Modelling Software Product Lines with the HATS Abstract Behavioural Modelling Language

Dave Clarke
Katholieke Universiteit Leuven, Belgium
dave.clarke@cs.kuleuven.be
Acknowledgements

- The following HATSters contributed to this tutorial:
  - Ina Schaefer (TU Braunschweig)
  - Reiner Hähnle (TU Darmstadt)
  - Einar Broch Johnson (U Oslo)
  - Rudi Schlatte (U Oslo)
  - Radu Muschevici (KU Leuven)
  - José Proença (KU Leuven)
  - Samir Genaim (UPM)
  - Elvira Albert (UPM)
  - German Puebla (UPM)
  - Jan Schäfer (U Kaiserslautern)
  - Richard Bubel (Chalmers UT)
Overview

- Software Product Line Engineering
- The HATS Approach – Abstract Behavioural Specification Language
- Core ABS Language
- Variability Modelling in ABS
- Product Line Analysis (Verification)
- Deployment Modelling in ABS
- Analysing ABS Models (Resource Usage)
- Challenges
Software Product Line Engineering
Motivation

A lot of companies produce products that are

• ... somehow all the same
• ... but all a little different
... somehow all the same
... but a little different
Motivation: Reflection

Why are there so many products from a single company that are more or less all the same?

- Different customer requirements (market needs)
  - features vs price
- Innovation to distinguish products from competitors’
- Products sell and earn money (customers want new version).
- The company cannot afford to develop each system from scratch.
Motivation

That products of a single company are all more or less the same is a common phenomenon in current industry

- ‘mass customisation’
- cell phones, stereos, cars, engine control systems, software, ...
- nearly all systems and devices that have been on the market for a long time

A few companies create a lot of products that have a lot of common artefacts

- there is obviously heavy reuse among these products
- otherwise no one could afford these products
# History

## The Old Days
- Every product was individually built for a single purpose
- Custom-development efforts – minimal artefact reuse

## Production Lines
- Enabled products for mass market
- Less expensive to build than individual products

## Less Diversity
- Production lines reduce the diversity in products

- Smith & Wesson, Ford, etc
Product Line Engineering

Mass Customisation

“Large scale production of goods tailored to meet individual customers’ needs” [Davis 87]

Customers

Get individual products

Company

Higher costs to make individual products

• use common building blocks (platforms) to reduce costs
Software Product Line Engineering

“a paradigm to develop software applications (software intensive systems and software products) using platforms and mass customisation” [Pohl et al ’05]

Software Platform

a set of software building blocks with common interfaces that can be combined to derive a variety of products
Software Product Lines

• A **software product line** (SPL) is a set of programs that share significant common functionality and structure.

• The differences between the set of programs are well-understood and organised in some form.

• Supports systematic re-use of configurable artefacts across development activities

• Product line targets specific market segment.
Why SPLE?

Large-Scale Reuse
- Less code
- Less docs
- Larger amount of tested code

Reduced:
- Time to market
- Development costs
- Maintenance costs
- Project risk

Increased:
- Reliability
- Testability
- Learnability
- Consistency
- Usability

Tuesday 8 November 11
Software Product Line Engineering

Domain/Family Engineering

• Extract commonality for a domain to build up the family.
• Develop for reuse.
• Build tools and components that are specialised to build family members

Application Engineering

• Build multiple applications in the domain.
• Develop with reuse.
• Use above mentioned tools to build products.

If you do good job designing the “domain”, the products will be easy to build and will have a lot in common.
SPL = Commonality + Explicit Variability

- Variability is a first-class concept
- Variability is explicitly managed
  - defined, represented, exploited, implemented, evolved

<table>
<thead>
<tr>
<th>Feature</th>
<th>Prod. 1</th>
<th>Prod. 2</th>
<th>Prod. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game Engine</td>
<td>3D, C++</td>
<td>3D, C++</td>
<td>3D, C++</td>
</tr>
<tr>
<td>Score Upload</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Lead Character</td>
<td>Mario</td>
<td>Ferrari</td>
<td>None, Puzzle</td>
</tr>
</tbody>
</table>

Commonality

Variability
Domain Engineering

Feature Model

Solution Space

Artefacts with Variability + Relations to Features

Application Engineering

Product Definitions

Artefacts without Variability
Design Efforts

• Full exploitation of the family concept requires explicit design effort.

• Design for agility – it won’t just happen

• Common aspects of family members should be decided first – can rarely make independently designed systems compatible.

• Need to know what will remain the same and what the expect changes are.

• Incorporate commonality in program generation tool and/or language. Compile new family members.
Practical Motivation

With pressure to get the first (or next) product out, why should we slow down now to prepare for future ones?

- Maintaining a set of “almost alike” products is expensive and difficult.
- Want to avoid doing the same thing again and again and ... .
- For users, unnecessary differences between products are annoying.
- Having a set of similar products to “move” through required changes – due to bugs and new requirements – slows a company down and increases development costs.
Partitioning

Each product line may be partitioned in many ways.

- partition by platform (hardware)
- partition by software platform
- partition by set of displayable natural languages
- partition by input panel
- partition by display type
- partition by programming tool used
- partition by interface type (e.g., protocol)
- partition by price considerations.
Variability Mechanisms

adaptation

replacement

extension

One core implementation. Adaptable behaviour.

Multiple Component Implementations. Choose one or develop product specific.

Generic Interface for adding components.
Adaptation Mechanisms

Inheritance

• subclass changes/overrides behaviour

Patching

• partial behaviour change with little maintenance
• delta-oriented programming
• aspect-oriented programming

Compile-time configuration

• pre-processors, macros. Makefiles

Configuration

• interface to choose between multiple implementations
• parameters/configuration file to make choice
Replacement Mechanisms

Code Generation

- generate code from high-level description (model, script)
- glue code or whole component/sub-systems

Component Replacement

- default component is replaced with another one
- often 3rd party components
- wrappers may be needed
Extension Mechanisms

Plug-ins

- architecture has interface to “plug-in” components
- example: CORBA, COM, EJB
- example: Strategy design pattern (functionality can be selected at run-time)
The HATS Project
HATS Facts

HATS: Highly Adaptable & Trustworthy Software using Formal Methods

- FP7 FET focused call Forever Yours
- Project started 1 March 2009, 48 months duration
- Integrated Project, academically driven
- 10 academic partners, 2 industrial research, 1 SME
- 9 countries
- 805 PM, EC contribution 5,63M€
- web: www.hats-project.eu
What HATS Does

In a nutshell, we ...

develop a tool-supported formal method for the design, analysis, and implementation of highly adaptable software systems characterised by high expectations on trustworthiness

for target software systems that are ...

• concurrent, distributed
• object-oriented
• built from components
• adaptable (variability, evolvability), hence reusable

Main focus: Software Product Line Engineering
Motivation

Why formal?

- informal notations cannot describe software behaviour with rigour: concurrency, modularity, correctness, security, resources, ...
- formalisation ⇒ more advanced tools
  - more complex products
  - higher automation: cost-efficiency

Why adaptable?

- changing requirements (rapid technological/market pace)
- evolution of software in unanticipated directions
- planned adaptability is key to successful reuse
Mind the Gap!

How to rigorously model behaviour of large, distributed OO systems?

Specification level

Design-oriented, architectural

Implementation-oriented

Languages (examples)

UML, FDL, ALs

Implementation-oriented
Mind the Gap!

How to rigorously model behaviour of large, distributed OO systems?

Specification level
- Design-oriented, architectural
- Abstract behavioural
- Implementation-oriented

Languages (examples)
- UML, FDL, ALs
- HATS ABS language
- Implementation-oriented
How?

A tool-supported formal method for building highly adaptable and trustworthy software

Main ingredients

1. Executable, formal modelling language for adaptable software Abstract Behavioural Specification (ABS) language

2. Tool suite for ABS/executable code analysis & development:
   - **Analytic**: functional/behavioural verification, resource analysis, feature consistency, types, visualisation
   - **Generative**: code generation, model mining, monitor inlining

   Develop method in tandem with ABS to ensure feasibility

3. Methodological and technological framework integrating HATS tool architecture and ABS language
Relevance

• Apply to empirically highly successful development method: **Software Product Line Engineering (SPLE)**

• Thorough requirements analysis, continuous evaluation
Feasibility

Ensure that analysis methods scale up

- Develop analysis methods in tandem with ABS language
  - **Incrementality:**
    - Delta modelling, delta specification, delta verification
  - **Compositionality:**
    - Concurrency model
    - Proof systems
Early Evaluation

- Develop Core ABS first
- Layered language design
- Provide tools early

Diagram:
- Assertion Language
- Composition (COGs)
  - Concurrency model
  - Object model
- Pure Functional Language
  - ADTs

Tuesday 8 November 11
Early Evaluation

- Develop Core ABS first
- Layered language design
- Provide tools early

Product Selection (PSL)

Product Line Configuration (CL)

Feature Model (µTVL)  Delta Modelling (DML)

Behavioural Interface Language

Assertion Language

Composition (COGs)

Concurrency model

Object model

Pure Functional Language

ADTs

Tuesday 8 November 11
Main Innovations of HATS

A formal, executable, abstract, behavioural modelling language

• Cutting-edge research on modelling of concurrent, OO systems
• Combines stat-of-art in verification, concurrency, specification, and programming languages communities
• Adaptability drives the design

Scalable technologies developed in tandem with ABS

• Incremental, compositional
• Analytic as well as generative technologies

Formalisation of SPLE-based development as main application

• Leveraging formal methods tools to SPLE
• Define FM-based development methodology for SPLE
Vision: a Model-Centric Development Method for SPLE

Product Line Models expressed in HATS ABS with uniform formal semantics

- consistency analysis
- correctness of reuse
- family visualisation
- test case generation
- validation, verification
- family evolution

rapid prototyping
code generation
product visualisation
test case generation
validation, verification
product evolution

Family Engineering

Application Engineering

Tuesday 8 November 11
The ABS Language
The ABS Language

Abstract Behavioural Specification Language

• A language for describing large, distributed information system families

Key Properties

• Class-based, imperative, and functional
• Sequential, concurrent, and distributed
• Expressive yet analysable
• Formal yet practical

Suitable for

• Static analysis
• Dynamic analysis
• Simulation
• Code generation
### Layered ABS Language Design

#### Behavioural Interface Specifications

<table>
<thead>
<tr>
<th>Delta Modelling</th>
<th>Feature Modelling Language</th>
<th>Architectural Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Contracts, Assertions</td>
<td>Syntactic Modules</td>
<td></td>
</tr>
<tr>
<td>Asynchronous Communication</td>
<td>Concurrent Object Groups (COGs)</td>
<td></td>
</tr>
<tr>
<td>Imperative Language</td>
<td>Object Model</td>
<td></td>
</tr>
<tr>
<td>Pure Functional Programs</td>
<td>Algebraic (Parametric) Data Types</td>
<td></td>
</tr>
</tbody>
</table>

**Done**

**Full ABS**

**Core ABS**
Core ABS
Built-in Data Types and Operators

Built-in Data Types

```haskell
data Bool = True | False;
data Unit = Unit;
data Int;  // 4, 2323, -23
data String; // "Hello World"
```
Built-in Data Types and Operators

Built-in Data Types

```haskell
data Bool = True | False;
data Unit = Unit;
data Int; // 4, 2323, -23
data String; // "Hello World"
```

Built-in Operators

- All types: `==` `!=`
- `Bool`: `~` `&&` `||`
- `Int`: `+` `-` `*` `/` `%` `<` `>` `<=` `>=`
- `String`: `+`
User-Defined Data Types

data Fruit = Apple | Banana | Cherry;
data Juice = Pure(Fruit) | Mixed(Juice, Juice);
User-Defined Data Types

```
data Fruit = Apple | Banana | Cherry;
data Juice = Pure(Fruit) | Mixed(Juice, Juice);
```

Parametric Data Types

```
data List<T> = Nil | Cons(T, List<T>);
```
User-Defined Data Types

**User-Defined Data Types**

```plaintext
data Fruit = Apple | Banana | Cherry;
data Juice = Pure(Fruit) | Mixed(Juice, Juice);
```

**Parametric Data Types**

```plaintext
data List<T> = Nil | Cons(T, List<T>);
```

**Optional Selectors**

```plaintext
data Person = Person(String name, Int age, String address);

implicitly defines corresponding functions, for example,

data String name(Person p) = ... ;
```
Functions and Pattern Matching

```scala
def Int length(IntList list) = // function names lower-case
  case list { // definition by case distinction and matching
    Nil => 0;
    Cons(n, ls) => 1 + length(ls);
    _ => 0; // wildcard pattern matches anything
  }

def A head(List<A> list) = // parametric function
  case list {
    Cons(x, xs) => 1 + length(xs);
  }

def A fromJust(Maybe<A> a) =
  case a {
    Just(x) => x; // bound variable used to extract value
  }
```
module ABS.StdLib;
export *;
data Maybe<A> = Nothing | Just(A);
data Either<A,B> = Left(A) | Right(B);
data Pair<A,B> = Pair(A,B);
data List<T> = ...;
data Set<T> = ...;
data Map<K,V> = ...;
...
def Int size<A>(Set<A> xs) = ...
def Set<A> union<A>(Set<A> set1, Set<A> set2) = ...
...
Object Model: Overview

- ABS is based on an **active-object concurrency** model **without** inheritance.
- Fields are **encapsulated** – accessible only via methods.
- At most one method active within an object – other calls are **queued**.
- Message passing between objects is **asynchronous**. Results are returned using **futures**.
- Concurrency model is enhanced with **concurrent object groups** (COGs).
Object Model: Interfaces

- Types of objects
- Multiple inheritance

```java
interface Baz { ... }
interface Bar extends Baz { ... 
    // method signatures
    Unit m();
    Bool foo(Bool b);
}
```
Object Model: Classes

- Only for object construction
- No type – classes are not types
- No inheritance

```java
// class parameters
class Foo(T u, U y) implements Bar, Baz{ ...
 }

// fields
Bool flag = False;
U g { // optional initialisation block
  g = y;
}

// method implementations
Unit m() { }
Bool foo_Bool b) { return ~b; }
```
# Imperative Constructs

## Sequential Control Flow

- **Loop** – `while (x) { ... }`
- **Conditionals** – `if (x == y) then ... else ...`
- **Synchronous method calls** – `x.m()`

## State Update and Access

- **Object creation** – `new Car(Blue);`
- **Field reads** – `x = this.f;`  
  *only on this*  
- **Field assignments** – `this.f = y;`  
  *only on this*
Concurrent Object Groups (COGs)

- Unit of distribution
- Own heap objects
- Communication by asynchronous method calls
- Cooperative multi-tasking inside COGs
Object and COG Creation

Local Object Creation

this: A
Object and COG Creation

Local Object Creation

this: A

new B();
Object and COG Creation

Local Object Creation

this: A  
new B();  
this: A  b: B
Object and COG Creation

Local Object Creation

this: A

new B();

this: A  b: B

COG Creation

this: A
Object and COG Creation

Local Object Creation

this: A

new B();

this: A  b: B

COG Creation

this: A

new cog B();
Object and COG Creation

Local Object Creation

```
this: A
new B();
this: A  b: B
```

COG Creation

```
this: A
new cog B();
this: A  b: B
```
Far and Near References

Legend:
- cog
- object
- near reference
- far reference
Type System for Far and Near References

Pluggable Type and Inference System

- Statically distinguishes near from far references
- Ensures that synchronous calls are performed only on near references

```java
{  
    [Near] Ping ping = new PingImpl();  
    [Far] Pong pong = new cog PongImpl();  
    ping.ping("Hi");          // okay  
    pong.pong("Hi");         // error: synchronous call on far reference  
}
Asynchronous Method Calls

- Syntax:
  \[ \text{target}!\text{methodName}(\text{arg1}, \text{arg2}, \ldots) \]
- Sends an asynchronous message to the target object
- Call continues and gets **future** to the result

  \[ \text{Fut<T> v = o!m(e)} \]
Cooperative Multitasking inside COGs

Multi-tasking

• A COG can have multiple tasks
• Only one is active, all others are suspended
• Asynchronous calls create new tasks

Scheduling

• Cooperative by special scheduling statements
• Non-deterministic otherwise
  • Application specific configuration of scheduling behaviour is topic of active research
Scheduling and Synchronisation

**Unconditional Scheduling**
- `suspend` command yields control to another task in COG
- Unconditional scheduling point

**Conditional Scheduling**
- `await g, where g is a guard`
- Guards can be
  - `b` – where `b` is a side-effect-free boolean expression
  - `f?` – future guards
  - `g & g` – conjunction
Futures

- Futures are first class. Can be passed around between objects.
- `f.get` – reads future `f` and blocks execution until it is resolved
- Deadlocks possible
- Use `await f?` to prevent blocking:
  ```
  Fut<T> v = o!m(e); ...; await v?; r = v.get
  ```
- `f.poll` – returns whether future has been resolved (not available in ABS)
Additional Features

Modules

```javascript
module M;
export *;
import * from N;
```

Local Assertions

```javascript
assert this.f = True;
```

Foreign Function Interface

- Use Java from ABS and vice versa
Ping Pong Example (1)

```haskell
data PingMsg = Fine | HelloPing | ByePing;

data PongMsg = NoMsg | Hello(Ping) | HowAreYou | ByePong;

interface Ping { 
    Unit ping(PingMsg m);
}

interface Pong { 
    Unit hello(Ping ping);
    Unit pong(PongMsg m);
}
```
class PingImpl(Pong pong) implements Ping {
    Unit run(){
        pong!hello(this);
    }

    Unit ping(PingMsg msg){
        PongMsg reply = case msg {
            HelloPing => HowAreYou;
            Fine => ByePong;
            ByePing => NoMsg;
        };

        if (reply != NoMsg) {
            Fut<Unit> fu = pong!pong(reply);
            fu.get;
        }
    }
}
class PongImpl implements Pong {
    Ping ping;

    Unit hello(Ping ping) {
        this.ping = ping;
        ping!ping(HelloPing);
    }

    Unit pong(PongMsg msg) {
        if (msg == HowAreYou) {
            ping!ping(Fine);
        } else {
            ping!ping(ByePing);
        }
    }
}
Ping Pong Example (4)

```java
{
    Pong pong = new cog PongImpl();
    new cogs PingImpl(pong);
}
```
Modelling Variability in ABS
Outline

• Feature Modelling
• Delta Modelling
• Product Line Configuration
• Feature Selection
• Product Generation
Running Example
Running Example:
Multi-lingual Hello World

- English: “Hello World”
- German: “Hallo Welt”
- Dutch: “Hallo Wereld”
- Swedish: “Hejsan Allihopa”
- French: “Bonjour tout le monde”
- Possibly with repetition
Ingredients of Variability in ABS

- Core ABS
- Feature Model (μTVL) – describing variability
- Deltas – implementing variability
- Configuration Language
- Feature Selection Language
The Core Functionality
interface Greeting {
    String say_hello();
}

class Greeter implements Greeting {
    String say_hello() {
        return "Hello world";
    }
}
The Core

interface Greeting {
    String say_hello();
}

class Greeter implements Greeting {
    String say_hello() {
        return "Hello world";
    }
}
interface Greeting {
    String say_hello();
}

class Greeter implements Greeting {
    String say_hello() {
        return "Hello world";
    }
}

The Core
interface Greeting {
    String say_hello();
}

class Greeter implements Greeting {
    String say_hello() {
        return "Hello world";
    }
}
Feature Modelling
Product Line Development

- Feature Model
- Family Engineering
- Product Line Artefacts Base
- Application Engineering
- Product
Feature Models

Diagram:
- **English**: Repeat → (times ≥ 2 && times ≤ 5)
- **Swedish**: Optional
- **French**
- **German**
- **Repeat**: times:[1,1000]

Constraints:
- Or
- Requires
- Excludes
- Cardinalities
Feature Model in μTVLs

```plaintext
root I18n {
  group allof {
    Language {
      group oneof {
        English, Dutch, French, German, Swedish
      }},
    }
  }

  opt Repeat {
    int [0,1000] times;
  }
}

extension English {
  ifin: Repeat ->
    (Repeat.times >= 2 && Repeat.times <= 5);
}
```
Constraint-based Semantics of Feature Model

\begin{align*}
0 \leq \text{MultiLingualHelloWorld} & \leq 1 \land \\
\text{Language} \rightarrow \text{MultiLingualHelloWorld} & \land \\
\text{Repeat}^\dagger & \rightarrow \text{MultiLingualHelloWorld} \land \\
\text{Language} + \text{Repeat}^\dagger & = 2 \land \\
0 \leq \text{Language} & \leq 1 \land \\
\text{English} \rightarrow \text{Language} & \land \text{Dutch} \rightarrow \text{Language} \land \text{French} \rightarrow \text{Language} \land \\
\text{German} \rightarrow \text{Language} & \land \\
1 \leq \text{English} + \text{Dutch} + \text{French} + \text{German} & \leq 1 \land \\
0 \leq \text{English} & \leq 1 \land 0 \leq \text{Dutch} \leq 1 \land 0 \leq \text{French} \leq 1 \land \\
0 \leq \text{German} & \leq 1 \land \\
0 \leq \text{Repeat}^\dagger & \leq 1 \land \\
\text{Repeat} \rightarrow \text{Repeat}^\dagger & \land \\
0 \leq \text{Repeat} & \leq 1 \land 0 \leq \text{Repeat}.\text{times} \leq 1000 \land \text{Repeat}.\text{times} > 0 \land \\
\text{English} \rightarrow (\text{Repeat} \rightarrow (\text{Repeat}.\text{times} \geq 2 \land \text{Repeat}.\text{times} \leq 5)).
\end{align*}
µTVL Feature Modelling Language

• µTVL adapts existing TVL language [Claessen, FUNDP, Namur U]

• Constraint solver written in Choco:
  • finds valid feature selections matching specified constraints
  • checks validity of feature selections
Problems of Feature Models

- Are there any valid feature combinations?
- Is a given feature selection valid?
- Is a given partial feature selection valid?
- Are there any unused features?
- Is there any redundant variation points?
- How many feature combinations are there?

Solvable using SAT and #SAT techniques
Delta Modelling
Delta-oriented Programming

Feature Model

Family Engineering

Product Line Artefacts Base

Feature Selection

Application Engineering

Product

Tuesday 8 November 11
Delta-oriented Programming
Delta-oriented Programming

Feature Model

Family Engineering

Product Line Artefacts Base

Product Selection

Automated Product Derivation

Core Products – complete product for some feature configuration

Product Deltas – additions, removals, modifications to core product

Product
Delta-oriented Programming

• Modifications on Class Level:
  • Addition, Removal and Modification of Classes

• Modifications of Class Structure:
  • Changing Super Class and Constructor
  • Adding/Removing Fields/Methods
  • Modifying Methods (wrapping original call)

[Delta-oriented Programming of Software Product Lines
Schaefer, Bettini, Bono, Damiani, Tanzarella. SPLC 2010.]
The Repeat Delta

delta Rpt (Int times) {
    modifies Greeter {
        modifies String say_hello() {
            String result = "";
            Int i = 0;
            while (i < times) {
                result = result + original();
                i = i + 1;
            }
            return result;
        }
    }
}
The German Delta

delta De {
    modifies Greeter {
        modifies String say_hello() {
            return "Hallo Welt";
        }
    }
}

Tuesday 8 November 11
The Dutch Delta

delta Nl {
  modifies Greeter {
    modifies String say_hello() {
      return "Hallo wereld";
    }
  }
}

Tuesday 8 November 11
The French Delta

delta Fr {
    modifies Greeter {
        modifies String say_hello() {
            return "Bonjour tout le monde";
        }
    }
}
The Swedish Delta

delta Sv {
    modifies Greeter {
        modifies String say_hello() {
            return "Hejsan allihopa";
        }
    }
}
Product Line Configuration
Features-Delta Mapping

Configuration

application conditions & delta ordering

Feature Model

Artefacts

Deltas

Core
Configuration

- **Links** feature model with deltas
- **Adds** application conditions to deltas:
  - constraints on features and attributes
- **Specifies** ordering between deltas
- **Nests** deltas to ensure atomic application
product line HelloMultiLingual {
    features Repeat, German, French, Dutch, Swedish;
    core English;

delta De when German && not Repeat;
delta Fr when French;
delta Nl when Dutch;
delta Sv when Swedish && Repeat;
delta Rpt(Repeat.times)
    after De, Fr, Nl, Sv when Repeat;
}
Configuration

```java
product line HelloMultiLingual {
    features Repeat, German, French, Dutch, Swedish;
    core English;

    delta De when German && not Repeat;
    delta Fr when French;
    delta Nl when Dutch;
    delta Sv when Swedish && Repeat;
    delta Rpt(Repeat.times) after De, Fr, Nl, Sv when Repeat;
}
```
Feature Selection
Feature Selection

Feature Selection

Configuration

Feature Model

Deltas

Core

Product

Artefacts

Uses

Satisfies

Generates
Feature Selection

- *Specifies* selected features and their attributes.
- *Final Ingredient* **required** to Generate Product.
- *Checked* against Feature Model.
Feature Selection

// basic product with no deltas
product P1 {
    Greeting bob;
    bob = new Greeter();
    String s = "";
    s = bob.say_hello();
}

\{ Initialisation (aka main) \}
// apply delta Fr and Repeat product P3 (French, Repeat{times=10}) {
  Greeting bob;
  bob = new Greeter();
  String s = "";
  s = bob.say_hello();
}
Product Generation
Product Line Development

Feature Model → Family Engineering → Product Line Artefacts Base → Application Engineering → Product

Feature Configuration → Family Engineering
Product Generation

• For a Feature Configuration:
  • **Select** product deltas with valid application condition
  • **Determine** linear ordering of product deltas compatible with partial ordering
  • **Apply** changes specified by product deltas to core product in the linear order
Given Feature Selection

// apply delta Fr and Repeat
product P3 (French, Repeat{times=10})
{
    Greeting bob;
    bob = new Greeter();
    String s = "";
    s = bob.say_hello();
}
Apply Delta Fr

class Greeter implements Greeting {
    String say_hello() {
        return "Hello world";
    }
}

delta Fr {
    modifies Greeter {
        modifies String say_hello() {
            return "Bonjour tout le monde";
        }
    }
}
Apply Delta Fr

class Greeter implements Greeting {
    String say_hello() {
        return "Bonjour tout le monde";
    }
}

Configure Repeat

times=10

delta Rpt (Int times) {
    modifies Greeter {
        modifies String say_hello() {
            String result = ""
            Int i = 0;
            while (i < times) {
                result = result + original();
                i = i + 1;
            }
            return result;
        }
    }
}
Configure Repeat

default Rpt {
    modifies Greeter {
        modifies String say_hello() {
            String result = "";
            Int i = 0,
            while (i < 10) {
                result = result + original();
                i = i + 1;
            }
            return result;
        }
    }
}
Apply Repeat

class Greeter implements Greeting {
    String say_hello() {
        return "Bonjour tout le monde";
    }
}

delta Rpt {
    modifies Greeter {
        modifies String say_hello() {
            String result = ""
            Int i = 0;
            while (i < 10) {
                result = result + original();
                i = i + 1;
            }
            return result;
        }
    }
}
class Greeter implements Greeting {
    String __say_hello_original() {
        return "Bonjour tout le monde";
    }

    String say_hello() {
        String result = "";
        Int i = 0;
        while (i < 10) {
            result = result + __say_hello_original();
            i = i + 1;
        }
        return result;
    }
}
Adding Initialisation

class Greeter implements Greeting {
    String __say_hello_original() {
        return "Bonjour tout le monde";
    }

    String say_hello() {
        String result = "";
        Int i = 0;
        while (i < 10) {
            result = result + __say_hello_original();
            i = i + 1;
        }
        return result;
    }
}

Greeting bob;
bob = new Greeter();
String s = "";
s = bob.say_hello();
Typing Issues

- Errors
  - Field, method, class missing
  - Incorrect type: field/method
  - Dependencies – collections of methods/classes and their types various deltas used. Need to be provided in feature selection.

- Warnings
  - Field/method/class not expected one
  - Conflicting implementation orderings

- Product line level checking instead of per product (= after generation) checking
Typing Issues

- Errors
  - Field, method, class missing
  - Incorrect type: field/method
  - Dependencies – collections of methods/classes and their types various deltas used. Need to be provided in feature selection.

- Warnings
  - Field/method/class not expected one
  - Conflicting implementation orderings

- Product line level checking instead of per product (= after generation) checking

These challenges are being addressed.
Summary
HATS ABS Features for Variability Modelling

- HATS ABS is a collection of 5 languages
  - Core Product
  - Product Deltas
  - Feature Modelling
  - Configuration: Links FM and Core+Deltas
  - Feature Selection
- Implementation available
Product Line Analysis
Product Line Analysis

Goal

Guarantee that every product variant satisfies its requirements

Product line analysis approaches
[Thüm et al ’11]
interface IAccount { Unit deposit(Int x); }

class Account implements IAccount { // Core Product
    Int balance = 0;
    Unit deposit(Int x) {
        balance = balance + x;
    }
}

delta DFee(Int fee) { // Implements Feature Fee
    modifies class Account {
        modifies Unit deposit(Int x) {
            if (x >= fee) original(x-fee);
        }
    }
}
Delta Modelling Example (cont’d)

delta DOverdraft() { // Implements Feature Overdraft
  modifies class Account {
    adds Int limit;
    modifies Unit deposit(Int x) {
      if (balance + x > limit) original(x);
    }
  }
}

productline AccountPL { // Product Line Declaration
  features Basic, Overdraft, Fee;
  delta DFee(Fee.amount) when Fee;
  delta DOverdraft after DFee when Overdraft;
}
Specification – Design by Contract

• Specify each class using method contracts and class invariants.

• A contract for a method \( m \) consists of:
  1. a first-order formula \( r \) called a precondition or requires clause
  2. a first-order formula \( e \) called postcondition or ensures clause
  3. a set of program locations \( a \), called an assignable clause, whose value can potentially be changed by \( m \).

• First-order formulas \( i \) can be attached as invariants to classes.
Design-by-contract: Example

class Account implements IAccount {
    Int balance = 0;

    @requires x > 0;
    @ensures balance <= \old(balance) + x;
    @assignable balance;
    Unit deposit(Int x) {
        balance = balance + x;
    }
}
Delta Specification

Specification level analogue of code deltas

Capabilities of delta specifications

- addition, removal and modification of contracts separately for requires clauses, ensures clauses, and assignable clauses
- usage of the keyword `original` in clauses of contracts

**Semantics:** compiler inserts most recent instance of contract

- invariants can only be explicitly added or removed (cumulative relative to current variant)
Delta Specification: Example

delta DFee(Int fee) {
    modifies class Account {
        // new invariant for fee parameter
        adds @invariant fee >= 0;

        // modifies contract, changes @ensures clause
        modifies
            @invariant balance <= \old(balance) + max(x-fee,0);

        // modified method (as before)
        modifies Unit deposit(Int x) {
            if (x >= fee) original(x-fee);
        }
    }
}
Verification of Delta-oriented Product Lines

- Application order can be specified as an ordered sequence of a partition of the set of deltas ⇒ may assume that delta model has the following form

\[
[\delta_1 \cdots \delta_{ln}] < \cdots < [\delta_1 \cdots \delta_{hn}]
\]

Complexity

- \( h \) is the height of the delta model and \( w = \max\{n_1, \ldots, n_h\} \) is the width

- Number of possible products in product line is bounded by \((2^w)^h\)

- Naive product-based verification many proofs \(O(2^{wh}M)\), where \( M \) is the maximal number of methods
Incremental Verification

1. Determine proof-obligations by delta-oriented slicing

2. Proof reuse for changed obligations

[Bruns, Klebanov, Schaefer FoVeOOS 2010]
Behavioural Subtyping

Liskov’s Principle for Object-oriented Programming

In all places where a superclass is expected, a subclass can also be used

Consequences for specifications

1. The precondition of a method overridden in a subclass must be implied by the precondition of the superclass method

2. The postcondition of a method overridden in a subclass must imply the postcondition of the superclass method

3. When assignable clauses are present, the assignable locations in a subclass must be a subset of the assignable locations in the superclass

4. The invariant of a subclass must imply the invariant of its superclass
Preserving Correctness in Delta Application

Case (1)-(3) as a formal definition

Definition (More Specific Contract)
For two methods $m, m'$, with contracts $m.r, m'.r'$, etc., the second contract is more specific than the first if

$$(m.r \rightarrow m'.r') \land (m'.e' \rightarrow m.e) \land (m'.a' \subseteq m.a)$$

A Liskov Principle for DOP

For a delta model $[\delta_{i1} \cdots \delta_{in_i}] < \cdots < [\delta_{h1} \cdots \delta_{hn_h}]$,

1. Contracts in larger deltas may only get more specific
2. Larger deltas may only introduce invariants that are implied by previous invariants

Tuesday 8 November 11
Delta-based Verification

Goal: A Feature-based Verification Method

Verify method contracts of products by analysing each delta in isolation

Outline

• The core product must be correct wrt its contracts.
• For each method \( m \) in a delta, we must show the proof obligation

\[
\delta.m.r \rightarrow \langle m(p) \rangle \delta.m.e \land A(\delta.m.a, \delta.m)
\]

(A(a, m) ensures correctness of assignable clause a wrt method \( m \))
• For each method \( n \) called in \( m \), use the first introduction of \( n \) in the given delta model (i.e., in a \( \delta_{ij} \) with minimal index \( i \))
  • Subsequent contracts of \( n \) can only get more specific
  • Ensures that the approximation of the call by its contract is valid for all of possible versions of \( n \)
• Likewise, use the “largest” assignable set of locations
Delta-based Verification

Theorem (Correctness of Delta-based Verification)

Given a delta model consisting of a core $C$ and a partition of deltas $D = \{[\delta_{11} \ldots \delta_{1n_1}], \ldots, [\delta_{h1} \ldots \delta_{hn_h}]\}$.

Assume the following holds:
1. $C$ satisfies the contract of all its methods
2. For all $\delta$ occurring in $D$:
   1. $\delta$ is verified (in the sense above)
   2. The contract of each method $m$ added or modified in $\delta$ either is the first occurrence of $m$ in the delta model or it must be more specific than the contract of the most recent method in $D$ from $\delta.m$

Then every product obtained from the given delta model satisfies its specification, i.e., each of its methods satisfies its contract.

**Complexity:** Number of Proof Tasks is in $O(hwM)$
Verification of Invariants

- Assume exactly one global invariant for each product (global visibility; modular schemes are orthogonal issue)
- Invariants are special case of method contracts, where pre- and post-condition are identical (implies that once cannot make them “more specific” like contracts)

Core Invariants

- According to Liskov principle for DOP, deltas may only introduce invariants implied by previous invariants
- **Consequence**: invariants can only be introduced in the core
- Delta-based verification extended by core invariant
  
  $$(m.r \land C.i) \rightarrow \langle m(p)\rangle m.e \land C.i \rightarrow \langle m(p)\rangle C.i$$

- Same complexity as above, but a serious limitation
Verification of Invariants

Family Invariants

- A $\delta$ may introduce new invariants conjoined with existing ones.
- **Consequence:** existing methods must ensure that the strengthened invariants emerging during delta application are established.
- A safe approximation for those invariants is $I_\delta = \bigwedge_{\delta' \leq \delta} \delta'.i$
- Need to establish $I_\delta$ for any existing method (not only for the methods mentioned in $\delta$) as part of the verification of $\delta$.
- Complexity is still in $O(whM)$.
- **Drawback:** $I_\delta$ may be stronger than necessary for specific product.
Constraint-based Verification

- Liskov principle for DOP is a very strong restriction.
- Instead of assuming more specific contracts, provide called methods with a required contract to establish the contract of the caller method.

Idea of Delta/Product-based Verification:
- Provide deltas with interfaces that express the requirements and guarantees of contained methods.
- Verify each delta against this interface in isolation.
- Show for each product that interface specifications are satisfied by the product’s methods.
Delta Interfaces

Delta interfaces specify expectations about contracts of called methods.

```plaintext
@expects Unit original_deposit(Int x) with
{  @requires x > 0;
    @ensures balance = \old(balance) + x; }

@requires x > 0;
@ensures balance <= \old(balance) + max(x-fee,0);
Unit deposit(Int x) {
  if (x >= fee) {
    @requires x - fee > 0;
    original(x-fee);
    @ensures balance = \old(balance) + x - fee;
  }
}
```
Product-based Constraint Generation

For each product, generate the set of product constraints from the interfaces of the applied deltas.

For each applied delta:

1. For added methods, add constraints to set of product constraints.
2. For removed methods, remove constraints from set of product constraints.
3. For modified methods, update set of product constraints to contain constraints of called methods.
4. original call is immediately bound. Expected contract is checked against provided contract.
Product-based Constraint Checking

- The product constraints consist of a set of contracts for each method that are expected from the called methods.

- For each called method $m$, check that the expected contract $\text{contr}_{\text{exp}}$ is more specific than the provided contract $\text{contr}_{\text{prov}}$ of the method contained in the product, i.e., that

\[
(m.r_{\text{prov}} \rightarrow m.r_{\text{exp}}) \land (m.e_{\text{exp}} \rightarrow m.e_{\text{prov}}) \land (m.a_{\text{exp}} \subseteq m.a_{\text{prov}})
\]

- Number of proof tasks:
  - $O(whM)$ proofs for verifying the deltas
  - $O(2^{whMN})$ product checks where $N$ is the maximal number of called methods (including original calls)
Summary

- **Family Based**

- **Product Based** \( O(2^{hwM}) \)
  - Delta/Product-based
  - \( O(hwM) \) delta proofs + \( O(2^{whMN}) \) product checks

- **Feature Based** \( O(hwM) \)
Deployment Modelling in ABS

Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and S. Lizeth Tapia Tarifa
Dynamic Resource Reallocation Between Deployment Components

Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and S. Lizeth Tapia Tarifa
Validating Timed Models of Deployment Components with Parametric Concurrency
Motivation

Software systems tend to be released for a range of different platforms
Motivation

Software systems tend to be released for a range of different platforms

Examples
Motivation

Software systems tend to be released for a range of different platforms

Examples

- Software Product Lines
Motivation

Software systems tend to be released for a range of different platforms

Examples

- Software Product Lines
- Embedded Systems
Motivation

Software systems tend to be released for a range of different platforms

Examples

- Software Product Lines
- Embedded Systems
- Sensors
Motivation

Software systems tend to be released for a range of different platforms

Examples

- Software Product Lines
- Embedded Systems
- Sensors
- Web Services
Motivation

Software systems tend to be released for a range of different platforms

Examples

- Software Product Lines
- Embedded Systems
- Sensors
- Web Services
- Operating Systems
Motivation

Software systems tend to be released for a range of different platforms

Examples

- Software Product Lines
- Embedded Systems
- Sensors
- Web Services
- Operating Systems
- Clouds, Virtual Architectures
Motivation

Software systems tend to be released for a range of different platforms

Examples

• Software Product Lines
• Embedded Systems
• Sensors
• Web Services
• Operating Systems
• Clouds, Virtual Architectures

Need to model software that ranges over deployment scenarios
Aim
Aim

Formalize

Platform Functionality

Deployment scenario 1

Deployment scenario 2

... 

Deployment scenario N

Tuesday 8 November 11
Aim

Model

Formalize

Platform Functionality

Deployment scenario 1

Deployment scenario 2

Deployment scenario N
Aim

Model

Formalize

Apply analysis techniques

Platform Functionality

Deployment scenario 1

Deployment scenario 2

... 

Deployment scenario N

Tuesday 8 November 11
Aim

How will models perform under different deployment assumptions?
Approach

Real-time ABS

- Extend ABS with notion of time: durations and duration guards
- Non-intrusive extension: untimed models work in Real-time ABS
- Run-to-completion semantics
- Associate deadlines with method calls

Apply performance analysis to OO models that range over deployment scenarios

Real-time ABS includes models of deployment scenarios

1. User-defined scheduling strategies
2. Deployment components: resource-restricted execution contexts
Real-time ABS

Modelling Timed Systems

- Time evolves uniformly in Real-Time ABS
- `now()` expression reads the current time
- ABS has a duality between blocking and suspending processes
- Real-Time ABS: `block` or `suspend` a process for a period of time
- `duration(min,max)`
- `await duration(min,max)`

Deadlines to Method Calls

- Processes have a local associated deadline
- `deadline()` expression: time left before local deadline?
- Default deadline is infinity, but can be overridden by annotations
Real-time ABS Example

Method Definition

```java
Bool m() {
    duration(2, 5);
    return deadline() > 0;
}
```

Method Calls

Regular method call

```java
Bool success = o.m();
```

Default deadline: infinity

```
Bool success = o.m();
```

Annotation with fixed deadline

```
[Deadline : 5] Bool success = o.m();
```

Annotation with state-dependant deadline

```
[Deadline : e] Bool success = o.m();
```
Scheduling

Real-time Scheduling Problem

Is a given workload (cost, deadline) schedulable?

• EDF is optimal (for non-preempting, single-processor)

Priority-based “conventional” scheduling

• Hard real-time: mission critical deadlines
• Soft real-time: missing deadlines = degradation of QoS

With soft real-time requirements: which jobs to miss?

What is a good priority for a video player decoding 24 frames per second without processing needs in-between?
Scheduling in ABS

- ABS has (non-preemptive) cooperative scheduling.
- The scheduling problem is to select a process from the queue when the object is idle.

Process Attributes for Scheduling

- Arrival time $r$
- Cost (computation time) $c$
- (Relative) deadline $d$
- Start time $s$
- Finish time $f$
- Criticality $crit$
- Value $v$

Derived Process Attributes

Buttazzo also uses the following attributes for real-time scheduling, which can be derived from the previous ones:

- Absolute deadline \( D=r+d \)
- Response time \( R=f-r \)
- Lateness \( L=f-D \)
- Tardiness \( E=\max(0,L) \)
- Laxity \( X=r-c \)
Processes in Real-time ABS

The datatype of processes in ABS

- Arrival time \( r \)
- Cost (computation time) \( c \)
- (Relative) deadline \( d \)
- Start time \( s \)
- Finish time \( f \)
- Criticality \( \text{crit} \)
- Value \( v \)

In Real-time ABS

```plaintext
data Pid;
data Process = Proc(Pid pid, String m, Time r, Duration c, Duration d, Time s, Time f, Bool crit, Int v);
```
User-defined Scheduling Policies

- Scheduler: \( \text{List<Process>} \rightarrow \text{Process} \)
- Arbitrary scheduler in Real-Time ABS:

```python
def Process schedule(List<Process> s) = head(s);
```

- \( s \) is guaranteed non-empty (if object has no processes to run, scheduler will not be called)
- No “fresh” process can be returned since there is no constructor for the \( \text{Pid} \) datatype
- Many schedulers simply sort \( s \) according to some criterion, but passing object state into scheduler is also possible
- Optional access to object state in the scheduling function
Example: EDF Scheduler

EDF: select process $s$ with earliest deadline

```java
def Process edf(List<Process> l) = edf_h(head(l), tail(l));
def Process edf_h(Process p, List<Process> l) =
    case l {
        Nil => p;
        Cons(p2,l2) =>
            if deadline(p) < deadline(p2) then edf_h(p, l2)
            else edf_h(p2, l2);
    };
```
Adding Schedulers to a Model

Annotations are used to override default schedulers in Real-Time ABS

- Define scheduler
  
  ```java
  def Process edf(List<Process> l) = ...
  ```

- Add to class definition
  
  ```java
  [Scheduler: edf(queue)] class C implements I { ... }
  ```

- Add to object instantiation
  
  ```java
  [Scheduler: fifo(queue)] I = new C
  ```

Hierarchy of Scheduling Policies

- System wide defaults
- Annotations at class and instance levels
Example: Reactive System

interface IServer {
    Bool request(String job, Int bc, Int wc);
}

class Server {
    List<Pair<String, Int>> history = Nil;
    Bool request(String job, Int bc, Int wc) {
        duration(bc, wc);
        history = Cons(Pair(job, deadline()), history);
        return deadline() >= 0;
    }
}

{ IServer s = new cog Server(); s!request("Job1", 1, 1);
    ...
}
Example: Client Behaviour (1)

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
<th>Job 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>bc</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>wc</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>deadline</td>
<td>3</td>
<td>10</td>
<td>7</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 5</th>
<th>Job 3</th>
<th>Job 4</th>
<th>Job 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDF</td>
<td>J1</td>
<td>J5</td>
<td>J3</td>
<td>J4</td>
<td>J2</td>
</tr>
<tr>
<td>FIFO</td>
<td>J1</td>
<td>J2</td>
<td>J3</td>
<td>J4</td>
<td>J5</td>
</tr>
</tbody>
</table>

Time:
- \(d_1\)
- \(d_5\)
- \(d_3\)
- \(d_4\)
- \(d_2\)
Example: Client Behaviour (2)

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
<th>Job 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>bc</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>wc</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>deadline</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

- **bc** (beginning of computation)
- **wc** (worst-case response time)
- **deadline**

**Graphical Representation**

EDF (Earliest Deadline First)

<table>
<thead>
<tr>
<th>Time</th>
<th>J1</th>
<th>J3</th>
<th>J2</th>
<th>J5</th>
<th>J4</th>
<th>J4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>J1</td>
<td>J2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>J1</td>
<td>J3</td>
<td>J3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>J1</td>
<td></td>
<td></td>
<td>J5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td>J4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>J5</td>
</tr>
<tr>
<td>6</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIFO (First In, First Out)**

<table>
<thead>
<tr>
<th>Time</th>
<th>J1</th>
<th>J3</th>
<th>J2</th>
<th>J5</th>
<th>J4</th>
<th>J5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>J1</td>
<td>J2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>J1</td>
<td>J3</td>
<td>J3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>J1</td>
<td></td>
<td></td>
<td>J5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td>J4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>J5</td>
</tr>
<tr>
<td>6</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>J1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Simulation Results in the ABS Interpreter

- EDF finds expected results (in single-processor, non-preemptive setting), minimizes lateness in above example
- Next: let’s overload the server:
  workload: 30% “expensive”, 70% “cheap” tasks

![Graph showing simulation results]
Deployment Components

A deployment component acts as an execution context

- Objects execute in the context of a deployment component
- Execution with bounds on available resources
- The resources are shared between the component’s objects
- Object execution uses resources in the deployment component(s)
- How resources are consumed depends on the specific cost model
Deployment Components with Parametric Concurrency

Client

- call(n) to tel
- sms() to sms

Clock

telcomp

Client

- call(n) to telcomp
- sms() to smscomp

smscomp

sms
Deployment Components with Parametric Concurrency

Clock

Deployment Component

Resources

Client

call(n)

sms()

tel

Client

smscomp

call(n)
sms()

sms

telcomp
Deployment Components with Parametric Concurrency

- Parametric bound on abstract processing resources [FoVeOOS 2010]
Deployment Components with Parametric Concurrency

- **Parametric bound** on abstract processing resources [FoVeOOS 2010]
- **Resources reflect** the execution capacity of the deployment component in a time interval
Deployment Components with Parametric Concurrency

- **Parametric bound** on abstract processing resources [FoVeOOS 2010]
- **Resources reflect** the execution capacity of the deployment component in a time interval
- **Resources abstract** from the number and speed of the physical processors available to the component
Propose an extension of the syntax and semantics of Real-Time ABS:

- \textbf{component}(r) creates a new deployment component
  \[ dc = \text{component}(r); \]

- An \textbf{optional clause} in the creation of cogs
  \[ \text{new cog } C(e) \text{ in } dc; \]
Example: Phone Services – Real-time ABS Model

Telephone Service

```java
interface TelephoneService {
  Unit call(Int duration);
}
class TelephoneService implements TelephoneService {
  Unit call(Int duration) {
    Time t; t = now;
    await now >= t + duration;
  }
}
```

SMS Service

```java
interface SMSService { Unit sendSMS(); } 
class SMSService implements SMSService {
  Unit sendSMS() { skip; }
}
```
Example: New Year’s Eve Client Behaviour

- Alternate sms and call
- Huge number of sms per time interval
- Alternate sms and call

Midnight Window

- 50
- 70

Tuesday 8 November 11
Example: New Year’s Eve Client Behaviour

class NYEbehavior (Int cycle, TelephoneService ts, SMSService smss)
{
    Time created = now; Bool call = false;

    Unit normalBehavior() {
        Time t = now;
        if (now > created+50 ∧ now < created+70) {
            this!midnightWindow();
        } else {
            if (call) { ts.call(1); } else { smss!sendSMS(); };
            call = ¬call; await now ≥ t+cycle; this!normalBehavior();
        }
    }

    Unit midnightWindow() {
        Time t = now; Int i = 0;
        if (now > created+70) {
            this!normalBehavior();
        } else {
            while (i < 10) { smss!sendSMS(); i = i+1; };
            await now > t; this!midnightWindow();
        }
    }

    Unit run() { this!normalBehavior(); }
Simulating and Testing the Real-time ABS Model

Main Block of the Real-time ABS model

```java
{
    Component smscomp = component(50);
    Component telcomp = component(50);

    SMSService sms = new cog SMSService() in smscomp;
    TelephoneService tel = new cog TelephoneService() in telcomp;

    Client c = new cog NYEbehavior(1,tel,sms);
    ...
}
```
Example: Simulation in the Maude Interpreter
Dynamic Resource Reallocation

Allow resources to be exchanged between deployment components
Dynamic Resource Reallocation

Allow resources to be exchanged between deployment components
Dynamic Resource Reallocation

Real-time ABS language constructs for resource reallocation

- Let components and resources be first-class citizens in the language
- Now, we can store and pass on components and resource values

Consider a variable $dc$ of type Component and $r$ of type Resource:

- The expression `thiscomp` returns $dc$ of the object.
- The expression `available` returns the number of resources currently allocated to `thiscomp`
- The expression `load(e)` returns the average number of used resources in `thiscomp` during the last $e$ time intervals
- The statement `transfer(dc,r)` reallocates $r$ resources from `thiscomp` to another component $dc`
Example: Load Balancing Strategy – Real-Time ABS Model

A simple load balancing scheme

```java
interface Balancer {
    Unit setPartner(Balancer p); Unit request(Component comp);
}

class Balancer implements Balancer {
    Balancer partner = null;
    Unit setPartner(Balancer p) { partner = p; }
    Unit request(Component comp) {
        if (now(1)<available -10) { transfer(comp, available/2);}
    }
    Unit run () {
        Time t = now; await now > t;
        if (partner != null ∧ available < now(1)*0.9){
            partner.request(thiscomp); this!run();
        }
    }
}
```
Example: Simulating and Testing the Real-time ABS Model

Main block of the Real-time ABS model with load balancing

```java
Component smscomp = component(50);
Component telcomp = component(50);

SMSService sms = new cog SMSService() in smscomp;
TelephoneService tel = new cog TelephoneService() in telcomp;

Balancer smsb = new cog Balancer in smscomp;
Balancer telb = new cog Balancer in telcomp;

smsb.setPartner(telb);
telb.setPartner(smsb);

Client c = new cog NYEbehavior(1,tel,sms);
...
```

Simply by commenting out the Balancer, we get the same functional behaviour model without load-balancing.
Example: Simulation in the Maude Interpreter

- Without Load Balancing
- With Load Balancing

[Graphs showing comparison of SMS and Telephony loads with and without load balancing.]
Object Mobility

- Objects may change execution context
- Object mobility complements resource allocation
Object Mobility

- Objects may change execution context
- Object mobility complements resource allocation
Combining Resource Analysis with Simulations

Cost Analysis: Worst case upper bounds for resources

- What is the cost of a transition?

Example: \( \text{cost}(x = f(y)) \) in terms of actual values \( y \mapsto \nu \)

- Applied to memory analysis for ABS models [FM 2011]
Abstract Resources

• Resources such as processor capacity and memory can be expressed in a general framework of user-defined resource usage, which can be expressed by annotations.

  [cost: e] s
  [free: e] s
  [cost: e1, free: e2] s

• How to represent the cost model in the surface syntax?
User-defined Scheduling

- User-defined scheduling policies, expressed in the modelling language
- Per-class and per-object schedulers
- Reflective and safe
- Using process attributes from real-time scheduling
- Modular, optional part of specification
Deployment Components

- Need models and analysis methods that range over different deployment scenarios
- Topologies of ABS deployment components are parametric in available resources
- Possible to express and compare interesting non-functional system properties
Analysing ABS Models

Elvira Albert, Samir Genaim, Miguel Gómez-Zamalloa, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa
Simulating Concurrent Behaviors with Worst-Case Cost Bounds
Analysing ABS Models

Different Analyses

• Runtime assertion checking
• Deadlock analysis
• Functional verification
• Resource guarantees
Analysing ABS Models

Different Analyses

• Runtime assertion checking
• Deadlock analysis
• Functional verification
• Resource guarantees

Based on the COSTA analyser
Resource Guarantees

- The aim is to statically obtain/verify upper and lower bounds of resource consumption
  - upper bounds guarantee that a program will run within the resources available on a particular device
  - lower bounds are useful for scheduling distributed execution
Resource Guarantees

• The aim is to statically obtain/verify upper and lower bounds of resource consumption
  • upper bounds guarantee that a program will run within the resources available on a particular device
  • lower bounds are useful for scheduling distributed execution

• The resources considered
  • processor cycles, number of executed (bytecode) instructions, etc
  • memory usage
  • billable events (calls to a specific method)
  • termination (it guarantees the existence of an upper bound).
What to expect from a resource guarantee analysis?

```
List l = null;
while (n > 0) {
    l = new List(new Integer(n), l);
    n = n - 1;
}
```

**Inference:** how much resources the loop consumes

- 2*n objects, or n*Integer + n*List objects
- 16*n bytes (e.g., when both Integer and List require 8 bytes)
- 19*n + 7 executed instructions
- \(O(n)\) bytes/objects/instructions (asymptotic), easier to read, easier to understand, more abstract, etc
Classical Approach to Cost Analysis

Classical approach [Wegbreit’75] to cost analysis consists of 2 phases:

1. express the cost in terms of recurrence relations

\[
\text{loop}(n) = \begin{cases} 
0 & [n \leq 0] \\
\text{size}(	ext{Integer}) + \text{size}(	ext{List}) + \text{loop}(n-1) & [n > 0]
\end{cases}
\]

\[
\text{loop}(n) = \begin{cases} 
7 & [n \leq 0] \\
19 + \text{loop}(n-1) & [n > 0]
\end{cases}
\]

2. solve the relations, obtaining a closed-form upper bound

\[
\text{loop}(n) = \begin{cases} 
n \times \text{size}(	ext{Integer}) + n \times \text{size}(	ext{List}) & \\
O(n)
\end{cases}
\]

\[
\text{loop}(n) = \begin{cases} 
19 \times n + 7 & \\
O(n)
\end{cases}
\]
COSTA – COSt and Termination Analyser

Cost and Termination analyser follows the two-phased approach of Wegbreit

Solves the cost relations to obtain a closed form

- COSTA relies on PUBS for solving cost relations
- cost relations are more general than linear recurrence relations
- many cost analyses failed due to the computer algebra system

COSTA is parametric on the notion of cost model $\mathcal{M}$

- NINST: $\mathcal{M}(i) = 1$ for every bytecode instruction $i$
- HEAP: $\mathcal{M}(\text{new } C) = \text{size}(C)$
Cost Models

- **Concurrent Objects** – counts the total number of objects created along the execution path
- **Task-level** – peak of activity along any program execution path
- **Remote requests** – counts the number of asynchronous calls to objects (remote requests to server)

Spawned processes & remote requests

Implemented in the COSTABS extension of COSTA for ABS
Cost Analysis for Functional ABS

data List<T> = Nil | Cons(T, List<T>);

def List<T> concat<A>(List<T> l1, List<T> l2) =
  case l1 {
    Nil => l2;
    Cons(head, tail) => Cons(head, concat (tail, l2));
  };

Inference of resource usage of this program is almost straightforward using underlying technology of COSTA:

Types define size functions
|Cons (A, B)| = |l| + |B| and |Nil| = 0

concat(0, 12) = 0
concat(11, 12) = 2 + concat(11 - 1, 12)

Worst-case bounds can be obtained for the functional part of ABS combined with simulation.
Cost Analysis of Imperative ABS

• Automatically infer the resource usage of concurrent programs is challenging due to their complexity.

• Computations can be suspended and resumed and shared variables can be modified by different tasks that all interact with each other.

• A cost analyser must consider all possible paths and interactions.

• Standard definition of cost aggregates the cost of all execution steps: keep different computing infrastructures separate.

Contributions of approach

• Sound and precise size relations: bound the number of iterations of loops in the presence of concurrent behaviours and shared data.

• Novel form of recurrence equations parametric on the notion of cost centre to keep the cost of distributed components separate.
Size Analysis for Concurrent Objects

Unit reqFile(Node s, FileName fname) {
...
    while (i > 0) {
        if (size > i) incr = i; else incr = size;
        i = i - incr;
        l2 = s ! getPacks(fname, incr, i);
        await l2?; ps = l2.get;
        f = app(ps, f);
    }
...}
Size Analysis for Concurrent Objects

Unit reqFile(Node s, FileName fname) {
...
    while (i > 0) {
        if (size > i) incr = i; else incr = size;
        i = i - incr;
        l2 = s ! getPacks(fname, incr, i);
        await l2?; ps = l2.get;
        f = app(ps, f);
    }
...
}

• By ignoring `await l2?`, field `size` refers to the same memory location in all iterations (no aliasing)
Size Analysis for Concurrent Objects

By ignoring `await l2?`, field `size` refers to the same memory location in all iterations (no aliasing).

But `await l2?` allows interleaving of tasks (of same object) which can modify the value of `size`.

```java
Unit p() { size = size - 2; }
```
Size Analysis for Concurrent Objects

Unit reqFile(Node s, FileName fname) {
...
    while (i > 0) {
        if (size > i) incr = i; else incr = size;
        i = i - incr;
        l2 = s ! getPacks(fname, incr, i);
        await l2?; ps = l2.get;
        f = app(ps, f);
    }
    ...
}

- By ignoring `await l2?`, field `size` refers to the same memory location in all iterations (no aliasing)

- But `await l2?` allows interleaving of tasks (of same object) which can modify the value of `size`

Unit p() { size = size - 2; }

Class invariant: `size = size_{init}`
Cost Centres in Recurrence Equations

- **Main idea:** recurrence equations use cost centres to keep the resource usage assigned to different components separate
- **Example:** all objects of the same class belong to the same cost centre (share the same processor)

```java
Unit reqFile(Node s, FileName fname) {
    c(Node)*O(1)+c(DB)*O(dbf_max)+
    l1 = s!lengthNode(fname); await l1? i = l1.get;
    while (i > 0) {
        O(dbf_max)*
        if (size > i) incr = i; else incr = size;
        i = i - incr;    c(Node)*O(1)+
        c(Node)*O(size_init*dbf_max)+c(DB)*O(dbf_max)+
        l2 = s ! get Packs(fname, incr, i);
        await l2?; ps = l2.get;
        f = app(ps, f);    c(Node)*O(dbf_max)+
    }
    db ! storeFile(fname, f);    c(DB)*O(1)
}
```
Cost Centres in Recurrence Equations

- **Main idea**: recurrence equations use *cost centres* to keep the resource usage assigned to different components separate.

- **Example**: all objects of the same class belong to the same cost centre (share the same processor).

```java
Unit reqFile(Node s, FileName fname) {
    c(Node)*O(1)+c(DB)*O(dbf_{max})+
    l1 = s!lengthNode(fname); await l1? i = l1.get;
    while (i > 0) {
        0(dbf_{max})*
        if (size > i) incr = i; else incr = size;
        i = i - incr;
        c(Node)*O(1)+
        c(Node)*O(size_{init}*dbf_{max})+c(DB)*O(dbf_{max})+
        l2 = s ! getPacks(fname, incr, i);
        await l2?; ps = l2.get;
        f = app(ps, f); c(Node)*O(dbf_{max})+
    }
    db ! storeFile(fname, f); c(DB)*O(1)}
```
Cost Centres in Recurrence Equations

- **Main idea**: recurrence equations use **cost centres** to keep the resource usage assigned to different components separate.

- **Example**: all objects of the same class belong to the same cost centre (share the same processor).

```plaintext
Unit reqFile(Node s, FileName fname) {
  c(Node)*O(1) + c(DB)*O(dbf_max) +
  l1 = s!lengthNode(fname); await l1? i = l1.get;
  while (i > 0) {
    O(dbf_max)*
    if (size > i) incr = i; else incr = size;
    i = i - incr;
    c(Node)*O(1) +
    c(Node)*O(size_init*dbf_max) + c(DB)*O(dbf_max) +
    l2 = s!getPacks(fname, incr, i);
    await l2?; ps = l2.get;
    f = app(ps, f); c(Node)*O(dbf_max) +
  }
  db ! storeFile(fname, f); c(DB)*O(1)
}
```

Tuesday 8 November 11
Cost Centres in Recurrence Equations

- **Main idea**: recurrence equations use *cost centres* to keep the resource usage assigned to different components separate

- **Example**: all objects of the same class belong to the same cost centre (share the same processor)

```c
Unit reqFile(Node s, FileName fname) {
    c(Node)*0(1)+c(DB)*O(dbf_max)+
    l1 = s!lengthNode(fname); await l1? i = l1.get;
    while (i > 0) {
        O(dbf_max)*
        if (size > i) incr = i; else incr = size;
        i = i - incr; c(Node)*O(1)+
        c(Node)*0(size_init*dbf_max)+c(DB)*O(dbf_max)+
        l2 = s!getPacks(fname, incr, i);
        await l2?; ps = l2.get;
        f = app(ps, f); c(Node)*O(dbf_max)+
    }
    db!storeFile(fname, f); c(DB)*O(1)
}
```
Cost Centres in Recurrence Equations

- **Main idea**: recurrence equations use **cost centres** to keep the resource usage assigned to different components separate.

- **Example**: all objects of the same class belong to the same cost centre (share the same processor).

```plaintext
Unit reqFile(Node s, FileName fname) {
  c(Node)*0(1) + c(DB)*O(dbf_max) +
  l1 = s!lengthNode(fname); await l1? i = l1.get;
  while (i > 0) {
    if (size > i) incr = i; else incr = size;
    i = i - incr;
    c(Node)*O(1) +
    c(Node)*0(size_init*dbf_max) + c(DB)*O(dbf_max) +
    l2 = s!getPacks(fname, incr, i);
    await l2?; ps = l2.get;
    f = app(ps, f); c(Node)*O(dbf_max) +
  }
  db! storeFile(fname, f); c(DB)*O(1)
}
```
Cost Centres in Recurrence Equations

- **Main idea**: recurrence equations use cost centres to keep the resource usage assigned to different components separate.
- **Example**: all objects of the same class belong to the same cost centre (share the same processor).

```plaintext
Unit reqFile(Node s, FileName fname) {
    c(Node)*O(1)+c(DB)*O(dbf_max) +
    l1 = s!lengthNode(fname); await l1? i = l1.get;
    while (i > 0) {
        O(dbf_max)*
        if (size > i) incr = i; else incr = size;
        i = i - incr; c(Node)*O(1)+
        c(Node)*O(size_init*dbf_max)+c(DB)*O(dbf_max)+
        l2 = s!getPacks(fname, incr, i);
        await l2?; ps = l2.get;
        f = app(ps, f); c(Node)*O(dbf_max)+
    }
    db!storeFile(fname, f); c(DB)*O(1)
}
```
Cost Centres in Recurrence Equations

- **Main idea**: recurrence equations use cost centres to keep the resource usage assigned to different components separate

- **Example**: all objects of the same class belong to the same cost centre (share the same processor)

```plaintext
Unit reqFile(Node s, FileName fname) {
    c(Node)*0(1) + c(DB)*O(dbf_{max}) + 
    l1 = s!lengthNode(fname); await l1? i = l1.get;
    while (i > 0) {
        O(dbf_{max})*
        if (size > i) incr = i; else incr = size;
        i = i - incr;  c(Node)*O(1) +
        c(Node)*0(size_{init}*dbf_{max}) + c(DB)*O(dbf_{max}) +
        l2 = s!getPacks(fname, incr, i);
        await l2?; ps = l2.get;
        f = app(ps, f);  c(Node)*O(dbf_{max}) +
    }
    db! storeFile(fname, f);  c(DB)*O(1)
}
```
Main idea: recurrence equations use cost centres to keep the resource usage assigned to different components separate.

Example: all objects of the same class belong to the same cost centre (share the same processor).

Unit reqFile(Node s, FileName fname) {
  c(Node)*0(1)+c(DB)*O(dbf_{max})+
  l1 = s!lengthNode(fname): await l1? i = l1.get;
  while (i > 0) {
    if (size > i) incr = i;
    else incr = size;
    i = i - incr;
    c(Node)*O(1)+
  } l2 = s!getPacks(fname, incr, i)
  await l2?; ps = l2.get;
  f = app(ps, f);  c(Node)*O(dbf_{max})+
} db ! storeFile(fname, f);  c(DB)*O(1)

Complexity per Cost Centre

DB: $O(dbf_{\text{max}}^2)$
Node: $O(dbf_{\text{max}}^2 + size_{init})$
Challenges
Problem: Legacy Product Lines

- Product line artefacts may already exist.
- New products are often created ad hoc, using suboptimal reuse mechanisms, such as duplication (clone-and-own).
- As number of products grows, ad hoc process stops scaling.
- The transition process is expensive.
Challenges

• How to extract variability information from existing systems?
  • What is the right level of abstraction?
  • What are the elements?
  • What are the relationships between the element?
• How to connect between development level variability and marketing-level terminology?
• How to identify modules and their commonalities and variabilities?
• How to refactor into a desired product structure?
Solution Approach

Model Mining

- Methods for variability extraction from “natural language” specifications
- Techniques for extracting variability from formal representations (code, requirement models, design models ...)
- Techniques for componentisation – identifying modules
- Techniques for building models from existing systems
Example: Feature Location

Locate code that is associated with a given feature

- Dynamic program analysis – given a test/scenario that activates the feature of interest, track all executed code
- Static program analysis – given a seed element, analyse program dependencies to find a set of related elements
- Information Retrieval – given a query that describes a feature of interest, analyse program elements based on their lexical similarity

- Useful for numerous software maintenance tasks, including PLE transformations
Problem 2: Test Optimisation

- Different variants of a product line are repeatedly tested to verify the same or similar functionality
- There is no leverage of PLE architecture for test design
- There is no leverage of PLE architecture for test optimisation
- Fraction of found bugs is low (or the testing effort is high)
Approaches

• Leverage knowledge about PLE commonalities and variabilities to improve test efficiency:

• Employ combinatorial testing techniques for reducing the set of tests that are to be executed

• Main idea: most defects come either from a single feature or from interactions of 2-3 feature

• Combinatorial testing techniques automatically create a minimal set of configurations that cover all n-wise interactions (n=1,2,3,...)
PLE Governance

Solving technical challenges is not enough for successful SPLE adoption

Organisational Challenges

• Structure project teams and cross organisation roles
• Relate to other methods and tools in the organisation
• Measure costs and benefits

Social Challenges

• Collaborate to increase SPLE efficiency
• Comply to SPLE approach
Formal Methods

Challenges

- Develop Truly Reusable Formal Methods
- Compositional Type Checking
- Compositional Verification
  - Verification at family level
  - Verification at feature level
Further information and Tools

www.hats-project.eu