A Constructive Modeling Language for Object Oriented Information Systems

M.Ornaghi, M.Benini, M.Ferrari, C.Fiorentini, A.Momigliano

Preview: the modeling language COOML

Two abstractions:

- **data types**: data + operations
  - semantics of programming languages, algorithms, constructive program synthesis, ...
- **data models**: *meaning of structured data*
  - data bases, OO modeling, WEB, ...

COOML is a modeling language for OO systems

- ***work in progress***
- the focus: ***a data model*** for OO systems, based on the constructive logic E* (Miglioli 89)

Overview

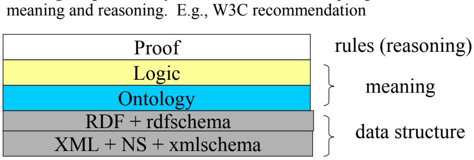
- Motivations
- The modeling language COOML through a toy example
- The logic E*
- Conclusions

1. Motivations

- Data models: ways of structuring and elaborating data according to their meaning
- Modeling languages: based on non-domain-specific data models. Examples:
  - ER data model in DB (recently, XML data bases)
  - UML in OO
  - semantic nets in AI
  - RDF, DAML+OIL (semantic WEB)
  - ....

Motivations (continued)

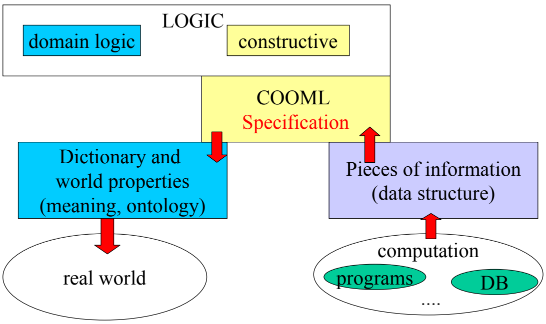
- ***Problems***:
  - Babel Tower. Different data models or *no data model*. Multiple meanings.
  - trade off expressive power / computability
  - dealing with incomplete information (e.g., null values in DB)
- ***Existing Proposals***. Layered architectures decoupling data structure, meaning and reasoning. E.g., W3C recommendation



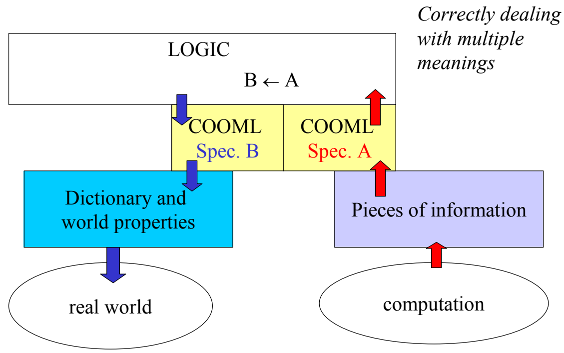
Motivations (continued)

- COOML is based on the logic E*, where
  - there is a ***domain logic layer***, for expressing properties of the problem domain and reasoning on it (using e.g. classical logic)
  - there is a ***constructive layer***, with decoupled
    - pieces of information (data and computing)
    - formulas (meaning and reasoning)
- This allows us to deal with partial information and multiple meanings and to partially overcome the trade-off expressive power/computability

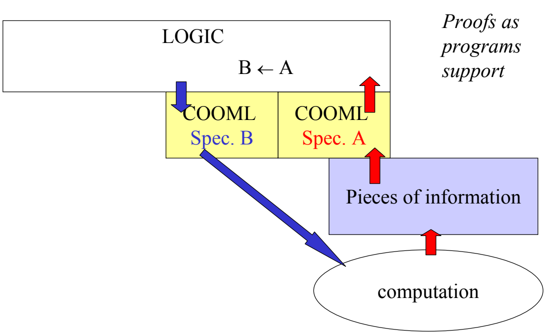
Motivations : the general architecture



Motivations : why constructive logic a)

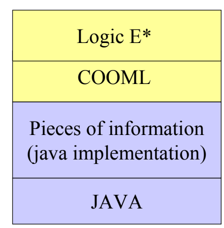


Motivations : why constructive logic b)



An experiment: a Java-like COOML

- Why OO?
  - The actual programming paradigm
  - A data model taking into account UML, OCL, JML
  - OO supports locality and reuse
    - is "local reasoning" simpler?



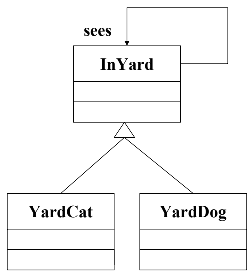
A toy example: building a specification

- **The problem domain:** *in a courtyard, there are dogs, cats, ...*
- **Dictionary and problem domain knowledge**
  - x.inYard(), x.cat(), x.dog(), x.sees(y), ... defined in terms of the real world
  - **Axioms:**
    $\forall$ x: x.cat() $\wedge$ x.inYard() $\rightarrow$
    (x.runsAway() $\leftrightarrow$ ($\exists$ y: y.dog() $\wedge$ y.inYard () $\wedge$ x.sees(y)))
    ....
  - **Java data types**

...toy example: building a specification

- Dictionary and axioms follow problem analysis and include
  - the relevant knowledge on the problem domain
  - the chosen data types
  - a formal/informal domain specification language
- Constructive Specifications use a separate Java Like Notation (JLN)
  - **true**{*boolean dictionary expression*}
    - we will omit true{...} when no confusion may arise
  - **And**{....}, **Or**{.....}, **For**{x|A: ....}, **Exi**{....}
    - **For**{x|A: ....} bounded quantification, A a finite domain

UML



COOML

Class InYard{
true{this.YardCat() ↔ this.cat()}
true{this.YardDog() ↔ this.dog()}
For{x | x.Obj():
Or{this.sees(x) ∧ x.InYard();
¬ this.sees(x);}}
}

Class YardCat extends InYard{
}

Class YardDog extends InYard{
}

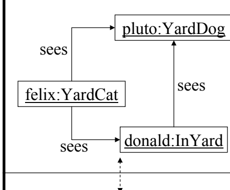
Underlying axiomatisation

Domain: true{
Problem Domain Axioms +
this.InYard() → (this.YardCat() ↔ this.cat());
this.InYard() → (this.YardDog() ↔ this.dog());
this.YardCat() → this.InYard();
this.YardDog() → this.InYard();}

Constructive:
Context this.InYard():
For{x | x.Obj(): Or{true{this.sees(x) ∧ x.InYard()};
true{¬ this.sees(x);}} }

.. toy example: data and meaning

- Data to represent pieces of information: lists containing Java data
  - every spec. S implicitly defines an information type Info(S)
- Semantics. Let S be a spec. and d ∈ Info(S):
  - S gives meaning to d, symmetrically d is an explanation of S
  - I ||= d : S indicates that the explanation is true in I
    I an interpretation, i.e., a world state



UML

COOML

PIECES OF INFORMATION
felix: ((pluto (1 true))
(felix (2 true))
(donald (1 true)))

pluto: ((pluto (2 true))
(felix (2 true))
(donald (2 true)))

donald: ((pluto (1 true))
(felix (2 true))
(donald (2 true)))

MEANING IN THE CURRENT CONTEXT:

Context this.inYard():
For{x | x.Obj():
Or{this.sees(x) ∧ x.InYard();
¬ this.sees(x);}}

A formula gives meaning to pieces of information of its type.
Symmetrically: a piece of information explains a formula.

this=felix: ((pluto (1 true))
(felix (2 true))
(donald (1 true)))

this.InYard():
For{x | x.Obj():
Or{this.sees(x) x.InYard();
¬ this.sees(x);}}

x=pluto
(**1**,true)
i.e. felix.sees(pluto)
∧ pluto.InYard()

x=felix
(**2**,true)
i.e. ¬ felix.sees(felix)

x=donald
(**1**,true)
i.e. felix.sees(donald)
∧ donald.InYard()

.. toy example: using logic for

- multiple meanings
- correct data transformations

The same pieces of information may have multiple meanings

Context this.YardCat():
For{x | x.Obj(): Or{**this.sees(x) ∧ x.InYard();** ¬this.sees(x);}}

↓ problem domain logic

For{x | x.Obj(): Or{**x.dog() → this.runsAway();** ¬this.sees(x);}}

THE SAME PIECES OF INFORMATION
felix: (pluto (1 true))
(felix (2 true))
(donald (1 true))

ANOTHER MEANING IN THE CURRENT CONTEXT:
For{x | x.Obj():
Or{x.dog() → this.runsAway();
¬this.sees(x);} }

i.e.:
pluto.dog() → felix.runsAway()
¬felix.sees(felix)
donald.dog() → felix.runsAway()

Constructive implication as correct information transformation:

Context this.YardCat():
For{x | x.Obj(): Or{this.sees(x)∧ x.InYard(); ¬this.sees(x);}}

↓

Or{this.runsAway(); ¬this.runsAway();}}

TRANSFORMED PIECES OF INFORMATION

felix: (1 true)

TRANSFORMED MEANING IN THE CURRENT CONTEXT:

Or{this.runsAway();
¬this.runsAway();}}

i.e.: felix.runsAway()

Proving: contextual Proofs

Context YardDog:
Theo. 1: **this.dog() ∧ this.inYard()**
from: this.YardDog(), class hierarchy

Context YardCat:
Theo. 2: **this.cat() ∧ this.inYard()** // similar toTheo.1

Proving: intermediate Logics

Grzegorzyk principle in our restricted syntax
(G) For{x|A(x): Or{B(x); C}} ==> Or{C; For{x|A(x): B(x)}}

A constructive iteration principle**.** For example it allows us to prove:

Context YardCat:
Theo. 3: **Or{this.runsAway(), ¬ this.runsAway()}**
proof:
For{x | x.Obj(): Or{this.sees(x) ∧ x.InYard(); ¬ this.sees(x)}} // inherited
For{x | x.YardDog(): Or{this.sees(x); ¬ this.sees(x)}} // class hierarchy
For{x | x.InYard () ∧ x.dog(): Or{this.sees(x); ¬ this.sees(x)}} // Theo 1
Or{Exi{x: x.InYard () ∧ x.dog() ∧ this.sees(x) ;
¬(∃ x: x.InYard () ∧ x.dog() ∧ this.sees(x));} **// using G**
Or{this.runsAway(), ¬ this.runsAway()}
// this.cat() ∧ this.inYard(), domain axioms

.. toy example: Java implementation ..

```
class InYard{
InYardPty pty;
//And{
        //true{this.YardCat() ↔ this.cat()}
        //true{this.YardDog() ↔ this.dog()}

        InYardInfo   sees;
              //if sees.contains(x): this.sees(x) ∧ x.InYard();
              //else: ¬ this.sees(x);
//}
}
class YardCut extends InYard{}
class YardDog extends InYard{}
```

class InYardInfo extends ForInfo{....}

- ForInfo belongs to a set of classes that represents pieces of information, with
  - query methods
  - update and creation methods

class InYardPty extends AndPty{}

- Pty, AndPty, ... are classes to represent constructive specifications; two methods are
  - **String explanation(Info i)**
    - extracts an explanation of **this** according to information i
  - **Map includes(Pty p):**
    - if **this** >> **p** (>> is implication in a decidable fragment of E*) the result is not null and contains a representation of a Java program correctly transforming pieces of information
    - >> may use problem-domain implications Atom* >> Atom in a package kb.*problemDomain*

.. toy example: Java implementation, what do we gain w.r.t. pure Java? ..

- Automatic information extraction, allowing us to:
  - print/show the meaning of the current program state in terms of the problem domain
  - deal with multiple meanings and correct information transformation
  - define a standard XML representation of structured information in distributed heterogeneous systems
  - develop the Proofs as Programs approach in our OO environment (not yet studied, but possible in principle)

On the logic E*

- E* is an intermediate constructive propositional logic (Miglioli 89) similar to Medvedev's logic of finite problems (Medvedev 62).
  E* uses a *classical truth operator* true(F).
  E* has a validity and completeness result.
- Here we consider a predicative extension with a restricted use of implication
- Syntax

  Atomic formulas Af as usual; DLF a domain logic formula

  Atom ::= true(DLF) | Af | $(\neg F)$

  Imp ::= Atom $\to$ F

  F ::= Atom | Imp | $(F \wedge F)$ | $(F \vee F)$ | $(\exists x\ F)$ | $(\forall x\ F)$

On E*: pieces of information and their truth

Each formula F defines a set of pieces of information Info(F)

Let I be a domain interpretation, I |= H (***ground***) be a truth relation in the domain, and $h \in$ Info(F): then I ||= h:H iff:

- I ||= t:Atom iff I |= Atom
- I ||= (a_1,a_2):$A_1 \wedge A_2$ iff I ||= a_1:A_1 and I ||= a_2:A_2
- I ||= (i,a_i):$A_1 \vee A_2$ iff I ||= a_i:A_i
- I ||= (t,a): $\exists x\ A(x)$ iff I ||= a:A(t)
- I ||= f: $\forall x\ A(x)$ iff I ||= f(t):A(t) for every ground t and I |= $\forall x\ A(x)$
- I ||= (t,b): Atom $\to$ B iff I ||= t:Atom entails I ||= b:B

On E*: (simplified) constructive validity

Constructive consequence: ***Ax*** ||= H iff there is a map m such that for every *ax* in Info(***Ax***), we have:

m(ax) in Info(H) and, for every interpretation I,

I ||= *ax*: ***Ax*** entails I ||= m(*ax*):H

On E*: validity (completeness?)

Calculus, with a classical domain logic: Int + Cl + KP + IP + G

Int: intuitionistic rules specialized to the restricted syntax for $\to$

Cl: a classical proof of Γ |- Atom
is also an E* proof (and nothing else)

KP: (true(A) $\to$ B $\vee$ C) |- (true(A) $\to$ B) $\vee$ (true(A) $\to$ C)

IP: (true(A) $\to \exists$x B(x)) |- $\exists$x (true(A) $\to$ B(x))

G: $\forall$x(Atom(x) $\to$ B(x) $\vee$ C) |- $\forall$x(Atom(x) $\to$ B(x)) $\vee$ C
Proofs as Programs works with a strong hypothesis on Atom(x)
[a kind of generalized quantifier]

Let ***Ax*** be a set of atoms:

Validity: ***Ax*** |- F ==> ***Ax*** ||= F

Completeness (with G?): ***Ax*** ||= F ==> ***Ax*** |- F

Conclusions

- We have presented a work in progress
- We believe that the approach is interesting and potentially fruitful
  - a logical model for semantically annotated structured data
  - information extraction and information transformation (beyond SQL)
  - proofs as programs
- There is a partial Java implementation including information types, the Java representation of the specifications, the extraction of explanations, a first partial version (in Java) of the implication >>

Conclusions

- Future work:
  - predicative E* (all of us)
  - link to semantic WEB (Benini)
  - Java implementation (students of Ornaghi, Ferrari, Fiorentini)
  - theorem proving (Momigliano)
  - proofs as programs, to be studied