Design of an abstract behavioral specification language

Martin Steffen

University of Oslo, Norway

FMCO Eindhoven
November 2009

http://www.hats-project.eu
HATS project objectives

Challenges

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

*High adaptability combined with high trustworthiness*
Main Project Objectives

Specification gap for large systems

Specification level

Design-oriented

Abstract behavioral

Implementation-oriented

Modeling formalisms

UML, FDL

Hats ABS language

Spec#, Java+JML
Project objectives

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

Ingredients

- **Executable** modeling language for adaptable software: *Abstract Behavioral Specification* (ABS) language
- Integrated framework and architecture
- Tool suite:
  - feature consistency, data integrity, security, correctness, code generation, visualization, test case generation, specification mining
Road map

- **Core Language**
  - object-oriented
  - abstract
  - components/interfaces
  - distributed/concurrent

- **advanced features (for later phases):**
  - feature modelling
  - variability
  - traits/mixins, aspects
A concurrent object model for the ABS language

- executable oo modelling language concurrent objects
- formal semantics in rewriting logics / Maude
- method invocations: synchronous or asynchronous
- targets open distributed systems
- concurrent objects by (first-class) futures
- the language design should support verification
Concurrency

- **“active” objects** (à la Creol)
- objects = unit of concurrency and of state
- field are “private”
- **futures** for “returning” results
- one object: acts as **monitor**
  - mutex + cooperative scheduling
  - **release** points
    - when claiming results from async. method calls (**futures**)
    - explicitly (**yield/release**)
- two ways of claiming a result from future ref.
  - **insistive**: no release
  - **polite**: release, when value not available

\[
e ::= \ldots e.\text{await} \mid e.\text{get} \mid \text{release} \ldots
\]
Method calls

- asynchronous and synchronous
- caller decides
- asynchronous: core of the concurrency model
- synchronous: inside 1 object (group of objects)
- note: synchronous calls ≠ async. call + immediate insisting on getting the value.

\[
e ::= \ldots e.m(\overline{e}) \mid e!m(\overline{e}) \ldots
\]
**Imperative/sequential core**

- **In tendency:**
  - imperative variables = fields = heap
  - non-imperative variables = “stack”, substitution semantics

- **Wish:**
  - unproblematic initialization
  - no nil-pointer-exceptions

\[
e ::= \text{full expression} \\
| \ x \ | \ \text{this} \ | \ \text{this.f} \ | \ \text{this.f := e'} \ | \ fn(e) \\
| \ e = e \ | \ \text{let} \ x : T = e \ \text{in} \ e \ | \ \text{var} \ x := e \\
| \ \text{case} \ e \ \text{of} \ \text{branchlist} \ \text{end}
\]

- **Three level of imperativeness**
  1. side effect free/functional
  2. usable in constructors and initializing expressions
  3. statements

- **Without** \( \text{var} \ x := e \): substitution semantics
Functional/data part

- side-effect free expressions
- simple language for inductive data types
- "generic"\(^1\)
- pattern matching
- part of the assertion language

\(^1\)No generics for classes currently
\[ D ::= \ldots \mid ADT \mid F \] declarations

\[ F ::= \text{def } fn \langle X \rangle (x : T) : T = \text{ea} \] functions

\[ ADT ::= \text{data } dn\langle X \rangle = \text{constrlist} \]

\[ \text{constrlist} ::= \text{constr } "\mid" \text{ constrlist} \]

\[ \text{constr} ::= \text{cn}(T) \] constructor

\[ e ::= \ldots fn(e) \mid \text{cn}(e) \]

\[ \mid \text{case } e \text{ of } \text{branchlist} \text{ end} \]

\[ \text{branchlist} ::= e \mid \text{branch } "\mid" \text{ branchlist} \]

\[ \text{branch} ::= \text{pattern } \rightarrow e \]

\[ \text{pattern} ::= x \mid "\_" \mid \text{cn}(\text{pattern}) \]
\[ D ::= I \mid D_C \ldots \]
\[ I ::= \text{interface } I \text{ extends } I \{ m : T \} \]
\[ D_C ::= \text{class } C(f : T) \text{ implements } I \{ f : T = e_i ; MD \} \]
\[ MD ::= T m(x : T)\{e\} \]
\[ e ::= \]
\[ \ldots \mid \text{this} \mid \text{this.f} \mid \text{this.f} := e' \mid \]
\[ e.m(e) \mid e!m(e) \ldots \]
AOG ("co-boxes")

- "active object groups"
- moderate extension of Creol
- "lightweight" component notion
- "unit of concurrency": group of objects
- synchronous calls: allowed inside an AOG
- instantiator determines whether an "ordinary" object or an "AOG" is created.
- not nested
- not referrable at user-level

\[ e_i ::= \text{new } C(ea) | ea | \text{new } aog \ C(\bar{e}) \]
Type system

- classes/class names are no types (for the user)
- no subtyping on class types
- interface as types
- “multiple subtyping” for interfaces (nominal)
- universal polymorphism (“generics”) for the data type language, not for classes
- no type inference
- no first-class functions in the data language

\[
T \ ::= \ dn\langle T \rangle \mid I \mid X \mid C \\
T_S \ ::= \ T \rightarrow T
\]
Proposal for syntax

\[ P ::= \overline{D} e \]

\[ D ::= I | D_C | ADT | F \]

\[ I ::= \text{interface } I \text{ extends } \overline{I} \{ m : \overline{T_S} \} \]

\[ D_C ::= \text{class } C(\overline{f} : \overline{T}) \text{ implements } \overline{I}\{\overline{f} : \overline{T} = \overline{e_i} ; MD\} \]

\[ F ::= \text{def fn } \langle X \rangle(\overline{x} : \overline{T}) : T = e_a \]

\[ MD ::= T \ m(\overline{x} : \overline{T})\{e\} \]

\[ ADT ::= \text{data } dn(\overline{X}) = \text{constrlist} \]

\[ \text{constrlist} ::= \text{constr } " | " \text{ constrlist} \]

\[ \text{constr} ::= \text{cn}(\overline{T}) \]

\[ e ::= \]

\[ e_i ::= e_a | \text{new } C(\overline{ea}) | \text{new aog } C(\overline{e}) \]

\[ ea ::= \ldots \]

\[ \text{branchlist} ::= e \text{ branch } " | " \text{ branchlist} \]

\[ \text{branch} ::= \text{pattern } \rightarrow e \]

\[ \text{pattern} ::= \overline{x} | " _ " | \text{cn}(\overline{\text{pattern}}) \]

\[ T ::= dn(\overline{T}) | I | X | C \]

\[ T_S ::= \overline{T} \rightarrow T \]

program
declaration
interface
class declaration
functions
method definition
constructor
full expression
initialising expressions
side-effect free expressions
Operational semantics and executable models

- standard operational semantics
- executable semantics in *rewriting logics* (AC-rewriting)
- rapid prototyping
- “execution engine” in Maude rewriter
- basis for “model exploration”: simulate and analyze
ABS tool platform — first steps

For the tool set, currently under work

- A bare-bones compiler (parser, type analysis, ...) for core ABS
- abstract syntax tree (AST) to integrate with other HATS tools
- A Maude VM for basic execution and simulation of models
- A code generator for the Maude VM
Overview of the current tool platform

ABS program

Compiler front-end

Abstract syntax tree

Maude VM back-end

Maude VM code

Tool XYZ

...
Further steps

- program logic + verification support via the KeY tool, symbolic execution
- testing and debugging
- feature description language
- evolvability and variability