Reverse Engineering Feature Models.

S. She, R. Lotufo, T. Berger, A. Wasowski, K. Czarnecki
Generative Software Development Lab
University of Waterloo
University of Leipzig
IT University of Copenhagen

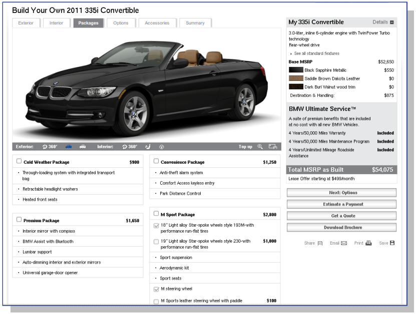
ICSE 2011. May 27, 2011.

What are feature models?

Feature models describe the common and variable characteristics of products in a product line.

What are feature models?

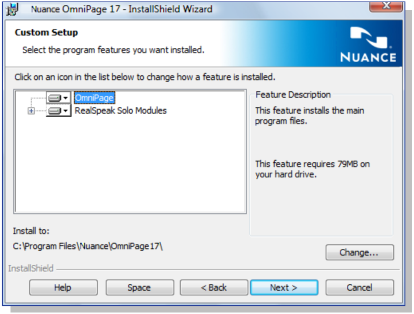
Feature models describe the common and variable characteristics of products in a product line.



Car configurator.

What are feature models?

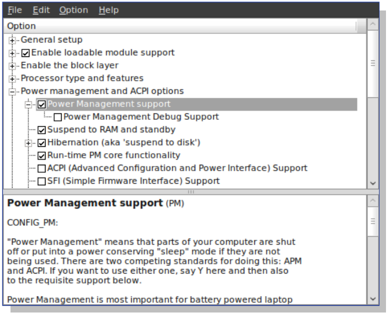
Feature models describe the common and variable characteristics of products in a product line.



Installation wizards.

What are feature models?

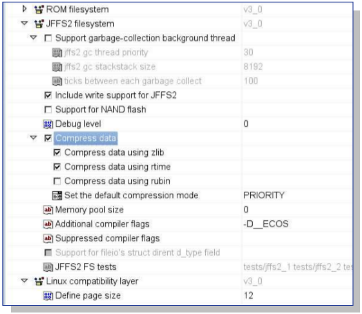
Feature models describe the common and variable characteristics of products in a product line.



Linux kernel configurator.

What are feature models?

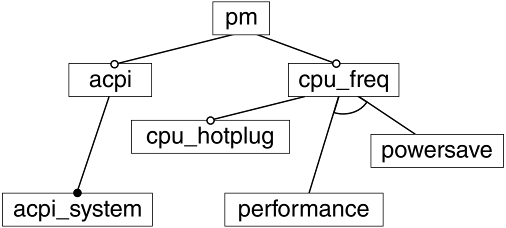
Feature models describe the common and variable characteristics of products in a product line.



eCos kernel configurator.

What are feature models?

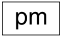
Feature models describe the common and variable characteristics of products in a product line.



$\text{powersave} \wedge \text{acpi} \rightarrow \text{cpu\_hotplug}$

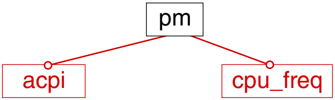
FODA feature model [Kang et al. 1990]

Feature model syntax.



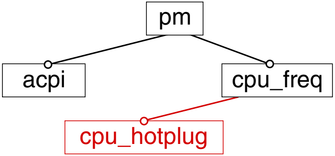
Root feature.

Feature model syntax.



Optional features.

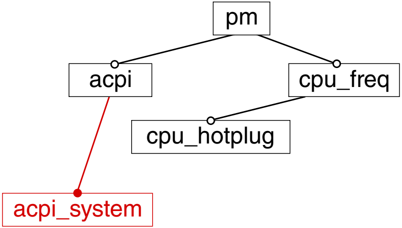
Feature model syntax.



Child features / feature hierarchy.

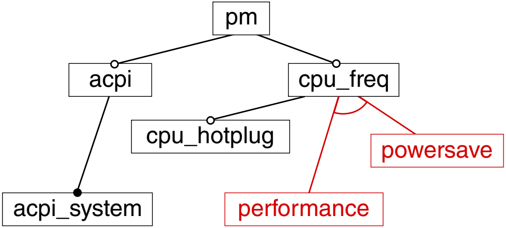
In feature models, *child* $\rightarrow$ *parent*

Feature model syntax.



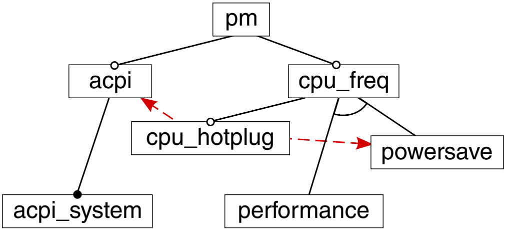
Mandatory feature.

Feature model syntax.



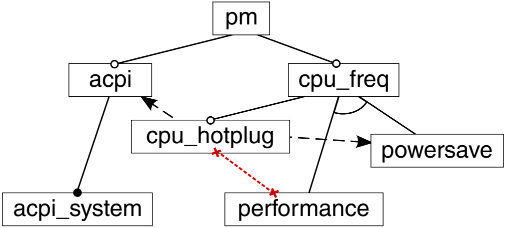
XOR-*group.*

Feature model syntax.



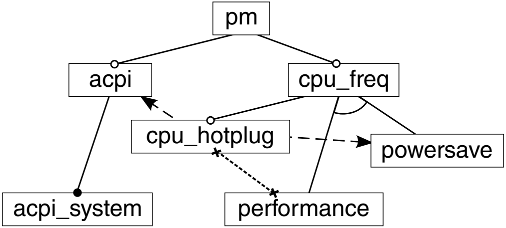
Implies edges.

Feature model syntax.



Excludes edges.

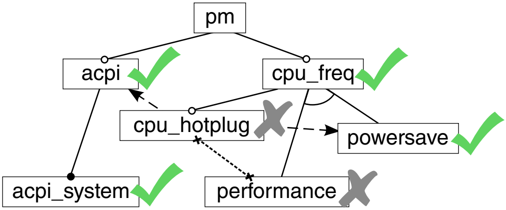
Feature model syntax.



$$\textbf{powersave} \wedge \textbf{acpi} \rightarrow \textbf{cpu\_hotplug}$$

Additional cross-tree constraints.

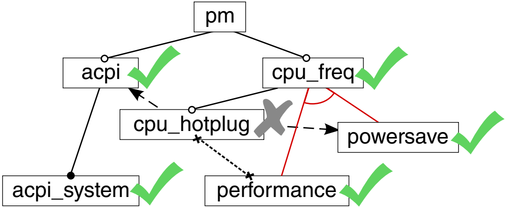
Legal configurations.



{ pm, acpi, acpi_system, cpu_freq, powersave }

Valid Configuration.

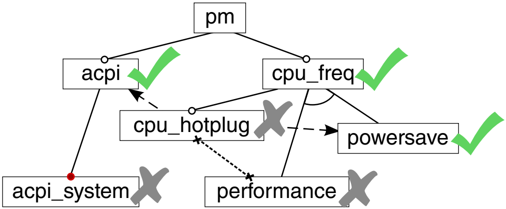
Legal configurations.



{ pm, acpi, acpi_system, cpu_freq, powersave, performance }

Invalid Configuration: violates XOR-group.

Legal configurations.



{ pm, acpi, cpu_freq, powersave }

Invalid Configuration: violates mandatory feature.

Why reverse-engineer a feature model?

- Many product lines manage variability in an ad-hoc manner.
  e.g. FreeBSD, vim, Mplayer, etc.
- For these systems, features and dependencies are hidden in documentation, code and build system.
- Feature models make features and dependencies explicit.
- Feature models are well-understood with tool support (e.g. configurators) and automated analysis.

FreeBSD.

Configuring FreeBSD:

```
# IPI_PREEMPTION relies on the PREEMPTION option

# Mandatory:
Device apic                # I/O apic

# Optional:
options MPTABLE_FORCE_HTT #enable HTT CPUs ...
options IPI_PREEMPTION
```

Code:

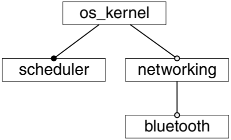
```
MODULE_DEPEND(at91_twi, iicbus, …);
#ifdef A … #endif
```

Features and dependencies are scattered through code and documentation.

Reverse-engineering steps.

```
#ifdef A
   #ifndef B
       #error ...
    #endif
#endif
```

scheduler $\leftrightarrow$ os_kernel
networking $\rightarrow$ os_kernel
bluetooth $\rightarrow$ networking



bluetooth is a network driver.

Descriptions

Codebase

Feature names

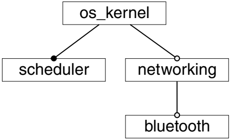
Feature Model

Dependencies

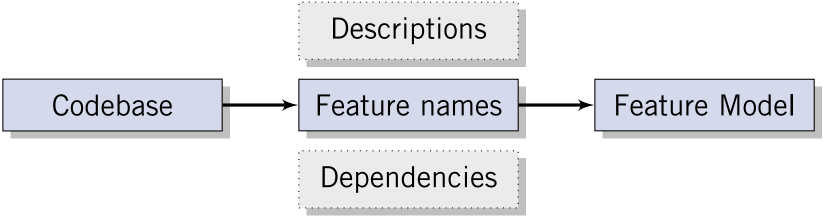
Reverse-engineering steps.

```
#ifdef A
   #ifndef B
       #error ...
    #endif
#endif
```

scheduler $\leftrightarrow$ os_kernel
networking $\rightarrow$ os_kernel
bluetooth $\rightarrow$ networking



bluetooth is a network driver.



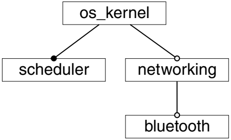
Feature names are needed to build a feature model.

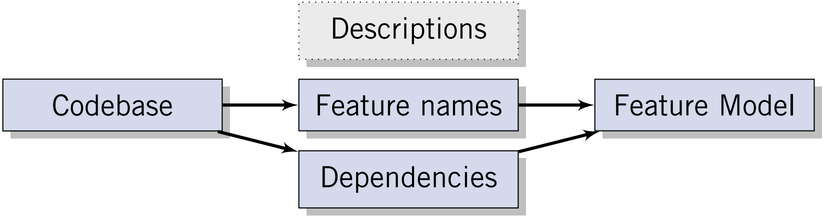
Reverse-engineering steps.

```
#ifdef A
   #ifndef B
       #error ...
    #endif
#endif
```

scheduler $\leftrightarrow$ os_kernel
networking $\rightarrow$ os_kernel
bluetooth $\rightarrow$ networking

bluetooth is a network driver.





Let's first try to reverse-engineer a feature model using just names and dependencies.

Using just names and dependencies.

Given these features:

$$\{\text{os\_kernel}, \text{scheduler}, \text{networking}, \text{bluetooth}\}$$

…and these dependencies:

1. $(\text{bluetooth} \vee \text{networking} \vee \text{scheduler} \rightarrow \text{os\_kernel})$
2. $\wedge\ (\text{os\_kernel} \rightarrow \text{scheduler})$
3. $\wedge\ (\text{bluetooth} \rightarrow \text{networking})$

- What are the legal configurations of features?
- What is the feature model that describes these legal configurations?

Using just names and dependencies.

Given these features:

$$\{\text{os\_kernel}, \text{scheduler}, \text{networking}, \text{bluetooth}\}$$

...and these dependencies:

1. $(\text{bluetooth} \vee \text{networking} \vee \text{scheduler} \rightarrow \text{os\_kernel})$
2. $\wedge\ (\text{os\_kernel} \rightarrow \text{scheduler})$
3. $\wedge\ (\text{bluetooth} \rightarrow \text{networking})$

- What are the legal configurations of features?
- What is the feature model that describes these legal configurations?

Using just names and dependencies.

Given these features:

$$\{\text{os\_kernel}, \text{scheduler}, \text{networking}, \text{bluetooth}\}$$

...and these dependencies:

1. $(\text{bluetooth} \vee \text{networking} \vee \text{scheduler} \rightarrow \text{os\_kernel})$
2. $\wedge\ (\text{os\_kernel} \rightarrow \text{scheduler})$
3. $\wedge\ (\text{bluetooth} \rightarrow \text{networking})$

- What are the legal configurations of features?
- What is the feature model that describes these legal configurations?

Using just names and dependencies.

Given these features:

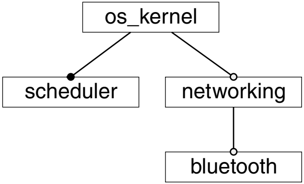
$$\{\text{os\_kernel}, \text{scheduler}, \text{networking}, \text{bluetooth}\}$$

...and these dependencies:

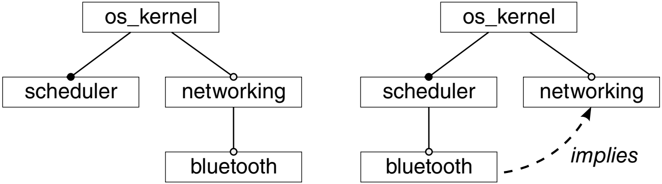
1. $(\text{bluetooth} \vee \text{networking} \vee \text{scheduler} \rightarrow \text{os\_kernel})$
2. $\wedge\ (\text{os\_kernel} \rightarrow \text{scheduler})$
3. $\wedge\ (\text{bluetooth} \rightarrow \text{networking})$

- What are the legal configurations of features?
- What is the feature model that describes these legal configurations?

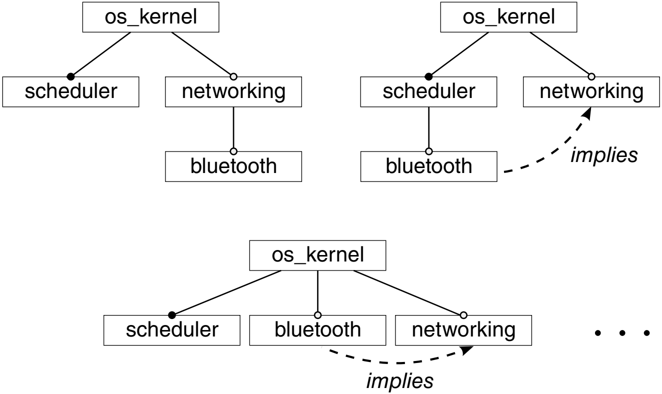
Many possible models.



Many possible models.



Many possible models.

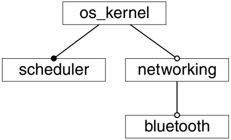


- All these models are refactorings.
- All describe the same features and dependencies.
- We need domain knowledge to identify the best model.

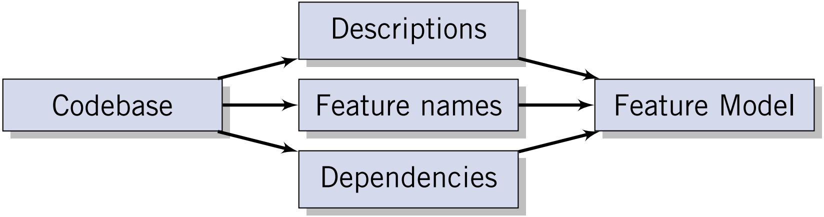
Reverse-engineering steps.

```
#ifdef A
   #ifndef B
       #error ...
    #endif
#endif
```

scheduler $\leftrightarrow$ os_kernel
networking $\rightarrow$ os_kernel
bluetooth $\rightarrow$ networking

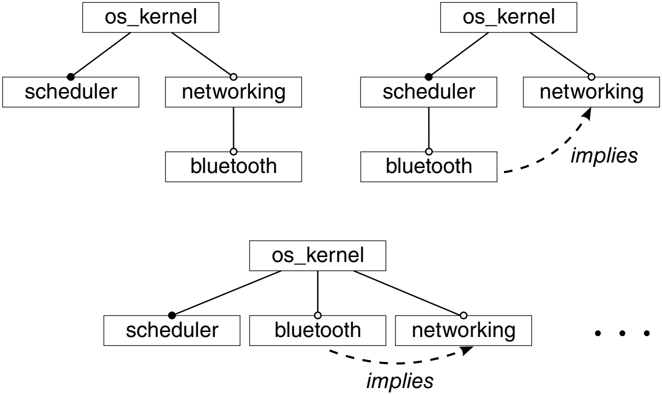


bluetooth is a network driver.



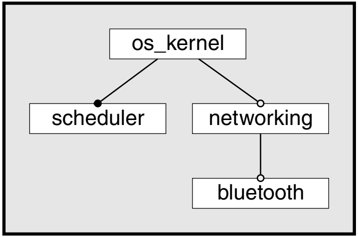
Leverage both names and descriptions for additional domain knowledge.

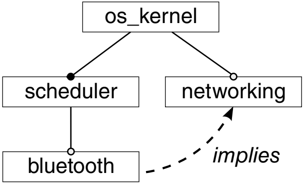
Many possible models.

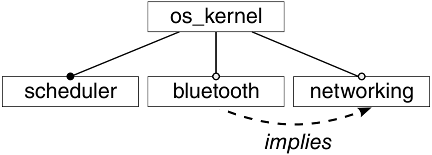


bluetooth is a network driver.

Many possible models.







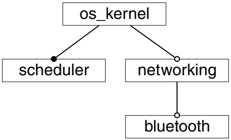
. . .

bluetooth is a network driver.

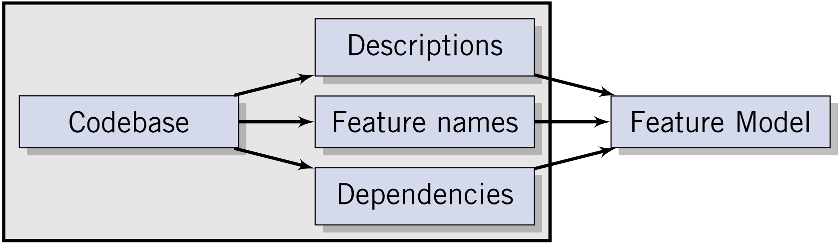
Reverse-engineering steps.

```
#ifdef A
   #ifdef B
       #error ...
    #endif
#endif
```

$\text{scheduler} \leftrightarrow \text{os\_kernel}$
$\text{networking} \rightarrow \text{os\_kernel}$
$\text{bluetooth} \rightarrow \text{networking}$



bluetooth is a network driver.

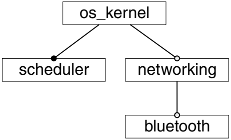


We rely on existing and ongoing work to extract necessary input from code and documentation. [Berger et al. 2010]

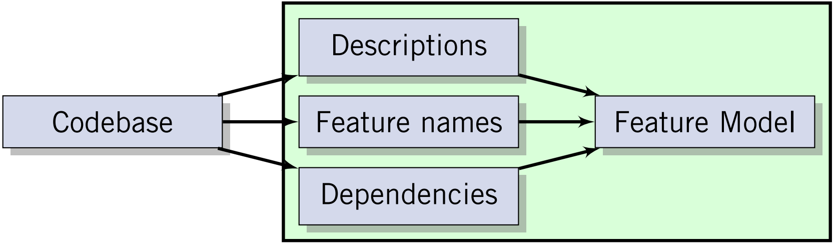
Reverse-engineering steps.

```
#ifdef A
   #ifndef B
      #error ...
   #endif
#endif
```

scheduler $\leftrightarrow$ os_kernel
networking $\rightarrow$ os_kernel
bluetooth $\rightarrow$ networking



bluetooth is a network driver.



This work uses feature names, descriptions and dependencies to build a feature model.

Goal.

Provide support for reverse-engineering a large-scale feature model from existing project artifacts.

- A project (e.g. FreeBSD) could benefit from a FM for configuration and analysis.
- Many possible FMs describe the same features and dependencies—exponential!
- Our work provides assistance for building feature hierarchy by significantly reducing choices for the model builder.
- Other FM elements, such as groups, are detected automatically.

Outline.

1 Introduction

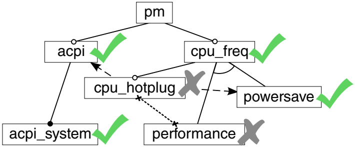
2 Procedures

3 Evaluation

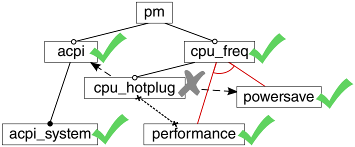
4 Conclusions

Configuration semantics.

The configuration semantics of a feature model is a set of legal configurations.



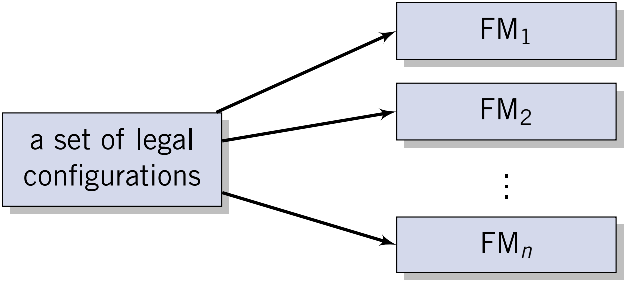
Legal configuration.



Illegal configuration.

Reverse-engineering.

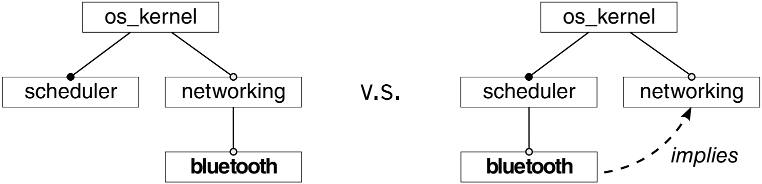
A set of legal configurations can be represented by many *possible feature models.*



- The configuration semantics alone are not enough to identify a unique FM.

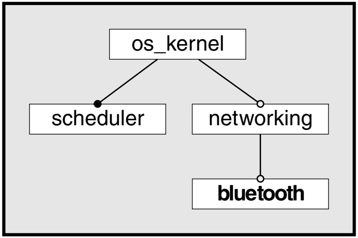
Domain semantics.

The domain semantics are the meaning of the features and are reflected in the names and hierarchy.

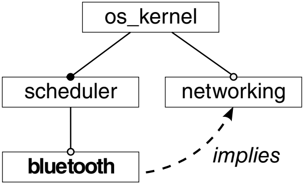


Domain semantics.

The domain semantics are the meaning of the features and are reflected in the names and hierarchy.

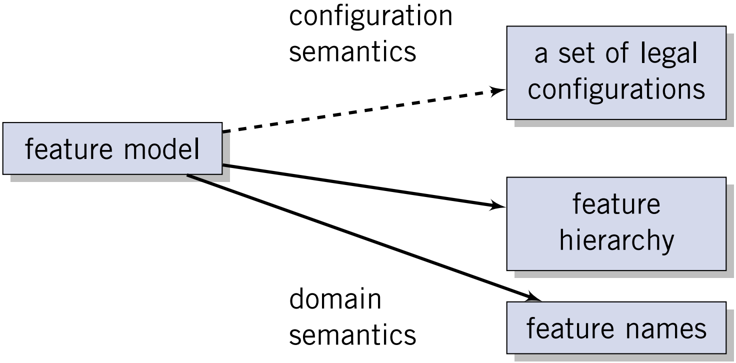


v.s.



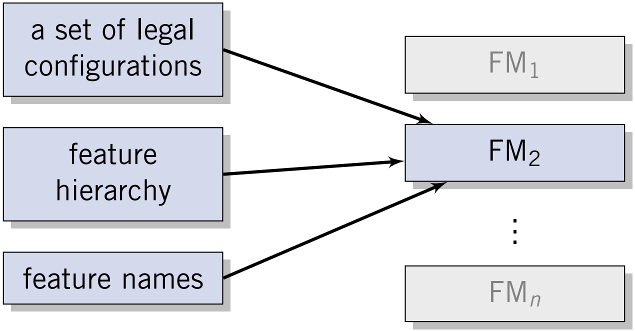
bluetooth is a network driver.

Domain semantics (cont.)

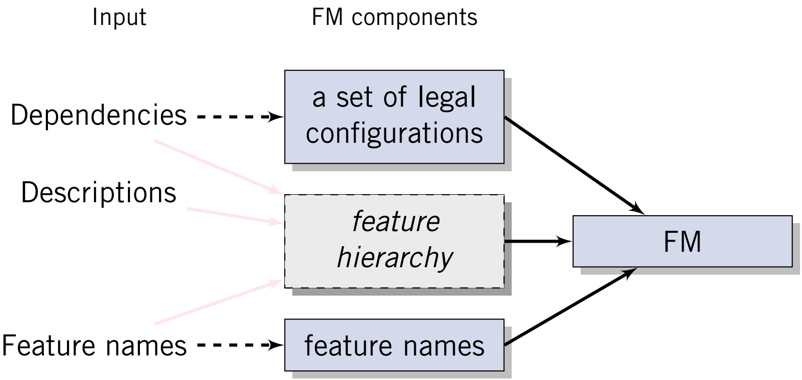


Reverse-engineering II.

Given a set of legal configurations, feature names and a hierarchy, a precise FM can be reverse-engineered.

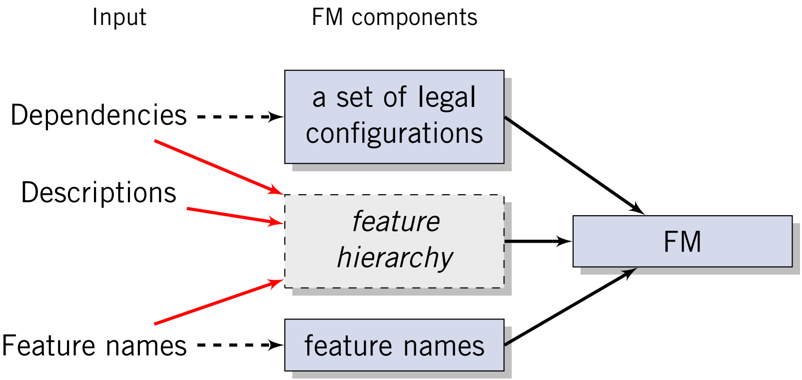


Reverse-engineering II (cont.)



- When reverse-engineering a FM, the feature hierarchy doesn't exist yet.

Reverse-engineering II (cont.)



- We can build the feature hierarchy using dependencies, names and descriptions.

Building the feature hierarchy.

1. Determine a parent for every feature:
   - We use the names and descriptions to propose a hierarchy that reflects domain semantics.
   - An interactive, tool-assisted procedure.
   - Given a feature, rank choices for its parent by similarity.
2. A child must imply its parent:
   - The meaning of the hierarchy in a feature model.
   - Generate an implication graph from dependencies.

Building the feature hierarchy.

1. Determine a parent for every feature:
    - We use the names and descriptions to propose a hierarchy that reflects domain semantics.
    - An interactive, tool-assisted procedure.
    - Given a feature, rank choices for its parent by similarity.

2. A child must imply its parent:
    - The meaning of the hierarchy in a feature model.
    - Generate an implication graph from dependencies.

Feature similarity.

Feature names and descriptions	
os_kernel	Operating system.
scheduler	I/O scheduling.
networking	Networking drivers.
ethernet	Type of local area networking.

Selecting a parent for:

bluetooth, a network driver.

Feature similarity.

Feature names and descriptions

os_kernel	Operating system.
scheduler	I/O scheduling.
networking	Networking drivers.
ethernet	Type of local area networking.

Selecting a parent for:

bluetooth, a network driver.

Feature similarity.

Feature names and descriptions

os_kernel	Operating system.
scheduler	I/O scheduling.
networking	**Networking** drivers.
ethernet	Type of local area **networking**.

Selecting a parent for:

bluetooth, a **network** driver.

Feature similarity.

Feature names and descriptions

os_kernel	Operating system.
scheduler	I/O scheduling.
networking	Networking **drivers**.
ethernet	Type of local area networking.

Selecting a parent for:

bluetooth, a network **driver**.

Feature similarity.

Feature names and descriptions

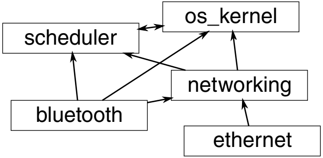
1.	networking	Networking drivers.
2.	ethernet	Type of local area networking.
3.	os_kernel	Operating system.
4.	scheduler	I/O scheduling.

Selecting a parent for:

bluetooth, a network driver.

Implication graph.

A child must imply its parent in the feature hierarchy.



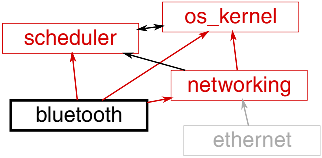
Selecting a parent for:

bluetooth, a network driver.

- ethernet is not shown—not a choice for parent.
- Implications significantly reduce the number of choices.
- But, in a practical setting, dependencies may be incomplete.

Implication graph.

A child must imply its parent in the feature hierarchy.



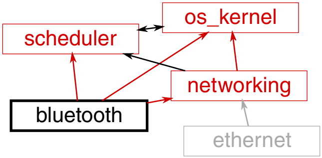
Selecting a parent for:

bluetooth, a network driver.

- ethernet is not shown—not a choice for parent.
- Implications significantly reduce the number of choices.
- But, in a practical setting, dependencies may be incomplete.

Implication graph.

A child must imply its parent in the feature hierarchy.



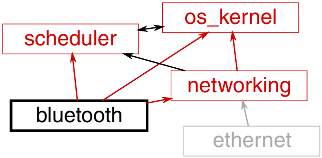
Selecting a parent for:

bluetooth, a network driver.

- ethernet is not shown—not a choice for parent.
- Implications significantly reduce the number of choices.
- But, in a practical setting, dependencies may be incomplete.

Implication graph.

A child must imply its parent in the feature hierarchy.

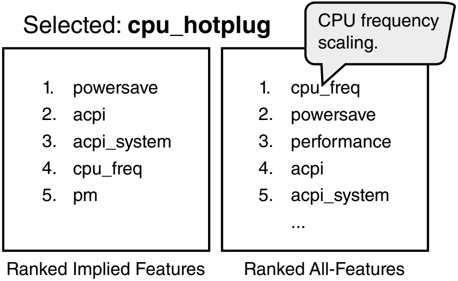


Selecting a parent for:

bluetooth, a network driver.

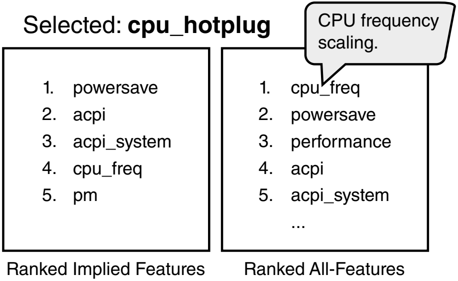
- ethernet is not shown—not a choice for parent.
- Implications significantly reduce the number of choices.
- But, in a practical setting, dependencies may be incomplete.

Two lists: RIFs and RAFs.



- **Ranked Implied Features (RIFs)**
  implied features sorted by similarity to the selected feature.
- **Ranked All-Features (RAFs)**
  all features sorted by similarity to the selected feature.

Two lists: RIFs and RAFs.

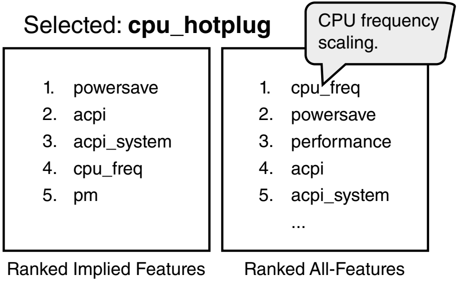


- **Ranked Implied Features (RIFs)**

  implied features sorted by similarity to the selected feature.
- **Ranked All-Features (RAFs)**

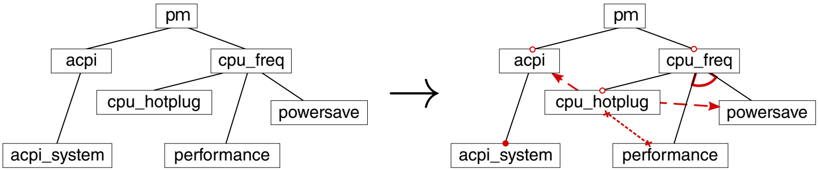
  all features sorted by similarity to the selected feature.

Two lists: RIFs and RAFs.



- **Ranked Implied Features (RIFs)**
  implied features sorted by similarity to the selected feature.
- **Ranked All-Features (RAFs)**
  all features sorted by similarity to the selected feature.

Other FM constructs.



- User selects a parent for every feature.
- Once a hierarchy is built, other constructs, such as mandatory features and groups, are automatically detected.
- If feature groups overlap, user selects groups to retain.

Outline.

1 Introduction

2 Procedures

3 Evaluation

4 Conclusions

Evaluation questions.

Our similarity measure should reduce the number of choices to only a few when determining a parent for a feature.

1. How many features have their reference parents ranked in the top 5 of our RIFs?
   - Evaluated on complete and incomplete input.

2. How many features are needed for finding 75% of reference parents using the RAFs?

Evaluation questions.

Our similarity measure should reduce the number of choices to only a few when determining a parent for a feature.

1. How many features have their reference parents ranked in the top 5 of our RIFs?
   - Evaluated on complete and incomplete input.

2. How many features are needed for finding 75% of reference parents using the RAFs?

Evaluation questions.

Our similarity measure should reduce the number of choices to only a few when determining a parent for a feature.

1. How many features have their reference parents ranked in the top 5 of our RIFs?
   - Evaluated on complete and incomplete input.

2. How many features are needed for finding 75% of reference parents using the RAFs?

Evaluation subjects.

Complete input:

- Reference feature models: Linux and eCos.
- Linux has roughly 5000 features; eCos 1200 features.

Incomplete input:

- A portion of FreeBSD.
- Domain analysis to create reference model of 90 features.
- Extracted input dependencies by analyzing preprocessor usage, documentation, etc.
- Simulated incomplete input on Linux and eCos by randomly removing dependencies and words.

Evaluation results for RIFs.

1. How many features have their reference parents ranked in the top 5 of our RIFs?
   - Linux: 76% of features, eCos: 79% of features.
   - Ignoring root features, 90% for Linux and 81% for eCos.
   - For incomplete descriptions, At least 50% of words needed for good results (roughly 10 words in Linux).

Evaluation results for RAFs.

2. How many features are needed for finding 75% of reference parents using the RAFs?
   - Linux: 3% of features, eCos: 6% of features.
   - For incomplete descriptions, 50% of words needed for good results.

More details in paper.

Outline.

Related work.

- Past work looked at only dependencies and didn't deal with multiple possible models.

  [CW 2007]

- Other works have applied textual similarity metrics, but don't take dependencies into account.

  [Alves et al. 2008, Niu et al. 2008]

- Past work attempts to create models automatically and not interactively.

Conclusions.

Future Work.

- Further develop the extraction of dependencies from a codebase.
- Integrate techniques into an existing FM editor.

Conclusions.

- Our procedure deals with incomplete input.
- Combine the use of dependencies and textual similarity.
- Problem requires domain knowledge—tool-assisted.
- Provide empirical data on how effective this technique is on three projects: Linux, eCos and FreeBSD.