Variability and Component Composition

Tijs van der Storm

storm@cwi.nl

CWI

Introduction

About myself:

- Master's thesis: *Implementing Actions* (April 2003)
- Phd student since July 2003 in DELIVER:

  Intelligent Software Knowledge Management and Delivery

- Focus on *Product Software* (ERP, CRM etc.) and the ASF+SDF Meta-Environment.

Started with:

- Dependency analysis of packages
- Checking of configuration switches

Generalization of this is the topic of this talk.

Overview

- Component variability
  - The need for configuration
  - Complications
- Description of components with variability
  - Component interfaces
  - Component Description Language
- Consistency requirements
- Implementation of checking configurations
  - Illustration with `Car`
  - Example results
- Final remarks & Future work

CWI

Component Variability

Traditionally:

- Variability at the level of *product families*
- Components as units of variation

But:

- Not all variability can be factored in component units
- This variability is exhibited by components themselves
- *Component populations*: every component = product

$\Rightarrow$ Feature Diagrams on component level.

The Need for Configuration

Two types of component clients:

- Customers that *acquire* a component
- Components that *depend* on another component

Configuration is needed:

- when configuring a product
- when composing components

Complications

But:

- Feature configuration: exponential complexity
- Components may require other components
- Composition may become variable itself

Need to control interaction of:

- dependencies
- feature diagrams

Choice to be made about ‘degree’ of variability.

CWI

Component Interfaces

Component Interfaces:

- Provided interface:
  - feature diagram (FDL)
- Required interface:
  - simple dependencies
  - guarded dependencies

Dependencies address *variants* of components.

NB: guards allow components to be units of variation.

Component Description Language (CDL)

component interface ⟨“meta-environment”, “1.6”⟩
  provides
    Meta : **all**(Rewriter, Type)
    Type : **more-of**(batch, studio)
    Rewriter : **one-of**(asf, elan)
  requires
    when asf {
      ⟨“asf”, “1.5”⟩
      ⟨“sdf”, “3.0”⟩
    }
    when elan {
      ⟨“elan”, “4.5”⟩
      ⟨“sdf”, “3.0”⟩ **with** strategies
    }

Consistency of Configuration

Consistency goals:

1. Internal consistency of FDL description
2. Actual configuration:
   - Consistent feature selection by customer
   - Passing of consistent features to dependencies

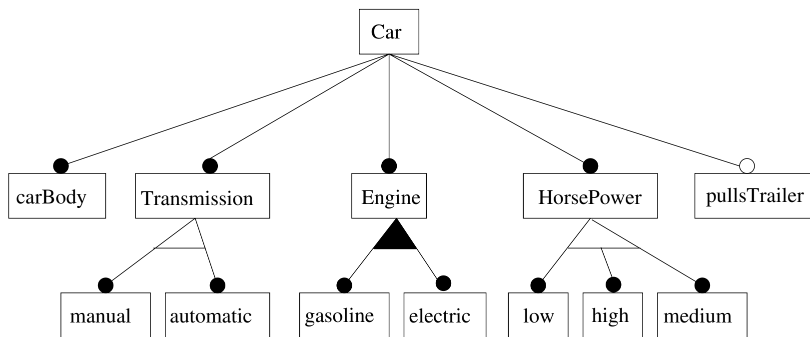
Ad 1. Translate FDL to Binary Decision Diagram (BDD)

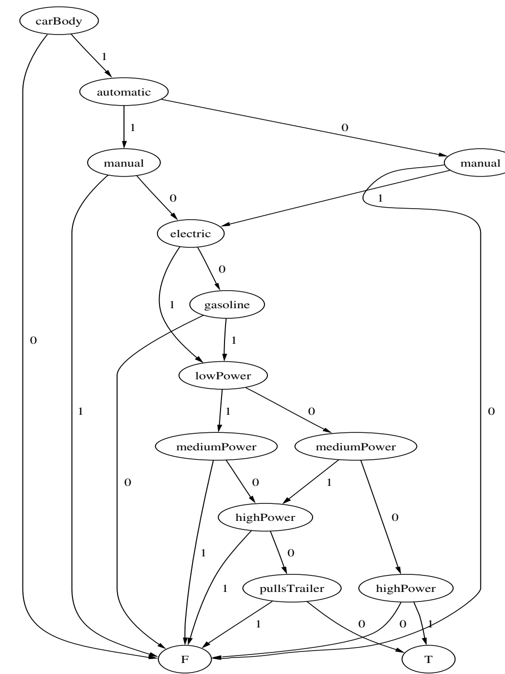
Ad 2. Use BDD for verification of feature selections

Implementation

Translate component interfaces to Relations.

FDL to Relations via Binary Decision Diagram:





Verification of feature selections formulated as *queries*.

Example: FDL for a Car

```
Car:              all(carBody,Transmission,
                      Engine,HorsePower,
                      pullsTrailer?)
Transmission: one-of(automatic, manual)
Engine:           more-of(electric, gasoline)
HorsePower:    one-of(lowPower,
                          mediumPower,highPower)

pullsTrailer requires highPower
```

FDL to Proposition

Inline FDL (without constraints):

```
all(carBody,one-of(manual,automatic),
   more-of(electric,gasoline),
   one-of(highPower,mediumPower,lowPower),
   pullsTrailer?)
```

Transform to Proposition (with constraints):

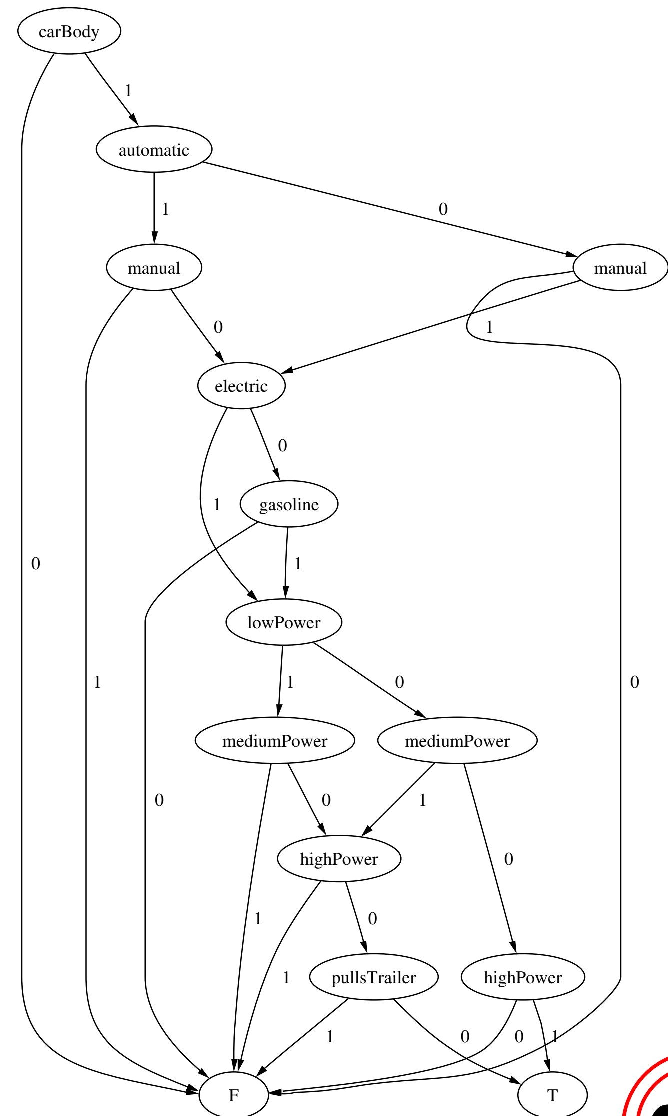
$$carBody \wedge \bigoplus\{manual, automatic\} \wedge$$
$$(electric \vee gasoline) \wedge \bigoplus\{highpower, mediumPower\} \wedge$$
$$(pullsTrailer \rightarrow highPower)$$

Proposition to Relations via BDD

- Binary Decision Diagrams (BDDs)
- Used for satisfiability
- Obtained via Shannon expansion

$$\varphi = \mathsf{ITE}(g, \varphi[g/\top], \varphi[g/\bot])$$

- Graphs can be queried as relations (in RSCRIPT)



Examples

Incorrect: $\{lowPower, pullsTrailer\}$

$$\{\}$$

Correct: $\{carBody, lowPower, manual, electric\}$

$$\{\{\}\}$$

Incomplete: $\{pullsTrailer, manual, electric\}$

$$\{\{highPower, carBody\}\}$$

Concluding...

We have discussed:

- Components as products need variability
- CDL allows the expression of this
- Correctness guaranteed through BDDs and RSCRIPT

CDL enables automatic variability management for CBSE.

Many directions still to explore...

Future Work

Future work:

- Components *inheriting variability* of their dependencies
  - NB: implications for order of configuring
- Feature *propagation* patterns
  - e.g., local vs. global features
- Generation of tools from interfaces
  - configurators, configuration-readers
- Integration with build/package environment

Feature Manipulation Environment

Questions?