Models and Structuring of Specifications

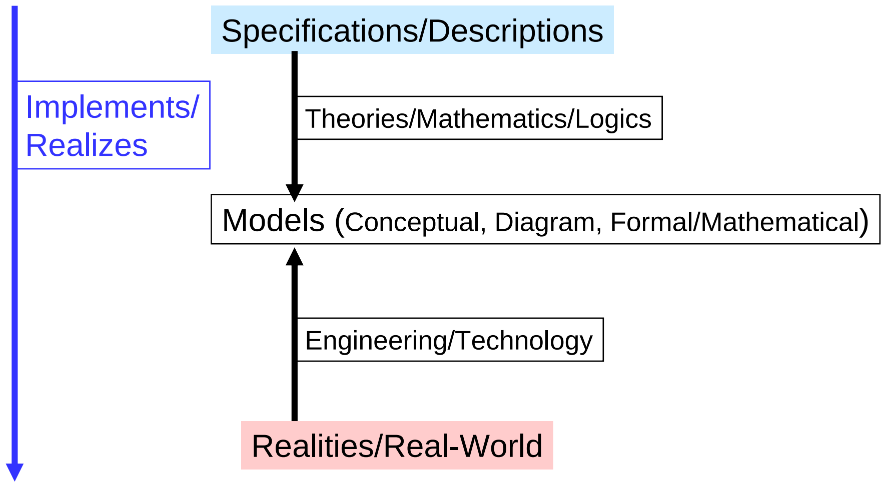
CafeOBJ Team of JAIST

Models and Satisfaction

Topics

- **Specification/Descriptions, Models, and Realities**
- **Order Sorted Term Algebra, Quotient Term Algebra, and Equation Reasoning**
- **Congruence defined by Specification**
  - **Equivalence Relation**
  - **Congruence Relation**
- **Initial/Tight denotation and Loose denotation**
- **Satisfaction of a Property prop by a Specification SPEC**
  - **SPEC |= prop**

Specifications, Models, Realities



Specification and its model (1)

An **equational specification SPEC** in CafeOBJ (a legitimate text in the CafeOBJ language with only equations as axioms) is defined as a pair **<** Σ, E **>** of order-sorted signature Σ and a set of conditional equations E .

A model of an equational specification **SPEC =** < Σ, E > is an **algebra**. An algebra is a mathematical object composed of operations over order-sorted sets. A signature Σ of a specification **SPEC =** < Σ, E > determines a set of **Σ-algebras** (order-sorted algebras) of the signature Σ .

5

Specification and its model (2)

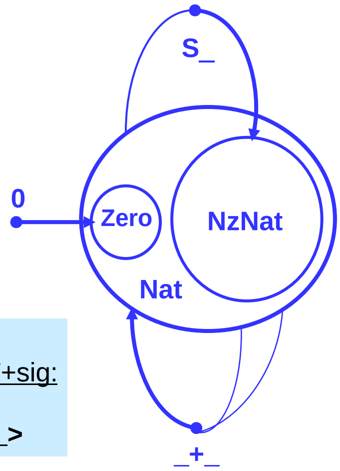
A **Σ-algebra A** is an order-sorted algebra. An order-sorted algebra is a mathematical object which is composed of operations defined over order-sorted sets. Order-sorted sets are many-sorted sets with subset relations.

A **Σ-algebra A** interprets a sort symbol s of the signature Σ as a (non empty) set $\mathbf{A}_\Sigma$ and an operation (function) symbol **f** of the specification as a function $\mathbf{A}_\mathbf{f}$. The interpretation respects the order-sort constrains, and ranks (types) of functions.

6

An example of Signature and its Algebra

```
-- signature NAT+sig
-- sort
[ Zero NzNat < Nat ]
-- operators
op 0 :   -> Nat
op s_ :  Nat -> NzNat
op _+_ : Nat Nat -> Nat
```



A MAT+sig-algebra
Order-Sorted Algebra with Signature NAT+sig:

`<Nat, NzNat, Zero; 0, s_, _+_>`

7

Order Sorted Term Algebra $T_{\texttt{NAT+sig}}$ of Signature NAT+sig

An *Order-Sorted Algebra* is a mathematical object composed of order-sorted carrier sets and operations over them.

Order-Sorted Carrier sets can be thought of Order-Sorted Sets of Terms:

Zero = { *0* }
NzNat = { *s n* | *n* ∈ Nat }
Nat = Zero ∪ NzNat ∪
{ *n1 + n2* | *n1* ∈ Nat ∧ *n2* ∈ Nat }

Operations over the order-sorted sets of terms:

0 = *0*
(for *n*∈Nat)(s *n* = *s n*)
(for *n1*,*n2*∈Nat)(*n1* + *n2* = *n1 + n2*)

8

valuation, evaluation, equation

A **valuation** (or an assignment) is a sort preserving map from the (order-sorted) set of variables of a specification to an order-sorted algebra (a model), and assigns values to all variables.

Given a model **A** and a valuation **v**, a **term t** of sort **s,** which may contain variables, is evaluated to a **value** $A_v(t)$ in a set A_s

Given terms t_1 and t_2 of a sort **s** and **c** of sort **Bool**, a **conditional equation** is a sentence of the form:

$$t_1 = t_2 \text{ if } c$$

An ordinary equation $t_1 = t_2$ is an abbreviation of

$$t_1 = t_2 \text{ if true}$$

9

LectureNote2, Sinaia School, 03-10/03/2008

Satisfiability of equation

Given a model (an ordered-sorted algebra) **A**,

A satisfies an equation $t_1 = t_2$

iff

$$A_v(t_1) = A_v(t_2)$$

for any valuation **v** .

The satisfaction of an equation by a model A is denoted by

$$A \models (t_1 = t_2)$$

10

LectureNote2, Sinaia School, 03-10/03/2008

SPEC-algebra

For a CafeOBJ specification **SPEC** = < Σ, E >,
a **SPEC-algebra** is a Σ-algebra which satisfy
(1) all equations in E
and
(2) semantic constrains of **SPEC.**

For a CafeOBJ specification **SPEC** = < Σ, E >,
a semantic constrains of **SPEC** are
(1) tight/loose denotations
and
(2) protecting/extending importations.

Tight denotation: Quotient Term Algebra $T_{NAT+sig}$/=NAT+

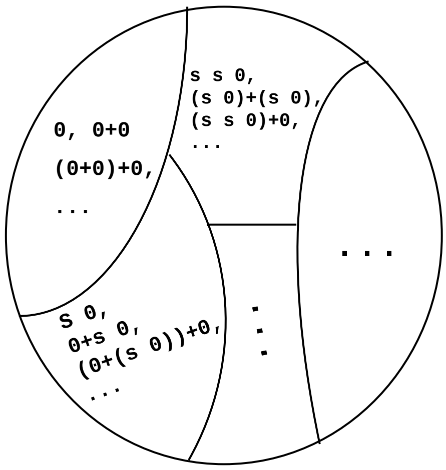
```
mod! NAT+ {
-- signature
-- sort
[ Zero NzNat < Nat ]
-- operators
op 0 : -> Nat
op s_ : Nat -> NzNat
op _+_ : Nat Nat -> Nat
-- equations
eq 0 + N:Nat = N .
eq (s M:Nat) + N:Nat
     = s(M + N) . }
```

mod! indicates the denotation to the initial/standard model

Two equations of **NAT** defines congruence relations **=NAT+** over the carrier sets of $T_{NAT+sig}$.

The specification **NAT+** denotes the quotient term algebra $T_{NAT+sig}$**/=NAT+** as the initial/standard model.

Quotient Term Algebra T_{NAT+}/=NAT+



$T_{NAT+sig}$

is also written as

T_{NAT+}

Congruence Relation =SPEC

Given a specification **SPEC**, a binary relation **=SPEC** (written just = in the following) on sorted sets of terms of **SPEC** is the congruence defined by **SPEC** if and only if it is the smallest relation which satisfies the following five properties:

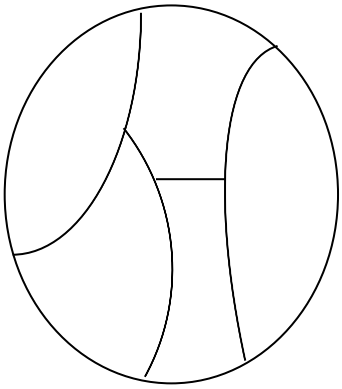
Reflexivity: t1= t1
Symmetry: t1=t2 implies t2=t1
Transitivity: (t1=t2 and t2=t3) implies t1=t3
Congruence: for any operator f,
(t1=t1' and … and ti=ti') implies
f(t1,…,ti)=f(t1',…,ti')
Substitutivity: for any conditional equation (e=e' if c) in **SPEC**
and any assignment a,
a(c)=`ture` implies a(e)=a(e')

=NAT+ is the congruence defined by **NAT+**

Equivalence Relation (等価関係) and Partition (分割)

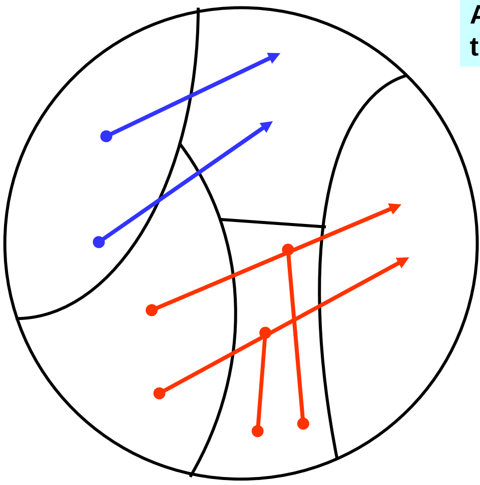
A binary relation (a set of pairs) which satisfies reflexivity, symmetry, and transitivity is defined to be an **equivalence relation.**

↔



Partition

Congruence Relation



Any operator preserves the equivalence relation!

Inference rules for conditional equations -- identical to =SPEC

Reflexivity: （反射律）

```
------------
  t1= t1
```

Symmetry: (对称律）

```
   t1=t2
-------------
   t2=t1
```

Transitivity: （推移律）

```
  t1=t2     t2=t3
--------------------------
        t1=t3
```

Congruence: （合同律）

```
  t1=t1' t2=t2' …  tn=tn'
--------------------------------   {f is an operator}
 f(t1,…,ti)=f(tn',…,tn')
```

Substitutivity: （代入律）

```
  a(c)=ture
------------------  {(ceq e=e' if c .),  a is an assignment}
  a(e)=a(e')
```


Initial = (No Junk + No Confusion)

No Junk = Any element of a carrier set is represented by the operators in the signature.

Examples of Junks in spec. **NAT+:**

(s s 0) * (s 0), ¾, ∞, a

No Confusion = No two elements of a carrier set are equivalent unless they can be shown to be equal using the axioms (equations) of the specification.

Examples of confusion in spec. **NAT+:**

s 0 = 0, s 0 + 0 = s s 0

Loose denotation: Non-Initial Model

```
mod* NAT+loose {
-- signature
-- sort
[ Zero NzNat < Nat ]
-- operators
op 0 : -> Nat
op s_ : Nat -> NzNat
op _+_ : Nat Nat -> Nat
-- equations
eq 0 + N:Nat = N .
eq (s M:Nat) + N:Nat
   = s(M + N) . }
```

A non-initial model of **NAT+loose**

Carrier sets:

Zero = { 0 }
NzNat = { s *n* | *n* ∈ Nat }
Nat =
  Zero ∪ NzNat ∪ { a } ∪
  { *n1* + *n2* |
    n1 ∈ Nat ∧ *n2* ∈ Nat }

Operations:

0 + 0 = 0 ,   0 + a = a
a + 0 = 0 ,   a + a = 0
s N:Nat = N

mod* indicates denotation to all models that satisfy the spec.

_+_ is not commutative and not associative!

Another Example:

`NAT+loose U {eq s s s … s 0 = 0 .}`

LectureNote2, Sinaia School, 03-10/03/2008

19

Satisfiability of boolean term

A SPEC-algebra **A satisfies a term** `t` **of sort** `Bool` iff $\mathbf{A_v(t)}$ = `true` for any valuation **v** (or iff **A** satisfies an equation `t = true`) .

The satisfaction of a predicate by a model **A** is denoted by:

A |= p

Only the satisfaction relation:

A |= p

is simulated by CafeOBJ System. The satisfaction relation

$\mathbf{A \mid\!\!= (t_1 = t_2)}$

can not be simulated by CafeOBJ system, because equation is a meta-entity and not in the object level of CafeOBJ.

20

LectureNote2, Sinaia School, 03-10/03/2008

Equality predicate _=_

There is a special operation (predicate) symbol **_=_** with an operation declaration "`op (_=_) : s s -> Bool`" for any sort symbol **s** of any signature Σ of any specification **SP** = < Σ, E >. For any model **A**, **_=_** is **postulated** to be interpreted as the equality (or identity) relation on the set A_s .

This is formulated as follows.

For any specification **SPEC** = < Σ, E >, any **SPEC-algebra A** is postulated to satisfy:

A |= (t_1 = t_2) iff **A** |e= (t_1 = t_2)

for any pair of terms t_1 and t_2 . That is, we only consider the model which satisfies this condition.

21

Satisfiability of property by specification: SPEC |= prop

A specification **SPEC** = **< S, E >** is defined to satisfy a property **p** (a term of sort **Bool)** iff **A |= p** holes for any **SPEC-algebra A.**

The satisfaction of a predicate **prop** by a specification **SPEC** = < Σ, E > is denoted by:

SPEC |= p or Σ,E |= p or E |= p

An important purpose of developing a specification **SPEC =** <Σ,E> in CafeOBJ is to check whether

SPEC |= prop

holds for a predicate **prop** which describe some important property of the system which **SPEC** specifies.

22

Structuring

- Module Imports:
  - protecting, extending, and using
- Parameterized Module
- Parameter Instantiation
- Module Expression

Module Imports

```
-- imported module
mod! BARE-NAT
{ [ NzNat Zero < Nat ]
  op 0 : -> Zero
  op s_ : Nat -> NzNat
}

-- importing module
mod! NAT+
{ protecting(BARE-NAT)

  op _+_ : Nat Nat -> Nat
  eq 0 + N:Nat = N .
  eq (s M:Nat) + N:Nat = s(M + N) .
}
```

Sufficiently completeness of **NAT+** over **BARE-NAT** guarantees the "**no junk**"

import declaration

module body

Three Importation Modes

Semantics definition of three modes

protecting: no junk and no confusion into the imported module

extending: may be junk but no confusion into the imported module

using: may be junk and confusion into the imported module

No semantics checks are done by CafeOBJ system w.r.t. protecting and extending

Examples of **protecting** and **extending**

```
-- imported module
mod! BARE-NAT
{ [ NzNat Zero < Nat ]
  op 0 : -> Zero
  op s_ : Nat -> NzNat }

-- importing module
mod! NAT+
{ protecting(BARE-NAT)
  op _+_ : Nat Nat -> Nat
  eq 0 + N:Nat = N .
  eq (s M:Nat) + N:Nat =
     s(M + N) .  }
```

Suff. Comp. guarantees no junk. Two equations preserve the number of **s_** in the term, hence no confusion.

```
-- imported module
mod! BARE-NAT
{ [ NzNat Zero < Nat ]
  op 0 : -> Zero
  op s_ : Nat -> NzNat
}

-- importing module
mod! NAT-INFINITY
{ extending(BARE-NAT)
  op omega : -> Nat
  eq s omega = omega . }
```

"omega" is a junk. No equations for a term like **s s s … s 0**, hence no confusion.

Parameterized Module

```
-- built-in module TRIV
mod* TRIV { [ Elt ] }



--> parameterized string
mod! STRG (X :: TRIV)
{
 -- any element is string of
 -- length one
 [ Elt < Strg ]
 -- a binary juxtaposing
 -- operation for strings
 op (_ _) : Strg Strg -> Strg
                       {assoc}
}
```

formal parameter module:
specifies possible actual parameters with loose denotation

parameter declaration:
specifies a pair of a (formal) parameter name and parameter module name

Parameter instantiations (1)

structSpecSTRG.mod

```
-- (0) standard way of declaring view first and instantiate
--     a formal parameter using it;
view natAsTriv from TRIV to NAT {sort Elt -> Nat}
make NAT-STRG0 (STRG(X <= natAsTriv))
make NAT-STRG0' (STRG(natAsTriv))

-- (1) on the fly view declaration
make NAT-STRG1
    (STRG(X <= view to NAT {sort Elt -> Nat}))

-- (2) on the fly view declaration of shorter version
--     this is a recommend way to instantiate parameters
make NAT-STRG2 (STRG(NAT{sort Elt -> Nat}))
```

Parameter instantiations (2)

-- (3) on the fly view declaration using the mod construct

```
mod! NAT-STRG3 {
  protecting(STRG(NAT {sort Elt -> Nat}))}
```

-- (4) on the fly view declaration

-- with sort renaming *{sort Strg -> NatStrg}

```
make NAT-STRG4
((STRG(NAT{sort Elt -> Nat}))*{sort Strg -> NatStrg})
```

-- (5) making use of default view mechanism:

-- it is possible because the sort Nat is declared to be

-- the principal-sort in the built-in module NAT;

-- it is not recommended if you are not get used to the notions of

-- principal-sort and default view;

-- with sort renaming "`*{sort Strg -> NatStrg}`"

```
make NAT-STRG5 ((STRG(NAT))*{sort Strg -> NatStrg})
```


Principal-sort and default view (1)

bareNatWithPsort.mod

```
--> BARE-NAT
mod! BARE-NAT {
  [ NzNat Zero < Nat ]
  op 0 : -> Zero
  op s_ : Nat -> NzNat
}

--> notice that the following does not work
--> because the pricipal sort is not declared
--> in the module BARE-NAT
make NAT-STRG8 (STRG(BARE-NAT))
make NAT-STRG9 (STRG(X <= BARE-NAT))
```

Principal-sort and default view (2)

```
-->  if the principal sort is declared as:
mod! BARE-NATwithPsort principal-sort Nat
{ [ NzNat Zero < Nat ]
  op 0 : -> Zero
  op s_ : Nat -> NzNat
}

--> then the following two work
make NAT-STRG10 (STRG(BARE-NATwithPsort))
make NAT-STRG11 (STRG(X <= BARE-NATwithPsort))
```


Parameterized lexicographic ordering (1)

stringOfStringOf.mod

```
--> a loose specification of totally ordered elements
mod* TOSET
{ us(EQL)
  [ Elt ]
  pred _<_ : Elt Elt   -- strict total ordering

  vars E1 E2 E3 : Elt
  eq E1 < E1 = false .
  eq ( ((E1 < E2) or (E2 < E1) or (E1 = E2))
        and
       not((E1 < E2) and (E2 < E1))
        and
       not((E2 < E1) and (E1 = E2))
        and
       not((E1 < E2) and (E1 = E2)) ) = true .
  eq (((E1 < E2) and (E2 < E3)) implies (E1 < E3)) = true .
}
```

Parameterized lexicographic ordering (2)

stringOfStringOf.mod

```
mod! STRGlex (Y :: TOSET) { [ Elt < Strg ]
  op _ _ : Strg Strg -> Strg {assoc}
  -- lexicographic ordering over strings
  op _<<_ : Strg Strg -> Bool
  eq (E1:Elt):Strg << (E2:Elt):Strg = (E1):Elt < (E2):Elt .

  ceq (E1:Elt):Strg << (E2:Elt  S2:Strg) = true
                if (E1 = E2) .
  ceq (E1:Elt):Strg << (E2:Elt  S2:Strg) = true
                if (E1 < E2).
  ceq (E1:Elt):Strg << (E2:Elt  S2:Strg) = false
                if (E2 < E1) .

  ceq (E1:Elt  S1:Strg) << (E2:Elt):Strg  = false
                if (E1 = E2) .
  ceq (E1:Elt  S1:Strg) << (E2:Elt):Strg  = true
                if (E1 < E2) .
  ceq (E1:Elt  S1:Strg) << (E2:Elt):Strg  = false
                if (E2 < E1) .

  ceq (E1:Elt  S1:Strg) << (E2:Elt  S2:Strg) = S1 << S2
                  if E1 = E2 .
  ceq (E1:Elt  S1:Strg) << (E2:Elt  S2:Strg) = true
              if (E1 < E2) .
  ceq (E1:Elt  S1:Strg) << (E2:Elt  S2:Strg) = false if (E2 < E1) . }
```


An Example of Module Expression

```
mod! NATeq {pr(NAT + EQL)}

make NAT-STRG-STRGlex2
(
 STRGlex
 ( ((STRGlex
    (NATeq{sort Elt -> Nat})
   )
   *{sort Strg -> St,
     op (_ _) -> (_&_),
     op (_<<_) -> (_<st_)}
  )
  {sort Elt -> St,
   op (E1:Elt < E2:Elt)
      ->(E1:St <st E2:St)} )
)
```

actual parameter

view

Rename

view body