The Abstract Behavioral Specification Language (ABS)

Martin Steffen
with thanks to Reiner, Einar, Rudi, Jan ...

University of Oslo, Norway

FCMO'10, Graz

November 2010

HATS: Highly Adaptable and Trustworthy Software using Formal Models

Ingredients

Main Ingredients

1. Executable, formal modeling language for adaptable software: Abstract Behavioral Specification (ABS) language
2. Tool suite for ABS/executable code analysis & development:

"Hard"	feature consistency, security, property verification, code generation, type safety...
"Soft"	simulation, visualization, test case generation, specification mining, ...

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

Project objectives

High adaptability combined with high trustworthiness

Challenges

- Concurrency
- Distributedness
- Invasive composition
- different deployment scenarios
- Rapidly changing requirements
- Unanticipated requirements
- Trustworthiness (correctness, security, reliability, efficiency)

Main objectives of ABS

Specification gap for large systems

Specification level	Modeling formalisms
Design-oriented	UML, FDL
Abstract behavioral	Hats ABS language
Implementation-oriented	Spec#, Java+JML

Design principles

ABS is designed with analysability and verifiability in mind

- Expressivity, richness, etc., represent trade-offs
- More practical than "pure" formalisms such as π-calculus, Petri-nets
- State-of-art programming language concepts
- Modeling of realistic software
- Easier to specify/analyse than implementation-level languages
- Various abstraction mechanisms:
  - modularize (separate concerns, encapsulate)
  - permit incremental algorithms
- Modeling of variability a first-class concept

ABS Language Features

Core ABS

- Formal semantics
- Layered architecture: simplicity, separation of concerns
- Executability: rapid prototyping, visualization
- Abstraction: underspecification, non-determinism
- Realistic, yet language-independent concurrency model
- Component object groups structure composition of concurrent objects
- Assertion language: first-order contracts for methods, classes

Full ABS

- Syntactic module system
- Feature modeling language
- Behavioural interface specifications

ABS Language Features

Core ABS

- Formal semantics
- Layered architecture: simplicity, separation of concerns
- Executability: rapid prototyping, visualization
- Abstraction: underspecification, non-determinism
- Realistic, yet language-independent concurrency model
- Component object groups structure composition of concurrent objects
- Assertion language: first-order contracts for methods, classes

Full ABS

- Syntactic module system
- Feature modeling language
- Behavioural interface specifications

Abstractions in the Core ABS

Abstractions coming with the Creol subset

- Communication environment: unordered asynchronous messages
- Release points: underspecified scheduling of internal activities
- Interfaces as types: implementation independent, modularity
- ADTs: avoid representation objects and related reasoning problems

Abstractions coming with Concurrent Object Groups

- Concurrency: lifts Creol's concept of cooperative scheduling to groups of objects
- At most one activity inside the group, all other activies are suspended

Core ABS

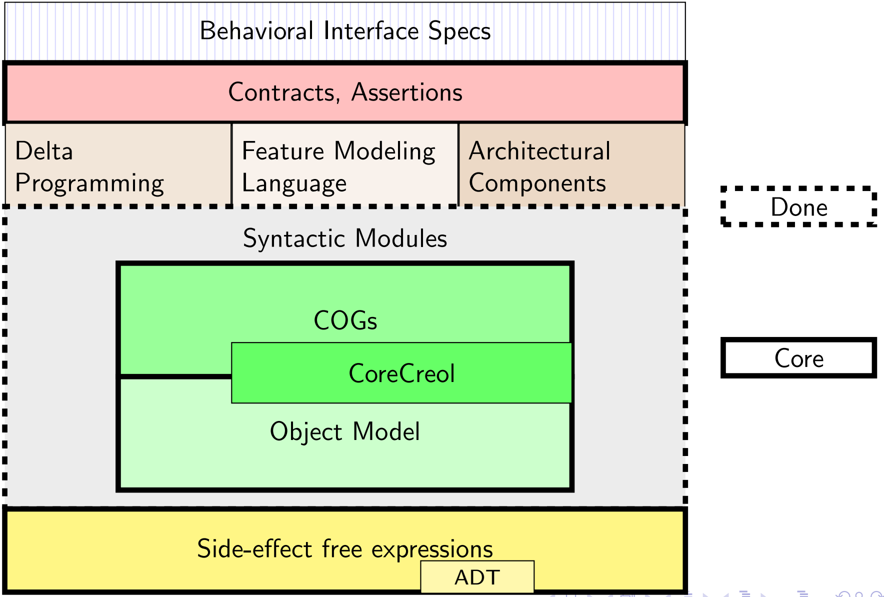
What Core ABS does

- Addresses distributed and concurrent software
- Features user defined ADTs to abstract from repr. objects
- Synchronization in Core ABS is user-decided
- Executable
- Prototype tool chain and Maude interpreter finished
- Rudimentary contract-based assertion language

What Core ABS does not

- Support SPL development, variability, features and feature integration
- Provide structuring concepts beyond interfaces, classes, and methods
- Modules, arch. components, superclasses, traits, deltas, . . .
- Behavioral interface specifications

Layered ABS Language Design



Abstract Datatypes

```
data Bool = True | False;   // built-in
data Unit = Unit;           // built-in
data IntList = Nil | Cons(Int, IntList);
data List<A> = Nil | Cons(A, List<A>); // Parametric type
type IntList = List<Int>; // type synonym
```

Functional Sublanguage

```
def Int length(IntList list) = //
  case list { // definition by case distinction and matching
    Nil          => 0 ;
    Cons(n, ls) => 1 + length(ls) ;
    _            => 0 ; // anonymous variable matches anythi
  } ;
```

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

```
def A fromJust<A>(Maybe<A> a) =
  case a {
    Just(x) => x; // unbound variable used to extract value
  } ;
```

Interfaces and classes

- No class/code inheritance
- Implementation of multiple interfaces ok
- Sub-interfaces ok

```
interface Bar extends Baz { // Baz must be interface
  Method1;
  Method2;
  ...
}

class Foo(T x, U y) implements Bar, Baz { // = constructo
  T f = expr ; U g ; // fields with optional initialization
  { Initblock } // optional initialization block
  Method1 // method declarations
  Method2
  ...
}
```

Interfaces and classes

- No class/code inheritance
- Implementation of multiple interfaces ok
- Sub-interfaces ok

```
interface Bar extends Baz { // Baz must be interface
  Method1;
  Method2;
  ...
}

class Foo(T x, U y) implements Bar, Baz { // = constructo
  T f = expr ; U g ; // fields with optional initialization
  { Initblock } // optional initialization block
  Method1 // method declarations
  Method2
  ...
}
```

Active Classes

- Objects from active classes start activity upon creation
- Characterized by presence of `run()` method
- Passive classes react only to incoming calls

```
Unit run() {
  // active behavior ...
}
```

Methods

```
File getFile(String f, DataBase d) {
   // Method Body (block)
}
```

Annotations

Methods (and classes, interfaces) can carry annotations:
contracts, invariants, . . .

Blocks, Statements

Blocks

- Sequence of variable declarations and statements
- Data type variables must be initialized
- Reference type variables are `null` by default
- Statements in block are scope for declared variables

Statements

- Variable declarations
- Assignments
- while−do, if −then−else
- await, suspend
- (Method calls are expressions and appear e.g. in right sides of assignments)

Method Calls

Synchronous Method Calls

- Syntax: caller .m(e)
- JAVA-like synatx and semantics
- Execution of caller method blocks
- Synchronisation is explicit decision of designer

Asynchronous Method Calls

- Syntax: caller !m(e)
- Execution of caller method continues
- futures
- Variables that contain not yet available values have future type
  - Fut<T> v; ...; v = o!m(e);

Method Calls

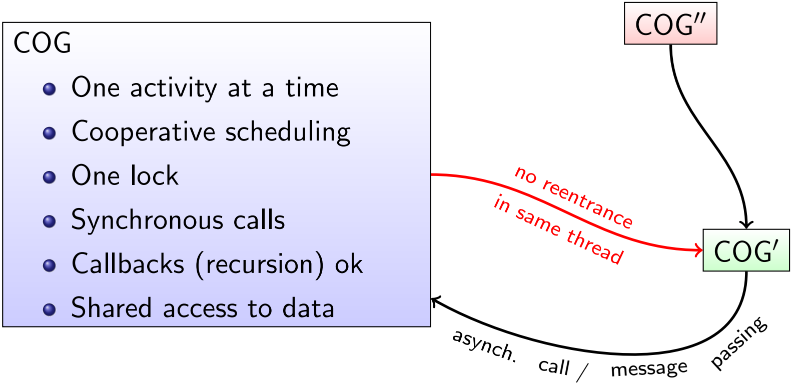
Synchronous Method Calls

- Syntax: caller .m(e)
- JAVA-like synatx and semantics
- Execution of caller method blocks
- Synchronisation is explicit decision of designer

Asynchronous Method Calls

- Syntax: caller !m(e)
- Execution of caller method continues
- futures
- Variables that contain not yet available values have future type
    - Fut<T> v; ...; v = o!m(e);

Component Object Groups (COGs)



Yielding execution

- suspend command yields lock to other task in COG
- Unconditional scheduling point

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
- 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;...; v = o!m(e);...; await v?; r = v.get;
- Conditional scheduling point

Scheduling and Synchronisation

Yielding execution

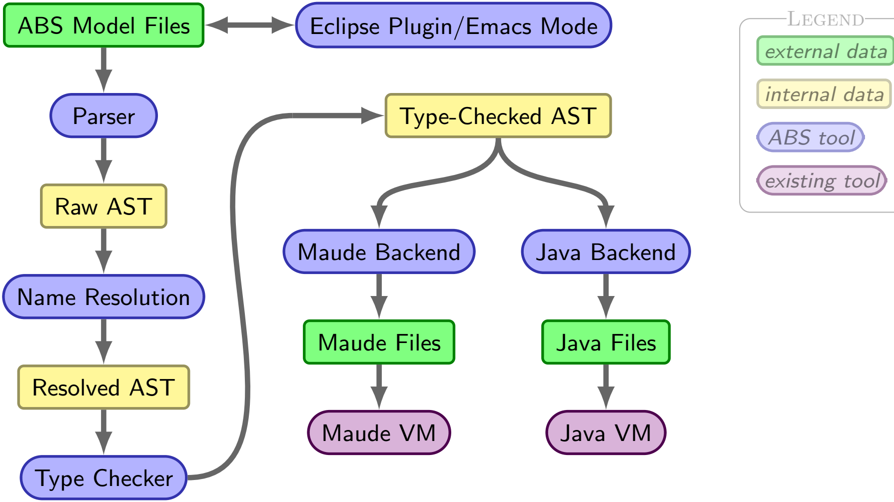
- suspend command yields lock to other task in COG
- Unconditional scheduling point

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
- 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;...; v = o!m(e);...; await v?; r = v.get;`
- Conditional scheduling point

HATS Basic Tool Chain



Layered ABS Language Design