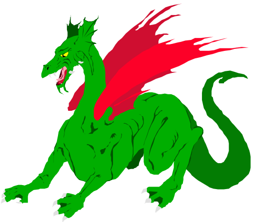
Formal Requirements Modeling Languages

From Informal to Formal Conceptual Models
General-Purpose Formal Specification Languages
The Requirements Modeling Language (RML)
GLIDER



Formality: From Informal to Formal

Informal box-and-arrow notations, minimal syntax, no ontology, no semantics	*e.g., graphs*
Informal box-and-arrow notations, minimal syntax, an ontology, no semantics	*e.g., SADT, UML*
Formal box-and-arrow notations, with an ontology, syntax, and semantics	*e.g., extended E-R, (parts of) UML*
Formal conceptual model, with an assertion language for specifying rules and constraints	*e.g., Classic, RML, KAOS*

Formal Modeling Notations

- *A notation is* ***formal*** *if it comes with a* ***formal set of rules*** *which define its* ***syntax*** *and* ***semantics****.*
- *These rules can be used to determine if an expression is syntactically or semantically well-formed.*
- ***BUT****, keep in mind that for many situations we want our models to be understandable by all stakeholders; for this reason, we want to show the stakeholders informal sketches of the (formal) models, also the results of analyses performed on it.*

Ingredients of Formal Notations?

- ***Ontology*** *- a set of assumptions about the nature of the applications being modeled.*
- ***Terminology*** *- terms for talking about the application*
  - *e.g.,* `entities` *and* `relationships` *for the E-R model, or* `time points` *and* `before`, `same`, `after` *relationships among them*
- ***Language*** *- statements one can write in the notation*
  - *e.g., well-formed formulas for First Order Logic*
- ***Abstraction mechanisms*** *-- structuring mechanisms used to organize and conceptualize a large model*
  - *e.g., generalization, aggregation, classification,...*

General-Purpose Formal Specification Notations

- Why not use **First Order Logic** or **Set Theory**? General-purpose formal mathematical notations have been in use for almost a century, are well-understood and well-known.
- However, such notations, notably First-Order Logic, were intended for formalizing mathematical theories (e.g., Number Theory), so they focus on things such as **infinity** and **deduction.**
- For real-world modeling, "common sense" type of reasoning may be more appropriate than deduction.
- Moreover, these notations don't support suitable abstractions for structuring large specifications.

Formal Specification Languages

- Were developed largely for specifying programs, rather than model parts of the world.
- Specification languages come in three basic flavours:
- **Operational** -- specification is executable abstraction of the implementation, e.g., Lisp, Prolog, Smalltalk
- **State-based** -- view a software system in terms of states and procedures, e.g., VDM, Z
- **Algebraic** -- view a program as a set of abstract data structures together with a set of operations; operations are defined in terms of algebraic axioms, e.g., Larch, CLEAR, OBJ

A Critique of General-Purpose Formal Specification Languages

- To model parts of the real world (physical, social, or psychological), it is useful to have notations which have built-in the notion of ***time***, ***entity***, ***activity***, ***agent***, ***goal***, etc.
- Formal specification languages are more appropriate for specifying ***what*** a software component needs to do during design, rather than model the world.
- Formal specification languages are also weak with respect to structuring; their structuring techniques, ***encapsulation***, ***parameterization***, motivated primarily by programming languages rather than knowledge representation and conceptual models.

RML: A Requirements Modeling Language

- Sol Greenspan's PhD thesis (DCS, 1984); Conceived as a formalization of SADT diagrams
- **Basic Idea** -- Use knowledge representation ideas to design a requirements modeling language
- **Constructs** -- include a logical sublanguage for integrity constraints and deductive rules
- **Abstractions** -- generalization, attribution, classific.
- **Time** -- requirements models as histories of the application domain [Greenspan86]
- **Metaclasses** -- which define the RML domain model

Last two features not fully addressed

An Entity Class

```
EntityClass Patients with
   necessary, unique, part
         record: MedicalRecords
   association
         location: NursingHomes; room: Rooms; physician: Doctors
   producer
         register: AdmitPatients(per<-this)
   modifier
         assessment: Assess(patient<-this)
   consumer
         release: Discharge(patient<-this) ...
   initially
         rightPlace?: record.place = location
         startClean?: paymentDue = 0
end Patients
```

An Activity Class

```
ActivityClass AdmitPatients with
        input
                per: Persons
        control
                home: NursingHome
                doc: Doctors
        output
                pat: Patients
        initially
                alreadyIn?: not(p in Patients)
        finally
                ...
        part
                getBasicInfo: Interview(whom<-per)
                place: AssignRoom(...)
                ...
end AdmitPatients
```

Assertion Classes

- *Assertion classes represent assertions with free variables.*
- *Instances of an assertion classes represent closed formulas (no free variables) which are true.*

For example,

AssertionClass IsTreatedWith with
arg
p: Patients
t: Treatments
part
c1: Available(tr<-t, at<-p.loc)
c2: Recommended(...)
end IsTreatedWith

Attribute Categories

	Entity	Activity	Assertion
Entity	part assoc ...	producer consumer modifier	initially finally invariant
Activity	input output control	part	initially finally trigger
Assertion	arg	arg trigger	part

Formalization of RML

τ: RML → L augmented, many-sorted Logic,
totally ordered, dense time points

- The Logic L includes axioms for
  - structuring mechanisms
  - built-in property categories
  - ...
- Sample axiom

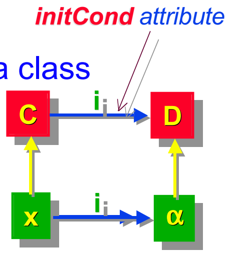
  $isA(C, D) \wedge in(x, C, t) \Rightarrow in(x, D, t)$

  "if C isA D and x is an instance of C at time t then x is an instance of D at time t"

Formalizing an Attribute Category

dvp means "definitional property value", represents an attribute ***type*** associated with a class

fvp means "factual property value", an attribute ***instance***



Assume that C is an entity and D an assertion class

$dpv(C, i) = D \wedge D \neq null \wedge InitCond(C, i) \Rightarrow$
$\forall x, t[\ Inserted(x, C, t) \Rightarrow fpv(x, i, t) \neq null$

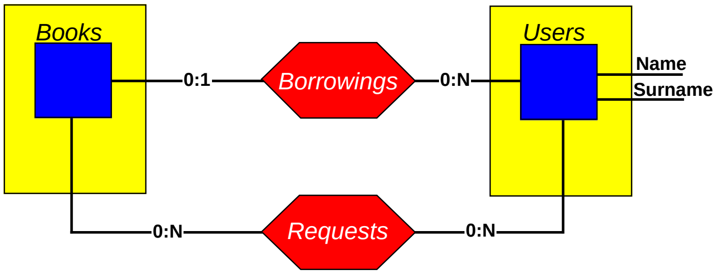
" If I is an initialCond attribute from (entity) class C to (assertion) class D then when x becomes an instance of class C, the assertion class D is true for object x"

GLIDER

- A formal language for expressing requirements
- Offers modal temporal operators for the representation of time
- Supports abstractions, including generalization, aggregation and a form of encapsulation
- Successor to ERAE and predecessor to ALBERT

[Dubois92]

Library Example



- Boxes represent entity types, polygons relationship types.
- From this sketch we can start putting together a requirements specification.

Type Definitions and Constraints

Fixed	Books:	BOOK
	Users:	USER
Varying	Borrowings:	BOOK × USER
	Requests:	BOOK × USER

Constraints -- start with connectivity, cardinality constraints,

$Borrowings(b, u) \Rightarrow Books(b) \wedge Users(u)$

- A user cannot issue a request for a book she has borrowed

  $Requests(b, u) \Rightarrow \neg Borrowings(b, u)$

- Books on the shelves for which there is a pending request, are allocated without delay:

  $\neg\exists u: Borrowings(b, u) \wedge \exists u': Requests(b, u') \Rightarrow$

  $\bigcirc(\exists u'': Borrowings(b, u''))$

More Constraints

- A book can only be allocated to a waiting user

  $Borrowings(b, u) \wedge \bullet\neg Borrowings(b, u) \Rightarrow$

  $\bullet Requests(b, u)$

- Borrowed books are returned within 30 days

  $Borrowings(b, u) \Rightarrow \blacktriangleright\!\blacktriangleright_{\leq 30days} \neg Borrowings(b, u)$

- A waiting user waits until she borrows the book she is waiting for

  $Requests(b, u) \Rightarrow \bigcirc(Requests(b, u) \vee Borrowings(b, u))$

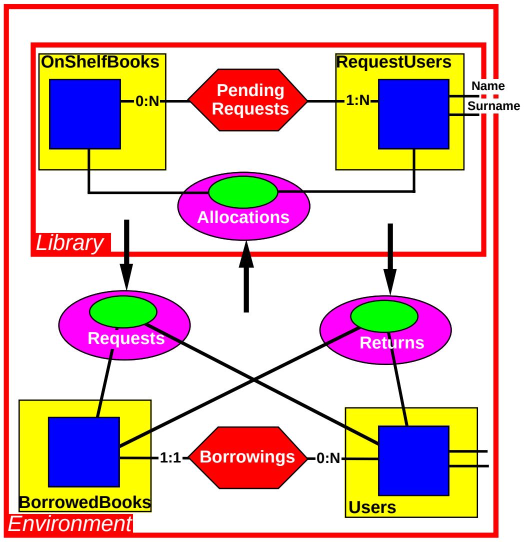
The Temporal Operators of GLIDER

❍φ - φ is true in the next state/time point
●φ - φ is true in the previous state/time point
⏩≤xφ - φ will be true sometime (within x)
⏪≤xφ - φ was true sometime (within x)
☐φ - φ will always be true
■φ - φ was always true
φ U ψ - φ is true until ψ becomes true
φ S ψ - φ has been true since ψ became true

Notation:

circle - previous/next state/time point
double arrow - sometime in the past/future
square - always in the past/future

Events in GLIDER



Parameterized Clusters

Type Cluster ResourceAlloc(RESOURCE, CONSUMER)

Fixed

Consumer: CONSUMER

Varying

Resources: RESOURCE

WaitingConsumers: CONSUMER

PendingRequests: RESOURCE × CONSUMER

Interface events

Grants: RESOURCE × CONSUMER

…

Parameterized Clusters

Constraints

- A grant occurs for an available resource and a waiting consumer
- A request is pending until the resource is granted

…

Type Cluster Library is ResourceAlloc(BOOK, USER)

…

A whole set of declarations, constraints can be derived from the parameterized cluster

Additional Reading

- [Dubois92] Dubois, E., Du Bois, P., Rifaut, A., "Elaborating, Structuring and Expressing Formal Requirements of Composite Systems", Proceedings Fourth International Conference on Advanced Information System Engineering (CAiSE'92), Manchester, May 1992.
- [Greenspan86] Greenspan, S., Borgida, A. and Mylopoulos, J., "A Requirements Modelling Language and its Logic", *Information Systems 11(1)*, January 1986.
- [Guttag85] Guttag, J., Horning, J., Wing, J., "The Larch Family of Specification Languages", IEEE Software, September 1985.