Software Engineering using Algebraic Specification (AS)

Dorel Lucanu
"Alexandru Ioan Cuza" University
Faculty of Computer Science
Iasi, Romania
email: dlucanu@infoiasi.ro

Outlines

- formal methods in SE
- main paradigms: MSA, OSA, MA, HA, RWL
- what is an algebraic specification
- proof engineering
  - rewriting
  - induction
  - coinduction
- projects in progress
  - more expressivity: HA and model checking
  - integration: HA and CCS/pi-calculus
- conclusion

Formal methods in SE

- main concerns
  - specification
  - verification
  - refinement
- a formal method

  involves

  specification languages

  logics

  tools

  to produce right specifications

Specification languages

- a classification
  - model oriented (VDM, Z, CTL …)
  - **algebraic (OBJ family**, CASL, …)
  - process model (CSP, CCS, pi-calculs, …)
  - logical (HOL, PVS, …)
  - broad spectrum (LOTOS, …)

AS Paradigms

- Many Sorted Algebra (MSA)
- Order Sorted Algebra (OSA)
- Membership Algebra (MA)
- Hidden Sorted Algebra (HA)
- Rewriting Logic (RWL)
- ...

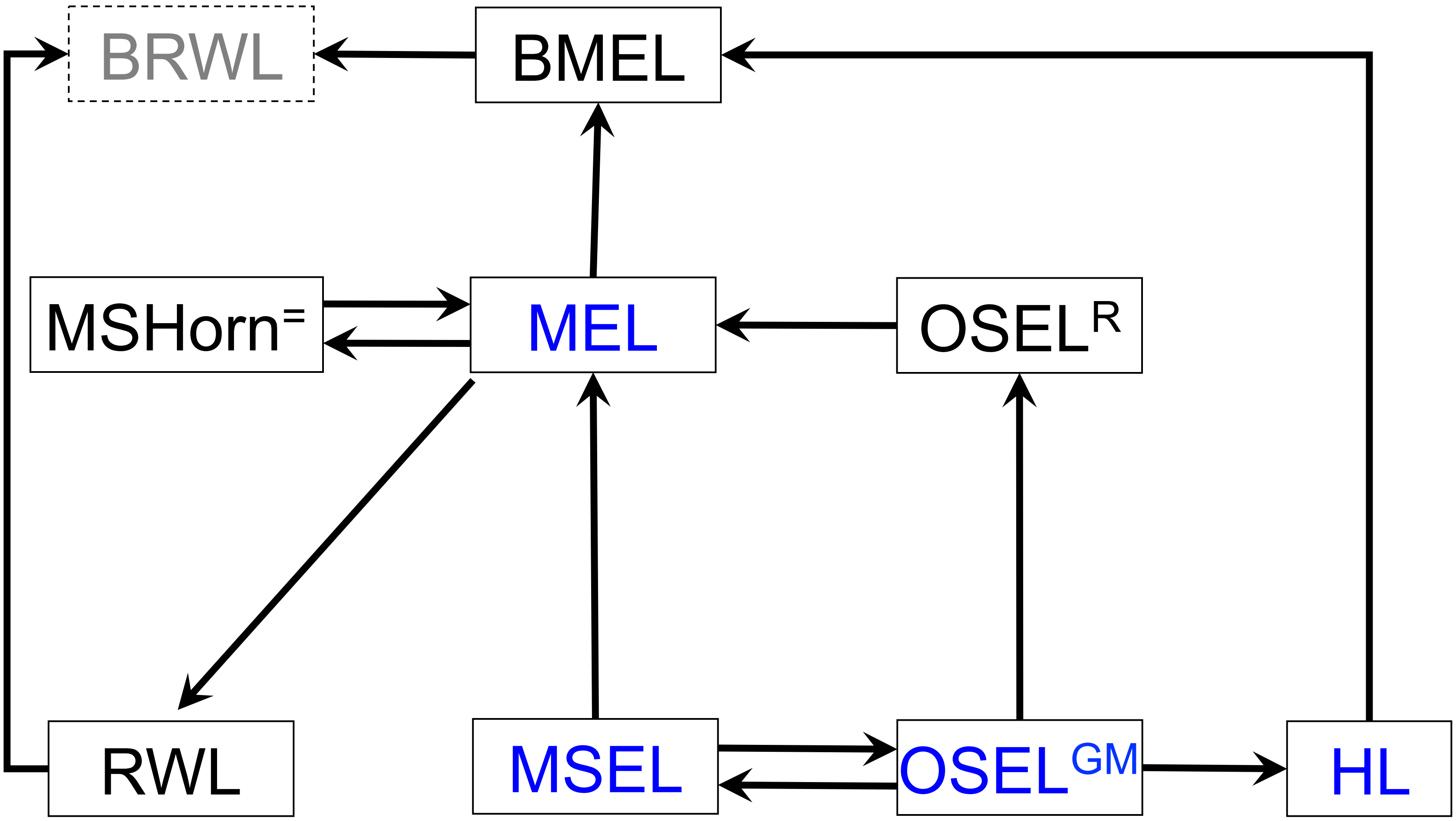
AS paradigms, logics & proof engines

Paradigm	Logic	Proof engines
MSA, OSA	Equational Logic (MSEL, OSEL) (sound, complete)	Rewriting
MA	Membership Equational Logic (MEL) (sound, complete)	Rewriting, Membership deduction
RWL	Rewriting Logic (sound, complete)	Rewriting
HA	Behavioral Equational Logic (sound, incomplete)	Coinductive behavioral rew.

Proof methodologies

What specify	Proof methodology
ADT	Confl. & terminat. rew. Induction
Objects, systems (behavioral specs)	Coinduction Circular Coinduction
Transitional systems	Induction Backtracking

Comparing paradigms



What is an algebraic specification?

- a specification (Σ, E) includes:
  - the signature Σ
    - sorts
    - a partial order on sorts (OSA)
    - operational symbols
  - properties of the operations (sentences) E
    - simple equations
    - conditional equations
    - membership assertions (MEL)
    - transition specifications (rewrite rules) (RWL)

Specification of naturals

```
obj NAT is
  sorts NzNat Nat .
  subsort NzNat < Nat .
  op 0 : -> Nat .
  op s_ : Nat -> NzNat .
  op _+_ : Nat Nat -> Nat .
  vars M N : Nat .
  eq M + 0 = M .
  eq M + s N = s(M + N) .
  *** ...
endo
```

Sets of naturals

```
obj SET is
 including NAT .
 sort NatSet .
 op {} : -> NatSet .
 op ins : Nat NatSet -> NatSet .
 op _U_ : NatSet NatSet -> NatSet .
 ...
 eq ins(N,ins(N',S)) = ins(N',ins(N,S))
    if N > N' .
 eq ins(N, ins(N', S)) = ins(N, S)
    if N == N' .
 eq S U {} = S .
 eq S U ins(N, S') = ins(N, S U S') .
endo
```

Membership assertions

- sets of even naturals

```
sort EvNat .
subsort EvNat < Nat .
mb 0 : EvNat .
mb s s N : EvNat if N : EvNat .
```

or equivalently

```
mb N : EvNat if N % 2 == 0 .
```

Models

- a model is an algebra

  N = algebra of naturals

  - a sort is interpreted as a set

    [[`Nat`]] = {0, 1, 2, ...}, [[Nz`Nat`]] = {1, 2, ...}

  - a subsort relation is interpreted as inclusion

    [[`NzNat`]] $\subseteq$ [[`Nat`]]

  - operation symbols are interpreted as operations

    [[`s`]] : [[`Nat`]] -> [[`NzNat`]]

    [[`s`]] (x) = x + 1

Models

- equations are equalities that must be satisfied:

  N $\models$ **(∀M,N)M + s N = s(M + N)**

  because

  x + (y+1) = (x + y) + 1   for all x, y $\in$ [[Nat]]

- membership assertions are interpreted as membership relations that must be satisfied:

N $\models$ (∀**M**)**s s M : EvNat if M : EvNat**

  because

  x + 2 $\in$ [[**EvNat**]] if x $\in$ [[**EvNat**]]

  [[**EvNat**]] = {0, 2, 4, …}

Algebra of ground terms $T_{\Sigma,E}$

- terms: a term is built from variables, constants, and operation symbols in the usual way
- examples of terms: `0`, `s0` , `ss0`, `X + sY`
- ground terms: no variable
- $T_{\Sigma,E}$ = the set of ground terms modulo E

```
[0] = { 0, 0 + 0, 0 + (0 + 0), ...}
[s0] = {s0, s0 + 0, 0 + s0,
        0 + s0 + 0, ...}
```

Abstract Data Types (ADT)

- M initial model iff $(\forall M')(\exists! h : M \to M')$
- initial models = standard models
  - no junks (is minimal)
  - no confusions
- two initial (Σ,E)-models are isomorphic
- ADT = isomorphism class of the initial models
- $T_{\Sigma,E}$ is initial

Proof engineering for ADT: rewriting

- equations are seen as rewrite rules
- one step rewriting relation:

  t => t' iff t' is obtained from t replacing a subterm by another subterm according to a rewrite rule (equation) and a computed substitution

  t =*=> t' means zero, one or more rewriting steps

Proof engineering for ADT: rewriting

- normal forms

  nf(t) = t' iff t =*=> t' and t' is irreducible

- normal form exists and it is unique if =*=> is **terminating** and **confluent**

$$
\begin{tikzcd}
 & \bullet \arrow[dl, Rightarrow, "\star"'] \arrow[dr, Rightarrow, "\star"] & \\
\bullet \arrow[dr, Rightarrow, "\star"] & & \bullet \arrow[dl, Rightarrow, "\star"'] \\
 & \bullet &
\end{tikzcd}
$$

confluency property

$$
\begin{tikzcd}
 & X+s(Y+0) \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
X+sY \arrow[dr, Rightarrow] & & s(X+(Y+0)) \arrow[dl, Rightarrow] \\
 & s(X+Y) &
\end{tikzcd}
$$

OBJ

BOBJ> in set.bob

BOBJ> select SET .

BOBJ> **red ins(0, ins(1,{})) U ins(1,{})** .

==

reduce in SET : ins(0, ins(s 0, {})) U ins(s 0, {})

result NatSet: **ins(0, ins(1, {}))**

rewrite time: 63ms   parse time: 16ms

BOBJ>

OBJ

```
BOBJ> red ins(0, ins(0, ins(0, {}))) == ins(0, {}) .
======================================
reduce in SET : ins(0, ins(0, ins(0, {}))) == ins(0, {})
result Bool: true
rewrite time: 0ms       parse time: 0ms
BOBJ>
```

Proof engineering for ADT: induction

- initial truth = properties that are true in initial models
- (structural) induction = a method to prove initial truth
- example

  {} U S = S

  - it cannot be proved using rewriting (no chance even we use equational deduction)
- the method

Proof engineering for ADT: induction

- define first the constructors
  - for SET they are `{}` and `ins(N, S)`
  - Q: How do we find them?
- check the property for each constant constructor

  BOBJ> red {} U {} .

  result NatSet: {}
- for each non-constant constructors
  - formulate the inductive hypothesis

  BOBJ> op s : -> NatSet .

  BOBJ> op n : -> Nat .

  BOBJ> eq {} U s = s .

Proof engineering for ADT: induction

- check the inductive conclusion

```
BOBJ> red {} U ins(n, s) == ins(n, s) .
==========================
reduce in SET : {} U ins(n, s) == ins(n, s)
result Bool: true
rewrite time: 46ms
```

Hidden Algebra (HA) : the problem

- stack

```
obj STACK[X :: TRIV] is
  sort Stack .
  op push : Elt Stack -> Stack .
  op pop : Stack -> Stack .
  op top : Stack -> Elt .
  var S : Stack .  var E : Elt .
  eq top(push(E, S)) = E .
  eq pop(push(E, S)) = S .
endo
```

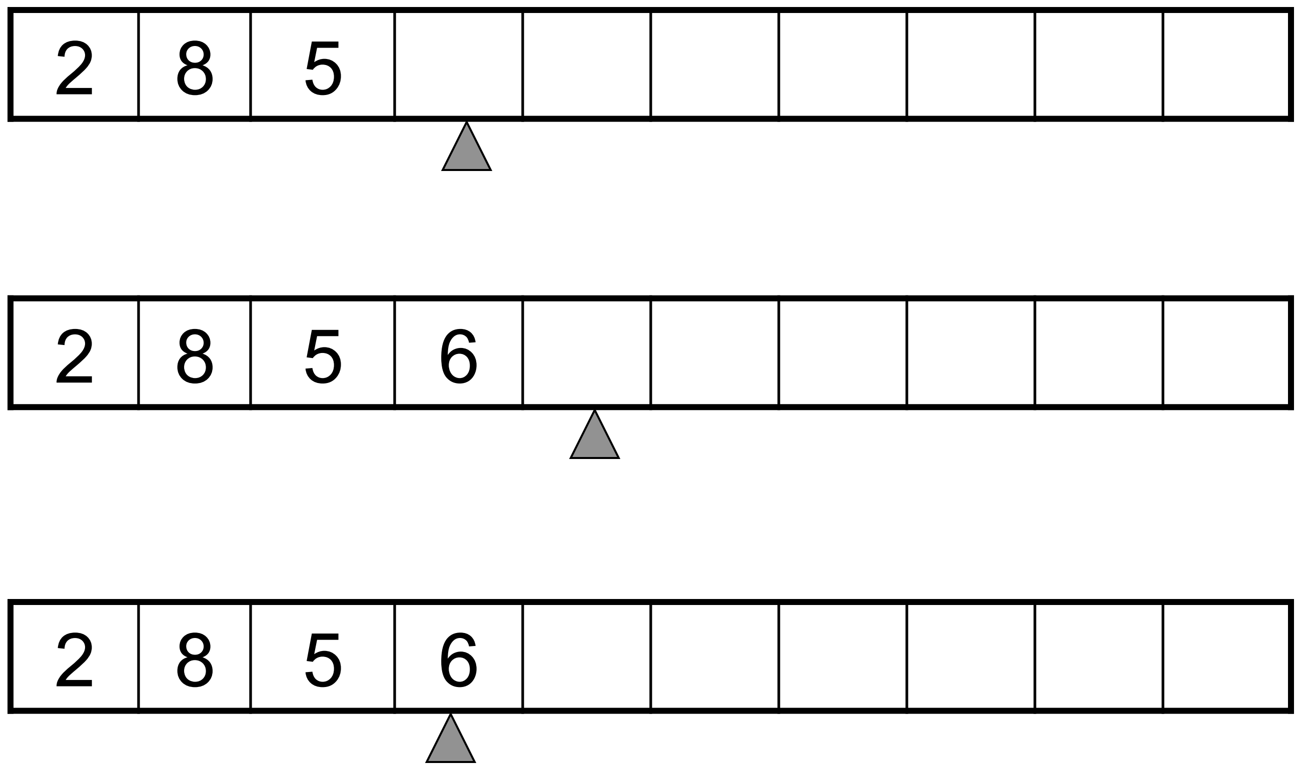
HA: the problem

- stack: implementation with arrays

S

push(S, 6)

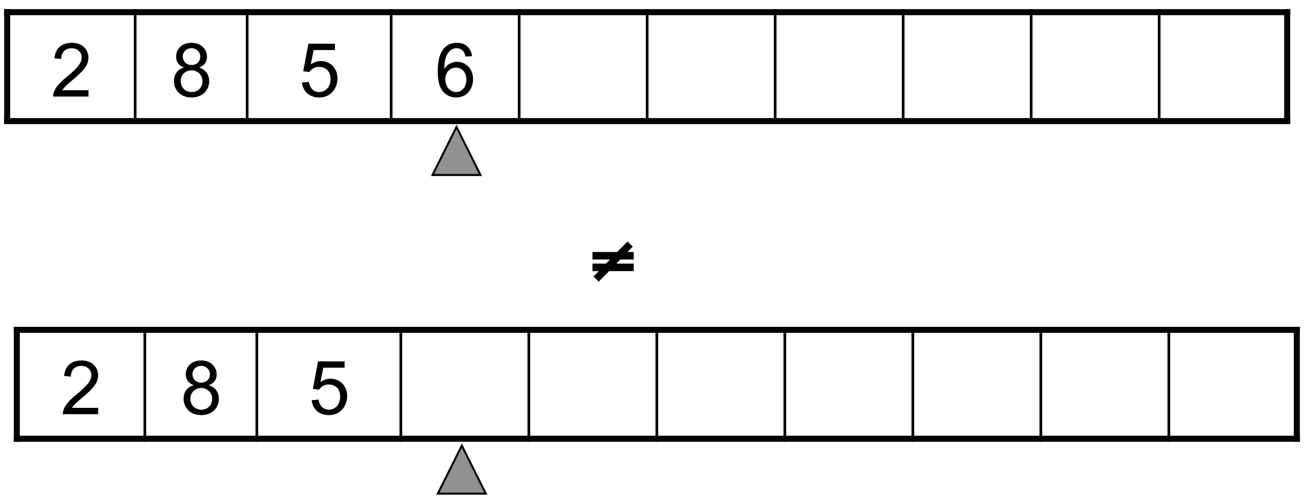
pop(S)



HA: the problem

- but

  `pop(push(S, 6)) ≠ S`



HA: the problem

- if we consider the experiments:

  `top(_),`

  `top(pop(_)),`

  `top(pop(pop(_)))`

  - the results are the same: **5 8 2**
- we say that S and S' are **behavioral equivalent** iff they return equal responses for each experiment

HA specification

- signature = (H, V, Σ, D)
  - H - hidden sorts (not observable)
  - V - visible sorts (observable)
  - Σ - (V U H)-sorted signature
  - D - $\Sigma|_V$ – algebra (data algebra)
- sentences: Σ - equations
- specification = (Σ, Γ, E)

  $\Gamma \subseteq \Sigma$ s.t. $\Gamma|_V = \Sigma|_V$

  E – a set of equations
  - operations in $\Gamma - \Sigma|_V$ are called ***behavioral***

HA: models

- model - Σ-algebra M s.t. $M|_{\Sigma|V} = D$
- $\equiv$ is the behavioral equivalence

$$a \equiv b \text{ iff } (\forall\Gamma\text{-experiment } c)[[c]](a) = [[c]](b)$$

- equations are behaviorally satisfied:

$$[[\theta(t)]] \equiv [[\theta(t')]]$$

HA: specification of objects

- hidden sorts – models the state space of the object
- operations:
  - methods: $f : hv_1 \ldots v_n \rightarrow h$
  - observers (attributes): $f : hv_1 \ldots v_n \rightarrow v$
- concurrent composition operator: _ | _

Unreliable Counter

```
bth UCOUNTER is
  sort Counter .
  pr NAT .
  op init : -> Counter .
  op dec : Counter -> Counter
  [ncong] .
  op read : Counter -> Nat .
  ops (inc) (reset) : Counter ->
                      Counter .
  var C : Counter .
  eq read(init) = 0 .
  eq val(reset(C)) = 0 .
  eq read(inc(C)) = read(C) + 1 .
end
```

Unreliable Counter

- Γ = { **`read()`**, **`reset()`**, **`inc()`** }
- dec() is unreliable: the result of the experiment **`read(dec(C))`** is not predictable
- therefore **`dec()`** is not congruent (**`ncong`**)

HA: proof engineering

- rewriting is not always sound
  - behavioral rewriting
    - disregards the beh. non-cong. opns
- semantics of a behavioral specification = the category of all models
  - induction is not appropriate (it is good only for the initial model, if any)
- behavioral reasoning is of coalgebraic nature
  - coinduction
- circular coinductive rewriting

HA: coinduction

- Input:

  $B = (\Sigma, \Gamma, E)$

  t, t' Σ-terms (representing states)
- Output

  Are t and t' Γ-behavioral equivalent?
- Method
  1. define an appropriate relation R
  2. prove that R is a Γ-behavioral congruence
  3. if $t\ R\ t'$ then return YES

Coinductive rewriting in BOBJ

```
BOBJ> open UCOUNTER .
BOBJ> ops c : -> Counter .
BOBJ> red reset(c) == init .
==========================================
reduce in UCOUNTER : reset(c) == init
result Bool: false
rewrite time: 0ms       parse time: 0ms
BOBJ> cred reset(c) == init .
==========================================
c-reduce in UCOUNTER : reset(c) == init
using cobasis for UCOUNTER:
   op read : Counter -> Nat
   ---------------------------------------
result: true
c-rewrite time: 0ms     parse time: 0ms
```

Can HA be model checked?

- behavioral attributes ⇨ queries
- behavioral methods ⇨ actions
  - execution of a behavioral method produces a transition (in a model)
- queries defines atomic CTL formulas
- canonical CTL model
- cooperative activity of the model checking algorithm and HA proof engines

HA and CCS

- Cells

```
bth CELL is
    sort Cell .
    pr DATA-CELL .
    op init : -> Cell .
    op read : Cell -> Data .
    op write : Cell Data -> Cell .
    var C : Cell . var D : Data .
    eq read(write(C, D)) = D .
  end
```

HA and CCS

- Buffers

```
bth CELL1 is
  inc CELL * (sort Cell to Cell1) .
end

bth CELL2 is
  inc CELL * (sort Cell to Cell2) .
end

bth  BUFFER2 is
  inc (CELL1 | CELL2)*
    (sort Configuration to Bufer2) .
end
```

HA and CCS

- in any state any operation is possible
- so, any scenario is possible
- sometimes we wish to investigate what happen for some given scenarios
- these scenarios can be described by CCS processes

HA and CCS

- considering the actions:

  i ≡ write.CELL1(_, D)

  t | ~t ≡ write.CELL2(_, read.CELL1(_))

  ~o ≡ read.CELL2(_)

- a scenario is:

  C1 ≡ i.o.C1

  C2 ≡ i.o.C2

  B ≡ new t ({t/o}C1 | {t/i}C2)

Conclusions

- Advantages
  - sentences are very simple
  - a good mathematical support
  - rigorous semantics
  - specifications are executable
  - there are software tools
- Drawbacks
  - mathematical notation
  - … too much mathematics
  - a gap between specification and implementation

Conclusions

- What do we can do
  - accompany AS with more powerful and intuitive software tools
  - integrate AS with complementary methods
  - fill the gap between specification and implementation