Why Should We Care about Similarity of Satisfiability Problems?

Part I

Part II

Part III

Part IV

Part I - Why Does Configurable Software Need Compositionality?

- **Highly-Configurable Systems**
- **Feature Models**
- **A Matter of Size**
- **Why to Solve Lots of Similar SAT Problems?**

Part II - Compositional Analyses with Feature-Model Interfaces

- **Compositionality Principle**
- **Feature-Model Interfaces**
- **Reduced Feature-Model Size**
- **Reduced Effort in Type Checking Linux?**

Part III - Solving Similar SAT/BDD/SMT Instances

- **Profiling the SAT4J Implementation**
- **Effect of Our SAT4J Optimization**
- **The Influence of Large Submodels**
- **Cumulative Solver Times with Threshold**
- **Solving Similar SAT/BDD/SMT Instances**
- **Conclusion**

Part IV - Backup Slides

- **Decomposition on Demand?**
- **Make Implicit Constraints Explicit**
- **Decomposition with Feature-Model Interfaces**
- **Compositionality with Feature-Model Interfaces**
- **Reasoning Strategies**
- **Setup for Measurements**
- **Time to Solve SAT Queries**
- **Time vs. Number of Selected Submodels**
- **Threats to Validity**

Technische Universität Braunschweig

ISF

Why Should We Care about Similarity of Satisfiability Problems?

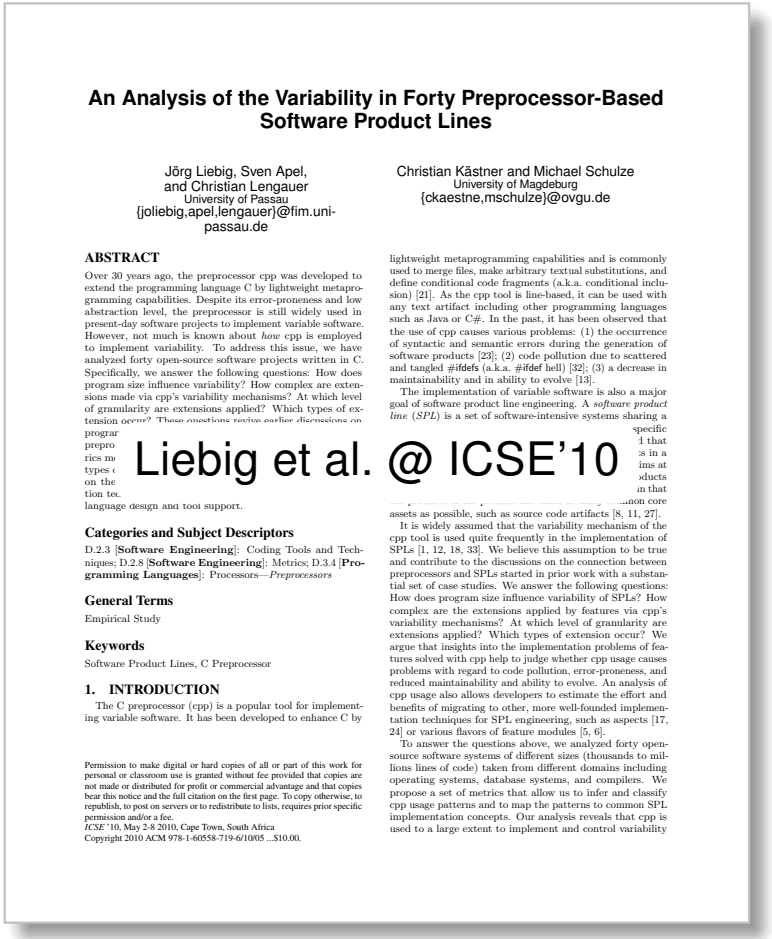
Thomas Thüm, Frederik Kanning, Stephan Mennicke, Ina Schaefer, ...

FOSD Meeting 2017 in Grasellenbach, March 16, 2017

Part I

Why Does Configurable Software Need Compositionality?

Highly-Configurable Systems



An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines

Jörg Liebig, Sven Apel, and Christian Lengauer
University of Passau
{joliebig,apel,lengauer}@fim.uni-passau.de

Christian Kästner and Michael Schulze
University of Magdeburg
{ckaestne,mschulze}@ovgu.de

ABSTRACT

Over 30 years ago, the preprocessor cpp was developed to extend the programming language C by lightweight metaprogramming capabilities. Despite its error-proneness and low abstraction level, the preprocessor is still widely used in present-day software projects to implement variable software. However, not much is known about *how* cpp is employed to implement variability. To address this issue, we have analyzed forty open-source software projects written in C. Specifically, we answer the following questions: How does program size influence variability? How complex are extensions made via cpp's variability mechanisms? At which level of granularity are extensions applied? Which types of extension occur? These questions revive earlier discussions on

Liebig et al. @ ICSE'10

language design and tool support.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.8 [**Software Engineering**]: Metrics; D.3.4 [**Programming Languages**]: Processors—*Preprocessors*

General Terms

Empirical Study

Keywords

Software Product Lines, C Preprocessor

1. INTRODUCTION

The C preprocessor (cpp) is a popular tool for implementing variable software. It has been developed to enhance C by lightweight metaprogramming capabilities and is commonly used to merge files, make arbitrary textual substitutions, and define conditional code fragments (a.k.a. conditional inclusion) [21]. As the cpp tool is line-based, it can be used with any text artifact including other programming languages such as Java or C#. In the past, it has been observed that the use of cpp causes various problems: (1) the occurrence of syntactic and semantic errors during the generation of software products [23]; (2) code pollution due to scattered and tangled `#ifdefs` (a.k.a. `#ifdef` hell) [22]; (3) a decrease in maintainability and in ability to evolve [13].

The implementation of variable software is also a major goal of software product line engineering. A *software product line* (*SPL*) is a set of software-intensive systems sharing a

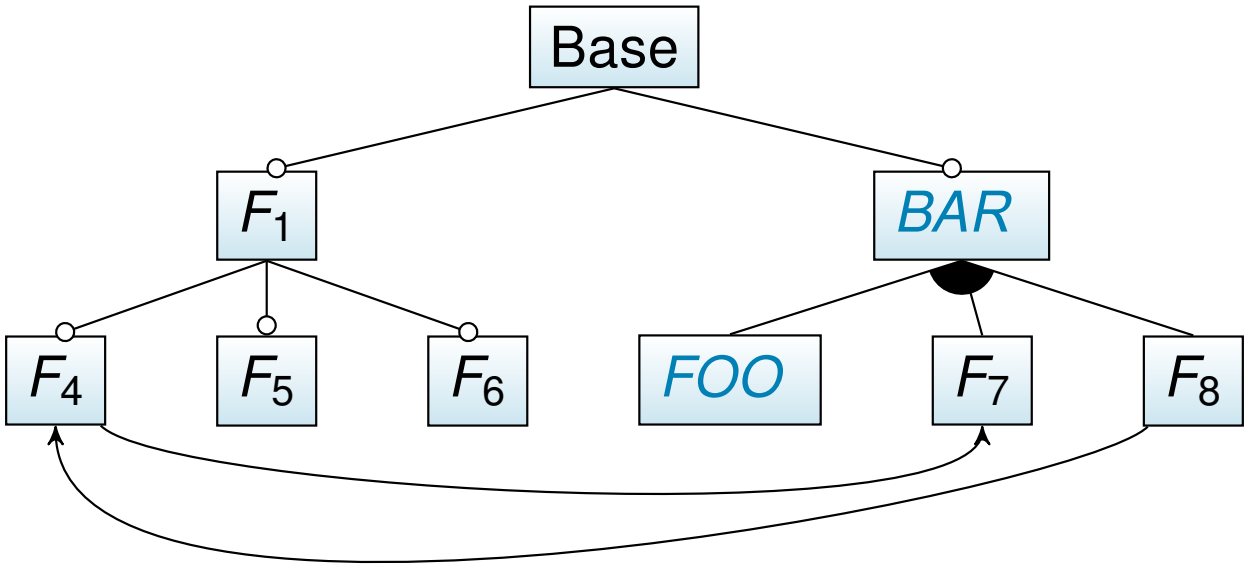
assets as possible, such as source code artifacts [8, 11, 27].

It is widely assumed that the variability mechanism of the cpp tool is used quite frequently in the implementation of SPLs [1, 12, 18, 33]. We believe this assumption to be true and contribute to the discussions on the connection between preprocessors and SPLs started in prior work with a substantial set of case studies. We answer the following questions: How does program size influence variability of SPLs? How complex are the extensions applied by features via cpp's variability mechanisms? At which level of granularity are extensions applied? Which types of extension occur? We argue that insights into the implementation problems of features solved with cpp help to judge whether cpp usage causes problems with regard to code pollution, error-proneness, and reduced maintainability and ability to evolve. An analysis of cpp usage also allows developers to estimate the effort and benefits of migrating to other, more well-founded implementation techniques for SPL engineering, such as aspects [17, 24] or various flavors of feature modules [5, 6].

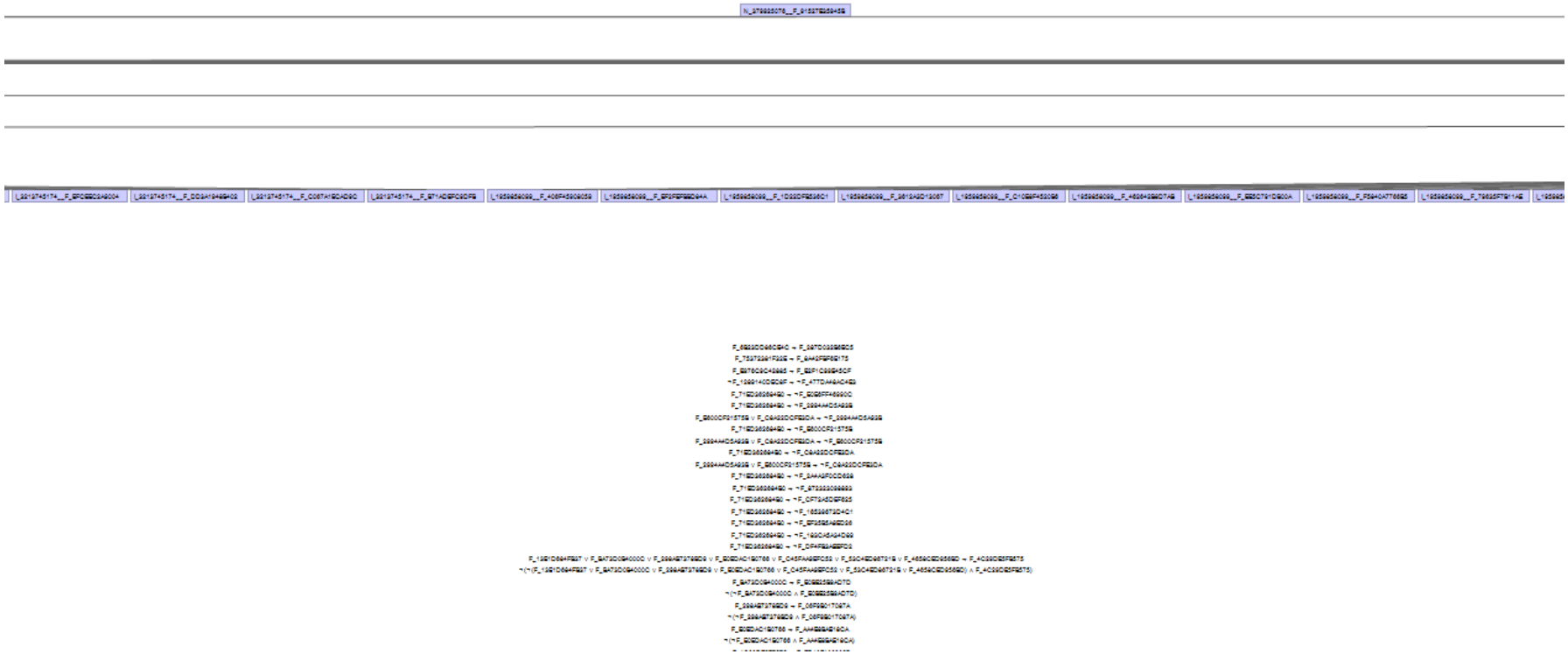
To answer the questions above, we analyzed forty open-source software systems of different sizes (thousands to millions lines of code) taken from different domains including operating systems, database systems, and compilers. We propose a set of metrics that allow us to infer and classify cpp usage patterns and to map the patterns to common SPL implementation concepts. Our analysis reveals that cpp is used to a large extent to implement and control variability

ICSE '10, May 2-8 2010, Cape Town, South Africa

Feature Models

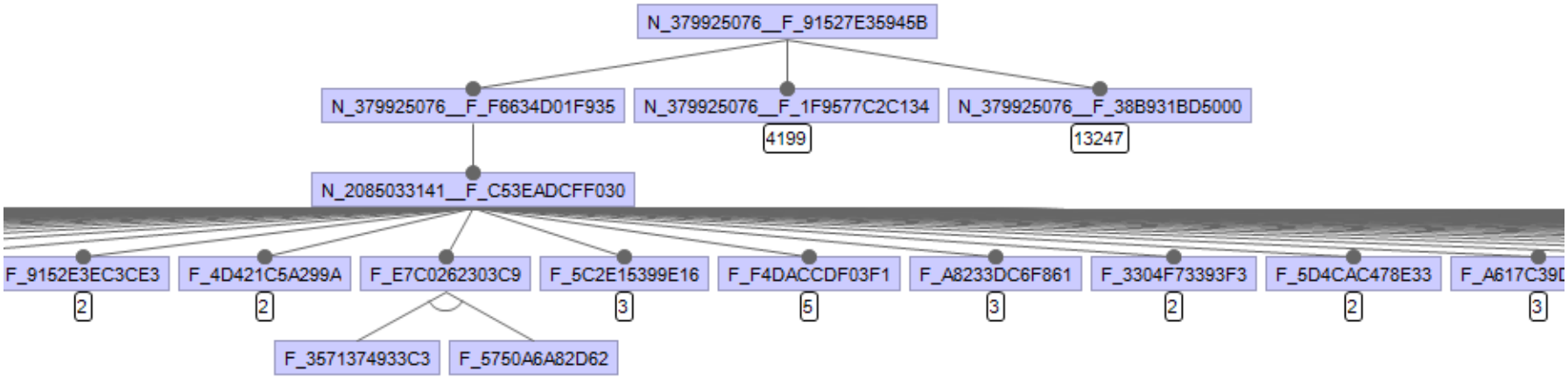


A Matter of Size



Feature models with thousands of features are challenging

A Matter of Size



$$F\_6B23DD86CE4C \Rightarrow F\_287D032B6EC5$$

$$F\_75372381F32E \Rightarrow F\_8A42FBF6E175$$

$$F\_A8233DC6F861 \wedge (F\_E528E7186064 \vee F\_536DAEEFC371 \vee F\_AD9CAB8377D4) \Rightarrow F\_1587476B18FF$$

$$F\_A4FD8098241E \wedge I\_2163186830\_\_F\_31B7538F7682 \Rightarrow F\_A62329B1F858$$

$$F\_8F6BEE85A1D3 \wedge I\_108785184\_\_F\_AEF4AFD38D98 \wedge F\_FDEAC6EBC906 \Rightarrow F\_7A7F1EBFAFDC$$

$$F\_0A7FB52A4A78 \wedge I\_3491892541\_\_F\_797E2906DD02 \Rightarrow F\_29CFF8E598E4$$

$$F\_FBFC87325974 \wedge I\_3441900933\_\_F\_E0A4648BDABC \wedge F\_FDEAC6EBC906 \vee F\_8F6BEE85A1D3 \wedge I\_108785184\_\_F\_AEF4AFD38D98 \wedge F\_A6909BC6DCEB \Rightarrow F\_0F39B09522E2$$

Feature models with thousands of features are challenging

Why to Solve Lots of SAT Problems?

```
#ifdef BAR
int x = 0;
#endif
```

Why to Solve Lots of SAT Problems?

```
#ifdef BAR
int x = 0;
#endif

// ...

#ifdef FOO
x++;
#endif
```

Why to Solve Lots of SAT Problems?

```
#ifdef BAR
int x = 0;
#endif

// ...

#ifdef FOO
x++;
#endif
```

$FOO \Rightarrow BAR$?

Why to Solve Lots of SAT Problems?

```
#ifdef BAR
int x = 0;
#endif

// ...

#ifdef FOO
x++;
#endif
```

$FM \models FOO \Rightarrow BAR$?

Why to Solve Lots of SAT Problems?

```
#ifdef BAR
int x = 0;
#endif

// ...

#ifdef FOO
x++;
#endif
```

$$\neg SAT(FM \wedge (FOO \Rightarrow BAR))\ ?$$

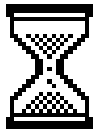
Why to Solve Lots of Similar SAT Problems?

```
#ifdef BAR
int x = 0;
#endif

// ...

#ifdef FOO
x++;
#endif
```

$$\neg SAT(FM \wedge (FOO \Rightarrow BAR))\ ?$$

Why to Solve Lots of Similar SAT Problems?

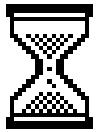
```
#ifdef BAR
int x = 0;
#endif

// ...

#ifdef FOO
x++;
#endif
```



$\neg SAT(FM \wedge (FOO \Rightarrow BAR))$?

Why to Solve Lots of Similar SAT Problems?

```
#ifdef BAR
int x = 0;
#endif

// ...

#ifdef FOO
x++;
#endif
```

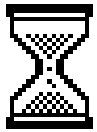


$\neg SAT(FM \wedge (FOO \Rightarrow BAR))$?

Idea: reduce size of feature model

Why to Solve Lots of Similar SAT Problems?

```
#ifdef BAR
int x = 0;
#endif

// ...

#ifdef FOO
x++;
#endif
```



$\neg SAT(FM \wedge (FOO \Rightarrow BAR))$?

Idea: reduce size of feature model

- Type checking
- Parsing
- Dataflow analysis
- Model checking
- Deductive verification
- Refactoring
- Feature-model analysis
- Configuration process

Part II

Compositional Analyses with Feature-Model Interfaces

Compositionality Principle

$$(FOO \Rightarrow X) \wedge Y \wedge Z \wedge (X \Rightarrow BAR) \models FOO \Rightarrow BAR$$

Compositionality Principle

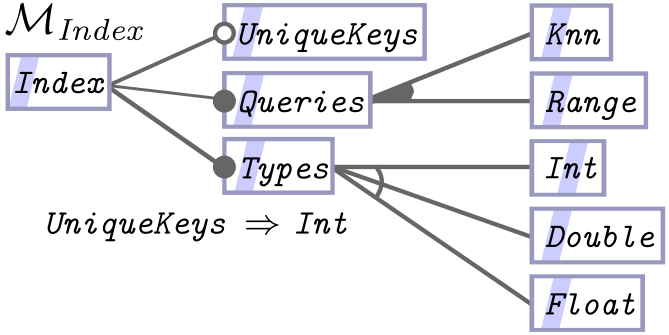
$$(FOO \Rightarrow X) \wedge Y \wedge Z \wedge (X \Rightarrow BAR) \models FOO \Rightarrow BAR$$

Compositionality Principle

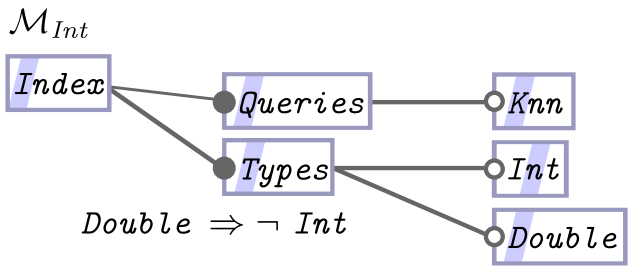
$$(FOO \Rightarrow X) \wedge Y \wedge Z \wedge (X \Rightarrow BAR) \models FOO \Rightarrow BAR$$

$$(FOO \Rightarrow X) \wedge (X \Rightarrow BAR) \models FOO \Rightarrow BAR$$

Feature-Model Interfaces [ICSE'16]



Interface Generation



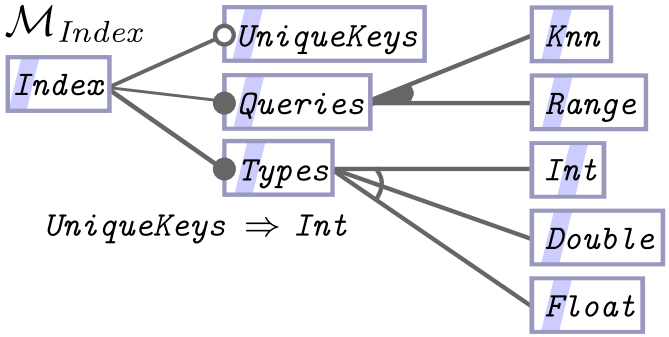
$\mathcal{F}_{Index} = \{$ I U Q T K R n D F $\}$

$\mathcal{P}_{Index} = \{\{$ I U Q T K R n $\}$
$\{$ I Q T K R n $\}$
$\{$ I U Q T K n $\}$
$\{$ I Q T K n $\}$
$\{$ I U Q T R n $\}$
$\{$ I Q T R n $\}$
$\{$ I Q T K R D $\}$
$\{$ I Q T K D $\}$
$\{$ I Q T R D $\}$
$\{$ I Q T K R F $\}$
$\{$ I Q T K F $\}$
$\{$ I Q T R F $\}\}$

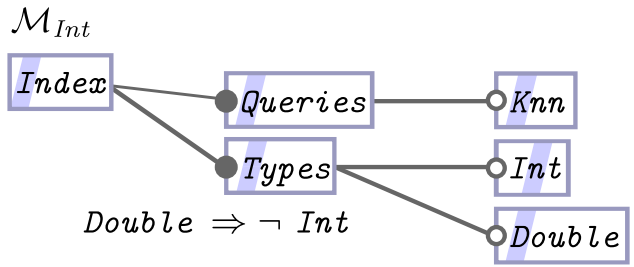
$\mathcal{F}_{Int} = \{$ I U Q T K R n D F $\}$

$\mathcal{P}_{Int} = \{\{$ I U Q T K R n $\}$
$\{$ I Q T K R n $\}$
$\{$ I U Q T K n $\}$
$\{$ I Q T K n $\}$
$\{$ I U Q T R n $\}$
$\{$ I Q T R n $\}$
$\{$ I Q T K R D $\}$
$\{$ I Q T K D $\}$
$\{$ I Q T R D $\}$
$\{$ I Q T K R F $\}$
$\{$ I Q T K F $\}$
$\{$ I Q T R F $\}\}$

Feature-Model Interfaces [ICSE'16]



Interface Generation



$\mathcal{F}_{Index} = \{$ I U Q T K R n D F $\}$

$\mathcal{P}_{Index} = \{\{$ I U Q T K R n $\}$
$\{$ I Q T K R n $\}$
$\{$ I U Q T K n $\}$
$\{$ I Q T K n $\}$
$\{$ I U Q T R n $\}$
$\{$ I Q T R n $\}$
$\{$ I Q T K R D $\}$
$\{$ I Q T K D $\}$
$\{$ I Q T R D $\}$
$\{$ I Q T K R F $\}$
$\{$ I Q T K F $\}$
$\{$ I Q T R F $\}\}$

$\mathcal{F}_{Int} = \{$ I U Q T K R n D F $\}$

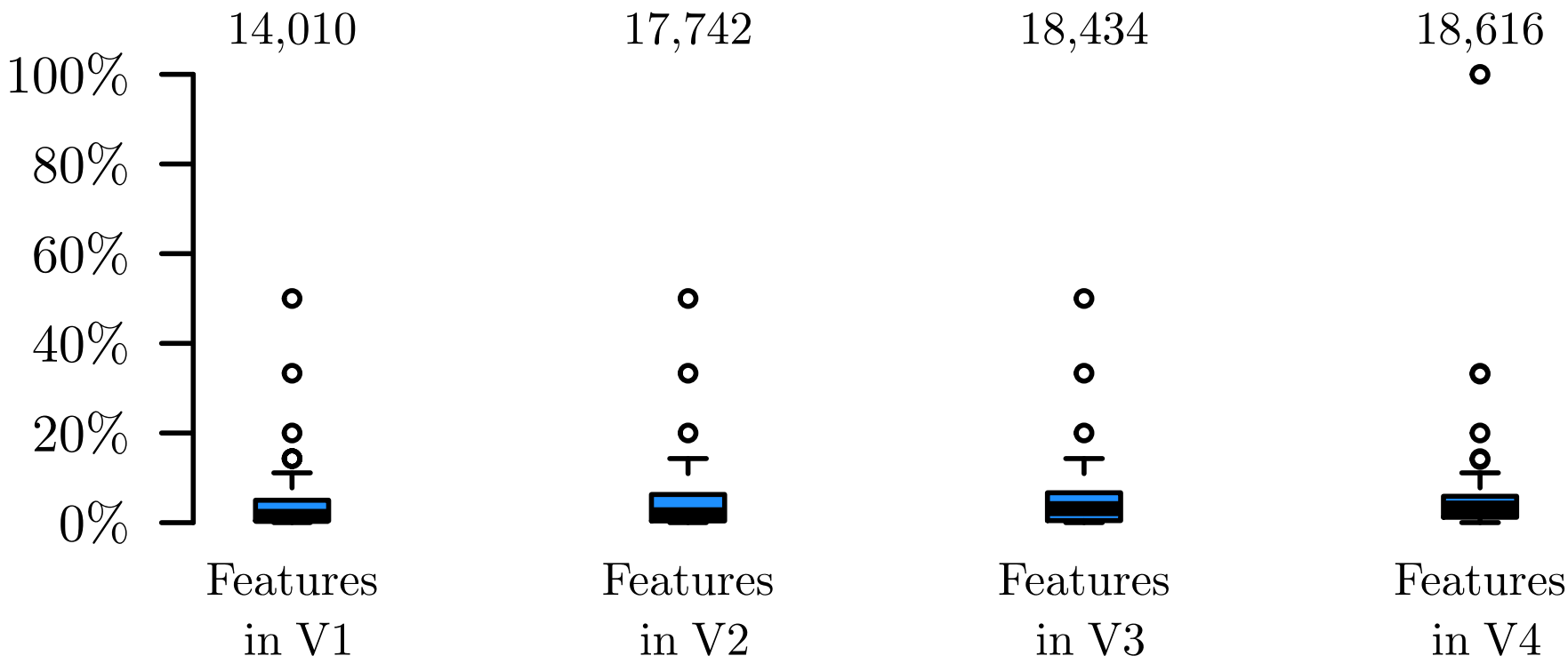
$\mathcal{P}_{Int} = \{\{$ I U Q T K R n $\}$
$\{$ I Q T K R n $\}$
$\{$ I U Q T K n $\}$
$\{$ I Q T K n $\}$
$\{$ I U Q T R n $\}$
$\{$ I Q T R n $\}$
$\{$ I Q T K R D $\}$
$\{$ I Q T K D $\}$
$\{$ I Q T R D $\}$
$\{$ I Q T K R F $\}$
$\{$ I Q T K F $\}$
$\{$ I Q T R F $\}\}$

For scalability: features eliminated from propositional formulas with existential quantification

Reduced Feature-Model Size [ICSE'16]

Given four monthly snapshots of an automotive feature model growing from 14k to 18k features

RQ1: How small can interfaces be compared to submodels?

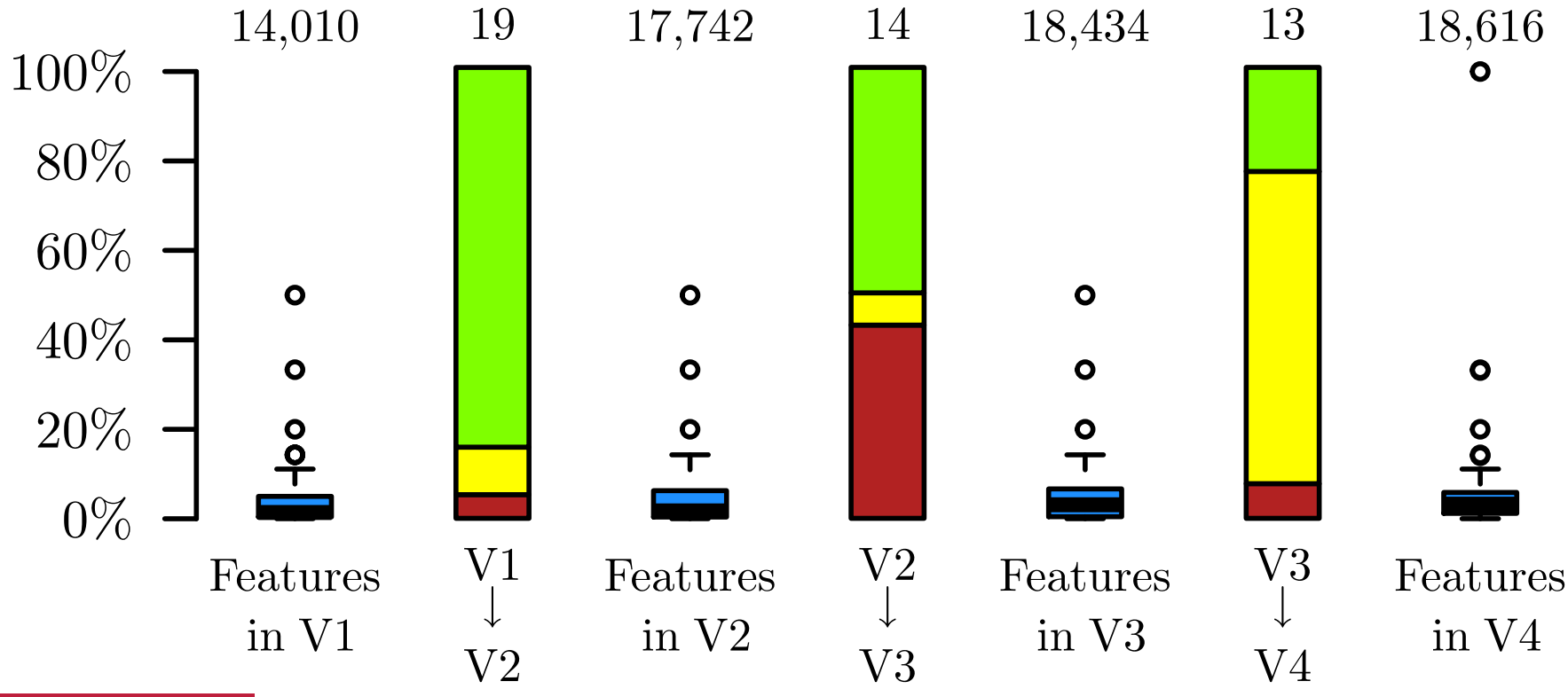


Reduced Feature-Model Size [ICSE'16]

Given four monthly snapshots of an automotive feature model growing from 14k to 18k features

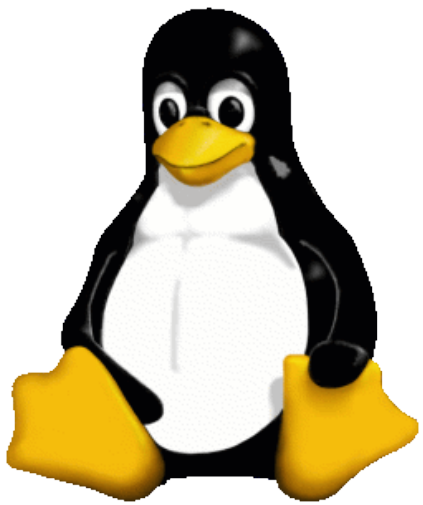
RQ1: How small can interfaces be compared to submodels?

RQ2: How often are feature-model interfaces compatible?



Reduced Effort in Type Checking Linux?

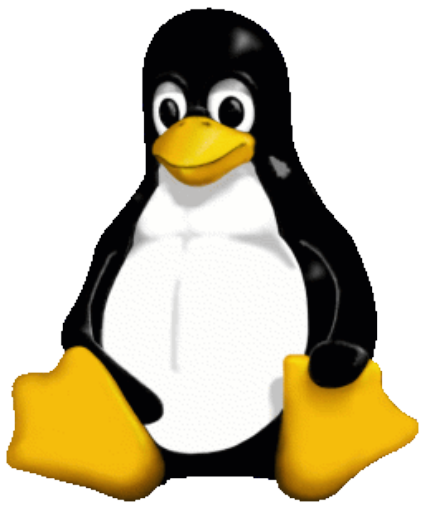
- Family-based type checking with TypeChef
- Linux kernel version 2.6.33.3 with 11,000 features
- 1,363 out of 7,760 files



Reduced Effort in Type Checking Linux?

- Family-based type checking with TypeChef
- Linux kernel version 2.6.33.3 with 11,000 features
- 1,363 out of 7,760 files

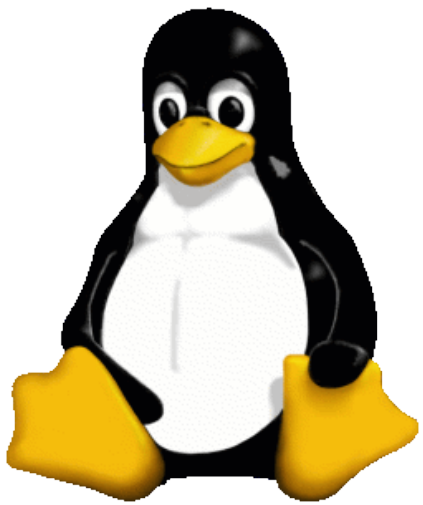
- 173,845 queries
- 90ms per query



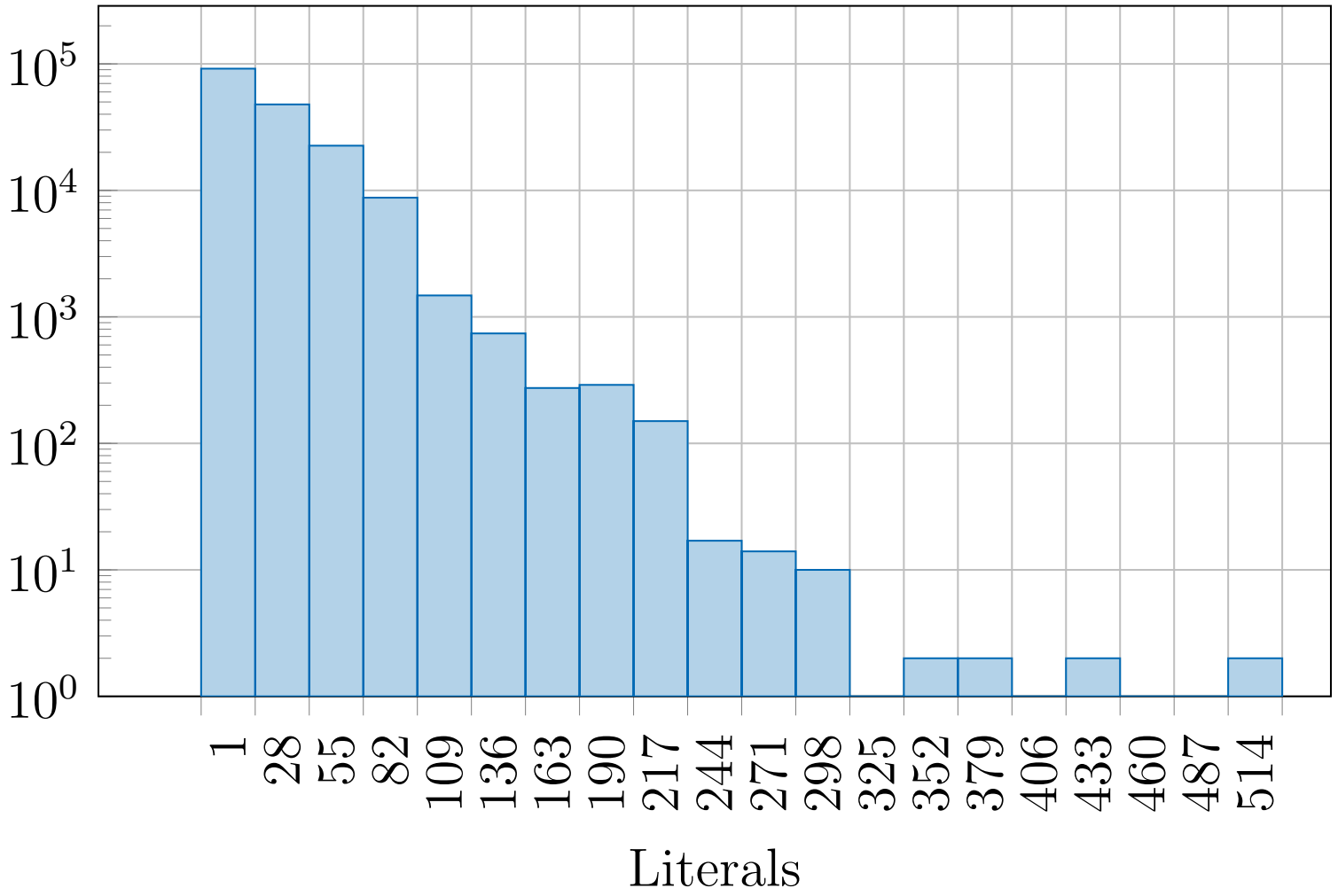
Reduced Effort in Type Checking Linux?

- Family-based type checking with TypeChef
- Linux kernel version 2.6.33.3 with 11,000 features
- 1,363 out of 7,760 files

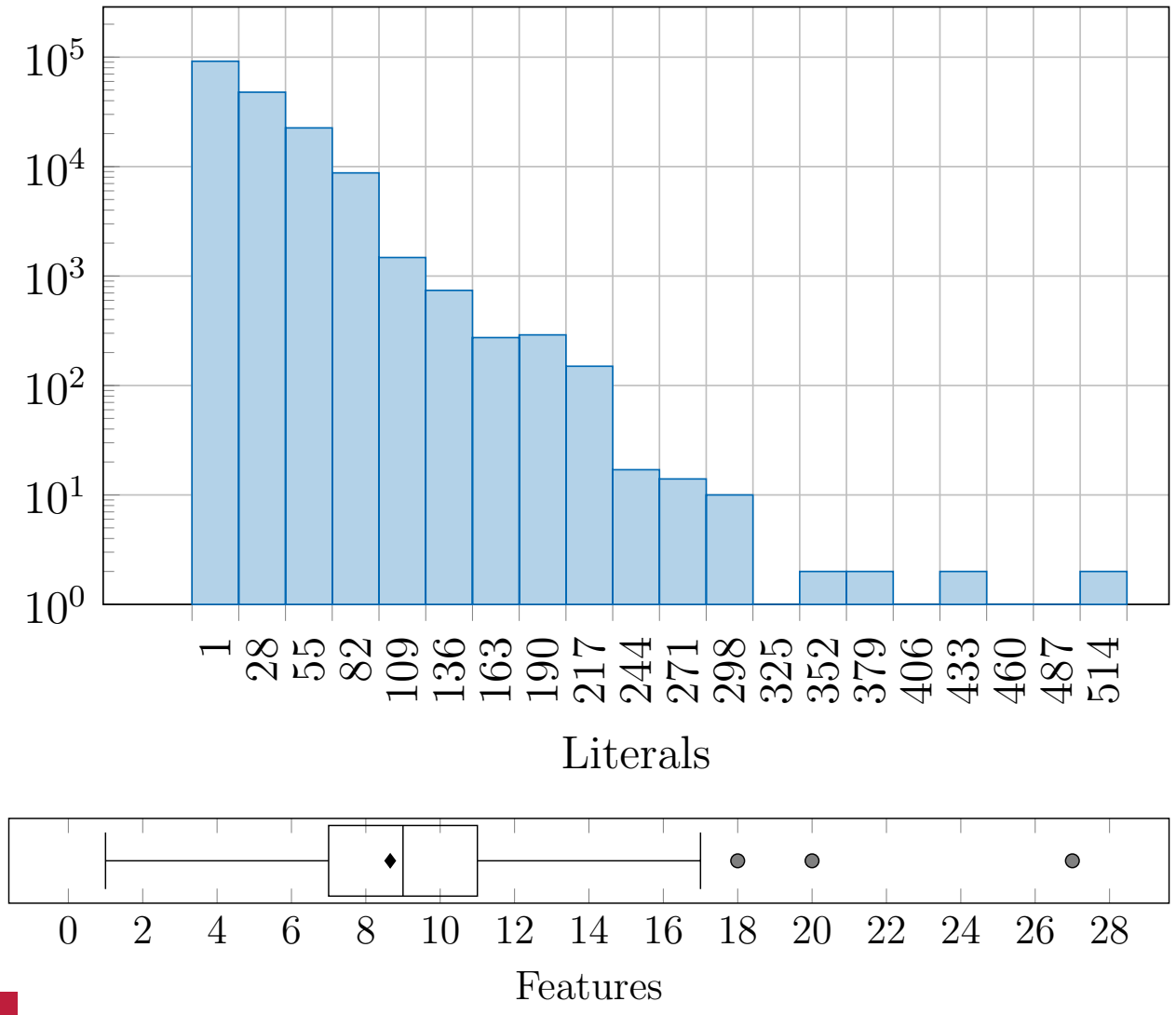
- 173,845 queries
- 90ms per query
- 12s reasoning time per file
- Reasoning: 60% of type checking phase



Literals and Distinct Features per Query



Literals and Distinct Features per Query



Apply Linux Queries to Automotive Feature Model

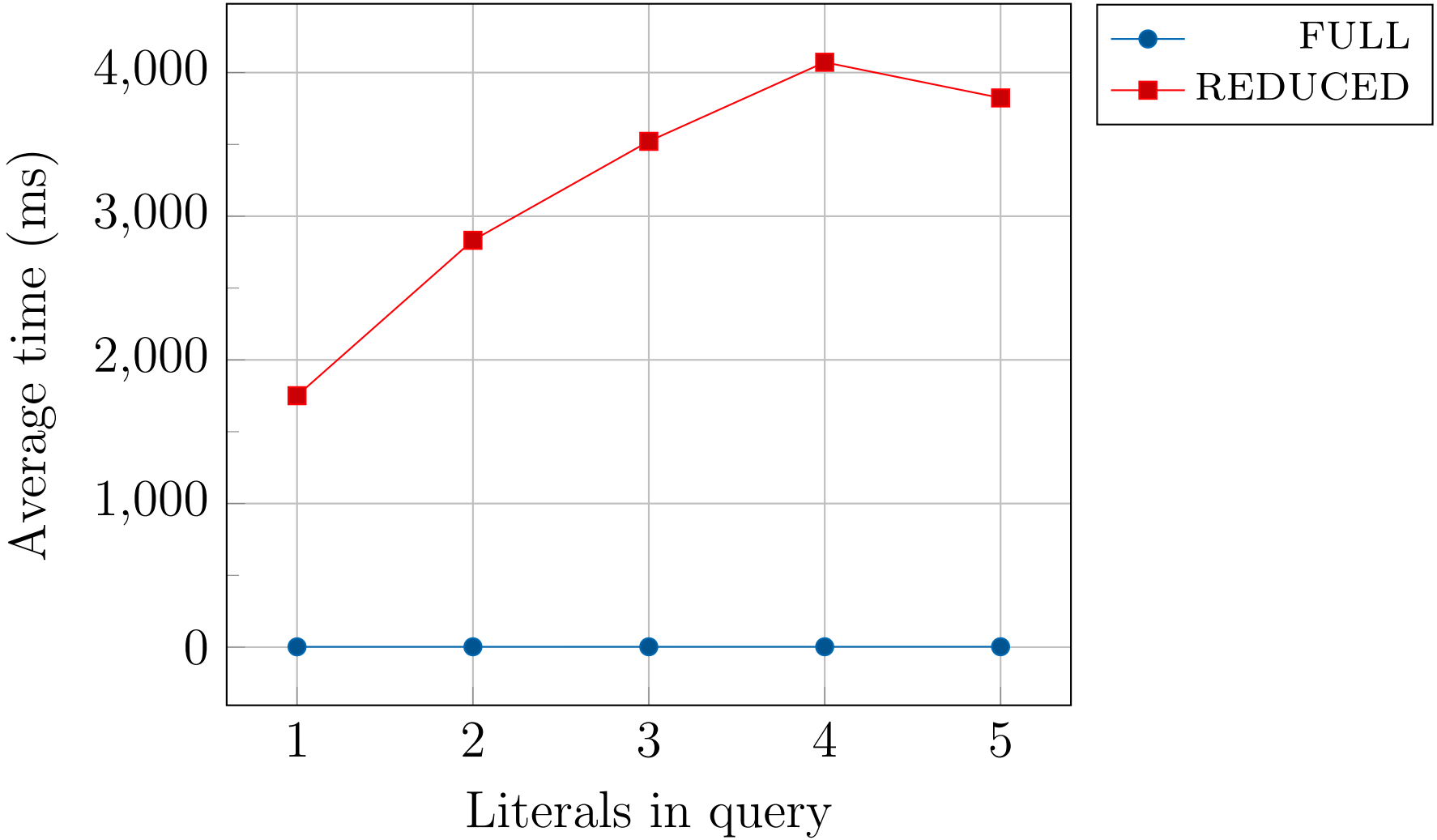
Problem: no decomposition for Linux feature model and no domain artifacts/queries for automotive

Idea: use queries from Linux evaluation as templates

$$(\neg A \wedge B) \vee A \quad \curvearrowright \quad (\neg P \wedge Q) \vee P$$

Snapshot	Features	Constraints	Clauses	Submodels
1	14,010	666	237,706	44
2	17,742	914	342,935	45
3	18,434	1,300	347,557	46
4	18,616	1,369	350,287	46

(No) Reduced Effort for Product-Line Analysis?

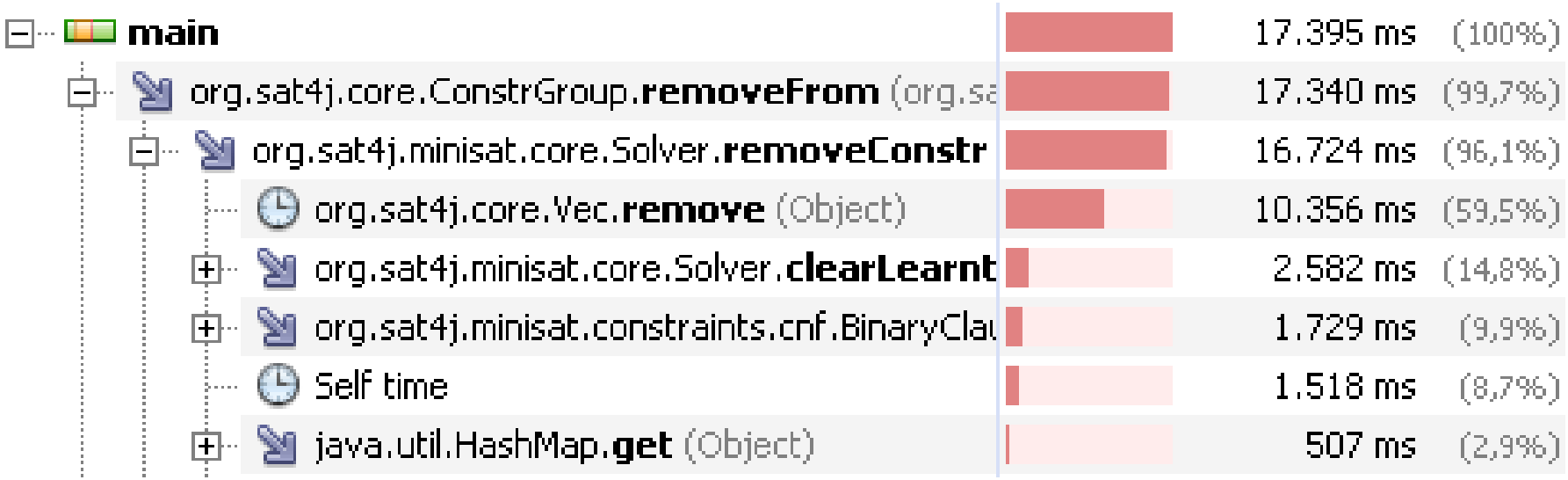


Part III

Solving Similar SAT/BDD/SMT Instances

Profiling the SAT4J Implementation

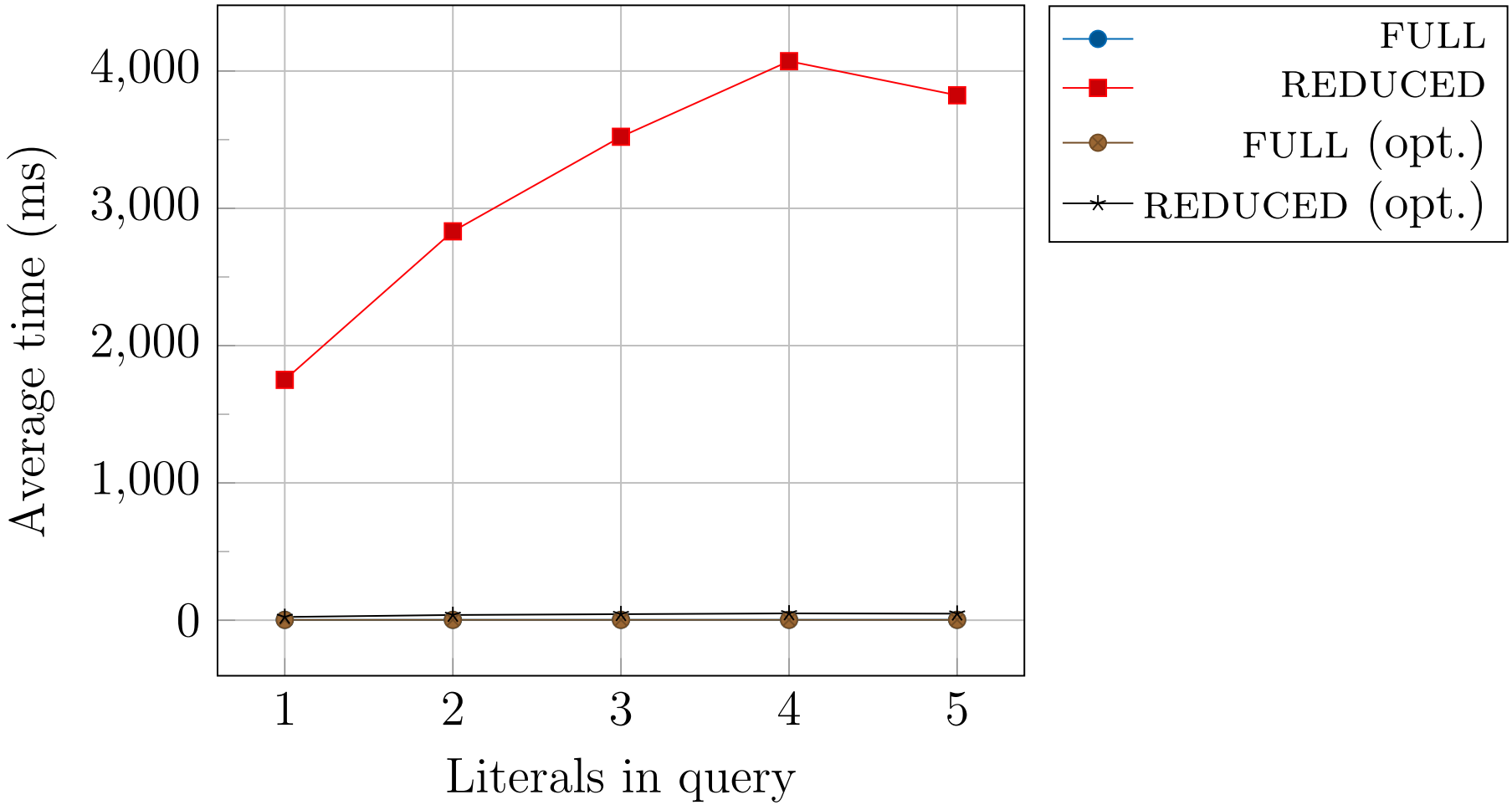
$$\neg SAT(FM \wedge (FOO \Rightarrow BAR))\ ?$$



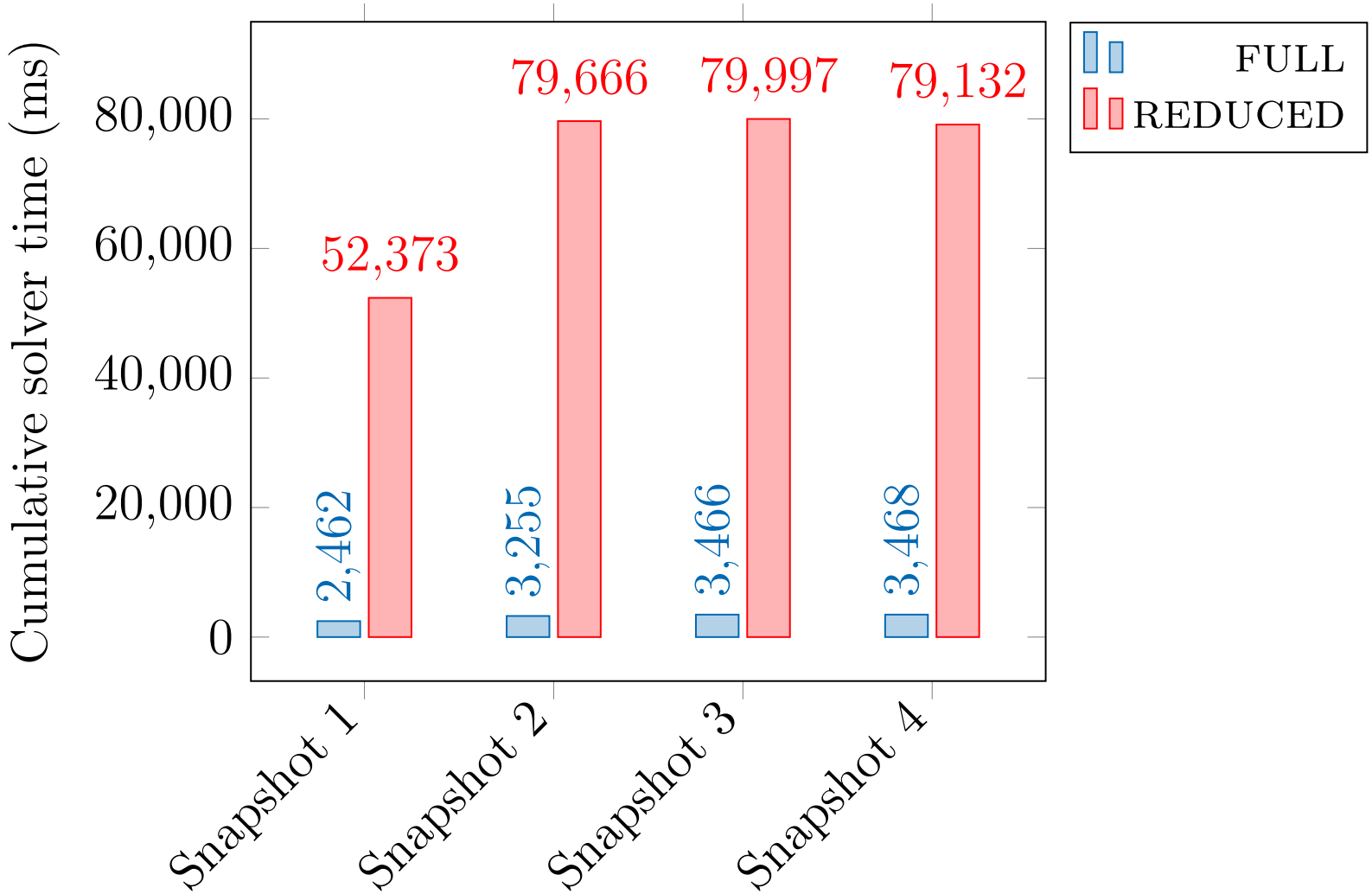
Profiling the SAT4J Implementation

```
public void remove(T elem) {
  int j = 0;

  for (; this.myarray[j] != elem; j++) {
    if (j == size())
      throw new NoSuchElementException();
  }

  System.arraycopy(this.myarray, j + 1,
  this.myarray, j, size() - j - 1);
  this.myarray[--this.nbelem] = null;
}
```

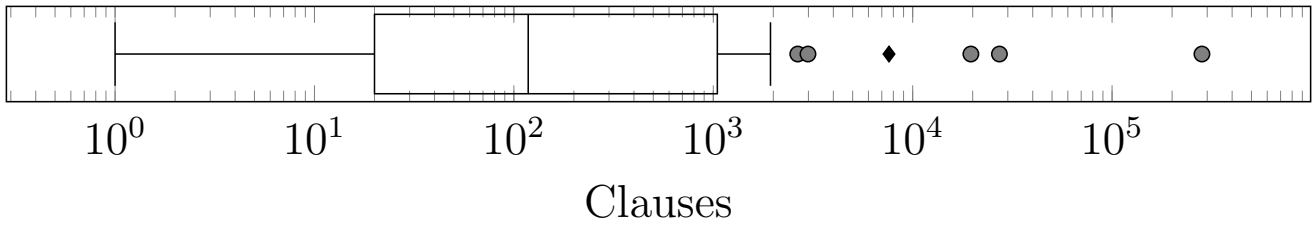
Effect of Our SAT4J Optimization



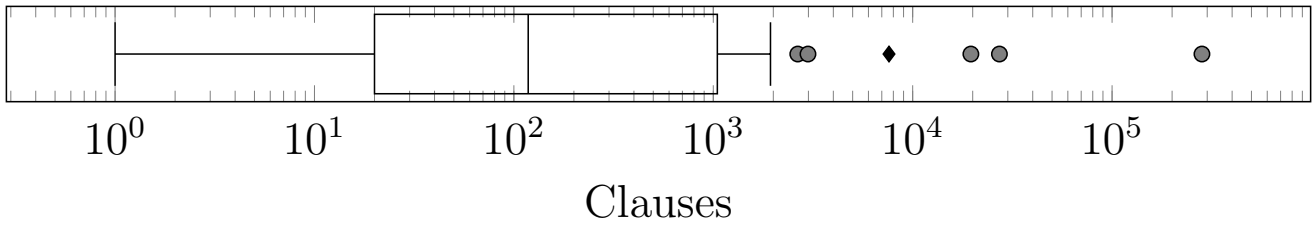
Cumulative Solver Time with Optimization

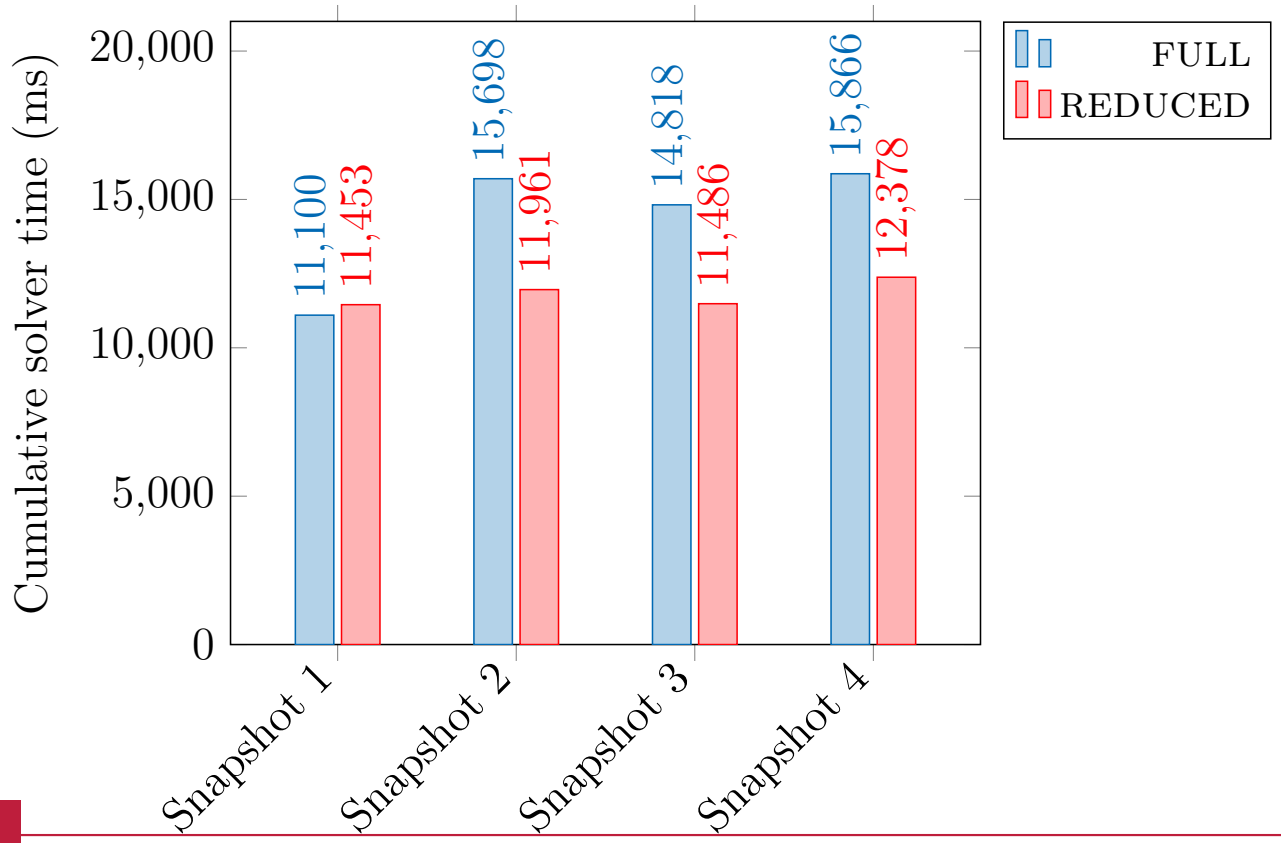


The Influence of Large Submodels

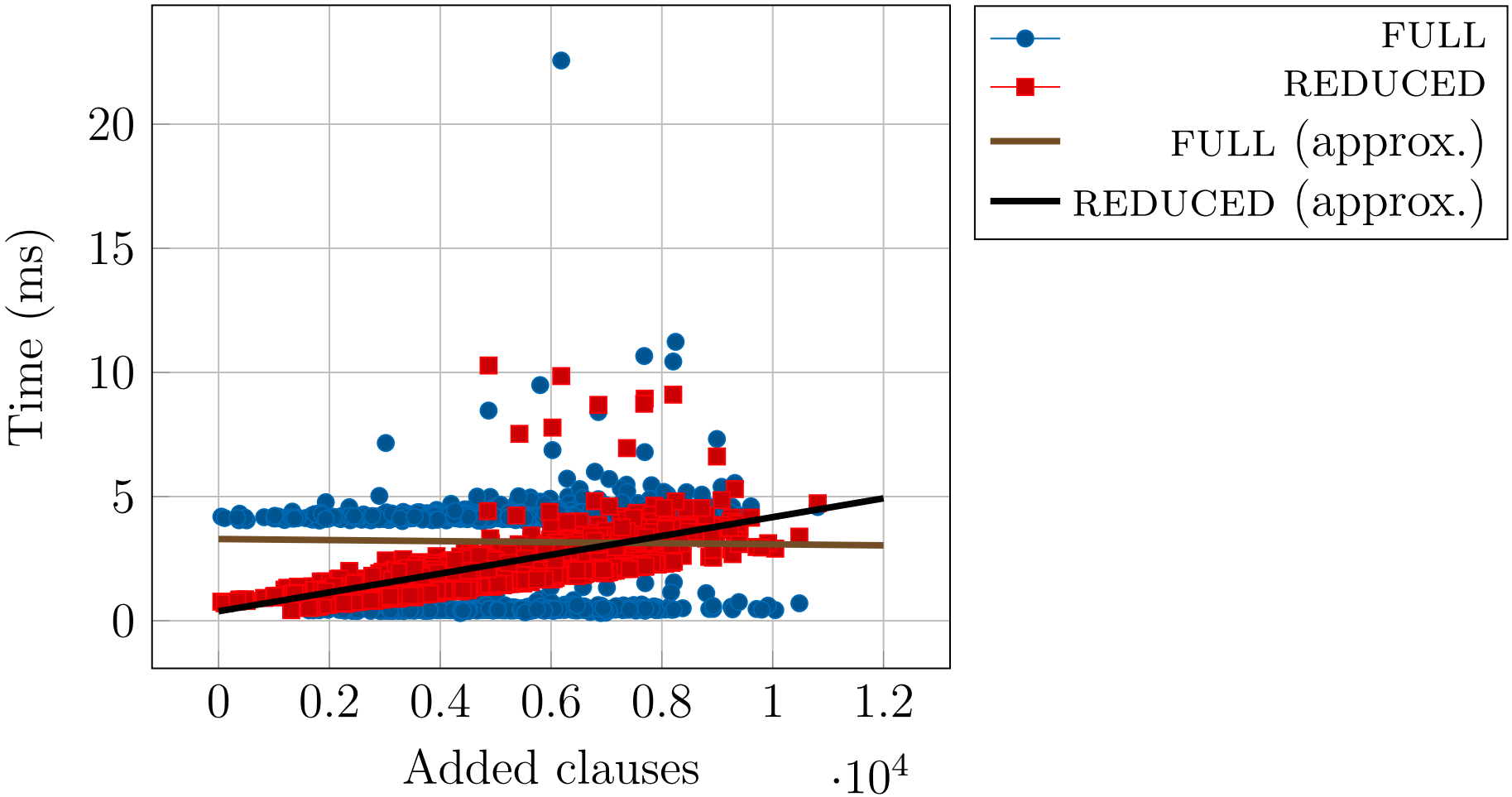


The Influence of Large Submodels

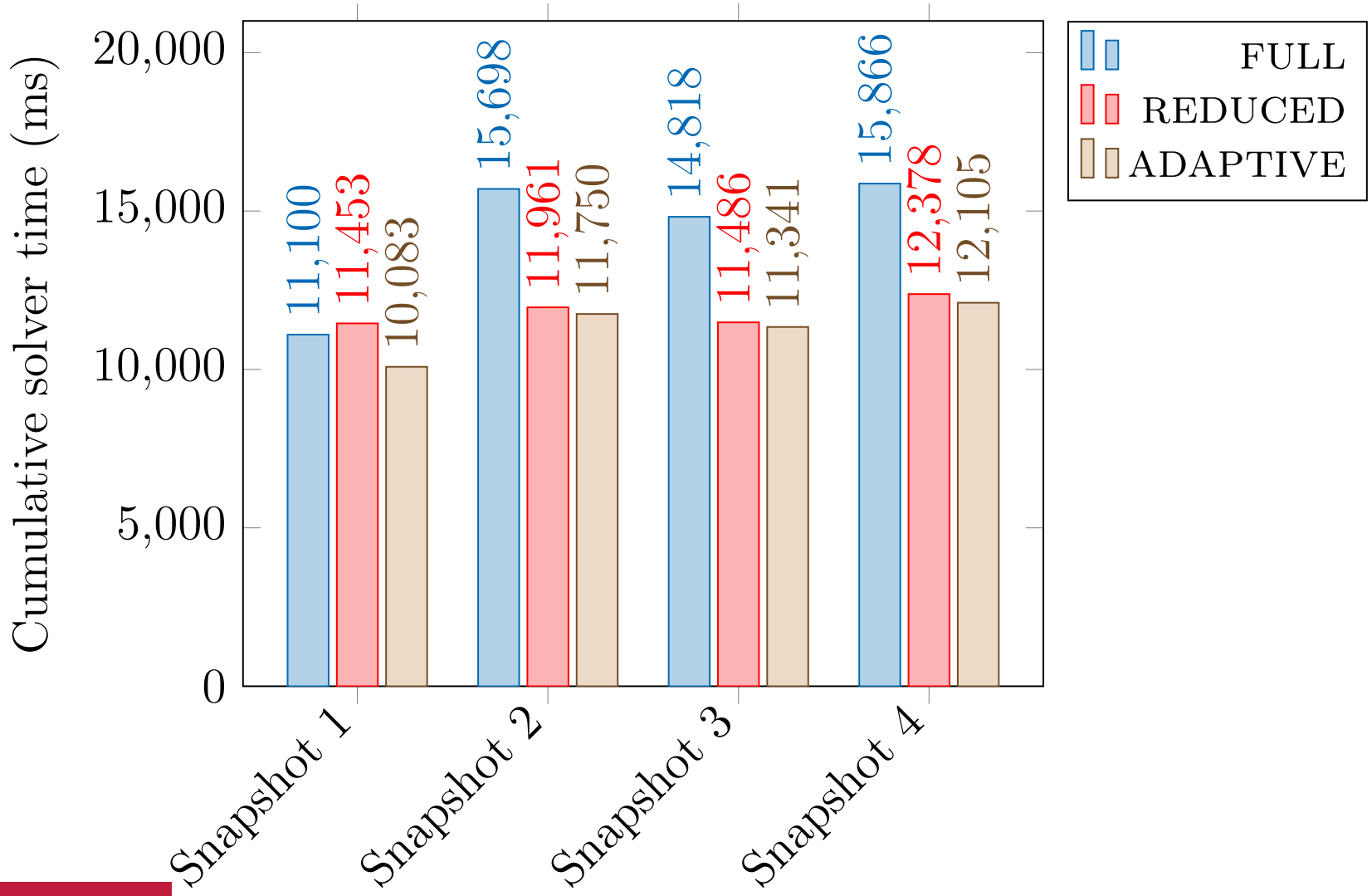




Time vs. Added Submodel Clauses



Cumulative Solver Times with Threshold



Solving Similar SAT/BDD/SMT Instances

- SAT: similar instances not part of satisfiability contests
  $\Rightarrow$ not optimized for this purpose
- BDD: no BDD for Linux, hard to join BDDs with different variable orderings
- SMT: recent developments for lightweight removal of formulas

How Should Superheroes Eat Cake?

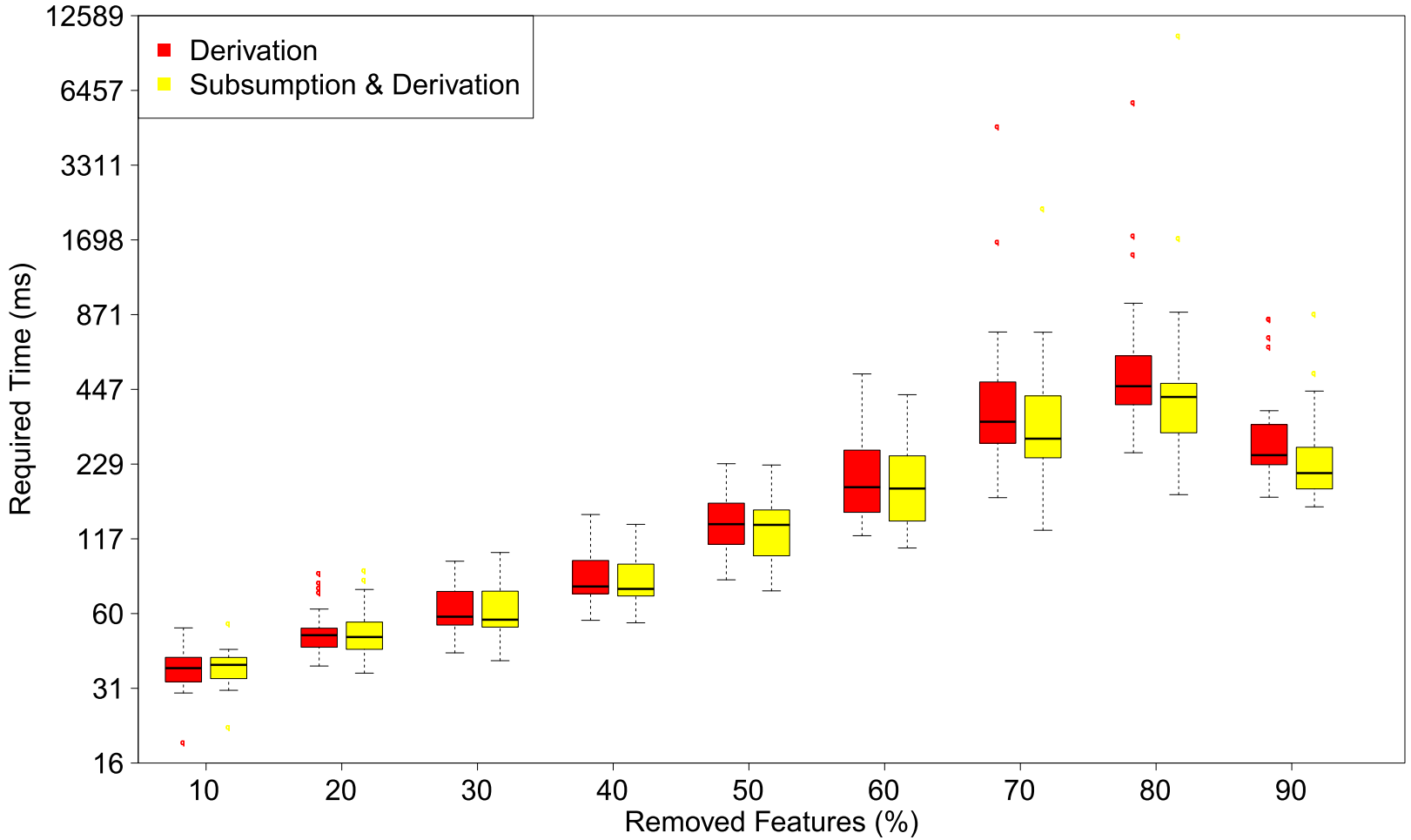
We Should Care about Similarity of SAT Problems

- Configurable software requires to solve many similar SAT instances: consumes 60% of type checking (without parsing)
- Feature-model interfaces can significantly reduce the similar part
- Performance gain for reasoning 10–24%
- Open question: What are good decompositions of feature models?
- Open question: Are solvers ready to solve similar instances?

Experience Report: How NOT to Travel to FOSD'17

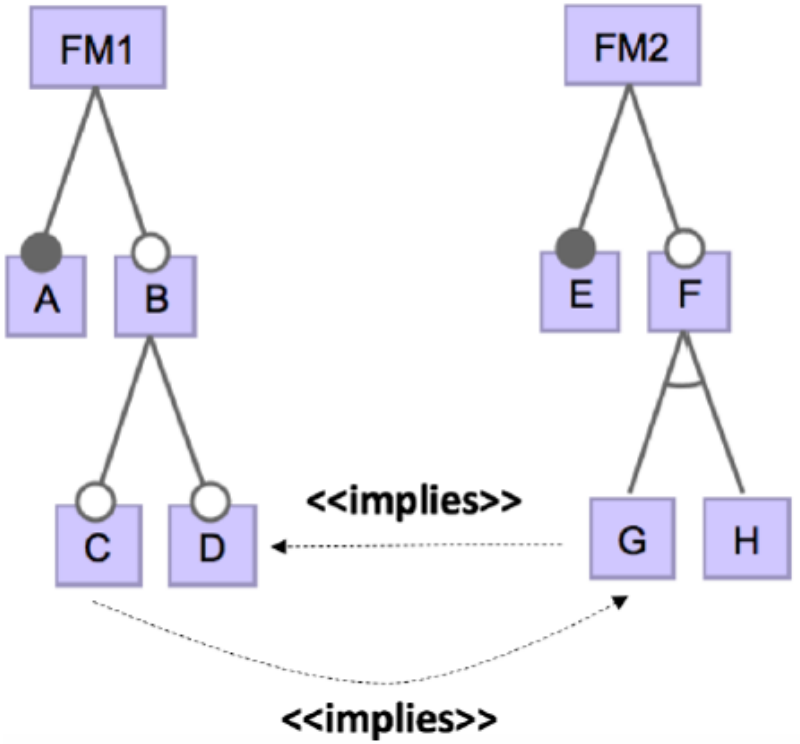
Decomposition on Demand? [SPLC'16]

Removing x % of 18k features of an automotive feature model:



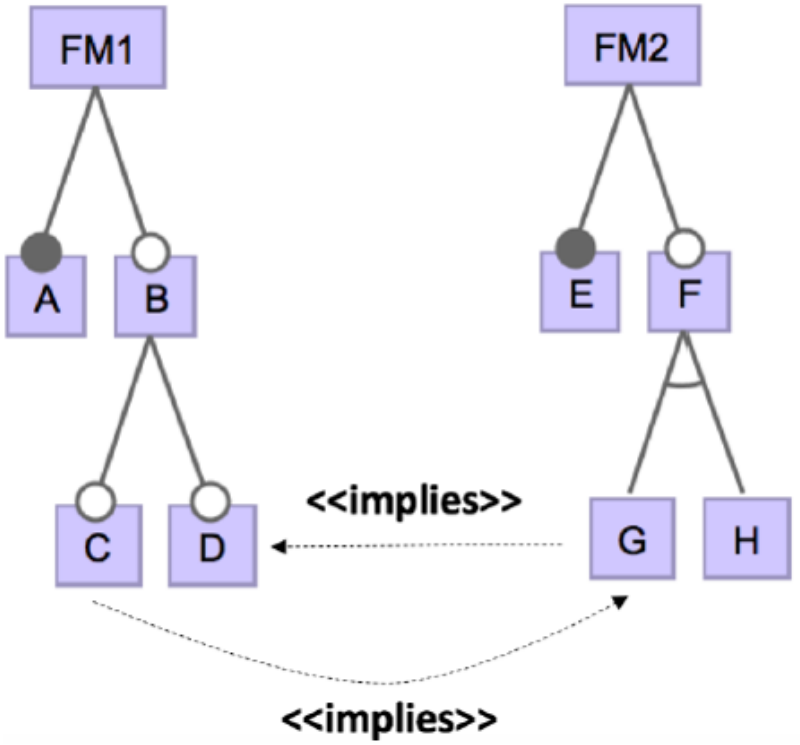
Make Implicit Constraints Explicit [FOSD'16]

What are implicit constraints?



Make Implicit Constraints Explicit [FOSD'16]

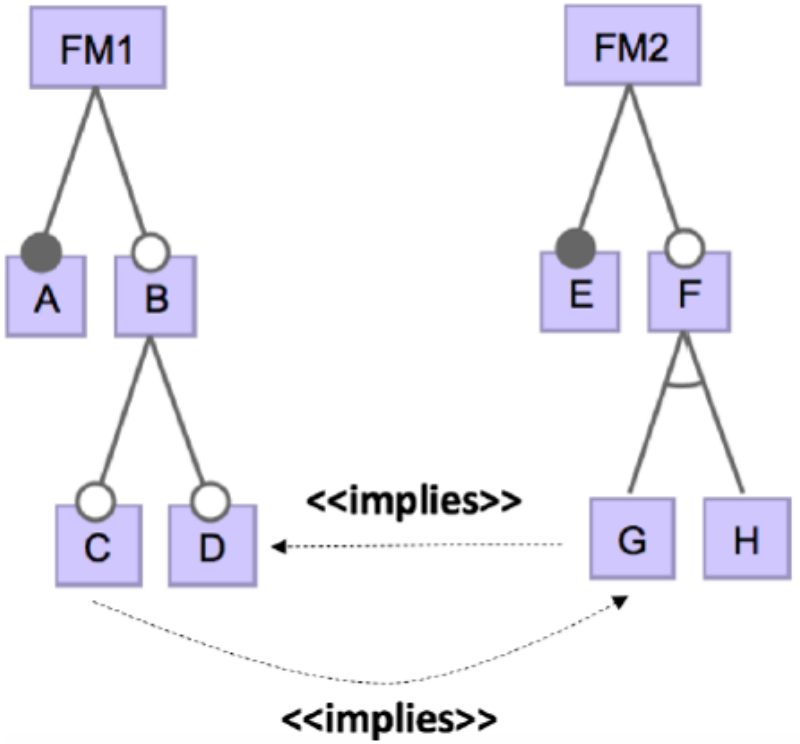
What are implicit constraints?

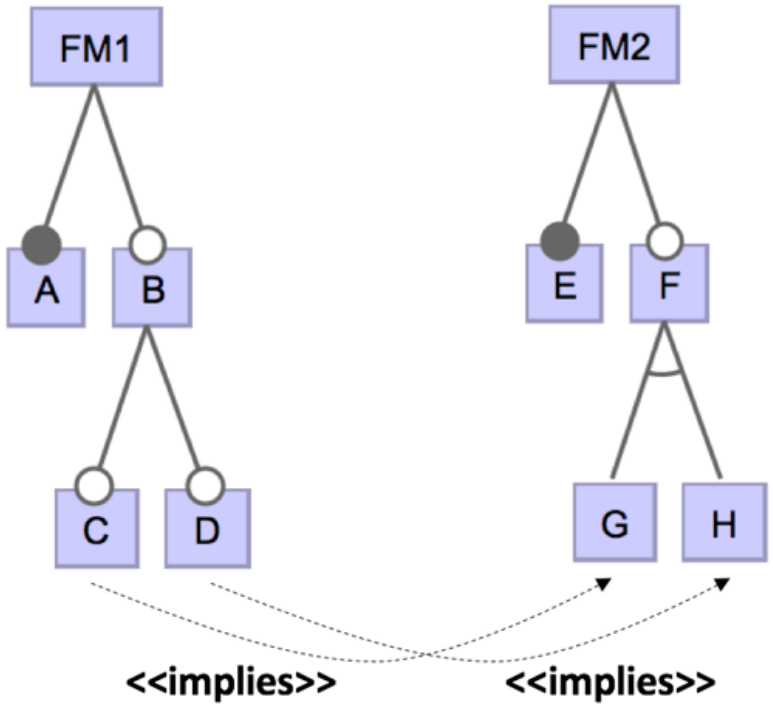


$C \Rightarrow D$

Make Implicit Constraints Explicit [FOSD'16]

What are implicit constraints?

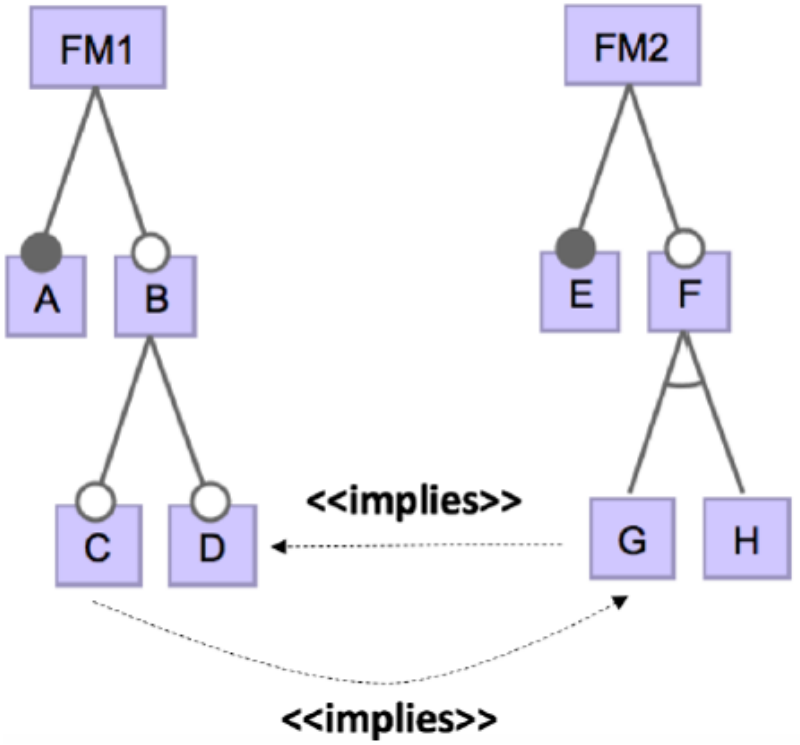


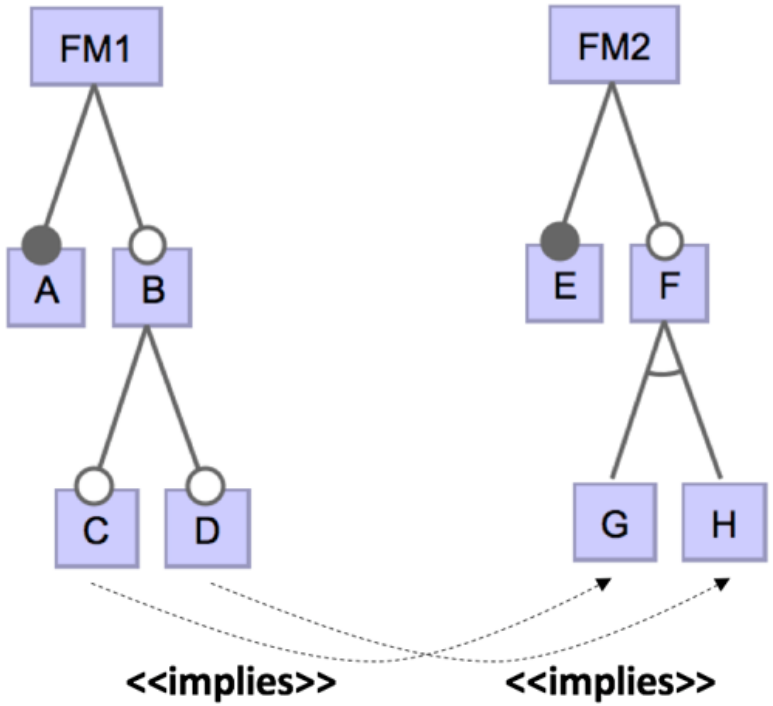


$C \Rightarrow D$

Make Implicit Constraints Explicit [FOSD'16]

What are implicit constraints?





$C \Rightarrow D$

$\neg C \vee \neg D$

Make Implicit Constraints Explicit [FOSD'16]

RQ1: How many implicit constraints exist in subtrees?

Automotive feature model with 2,513 features and 2,833 constraints

Depth 1: 6 features and 12 implicit constraints

Depth 2: 25 features and 186 implicit constraints

Make Implicit Constraints Explicit [FOSD'16]

RQ2: What is the structure of implicit constraints?

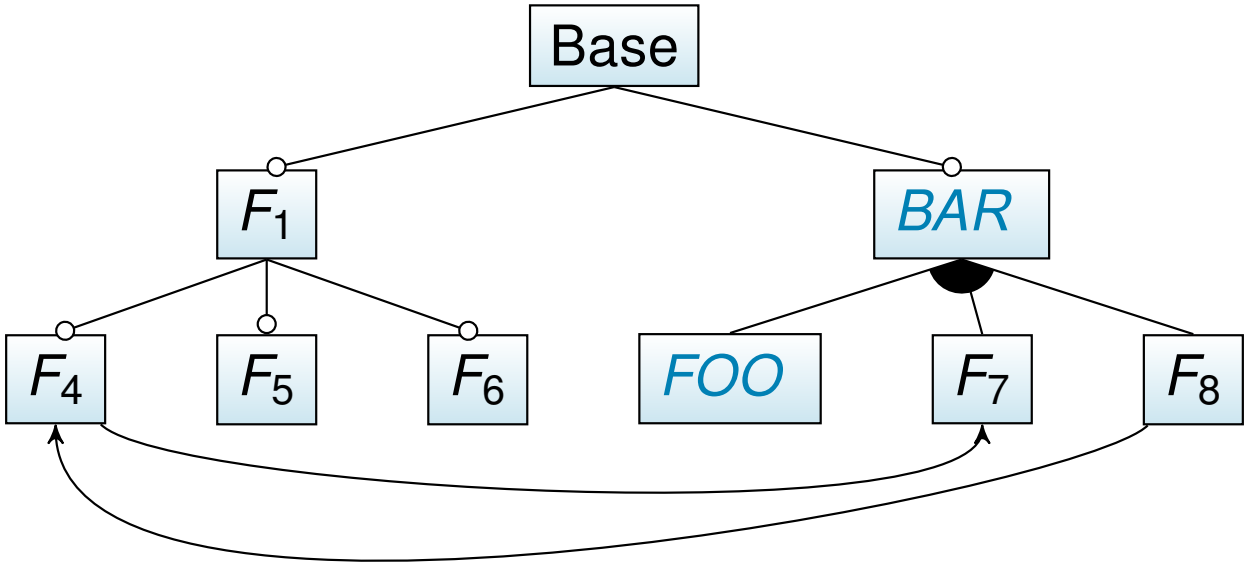
Expression	CNF Pattern	# I.C. Depth 1	# I.C. Depth 2	Overall (%)
Negation	¬ A	1	16	8,6
Implication	¬ A v B	11	18	14,6
Exclusion	¬ A v ¬ B	-	31	15,7
Other	A v B v C	-	1	0,5
	¬ A v ¬ B v C	-	115	58,1
	¬ A v B v C	-	5	2,5

RQ3: What is the number of involved subtrees?

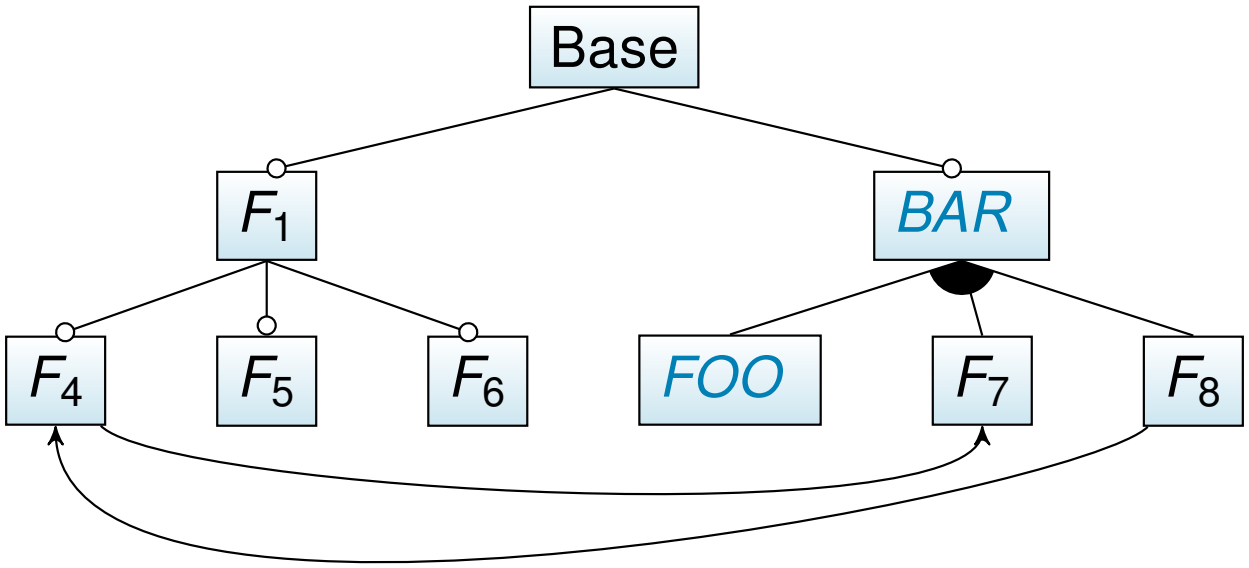
Depth 1: max. 4/6 partial models

Depth 2: max. 5/25 partial models

Decomposition with Feature-Model Interfaces

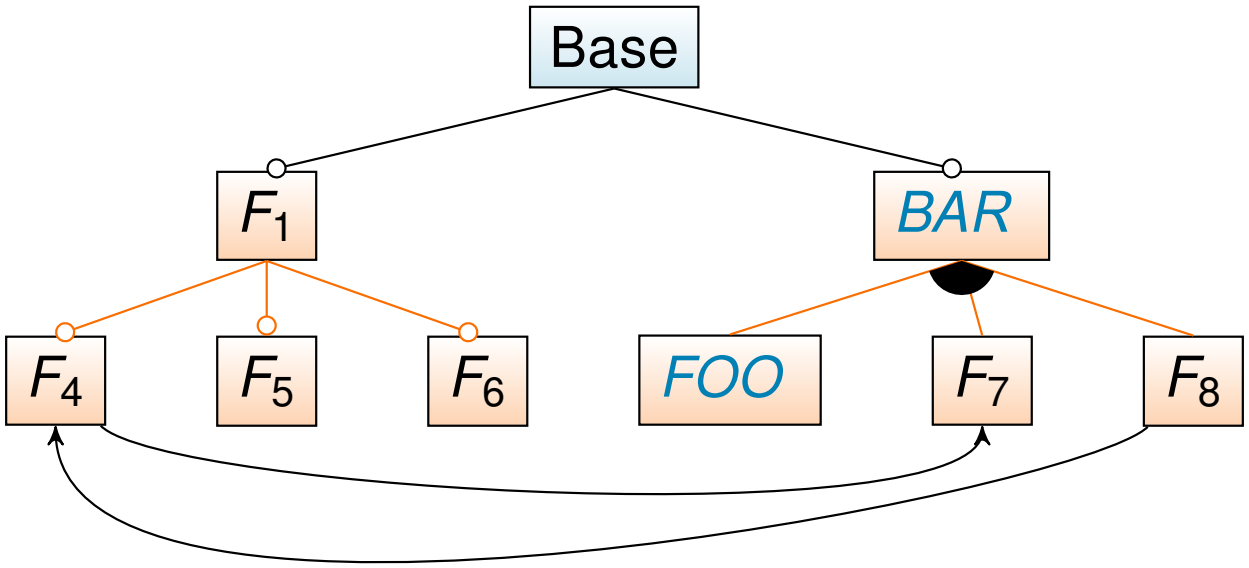


Decomposition with Feature-Model Interfaces



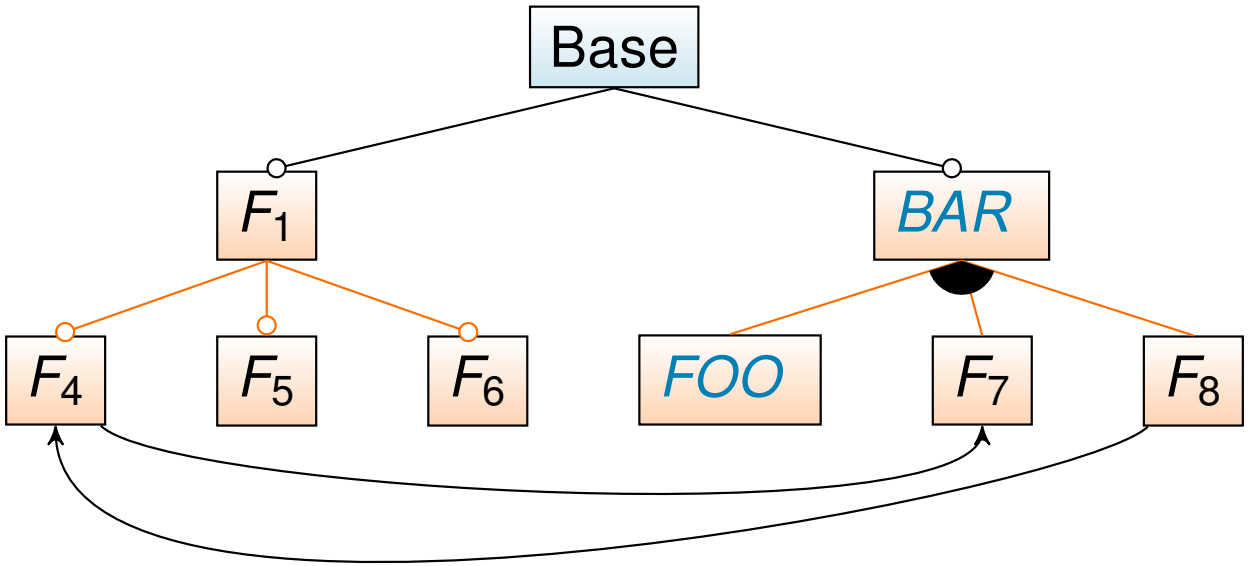
select subtrees

Decomposition with Feature-Model Interfaces



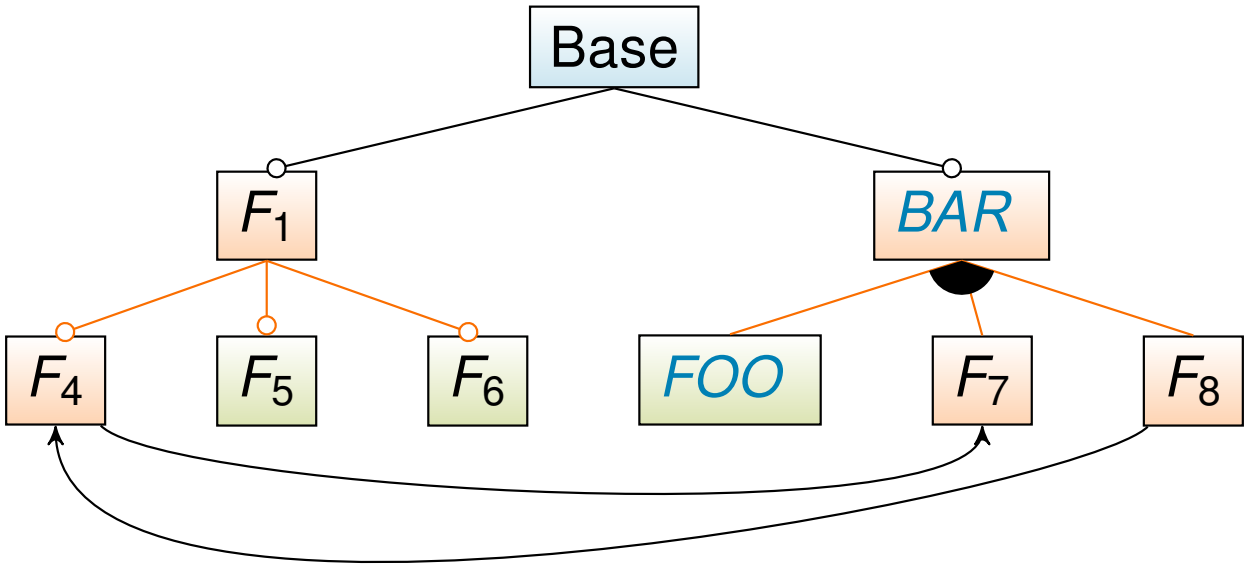
select subtrees

Decomposition with Feature-Model Interfaces



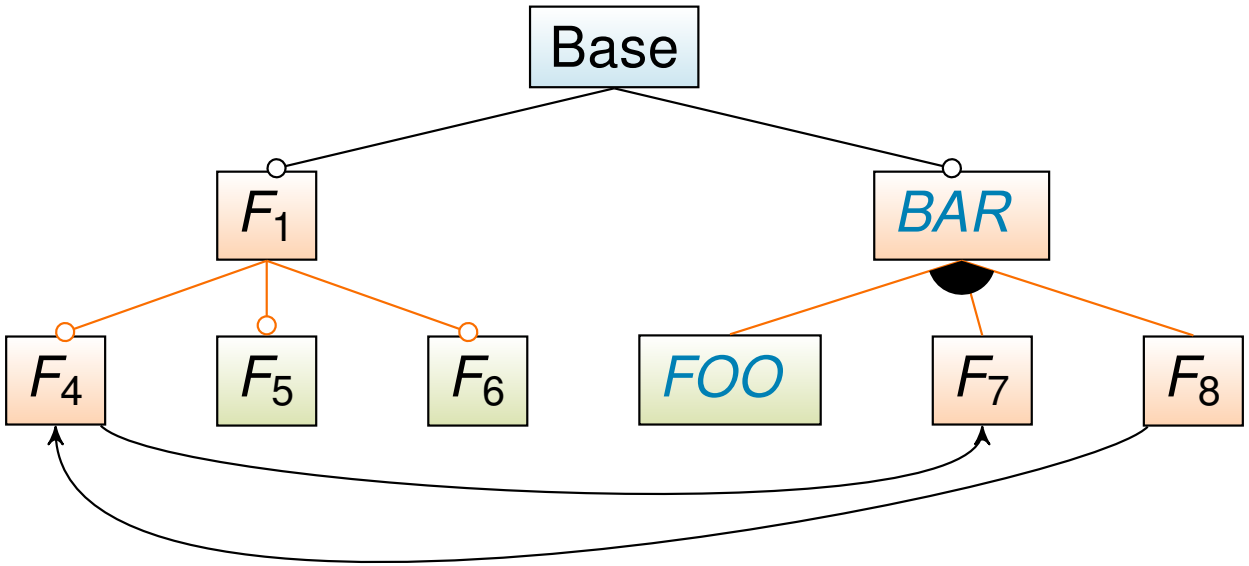
identify local features & local constraints

Decomposition with Feature-Model Interfaces



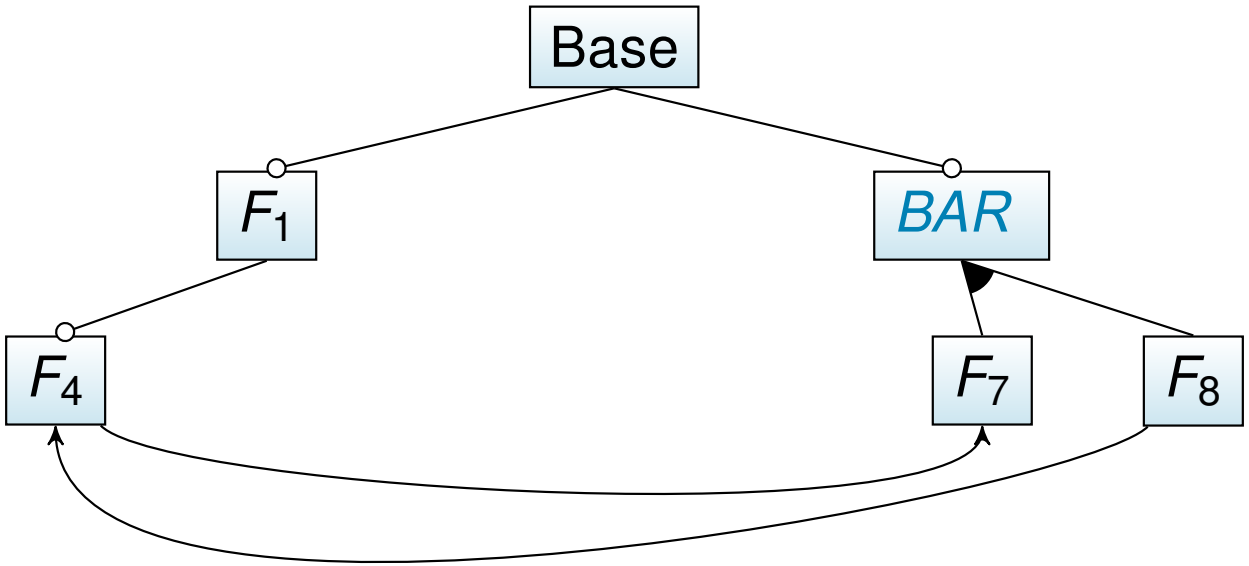
identify local features & local constraints

Decomposition with Feature-Model Interfaces



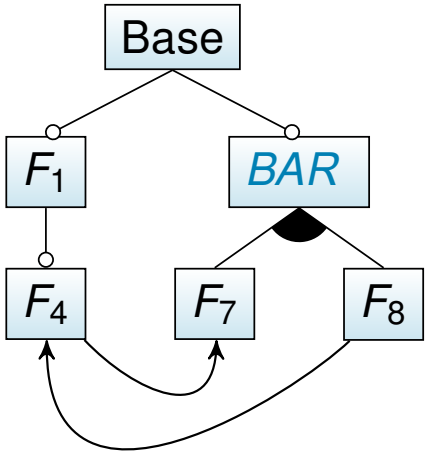
decompose model by removing local features

Decomposition with Feature-Model Interfaces

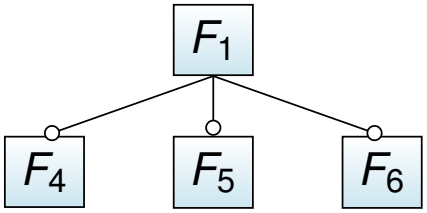


decompose model by removing local features

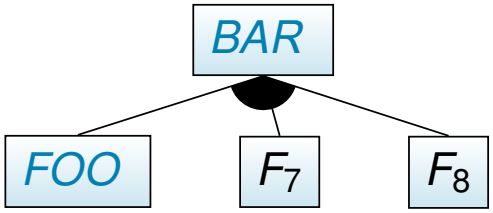
Compositionality with Feature-Model Interfaces



Γ

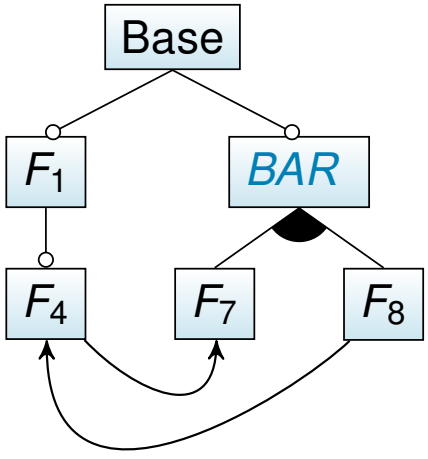


Δ_1

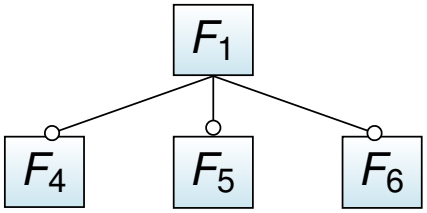


Δ_2

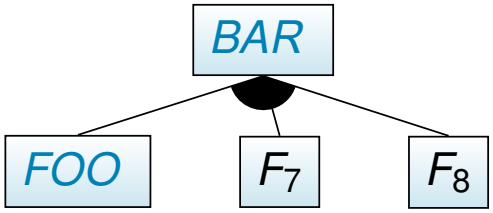
Compositionality with Feature-Model Interfaces



Γ



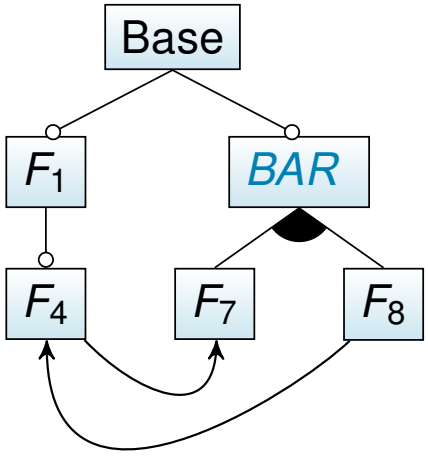
Δ_1



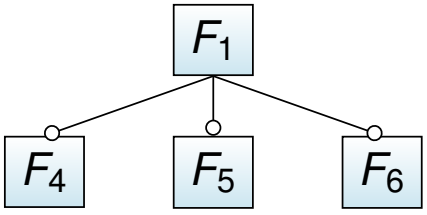
Δ_2

$FOO \Rightarrow BAR$?

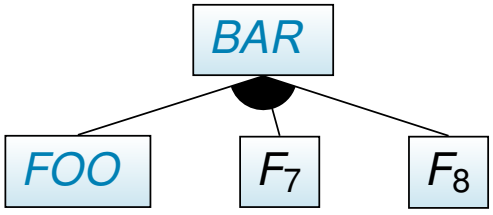
Compositionality with Feature-Model Interfaces



Γ



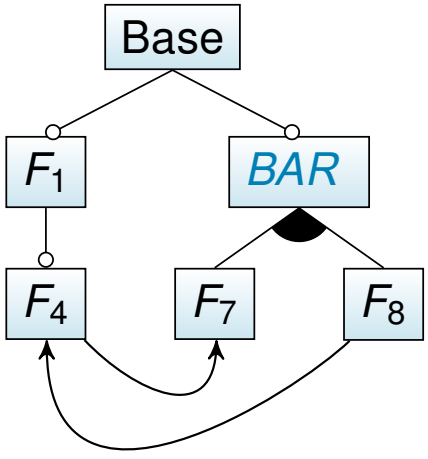
Δ_1



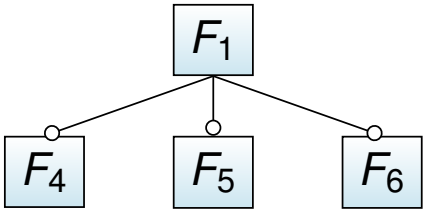
Δ_2

$$\Gamma \cup \Delta_2 \models FOO \Rightarrow BAR \; ?$$

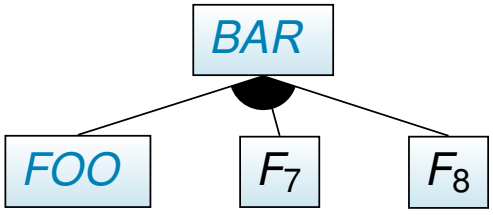
Compositionality with Feature-Model Interfaces



Γ



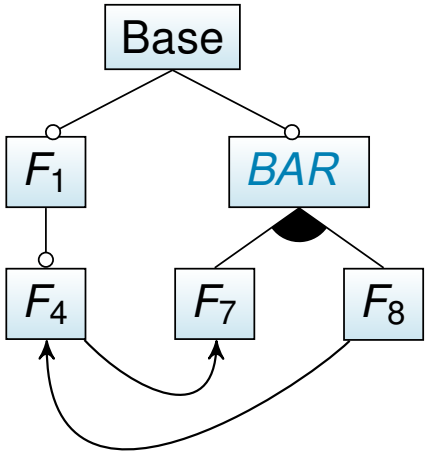
Δ_1



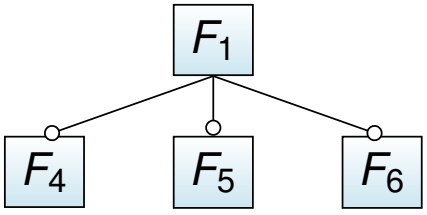
Δ_2

$$\Gamma \cup \Delta_2 \models FOO \Rightarrow BAR \quad \checkmark$$

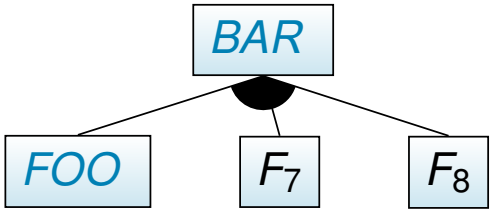
Compositionality with Feature-Model Interfaces



Γ



Δ_1

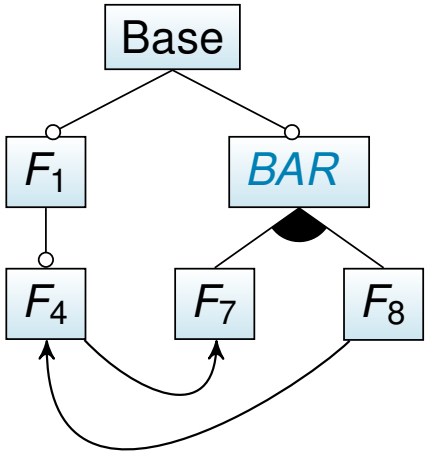


Δ_2

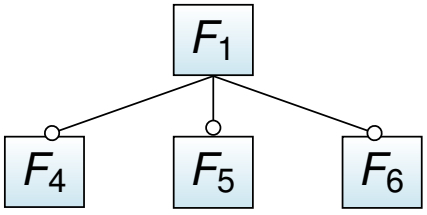
$\Gamma \cup \Delta_2 \models FOO \Rightarrow BAR$ ✓

$F_6 \Rightarrow Base$?

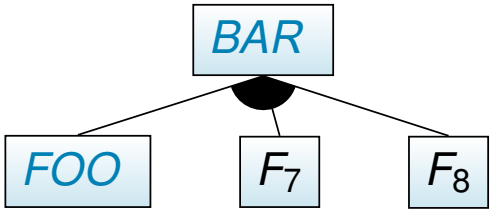
Compositionality with Feature-Model Interfaces



Γ



Δ_1

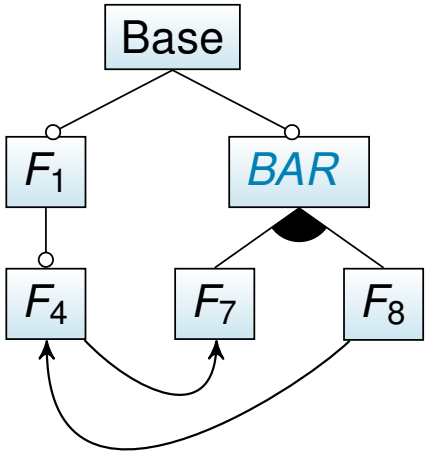


Δ_2

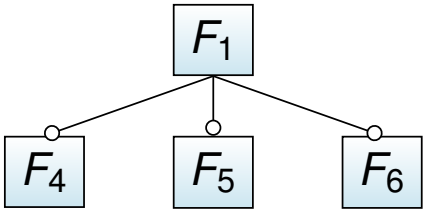
$\Gamma \cup \Delta_2 \models FOO \Rightarrow BAR$ ✓

$\Gamma \cup \Delta_1 \models F_6 \Rightarrow Base$?

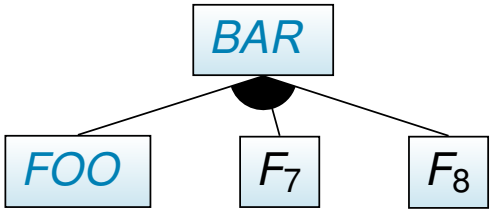
Compositionality with Feature-Model Interfaces



Γ



Δ_1

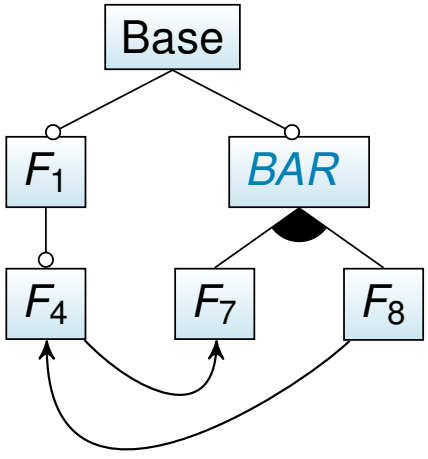


Δ_2

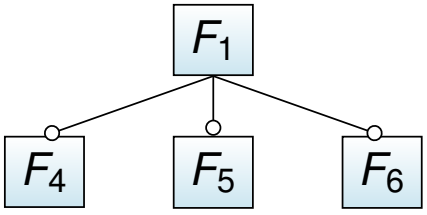
$\Gamma \cup \Delta_2 \models FOO \Rightarrow BAR$ ✓

$\Gamma \cup \Delta_1 \models F_6 \Rightarrow Base$ ✓

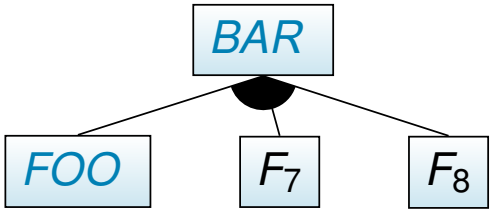
Compositionality with Feature-Model Interfaces



Γ



Δ_1



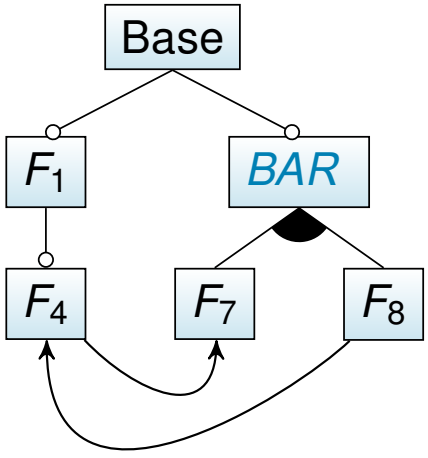
Δ_2

$\Gamma \cup \Delta_2 \models FOO \Rightarrow BAR$ ✓

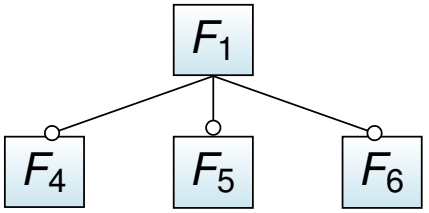
$\Gamma \cup \Delta_1 \models F_6 \Rightarrow Base$ ✓

$F_1 \wedge F_6 \wedge FOO$?

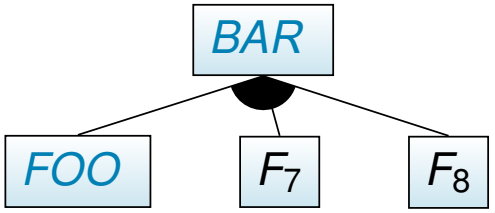
Compositionality with Feature-Model Interfaces



Γ



Δ_1



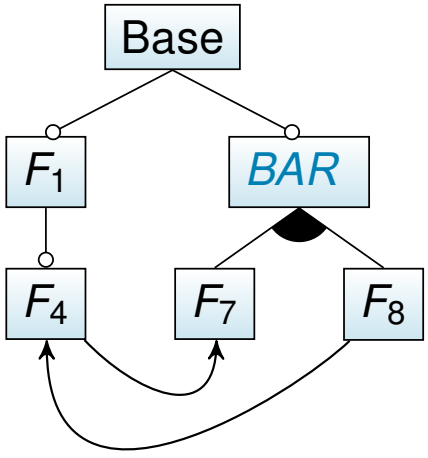
Δ_2

$$\Gamma \cup \Delta_2 \models FOO \Rightarrow BAR \quad \checkmark$$

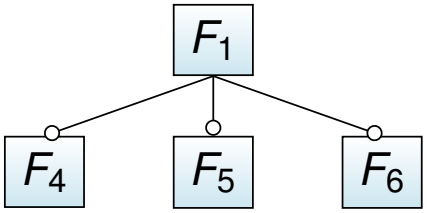
$$\Gamma \cup \Delta_1 \models F_6 \Rightarrow Base \quad \checkmark$$

$$\Gamma \cup \Delta_1 \cup \Delta_2 \models F_1 \wedge F_6 \wedge FOO \; ?$$

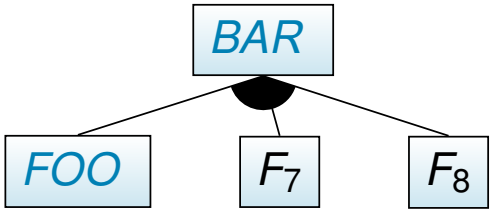
Compositionality with Feature-Model Interfaces



Γ



Δ_1



Δ_2

$$\Gamma \cup \Delta_2 \models FOO \Rightarrow BAR \quad \checkmark$$

$$\Gamma \cup \Delta_1 \models F_6 \Rightarrow Base \quad \checkmark$$

$$\Gamma \cup \Delta_1 \cup \Delta_2 \not\models F_1 \wedge F_6 \wedge FOO \quad \checkmark$$

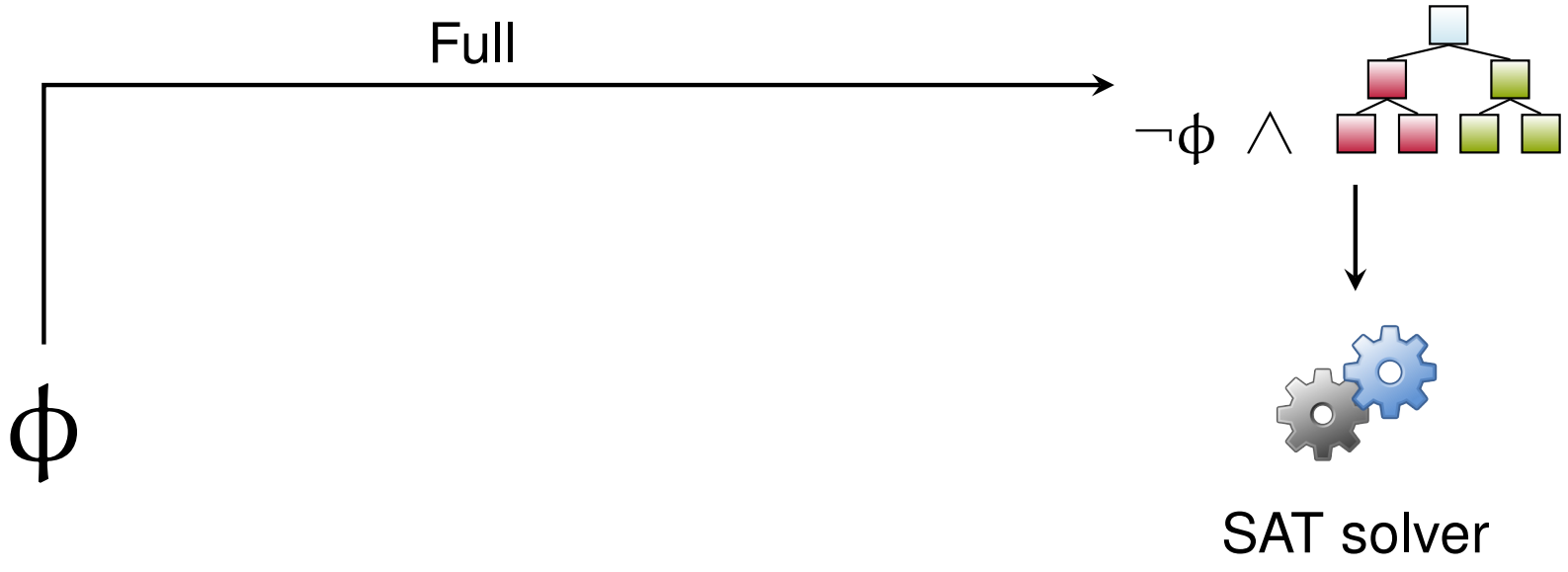
Reasoning Strategies

ϕ

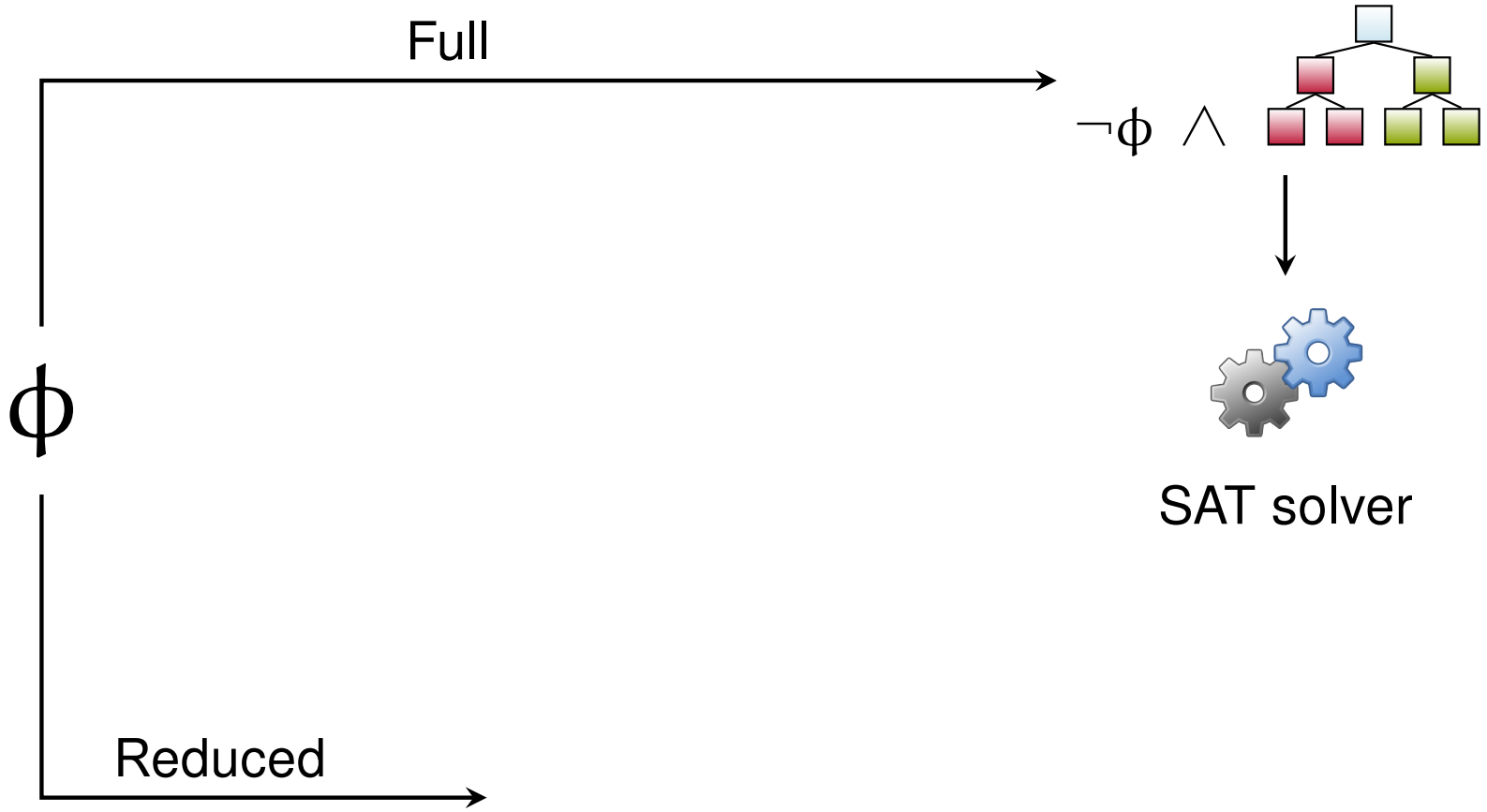
Reasoning Strategies

Reasoning Strategies

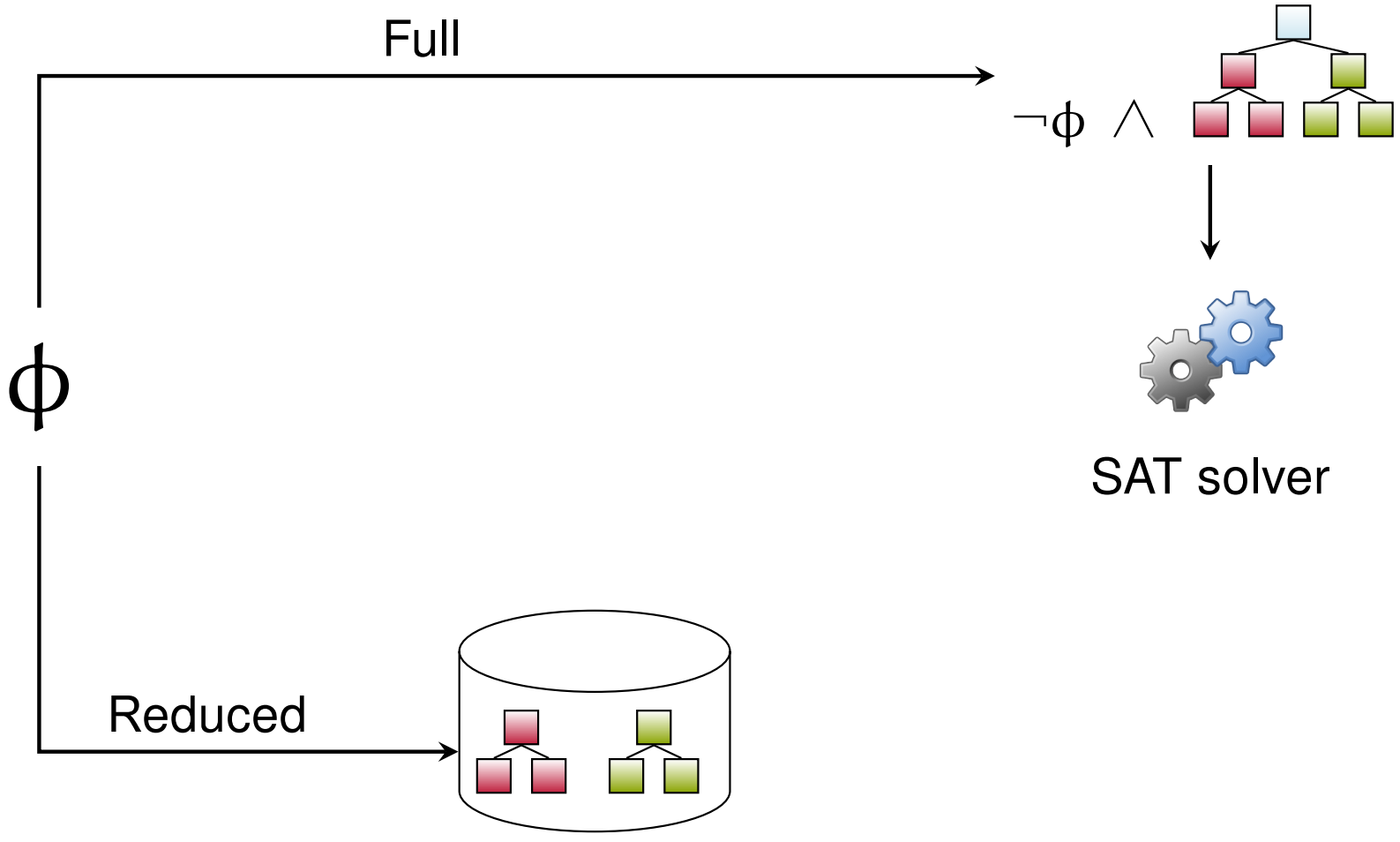
Reasoning Strategies



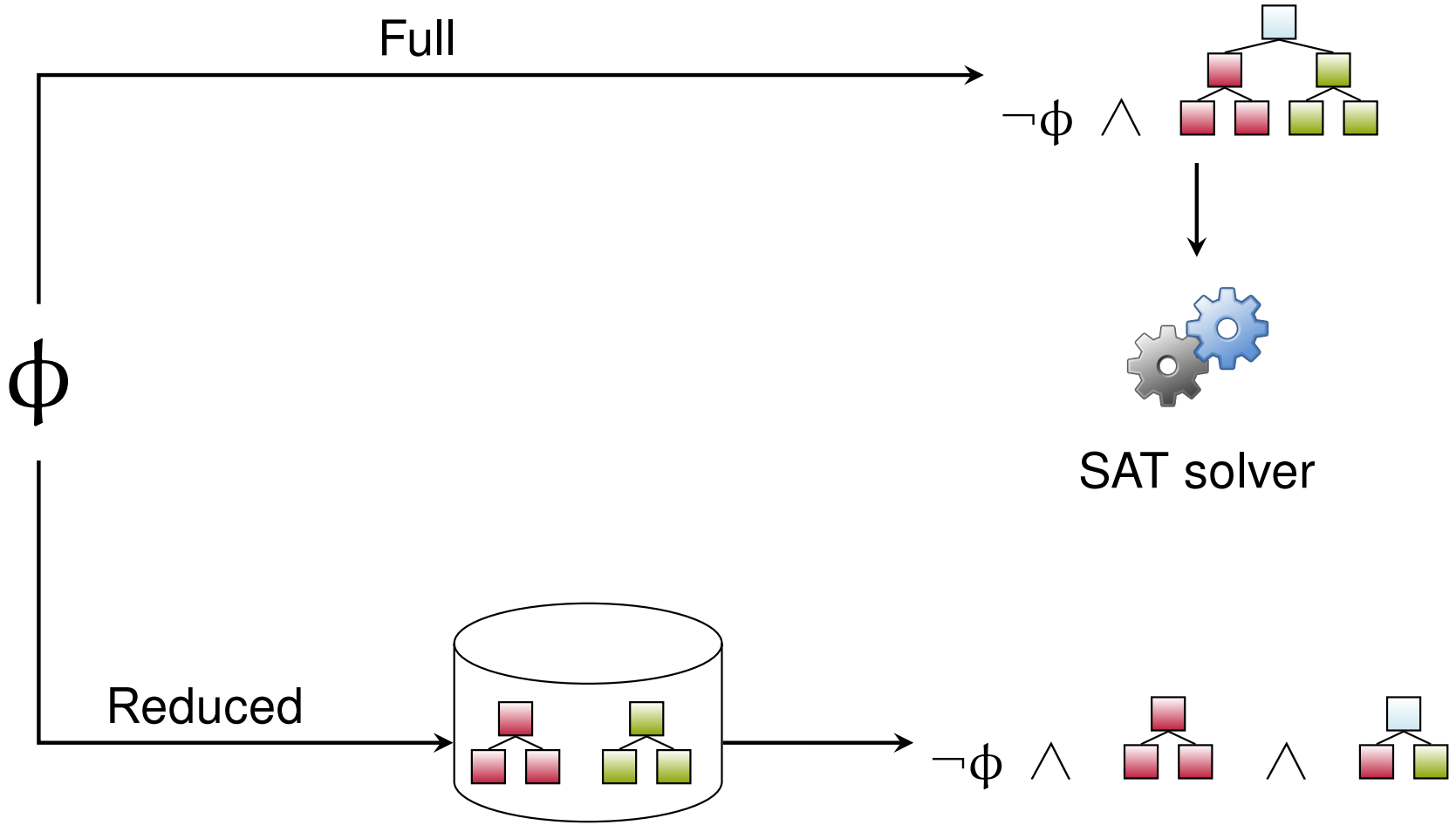
Reasoning Strategies



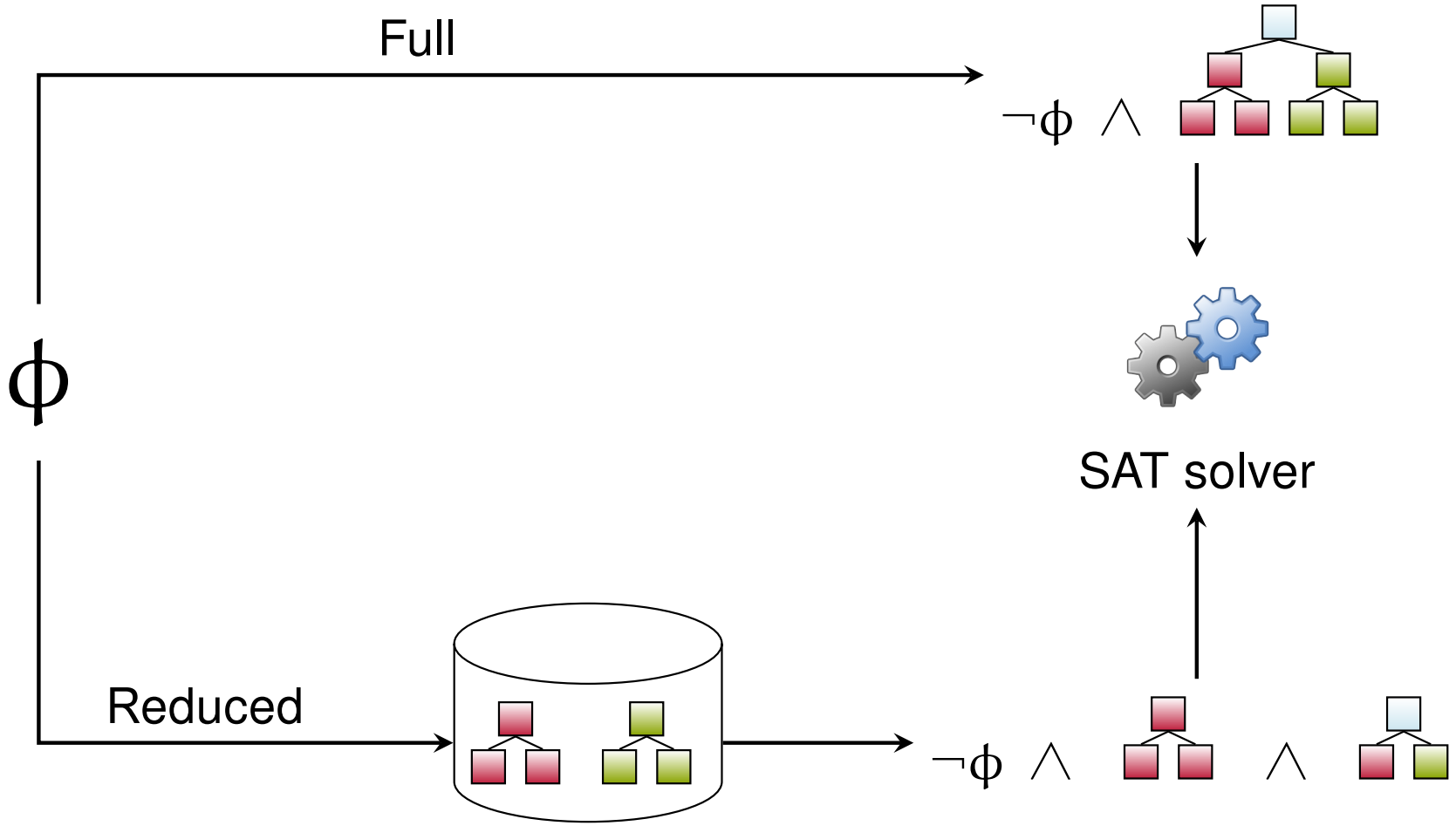
Reasoning Strategies



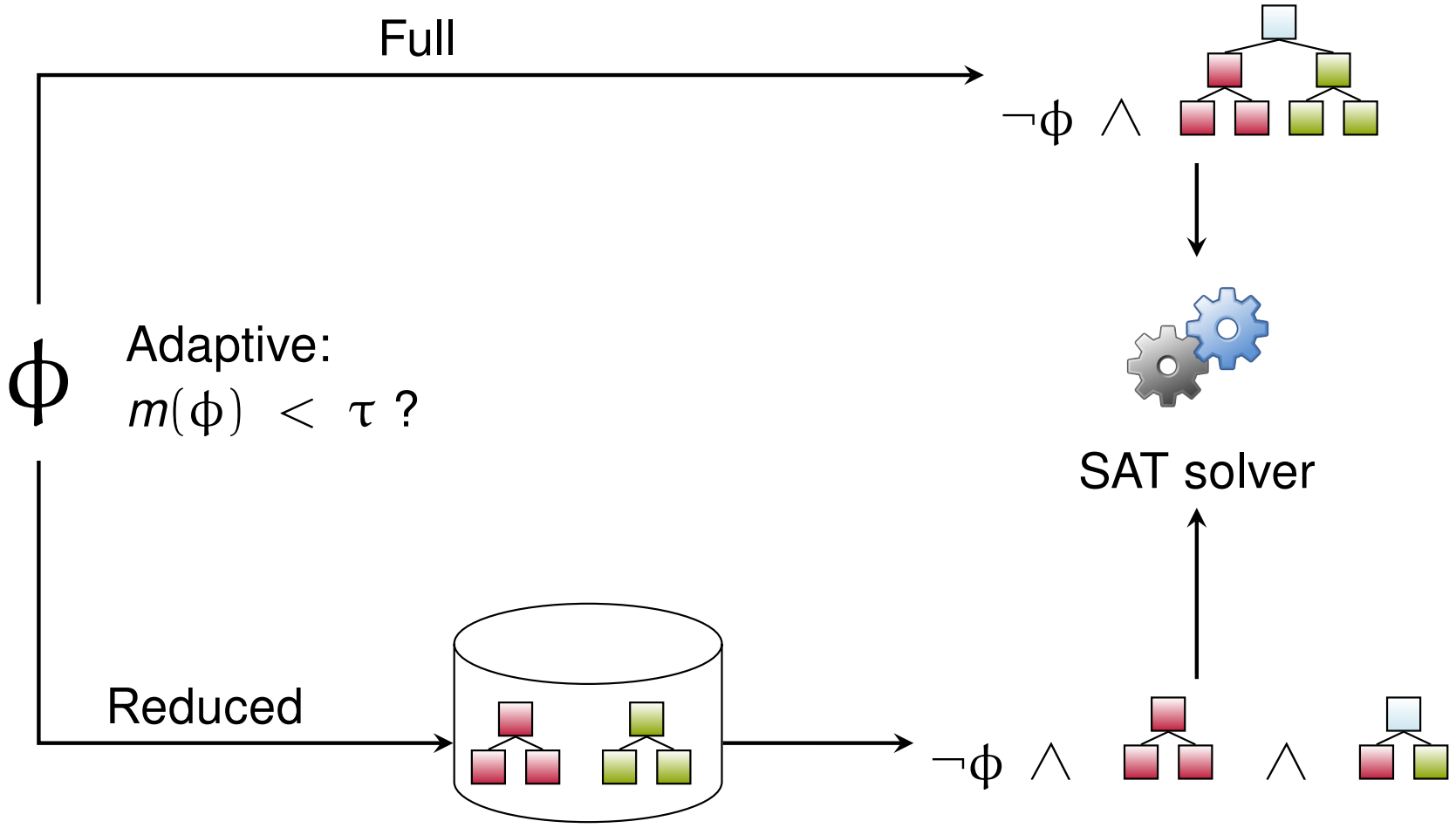
Reasoning Strategies



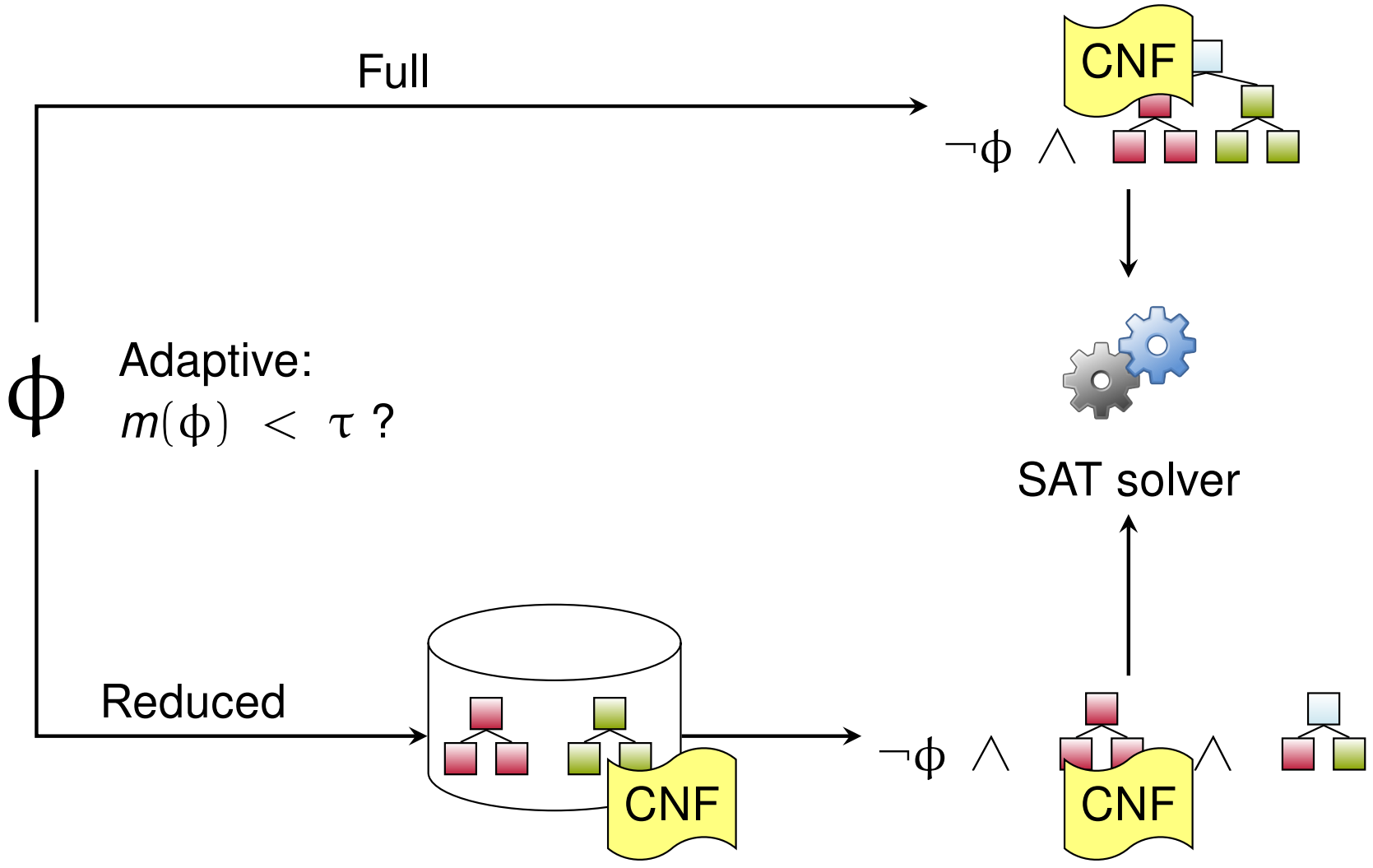
Reasoning Strategies



Reasoning Strategies

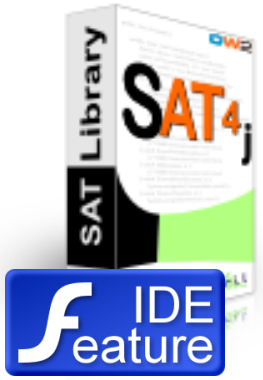


Reasoning Strategies



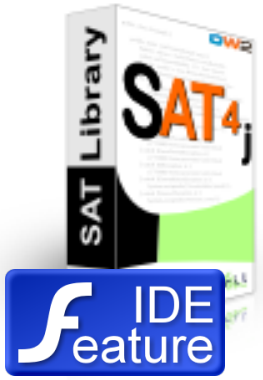
Setup for Measurements

- Generate sets of 1,000 and 5,000 queries
- Full: pure solver time
- Reduced: selection + composition + solver times



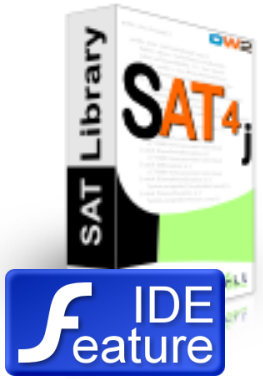
Setup for Measurements

- Generate sets of 1,000 and 5,000 queries
- Full: pure solver time
- Reduced: selection + composition + solver times
- JVM: garbage collection & just-in-time compilation



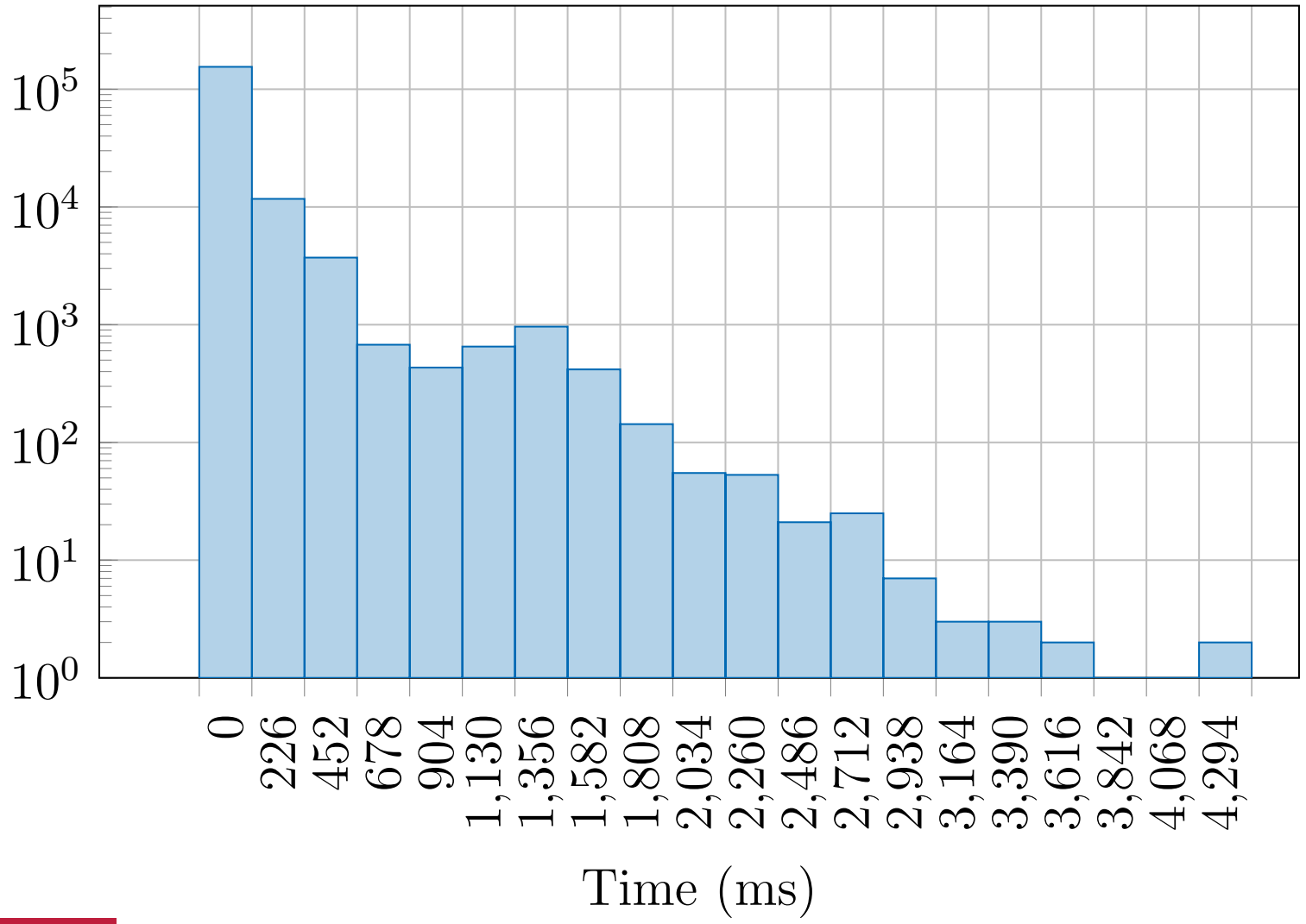
Setup for Measurements

- Generate sets of 1,000 and 5,000 queries
- Full: pure solver time
- Reduced: selection + composition + solver times

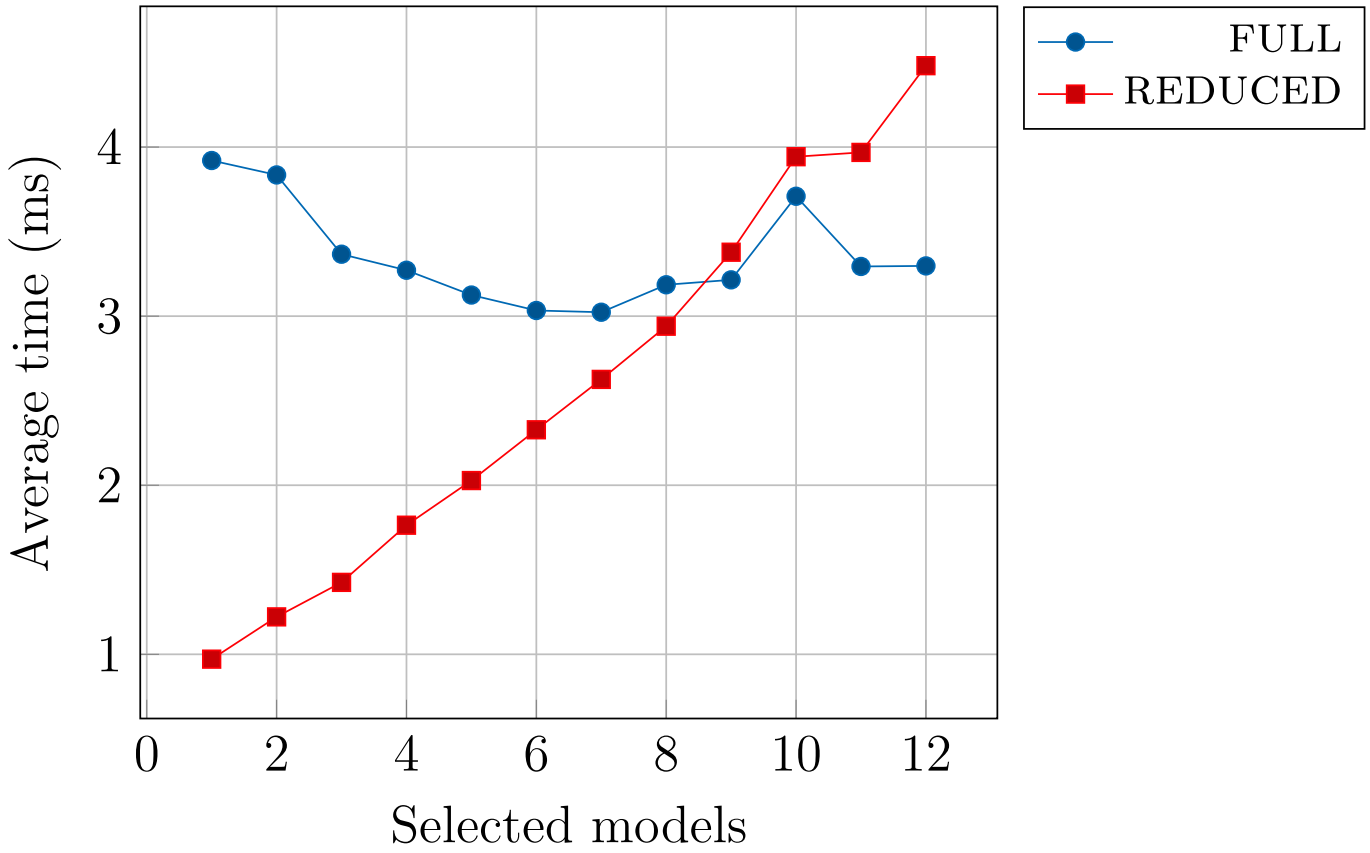


- JVM: garbage collection & just-in-time compilation
- ScalaMeter framework
  - Multiple measurements across different JVM runs
  - Warm-up runs
  - Outlier detection

Time to Solve SAT Queries



Time vs. Number of Selected Submodels



Threats to Validity

- Type checking
  - 1/6 of available files
  - Other analyses, product lines?

Threats to Validity

- Type checking
  - 1/6 of available files
  - Other analyses, product lines?
- Reasoning with interfaces
  - Semi-random queries
  - Only snapshots of a single model
  - Single solver implementation
  - Isolated measurements