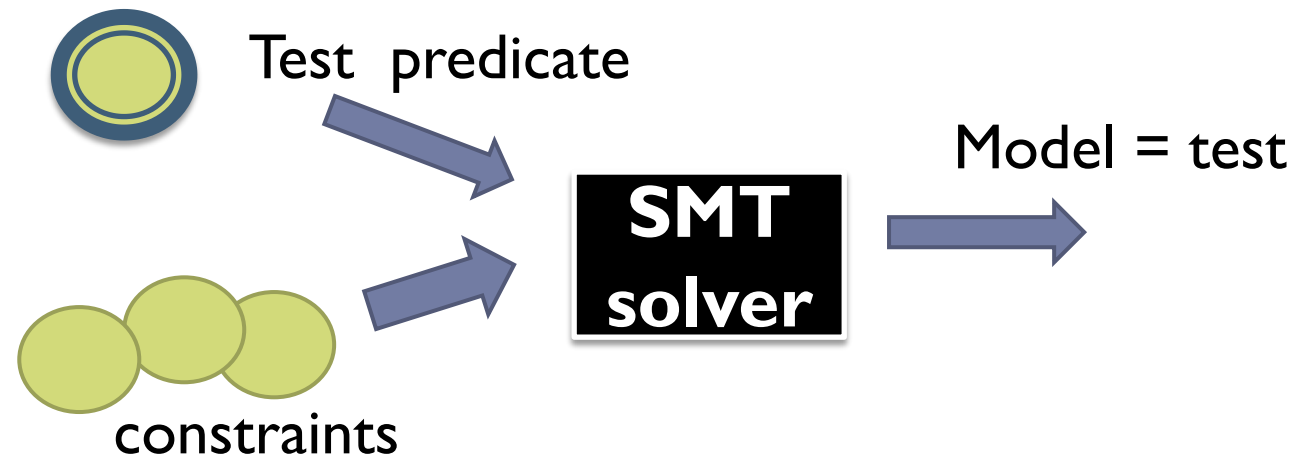
- ▶ **test**: an assignment to all the inputs of the system
 - ▶ A test is not a sequence of inputs,
 - ▶ No model is given, so the test must satisfy only the test predicate
- ▶ a test ts covers a test predicate tp if and only if it is a *model* of tp :
 - ▶ $ts \models tp$
- ▶ Finding a test suite \rightarrow finding a model for each test predicate

Adding constraints

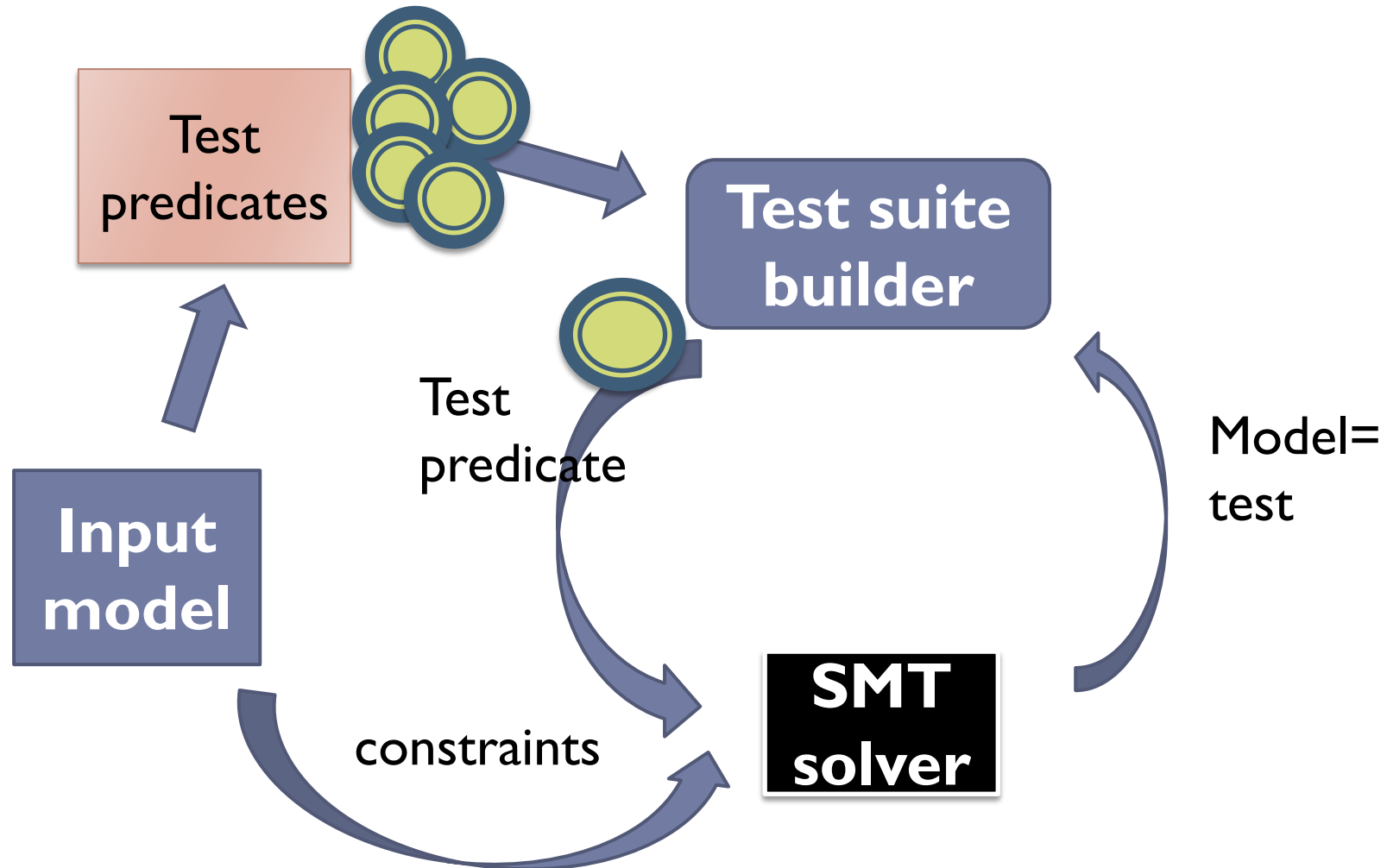
- ▶ With the constraints $c_1 \dots c_n$, a test is a model of tp and the constraints c_i
 - ▶ $ts \models tp \wedge c_1 \wedge \dots \wedge c_n$
- ▶ the constraints become first class citizens and they can be represented by logical expressions too
- ▶ Finding tests becomes a problem of finding a model of a complex expression
 - ▶ many techniques like, constraint solvers, model checkers [AFM08], SAT solver, MDD, SMT solvers ...

Yices [csl.sri.com]

- ▶ Efficient SMT solver that decides the satisfiability of arbitrary formulas
 - ▶ Expressive language for constraints and input models
 - ▶ Powerful algorithms to find models

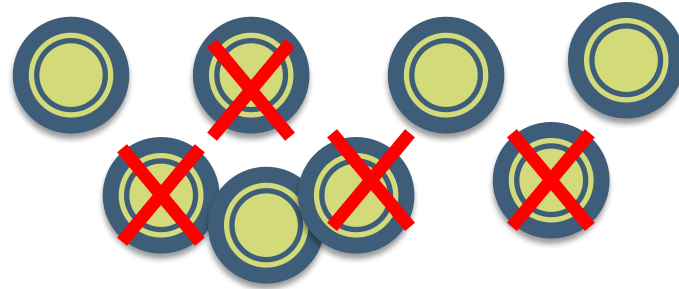


Test Suite generation a simplified version

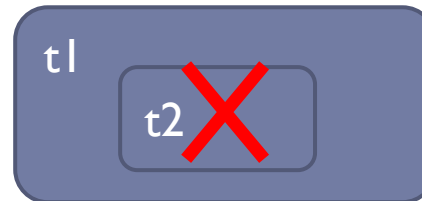


Monitoring and reduction

- ▶ **Monitoring:** A test covers may cover several test predicates which can be removed from the pool



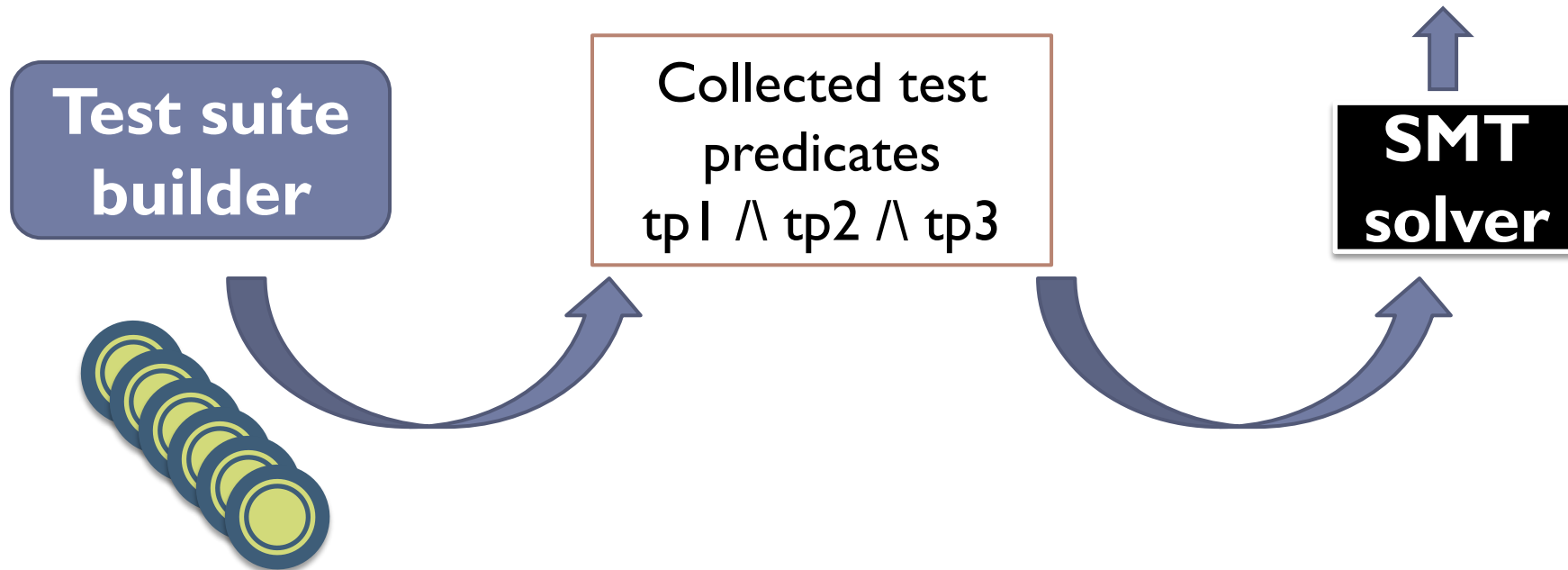
- ◎ **Reduction:** at the end, a minimal set of tests that cover all the test predicates can be found by a greedy algorithm



Collecting

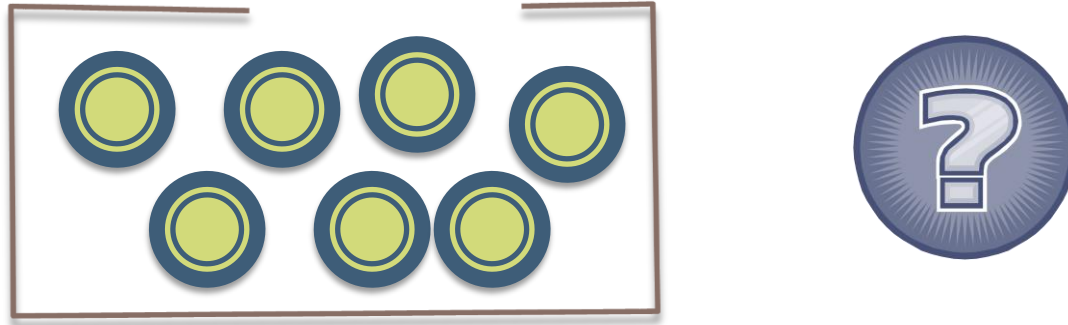
Instead of one test for every tp, collect the tps to build a conjoint

Model = test that covers all the test predicates collected



Ordering

- ▶ In which order test predicates can be collected?



- ◎ Possible order policies:
 1. the order in which tp are generated
 2. random order
 3. Find some heuristics to find a better order

Open problems

- ▶ Ordering
- ▶ Using other techniques to solve the constraints
 - ▶ Other SMT/SAT ... solvers
 - ▶ Non linear constraints?
 - ▶ Temporal constraints?



CT – some directions

CT some new ideas -CT problems revisited

- ▶ Expressivity of combinatorial models
 - ▶ Soft constraints – like testing requirements
 - ▶ Probabilities
 - ▶ Existing tests
 - ▶ Mixed CT
- ▶ Comparison with other testing criteria
 - ▶ Fault detection capabilities
 - ▶ Comparison with random

CT some new ideas

- ▶ CT used in conjunction with other testing approaches
- ▶ With ontologies
 - ▶ Ask Franz Wotawa
- ▶ With Machine Learning
 - ▶ Algorithms for ML
 - ▶ Explainability of solutions
- ▶ CT and software product lines
 - ▶ Rob tomorrow?

Fault localization of CT

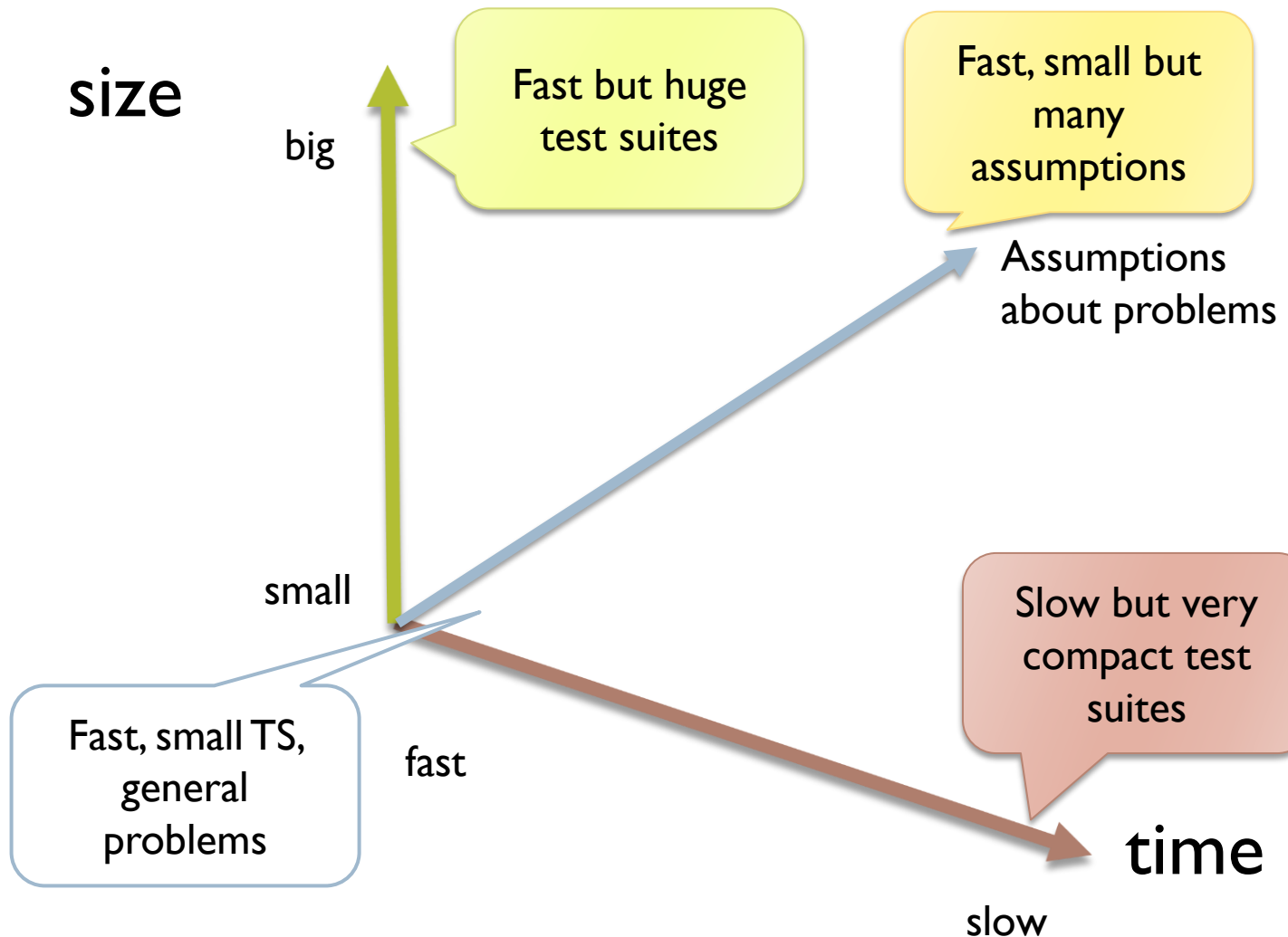
- ▶ CT can detect a fault and then?
- ▶ Can we localize which combination causes the fault

A	B	C	
1	2	3	pass
1	2	4	fail

Which pair casues the fault?

- We can exclude (1,2)
- BUT (1,4) and (2,4)?

Test generation



- ▶ New algorithms can improve the state of the art
- ▶ Multi objective
- ▶ Let's compete

Conclusions

- ▶ Combinatorial testing
- ▶ Useful for **testers**
 - ▶ Better than random, easier than structural
 - ▶ Many tools available
- ▶ Open field for **researchers**
 - ▶ Still room to improve old methods or invent new ones
 - ▶ Make the technique more usable (constraints,)