GENERATING MINIMAL FAULT DETECTING TEST SUITES FOR BOOLEAN EXPRESSIONS

ANGELO GARGANTINI
UNIVERSITY OF BERGAMO
ITALY
GORDON FRASER
UNIVERSITY OF SAARLANDES
GERMANY

A-MOST 2010

OUTLINE

1. Intoduction about testing of (DNF) logic expressions
   - Boolean expressions: where to find them, how to test them
   - For boolean specification in DNF
     - Fault classes
     - Classical testing criteria
2. A new way of generating fault detecting tests
   - How to discover a fault
   - Using SAT solvers to generate tests
   - Optimizations
3. Experiments

LOGIC PREDICATES AND CLAUSES

A predicate is an expression that evaluates to a Boolean value

Predicates can contain

- boolean variables
- non-boolean variables that contain >, <, ==, >=, <=, !=
- boolean function calls

Internal structure is created by logical operators

$\neg$ the negation operator
$\wedge$ the and operator
$\vee$ the or operator
$\rightarrow$ the implication operator
$\oplus$ the exclusive or operator
$\leftrightarrow$ the equivalence operator

A clause is a predicate with no logical operators

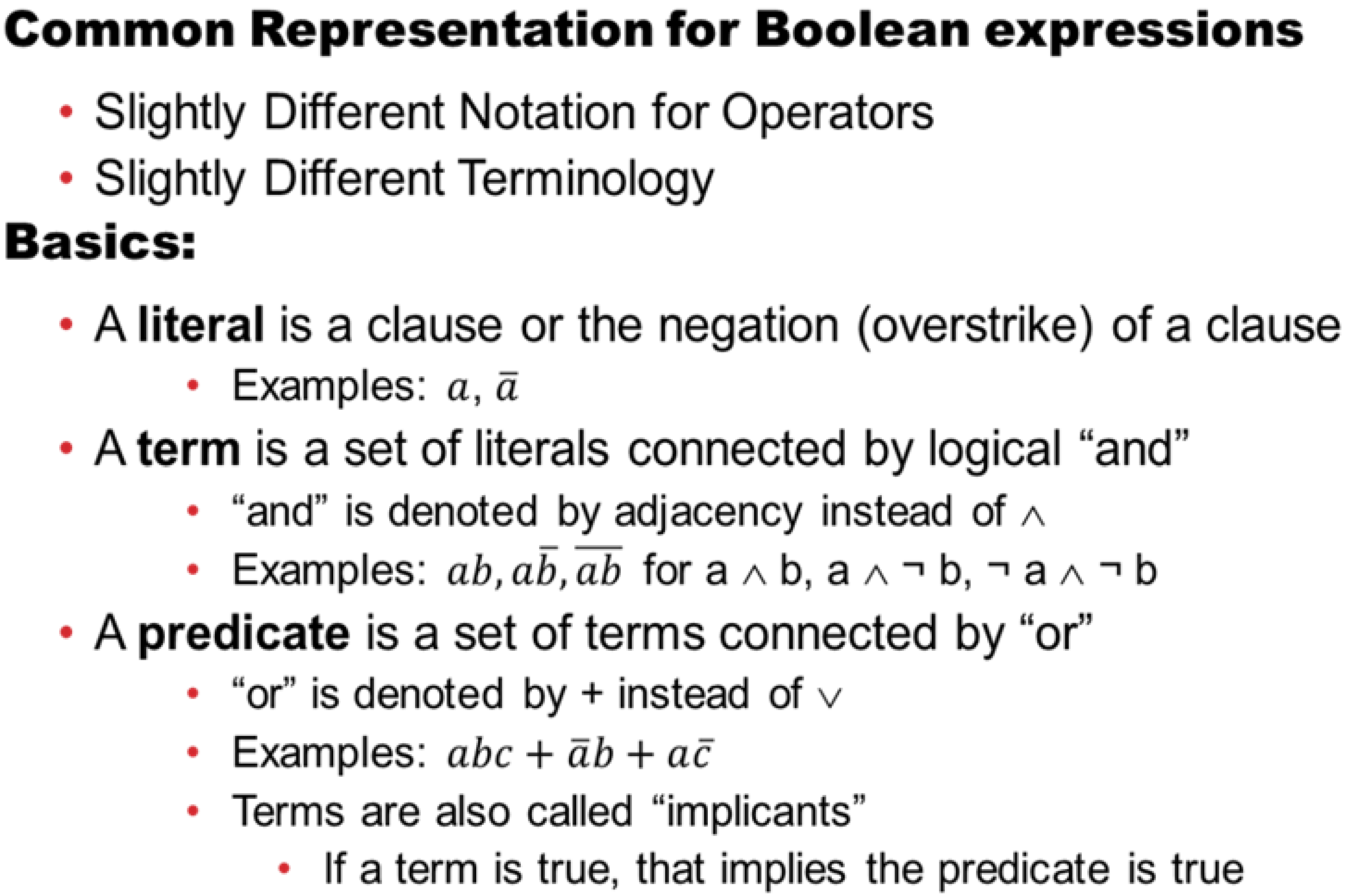
EXAMPLES

$(a < b) \lor f(z) \land D \land (m >= n*o)$

Four clauses:

- (a < b) – relational expression
- f (z) – boolean-valued function
- D – boolean variable
- (m >= n*o) – relational expression

DISJUNCTIVE NORMAL FORM



Common Representation for Boolean expressions

- Slightly Different Notation for Operators
- Slightly Different Terminology

Basics:

- A **literal** is a clause or the negation (overstrike) of a clause
    - Examples: $a, \bar{a}$
- A **term** is a set of literals connected by logical "and"
    - "and" is denoted by adjacency instead of ∧
    - Examples: $ab, a\bar{b}, \overline{ab}$ for a ∧ b, a ∧ ¬ b, ¬ a ∧ ¬ b
- A **predicate** is a set of terms connected by "or"
    - "or" is denoted by + instead of ∨
    - Examples: $abc + \bar{a}b + a\bar{c}$
    - Terms are also called "implicants"
        - If a term is true, that implies the predicate is true

FAULT CLASSES FOR BOOLEAN EXPRESSIONS

There exist typical errors done by programmers

Errors cause faults in the expression

- Faults grouped in fault classes
- For DNF expressions there classical fault classes

DNF FAULT CLASSES

ENF: Expression Negation Fault	**f =** $ab + c$	**f' =** $\overline{ab + c}$
TNF: Term Negation Fault	**f =** $ab + c$	**f' =** $\overline{ab} + c$
TOF: Term Omission Fault	**f =** $ab + c$	**f' =** ab
LNF: Literal Negation Fault	**f =** $ab + c$	**f' =** $\bar{a}b + c$
LRF: Literal Reference Fault	**f =** $ab + bc$	**f' =** $ad + bc$
LOF: Literal Omission Fault	**f =** $ab + c$	**f' =** $a + c$
LIF: Literal Insertion Fault	**f =** $ab + c$	**f' =** $ab + bc$
ORF+: Operator Reference Fault	**f =** $ab + c$	**f' =** abc
ORF*: Operator Reference Fault	**f =** $ab + c$	**f' =** $a + b + c$

Key idea is that fault classes are related with respect to testing:

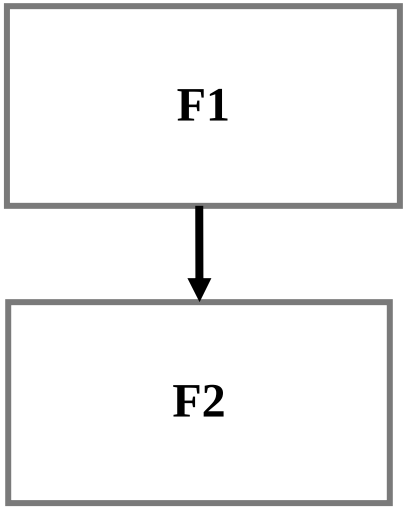
- Test sets should guarantee to detect certain fault classes

FAULT CLASS HIERARCHY

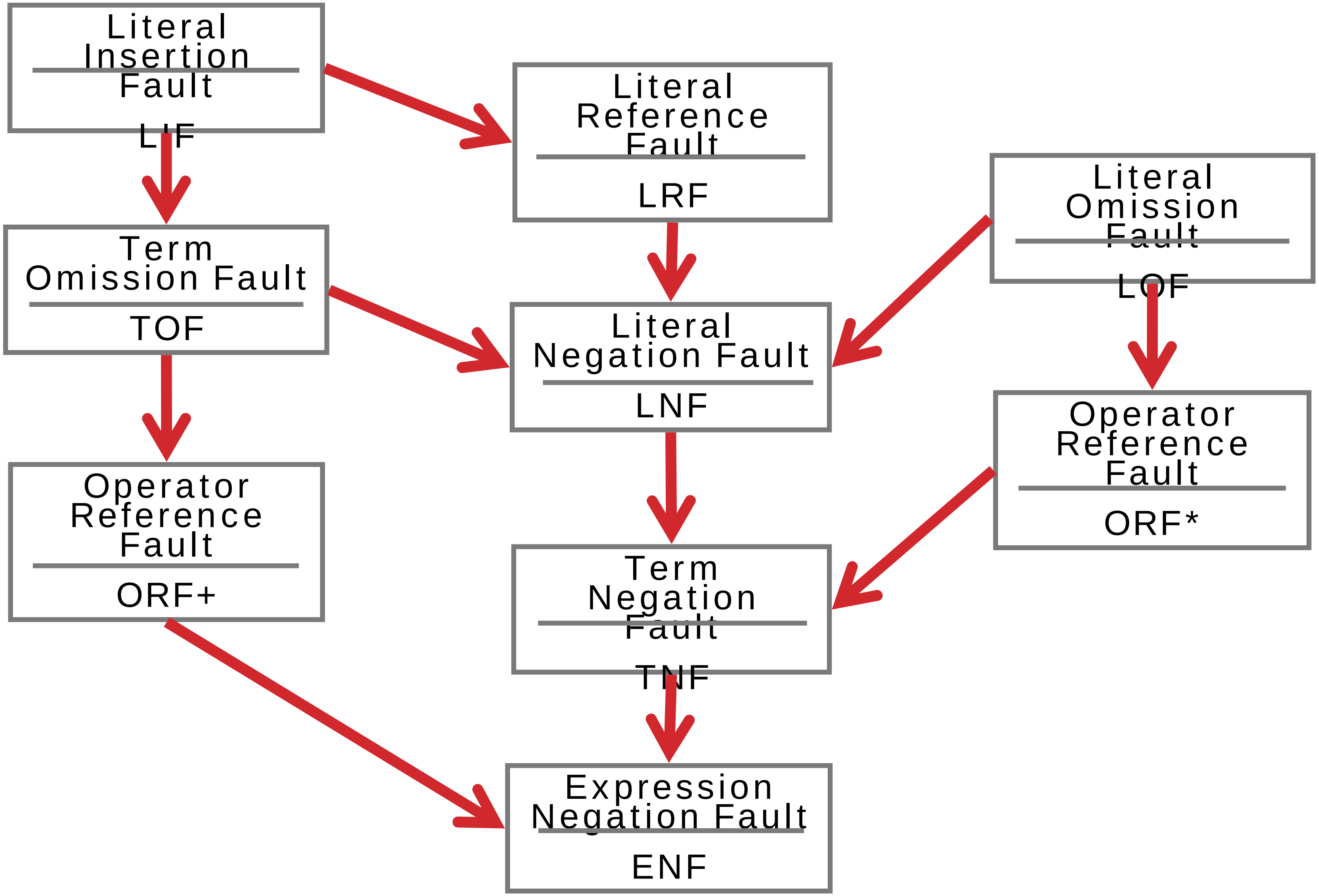
Not all the faults are equal

- Among the fault classes it may exist a **hierarchy**
- A class F1 subsumes another F2 if a test suite that is able to detect all the faults in F1 then it will also detect all the faults in F2.

The hierarchy is useful when generating tests



FAULT DETECTION RELATIONSHIPS


Literal Insertion Fault

LIF

Literal Reference Fault

LRF

Literal Omission Fault

LOF

Term Omission Fault

TOF

Literal Negation Fault

LNF

Operator Reference Fault

ORF*

Operator Reference Fault

ORF+

Term Negation Fault

TNF

Expression Negation Fault

ENF

TESTING CRITERIA

To target these faults, several testing criteria have been (and are continusly) introduced

A testing criteria must define an algorithm to derive the tests

- It analyzes the structure of the expression
- It find the right truth values for the clauses

simplest: implicant Coverage

- Make each implicant evaluate to “true”

OTHER TESTING CRITERIA

MAX-A and MAX-B

- Weyuker, Goradia, and Singh

Multiple Unique True Points (MUTP)

Multiple Near False Points (MNFP)

Corresponding Unique True Point Near False Point (CUTPNFP)

MUMCUT = MUTP + MNFP + CUTPNFP

- Chen, Lau, and Yu

It has been proved that MUMCUT criteria detect all the faults in the hierarchy

- Very efficient (faults/number of tests)
- Several variations to reduce number of tests
- New criteria with different fault detection capability

A NEW WAY TO GENERATE FAULT DETECTING TESTS

BASIC PRINCIPLES

Instead of introducing a new testing criterion

a generation methods that targets explicitly the fault classes

- new fault classes can be added if neeeded
- or removed

trend

Testing and proving become complementary:
tools, methods and techinques generally used for property verification can be effiently employed to solve testing problems

A.

DETECTION CONDITION

Let φ be a Boolean expression

and φ' be one faulty implementation

Definition detection condition $dc = \phi \oplus \phi'$

- where $\oplus$ is the exclusive or

dc **is is true only if φ' evaluates to a different value than the correct predicate φ**

- $\phi: true, \phi': false$
- $\phi: false, \phi': true$

The fault can be discovered only when there exists a test case t in which the condition φ $\oplus$ φ' evaluates to true, i.e., $t \vDash \phi \oplus \phi'$

This dc is also called Boolean *difference* or *derivative*

dc represents our test goal -> *test predicate*

DETECTION CONDITION EXAMPLE

- Literal Omission Fault
- If the Boolean predicate $\phi = ab$
- is implemented as $\phi' = a$
- The detection condition is $dc = ab \oplus a$
    - $\equiv a \wedge \neg b$
    - Is true only if a is true and b is false
- Our test predicate is $a \wedge \neg b$

DETECTING ALL THE FAULTS IN A CLASS

Let ϕ be a predicate and C a fault class.

$F_C(\phi)$ **the set of all the possible faulty implementations of φ according to the fault class** C

- $F_C(\phi)$ can be obtained by applying the mutation operator μ_C that represents the fault class C to ϕ .

The set of test predicates to discover the C **faults in** ϕ **are the expressions** $TP_C(\phi) = \{ \phi \oplus \phi' \mid \forall \phi' \in F_C(\phi)\}$

Example

- Consider the expression $a \wedge b$ and let the fault class C be TOF,
- $F_{TOF} = \{a, b\}$
- $TP = \{ab \oplus a, ab \oplus b\}$

$$\{a \wedge \neg b, b \wedge \neg a\}$$

ADEQUACY OF A TEST SUITE

Let ϕ be a predicate and C a fault class.

Let $TP_C(\phi)$ bet the set of the test predicates for ϕ and C

Definition A test suite T is adequate to test the predicate ϕ with respect to a fault class C if it covers every test predicate in $TP_C(\phi)$. Formally:

$$\forall tp \in TP_C(\phi)\ \exists t \in T\ t \vDash tp$$

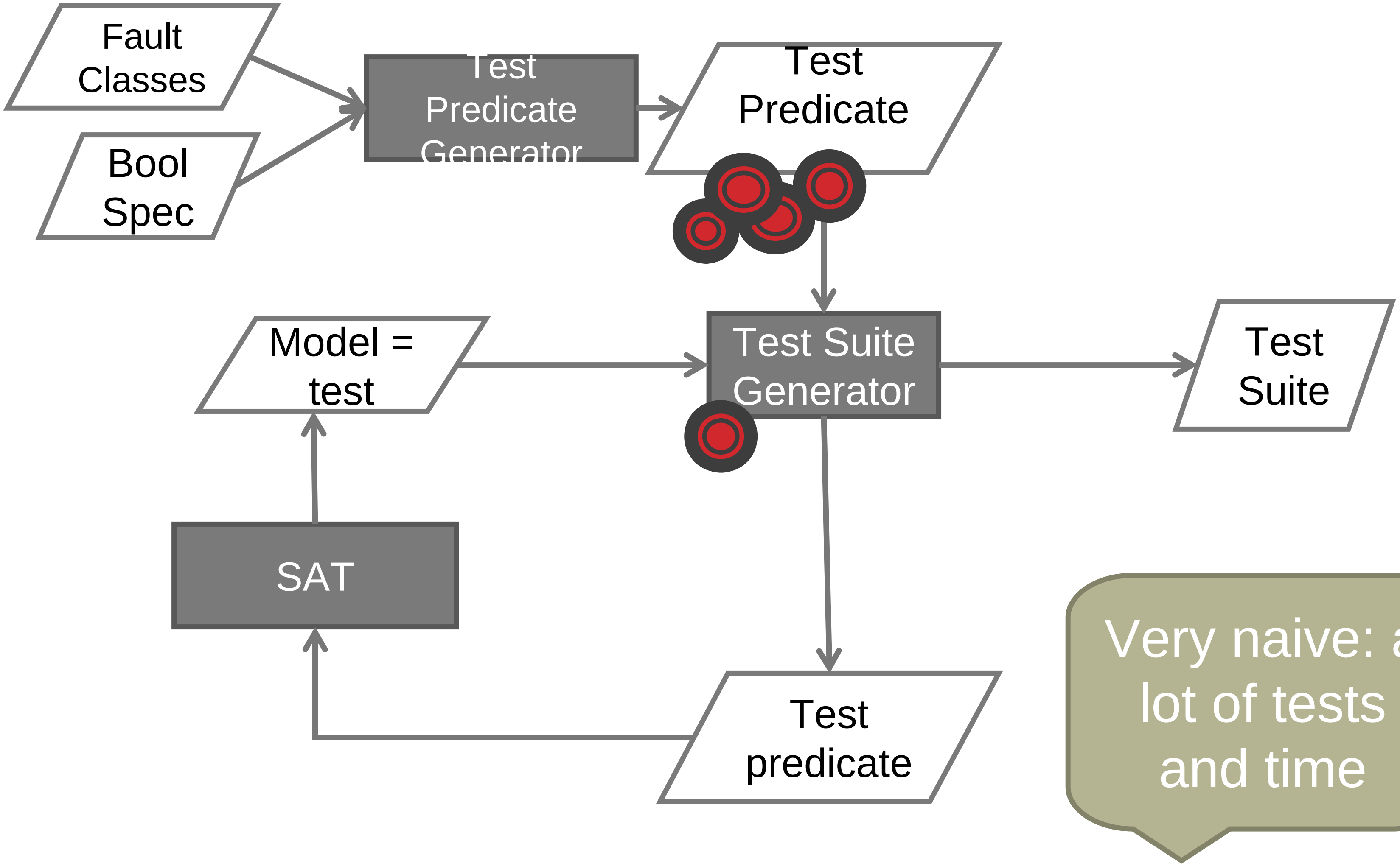
Finding a test suite = finding models for test predicates

- a SAT solver can be used for this

Example:

- To find the TOF in $a \wedge b$ we have to find the models of the two test predicates: $a \wedge \neg b, b \wedge \neg a$
- Easy (a,b): [TF], [FT]

SAT-BASED TEST GENERATION METHOD



Very naive: a lot of tests and time

UNFEASIBLE TEST PREDICATES

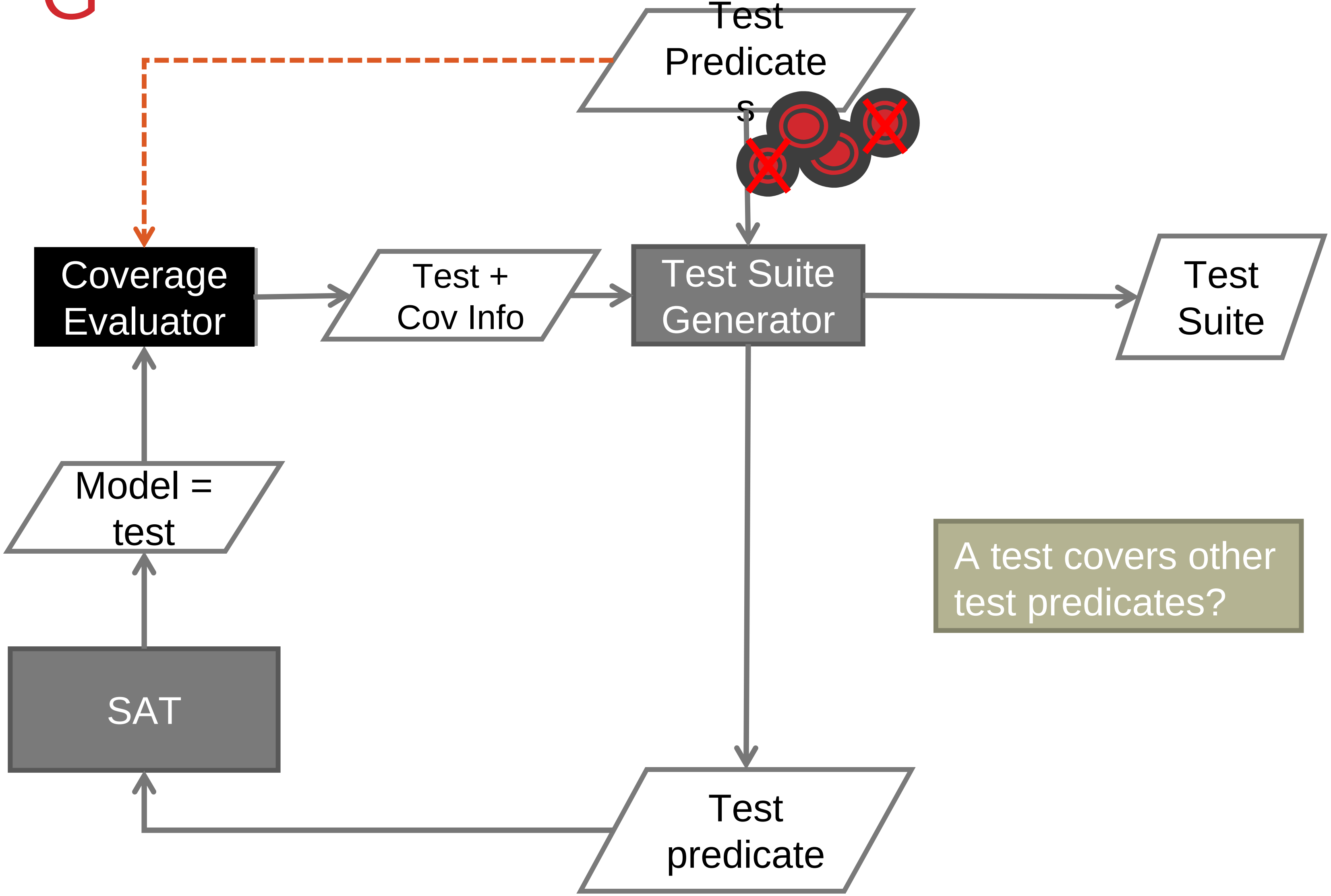
Not all faulty implementations can be distinguished from the original Boolean predicate:

- Some are “equivalent”

Equivalent faults of Boolean predicates can be detected by the SAT solver:

$\phi \oplus \phi'$ is unsatisfiable iff ϕ is equivalent to ϕ'.

MONITORING



Test Predicates

Coverage Evaluator

Test + Cov Info

Test Suite Generator

Test Suite

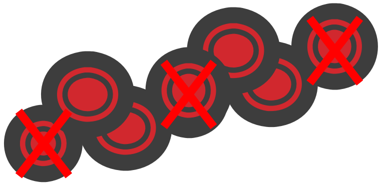
Model = test

SAT

Test predicate

A test covers other test predicates?

MONITORING COVERAGE



A test case generated for one test predicate may satisfy a number of further test predicates.

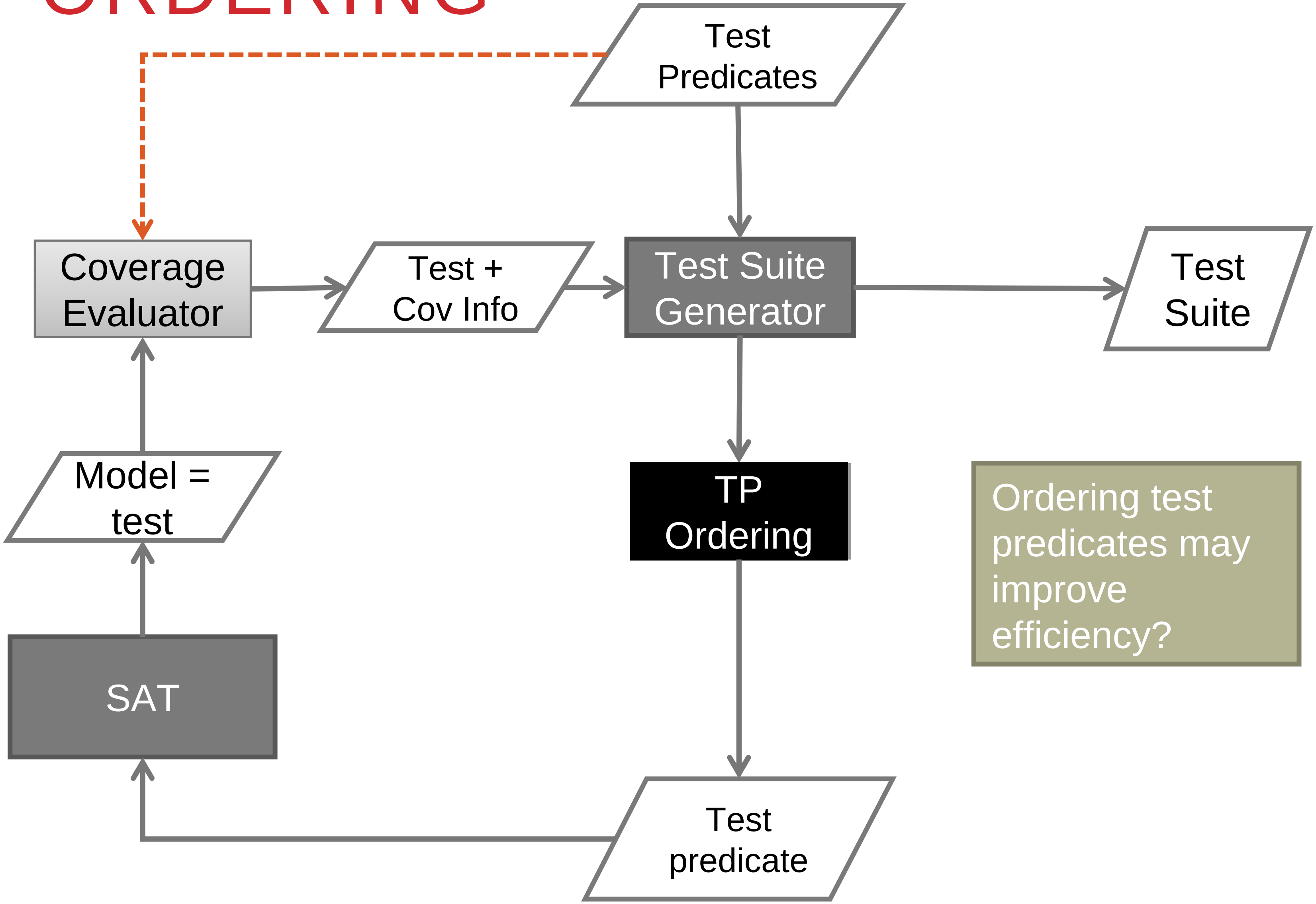
- test predicates can be skipped because they are covered by tests already generated

Checking whether a test predicate tp is covered by a test case t simply requires evaluating the test predicate with the model that t represents, i.e.

$$t \vDash tp$$

it is computationally less expensive then finding a model for tp

ORDERING



Test
Predicates

Coverage
Evaluator

Test +
Cov Info

Test Suite
Generator

Test
Suite

Model =
test

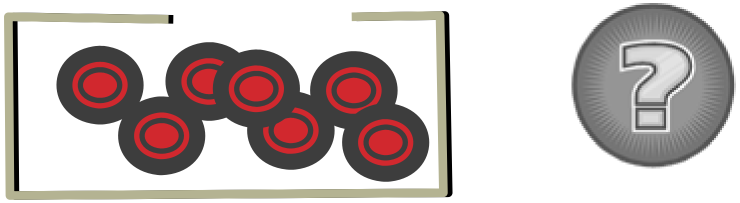
TP
Ordering

Ordering test predicates may improve efficiency?

SAT

Test
predicate

ORDERING TEST PREDICATES



When monitoring is applied the order in which test predicates are selected may impact the size of the resulting test suite.

- Gordon Fraser, Angelo Gargantini, and Franz Wotawa. On the order of test goals in specification-based testing. *Journal of Logic and Algebraic Programming*, 78(6), 472-490, 2009.

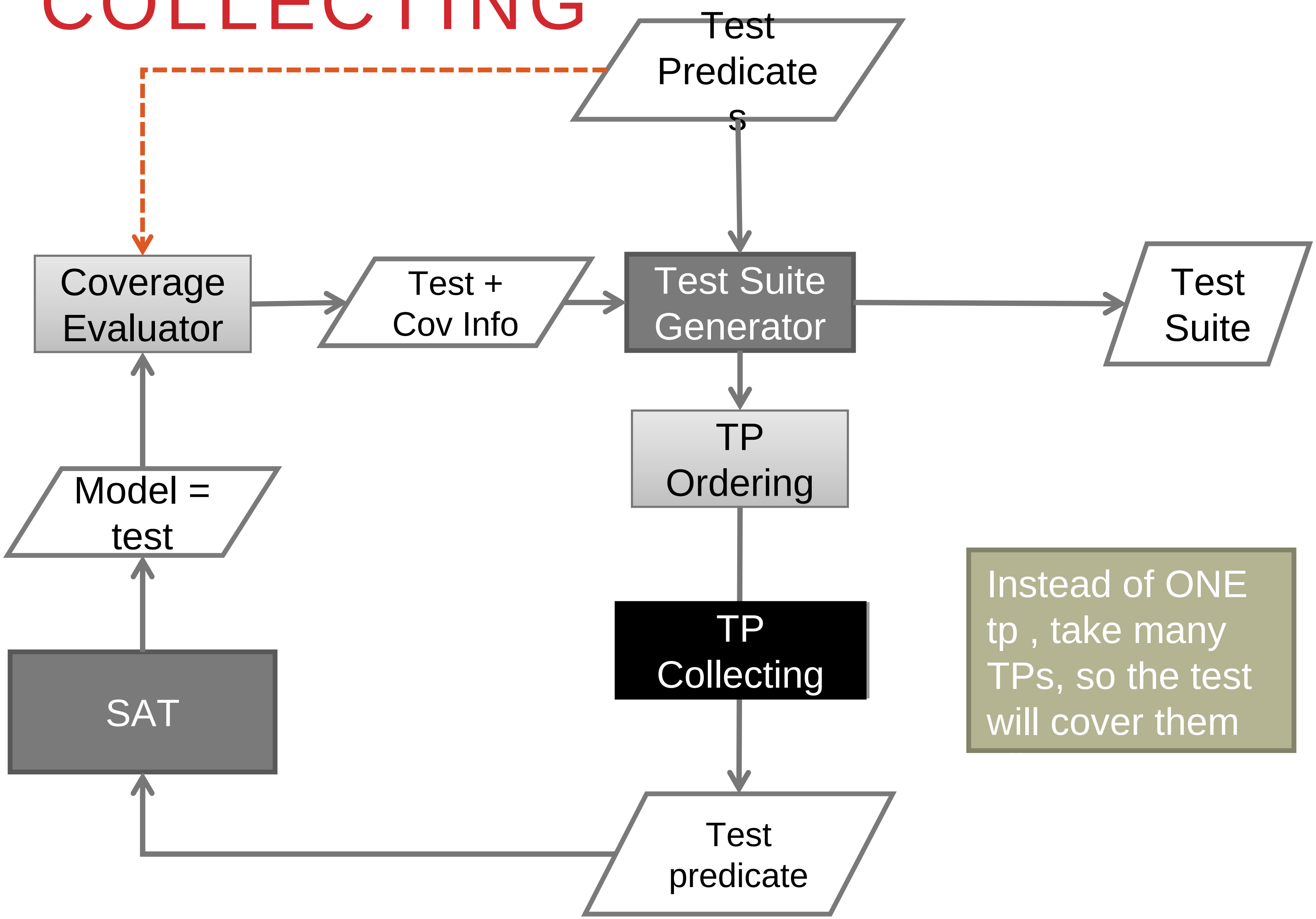
Random order

- randomly take the next tp

Subsuming order

- If the subsuming relation between fault classes is known, or at least a subsumption relationship is suspected to be in place due some empirical data, it can be used to order tps
- Start with the test predicates coming from top classes in the hierarchy
- LIF, LRF, LOF, TOF, LNF, ORF+, ORF*, TNF, and ENF.

COLLECTING



Test Predicates

Coverage Evaluator

Test + Cov Info

Test Suite Generator

Test Suite

TP Ordering

Model = test

TP Collecting

SAT

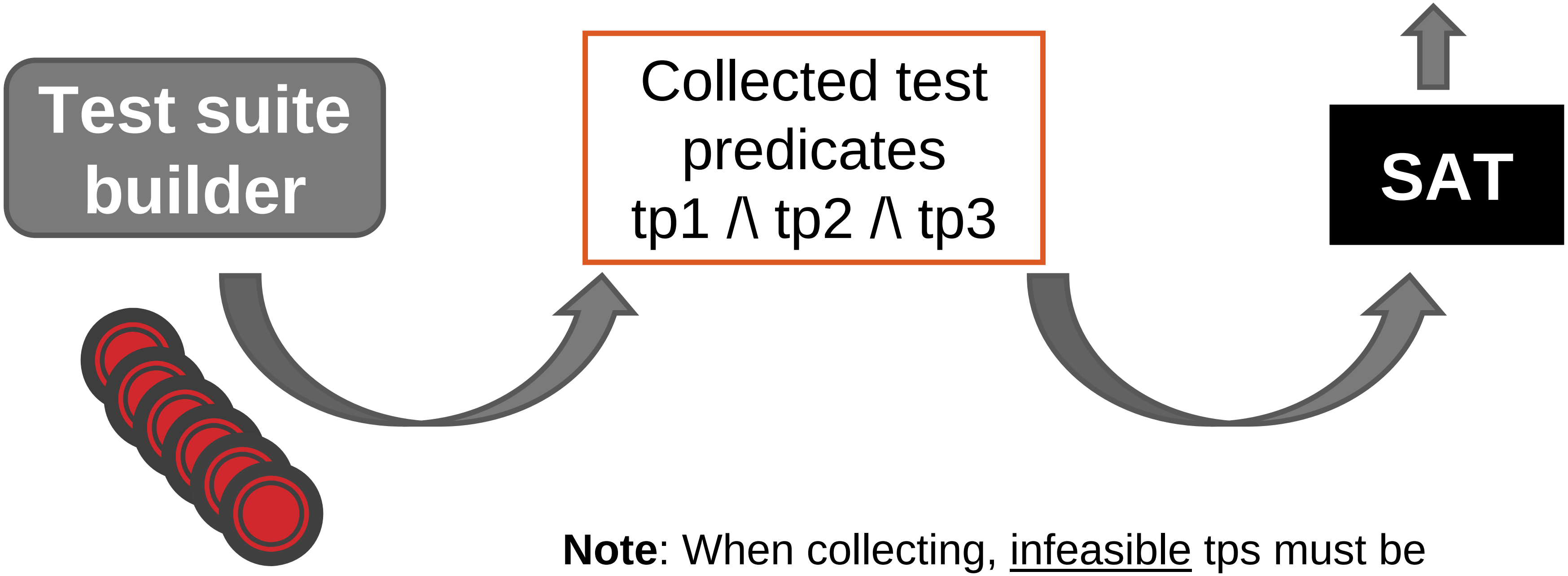
Test predicate

Instead of ONE tp , take many TPs, so the test will cover them

COLLECTING TEST PREDICATES

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

Model = test that covers all the test predicates collected



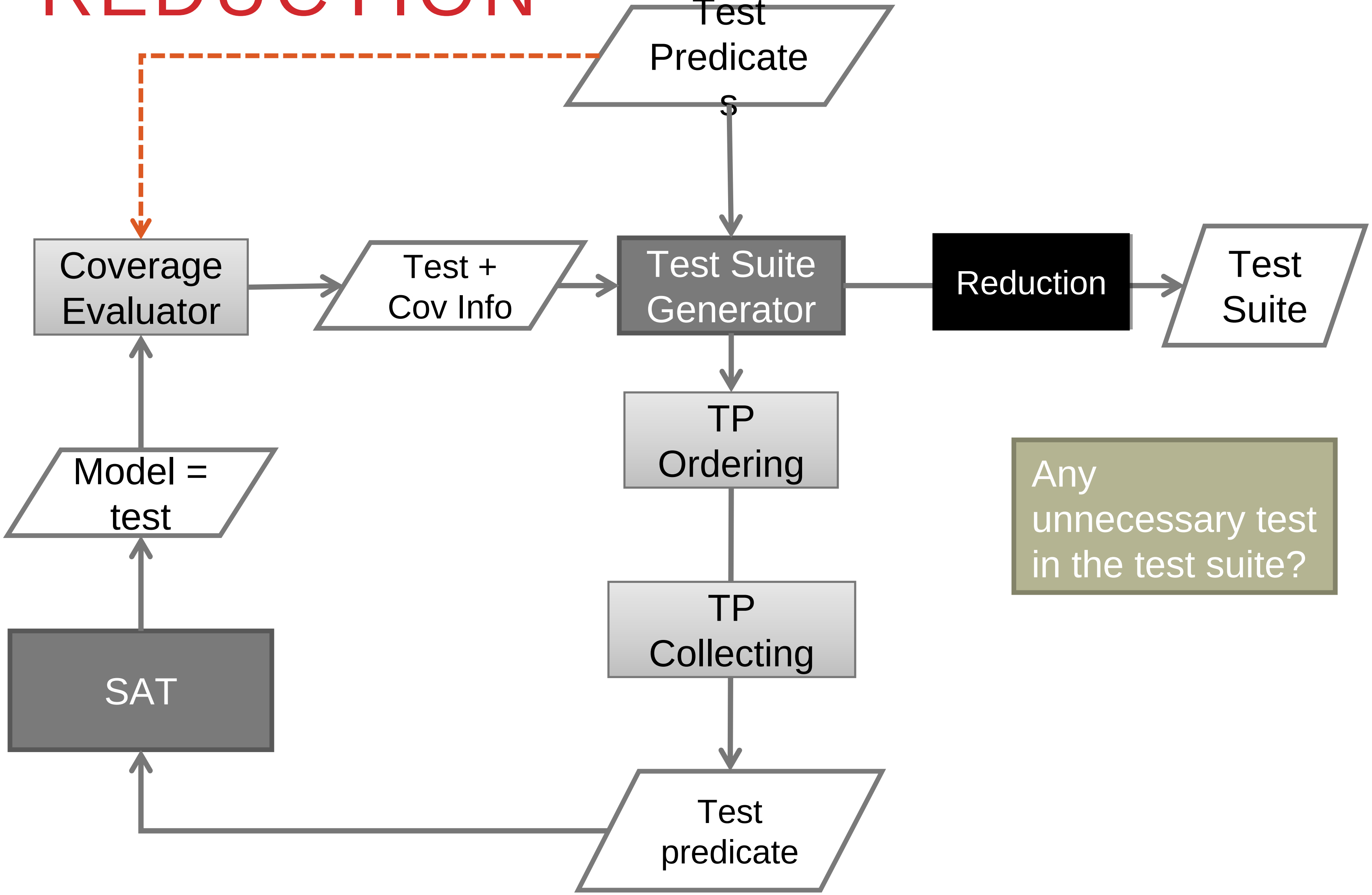
Test suite builder

Collected test predicates
tp1 /\ tp2 /\ tp3

SAT

Note: When collecting, <u>infeasible</u> tps must be ignored, <u>*incompatible*</u> tps must be skipped

REDUCTION


Test Predicate s

Coverage Evaluator

Test + Cov Info

Test Suite Generator

Reduction

Test Suite

TP Ordering

TP Collecting

Model = test

SAT

Test predicate

Any unnecessary test in the test suite?

POST REDUCTION (MINIMIZATION)

A test suite is minimal with regard to an objective if removing any test case from the test suite will lead to the objective no longer being satisfied.

- Some tests may be useless

simple greedy heuristic to the minimum set covering problem for test suite minimization

Note: Monitoring and minimization can behave very differently:

- Minimization requires existing, full test suites
- while monitoring checks test predicates on the fly during test case generation

EXPERIMENTS

EXPERIMENTS

Benchmark: 20 Boolean expressions in a traffic collision avoidance system (TCAS).

- Introduced for MAX-A and MAX-B (Weyuker et al.)
- Used by MUMCUT (Chen, Lau, and Yu)
- And minimal-MUMCUT (Kaminksy and Ammann)

GOAL: reduce the test suite size

COMPARISON AMONG OUR STRATEGIES

Optimization	Reduction of the test suite size		
	Avg	Var	Max
Subsumption order instead of random order	5%	0.4%	19%
Reduction	6%	0.4%	31%
Collection	24%	4%	71%

The smallest test suites are generated with monitoring, ordering by subsumption, collecting, and minimizing.

COLLECTION IS EXPENSIVE

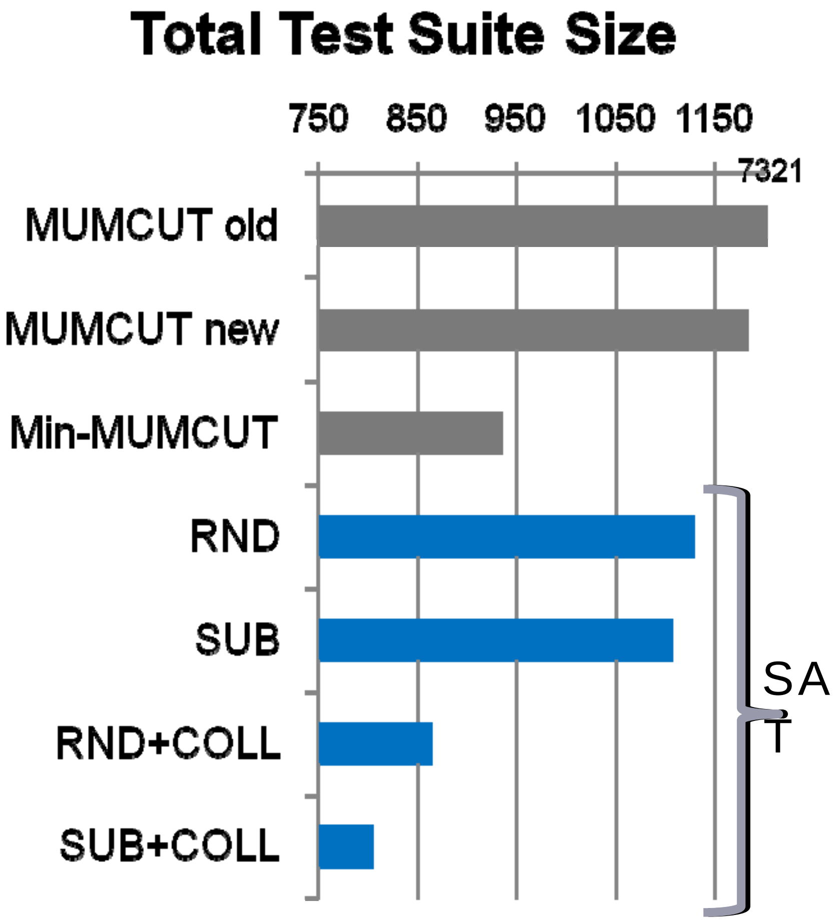
	NO COLL		COLL	
	RND	SUB	RND	SUB
Time (sec)	190.0	44.2	45821.2	18380

328 times the time required by the strategy without collection

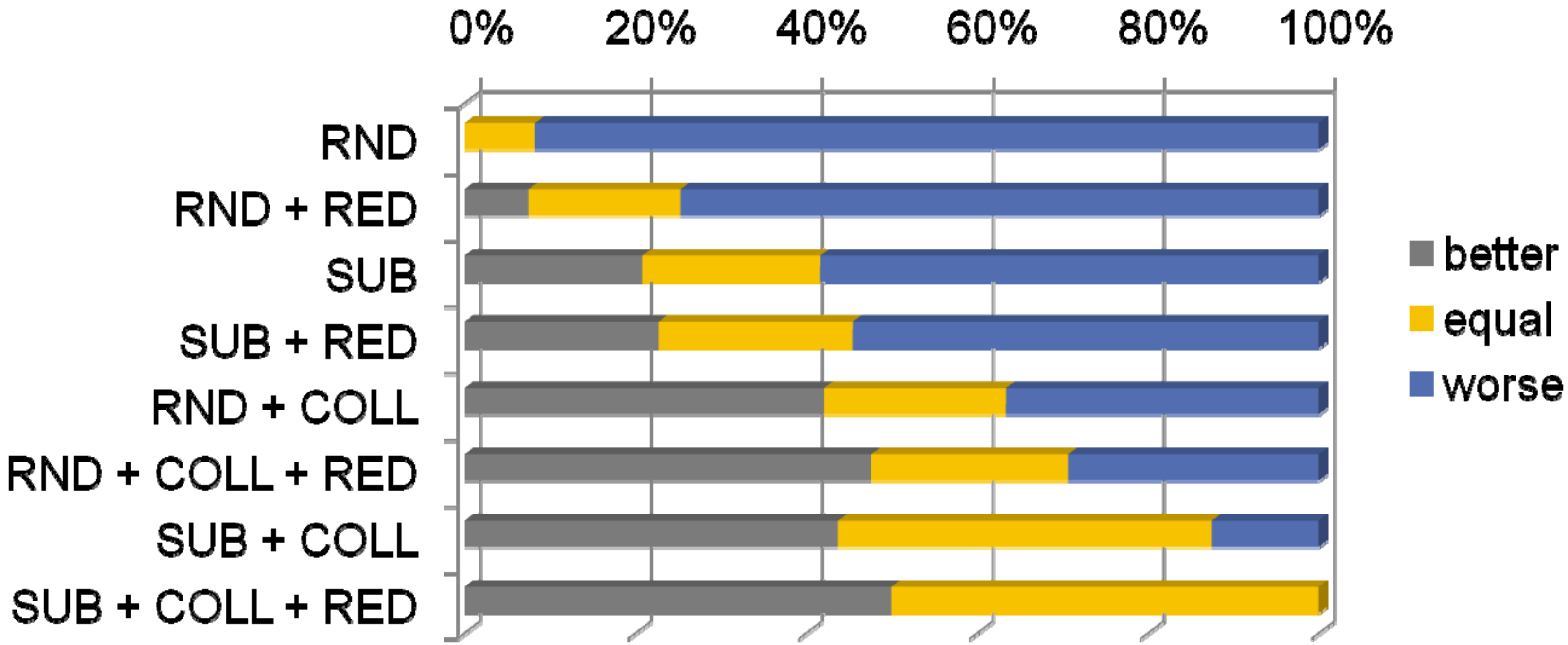
Collecting test predicates is effective at reducing the number of test cases, but computationally expensive.

COMPARISON WITH MUMCUTS

- Much better than the original MUMCUT strategy
- Always better than the new MUMCUT strategy
- Comparable w.r.t Minimal MUMCUT



COMPARISON WITH MIN-MUMCUT



Our method reduces the number of test cases necessary to cover all faults of these classes in comparison to MinimalMUMCUT.

CONCLUSIONS

It is possible to generate tests explicitly targeting faults

- SAT solvers can be employed

Several optimizations can be applied

- Monitoring, ordering, collecting, minimization

In comparison to *MUMCUT, it reduces the number of test cases necessary to cover all faults of these classes

Future work:

- Not only DNF
- Improve efficiency: reducing the number of runs of the SAT