Basics of Modeling, Specification, Verification in CafeOBJ

CafeOBJ Team of JAIST

Topics

- Basic concepts for modeling, specification, verification in CafeOBJ
- Basics of CafeOBJ language system: module, signature, equations, term, parsing, debugging, trace

Basics of Modeling/Specification/Verification

Modeling/Specifying and Verifying in CafeOBJ

1. By understanding a problem to be modeled/specified, determine several sorts of <u>objects</u> (entities, data, agents, states) and <u>operations</u> (functions, actions, events) over them for describing the problem

2. Define the meanings/functions of the operations by declaring <u>equations</u> over expressions/terms composed of the operations

3. Write <u>proof scores</u> for properties to be verified

Natural Numbers -- Signature --

0 0+1 0+1+1 0+1+1+1 0+1+1+1+1 …

0 s(0) s(s(0)) s(s(s(0))) s(s(s(s(0)))) …

objects: Nat
operations: 0 : returns zero without arguments
s : given a natural number *n* returns the next natural number (s *n*) of *n*

```
-- sort
[ Nat ]
-- operations
op 0 :  -> Nat
op s_:  Nat -> Nat
```

Natural Number
-- Expressions/terms composed of operations --

```
mod! BASIC-NAT
  { [ Nat ] op 0: -> Nat op s_: Nat -> Nat }
```

1. 0 is a natural number
2. If *n* is natural number then (s *n*) is a natural number
3. An object which is to be a natural number by 1 and 2 is only a natural number

Peano's definition of natural numbers (1889), Giuseppe Peano (1858-1932)

Nat = {0, s(0), s(s(0)), s(s(s(0))), s(s(s(s(0)))) … }

Nat = {0, s 0, s s 0, s s s 0, s s s s 0, … }

Describe a problem in expressions/terms!

Mathematical Induction over Natural Numbers

The recursive definition of Nat induces the following induction scheme!

<u>Goal:</u> Prove that for any natural number $n \in \{0,\ s\ 0,\ s\ s\ 0, \ldots\}$ $P(n)$ is true

<u>Induction Scheme:</u>

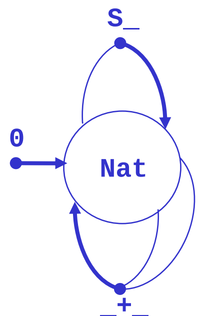
$$\frac{P(0) \qquad \forall n \in N.[P(n) => P(s\ n)]}{\forall n \in N.P(n)}$$

<u>Concrete Procedure:</u> (induction with respect to n)

1. Prove $P(0)$ is true
2. Assume that $P(n)$ holds, and prove that $P(s\ n)$ is true

Natural numbers with addition operation -- signature and expressions/terms --

```
-- sort
[ Nat ]
-- operations
op 0 :   -> Nat
op s_:  Nat -> Nat
op _+_: Nat Nat -> Nat
```



$$\text{Nat} = \{ 0 \} \cup \{ s\ n \mid n \in \text{Nat} \} \cup \{ n1 + n2 \mid n1 \in \text{Nat} \wedge n2 \in \text{Nat} \}$$

Natural numbers with addition
-- expressions/terms composed by operations --

```
op 0: -> Nat . op s_:  Nat -> Nat . op _+_: Nat Nat -> Nat .
```

```
Nat = {
0, s 0, s s 0, s s s 0, ... ,
0 + 0, 0 + (s 0), 0 + (s s 0), 0 + (s s s 0), ...,
(s 0) + 0, (s 0) + (s 0), (s 0) + (s s 0),
                           (s 0) + (s s s 0), ...,
(s s 0) + 0, (s s 0) + (s 0), (s s 0) + (s s 0),
                         (s s 0) + (s s s 0), ...,
... ...
0 + (0 + 0), 0 + (0 + (s 0)), ...
...
(0 + 0) + 0, (0 + (s 0)) + 0, ...
...
. }
```


Natural numbers with addition
-- equations defining meaning/function -- natPlus.mod basicNatPlus.mod

CafeOBJ module NAT+ defining Natural numbers with addition

```
mod! NATplus {
-- sort
[ Nat ]
-- operations
op 0 :  -> Nat {constr}
op s_:  Nat -> Nat {constr}
op _+_: Nat Nat -> Nat
-- equations
eq          0 + N:Nat = N .
eq (s M:Nat) + N:Nat = s(M + N) .
}
```

Inference/Computation with the equations

```
  (s s 0) + (s 0)
= s((s 0) + (s 0))
= s s(0 + (s 0))
= s s s 0
```

```
…> select NATplus
NATplus> red s s 0 + s 0 .
NATplus> -- reduce in NAT+
: ((s (s 0)) + (s 0)):Nat
(s (s (s 0))):NzNat
(0.000 sec for parse,
3 rewrites(0.000 sec),
5 matches)
```

Proof Score
for the proof of associativity of addition (_+_)

natPlusAssocPS.mod

```
-- opening module NATplus and EQL
open (NATplus + EQL)
--> declaring constants for arbitrary values
ops i j k : -> Nat .

**> Prove associativity: (i + j) + k = i +(j + k)
**> by induction on i

**> base case proof for 0:
red 0 + (j + k) = (0 + j) + k .
**> induction hypothesis:
eq (i + J:Nat) + K:Nat = i + (J + K) .
**> induction step proof for (s k):
red ((s i) + J:Nat) + K:Nat = (s i) + (J + K) .
**> QED {end of proof for associativity of (_+_)}
close
```


module, signature, equation, term, order-sort

Three kinds of modules

CafeOBJ specification is composed of modules. There are three kinds of modules.

```
mod! <module_name> {
  <modlue_element> *
}
```

```
mod* <module_name> {
  <modlue_element> *
}
```

```
mod <module_name> {
  <modlue_element>*
}
```

`mod!` **declares that the module denotes tight denotatin**
`mod*` **declares that the module denotes loose denotation**
`mod` **does not declare any semantic denotationd**

[Naming convention] module name starts with two successive upper case charactors (example: **`TEST, NAT, NATplus`**, **`ACCOUNT-SYS,…`**)

Module NATplus

A module is composed of signature and axioms

```
mod! NATplus {
  [Nat]
  op 0 : -> Nat
  op s_ : Nat -> Nat
  op _+_ : Nat Nat -> Nat
  vars M N : Nat
  eq     0   + N = N .
  eq (s M) + N = s(M + N) .
}
```

signature (the lines from `[Nat]` to `op _+_ : Nat Nat -> Nat`)

axioms/equations (the lines from `vars M N : Nat` to the second `eq`)

Signature:
sort name, operator name, arity, co-arity, rank

A signature is a pair of a set of sorts and a set of operations.

Signature

```
mod! NATplus {
 [Nat]
 op 0 : -> Nat
 op s : Nat -> Nat
 op _+_ : Nat Nat -> Nat
  ... }
```

sorts

operations

[Convention] The first and second letter of a sort name is written in a upper case and lower case letter respectively. (E.g. **Nat, Set**)

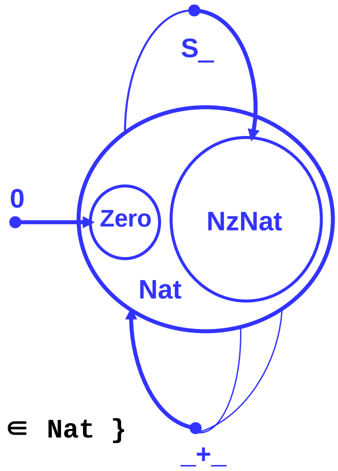
[Convention] The first letter of an operation name is written in a lowerl case letter or a non-alphabet letter. (E.g. **0, s, +**)

op _+_ : $\boxed{\textbf{Nat Nat}}$ **->** $\boxed{\textbf{Nat}}$

arity co-arity (rang) rank

Natural numbers with addition
-- order sorted signature and sorted terms –

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



Sorted terms

Zero = { 0 }

NzNat = { s ***n*** **|** ***n*** **∈ Nat }**

Nat = Zero ∪ NzNat ∪

{ ***n1*** **+** ***n2*** **|** ***n1*** **∈ Nat ∧** ***n2*** **∈ Nat }**

Recursive Definition of Terms

For a given signature, t is a term of a sort S if and only if t is

- a variable X:S,
- a constant c declared by "op c : -> S", or
- a term $f(t_1, , t_n)$ for "op f : S_1 S_n -> S" and a term t_i of a sort S_i (i =1, ,n).
- a term of a sort S' which is a sub-sort of S (Example: Since Zero < Nat, a term 0:Zero is also a term of sort Nat)

Several forms of function application: standard, prefix, infix, postfix, distfix

```
op f : Nat Nat -> Nat .
    f(2,3)  standard
op (f_ _) : Nat Nat -> Nat .   -- recommended
                               -- for succesive __
    (f 2 3) prefix
op f__ : Nat Nat -> Nat .
    (f 2 3) prefix
op _+_ : Nat Nat -> Nat .
    (2 + 3) infix
op  _! : Nat -> Nat .
    (5 !)   postfix
op if_then_else_fi : Bool Nat Nat -> Nat .
    (if 2 < 3 then 4 else 5 fi)  distfix
```

Term = Tree = Expression

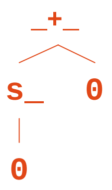
nat+.mod

A tree data structure having operators as node and constants or variables as leaf is called a term. A term is also called an expression.

- `(s 0) + 0` represents term/tree/expression

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

(S 0) + 0



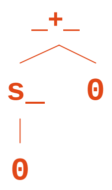
Parsing – precedence of operators-

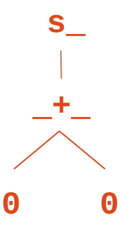
nat+.mod

`s 0 + 0` represents `(s 0) + 0`, because the operator `(s _)` has high precedence than the operator `(_ + _)`

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

s 0 + 0





(s 0) + 0 s(0 + 0)

Error handling with subsorts

ratDiv.mod

```
RAT> parse 2 / 2 .
(2 / 2) : NzRat

RAT> reduce 2 / 2 .
1 : NzNat

RAT> parse 2 / 0 .
(2 / 0) : ?Rat
```

```
RAT> parse 2 / ((3 / 2) + (1 / 2)) .
(2 / ((3 / 2) + (1 / 2))) : ?Rat

RAT> red 2 / ((3 / 2) + (1 / 2)) .
1 : NzNat
```


Equation

An equation is a pair of terms of a same sort, and written as:

eq l = r .

in CafeOBJ. Where **l** is called the left-hand side (LHS) of the equation and **r** is the right-hand side (RHS). An equation can have a condition (COND) **c** like:

ceq l = r if c .

- **Most important kind of axioms of CafeOBJ specification are equations**
- **Properties to be verified are also expressed as equations**

Two way of declaring variables
- both should be used based on situations -

```
mod! NAT+ {
  [Zero NzNat < Nat]
  . . .
  eq    0       + N:Nat = N .
  eq (s M:Nat) + N:Nat = s(M + N) .
}
```

axioms

Variables can be declared before axioms

```
mod! NAT+ {. . .
  vars M N : Nat
  eq    0   + N = N .
  eq (s M) + N = s(M + N) .
}
```

axioms

Basics of Verification

How to do verification with CafeOBJ specifications

- **The basic mechanism of CafeOBJ verification is equational reasoning. Equational reasoning is to deduce an equation (a candidate of a theorem) from a given set of equations (axioms of a specification).**
- **The CafeOBJ system supports an automatic equational reasoning based on rewriting (or TRS: Term Rewriting System).**
- **"`reduce`" or "`red`" (reduction) command to do equational reasoning is provided by CafeOBJ System.**

Reduction command: Equational reasoning by rewritings

There are two ways to do equational reasoning in CafeOBJ by rewritings: `red <term>.` and `red <term> = <term>.`

```
NAT+> red +(0, s(0)) .
-- reduce in NAT+ : +(0,s(0))
s(0) : NzNat
(0.000 sec for parse, 1 rewrites(0.000 sec), 1 matches)
```

This means that the input term is equivalent to the output term.

```
NAT+> open (NAT+ + EQL) –- for using equality predicate (_=_)
NAT+ + EQL%> red +(0, s(0)) = +(s(0), 0) .
-- reduce in NAT+ : +(0,s(0)) = +(s(0),0)
true : Bool
(0.000 sec for parse, 4 rewrites(0.000 sec), 5 matches)
```

This means that the one side is equivalent to the other side.

What can be done with red (reduction) command?

red.mod

A reduction command of CafeOBJ:

`MODULE> red inputTerm .`

returns a most simplified term of the given term ***inputTerm*** by using all equations of the module ***MODULE*** as rewriting rules from LHS to RHS. For any context,

`any-module> red in MODULE : red inputTerm .`

returns the same result.

Let us fix a context M (a module M in CafeOBJ), and let (**t1 =*M> t2**) denote that **t1** is reduced to **t2** in the context**.** That is, (**red in M : t1 .)** returns **t2** . Let (**t1 =M t2**) denote that **t1** is equal to **t2** in the context **M**. It is important to notice:

(**t1 =*M> t2**) implies (**t1 =M t2**)

but

(**t1 =M t2**) does not implies (**t1 =*M> t2**)

Two equality predicates _=_ and _==_

=and==

Assume that (**t1 =*> t1'**) and (**t2 =*> t2'**) in any context then

if (**t1'** and **t2'** are the same term)

then (**red t1 = t2 .**) returns **true**

and

(**red t1 == t2 .**) returns **true**

if (t1' and t2' are different terms)

then (**red t1 = t2 .**) returns (**t1' = t2'**)

but

(**red t1 == t2 .**) returns **false**

Soundness of _=_ and _==_

=and==

- The result of **“red <term1> == <term2> .”** is sound but not complete, that is:
  - If it returns true, then the two terms **<term1>** and **<term2>** is proved to be equal.
  - But if it returns false, then the two terms may equal or not equal.
- The reduction of Boolean term involving _**==**_ may return true even if it is not true w.r.t. the set of axioms (or the specification). That is, _**==**_ may not be sound.
- If the reduction of Boolean term involving only _**=**_ returns true, then it is true w.r.t. the set of axioms (or the specification) .

Proof scores in a wide sense

- **A fragment proof score begins at “open” command which opens a module, and ends with “close” command.**
- **While a module is opened (between open and close), we can declare operations and equations for doing verification.**

```
NAT+> open (NAT+ + EQL)
-- opening module NAT+.. done.
%NAT+ + EQL> op n : -> Nat .
%NAT+ + EQL> eq n = 0 .
%NAT+ + EQL> red +(n, n) = 0 .
*
-- reduce in %NAT+ + EQL : +(n,n) = 0
true : Bool
(0.000 sec for parse, 4 rewrites(0.000 sec), 4 matches)
%NAT+ + EQL> close
NAT+>
```

Arbitrary element

- **After opening a module, a declared constant operation**
  `op e : -> S .`
  stands for an arbitrary element of the sort S **whose scope is from its declaration to the end of a proof score (i.e.** close**).**

```
NAT+> open (NAT+ + EQL)
-- opening module NAT+.. done.
%NAT+ + EQL> op n : -> Nat .
%NAT+ + EQL> red +(0, n) = n .
-- reduce in %NAT+ : +(0,n) = n
true : Bool
(0.000 sec for parse, 2 rewrites(0.000 sec), 2 matches)
%NAT+ + EQL> close
NAT+>
```

This is a proof score for the claim that **`+(0, N) = N`** for any natural number ***N***. Since the reduction returns "**`true`**", it holds.

Declaring assumptions

nat+ps

- **`While a module is opening, a declared equation represents an assumption of the proof score.`**

```
NAT+> open (NAT+ + EQL)
-- opening module NAT+.. done.
%NAT+ + EQL> op n : -> Nat .
%NAT+ + EQL> eq +(n, 0) = n .
_
%NAT+ + EQL> red +(s(n), 0) = s(n) .
*
-- reduce in %NAT+ : +(s(n),0) = s(n)
true : Bool
(0.000 sec for parse, 3 rewrites(0.000 sec), 5 matches)
```

This is a proof for "**`+(N, 0) = N`** implies **`+(s(N), 0) = s(N)`** for any natural number ***N***" (it holds).

Constant v.s. variable

constVsVar

- Using a variable in an equation instead of a constant makes a drastic change of meaning of the proof score. Be careful!
  - The scope of a constant is to the end of a open-close session assuming that the declared constants are fresh.
  - The scope of a variable is inside of the equation.

```
open (NAT+ + EQL)
op n : -> Nat .
eq +(n, 0) = n .
red +(s(n), 0) = s(n) .
close
```

```
open (NAT+ + EQL)
var N : Nat .
eq +(N, 0) = N .
red +(s(N), 0) = s(N) .
close
```

Constant: $\forall N{:}Nat.\ [+(N,0)=N \Rightarrow +(s(N),0)=s(N)]$

Variable: $\forall N{:}Nat.[+(N,0)=N\ ] \Rightarrow \forall N{:}Nat.[\ +(s(N),0)=s(N)]$

Mathematical Induction over Natural Numbers

Goal: Prove that for any natural number $n \in \{0, s\ 0, s\ s\ 0, \ldots\}$ $P(n)$ is true

Induction Scheme:

$$\frac{P(0) \qquad \forall n \in N.[P(n) \Rightarrow P(s\ n)]}{\forall n \in N.P(n)}$$

Concrete Procedure: (induction with respect to n)

1. Prove $P(0)$ is true
2. Assume that $P(n)$ holds, and prove that $P(s\ n)$ is true

Induction

- The following is a proof score of "$\forall$`n:Nat.+(n,0) = n`":

```
open (NAT+ + EQL)
red +(0, 0) = 0 .
op n : -> Nat .
eq +(n, 0) = n .
red +(s(n), 0) = s(n) .
close
```

Base case

Induction step

```
-- opening module (NAT+ + EQL).. done.
%NAT+ + EQL> -- reduce in %NAT+ + EQL : +(0,0) = 0
true : Bool
(0.000 sec for parse, 2 rewrites(0.000 sec), 2 matches)

-- reduce in %NAT+ + EQL : +(s(n),0) = s(n)
true : Bool
(0.000 sec for parse, 3 rewrites(0.000 sec), 5 matches)
%NAT+ + EQL>
NAT+>
```


Complete proof score

nat+ps

```
--> This is a proof of +(N, 0) = N
open (NAT+ + EQL)
--> Base case
red +(0, 0) = 0 .
--> Induction step
op n : -> Nat .
eq +(n, 0) = n . -- I.H.
red +(s(n), 0) = s(n) .
close
```

```
NAT+> in nat+ps.mod
processing input : /.../proof.mod
--> This is a proof of +(N, 0) = N
-- opening module NAT+ + EQL .. done.
--> Base case
-- reduce in %NAT+ + EQL : +(0,0) = 0
true : Bool
(0.000 sec for parse, 2 rewrites(0.000 sec), 2 matches)
--> Induction step_*
-- reduce in %NAT+ + EQL : +(s(n),0) = s(n)
true : Bool
(0.000 sec for parse, 3 rewrites(0.000 sec), 5 matches)
NAT+>
```