# Efficient Combinatorial Test Generation based on Multivalued Decision Diagrams

Angelo Gargantini and Paolo Vavassori

Dip. di Ingegneria, University of Bergamo, Italy

{angelo.gargantini,paolo.vavassori}@unibg.it

**http://cs.unibg.it/gargantini/**


HVC 2014 - Haifa Verification Conference 2014

# Outline

1. Combinatorial interaction testing
2. Novelty of our approach for CIT test generation:
   - Using Multivalued Decision Diagrams (MDDs)
   - Greedy + Weighting compatibility
3. Experiments

# Combinatorial interaction testing

- Combinatorial Interaction Testing (CIT) helps tester to find defects due to the interaction of components or inputs.
- It is based on the assumption that faults are generally caused by interactions among parameters.
- CIT tests the interaction in a systematic way.
- For instance, pairwise testing requires that every pair of parameter value be tested at least once.
  - It can be generalized by the t-way testing.
  - CIT has been proved to be very effective in finding faults.

Disclaimer: BTW, Haifa IBM research lab (Segall, Tzoref-Brill, …) is very active in this area, including using BDDs, dealing with constraints/restrictions …

# Example

- We exploit in this work our framework, CitLab
  - language/editor
  - Validation
  - APIs + benchamarks ….
  - Interface to other tools (CASA, ACTS, …)
  - https://code.google.com/a/eclipselabs.org/p/citlab/

- Example, a simple phone:

```
Model phone
 Parameters:
   Enumerative display { 16MC 8MC BW };
   Enumerative frontCamera { 2MP 1MP NOC };
   Boolean emailViewer;
 end
 Constraints:
   # emailViewer => display != BW #
End
```

Constrained CIT = CCIT

Constraint: If the phone has an email viewer, the display cannot be black and white

# A «novel» test generation algorithm

- **YET ANOTHER TEST GENERATION ALGORITHM !**
- Classical greedy algorithm producing a test at the time.
  - for a single test, it chooses an *optimal* parameter and assigns an *optimal* value to it until a test is complete.
- BUT:

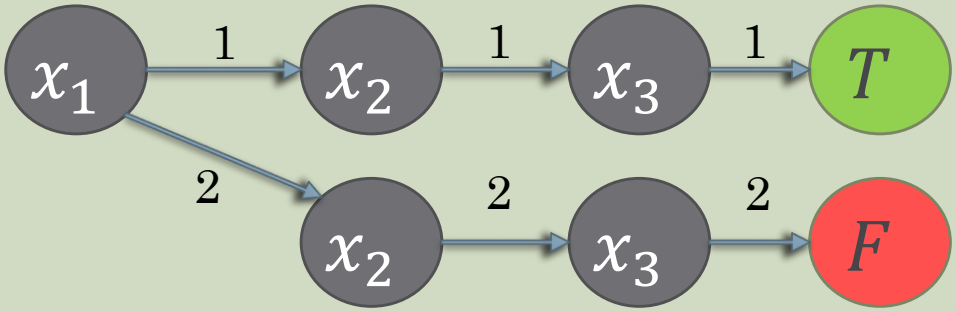
1. We employ a data structure, **Multivalued Decision Diagram** (MDD),
   - to represent inputs, their domains, and constraints over those inputs
2. We soften the classical greedy algorithm
   - reducing the importance of the number of tuples covered by the test currently built,
   - We **weight** parameters and tuples depending on the constraints

# MDD for combinatorial testing

# What is a Multivalued Decision Diagram?

- A decision diagram is a graph that represents a function $f : \mathrm{D} \rightarrow B$ over $n$ variables where $\mathrm{D} = \mathrm{D}_1 \times \ldots \times \ldots \mathrm{D}_n$ and $B$ is the Boolean domain, i.e., $B = \{\mathbf{F}, \mathbf{T}\}$.
  - In Binary Decision Diagram, the domain of variables are Boolean
- In MDDs, every variable can have a different domain.
- A MDD is a directed acyclic graph. The graph has only two terminal nodes each labeled F or T. Every non-terminal node is labeled by an input variable $x_i$ and has $|D_i|$ outgoing labeled edges

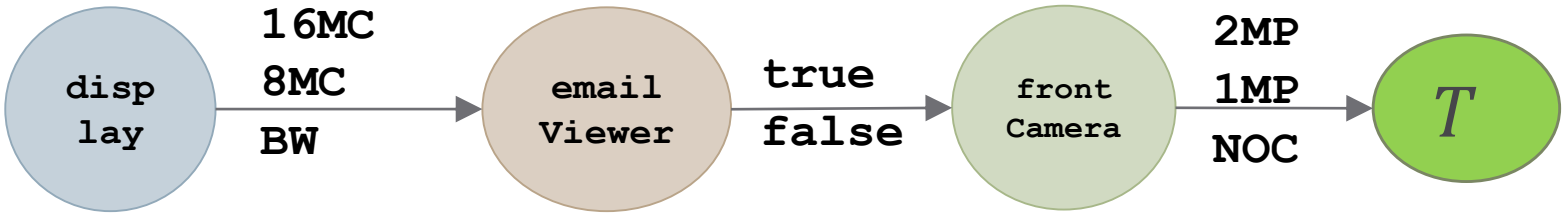
Example:
$x_1 \in \{1,2,3\}$
$x_2 \in \{1,2,3\}$
$x_2 \in \{1,2,3\}$
$f(1,1,1) = \mathrm{T}$
$f(2,2,2) = \mathrm{F}$ …



# MDD for the combinatorial problem

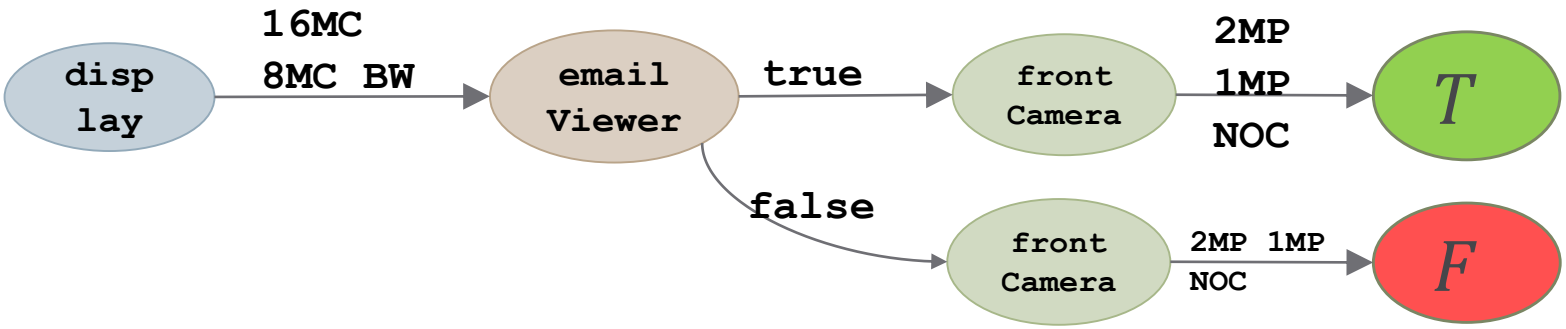
If one ignores the constraints, a combinatorial model can be very easily represented by an MDD

```
Enumerative display { 16MC 8MC BW };
Boolean emailViewer;
Enumerative frontCamera { 2MP 1MP NOC };
```

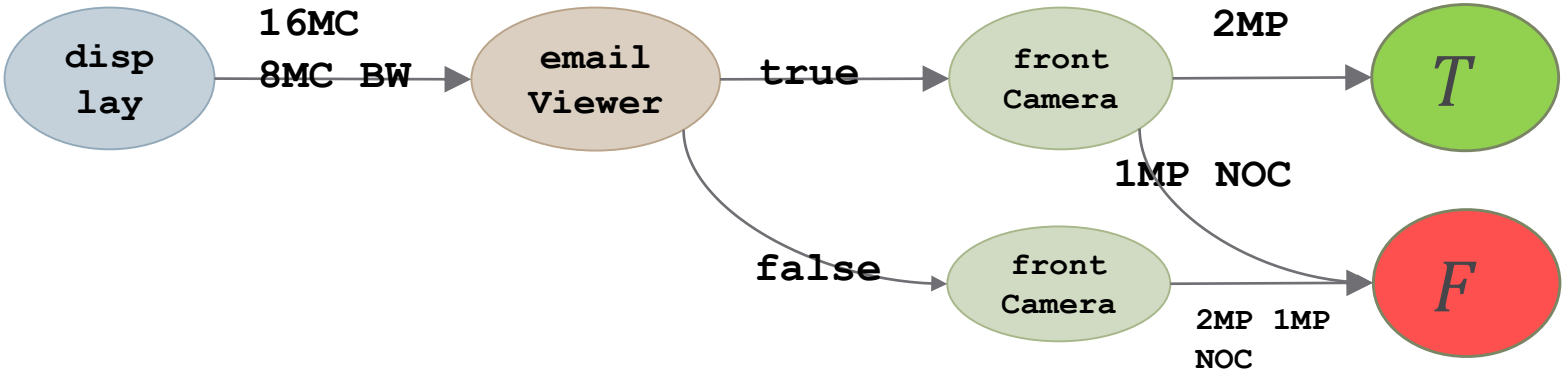


# Using MDDs for = and tuples

- Single value equality **`emailViewer = true`**

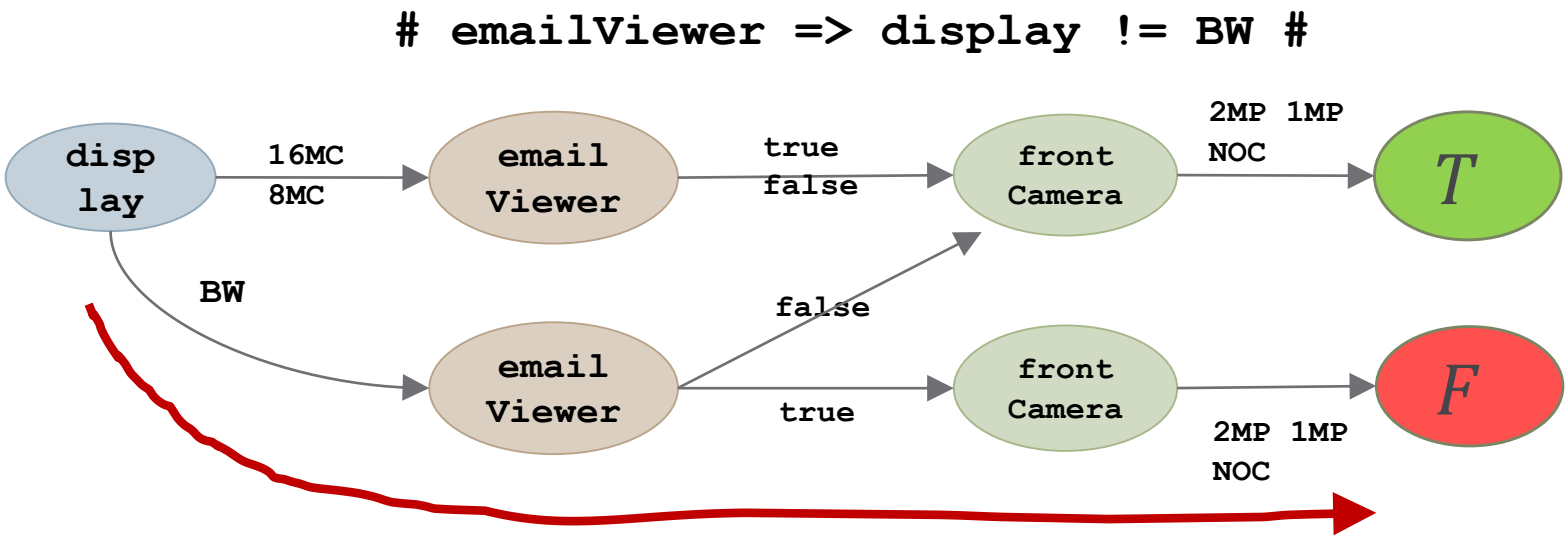



- Tuple **`emailViewer = true , frontCamera = 2MP`**



# MDDs for constraints and constrained problems

- Constraints can be represented in MDD as well
  - No CNF required
  - the usual logical operations can be represented by operation between MDDs
    - $f_1 \wedge f_2$ becomes $M_1 \sqcap M_2$ ….
- The entire MDD can be modified in order to include the constraints



# MDDs vs BDDs (vs Logic solvers)

- MDD are more efficient than BDDs [literature]:
  - More **compact** to represent domain with *n* values
    - One level for each variable
  - No need of representation of the implicit constraints
- More efficient than logic solvers in several operations
  - **to count** of number of models
  - to find intersections of constraints (further experiments?)

# An MDD-based algorithm for CCIT

# Basic algorithm

**Input:** $M_{VS}$:
MDD for the model with the constraints
**Output:** $R$:
set of MDDs representing the test set
$T_{TC} \leftarrow feasibleTuples(M_{VS})$
$R \leftarrow \emptyset$
**while** $T_{TC} \neq \emptyset$ **do**
$M_{nc} \leftarrow M_{VS}$
$P \leftarrow sortParamList(T_{TC})$
**for all** $P_i \in P$ **do**
$value \leftarrow chooseBest(P_i, M_{nc}, T_{TC})$
$M_{nc} \leftarrow M_{nc} \sqcap P_i = value$
**if** $|M_{nc}| = 1$ **then break end if**
**end for**
$T_{TC} \leftarrow removeCoveredTuples(M_{nc})$
$R \leftarrow R \cup M_{nc}$
**end while**

It builds one test at the time until all the testing requirements are achieved.

It proceeds in a greedy manner: it chooses one *optimal* parameter and its *optimal* value and it adds the assignment to the test to be built

USE OF MDDs

- To represent the test
- Check if P can take a value
- Add assignment
- Check if it is already a test

# Weighting compatibility

- Classical greedy policy:
  best **value** = it covers most uncovered tuples
- It is inefficient
  - it tends to leave at the end all the tuples that are "difficult" to cover, because the constraints limit the number of valid test cases that can cover them.
  - last generated tests cover only a few tuples not covered yet,
  - leading to bigger test suites
- **PROPOSED SOLUTION**
  - Weighting compatibility between tuples
  - Every tuple has a weight

# How to weight tuples

- We propose to weight every tuple depending on its compatibility with respect to the other tuples not covered yet considering also the constraints.

**for all** $(T_i, T_j) \in T_{TC} \times T_{TC}$ **with** $i < j$ **do**
**if** $M_{VS} \sqcap T_i \sqcap T_j = \emptyset$ **then**
$weight(T_i)$ ++; $weight(T_j)$ ++;
**end if end for**

Use MDD to check compatibility

**`# emailViewer => display != BW #`**

| weight | … | `frontCamera = 2MP`<br>`display = BW` | |
|---|---|---|---|
| … | | | |
| `emailViewer = true`<br>`frontCamera = 2MP` | | +++++ | |
| … | | | |

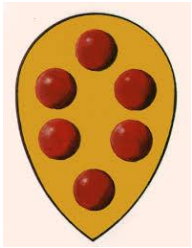
# Other optimizations

- **Threshold**
  - If the tuples are greater than THRESHOLD, weight compatibility in a classical way (simply choose the value that covers more uncovered tuples)
- **Repetitions**
  - The algorithm in case of tie randomly chooses a value for a parameter
  - REPEAT the algorithm with a different seed,
  - Policy:
    - Generate at least $R_{MIN}$ test suites, stop if for $R_{BETTER}$ times the size does not decrease, and never pass $R_{MAX}$ repetitions

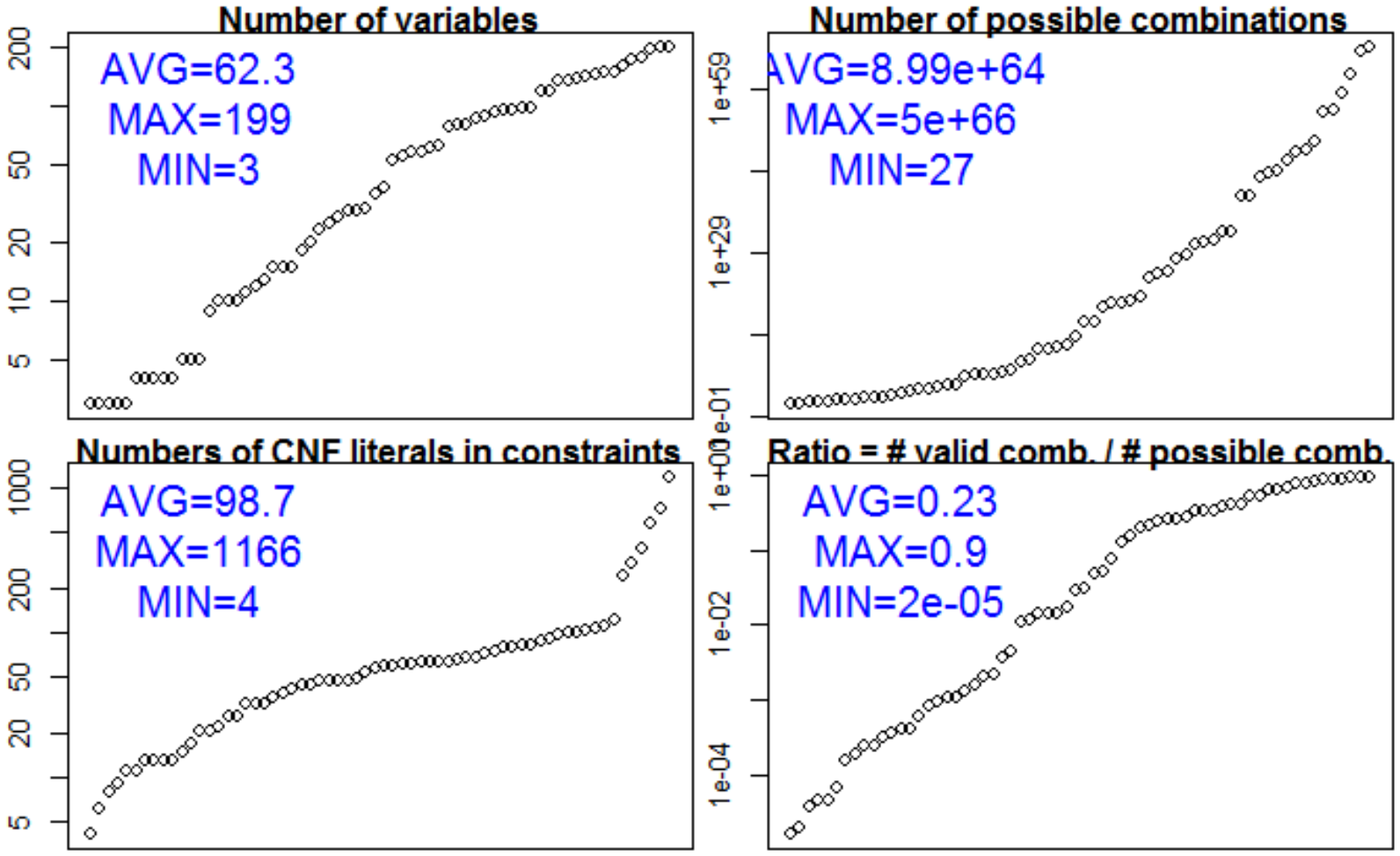
# Experiments

# Benchmarks

- As benchmarks for CCIT problems we have gathered 115 models with constraints taken from the literature (Casa, FoCuS, ACTS, and IPO-S) and from SPLOT SPLs repository, and used (in subsets) also by many other papers.
  - Thanks to CitLab we were able to collect a standard set of CIT problems and use them with different tools
- Implemented a tool called MEDICI
  - In C++, using meddly (open source library for MDDs)
- 50 runs for each model



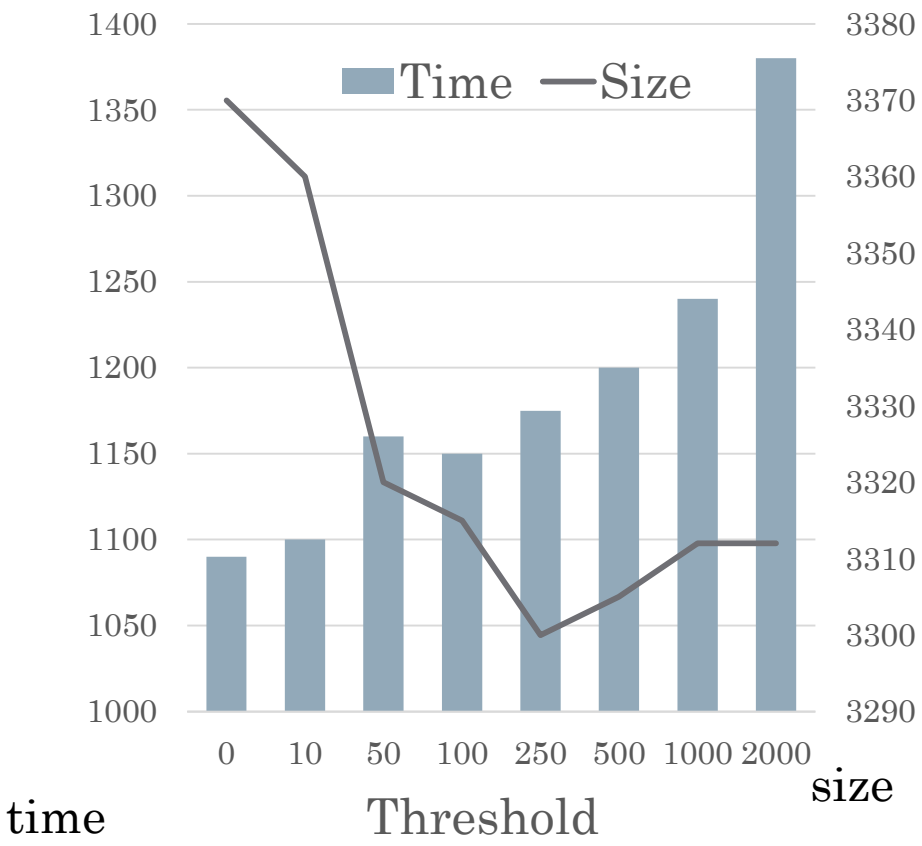
# Benchmark data



# How to evaluate a combinatorial generation algorithm

- Two main factors:
- Size
  - Let $\overline{size_m}$ be the average of the test suite size for model m over all the runs
  - $\boldsymbol{size} = \Sigma\, \overline{size_m}$
- Time
  - Let $\overline{time_m}$ be the average of the test suite size for model m over all the runs
  - $\boldsymbol{time} = \Sigma\, \overline{time_m}$

# Optimal threshold value when weighting compatibility



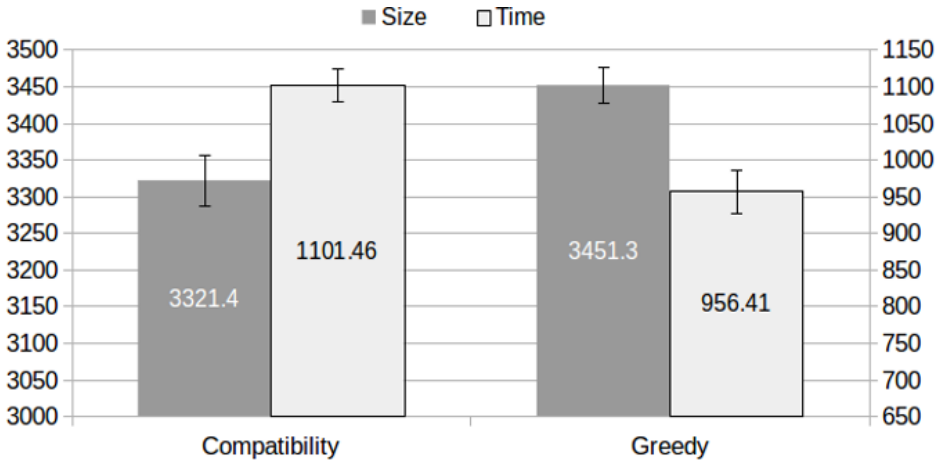

Weight compatibility only if the number of tuples is below a threshold

0 → classical greedy algorithm
2000 → always weighting the compatibility

For small threshold is faster but produces bigger test suites
For big threshold is slow but produces small test suites

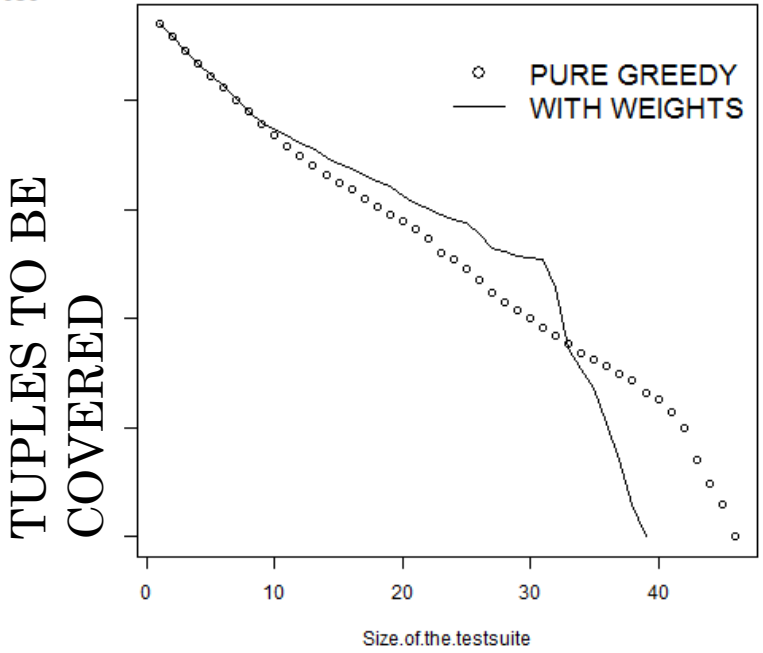
We choose **250**

# Weighting compatibility

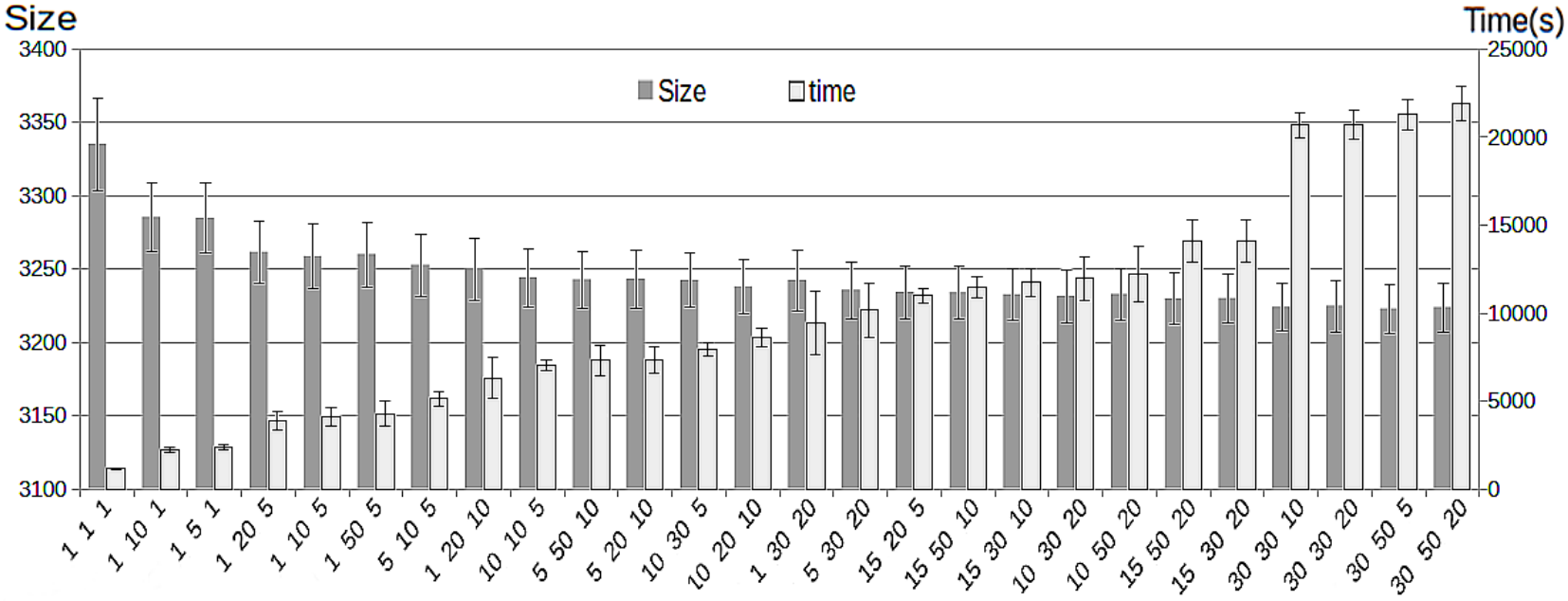



Compatibility reduces the test suite at the end of the generation

Using the compatibility leads to
- smaller test suites (4%)
- an increase of the time (15%)

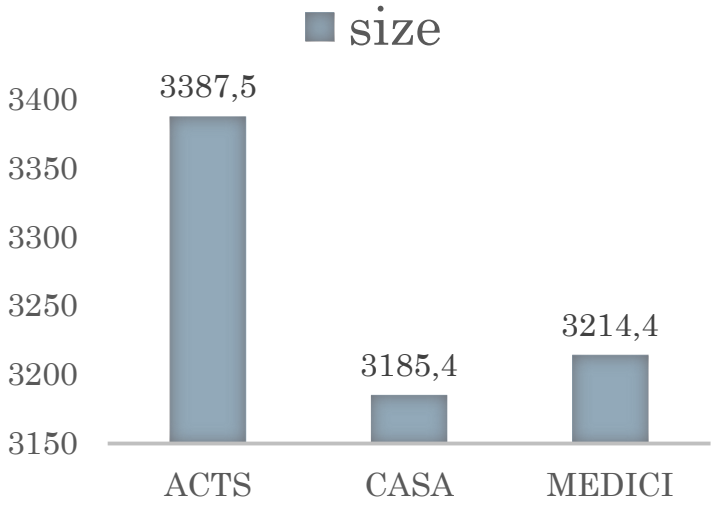


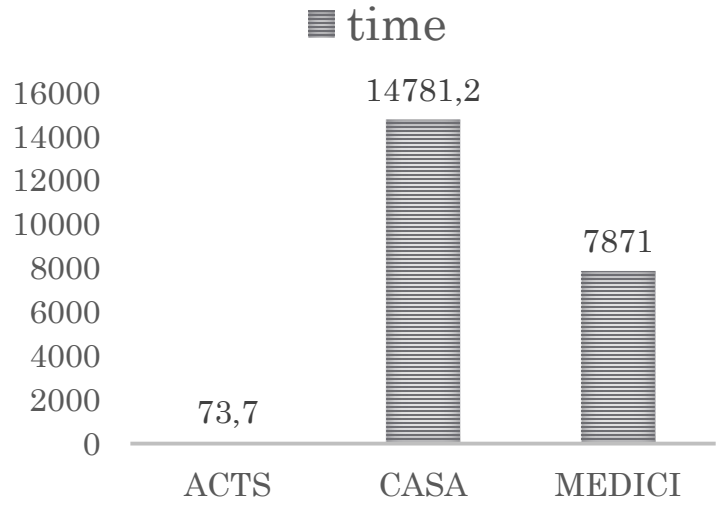
# Number of repetitions




- Increasing the number of repetitions increase the time and decrease the number of tests.
- Typical multi-objective optimization: which is the best compromise?
- Default : $R_{MIN} = 10, R_{MAX} = 30$, and $R_{BETTER} = 5$

# Comparison with CASA and ACTS




ACTS is the faster but produces the biggest test suite

CASA is the slowest but produces the smallest test suite

MEDICI is in the middle

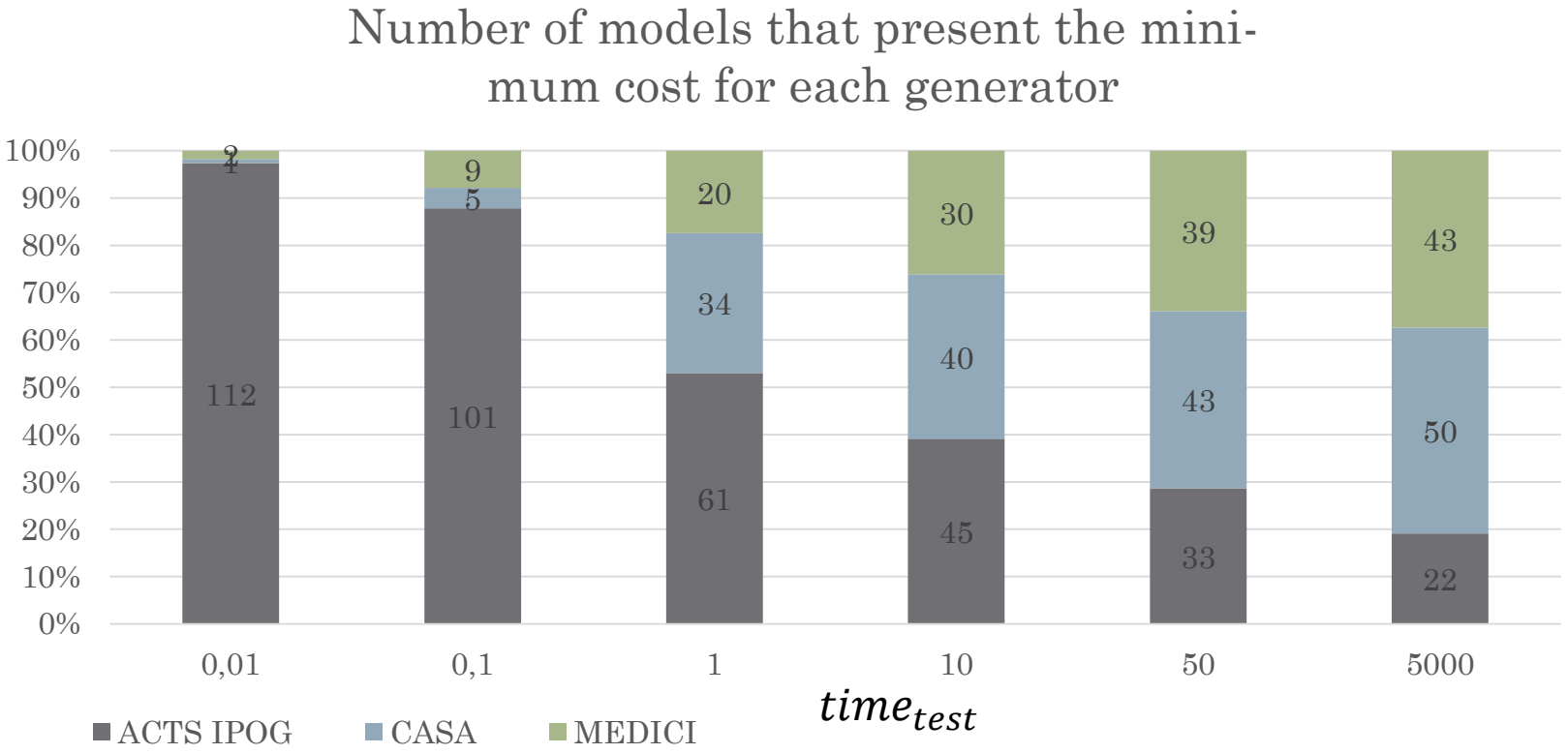
Which one is better?

It depends on the cost of executing each test

# A cost-based comparison

Computing the cost

$$cost = time_{total} = time_{generation} + size \times time_{test}$$



# Conclusions

- **MDDs for CCIT**
  - MDDs can be efficiently used to represent and solve combinatorial models in the presence of constraints
- **Weighting tuples**
  - Classical greedy policies are not efficient
  - Not all tuples are equal
- **Implementation: MEDICI**
  - Integrated in CitLab
  - It can compete with state of the art CCIT tools

**Thank you**