Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language

Ina Schaefer

Technische Universität Braunschweig, Germany

18 June 2011

http://www.hats-project.eu

Acknowledgment

The following HATerS contributed to this tutorial:

- Richard Bubel (Chalmers UT)
- Jan Schäfer (TU Kaiserslautern)
- Reiner Hähnle (Chalmers UT)
- Dave Clarke (KU Leuven)
- Einar Broch Johnson (U Oslo)
- Rudi Schlatte (U Oslo)
- Radu Muschevici (KU Leuven)

HATS Facts

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
- 9 academic partners, 2 industrial research, 1 SME
- 8 countries
- 805 PM, EC contribution 5,64 M€ over 48 months
- web: `www.hats-project.eu`

What Does HATS?

In a nutshell, we . . .

develop a tool-supported formal method

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, modularity, correctness, security, resources . . .
- formalization $\Rightarrow$ 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 a key to successful reuse

Mind the Gap!

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

Specification level	Languages (examples)
Design-oriented, architectural	UML, FDL, ALs
$\updownarrow$? $\updownarrow$	$\updownarrow$? $\updownarrow$
Implementation-oriented	Spec#, Java+JML

Mind the Gap!

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

Specification level	Languages (examples)
Design-oriented, architectural	UML, FDL, ALs
↕	↕
Abstract behavioral	HATS ABS language
↕	↕
Implementation-oriented	Spec#, Java+JML

How?

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

How?

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

Main ingredients

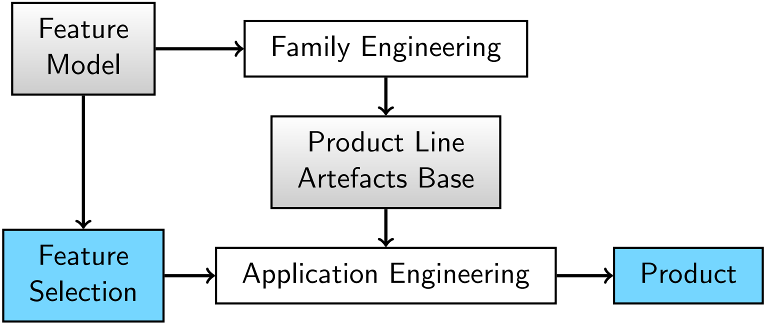
1. Executable, formal modeling language for adaptable software: Abstract Behavioral Specification (ABS) language
2. Tool suite for ABS/executable code analysis & development:
   - **Analytic** functional/behavioral verification, resource analysis, feature consistency, RAC, types, TCG, visualization
   - **Generative** code generation, model mining, monitor inlining, ...

   Develop methods in tandem with ABS to ensure feasibility
3. Methodological and technological framework integrating HATS tool architecture and ABS language

Important Project Principles (I)

Ensuring relevance

- Apply to empirically highly successful development method: Software product line engineering(PLE)
- Thorough requirements analysis, continuous evaluation



Important Project Principles (II)

Feasibility: ensure that analysis methods scale up

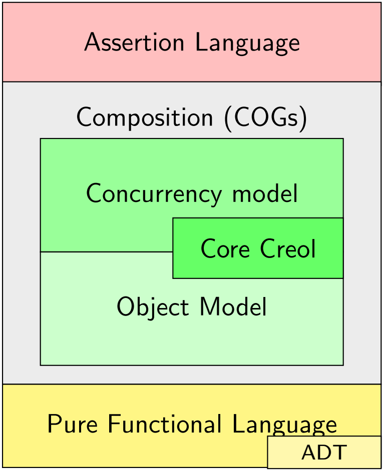
Develop analysis methods in tandem with ABS language

- Incrementality
  - Delta modeling, delta specification, delta verification
- Compositionality
  - Concurrency model
  - Proof systems

Important Project Principles (III)

Early evaluation

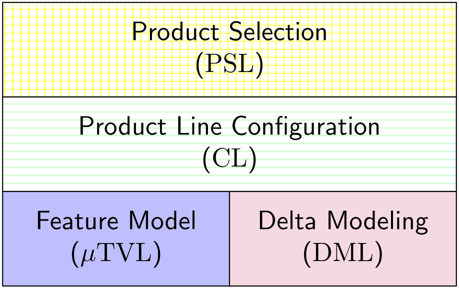
- Develop Core ABS first

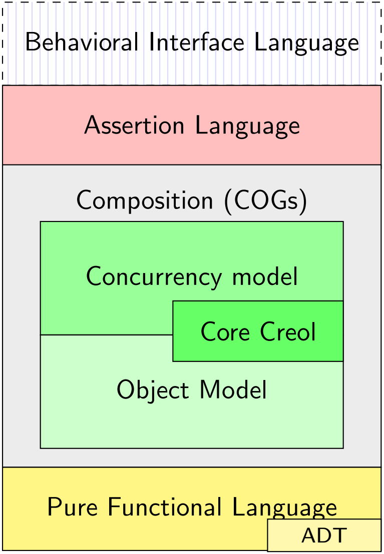


Important Project Principles (III)

Early evaluation

- Develop Core ABS first
- Layered language design

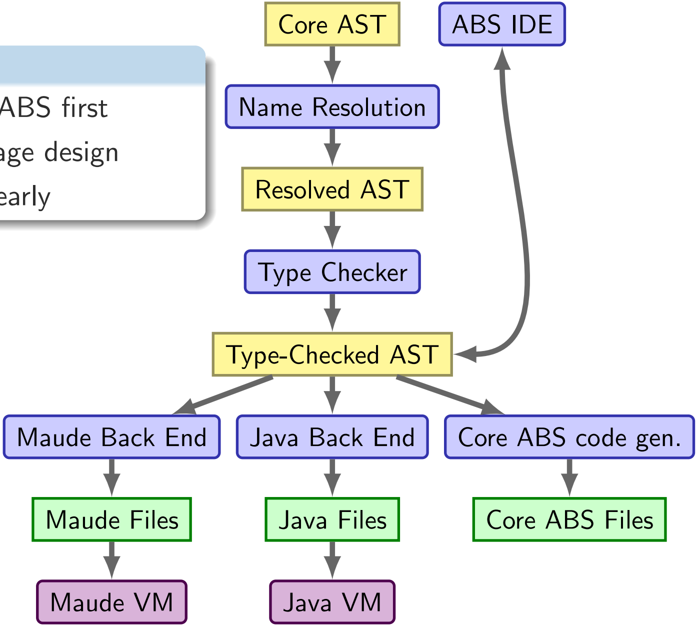




Important Project Principles (III)

Early evaluation

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



The Main Innovations of HATS

A formal, executable, abstract, behavioral modeling language

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

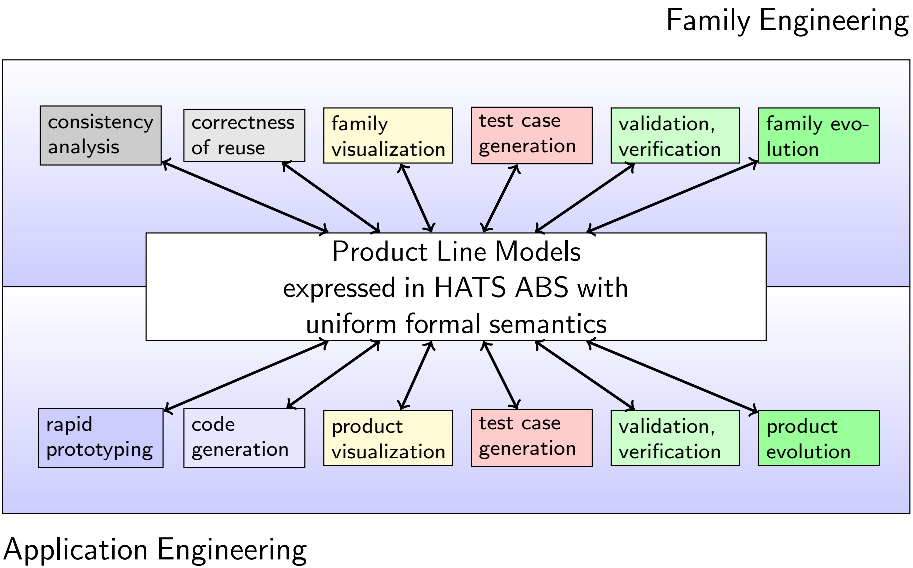
Scalable technologies developed in tandem with ABS

- Incremental, compositional
- Analytic as well as generative technologies

Formalization of PLE-based development as main application

- Leveraging formal methods tools to PLE
- Define FM-based development methodology for PLE

Vision: a Model-Centric Development Method for PLE



[Schaefer & Hähnle, IEEE Computer, Feb. 2011]

Main Design Goals of ABS

ABS

A language for describing large, distributed information system families

Key Properties

- Object-based, imperative, and functional
- Sequential, concurrent, and distributed
- Expressive yet analyzable
- Formal yet practical

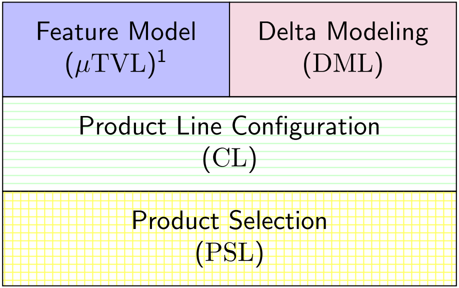
Suitable for

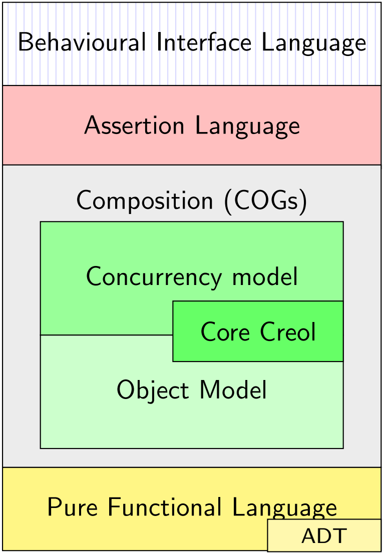
- Static analysis
- Dynamic analysis
- Simulation
- Code generation

Outline

- Modeling Concurrent Systems with Core ABS
- Modeling Spatial Variability in Full ABS
- Modeling Temporal Variability in Full ABS

Layered ABS Language Design





[1]Based on: A. Classen, Q. Boucher, P. Heymans. A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL. SCP 2010.

Core ABS

Built-In Data Types and Operators

Built-In Data Types

```
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

```
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

User-Defined Data Types

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

User Defined Data Types

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

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

Parametric Data Types

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

Optional Selectors (since v1.1)

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

implicitly defines corresponding functions, e.g.,

```
def String name(Person) = ... ;
```

Functions and Pattern Matching

```
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 ; // anonymous variable matches anything
  } ;
```

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

ABS Standard Library

ABS Standard Library

```
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: Interfaces

Interfaces

- Types of objects
- Multiple inheritance

```
interface Baz { ... }
interface Bar extends Baz {
  // method signatures
  Unit m();
  Bool foo(Bool b);
}
```

Object Model: Classes

Classes

- Only for object construction
- No type
- No inheritance

```
// class parameters
class Foo(T x, U y) implements Bar, Baz {
  // fields
  Bool flag = False;
  U g;
  { // optional initialization block
    g = y;
  }
  Unit m() { }  // method implementations
  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 = 5;`) (only on `this`)

Concurrency Model

Layered Concurrency Model

Upper tier: asynchronous, no shared state, actor-based

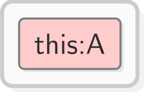
Lower tier: synchronous, shared state, cooperative multitasking

Concurrent Object Groups (COGs)

- Unit of distribution
- Own heap of objects
- Communicate by asynchronous method calls
- Cooperative multitasking inside COGs

Object and COG Creation

Local Object Creation

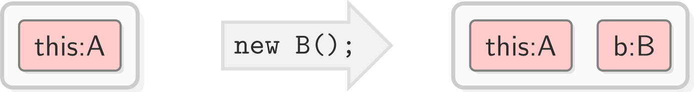


Object and COG Creation

Local Object Creation

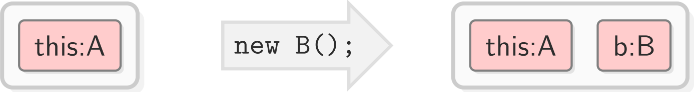
Object and COG Creation

Local Object Creation

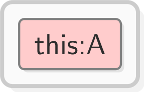


Object and COG Creation

Local Object Creation

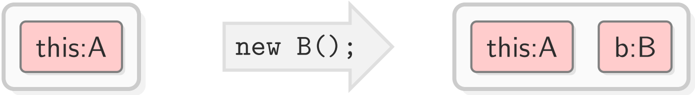


COG Creation



Object and COG Creation

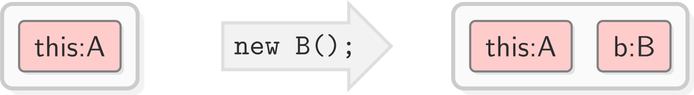
Local Object Creation



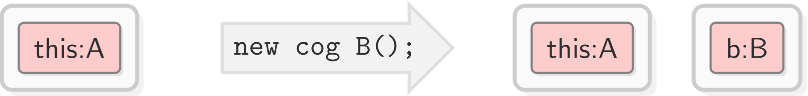
COG Creation

Object and COG Creation

Local Object Creation



COG Creation



Far and Near References

Far References

Refer to objects belonging to a different COG

Near References

Refer to objects belonging to the current COG

Far and Near References

Far References

Refer to objects belonging to a different COG

Near References

Refer to objects belonging to the current COG

Pluggable Type and Inference System

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

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

Asynchronous Method Calls

Asynchronous Method Calls

- Syntax: `target ! methodName(arg1, arg2, ...)`
- Sends an asynchronous message to the target object
- Caller continues and gets a future to the result
  - `Fut<T> v = o!m(e);`

Cooperative Multitasking inside COGs

Multitasking

- 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
  - Configuration of scheduling is worked on

Scheduling and Synchronization

Unconditional Scheduling

- `suspend` command yields control to other 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

Future Reading

- `f.get` - reads future `f` and blocks execution until is resolved
- Deadlocks possible
- Use `await f?` to prevent blocking, e.g.,
  - `Fut<T> v = o!m(e);...; await v?; r = v.get;`

Scheduling and Synchronization (2)

Synchronization of Concurrent Activities

- Wait until result of an asynchronous computation is ready
  - `await g`, where `g` is a monotonically behaving polling guard expression over `v?` and `v` is a future reference (has future type)
- Retrieve result of asynchronous computation and copy into a future
  - `v.get`, where `v` is a future referring to a finished task
- Programming idiom:

```
Fut<T> v = o!m(e);...; await v?; r = v.get;
```

- Conditional scheduling point

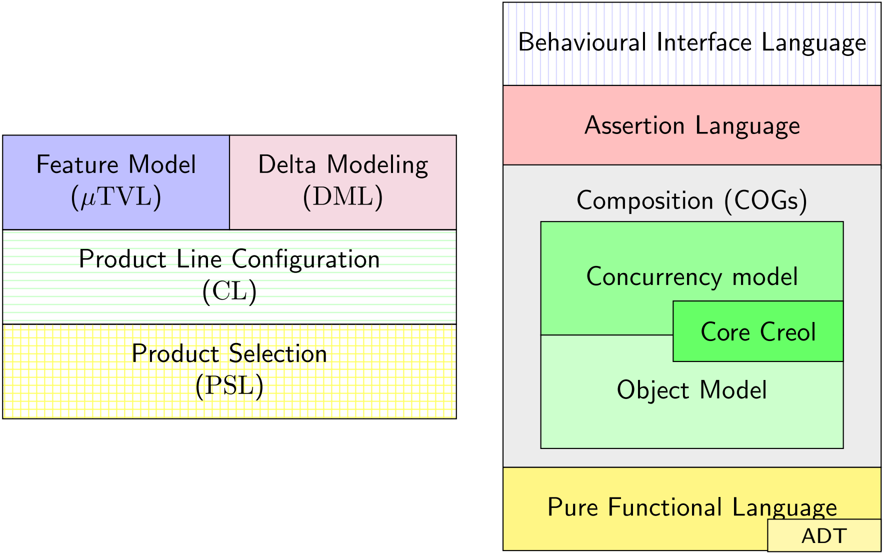
Tool Demo - Core ABS

- Eclipse-Plugin
- Type Checking
- Java Code Generation
- Simulation

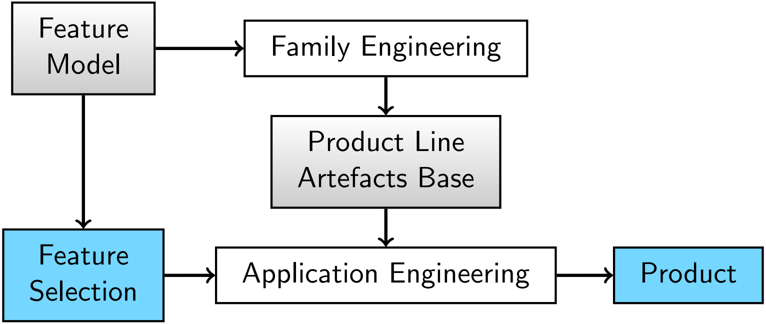
Tools are available at `http://tools.hats-project.eu/`

Modeling Spatial Variability

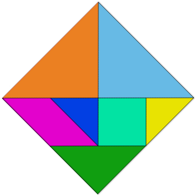
ABS Language Layers



Spatial Variability - Product Line Engineering

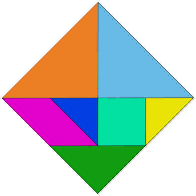


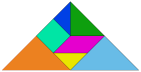
A product can be seen as selection of features.



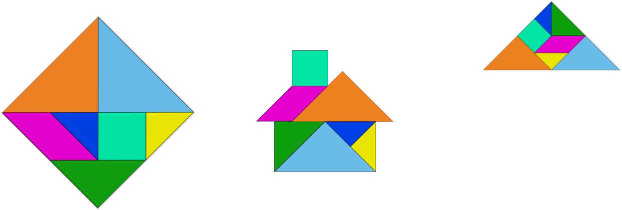
Variability Modelling: Feature Modelling

A product can be seen as selection of features.

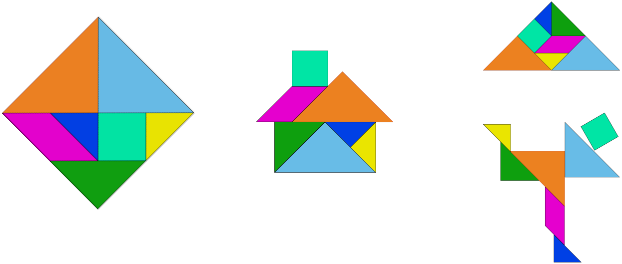




A product can be seen as selection of features.

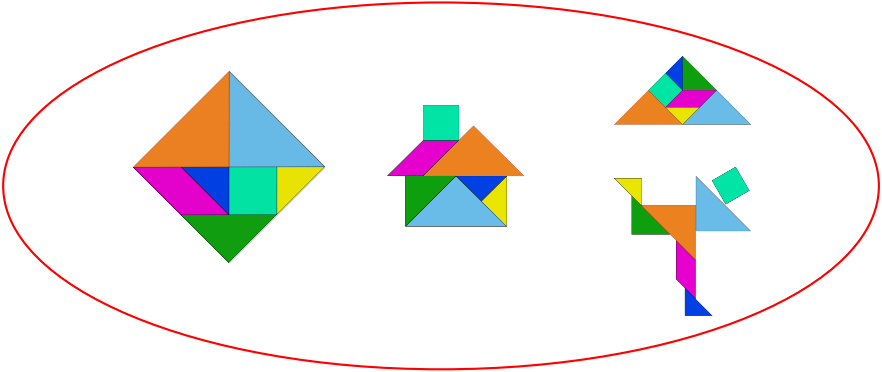


A product can be seen as selection of features.

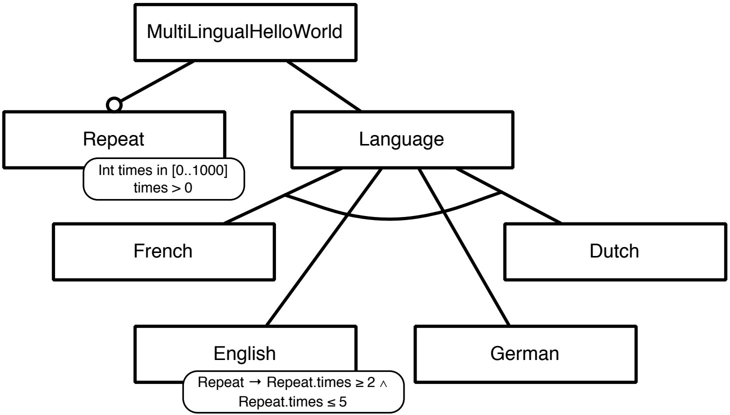


Variability Modelling: Feature Modelling

A product can be seen as selection of features.



Feature model describes all possible feature combination.


MultiLingualHelloWorld
Repeat
Int times in [0..1000]
times > 0
Language
French
Dutch
English
Repeat → Repeat.times ≥ 2 ∧
Repeat.times ≤ 5
German

Feature Modelling Language μTVL

μTVL: micro textual variability language

Extended subset of TVL

- Attributes: only integers (no enums)
- Feature extensions: only additional constraints
- But: Multiple roots for orthogonal features

Feature Modelling Language μTVL

μTVL: micro textual variability language

Extended subset of TVL

- Attributes: only integers (no enums)
- Feature extensions: only additional constraints
- But: Multiple roots for orthogonal features

Why?
Reduction of many semantical constraints in TVL to pure syntactical constraints.

Hello World Example

```
root MultiLingualHelloWorld {
  group allof {
    Language {
      group oneof { English, Dutch, French, German }
    },
    opt Repeat {
      Int times in [0..1000];
      times > 0;
    }
  }
}
extension English {
  ifin: Repeat ->
        (Repeat.times >= 2 && Repeat.times <= 5);
}
```

Grammar of μTVL

```
Model            ::=  (root FeatureDecl)* FeatureExtension*

FeatureDecl      ::=  FID [{ [Group] AttributeDecl* Constraint* }]
FeatureExtension ::=  extension FID { AttributeDecl* Constraint*}

Group            ::=  group Cardinality
                      { [opt] FeatureDecl , ([opt] FeatureDecl)* }
Cardinality      ::=  allof | oneof | [n1 .. *] | [n1 .. n2]
AttributeDecl    ::=  Int AID ; | Int AID in [ Limit .. Limit ] ;
                   |  Bool AID ;
Limit            ::=  n | *

Constraint       ::=  Expr ; | ifin: Expr ; | ifout: Expr ;
                   |  require: FID ; | exclude: FID ;
Expr             ::=  True | False | n | FID | AID | FID.AID
                   |  UnOp Expr | Expr BinOp Expr | ( Expr )
UnOp             ::=  ! | -
BinOp            ::=  || | && | -> | <-> | == | != | > | <
                   |  >= | <= | + | - | * | / | %
```

Hello World Example
Semantics

μTVL models translated into integer constraints.

$$
\begin{aligned}
&0 \leq \texttt{MultiLingualHelloWorld} \leq 1 \;\; \wedge \\
&\texttt{Language} \rightarrow \texttt{MultiLingualHelloWorld} \;\; \wedge \\
&\texttt{Repeat}^{\dagger} \rightarrow \texttt{MultiLingualHelloWorld} \;\; \wedge \\
&\texttt{Language} + \texttt{Repeat}^{\dagger} = 2 \;\; \wedge \\
&0 \leq \texttt{Language} \leq 1 \;\; \wedge \\
&\texttt{English} \rightarrow \texttt{Language} \;\; \wedge \;\; \texttt{Dutch} \rightarrow \texttt{Language} \;\; \wedge \;\; \texttt{German} \rightarrow \texttt{Language} \;\; \wedge \\
&1 \leq \texttt{English} + \texttt{Dutch} + \texttt{German} \leq 1 \;\; \wedge \\
&0 \leq \texttt{English} \leq 1 \;\; \wedge \;\; 0 \leq \texttt{Dutch} \leq 1 \;\; \wedge \;\; 0 \leq \texttt{German} \leq 1 \;\; \wedge \\
&0 \leq \texttt{Repeat}^{\dagger} \leq 1 \;\; \wedge \\
&\texttt{Repeat} \rightarrow \texttt{Repeat}^{\dagger} \;\; \wedge \\
&0 \leq \texttt{Repeat} \leq 1 \;\; \wedge \;\; 0 \leq \texttt{Repeat.times} \leq 1000 \wedge \texttt{Repeat.times} > 0 \;\; \wedge \\
&\texttt{English} \rightarrow (\texttt{Repeat} \rightarrow (\texttt{Repeat.times} \geq 2 \wedge \texttt{Repeat.times} \leq 5)).
\end{aligned}
$$

How is variability realised on the ABS program level?

- No subclassing (only subtyping)
- No traits

Variability Modelling: Delta Modelling

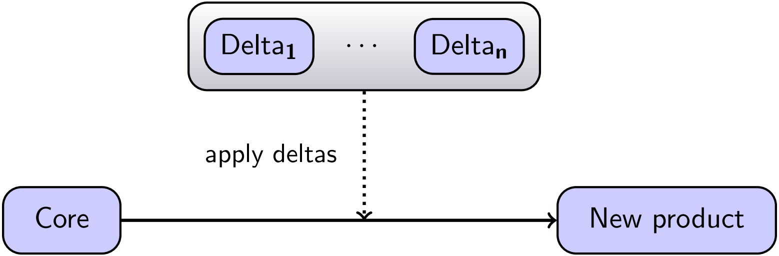
How is variability realised on the ABS program level?

- No subclassing (only subtyping)
- No traits

Approach: (Core-)Delta Modelling

- Base product (the core)
- Variants are composed by applying deltas to the base product

Application of Delta Modules



- Delta modules add, remove or modify classes
- Class modification consists of adding, removing or wrapping fields and methods, adding new interfaces, etc.
- Applies to a core model.

Core Hello World

```
interface Greeting {
    String sayHello();
}
class Greeter implements Greeting {
    String sayHello() {
        return "Hello world";
    }
}
class Application {
    Unit run() {
        Greeting bob;
        bob = new Greeter();
        String s = "";
        s = bob.sayHello();
    }
}
```

Delta Modules of Hello World

```
delta Nl {
    modifies class Greeter {
        modifies String sayHello() {
            return "Hallo wereld";
}   }   }

delta Rpt (Int times) {
    modifies class Greeter {
        modifies String sayHello() {
            String result = "";
            Int i = 0;
            while (i < times) {
                result = result + original();
                i = i + 1;
            }
            return result;
}   }   }
```

Application of Delta Modules

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

```
delta Nl {
   modifies class Greeter {
      modifies String sayHello() {
         return "Hallo wereld";
}  }  }
```

```
class Greeter implements Greeting {
   String sayHello() {
      return "Hallo wereld";
}  }
```

Syntax

$$
\begin{array}{rcl}
\mathit{DeltaDecl} & ::= & \texttt{delta}\ \mathit{TypeId}\ [\mathit{DeltaParams}] \\
& & \{\ \mathit{ClassOrIfaceModifier}^*\ \} \\
\mathit{ClassOrIfaceModifier} & ::= & \texttt{adds}\ \mathit{ClassDecl} \\
& | & \texttt{modifies class}\ \mathit{TypeName} \\
& & \mathit{ImplModifier}^*\ \{\ \mathit{Modifier}^*\ \} \\
& | & \texttt{removes class}\ \mathit{TypeName}\ ; \\
& | & \texttt{adds}\ \mathit{InterfaceDecl} \\
& | & \texttt{modifies interface}\ \mathit{TypeName} \\
& & \mathit{ImplModifier}^*\ \{\ \mathit{Modifier}^*\ \} \\
& | & \texttt{removes interface}\ \mathit{TypeName}\ ; \\
\mathit{ImplModifier} & ::= & \texttt{adds}\ \mathit{TypeName} \\
& | & \texttt{removes}\ \mathit{TypeName} \\
\mathit{Modifier} & ::= & \texttt{adds}\ \mathit{FieldDecl}\ |\ \texttt{removes}\ \mathit{FieldDecl} \\
& | & \texttt{adds}\ \mathit{MethDecl}\ |\ \texttt{modifies}\ \mathit{MethDecl} \\
& | & \texttt{removes}\ \mathit{MethSig}
\end{array}
$$

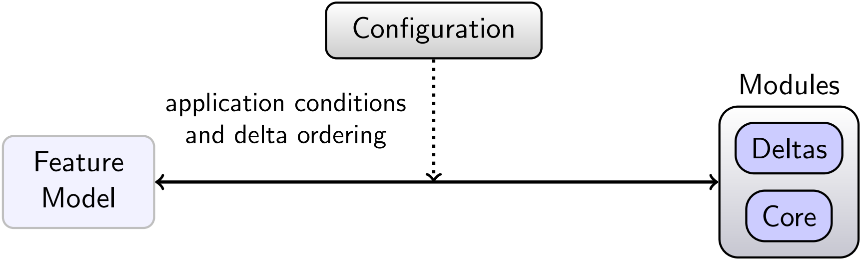
$$
\begin{array}{rcl}
\mathit{DeltaParams} & ::= & (\mathit{DeltaParam}\ (,\ \mathit{DeltaParams})^{*}\ ) \\
\mathit{DeltaParam} & ::= & \mathit{Identifier}\ \mathit{HasCondition}^{*} \\
 & | & \mathit{Type}\ \mathit{Identifier} \\
\mathit{HasCondition} & ::= & \texttt{hasField}\ \mathit{FieldDecl} \\
 & | & \texttt{hasMethod}\ \mathit{MethSig} \\
 & | & \texttt{hasInterface}\ \mathit{TypeName}
\end{array}
$$

Two models: Feature Model and Delta Model

How are they connected?

Variability Modelling: Product Line Configuration

Two models: Feature Model and Delta Model



How are they connected? Product Configuration Language

Example of a Product Line Configuration

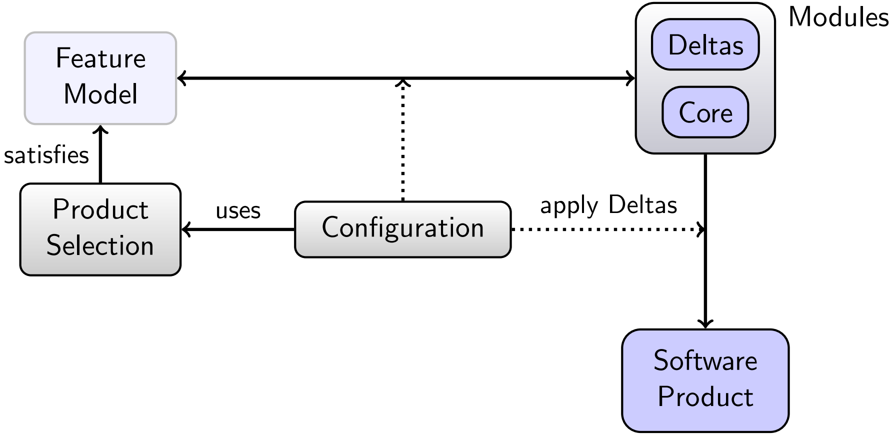
```
productline MultiLingualHelloWorld {
    features English, Dutch, French, German, Repeat;

    delta Nl when Dutch;
    delta Fr when French;
    delta De when German;
    delta Rpt(Repeat.times) after De, Nl, Fr when
 Repeat;
}
```

Syntax of a Product Line Configuration

Configuration	::=	`productline` *TypeId* { *Features* ; *Deltas* }
Features	::=	`features` FID (, FID)*
DeltaClauses	::=	*DeltaClause* (, *DeltaClause*)*
DeltaClause	::=	`delta` *DeltaSpec* [*AfterCondition*] [*ApplicationCondition*] ;
DeltaSpec	::=	*TypeName* [(*DeltaArgs*)]
DeltaArgs	::=	*DeltaArg* (, *DeltaArg*)*
DeltaArg	::=	FID | FID.AID | *DataExp*
AfterCondition	::=	`after` *TypeName* (, Name)*
ApplicationCondition	::=	`when` *Expr*

Product Selection



- Compiler flattens Deltas and Core Module into Core ABS model

Example of Product Selections

```
// basic product with no deltas
product P1 (English) {
    new Application();
}

// apply deltas Nl and Repeat
product P2 (Dutch, Repeat{times=10}) {
    new Application();
}

// apply deltas En and Repeat, but it should be refused
 because "times > 5"
product P3 (English, Repeat{times=6}) {
    new Application();
}
```

Tool Demo - Spatial Variability Modeling

- μTVL - Checking Product Selections
- μTVL - Finding Valid Feature Selections
- Product Generation from Full ABS Models

Tools are available at `http://tools.hats-project.eu/`

Modeling Temporal Variability

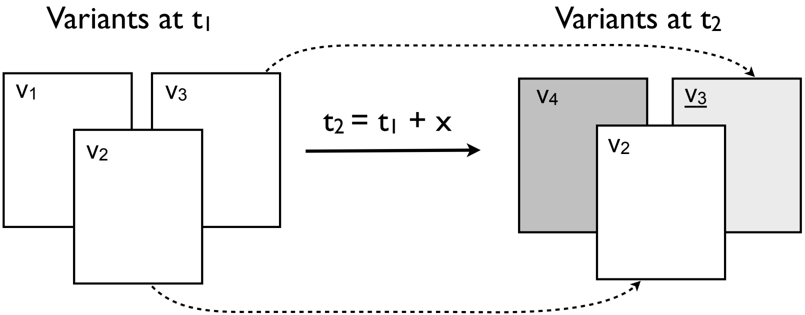
Temporal Variability

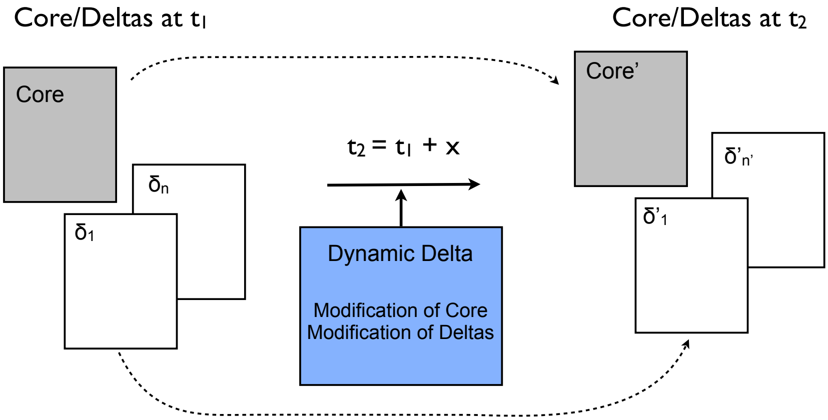
Temporal Variability refers to unanticipated changes of systems.

System evolution occurs due to

- changing user requirements
- changing application contexts and environments
- feature extensions, removals and modifications
- products no longer maintained
- bug fixes and quality improvements

Product Line Evolution




Core/Deltas at t1
Core/Deltas at t2
Core
δn
δ1
t2 = t1 + x
Dynamic Delta
Modification of Core
Modification of Deltas
Core'
δ'n'
δ'1

Dynamic Delta Modeling

Dynamic delta modules can

- add new class and interface declarations
  to the core module or to delta modules
- modify existing class declarations by
  - adding new field and method definitions
  - modifying existing method definitions
  - simplifying classes by
    removing redundant field and method declarations

Dynamic delta modules are more restrictive than spatial delta modules to avoid errors during (asynchronous) runtime evolution.

Dynamic Delta Modeling - Example

```
dyndelta ExtendedGreeting  {
  modifies class Greeter {
    // Adds a new method to the class Greeter
    adds String i_am_bob () { return ", I am Bob!"; }
    modifies String say_hello() {
      return = original() + this.i_am_bob();
     }
   }
```

Dynamic Delta Modeling - Example

Continuation

```
modifies delta De {
 modifies class Greeter {
  modifies String i_am_bob() {return ", ich bin Bob!"
 ;}
  }
 }

modifies delta Nl {
 modifies class Greeter {
  modifies String i_am_bob() {return ", ick ben Bob!"
 ;}
  }
 }
}
```

Summary and Upcoming Features

This Tutorial

- Modeling of Concurrent Systems with Core ABS
- Modeling of Spatial Variability with μTVL, Delta Modeling, Product Line Configuration, Product Selection
- Modeling of Temporal Variability with Dynamic Deltas
- Demos of Tool Suite and Development Environment

Summary and Upcoming Features

This Tutorial

- Modeling of Concurrent Systems with Core ABS
- Modeling of Spatial Variability with μTVL, Delta Modeling, Product Line Configuration, Product Selection
- Modeling of Temporal Variability with Dynamic Deltas
- Demos of Tool Suite and Development Environment

Upcoming Developments

- Improvement of Languages and Tool Support
- Type Checking of Product Lines
- Tool Support for Dynamic Deltas