An Algorithm for Generating t-wise Covering Arrays from Large Feature Models

Martin Fagereng Johansen

Øystein Haugen

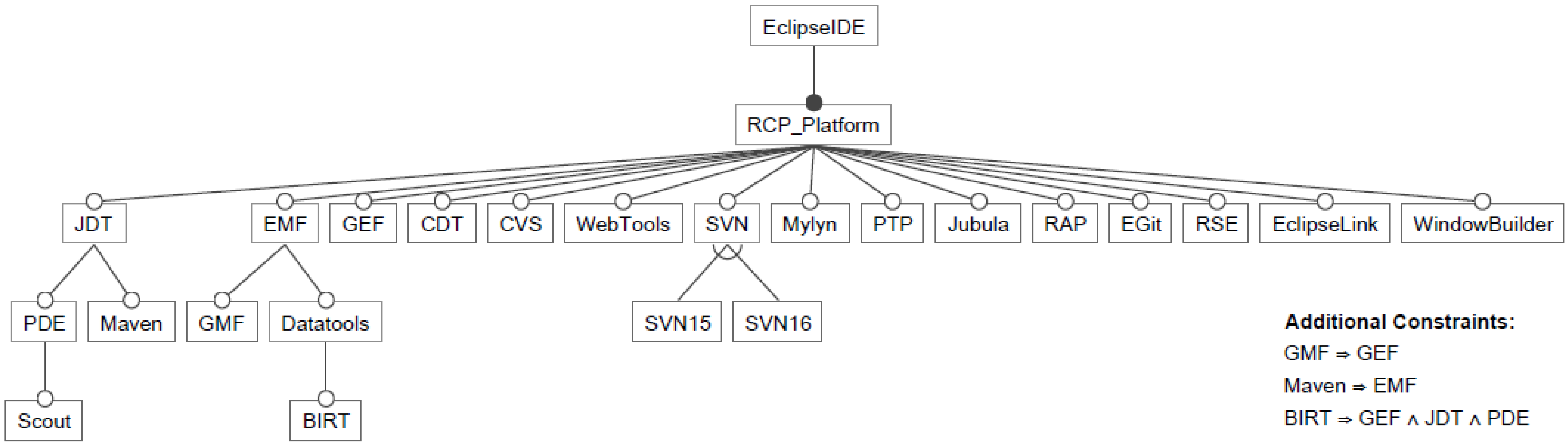
Franck Fleurey

SPLC 2012 – Salvador, Brazil

Example Product Line: The Eclipse IDEs

	Java	Java EE	C/C++	C/C++ Linux	RCP/RAP	Modeling	Reporting	Parallel	Scout	Testers	Javascript	Classic
RCP/Platform	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CVS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
EGit			✓	✓	✓	✓						
EMF	✓	✓				✓	✓					
GEF	✓	✓				✓	✓					
JDT	✓	✓			✓	✓	✓		✓			✓
Mylyn	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Web Tools		✓					✓				✓	
Linux Tools			✓	✓				✓				
Java EE Tools		✓					✓					
XML Tools	✓	✓			✓		✓	✓				
RSE		✓	✓	✓			✓	✓				
EclipseLink		✓					✓			✓		
PDE		✓			✓	✓	✓		✓			✓
Datatools		✓					✓					
CDT			✓	✓				✓				
BIRT							✓					
GMF						✓						
PTP								✓				
MDT						✓						
Scout									✓			
Jubula										✓		
RAP					✓							
WindowBuilder	✓											
Maven	✓											

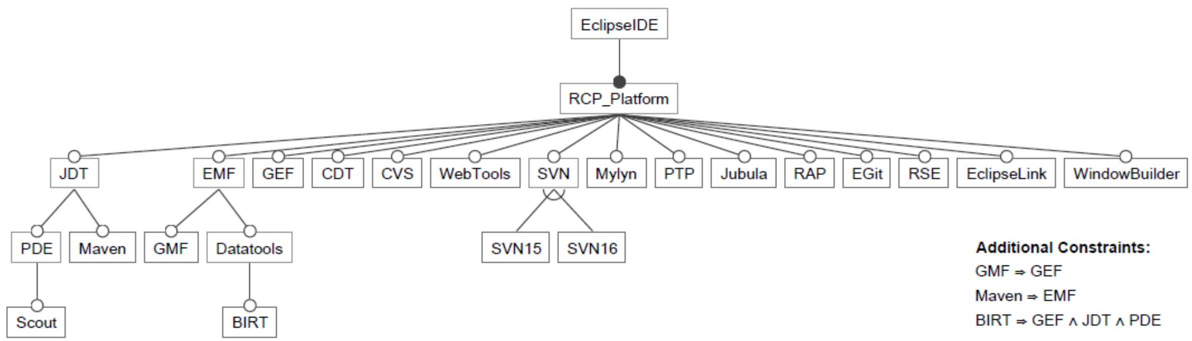
Constraints Between Features



356,352 possible products

Product Line Verification

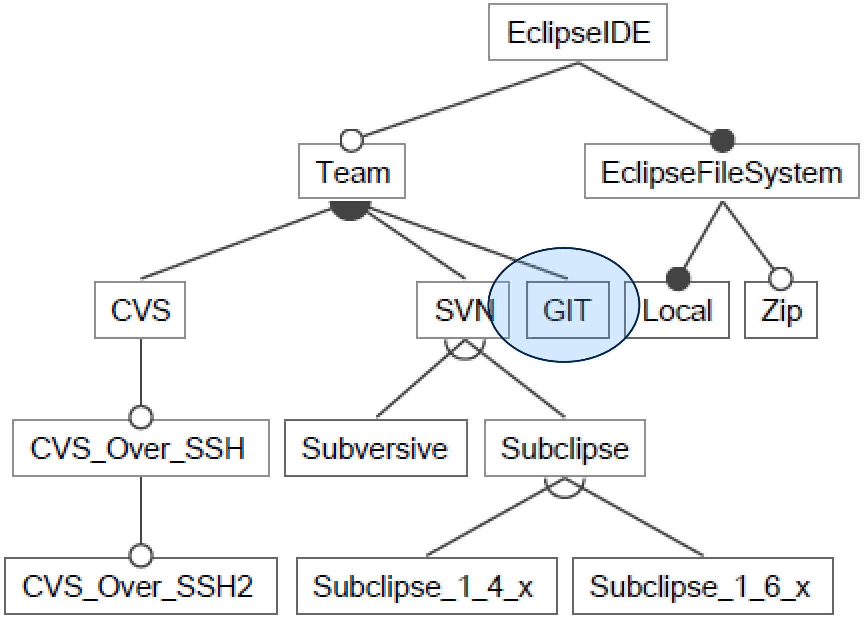
- How do we gain confidence that <u>any valid product</u> works?



	Java	Java EE	C/C++	C/C++ Linux	RCP/RAP	Modeling	Reporting	Parallel	Scout	Testers	Javascript	Classic
RCP/Platform	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CVS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
EGit			✓	✓	✓	✓						
EMF	✓	✓				✓	✓					
GEF	✓	✓				✓	✓					
JDT	✓	✓			✓	✓	✓		✓			✓
Mylyn	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Web Tools		✓					✓				✓	
Linux Tools			✓	✓				✓				
Java EE Tools		✓					✓					
XML Tools	✓	✓			✓		✓	✓				
RSE		✓	✓	✓			✓	✓				
EclipseLink		✓					✓			✓		
PDE		✓			✓	✓	✓		✓			✓
Datatools		✓					✓					
CDT			✓	✓				✓				
BIRT							✓					
GMF						✓						
PTP								✓				
MDT						✓						
Scout									✓			
Jubula										✓		
RAP					✓							
WindowBuilder	✓											
Maven	✓											

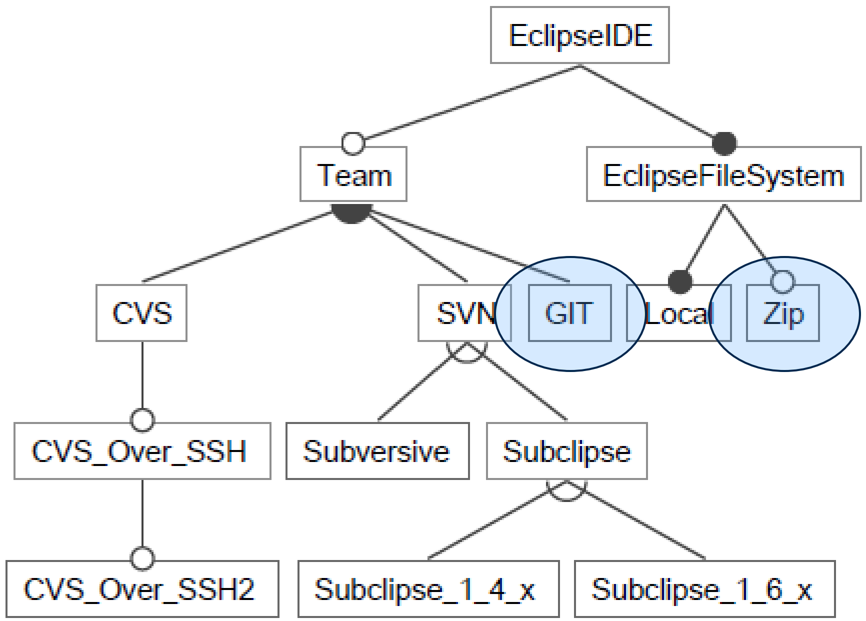
Faulty Features

- Unit tests may find faults inside a single feature.
  - n test suites required for a product line with n features.
- What about faulty cooperation between features?
  - What if they *interact* incorrectly?



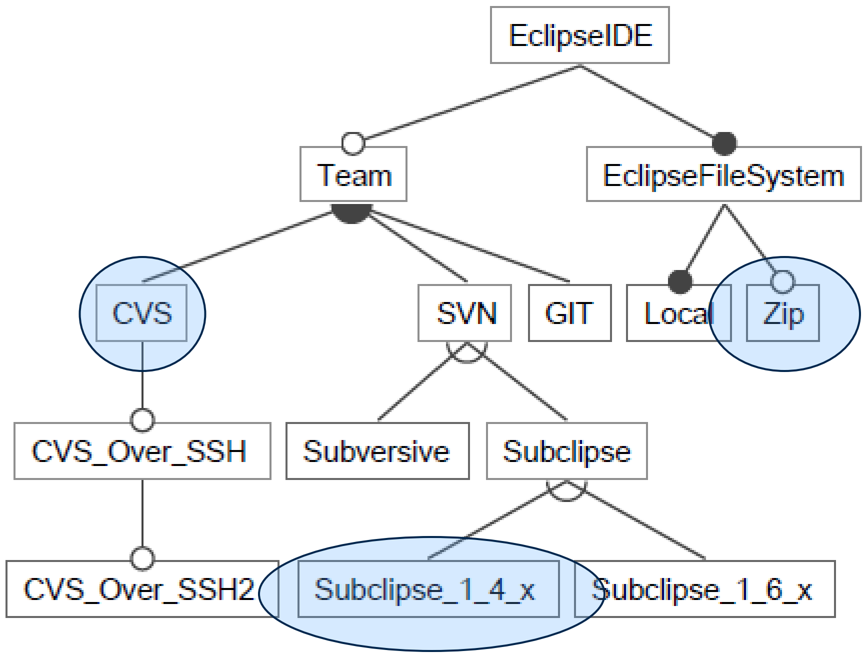
Interaction Faults

- 2-wise interaction fault
  - reproducible by including 2 specific features
  - the others do not matter



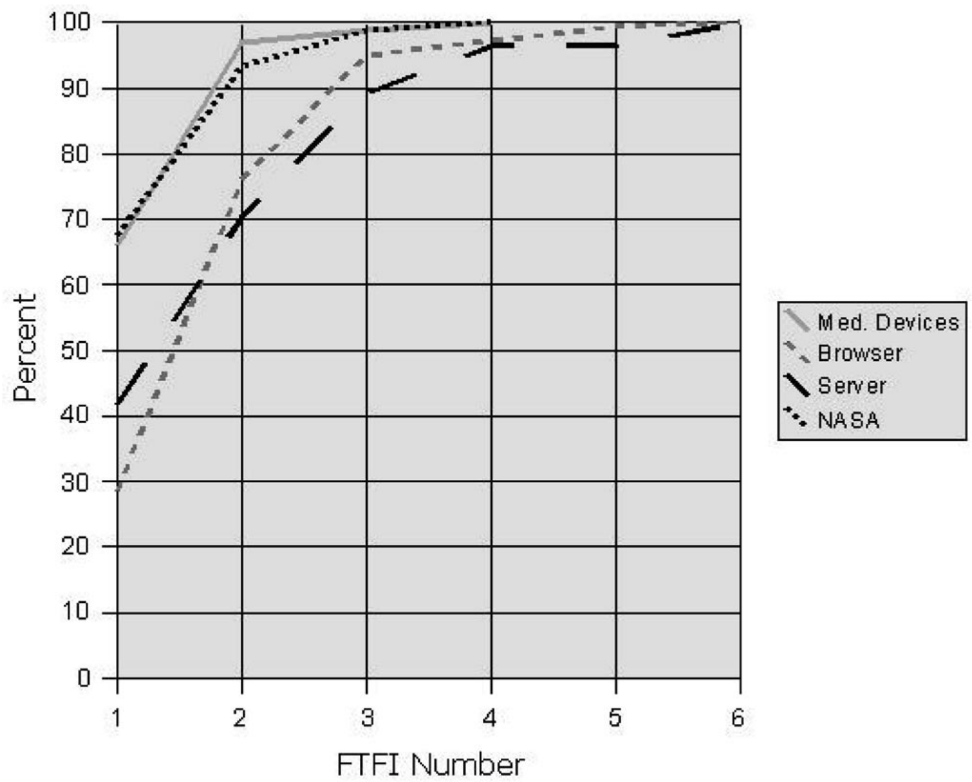
Interaction Faults

- 3-wise interaction fault
  - reproducible by including 3 specific features
  - the others do not matter



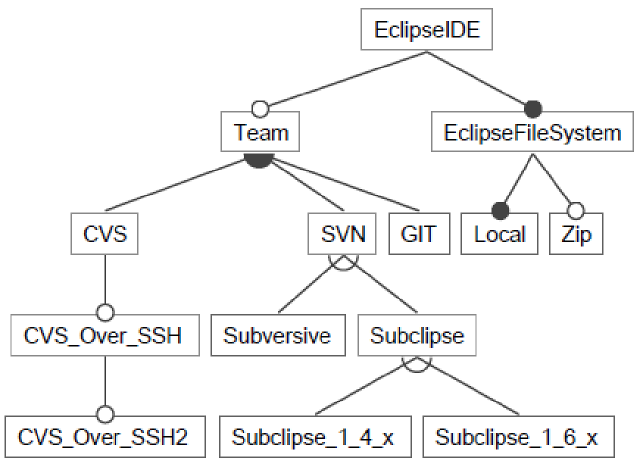
Empirics Show:

- Kuhn et al. 2004:
  - Most bugs can be attributed to the interaction of a few features.



Covering Arrays

- Mathematical property:
  - Only a few products needed to cover all simple interactions



Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12
EclipseIDE	X	X	X	X	X	X	X	X	X	X	X	X
Team	X	X	-	X	X	X	X	X	X	X	-	X
CVS	X	X	-	X	X	-	X	-	X	-	-	-
CVS_Over_SSH	X	-	-	X	X	-	X	-	X	-	-	-
CVS_Over_SSH2	-	-	-	X	X	-	X	-	X	-	-	-
SVN	X	X	-	X	X	X	-	X	X	X	-	-
Subversive	X	-	-	X	-	-	-	X	-	-	-	-
Subclipse	-	X	-	-	X	X	-	-	X	X	-	-
Subclipse_1_4_x	-	X	-	-	X	-	-	-	-	X	-	-
Subclipse_1_6_x	-	-	-	-	-	X	-	-	X	-	-	-
GIT	X	-	-	-	X	X	-	X	-	-	-	X
EclipseFileSystem	X	X	X	X	X	X	X	X	X	X	X	X
Local	X	X	X	X	X	X	X	X	X	X	X	X
Zip	-	X	X	X	-	-	-	X	X	-	-	-

- Other examples (pair-wise testing):
  - For the "e-shop product line" with 287 features: 21 products
  - For the Linux kernel with almost 7,000 features: 480 products

Combinatorial Interaction Testing (CIT)

1. Generate a covering array ← **scalability issue**
   - Can be reused until the feature model is changed
2. Build each product
3. Apply a single system testing technique to each product

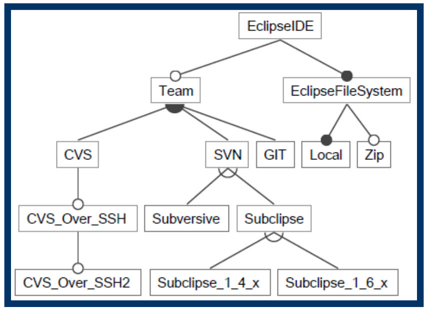
- Note: CIT was originally intended for single system testing
  - Covering arrays over input instead of interactions.

Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12
EclipseIDE	X	X	X	X	X	X	X	X	X	X	X	X
Team	X	X	-	X	X	X	X	X	X	X	-	X
CVS	X	X	-	X	X	-	X	-	X	-	-	-
CVS_Over_SSH	X	-	-	X	X	-	X	-	X	-	-	-
CVS_Over_SSH2	-	-	-	X	X	-	X	-	X	-	-	-
SVN	X	X	-	X	X	X	-	X	X	X	-	-
Subversive	X	-	-	X	-	-	-	X	-	-	-	-
Subclipse	-	X	-	-	X	X	-	-	X	X	-	-
Subclipse_1_4_x	-	X	-	-	X	-	-	-	-	X	-	-
Subclipse_1_6_x	-	-	-	-	-	X	-	-	X	-	-	-
GIT	X	-	-	-	X	X	-	X	-	-	-	X
EclipseFileSystem	X	X	X	X	X	X	X	X	X	X	X	X
Local	X	X	X	X	X	X	X	X	X	X	X	X
Zip	-	X	X	X	-	-	-	X	X	-	-	-

Background

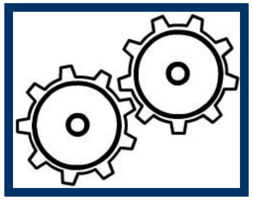
- Our MODELS 2011 paper concludes:
  - Covering array generation is tractable in practice.
    - Difficult to satisfy FMs imply no products to sell, which is absurd.
  - An efficient algorithm was not provided.
    - 2-wise testing limit: about 500 features
    - 3-wise testing limit: about 200 features
- An efficient algorithm is contributed in this paper.
  - 2-wise testing
    - Now works for the Linux Kernel feature model (6888 features)
  - 3-wise testing
    - Now works for the eCos feature model (1244 features)
    - (An optimized C/C++ implementation + some good hardware should work even for the Linux Kernel feature model.)

Overview of the Algorithm



t

ICPL

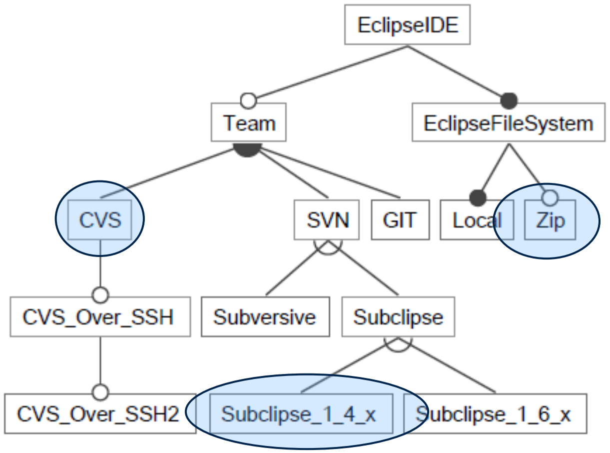


Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12
EclipseIDE	X	X	X	X	X	X	X	X	X	X	X	X
Team	X	X	-	X	X	X	X	X	X	X	-	X
CVS	X	X	-	X	X	-	X	-	X	-	-	-
CVS_Over_SSH	X	-	-	X	X	-	X	-	X	-	-	-
CVS_Over_SSH2	-	-	-	X	X	-	X	-	X	-	-	-
SVN	X	X	-	X	X	X	-	X	X	X	-	-
Subversive	X	-	-	X	-	-	-	X	-	-	-	-
Subclipse	-	X	-	-	X	X	-	-	X	X	-	-
Subclipse_1_4_x	-	X	-	-	X	-	-	-	-	X	-	-
Subclipse_1_6_x	-	-	-	-	-	X	-	-	X	-	-	-
GIT	X	-	-	-	X	X	-	X	-	-	-	X
EclipseFileSystem	X	X	X	X	X	X	X	X	X	X	X	X
Local	X	X	X	X	X	X	X	X	X	X	X	X
Zip	-	X	X	X	-	-	-	X	X	-	-	-

- Implementation Supports
  - Simple XML Feature Models (SXFM)
  - GUI DSL
  - DIMACS
  - CVL (Proposed OMG standard)

- **CSV-file**

Groundwork



- Data Structures
  - a = (feature, included) – an assignment
  - e = {a_1, a_2, ..., a_t} – a t-set – a set of t assignments
  - T_t – the set of all t-sets
  - I_t – the set of all invalid t-sets
  - U_t – the set of all valid t-sets (the "universe")
  - C – configuration – a set of assignments, one for each feature
  - CA_t – {C_1, C_2, …, C_x} – a Covering Array of strength t
- Equations
  - $|T_t| = 2^t\binom{f}{t}$, i.e. 95 million pair-wise interactions for the Linux kernel
    - Empirically, $|I_t| \ll |U_t|$
  - $CA_{t-1} \subseteq CA_t$, thus, generating CA_{t-1} before CA_t is an option.

- Recursive generation of covering arrays of lesser strength
- Greedy Loop
  - Fit as many as possible
  - Remove those covered
  - … and repeat until all inter-actions are covered
- Not parallel

Algorithm 2 $ICPL(FM, t) : (C_t, I_t)$

if $t = 1$ **then**
  $(C_t, I_t) \leftarrow genCompleteI_1(FM)$
  $T_t \leftarrow FM.genT(1) \setminus I_t$
  $invalidRemoved \leftarrow true$
else
  $(C_t, I_{t-1}) \leftarrow ICPL(FM, t-1)$
  $(T_t, I_t) \leftarrow generateTSets(FM.getT(t), I_{t-1}, C_t)$
  $invalidRemoved \leftarrow false$
end if
while $T_t \neq \emptyset$ **do**
  $C \leftarrow genConfiguration(FM, T_t)$
  $C_t \leftarrow C_t \cup \{C\}$
  $CO \leftarrow getCovered(FM, C, T_t)$
  $T_t \leftarrow T_t \setminus CO$
  if $(\neg invalidRemoved) \wedge (\lfloor log_{10}|CO| \rfloor \leq \lfloor log_{10}|FM| \rfloor)$ **then**
    $(T_t, TempI) \leftarrow genInvalid(FM, T_t)$
    $I_t \leftarrow I_t \cup TempI$
    $invalidRemoved \leftarrow true$
  end if
end while
return (C_t, I_t)

- Pick an interaction
  - Skip if covered
- Does it fit the product?
  - yes
  - no
- Make sure all features are assigned
- Not parallel

Algorithm 6 *genConfiguration*$(FM, T) : C$

```
(C, CF, PF) ← (∅, ∅, ∅)
for each t-set e in T do
    for each assignment a in e do
        PF ← PF ∪ {a.f}
    end for
end for
Loop:
for each t-set e in T do
    F ← ∅
    for each assignment a in e do
        F ← F ∪ {a.f}
    end for
    if F ⊆ CF then
        continue
    end if
    if FM.is_satisfiable(C ∪ e) then
        C ← C ∪ e
        CF ← CF ∪ F
    else
        for each assignment a in e do
            if ((e \ {a}) ⊆ C) ∧ (a ∉ C) then
                b ← a with assignment inverted
                C ← C ∪ {b}
                CF ← CF ∪ {b.f}
            end if
        end for
    end if
    if |CF| = |PF| then
        break Loop
    end if
end for
return FM.satisfy(C)
```

Quick!

- Algorithm 3
  - Is the assignment valid?
- Algorithm 7
  - For all uncovered interactions
  - Is the interaction covered?
- Algorithm 8
  - Pick an interaction
  - Check if it is invalid
- These are data-parallel sub-algorithms

Algorithm 3 $getInvalidAssignments(FM, T_1) : I_1$

$I_1 \leftarrow \emptyset$
for each t-set e in T_1 **do**
  if $\neg FM.is\_satisfiable(e)$ **then**
    $I_1 \leftarrow I_1 \cup \{e\}$
  end if
end for
return I_1

Algorithm 7 $genCovered(FM, C, T) : CO$

$CO \leftarrow \emptyset$
for each t-set e in T **do**
  if $e \subseteq C$ **then**
    $CO \leftarrow CO \cup \{e\}$
  end if
end for
return CO

Algorithm 8 $genInvalid(FM, T) : (V, I)$

$(I, V) \leftarrow (\emptyset, \emptyset)$
while $T \neq \emptyset$ **do**
  $e \leftarrow$ any t-set in T
  if $\neg FM.is\_satisfiable(e)$ **then**
    $I \leftarrow I \cup \{e\}$
    $T \leftarrow T \setminus \{e\}$
  else
    $V \leftarrow V \cup \{e\}$
    $T \leftarrow T \setminus \{e\}$

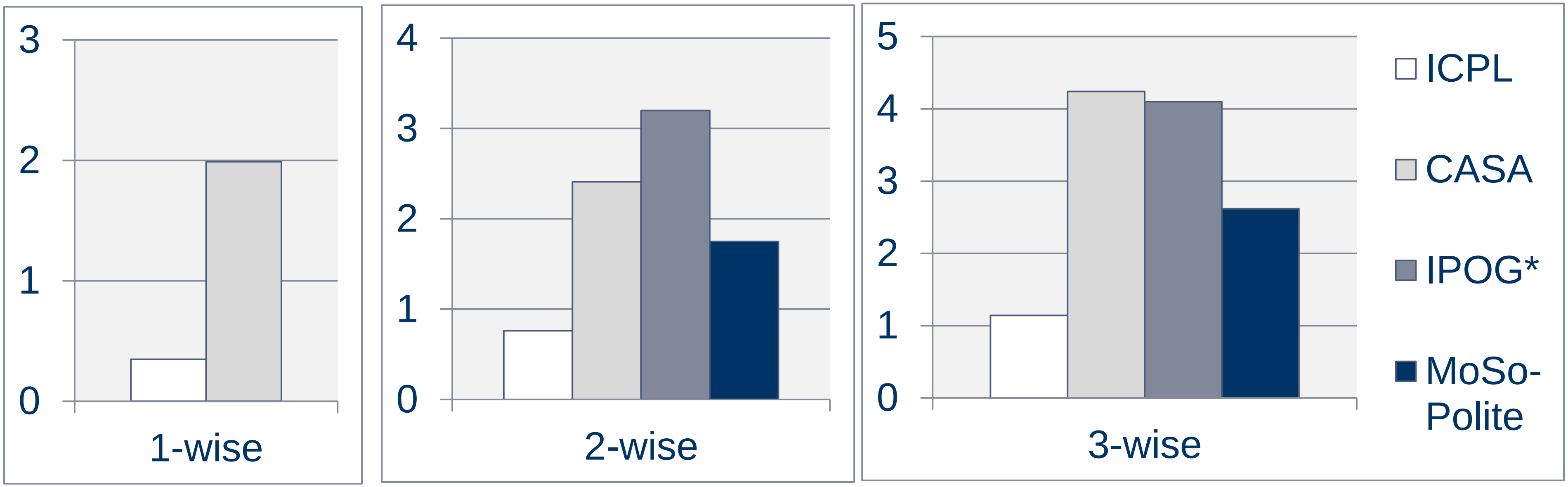
Compared to Other Tools

- ICPL – our new algorithm
- CASA – Simulated annealing algorithm by Garvin et al.
- MoSo-Polite – algorithm by Oster et al.
- IPOG – algorithm by Lei et al.

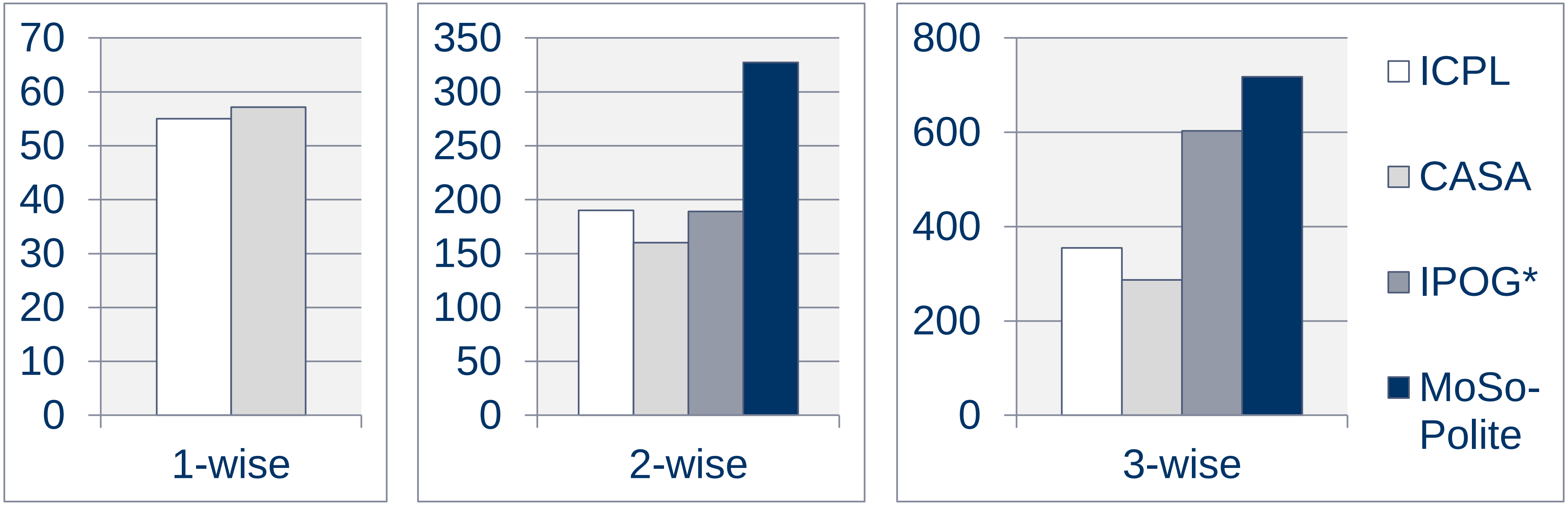
- Experiment Machine
  - Could execute 6 threads in parallel
  - 32 GiB RAM

Time Taken to Generate

- Statistic estimates
  - The c in $O(f^c)$ where f is the number of features



Size of Covering Arrays



Large Feature Models

Feature Model\keys	Features	Constraints (CNF clauses)	SAT time (ms)	1-wise size	1-wise time (s)	2-wise size	2-wise time (s)	3-wise size	3-wise time (s)
2.6.28.6-icse11.dimacs	6,888	187,193	125	25	89	480	33,702	n/a	n/a
freebsd-icse11.dimacs	1,396	17,352	18	9	10	77	240	*78	*2,540
ecos-icse11.dimacs	1,244	2,768	12	6	2	63	185	*64	*973
Eshop-fm.xml	287	22	5	3	0.16	21	5	108	457

...

Summary

- Our contribution
  - A scalable algorithm for t-wise (1-3) covering array generation
  - An empirical evaluation and comparison
- Implementation available
  - The implementation is available as open source (EPL)
  - Experiments are reproducible
- All the data is available
  - All 28,500 measurements available for the paper's resource website + charts and summaries