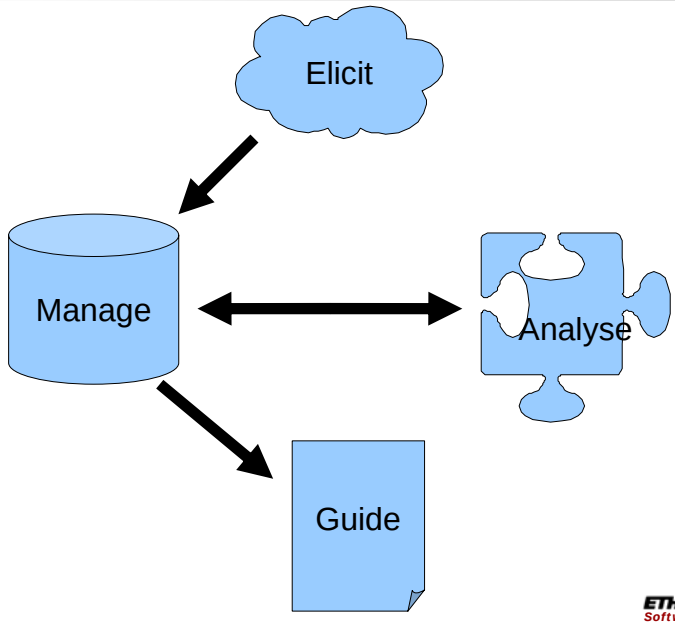
Requirements and Formal Methods

Overview

- Overview on the RE process
- What are Formal Methods?
- Advantages and Disadvantages of Formal Methods
- Formal Methods in the Requirement Process
- Mathematical Formulas and Free Text
- Tools for Formal Methods
- The B Method and Language
  - Analysis of a problem in B
- Summary

The four activities



The Analysis Problem

- Is it complete ?
- Is it sound ?

- Have I really understood it ?
- Do different people say different things ?
- Am I too abstract ? No abstract enough ?

What are Formal Methods

Formal = Mathematical

Methods = Structured Approaches, Strategies

Using mathematics in a structured way to analyze and describe a problem.

Formal Methods in Industrial Use

- Hardware
  - no major chip is developed without it
- Software
  - software verification and model checking
  - Design by Contract
  - Blast, Atelier B, Boogie
- Design
  - UML's OCL, BON, Z, state charts
- Testing
  - automatic test generation
  - parallel simulation

Why don't we like Math?

- "Very abstract."
- "Lots of Greek letters."
- "Difficult to learn and read."
- "Can communicate with a normal person."

Useful Mathematics

The type of math required consists of

- Set theory
- Functions and Relations
- First-order predicate logic
- Before-After predicates

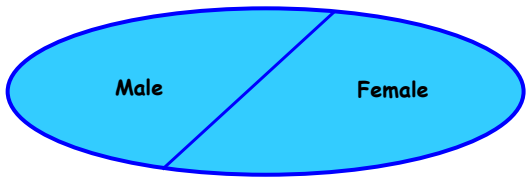
Set theory

"All humans are male or female."

$$Humans = Male \cup Female$$

"Nobody is male and female at the same time."

$$Male \cap Female = \varnothing$$



Functions and Relations

"Every customer must have a personal attendant."

$$attendant : Customers \rightarrow Employees$$

"Every customer has a set of accounts."

$$AccountsOf: Customers \rightarrow P(Accounts)$$

First-order Predicate Logic

"Everybody who works on a Sunday needs to have a special permit."

$\forall p \in Employee: workOnSunday(p) \Rightarrow hasPermit(p)$

"Every customer must at least have one account."

$\forall c \in Customers: \exists a \in Accounts: a \in AccountsOf(c)$

Before-After Predicates

"People can enter the building if they have their ID with them. When entering, they have to leave their ID card at the registration desk."

EnterBuilding(p) =
PRE
hasAuthorization(p)
carriesPassport(p)
THEN
peopleInBuilding' = peopleInBuilding ∪ { p }
passportsAtDesk' = passportsAtDesk ∪ {passportOf(p)}
not carriesPassport(p)

Advantages of Formal Methods

The advantages of using math for any analytical problem

- Short notation
- Forces you to be precise
- Identifies ambiguity
- Clean form of communication
- Makes you ask the right questions

Short Notation

Compare

"For every ticket that is issued, there has to be a single person that is allowed to enter. This person is called the owner of the ticket."

with

$$\text{TicketOwner}: \text{IssuedTickets} \rightarrow \text{Person}$$

Forced Precision

"On red traffic lights, people normally stop their cars."

What does "normally" mean? How should we build a system based on this statement? What are the consequences? What happens in the exceptional case?

Formalization Fails

Identified Ambiguity

"When the temperature is too high, the ventilation has to be switched on or the maintenance staff has to be informed."

May we do both?

$$\text{TemperatureIsHigh} \Rightarrow (\text{NotifyStaff or VentilationOn})$$

or

$$\text{TemperatureIsHigh} \Rightarrow (\text{NotifyStaff xor VentilationOn})$$

Clean Form of Communication

- Every mathematical notation has a precise semantic definition.
- New constructs can be added defined in terms of old constructs.
- Math does not need language skills and can be easily understood in an international context.

Asking the Right Questions

"Every customer has is either trusted or untrusted."

$$\forall\ c \in \text{customer}: \text{trusted}(c)\ \text{xor}\ \text{untrusted}(c)$$

"Upon internet purchase, a person is automatically registered as a new customer."

InternetPurchase (by) =

$\text{customers}' = \text{customers} \cup \{\text{by}\}$

Is the new customer trusted or untrusted ?!

A Short Remark

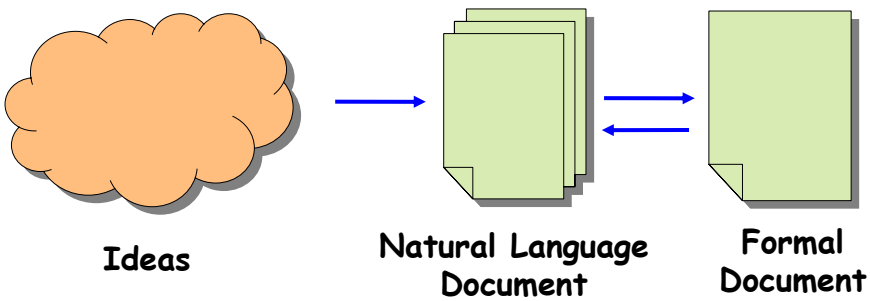
This is not programming:

- Programming describes a solution and not a problem
- Programming is constructive

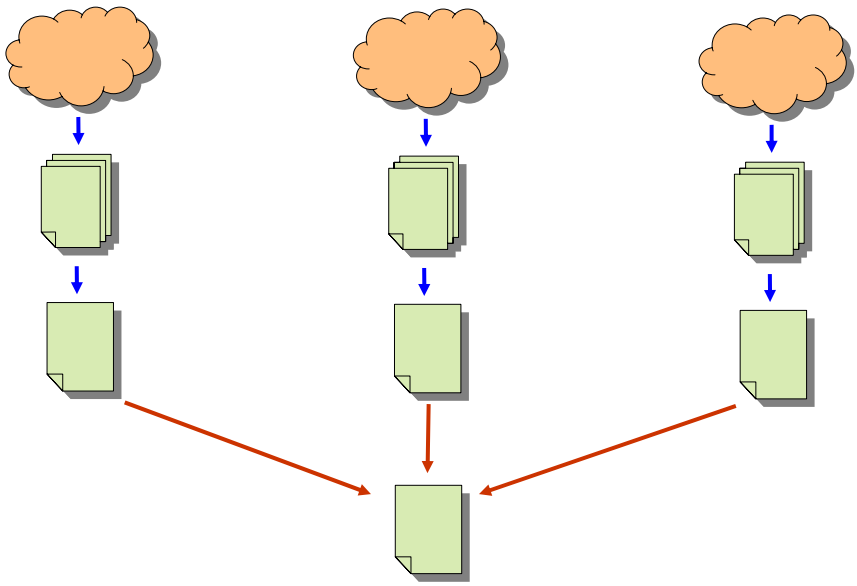
This is not design:

- We do not only describe the software
- We describe the full system (software and environment)
- No separation between software and environment
- We do so in an incremental way
- We want to understand the system

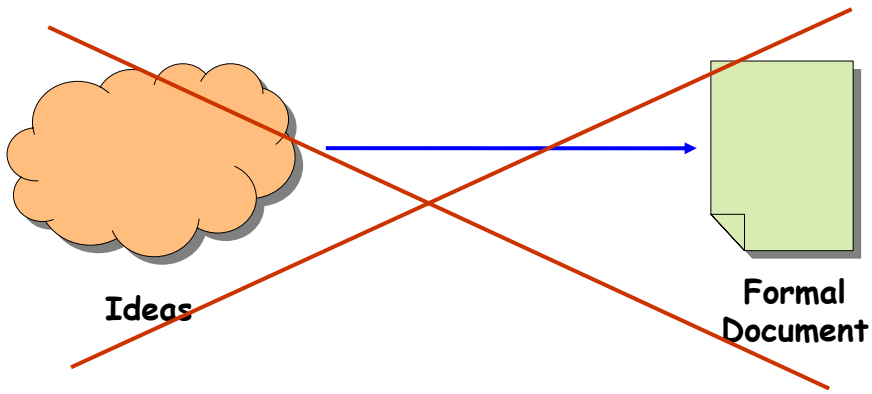
General Approach



Merging Formal Requirements



No Natural Language?

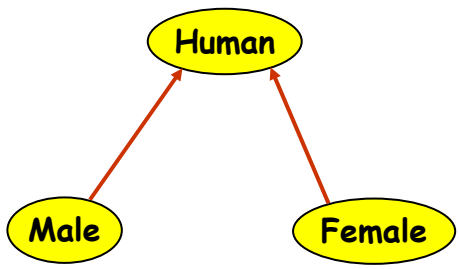


Graphical Notations

- Once we have a formal document
  - we can transform it back into a natural language document.
  - we can also transform it into a graphical document.
- There are many graphical notations out there.
- Be careful when choosing a graphical notation:
  - Does it have a well defined semantics ?
  - Does it really make things clearer than the formal or natural description ?

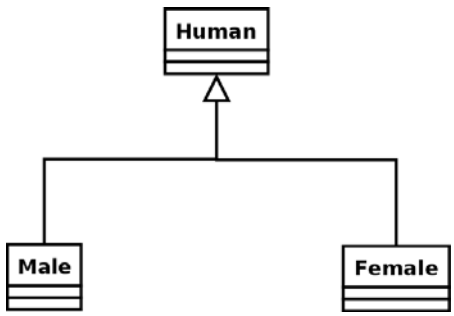
Graphical Notations (cont.)

- Sets as Classes
- Subsets as Subclasses



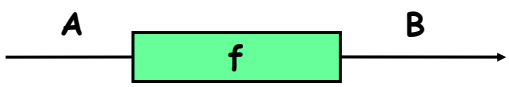
Graphical Notations (cont.)

- Sets as Classes
- Subsets as Subclasses



Graphical Notations (cont.)

- Functions



instead of $f : A \rightarrow B$

Tiny Example Problem

"The software should control the temperature of the room. It can read the current temperature from a thermometer. Should the temperature fall below a lower limit, then the heater should be switched on to raise the temperature. Should it rise above an upper limit, then the cooling system should be switched on to lower the temperature."

[...]

"Safety concern: the heater and the cooler should never be switched on at the same time."

Formal Specification

currentTemparature : INTEGER

lowerLimit: INTEGER

upperLimit: INTEGER

Formal Specification (cont.)

coolingSystem : { on, off }
heatingSystem : { on, off }

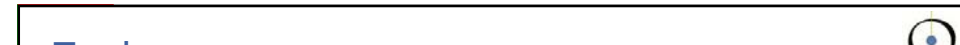
(coolingSystem = on) ⇒ (heatingSystem = off)
(heatingSystem = on) ⇒ (coolingSystem = off)

Formal Specification (cont.)

Switch on event

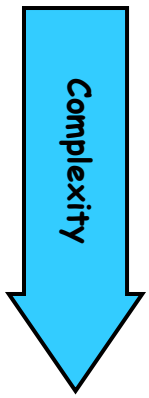
```
startCooling =
PRE
   coolingSystem = off &
   currentTemperature > upperLimit
THEN
   coolingSystem := on
END
```

Tools



Categories

- Beautifiers, Editors
- Syntax Checkers
- Type Checks
- Exercisers
- Model Checkers
- Interactive Provers
- Automatic Provers



Languages for Formal Methods

How should we formalize the requirements?

The Z notation

- Developed in the late 1970 at Oxford
- ISO Standard since 2002 (ISO/IEC 13568:2002)
- Support of large user community
- Large number of tools available

Languages for Formal Methods (cont.)

The B Method

- Simplified version of Z
- Goal: Provability
- Introduction of "Refinement"
- Industrial Strength proof tools
- Methodological Approach
- Can also be used for Design and Implementation

Languages for Formal Methods (cont.)

Other Candidates

- There are numerous languages out there
- Most tools invent an own language
- (Nearly) all are based on the same mathematical concepts
- Biggest difference: The US keyboard does not have Greek letters.

In the end, it is all just math

Pro B

Pro B is an exerciser (animator) and (limited) model-checker for the B language

- Accepts B (without refinement)
- Developed by Michael Leusche, Southampton
- http://www.ecs.soton.ac.uk/~mal/systems/prob.html

Alloy

Alloy is a full model-checker for model based on a relational logic

- Own input language modeled close to object-oriented languages
- Developed by Daniel Jackson, MIT
- http://alloy.mit.edu/

Atelier B and Click'n'Prove

Prover for the B Method

- Supports the B Method, including refinement, analysis, design and code generation
- Interactive Prover
- Developed by Jean-Raymond Abrial and Dominique Cansell, LORIA, France
- New version currently developed at the ETH as part of the EU Rodin project
- http://www.loria.fr/~cansell/cnp.html

ProB

Tool Demo

Summary

- New approach for Requirements Engineering
- Powerful tools are currently developed

Pros

- Clear and precise notation
- Makes you understand you problem
- Discoveres contradictions
- Helps you to merge requirements
- Makes you ask the right questions

Cons

- Notation requires some skills to master
- Not suitable for non-functional requirements