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

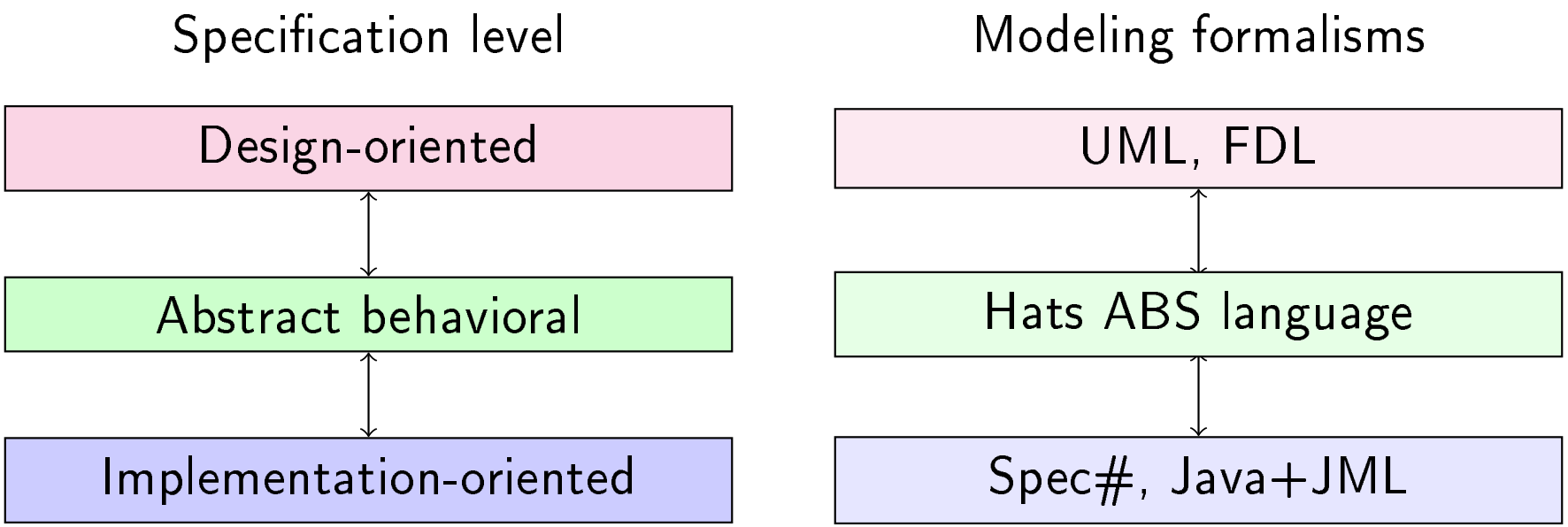
Challenges

- 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



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

$$e ::= \ldots e.\texttt{await} \mid e.\,\text{get} \mid \text{release} \ldots$$

Method calls

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

$$e \quad ::= \quad \ldots \qquad e.m(\overline{e}) \mid e!m(\overline{e}) \ldots$$

Imperative/sequential core

- in tendency:
  - imperative variables = fields = heap
  - non-imperative variables = "stack", substitution semantics
- wish:
  - unproblematic initialization
  - no nil-pointer-exceptions

$$
\begin{array}{lll}
e & ::= & \text{full expression} \\
& \mid & x \mid \texttt{this} \mid \texttt{this}.f \mid \texttt{this}.f := e' \mid fn(e) \\
& \mid & e = e \mid \text{let } x\!:\!T = e \text{ in } e \mid \texttt{var}x := e \\
& \mid & \texttt{case } e \texttt{ of } \textit{branchlist} \texttt{ end}
\end{array}
$$

- three level of imperativeness
  1. side effect free/functional
  2. usable in constructors and initializing expressions
  3. statements
- without $\texttt{var}x := e$: 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

$$
\begin{array}{rcll}
D & ::= & \ldots \mid ADT \mid F & \text{declarations} \\
F & ::= & \texttt{def}\ fn\ \langle \overline{X} \rangle (\overline{x} : \overline{T}) : T = ea & \text{functions} \\
ADT & ::= & \texttt{data}\ dn \langle \overline{X} \rangle = constrlist & \\
constrlist & ::= & constr\ "|"\ constrlist & \\
constr & ::= & cn(\overline{T}) & \text{constructor} \\
e & ::= & \ldots fn(e) \mid cn(\overline{e}) & \\
 & \mid & \texttt{case}\ e\ \texttt{of}\ branchlist\ \texttt{end} & \\
branchlist & ::= & e \mid branch\ "|"\ branchlist & \\
branch & ::= & pattern \rightarrow e & \\
pattern & ::= & x \mid "\_" \mid cn(\overline{pattern}) &
\end{array}
$$

$$
\begin{array}{rcll}
D & ::= & I \mid D_C \ldots & \text{declarations} \\
I & ::= & \texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{\overline{m} : \overline{T}\} & \text{interface} \\
D_C & ::= & \mathit{class}\ C(\overline{f} : \overline{T})\ \texttt{implements}\ \overline{I}\{f{:}T = \overline{ei}; \overline{MD}\} & \text{class def.} \\
MD & ::= & T\ m(\overline{x} : \overline{T})\{e\} & \text{method definit} \\
 & & & \\
e & ::= & & \text{full expression} \\
 & \mid & \ldots \mid \texttt{this} \mid \texttt{this}.f \mid \texttt{this}.f := e' \mid & \\
 & \mid & e.m(\overline{e}) \mid e!m(\overline{e}) \ldots &
\end{array}
$$

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

$$e_i \quad ::= \quad \text{new } C(\overline{ea}) \mid ea \mid \text{new aog } C(\overline{e})$$

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

$$\begin{array}{rcl} T & ::= & dn\langle \overline{T} \rangle \mid I \mid X \mid C \\ T_S & ::= & \overline{T} \to T \end{array}$$

Proposal for syntax

P	::=	$\overline{D}\ e$	program
D	::=	$I \mid D_C \mid ADT \mid F$	declaration
I	::=	$\texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{\overline{m} : \overline{T_S}\}$	interface
D_C	::=	$class\ C(\overline{f} : \overline{T})\ \texttt{implements}\ \overline{I}\{\overline{f}{:}\overline{T} = \overline{ei}; \overline{MD}\}$	class declaration
F	::=	$\texttt{def}\ fn\ \langle\overline{X}\rangle(\overline{x} : \overline{T}){:}T = ea$	functions
MD	::=	$T\ m(\overline{x} : \overline{T})\{e\}$	method definition
ADT	::=	$\texttt{data}\ dn\langle\overline{X}\rangle = constrlist$	
$constrlist$	::=	$constr\ "\mid"\ constrlist$	
$constr$	::=	$cn(\overline{T})$	constructor
e	::=		full expression
	$\mid$	$x \mid \texttt{this} \mid \texttt{this}.f \mid \texttt{this}.f := e' \mid fn(e)$	
	$\mid$	$e.m(\overline{e}) \mid e!m(\overline{e}) \mid e = e \mid \text{let}\ x{:}T = e\ \text{in}\ e \mid cn(\overline{e})$	
	$\mid$	$[\texttt{var}x := e]$	
	$\mid$	$\texttt{case}\ e\ \texttt{of}\ branchlist\ \texttt{end}$	
	$\mid$	$e.\texttt{await} \mid e.\,\text{get} \mid \texttt{yield}$	
e_i	::=	$ea \mid \text{new}\ C(\overline{ea}) \mid \text{new}\ \texttt{aog}\ C(\overline{e})$	initialising expressions
ea	::=	$\ldots$	side-effect free expressions
$branchlist$	::=	$e \mid branch\ "\mid"\ branchlist$	
$branch$	::=	$pattern \rightarrow e$	
$pattern$	::=	$x \mid "\_" \mid cn(\overline{pattern})$	
T	::=	$dn\langle\overline{T}\rangle \mid I \mid X \mid C$	
T_S	::=	$\overline{T} \rightarrow T$	

Operational semantics and executable models

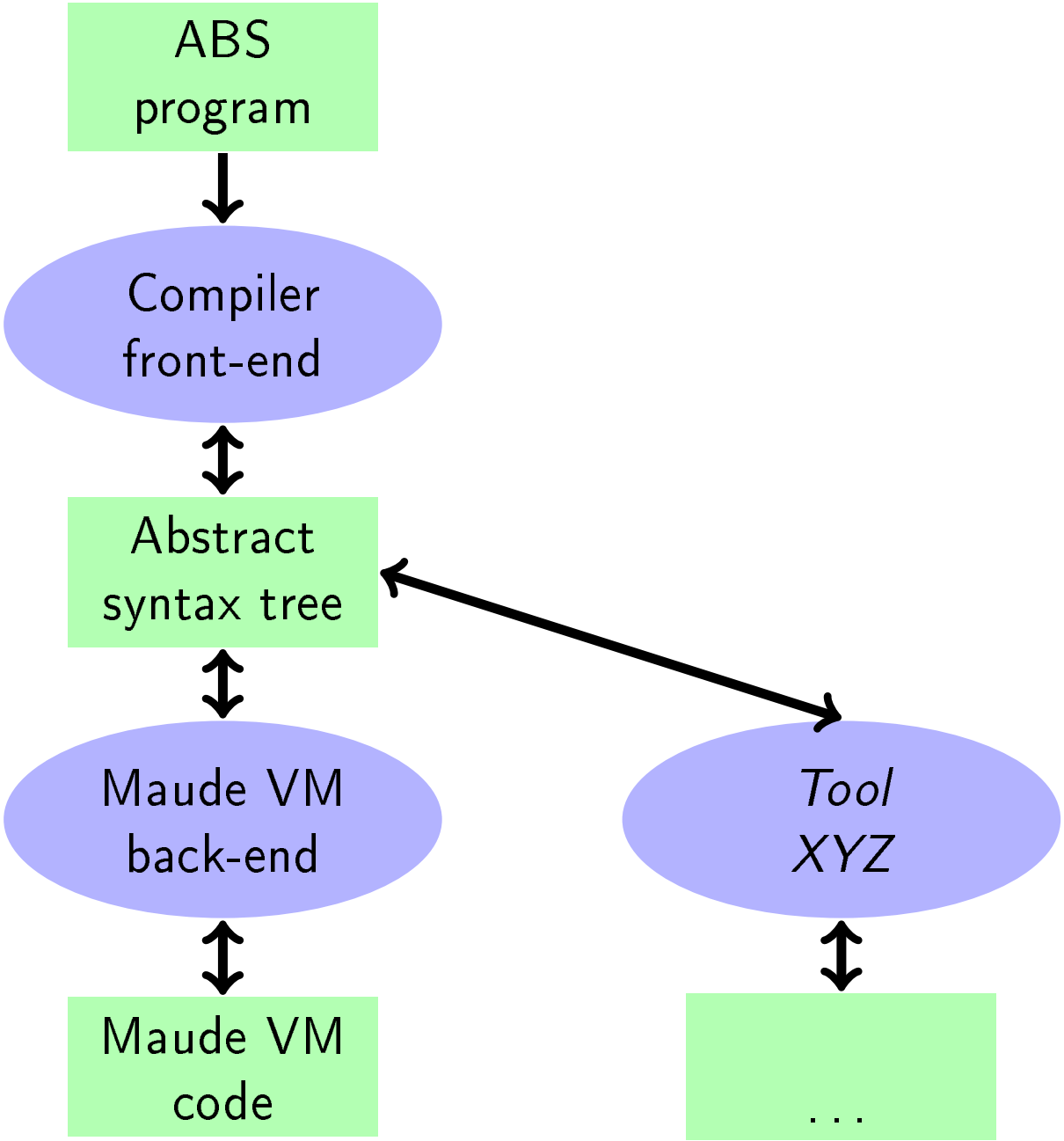
- standard operational semantics
- executable semantics in *rewriting logics* (AC-rewriting)
- rapid prototyping
- "execution engine" in Maude rewriter
- basis for "model exploration": simulate and analyze

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



Further steps

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