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
  - databases, 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

- Domain logic
- Dictionary and world properties (meaning, ontology)
- Pieces of information (data structure)
- COOML Specification
- REAL WORLD
- Computation
- Programs

Motivations: why constructive logic a)

- Correctly dealing with multiple meanings
- B ← A

Motivations: why constructive logic b)

- Proofs as programs support
- B ← A

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:
  - ∀ x: x.cat() ∧ x.inYard() → (x.runsAway() ↔ ∃ y: y.dog() ∧ y.inYard() ∧ 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}
  - For [x:A: ...] bounded quantification, A a finite domain
.. toy example: data and meaning

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

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

.. 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);}
TRANSFORMED MEANING IN THE CURRENT CONTEXT:
felix: (1 true)

TRANSFORMED PIECES OF INFORMATION
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 to Theo. 1

Proving: intermediate Logics

Grzegorzyk principle in our restricted syntax
(G) For [A(x): Or [B(x); C]] ⇒ Or [C; For [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

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 \land F) | (F \lor F) | (\exists x F) | (\forall x F) 

On $E^*$: (simplified) constructive validity

Constructive consequence: $ Ax \models 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 \models ax \iff m(ax) \models I $.

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 \models H $ (ground) be a truth relation in the domain, and $ h \in Info(F) $.

- $ I \models t : Atom \iff I \models Atom $.
- $ I \models (a_1, a_2) : A_1 \land A_2 \iff I \models a_1 : A_1 $ and $ I \models a_2 : A_2 $.
- $ I \models (t, a) : \exists x A(x) \iff I \models a : A(t) $.
- $ I \models \forall x A(x) \iff I \models \forall t : A(x) $.
- $ I \models (t, b) : Atom \to B \iff I \models t : Atom \entails I \models b : B $.

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 $ \Gamma \vdash \text{Atom} $ is also an $ E^* $ proof (and nothing else).

KP: $ (true(A) \to B \lor C) \vdash (true(A) \to B) \lor (true(A) \to C) $.

IP: $ (true(A) \to \exists x B(x)) \vdash \exists x (true(A) \to B(x)) $.

G: $ \forall x (Atom(x) \to B(x)) \lor C \vdash \exists x (Atom(x) \to B(x)) \lor 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 \models F \implies Ax \models F $.

Completeness (with $ G $): $ Ax \models F \implies Ax \models F $.

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

Conclusions

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