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

**Specification level**
- Design-oriented, architectural
- Implementation-oriented

**Languages (examples)**
- UML, FDL, ALs
- Spec#, Java+JML

Mind the Gap!
How to rigorously model behavior of large, distributed OO systems?

Specification level

Design-oriented, architectural

Abstract behavioral

Implementation-oriented

Languages (examples)

UML, FDL, ALs

HATS ABS language

Spec#, Java+JML
How?

A tool-supported formal method for building highly adaptable and trustworthy software
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, . . .
3. Develop methods **in tandem** with ABS to ensure feasibility

3. **Methodological and technological framework** integrating HATS tool architecture and ABS language
Ensuring relevance

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

### Diagram

- Assertion Language
  - Composition (COGs)
    - Concurrency model
      - Core Creol
    - Object Model
  - Pure Functional Language
    - ADT
Early evaluation

- Develop Core ABS first
- Layered language design

<table>
<thead>
<tr>
<th>Feature Model ($\mu$TVL)</th>
<th>Delta Modeling (DML)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Line Configuration (CL)</td>
<td></td>
</tr>
<tr>
<td>Product Selection (PSL)</td>
<td></td>
</tr>
</tbody>
</table>

**Behavioral Interface Language**

- Assertion Language
- Composition (COGs)
- Concurrency model
- Core Creol
- Object Model
- Pure Functional Language
- ADT
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

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 & Hänkle, 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

Feature Model ($\mu$TVL)$^1$  |  Delta Modeling (DML)
---|---
Product Line Configuration (CL)  |  
Product Selection (PSL)  

---

Behavioural Interface Language  
Assertion Language  
Composition (COGs)  
Concurrency model  
Core Creol  
Object Model  
Pure Functional Language  
ADT

---

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

User-Defined Data Types

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

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

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

```java
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

```java
// 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

```java
this; A
```
Local Object Creation

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

Local Object Creation

```
this:A
new B();
this:A  b:B
```
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();
```
Local Object Creation

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

COG Creation

```javascript
this:A
new cog B();
this:A b:B
```
### 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

```java
{[Near] Ping ping = new PingImpl();
[Near] Pong pong = new cog PongImpl();

 ping.ping("Hi");  // ok
 pong.pong("Hi");  // error: synchronous call on far reference
```
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

```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
}
```
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;
Synchronization of Concurrent Activities

- Wait until result of an asynchronous computation is ready
  - \texttt{await g}, where \( g \) is a monotonically behaving polling \texttt{guard} expression over \( v? \) and \( v \) is a future reference (has future type)

- Retrieve result of asynchronous computation and copy into a future
  - \( v\texttt{.get} \), where \( v \) is a future referring to a finished task

- Programming idiom:
  \[
  \text{Fut<T> } v = o!m(e)\ldots ; \text{ await } v?; \text{ r = v.get;}
  \]

- 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

- Feature Model ($\mu$TVL)
- Delta Modeling (DML)
- Product Line Configuration (CL)
- Product Selection (PSL)

Diagram:

- Behavioural Interface Language
- Assertion Language
- Composition (COGs)
- Concurrency model
- Core Creol
- Object Model
- Pure Functional Language
- ADT

I. Schaefer
Spatial Variability - Product Line Engineering

Feature Model → Family Engineering → Product Line Artefacts Base → Application Engineering → Product
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.
A product can be seen as selection of features.
A product can be seen as **selection of features**.

Feature model describes **all possible feature combination**.
Hello World Example
Feature Diagram

MultiLingualHelloWorld

Repeat
- Int times in [0..1000]
  - times > 0

Language

- French
- Dutch
- English
- German

Repeat → Repeat.times ≥ 2 ∧ Repeat.times ≤ 5
Feature Modelling Language $\mu$TVL

$\mu$TVL: 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 $\mu$TVL

$\mu$TVL: 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

```plaintext
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 $\mu$TVL

Model ::= (root FeatureDecl)* FeatureExtension*

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

FeatureExtension ::= extension FID { AttributeDecl* Constraint*}

Group ::= group Cardinality

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 ::= || | && | -> | <-> | == | != | > | < | >= | <= | + | - | * | / | %
\( \mu TVL \) models translated into integer constraints.

\[
0 \leq \text{MultiLingualHelloWorld} \leq 1 \land \\
\text{Language} \rightarrow \text{MultiLingualHelloWorld} \land \\
\text{Repeat}^{†} \rightarrow \text{MultiLingualHelloWorld} \land \\
\text{Language} + \text{Repeat}^{†} = 2 \land \\
0 \leq \text{Language} \leq 1 \land \\
\text{English} \rightarrow \text{Language} \land \text{Dutch} \rightarrow \text{Language} \land \text{German} \rightarrow \text{Language} \land \\
1 \leq \text{English} + \text{Dutch} + \text{German} \leq 1 \land \\
0 \leq \text{English} \leq 1 \land 0 \leq \text{Dutch} \leq 1 \land 0 \leq \text{German} \leq 1 \land \\
0 \leq \text{Repeat}^{†} \leq 1 \land \\
\text{Repeat} \rightarrow \text{Repeat}^{†} \land \\
0 \leq \text{Repeat} \leq 1 \land 0 \leq \text{Repeat}.\text{times} \leq 1000 \land \text{Repeat}.\text{times} > 0 \land \\
\text{English} \rightarrow (\text{Repeat} \rightarrow (\text{Repeat}.\text{times} \geq 2 \land \text{Repeat}.\text{times} \leq 5)).
\]
How is variability realised on the ABS program level?

- **No** subclassing (only subtyping)
- **No** traits
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.
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 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

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

delta N1 {
    modifies class Greeter {
        modifies String sayHello() {
            return "Hallo wereld";
        }
    }
}

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

\[ \text{DeltaDecl} ::= \delta \text{ Typeld} [\text{DeltaParams}]
\{ \text{ClassOrIfaceModifier}* \} \]

\[ \text{ClassOrIfaceModifier} ::= \text{adds ClassDecl} \]
\[ | \text{modifies class typeName ImplModifier}* \{ \text{Modifier}* \} \]
\[ | \text{removes class typeName ;} \]
\[ | \text{adds InterfaceDecl} \]
\[ | \text{modifies interface typeName ImplModifier}* \{ \text{Modifier}* \} \]
\[ | \text{removes interface typeName ;} \]

\[ \text{ImplModifier} ::= \text{adds typeName} \]
\[ | \text{removes typeName} \]

\[ \text{Modifier} ::= \text{adds FieldDecl} | \text{removes FieldDecl} \]
\[ | \text{adds MethDecl} | \text{modifies MethDecl} \]
\[ | \text{removes MethSig} \]
\[\text{DeltaParams} ::= (\text{DeltaParam} \ (, \ \text{DeltaParams})^* )\]
\[\text{DeltaParam} ::= \text{Identifier} \ \text{HasCondition}^* \]
\[\quad \mid \quad \text{Type} \ \text{Identifier}\]
\[\text{HasCondition} ::= \text{hasField} \ \text{FieldDecl}\]
\[\quad \mid \quad \text{hasMethod} \ \text{MethSig}\]
\[\quad \mid \quad \text{hasInterface} \ \text{TypeName}\]
Variability Modelling: Product Line Configuration

Two models: Feature Model and Delta Model

How are they connected?
Two models: Feature Model and Delta Model

How are they connected? Product Configuration Language
Example of a Product Line Configuration

```java
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

\[
\begin{align*}
\text{Configuration} & \::= \text{productline } \text{Typeld} \{ \text{Features} ; \text{Deltas} \} \\
\text{Features} & \::= \text{features} \text{FID} (, \text{FID})^* \\
\text{DeltaClauses} & \::= \text{DeltaClause} (, \text{DeltaClause})^* \\
\text{DeltaClause} & \::= \text{delta } \text{DeltaSpec} \\
& \quad [\text{AfterCondition}] [\text{ApplicationCondition}] ; \\
\text{DeltaSpec} & \::= \text{TypeName} [ ( \text{DeltaArgs} ) ] \\
\text{DeltaArgs} & \::= \text{DeltaArg} (, \text{DeltaArg})^* \\
\text{DeltaArg} & \::= \text{FID} | \text{FID.AID} | \text{DataExp} \\
\text{AfterCondition} & \::= \text{after } \text{TypeName} (, \text{Name})^* \\
\text{ApplicationCondition} & \::= \text{when } \text{Expr}
\end{align*}
\]
Compiler flattens Deltas and Core Module into Core ABS model
/* 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

- $\mu$TVL - Checking Product Selections
- $\mu$TVL - Finding Valid Feature Selections
- Product Generation from Full ABS Models

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

Variants at \( t_1 \):
- \( v_1 \)
- \( v_2 \)
- \( v_3 \)

Variants at \( t_2 \):
- \( v_2 \)
- \( v_3 \)
- \( v_4 \)

\[ t_2 = t_1 + x \]
Evolving Delta-oriented Product Lines

\[ t_2 = t_1 + x \]

Core/Deltas at \( t_1 \)

Core

\( \delta_n \)

\( \delta_1 \)

Dynamic Delta

Modification of Core
Modification of Deltas

Core/Deltas at \( t_2 \)

Core'

\( \delta'_n \)

\( \delta'_1 \)
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();
        }
    }
}
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 $\mu$TVL, 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
Summary and Upcoming Features

This Tutorial

- Modeling of Concurrent Systems with Core ABS
- Modeling of Spatial Variability with $\mu$TVL, 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