Abstract Behavioral Specification of Distributed Object-Oriented Systems

Reiner Hähnle

Department of Computer Science, Software Engineering Group

HATS International School on
Formal Models for Components and Objects
University Residential Center of Bertinoro

24–28 September, 2012

http://www.hats-project.eu
Structure of this Talk

1. Introduction: The HATS Project
2. The ABS Modeling Language
3. Software Product Line Engineering
4. Tool Chain, Case Studies
Part I: Introduction

The HATS Project
HATS: Highly Adaptable & Trustworthy Software Using Formal Models

- FP7 FET focused call Forever Yours
- Project started 1 March 2009, 48 months runtime
- Integrated Project, academically driven
- 10 academic partners, 2 industrial research, 1 SME
- 8 countries
- 805 PM, EC contribution 5,64 M€ over 48 months
- web: www.hats-project.eu
In a nutshell, we ... 

develop a tool-supported formal modeling language (ABS) for the design, analysis, and implementation of highly adaptable software systems characterized by a 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 can’t describe software behavior with rigor: concurrency, compositionality, correctness, security, resource consumption . . .
- formalization ⇒ more advanced tools
  - more complex products
  - higher automation: cost-efficiency
Motivation

Why **formal**?

- informal notations can’t describe software behavior with rigor: concurrency, compositionality, correctness, security, resource consumption . . .
- formalization $\Rightarrow$ more advanced tools
  - more complex products
  - higher automation: cost-efficiency

Why **adaptable**?

- feature-rich software, deployment scenarios
- changing requirements (technology/market)
- evolution of software in unanticipated directions
- **language-supported adaptability** is a key to successful reuse
Mind the Gap!

Design-oriented, architectural, structural
UML, FDL, etc.

Implementation level
JML, SPEC#, etc.

Minimalistic foundational
$\pi$-calculus, ambient c., etc.
Mind the Gap!

Design-oriented, architectural, structural
UML, FDL, etc.

+ executability

Abstract Behavioural Specification
ABS

+ verifiability

Implementation level
JML, SPEC#, etc.

+ usability

Minimalistic foundational
π-calculus, ambient c., etc.

Realistic

Abstract

R. Hähnle
A Single-Source Technology for Highly Adaptive, Concurrent Software Systems

- bytecode
- UML sequence chart
- feature descr. lang.
- UML class diagram
- runtime components
- ABS
- PROSA specification
- Maude
- Scala
- Petri net
- UML Behavior tree
The Main Innovations of HATS

A formal, executable, abstract, behavioral modeling language

- Combines state-of-art in verification, concurrency, specification, and programming languages communities
- Adaptability drives the design

Scalable tools developed in tandem with ABS

- Incremental, compositional
- Analytic and generative, static and dynamic techniques

Formalization of PLE-based development as main application

- Seamless, formalized models from feature level to executable code
- Define FM-based development methodology for PLE
The HATS Approach

A tool-supported formal modeling language for building highly adaptable and trustworthy software
The HATS Approach

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

Main ingredients

1. Executable, formal modeling language for adaptable software: Abstract Behavioral Specification (ABS) language

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

   Develop methods in tandem with ABS to ensure scalability

3. Methodological, technological, and tool framework integrating HATS tool architecture and ABS language
Part II: ABS

The Abstract Behavioral Modeling Language (ABS)
Main Design Goals of ABS

ABS is designed with analysis/code generation tools in mind

- Expressivity carefully traded off with analysability
  - permit incremental/compositional static and dynamic analyses

- State-of-art programming language concepts
  - ADTs + functions + objects
  - type-safety, data race-freeness by design
  - modules, components
  - pluggable type systems, annotations

- Layered concurrency model
  - Upper tier: asynchronous, no shared state, actor-based
  - Lower tier: synchronous, shared state, cooperative multitasking

- Modeling of variability/deployment a first-class concept
  - feature models, delta-oriented programming
  - deployment components

- Not only code analysis, but also code generation/model mining
ABS Design Principles

- Uniform, **formal** semantics
- **Layered** architecture: simplicity, separation of concerns
- **Executability**: simulation, rapid prototyping, visualization
- **Abstraction**: underspecification, non-determinism
- Tool integration as Eclipse-plugin, easy to install & use
- Feature-based reuse realized as delta-oriented programming
- Formalization of product families

Out of scope for this lecture (but addressed in ABS):

- Real-time ABS with deployment components
  ⇒ Einar’s lecture!
- Code evolvability supported through runtime components
- Behavioural interface contracts
Layered ABS Language Design

<table>
<thead>
<tr>
<th>Core ABS</th>
<th>Full ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layered ABS Language Design</td>
<td></td>
</tr>
<tr>
<td>Delta Modeling Languages</td>
<td>Real-Time ABS Deployment Components</td>
</tr>
<tr>
<td>Behavioral Interface Specs</td>
<td></td>
</tr>
<tr>
<td>Local Contracts, Assertions</td>
<td></td>
</tr>
<tr>
<td>Syntactic Modules</td>
<td></td>
</tr>
<tr>
<td>Asynchronous Communication</td>
<td></td>
</tr>
<tr>
<td>Concurrent Object Groups (COGs)</td>
<td></td>
</tr>
<tr>
<td>Imperative Language</td>
<td></td>
</tr>
<tr>
<td>Object Model</td>
<td></td>
</tr>
<tr>
<td>Pure Functional Programs</td>
<td></td>
</tr>
<tr>
<td>Algebraic (Parametric) Data Types</td>
<td></td>
</tr>
</tbody>
</table>
### Layered ABS Language Design

<table>
<thead>
<tr>
<th>Delta Modeling Languages</th>
<th>Real-Time ABS Deployment Components</th>
<th>Runtime Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavioral Interface Specs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local Contracts, Assertions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Syntactic Modules</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asynchronous Communication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concurrent Object Groups (COGs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Imperative Language</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pure Functional Programs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algebraic (Parametric) Data Types</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Full ABS**

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

Built-In Data Types

```haskell
data Bool = True | False;
data Unit = Unit;  // for void return types
data Int;  // 4, 2323, −23
data String;  // "Hello World"
```
Built-In Data Types

```haskell
data Bool = True | False;
data Unit = Unit;  // for void return types
data Int;  // 4, 2323, −23
data String;  // ”Hello World”
```

Built-In Operators (Java-like Syntax)

- All types: `==` `!=`
- `Bool`: `~` `&&` `||`
- `Int`: `+` `−` `*` `/` `%` `<` `>` `<=` `>=`
- `String`: `+`

Construction of side effect-free operator expressions as usual
User-Defined Algebraic Data Types

User-Defined Data Types

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

Parametric Data Types

```haskell
data List<T> = Nil | Cons(T, List<T>);
// predefined
List<Int> l = [1,2,3];
// concrete list syntax
```

Type Synonyms

```haskell
type Saft = Juice;
type Fruits = List<Fruit>;
```
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>); // predefined
List<Int> l = [1,2,3]; // concrete list syntax
```
User-Defined Data Types

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

Parametric Data Types

```hs
data List<T> = Nil | Cons(T, List<T>);
List<Int> l = [1,2,3]; // concrete list syntax
```

Type Synonyms

```hs
type Saft = Juice;
type Fruits = List<Fruit>;
```
def Int length(IntList list) = // function names lower-case
  case list { // definition by case distinction and matching
    Nil => 0 ; // data constructor pattern
    Cons(n, ls) => 1 + length(ls) ;
      // data constructor pattern with pattern variable
    _ => 0 ; // underscore pattern (anonymous variable)
  } ;

def Int sign(Int n) =
  case n {
    0 => 0 ; // literal pattern
    n => if (n > 0) then 1 else -1 ; // bound variable pattern
  } ;

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

If no pattern in a case expression matches, a runtime error results
Implicit Selector Functions

```haskell
data File = File(String path, Int content) ;
// implicitly defines selector functions:
def String path(File f) = ... ;
def Int content(File f) = ... ;
```
Syntactic Sugar for Functions

Implicit Selector Functions

```haskell
data File = File(String path, Int content) ;
// implicitly defines selector functions:
def String path(File f) = ... ;
def Int content(File f) = ... ;
```

N-Ary Constructors for Associative Collection Types

```haskell
data Set<A> = EmptySet | Insert(A, Set<A>);
def Set<A> set<A>(List<A> l) = // convention: lower case type name
    case l {
        Nil => EmptySet;
        Cons(hd, tl) => Insert(hd, set(tl));
    } ;

Set<Int> s = set[1,2,3];
```
module Drinks; // upper-case, defines syntactic scope, part of type names
export Drink, Milk; // Water is not usable by other modules
import * from N; // import anything
data Drink = Water | Milk;
...
module Bar; // new module scope
import Drinks.Drink; // qualified import
import Milk from Drinks; // unqualified import
import Water from Drinks; // not allowed – compilation error
Module System (inspired by Haskell)

```haskell
module Drinks;  // upper-case, defines syntactic scope, part of type names
export Drink, Milk;  // Water is not usable by other modules
import * from N;  // import anything

data Drink = Water | Milk;
...

module Bar;  // new module scope
import Drinks.Drink;  // qualified import
import Milk from Drinks;  // unqualified import
import Water from Drinks;  // not allowed – compilation error
```

Abstract Data Types

```haskell
module Stack;  // Module as Abstract Data Type
export Stack<A>;  // type constructors are hidden, only functions usable

data Stack<A> = EmptyStack | push(A, Stack<A>);  // hidden
def Stack<A> createStack<A>(List<A> s) = ... ;  // usable
```
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) = ...
...
ECLIPSE ABS Perspective, Database.abs, F3, abslang.abs
## Layered ABS Language Design

<table>
<thead>
<tr>
<th>Delta Modeling Languages</th>
<th>Real-Time ABS Deployment Components</th>
<th>Runtime Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavioral Interface Specs</td>
<td>Local Contracts, Assertions</td>
<td>Core ABS</td>
</tr>
<tr>
<td>Syntactic Modules</td>
<td>Asynchronous Communication</td>
<td></td>
</tr>
<tr>
<td>Concurrent Object Groups (COGs)</td>
<td>Imperative Language</td>
<td></td>
</tr>
<tr>
<td>Object Model</td>
<td>Pure Functional Programs</td>
<td></td>
</tr>
<tr>
<td>Algebraic (Parametric) Data Types</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Full ABS**

**Core ABS**
Interfaces

- Provide reference types of objects (implementation abstraction)
- Subinterfaces allowed
- Multiple inheritance allowed
- JAVA-like syntax
- Reference types may occur in data types, but: no method calls in function definitions (possible side effects)

```java
interface Baz { ... }
interface Bar extends Baz {
    // method signatures
    Unit m();
    Bool foo(Bool b);
}
data BarList = Nil | Cons(Bar, BarList);
```
Classes

- Only for object construction
- Class name is not a type
- Classes can implement several interfaces
- No code inheritance (instead delta-oriented programming is used)

```java
// class declaration with parameters, implicitly defines constructor
class Foo(T x, U y) implements Bar, Baz {
    // field declarations
    Bool flag = False; // primitive type field initialization mandatory
    U g; // object type field initialization optional
    { // optional class initialization block
        g = y;
    }
    Unit m() { } // method implementations
    Bool foo(Bool b) { return ~b; }
}
```
Names of Interfaces and Classes

Suggested Naming Convention

- Class names only used at object creation
- All type declarations strictly to interfaces

Use e.g. `Name` for interface and `NameImpl` for implementing class

Modules Revisited

Type names and class names can be (and must be) imported/exported
Class Initialization

▶ Optional class parameters = fields = constructor signature
▶ Fields with primitive types must be initialized when declared
▶ Optional `init` block executed first

Active Classes

▶ Characterized by presence of `run()` method
▶ Objects from **active classes** start activity after initialization
▶ Passive classes react only to incoming calls

```java
Unit run() {
    // active behavior ...
}
```
Imperative Constructs

Sequential Control Flow

Loop while (x) { ... }
Conditional if (x == y) { ... } [else { ... }]
Synchronous method call x.m()

Local State Update and Access (Assignment)

Object creation new Car(Blue);
Field read x = [this.]f; (only on this object)
Field assignment [this.]f = 5; (only on this object)

Strong Encapsulation

ABS fields are object private (not merely class private as JAVA)
- Field access of a different object only via getter/setter methods
Other Statements

Skip

Do Nothing  skip;

Return

Return value  return  PureExpr;

Must be last statement of method body

- Standard idiom: collect return value in result variable

Expressions as Statement

Execute for side effect  Expr;
ABS is a Block-structured Language

Blocks appear as:
- Statement
- Method body
- Optional class initialization block (between field and method decls.)
- Optional implicit “main” method at end of module
  - serves as starting point for program execution
  - at least one main block necessary for executing a program
  - module with main block selectable as execution target

Blocks
- Sequence of variable declarations and statements
- Data type variables are initialized, reference types default to \texttt{null}
- Statements in block are \texttt{scope} for declared variables
ECLIPSE ABS Perspective, F4, Account.abs
## Layered ABS Language Design

<table>
<thead>
<tr>
<th>Delta Modeling Languages</th>
<th>Real-Time ABS Deployment Components</th>
<th>Runtime Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavioral Interface Specs</td>
<td>Local Contracts, Assertions</td>
<td>Syntactic Modules</td>
</tr>
<tr>
<td>Asynchronous Communication</td>
<td>Concurrent Object Groups (COGs)</td>
<td>Imperative Language</td>
</tr>
<tr>
<td>Object Model</td>
<td>Pure Functional Programs</td>
<td>Algebraic (Parametric) Data Types</td>
</tr>
</tbody>
</table>

### Full ABS

### Core ABS
Concurrent Object Groups (COGs)

- Unit of distribution
- Own heap of objects
- Cooperative multitasking inside COGs
  - One processor, several tasks
  - Intra-group communication by synchronous/asynchronous method calls
  - Multiple tasks originating from asynchronous calls within COG
- Inter-group communication only via asynchronous method calls
Cooperative Multitasking inside COGs

Multitasking
- A COG can have multiple tasks
- Only one is active, all others are suspended
- Asynchronous calls create new tasks
- Synchronous calls block caller thread
  - Java-like syntax: `target.methodName(arg1, arg2, ...)`

Scheduling
- Cooperative by special scheduling statements
  - Explicit decision of modeller
- Non-deterministic otherwise
  - User-defined configuration of schedulers via annotations
Local Object Creation

- `this:A`
Local Object Creation

```java
this:A
new B();
```
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
Pluggable Type and Inference System

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

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

- Most annotations can be inferred automatically (safe approximation)
Asynchronous Method Calls

- Syntax: `target ! methodName(arg1, arg2, ...)`
- Sends an asynchronous message to the target object
- Creates new task in COG of target
- Caller continues execution and allocates a `future` to store the result
  - `Fut<T> v = o!m(e);`
### Scheduling

#### Unconditional Scheduling
- `suspend` command yields control to other task in COG
- Unconditional scheduling point

#### Conditional Scheduling
- `await g`, where `g` is a polling guard
- Guards are monotonically behaving expression, inductively defined as:
  - `b` - where `b` is a side-effect-free boolean expression
  - `f?` - future guards
  - `g & g` - conjunction (not Boolean operator)
- Yields task execution until guard is true
  (continue immediately if guard is true already)
Synchronization and Blocking

Reading Futures

- \( f\.get \) - reads future \( f \) and blocks execution until result is available
- Deadlocks possible (use static analyzer for detection)
- Programming idiom: use \( \text{await } f? \) to prevent blocking (safe access)
  - \( \text{Fut}<T> \ v = o!m(e); \ldots; \text{await } v?; \ r = v\.get; \)

Blocking vs. Suspension

**Suspension** lets other task in same COG continue (if any)
**Blocking** no task in same COG can continue until future resolved
Summary: ABS Concurrency Model

Method calls with shared heap access encapsulated in COGs

COG
- One activity at a time
- One lock
- Cooperative scheduling
- Callbacks (recursion) ok
- Shared access to data

```java
this:A new B();
this:A b:B
```
Method calls with shared heap access encapsulated in COGs

This: A

new B();

this: A  b: B
Summary: ABS Concurrency Model

Distributed computation: async. calls/message passing/separate heap

COG

COG′

COG″

no reentrance in same thread

asynchr. message passing

new cog B();

this:A b:B

this:A
Driver.abs, Far/Near, Run as Java Backend (Debug)
Annotations and Assertions

**Assertions**

```assert PureExpr;```
To be used for runtime assertion checking or during formal verification

**Annotations**

Methods (and classes, interfaces) can carry **annotations**:

- Local contracts, including invariants
- Type annotations before any declaration

```Predefined [LocationType: Near] T n; or briefly [Near] T n;```

**Meta annotation**

```[TypeAnnoptation] // declares data type as type annotation```

```data LocationType = Far | Near | Somewhere | Infer;```

**Easy realization of pluggable type systems**

- Code instrumentation (e.g., resources, time bounds) ⇒ Einar’s lecture
Interaction with Environment

ABS has no I/O (implementation-independent)
Interaction with Environment

ABS has no I/O (implementation-independent)

Foreign Language Interface, ABS side

```
import * from ABS.FLI;

interface Hi { String hello(String msg); }

[Foreign]
class HiImpl implements Hi {
    String hi(String msg) { return "default implementation"; }
}
```

Foreign Language Interface, JAVA side

- Extend JAVA class HiImpl generated by ABS with desired behavior
- In JAVA use types ABSString, etc., in abs.backend.java.lib.types.*
SOS-style Semantics

Reduction rules on ABS runtime configurations

```
object  ||  task  ||  lock
o[b, C, σ]  ||  n⟨b, o, σ, s⟩  ||  b[l]  ||  ...
```

class  state  PC  COG ⊤/⊥

Task id is used as future
SOS-style Semantics

Reduction rules on ABS runtime configurations

Task id is used as future

A Typical Reduction Rule: creation of a new COG

\[ n(b, o, \sigma, T \{ z = \text{new} \ cog \ C(v) \}; s) \]
SOS-style Semantics

Reduction rules on ABS runtime configurations

Object
\[ o[b, C, \sigma] \]

Task
\[ n(b, o, \sigma, s) \]

Lock
\[ b[l] \]

Class
\[ \text{class} \]

State
\[ \text{state} \]

PC
\[ \text{PC} \]

COG
\[ \top / \bot \]

Task id is used as future

A Typical Reduction Rule: creation of a new COG
\[ n(b, o, \sigma, \top \ z = \text{new cog } C(v); s) \rightarrow b'(
\top) \]
Formal Semantics

SOS-style Semantics

Reduction rules on ABS runtime configurations

object \[ \triangleright o[b, C, \sigma] \]

\begin{itemize}
  \item task \[ \triangleright n\langle b, o, \sigma, s \rangle \]
  \item lock \[ \triangleright b[l] \]
\end{itemize}

class \quad state \quad PC \quad COG \text{ } \top/\bot

Task id is used as future

A Typical Reduction Rule: creation of a new COG

\[ n\langle b, o, \sigma, \top \rangle z = \textbf{new} \text{ } \text{cog} \text{ } C(v) ; s \rangle \rightarrow \]

\[ b'(\top) \triangleright n'\langle b', o', \sigma'_{\text{init}}, s_{\text{task}} \rangle \]
Formal Semantics

SOS-style Semantics

Reduction rules on ABS runtime configurations

\[
\begin{align*}
\text{o}[b, C, \sigma] & \quad || \quad n\langle b, o, \sigma, s \rangle & \quad || \quad b[l] & \quad || \quad \cdots \\
\text{class} & \quad \downarrow & \quad \text{task} & \quad \downarrow & \quad \text{lock} & \quad \downarrow \\
\text{state} & \quad \text{PC} & \quad \text{COG} & \quad \top/\bot
\end{align*}
\]

Task id is used as future

A Typical Reduction Rule: creation of a new COG

\[
n\langle b, o, \sigma, T \ z = \text{new} \ cog \ C(v); s \rangle \rightarrow \\
b'(T) \ || \ n'\langle b', o', \sigma_{\text{init}}, s_{\text{task}} \rangle \ || \ o'[b', C, \sigma_{\text{init}}]
\]
Formal Semantics

SOS-style Semantics

Reduction rules on ABS runtime configurations

A Typical Reduction Rule: creation of a new COG

\[ n\langle b, o, \sigma, T \rangle \ z = \text{new cog} \ C(v) ; s \rightarrow b'(T) \ || \ n'\langle b', o', \sigma'_{\text{init}}, s_{\text{task}} \rangle \ || \ o'[b', C, \sigma_{\text{init}}] \ || \ n\langle b, o, \sigma, s\{z/o'\} \rangle \]
Formal Semantics

**SOS-style Semantics**
Reduction rules on ABS runtime configurations

\[
\begin{align*}
\text{object} & \quad \downarrow & \text{task} & \quad \downarrow & \text{lock} & \quad \downarrow \\
o[b, C, \sigma] & \quad || & n\langle b, o, \sigma, s \rangle & \quad || & b[l] & \quad || \quad \ldots \\
\text{class} & \quad \downarrow & \text{state} & \quad \downarrow & \text{PC} & \quad \downarrow & \text{COG} & \top/\bot
\end{align*}
\]

Task id is used as future

**A Typical Reduction Rule: creation of a new COG**

\[
\begin{align*}
n\langle b, o, \sigma, T \rangle \ x = \textbf{new} \ cog \ C(v) ; s \rangle \rightarrow \\
b'(\top) \ || \ n '\langle b', o', \sigma'_{init}, s_{task} \rangle \ || \ o'[b', C, \sigma_{init}] \ || \ n\langle b, o, \sigma, s\{z/o'\} \rangle \\
b', o', n', \text{new; } \overline{T}f; \ s' \text{ init block } C; \ \sigma_{init} = \overline{T}f; \ s_{task} = s'\{\text{this}/o'; \text{suspend}\}
\end{align*}
\]
Feature Modelling and Software Product Lines
Background: Software Product Line Engineering

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

Feature Selection → Family Engineering
Vision: A Model-Centric Development Method for PLE

Product Line Models expressed in HATS ABS with uniform formal semantics

Family Engineering

consistency analysis

correctness of reuse

family visualization

test case generation

validation, verification

family evolution

rapid prototyping

code generation

product visualization

test case generation

validation, verification

product evolution

Application Engineering

[Schaefer & Hähnle, IEEE Computer, Feb. 2011]
Modelling Variability with ABS

Core ABS +

1. Feature Model documents variability abstractly
2. Delta Modules define units of behaviour
3. Configuration connects features to behaviour
4. Product Selection specifies deployment configurations
The Case for Feature-Oriented Programming

Feature Hierarchy often incompatible with Class Hierarchy

- Modern SW development (not just SPFE) often feature-driven
- Mismatch between artefacts created in analysis vs. coding phases
- Results in brittle/awkward class hierarchies
- “Built-in” disconnect between analysts and implementors
Feature Hierarchy often incompatible with Class Hierarchy

- Modern SW development (not just SPFE) often feature-driven
- Mismatch between artefacts created in analysis vs. coding phases
- Results in brittle/awkward class hierarchies
- “Built-in” disconnect between analysts and implementors

Achieve separation of concerns: hierarchy vs. functionality of features
Feature Modelling by Example
Feature Diagram of Account Example

Account

Type

- Int interest
- Check
  - interest=0

Overdraft

- Int amount in [0..5]
  - excludes

Fee

- Int amount in [0..5]
  - excludes
- Save
  - interest>0
Feature Modelling Language $\mu$TVL

$\mu$TVL: micro Textual Variability Language

Subset of TVL [Classen et al., SoCP 76(12):1130–1143, 2011]

- Attributes: only integers and booleans (no reals, enumerated types)
- Feature extensions: only additional constraints
- But: Multiple roots for orthogonal variability
root Account {
  group allof {
    Type {
      group oneof {
        Check {ifin: Type.i == 0;},
        Save {ifin: Type.i > 0;
          exclude: Overdraft;}
      }
      Int i; // interest rate of account
    },
    opt Fee {Int amount in [0..5];},
    opt Overdraft
  }
}
Straightforward translation to Boolean/Integer constraint formula

\[0 \leq \text{Account} \leq 1 \land \]
\[\text{Type} \rightarrow \text{Account} \land \]
\[\text{Overdraft}^\dagger \rightarrow \text{Account} \land \]
\[\text{Fee}^\dagger \rightarrow \text{Account} \land \]
\[\text{Type} + \text{Fee}^\dagger + \text{Overdraft}^\dagger = 3 \land \]
\[0 \leq \text{Type} \leq 1 \land \]
\[\text{Check} \rightarrow \text{Type} \land \text{Save} \rightarrow \text{Type} \land \text{Save} \rightarrow \neg \text{Overdraft} \land \]
\[\text{Check} + \text{Save} = 1 \land \]
\[0 \leq \text{Check} \leq 1 \land 0 \leq \text{Save} \leq 1 \land 0 \leq \text{Fee}^\dagger \leq 1 \land 0 \leq \text{Overdraft}^\dagger \leq 1 \land \]
\[\text{Fee} \rightarrow \text{Fee}^\dagger \land \text{Overdraft} \rightarrow \text{Overdraft}^\dagger \land \]
\[0 \leq \text{Save} \leq 1 \land 0 \leq \text{Check} \leq 1 \land \]
\[\text{Fee} \rightarrow (\text{Fee}.\text{amount} \geq 0 \land \text{Fee}.\text{amount} \leq 5) \land \]
\[\text{Check} \rightarrow (\text{Type}.i = 0) \land \text{Save} \rightarrow (\text{Type}.i > 0).\]
ABS constraint solver can:

- find solutions for a feature model
- check whether a product selection is a solution of a feature model

\[ \text{Product Selection} \models \boxed{\text{Feature Model}} \]
Delta Modelling

A software reuse mechanism aligned to feature-driven development

- No subclassing, only subtyping (no extends, only implements)
- No traits, mixins, ...
Delta Modelling

A software reuse mechanism aligned to feature-driven development

- No subclassing, only subtyping (no extends, only implements)
- No traits, mixins, ...

Instead: Delta Modelling

- Base product (the core) with minimal functionality
- Variants (products) obtained by applying deltas to the base product
Application of Delta Modules

- Delta modules add, remove or modify classes
- Class modifications: add, remove or wrap fields and methods, add new interfaces, ...
- Granularity of deltas at the method level
- Compiler checks applicability of deltas and generates resulting product
module Account;

interface Account {
   Int deposit(Int x);
}

class AccountImpl(Int aid, Int balance) implements Account {
   Int deposit(Int x) {
      balance = balance + x;
      return balance;
   }
}

delta DFee (Int fee);  // Implements feature Fee
uses Account;
modifies class AccountImpl {
    modifies Int deposit(Int x) {
        Int result = x;
        if (x>=fee) result = original(x-fee);
        return result;
    }
}

delta DSave (Int i);  // Implements feature Save
uses Account;
modifies class AccountImpl {
    removes Int interest;  // field removed & added with new initial value
    adds Int interest = i;  // modification of init blocks not supported
}
Application of Delta Modules

class AccountImpl(Int aid, Int balance) implements Account {
    Int interest = 0;
    ... }

delta DSave(3);
modifies class AccountImpl {
    removes Int interest;
    adds Int interest = 3; }

class AccountImpl(Int aid, Int balance) implements Account {
    Int interest = 3;
    ... }
Two models: Feature Model and Delta Model (feature implementation)

How are they connected?
Product Line Configuration

Two models: Feature Model and Delta Model (feature implementation)

How are they connected? Product Configuration Language:

- application conditions to associate features and delta modules
- temporal delta ordering (partial)
- feature attribute value passing to delta modules
Product Line Configuration Example

```plaintext
productline Accounts;
features Type, Fee, Overdraft, Check, Save;

delta DType (Type.i) when Type;

delta DFee (Fee.amount) when Fee;

delta DOverdraft after DCheck when Overdraft;

delta DSave (Type.i) after DType when Save;

delta DCheck after DType when Check;
```

- application condition (ensure suitable feature implementation)
- feature attribute value passing
- order of delta application (conflict resolution)
Product Selection

Feature Model

ensures satisfaction

Configuration

Product Selection
Compiler flattens delta and core modules into core ABS model
Examples of Product Selection

// basic product
product CheckingAccount (Type{i=0},Check);

// Account with Fee and parameter
product AccountWithFee (Type{i=0},Check,Fee{amount=1});

// should be refused
product SavingWithOverdraft (Type{i=1},Save,Overdraft);
Run Configurations
Choice of Modeling Level

Application

Product variability

Architecture

Concurrency, distribution

Functionality

Data structures
Choice of Modeling Level

Application → JAVA

- Product variability
- Architecture
- Concurrency, distribution
- Functionality
- Data structures
Choice of Modeling Level

Application --> JAVA

Product variability

Architecture

Concurrency, distribution

Functionality --> Classes

Data structures
Choice of Modeling Level

Application → JAVA

Product variability → Classes

Architecture

Concurrency, distribution

Functionality → Classes

Data structures
Choice of Modeling Level

- Application → JAVA
  - Product variability → Classes
  - Architecture → Classes
  - Concurrency, distribution
  - Functionality → Classes
  - Data structures
Choice of Modeling Level

- **Application**
  - Product variability
  - Architecture
  - Concurrency, distribution
  - Functionality

- **JAVA**
  - Classes

- **Data structures**
Choice of Modeling Level

Application → JAVA

Product variability → Classes

Architecture → Classes

Concurrency, distribution → Classes

Functionality → Classes

Data structures → Classes
Choice of Modeling Level

- Application
- Product variability
- Architecture
- Concurrency, distribution
- Functionality
- Data structures

ABS
Choice of Modeling Level

Application → ABS

Product variability → Delta Modeling

Architecture

Concurrency, distribution

Functionality

Data structures
Choice of Modeling Level

- Application
- Product variability
- Architecture
- Concurrency, distribution
- Functionality
- Data structures

ABS

Delta Modeling

Components, Modules
Choice of Modeling Level

- Application → ABS
  - Product variability → Delta Modeling
  - Architecture → Components, Modules
    - Concurrency, distribution → Actors, COGs
  - Functionality
  - Data structures
Choice of Modeling Level

- Application
  - Product variability
  - Architecture
  - Concurrency, distribution
  - Functionality
  - Data structures

- ABS
  - Delta Modeling
  - Components, Modules
  - Actors
  - COGs
  - Classes
Choice of Modeling Level

Application

Product variability

Architecture

Concurrency, distribution

Functionality

Data structures

ABS

Delta Modeling

Components, Modules

Actors

COGs

Classes

Algebraic Data Types
Choice of Modeling Level

- Application
- Product variability
- Architecture
- Concurrency, distribution
- Functionality
- Data structures

ABS

Delta Modeling

Components, Modules

Actors

COGs

Classes

Algebraic Data Types
Choice of Modeling Level

Application → ABS

Product variability → Delta Modeling

Architecture → Components, Modules

Concurrency, distribution → Actors, COGs

Functionality → Classes

Data structures → Algebraic Data Types
Choice of Modeling Level

Application → ABS

Product variability → Delta Modeling

Architecture → Components, Modules

Concurrency, distribution → Actors, COGs

Functionality → Classes

Data structures → Algebraic Data Types
State of Implementation
The ABS Basic Tool Chain

Feature models, Product selections, Configurations, Delta modules, Core ABS code

Parser

Extended AST

Rewriter

Core AST

Name Resolution

Resolved AST

Type Checker

Emacs Mode

ABS Integrated Development Environment

Type-Checked AST

Maude Back End

Java Back End

Core ABS code gen.

Maude Files

Java Files

Core ABS Files

Maude VM

Java VM

Legend

external data

internal data

ABS tool

existing tool

R. Hähnle
HATS-FMCO Bertinoro
Capabilities of the ABS Tool Set

- ABS IDE (Eclipse-based), parser, compiler, type checker
- Type-based far/near analysis immersed into IDE
- Java, Maude, Scala† code generation
- Execution visualization
- Behavioral verification
- Monitor inlining
- Runtime components†
- Deployment components with timing constraints
- A type system for feature models and deltas*
- Deadlock analysis†
- Automated resource (time, space) analysis ⇒ Elvira
- Automated test case generation
- Functional verification with a program logic (based on KeY)†

(† = under construction, * = under construction)
Beyond “Hello World”

Case Studies

- Trading System (CoCoME)
- Model of “Availability-to-Promise” functionality of SAP HANA DB
- Fredhopper Access Service (FAS) by SDL
  - part of replication system of e-commerce application
  - model based on Java code of actual product
  - runtime vs. simulation cost described by linear polynomial

![Graph showing running time and simulation cost across environments. The x-axis represents environments from 1 to 20, and the y-axis represents running time and simulation cost from 0 to 70, with specific values like 7500, 15000, 22500, and 30000 shown. The graph uses different colors and bar lengths to represent model simulation cost and implementation running time.]
Further Reading

ABS: A core language for abstract behavioral specification.
In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, Proc. 9th

D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer,
R. Schlatte, and P. Y. H. Wong.
Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral
Modeling Language.
In M. Bernardo and V. Issarny, editors, Formal Methods for Eternal Networked
Software Systems, volume 6659 of Lecture Notes in Computer Science, pages

R. Hähnle, M. Helvensteijn, E. B. Johnsen, M. Lienhardt, D. Sangiorgi, I. Schaefer,
and P. Y. H. Wong.
HATS abstract behavioral specification: the architectural view.
In B. Beckert, F. Damiani, and D. Gurov, editors, Proc. 10th International
Symposium on Formal Methods for Components and Objects (FMCO 2011),

... + the tutorial that will be written based on my lectures
Case studies, papers available at http://www.hats-project.eu/

Documentation, code available at http://tools.hats-project.eu/
ABS, a language for abstract modeling of realistic systems
- Designed with analysability in mind
- Integrated tool set for design, analysis, and generation of artefacts
- Delta-oriented programming as feature-based reuse principle
- Formalization of product lines relates features and their implementation