Software Engineering

Lydie du Bousquet

Frédéric Lang

Software Engineering – Verification using formal methods

Part I: Sequential programs

1. INTRODUCTION

Code should be « **working** » not only « **running** »

- This is why testing was introduced
- Testing is good and necessary, but it has limitations

Limitation of testing #1

- 100 % test coverage is out of reach

  too many lines of code,
  too many branches,
  parallelism,
  …

$\Rightarrow$ Many bugs may survive the testing phase

- The probability of a rare bug to occur during the software lifetime may be far above the probability that it occurs during testing

Limitation of testing #2

- Tested program may behave unpredictably
- Several possible causes:
  - Diverse execution environment
    e.g., compiler, architecture, load, …
  - Unpredictable effect of uncaught programming errors
    e.g., use of non-initialized variable, div-by-0, …
  - Intrinsic program nondeterminism
    (same input => different output)
    e.g., parallel systems (variable communication delays, asynchrony)
- It is difficult/impossible to test all situations

Example (1/3)

Test the following C program

```
int main () {
        int x = 1;
        x = x++;
        assert (x == 2);
}
```

Example (2/3)

- Tested on Linux iX86 with Gnu CC 4.4.5 compiler: test passes
- Test is **exhaustive** and **successful**!
- Program can thus be safely deployed in the customer environment… Really?

Example (3/3)

- Customer uses 32-bit SunCC/Solaris compiler
- Assertion is violated: x == 1
- Cause: ambiguity of x = x++, unspecified order between assignment x = … and increment x++

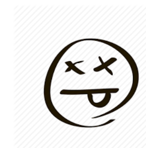
/* x == 1 */	R = x; (read x)	x = R + 1; (Increment x)	x = R; (assign x)	/* x == 1 */
vs.				
/* x == 1 */	R = x; (read x)	x = R; (assign x)	x = R + 1; (increment x)	/* x == 2 */

where R is a register used to store the initial value of x

General problem: improve predictability

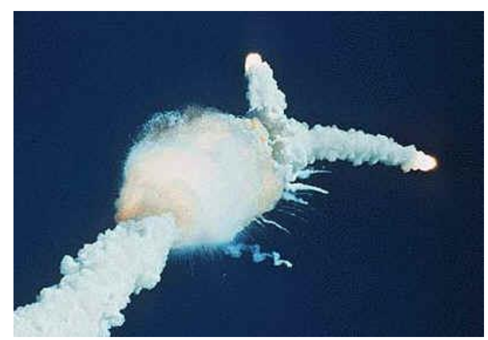
- Motivation: errors are costly

  « Normal » software: 

  Critical software:   

  e.g., avionics, aerospace, automotive, nuclear, chemicals, …

- Cost increases along development phases!
- Verification methods complementary to test are needed to find bugs early



How to improve predictability?

- Use « clean » programming languages:
  - Static semantic checks to avoid common errors (uninitialized variable, division by 0, etc.)
  - Well-defined semantics (cf. the PLCD course)
- But this is not enough to ensure that programs will always provide a correct result:
  - Need to describe the programmer’s intent: logic
  - Need to determine how intent is achieved by the program: reasoning
  - Example: Hoare logic, but not only

Programs vs. models

- Programs (programming languages) are generally too low-level for formal reasoning
- Rather use **models** of **higher abstraction level**
  - Abstract away from implementation details to focus on algorithmic problems
  - Example: **nondeterminism** used to underspecify parts that are not essential to correctness
  - Helps getting convinced of correctness
- In the sequel we indifferently use the words program or model but generally mean model of a program

Formal verification methods

- Formal = founded on mathematics
- Relies on formal languages to model:
  - Programs
  - Requirements
- **Advantages**:
  - Eliminate the risk of ambiguities
  - Offer mathematically based (rigorous) verification methods

Formal verification

- Several formal verification methods exist, with many criteria of choice
- One major criteria is whether the program is *transformational* or *reactive/concurrent*

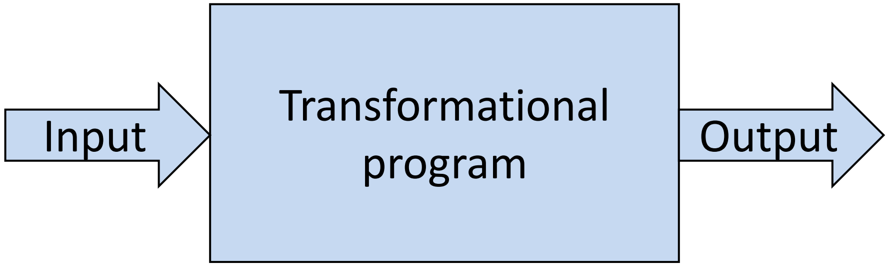
This lecture:

- Week 10: transformational programs – proof techniques
- Week 11: reactive & concurrent programs – automata-based verification

2. PROVING THE CORRECTNESS OF TRANSFORMATIONAL PROGRAMS

Transformational program

- Program (or part of a program)
  - Computes an output in function of an input
  - Essentially behaves sequentially (even though implementation may be parallel)
  - Execution should terminate (otherwise error)



- Example : programs of the While and Proc languages seen in PLCD

Proving transformational programs correct

- **Goal**: ensure that program behaves *as expected*
- Several possible notions of *as expected*
  - Absence of crash: No unexpected termination
    Examples: division by zero, out-of-bound array access, etc.
  - Correctness: A particular relation between program inputs and outputs holds
  - Termination: No infinite execution
  - Performance: Bounded usage of resources (e.g., time, memory, etc.)
- This lecture focuses essentially on program correctness

Program correctness

Proving a program correct requires:

- A formal **specification** (model) of the program
- A formal **specification** of the property that the program should satisfy
- Formal deduction rules to relate property and program (reasoning)

Methods to prove specifications of transformational programs

- Algebraic methods
- Hoare logic
- Set-based methods:
  - Z, VDM
  - B: combines ideas from Z and from Hoare logic

2.1. ALGEBRAIC METHODS

Principle of algebraic methods

- Formal framework to prove mathematical properties of programs
- Use of algebra and equational logic
- Specification by properties: objects are defined by the operations that generate or use them, as mathematical equations
- Implementation *should* be derivable from the equations
- There are many algebraic specification languages (ACT ONE, LARCH, LPG, ...)

First example: Booleans

- Boolean values are written **True/False**; **Bool** = {**True, False**}
- Opns **Not**, **Or**, **And** are defined by the following equations:

Not (**True**) = **False**

Not (**False**) = **True**

$(\forall X \in$ **Bool**$)$ **Or** (**True**, X) = **True**

$(\forall X \in$ **Bool**$)$ **Or** (**False**, X) = X

$(\forall X \in$ **Bool**$)$ **And** (**True**, X) = X

$(\forall X \in$ **Bool**$)$ **And** (**False**, X) = **False**

- We call *terms* the variables, constants and operations applied (recursively) to terms

Second example: Natural numbers

- Natural numbers are written 0, **S** (0), **S** (**S** (0)), ...; **Nat** is the set of natural numbers (**S** and 0 are called ***constructors***)
- Operations **Pred** (predecessor), +, and × are defined by the following equations:

$(\forall X \in \textbf{Nat})$	$\textbf{Pred}\ (\textbf{S}\ (X)) = X$	Déf. of **Pred**
$(\forall Y \in \textbf{Nat})$	$0 + Y = Y$	Déf. of +
$(\forall X, Y \in \textbf{Nat})$	$\textbf{S}\ (X) + Y = \textbf{S}\ (X + Y)$	
$(\forall Y \in \textbf{Nat})$	$0 \times Y = 0$	Déf. of ×
$(\forall X, Y \in \textbf{Nat})$	$\textbf{S}\ (X) \times Y = Y + (X \times Y)$	

Example of proof (1)

Prove that $\mathbf{S}\ (\mathbf{S}\ (0)) \times \mathbf{S}\ (\mathbf{S}\ (0)) = \mathbf{Pred}\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{S}\ (0))))))$ (i.e., $2 \times 2 = 5 - 1$)

Simple application of the equations in algebraic logic:

$\mathbf{S}\ (\mathbf{S}\ (0)) \times \mathbf{S}\ (\mathbf{S}\ (0))$

$= \mathbf{S}\ (\mathbf{S}\ (0)) + (\mathbf{S}\ (0) \times \mathbf{S}\ (\mathbf{S}\ (0)))$	from $(\forall X, Y \in \mathbf{Nat})\ \mathbf{S}\ (X) \times Y = Y + (X \times Y)$
$= \mathbf{S}\ (\mathbf{S}\ (0)) + (\mathbf{S}\ (\mathbf{S}\ (0)) + (0 \times \mathbf{S}\ (\mathbf{S}\ (0))))$	from $(\forall X, Y \in \mathbf{Nat})\ \mathbf{S}\ (X) \times Y = Y + (X \times Y)$
$= \mathbf{S}\ (\mathbf{S}\ (0)) + (\mathbf{S}\ (\mathbf{S}\ (0)) + 0)$	from $(\forall Y \in \mathbf{Nat})\ 0 \times Y = 0$
$= \mathbf{S}\ (\mathbf{S}\ (0) + \mathbf{S}\ (\mathbf{S}\ (0)))$	from $(\forall X, Y \in \mathbf{Nat})\ \mathbf{S}\ (X) + Y = \mathbf{S}\ (X + Y)$
$= \mathbf{S}\ (\mathbf{S}\ (0 + \mathbf{S}\ (\mathbf{S}\ (0))))$	from $(\forall X, Y \in \mathbf{Nat})\ \mathbf{S}\ (X) + Y = \mathbf{S}\ (X + Y)$
$= \mathbf{S}\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{S}\ (0))))$	from $(\forall Y \in \mathbf{Nat})\ 0 + Y = Y$
$= \mathbf{Pred}\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{S}\ (0))))))$	from $(\forall X \in \mathbf{Nat})\ \mathbf{Pred}\ (\mathbf{S}\ (X)) = X$

Example of proof (2)

- Prove that $(\forall X \in \mathbf{nat})\ X + 0 = X$
- ***By structural induction on X:*** **Base case** $X = 0$.

$X + 0 = 0 + 0$	from the hypothesis $X = 0$
$= 0$	from the equation $(\forall Y \in \mathbf{nat})\ 0 + Y = Y$
$= X$	from the hypothesis $X = 0$

- **Inductive case** suppose that $(\exists X' \in \mathbf{nat})\ X' + 0 = X'$ (***induction hypothesis***) and consider $X = \mathbf{S}(X')$.

$X + 0$	$= \mathbf{S}(X') + 0$	from the hypothesis $X = \mathbf{S}(X')$
	$= \mathbf{S}(X' + 0)$	from $(\forall X, Y \in \mathbf{nat})\ \mathbf{S}(X) + Y = \mathbf{S}(X+Y)$
	$= \mathbf{S}(X')$	from the ***induction hypothesis***
	$= X$	from the hypothesis $X = \mathbf{S}(X')$

Third example: factorial

- The factorial operation **fact** (X) of a natural number X can be characterized by the following equations:

$$\mathbf{S}\ (0) = \mathbf{fact}\ (0)$$

$$(\forall X \in \mathbf{Nat}) \qquad \mathbf{S}\ (X) \times \mathbf{fact}\ (X) = \mathbf{fact}\ (\mathbf{S}\ (X))$$

Example of proof

- Prove that

$$(\forall X \in \textbf{nat})\ X \neq 0 \Rightarrow \textbf{fact}\ (X) = X \times \textbf{fact}\ (\textbf{Pred}\ (X))$$

- Suppose $X \neq 0$. Then there exists X' such that $X = \textbf{S}\ (X')$.

$\textbf{fact}\ (X)$	$= \textbf{fact}\ (\textbf{S}\ (X'))$	(1)
	$= \textbf{S}\ (X') \times \textbf{fact}\ (X')$	(2)
	$= \textbf{S}\ (X') \times \textbf{fact}\ (\textbf{Pred}\ (\textbf{S}\ (X')))$	(3)
	$= X \times \textbf{fact}\ (\textbf{Pred}\ (X))$	(1)

(1) from $X = \textbf{S}\ (X')$

(2) from $(\forall X \in \textbf{Nat})\ \textbf{S}\ (X) \times \textbf{fact}\ (X) = \textbf{fact}\ (\textbf{S}\ (X))$

(3) from $(\forall X \in \textbf{Nat})\ \textbf{Pred}\ (S\ (X)) = X$

Exercise

- A type **nat_list** representing lists of natural numbers is defined using the constructors **Nil**: → **nat_list** and **Cons**: **nat**, **nat_list** → **nat_list**
  Ex. [] is Nil, [*n*] is Cons (*n*, Nil), [n_1, n_2] is Cons (n_1, Cons(n_2, Nil))
- Define the operation **last**: **nat_list** → **nat**, which returns the last element of a non-empty list
- Define the operation **append** : **nat**, **nat_list** → **nat_list** which appends an element at the end of a list
- Show that:
  (∀X∈ **nat**, L ∈ **nat_list**) **last** (**append** (X, L)) = X

Solution (1/2)

- **last** : **nat_list** → **nat**

  (∀ X ∈ **nat**) **last** (**Cons** (X, **Nil**)) = X

  (∀ X, Y ∈ **nat**, L ∈ **nat_list**)

  last (**Cons** (X, **Cons** (Y, L))) = **last** (**Cons** (Y, L))

- **append** : **nat**, **nat_list** → **nat_list**

  (∀ X ∈ **nat**) **append** (X, **Nil**) = **Cons** (X, **Nil**)

  (∀ X, Y ∈ **nat**, L ∈ **nat_list**)

  append (X, **Cons** (Y, L)) = **Cons** (Y, **append** (X, L))

Solution (2/2)

We first prove the following **Lemma** :

$(\forall X \in$ **nat**, $L \in$ **nat_list**) $(\exists\ Y \in$ **nat**, $L' \in$ **nat_list**) **append** (X, L) = **Cons** (Y, L')

Proof: By case on L (immediate from the definition of **append**)

We now prove by induction on L :

$(\forall X \in$ **nat**, $L \in$ **nat_list**) **last** (**append** (X, L)) = X

- **Base case** : L = **Nil**
  last (**append** (X, **Nil**)) = **last** (**Cons** (X, **Nil**)) = X
- **Inductive case** : Assume that for some L, **last** (**append** (X, L)) = X
  and show that for L' = **Cons** (Y, L), **last** (**append** (X, L')) = X

  last (**append** (X, L')) = **last** (**append** (X, **Cons** (Y, L)))
  = **last** (**Cons** (Y, **append** (X, L))) by def. of **append**
  = **last** (**Cons** (Y, **Cons** (Y', L''))) **for some** Y', L'' (**Lemma**)
  = **last** (**Cons** (Y', L'')) by def. of **last**
  = **last** (**append** (X, L))
  = X by **induction hypothesis**

Automated proofs

- In general, it is **not possible** to automate proofs
- In specific cases, proofs can be made automatic by orienting the equations in the form of ***rewriting rules***

  Example :

  fact (0) $\rightarrow$ S (0)

  fact (**S** (X)) $\rightarrow$ **S** (X) x **fact** (X)
- Certain conditions (***confluence*** and ***termination*** of the rewriting rules) must be fulfilled for the proof to be automated
- Confluence and termination of the rewriting rules cannot be proven automatically in general
- **Remark**: rewriting rules also provide implementation

Algebraic languages

Many algebraic languages exist:

- **OBJ** (J. Goguen - Université de Californie, USA - 1976)

  Ancestor of many dialects : OBJ3, CafeOBJ, BOBJ, ...
- **Larch** (J. Wing - MIT, Massachussets, USA – 1983)
- **ACT ONE** (H. Ehrig - Technische Universität Berlin, Allemagne – 1983)

  Reused in the LOTOS process algebra, followed by ACT TWO
- **PLUSS** (M.-C. Gaudel - Université Paris Sud, France – 1984)
- **LPG** (D. Bert et R. Echahed - Grenoble, France - 1984, révision 1991)
- **CASL** (*Common Algebraic Specification Language* - Initiative of a group of researchers from several origins, 1997)
- etc.

Software tools for algebraic methods

- **Larch prover** (MIT, Massachussets, USA) www.sds.lcs.mit.edu/spd/larch
  - Interactive prover for Larch (maintained but not anymore developed)
- **Maude** (SRI, Californie, USA) www.csl.sri.com/projects/maude
  - Rewrite engine based on an inheritor of OBJ
- **Elan** (INRIA, Nancy, France) http://elan.loria.fr
  - Rewrite engine
- **CASL consistency checker** (Bremen University, Germany) www.informatik.uni-bremen.de/cofi
  - Verification of the consistency of a specification
- **ACL2** (Texas University, USA) www.cs.utexas.edu/users/moore/acl2
  - Theorem prover by rewriting based on a dialect of Lisp
  - Used by AMD to verify correctness of elementary operations on the floating point numbers of the Athlon processor
  - Laureat in 2005 of the "ACM Software System Award"

Conclusion on algebraic specifications

- Algebraic specifications are a formal framework to reason on transformational programs

But

- Writing algebraic specifications is hard
  - One does not always know whether enough equations have been written to fully model the program (***completeness***)
  - One does not always know whether contradicting equations have been written (***consistency***)
  - Completeness and consistency cannot be proven automatically
- Algebraic specifications are not executable if they are not oriented as rewriting rules
- Algebraic specification languages are not well adapted to express the notion of ***program state***

Exercise (1/4)

Consider the following specification of Booleans and lists of Booleans:

Definition : true : $\rightarrow$ Bool false : $\rightarrow$ Bool and : Bool × Bool $\rightarrow$ Bool

for all $b \in$ Bool:

(*a1*) true and b = b

(*a2*) false and b = false

Definition: nil : $\rightarrow$ Bool_List cons : Bool × Bool_List $\rightarrow$ Bool_List

cat : Bool_List × Bool_List $\rightarrow$ Bool_List

for all $b \in$ Bool, l_0, l_1, $l_2 \in$ Bool_List:

(*c1*) cat (nil, l_0) = l_0

(*c2*) cat (cons (b, l_1), l_2) = cons (b, cat (l_1, l_2))

Definition: and_list : Bool_List $\rightarrow$ Bool

for all $b \in$ Bool, $l \in$ Bool_List:

(*l1*) and_list (nil) = true

(*l2*) and_list (cons (b, l)) = b and and_list (l)

We propose to show that for all l_1, $l_2 \in$ Bool_List:

(*eqn*) and_list (cat (l_1, l_2)) = and_list (l_1) and and_ list (l_2)

Exercise (2/4)

What does the cat operation do?

Exercise (2/4)

What does the cat operation do?

Concatenation of lists

example:

cat (cons (true, cons (false, nil)), cons (true, nil))

= cons (true, cat (cons (false, nil), cons (true, nil))) (*c2*)

= cons (true, cons (false, cat (nil, cons (true, nil)))) (*c2*)

= cons (true, cons (false, cons (true, nil))) (*c1*)

Exercise (3/4)

In (*eqn*), consider the particular case where
l_1 = nil and l_2 is an arbitrary list

complete the following lines:

and_list (cat (nil, l_2)) = . . . by (*c1*)

and_list (nil) and and_list (l_2)
= . . . by (*l1*)
= . . . by (*a1*)

What can you conclude about (*eqn*) when l_1 = nil?

Exercise (3/4)

In (*eqn*), consider the particular case where
l_1 = nil and l_2 is an arbitrary list

complete the following lines:

and_list (cat (nil, l_2)) = and_list (l_2) by (*c1*)

and_list (nil) and and_list (l_2)
= true and and_list (l_2) by (*l1*)
= and_list (l_2) by (*a1*)

What can you conclude about (*eqn*) when l_1 = nil?

It holds: and_list (cat (*nil*, l_2)) = and_list (*nil*) and and_list (l_2)

Exercise (4/4)

We now assume that there exists at least one list $l_3 \in$ Bool_List such that (*eqn*) holds, i.e., for all $l_2 \in$ Bool_List:

(*ih*) and_list (cat (l_3, l_2)) = and_list (l_3) and and_list (l_2)

We consider the list l_4 = cons (b, l_3) where b is an arbitrary Boolean, and we then show the following using (*ih*):

for all $l2 \in$ Bool_List:

and_list (cat (*l4*, *l2*)) = and_list (*l4*) and and_list (*l2*)

(The proof is not asked)

What can we conclude about (*eqn*)?

How is called this kind of reasoning?

What do the initials *ih* stand for?

Exercise (4/4)

We now assume that there exists at least one list $l_3 \in$ Bool_List such that (*eqn*) holds, i.e., for all $l_2 \in$ Bool_List:

(*ih*) and_list (cat (l_3, l_2)) = and_list (l_3) and and_list (l_2)

We consider the list l_4 = cons (b, l_3) where b is an arbitrary Boolean, and we then show the following using (*ih*):

for all *l2* $\in$ Bool_List:

and_list (cat (*l4*, *l2*)) = and_list (*l4*) and and_list (*l2*)

(The proof is not asked)

What can we conclude about (*eqn*)? holds for any l_1 : nil or cons (...)

How is called this kind of reasoning? reasoning by induction

What do the initials *ih* stand for? induction hypothesis

2.2. HOARE LOGIC AND DESIGN BY CONTRACT

Hoare logic

- Seen earlier in the PLCD course
- A framework for proving programs, proposed by Tony Hoare in 1969, inspired by Robert Floyd
- Mathematical formalization of deduction rules for reasoning on programs
- Motivations:
  - Rigorous definition of reasoning (teaching, research papers, …)
  - Implementation in tools

Reminder about Hoare logic

- Hoare triples { P } S { Q } where P, Q are assertions in first-order predicate logic, called **precondition** and **postcondition**
- **Meaning**: If P holds before executing S, then Q holds after executing S
- Hoare logic is about proving Hoare triples
- Proof requires additional user-given assertions called **loop variants** and **loop invariants**
- Many systems for proving sequential programs somehow rely on extensions of Hoare logic
  Example: The B method

Reminder about first-order predicate logic

- ***Terms*** represent data: constants, variables, function applications

    Examples : x, 7, **true**, **false**, **sin**(x), x < y, y + 1

- ***Formulas*** may take several forms:
    - ***Predicates***: terms that evaluate to true or false

      Examples: **true**, **false**, **even** (x), x < y, etc.

    - ***Propositional formulas***: built using predicates and the logic connectors $\wedge$, $\vee$, $\Rightarrow$, $\neg$, etc.

      Examples : $f(x, y) \wedge f(y, z) \Rightarrow f(x, z)$

    - ***Quantified formulas***: with logic quantifiers $\forall$ x: A, $\exists$ x: A

Programming with Hoare logic assertions: design by contract

- A methodology proposed by B. Meyer (1986) and first implemented in the Eiffel programming language
- Write contract (*what should be done*) together with code (*how this is done*):
  - A pre and a postcondition with each function
  - An invariant and a variant with each loop
  - In OO programming, a property on state variables that should hold before and after every method call, named class invariant
- Contracts may be checked at runtime or, if the programming and assertion languages have formal semantics, connexion to provers is possible

Contracts in SPEC#

SPEC# (Microsoft Research)

http://research.microsoft.com/en-us/projects/specsharp

- Formal language for contracts
- Extends C#, integrated in Visual Studio
- Connection to an automatic prover of logic properties
- **Homework**: watch https://www.youtube.com/watch?v=HOl11mP4V68

Factorial in SPEC#

```
class Factorial {
    static int fact (int n)
    requires n >= 0;
    ensures
      result == product{int i in (1:n+1); i};
      /* product of ints from 1 to n */
    {
        int x, r;
        x = n;
        r = 1;
        while (x > 0)
            invariant x >= 0;
            invariant x <= n;
            invariant
              r == product{int i in (x+1:n+1); i};
              /* product of ints from x+1 to n */
            {
                r *= x;
                x--;
            }
            return r;
    }
}
```

Conclusion on Hoare logic and contracts

Beyond formal proof, assertions radically change the nature of software development in several ways:

- **Design aid**: build program + arguments that justify its correctness
- **Testing and debugging**: assertions can be checked at runtime
- **Documentation**: non-ambiguous and concise description of what the program does (instead of how this is done)
- **Limitation**: No abstraction primitive

2.3. SET-BASED METHODS

Set-based specification languages

- They are formal languages appropriate to describe:
  - The notion of **program state**, defined by a set of typed **variables**
  - Program **operations**, defined by their inputs, their outputs, an **application condition** (**precondition**) and an **effect** on the state variables
- They use set-based notations, first-order predicate logic, and are derived from Hoare logic
- They generally come with a method that defines good development practices and rely on software tools

Main set-based methods

- **VDM** (*Vienna Development Method*)
  - Ancestor of set-based methods
  - Invented at the Vienna IBM laboratory in the 60's
  - Used by some industries including DCC-International (Ada compiler), British Aerospace, Adelard, ...
- **The Z notation**
  - Set-based notation proposed par J.-R. Abrial (1977) and developed by the team of Tony Hoare in Oxford
  - Standardized at ISO in 2002
- **The B method**
  - Method proposed by J.-R. Abrial in the 80's

The B method

- Method based on ***Abstract State Machines***, which unify the notions of (set-based) specification, proof and executable code
- With industrial usages:
  - **Matra Transport** and **RATP** : verification of the control system for safety equipments of Paris metro line 14 (automatic trains), from specification to Ada code generation
  - But also: **Gemalto** (smart cards), **Siemens**, **Leirios Technologies**, ...

B: Methodology

Seamless methodology from specification to executable code:

1. Program specification as an abstract state machine
2. Automatic generation of properties to be proven for the satisfaction of variants and invariants: the ***proof obligations***
3. ***Progressive refinement*** of the state machine : Manual replacement of non-executable elements by executable ones
4. Automatic generation and proof of new **proof obligations** that express the preservation of properties proven at the previous step
5. Back to point 3 until obtention of executable code

B: ASM (Abstract State Machine)

It is defined by:

- Its **name**
- Its **state variables**
- A **state invariant** (formula of first-order predicate logic): property that must be true at initialisation and remain true after each application of an operation
- A **variable initialisation** clause
- A list of **operations** that read **inputs**, return **outputs** and modify the state

ASM example: Plane boarding system (1/2)

MACHINE Plane — The abstract machine name

SEES Bool_TYPE — Imported type (library)

SETS PASSENGERS — Unspecified (abstract) set

CONSTANTS cap

PROPERTIES $cap \in N$ — Property of constants

VARIABLES onboard

INVARIANT

$(onboard \subseteq PASSENGERS) \wedge (card\ (onboard) \leq cap)$

INITIALISATION

$onboard := \varnothing$ — Generalised substitution (variable assignment)

OPERATIONS

.../...

ASM example: Plane boarding system (2/2)

.../...

Precondition

OPERATIONS

Boarding (p) =
PRE (p $\in$ PASSENGERS) $\wedge$ (p $\notin$ onboard)
THEN onboard := onboard $\cup$ { p }
END;

Effect (generalized substitution)

b $\leftarrow$ Onboard (p) =
PRE p $\in$ PASSENGERS
THEN **IF** p $\in$ onboard
THEN b := TRUE
ELSE b := FALSE
END
END

END

B: Operations

An operation has the following form:

outputs ← op (***inputs***) =

PRE

precondition

THEN

effect

END

where:

- **precondition** is a formula of first-order predicate logic
- **inputs**, **outputs** are lists of local variables
- **effect** is a statement (generalised substitution)

Examples of generalised substitutions

- Deterministic assignment: X := E
- Nondeterministic assignment: X :∈ T

  X takes any value in the set T
- Sequential composition: S_1; S_2

  substitution S_1 followed by S_2
- Conditional branch: **IF** P **THEN** S_1 **ELSE** S_2 **END**
- Nondeterministic branch: **CHOICE** S_1 **OR** … **OR** S_n **END**

  Arbitrary choice among S_1, …, S_n
- Loop: **WHILE** P **DO** S_0 **VARIANT** E **INVARIANT** Q **END**
- etc.

Proof obligations

- The B method defines first-order predicate logic formulas that must be proven for the invariant to hold: the **proof obligations**
- Calculated by applying a generalized substitution S to a formula Q, written "[S] Q"
- Relationship with Hoare logic: [S] Q = the weakest precondition that S must satisfy for postcondition Q to be satisfied

  Generalization of wp (S, Q) seen in PLCD
- Proving "P $\Rightarrow$ [S] Q" is thus analoguous to proving { P } S { Q }, with S generalized to nondeterministic statements

Application of substitutions (1/3)

- $[\ X := E\ ]\ Q \equiv Q$ in which E replaces X

  Same as Q [E/X] in Hoare logic

  Example : $[\ x := 1\ ]\ (x \leq c) \equiv (1 \leq c)$

- $[\ X :\in S\ ]\ Q = \forall y\text{: } y \in S \Rightarrow [\ X := y\ ]\ Q$

  Example : $[\ x :\in N\ ]\ p(x) \equiv$

  $\forall y\text{: } y \in N \Rightarrow p(y) \equiv$

  $p(0) \wedge p(1) \wedge p(2) \wedge \ldots$

- [**CHOICE** S_1 **OR** S_2 **END**] $Q \equiv ([\ S_1\ ]\ Q) \wedge ([\ S_2\ ]\ Q)$

Application of substitutions (2/3)

- [**IF** P **THEN** S_1 **ELSE** S_2 **END**] Q $\equiv$

  $(P \Rightarrow [\ S_1\ ]\ Q) \wedge (\neg P \Rightarrow [\ S_2\ ]\ Q)$

  Example:

  [**IF** x > 0 **THEN** x := x-1 **ELSE** x := x+1 **END**] p(x) $\equiv$

  $(x > 0 \Rightarrow p(x-1)) \wedge (x \leq 0 \Rightarrow p(x+1))$

- $[\ S_1; S_2\ ]\ Q \equiv [\ S_1\ ]\ [\ S_2\ ]\ Q$

  Example: [x := 2 * z; y := x+1] p(y) $\equiv$

  [x := 2 * z] p(x+1) $\equiv$

  p((2 * z) + 1)

Application of substitutions (3/3)

- [**WHILE** P **DO** S_0 **INVARIANT** I **END**] Q ≡

  $$I \wedge (\forall \mathbf{X}) (I \wedge P \Rightarrow [S_0]\, I) \wedge (\forall \mathbf{X}) (I \wedge \neg P \Rightarrow Q)$$

  where **X** is the set of variables occurring in I and P

(we omit the **VARIANT** part here for simplification)

Example:

[**WHILE** $X \neq N$ **DO** Y := Y+X; X := X+1
INVARIANT $Y = \Sigma_{i \in 0..X-1}\, i$ **END**] $(Y = \Sigma_{i \in 0..N-1}) \equiv$

$$Y = \Sigma_{i \in 0..X-1}\, i$$

$$\wedge$$

$$(\forall X,Y,N)\ (Y = \Sigma_{i \in 0..X-1}\, i \wedge X \neq N \Rightarrow [Y:=Y+X;\ X:=X+1\ ]\ Y = \Sigma_{i \in 0..X-1}\, i)$$

$$\wedge$$

$$(\forall X,Y,N)\ (Y = \Sigma_{i \in 0..X-1}\, i \wedge X = N \Rightarrow Y = \Sigma_{i \in 0..N-1})$$

Exercise

Compute the following substitution

[**IF** X < Y **THEN** MIN := X **ELSE** MIN := Y **END**] (MIN = X)

Solution

[**IF** X < Y **THEN** MIN := X **ELSE** MIN := Y **END**] (MIN = X) ≡

(*Applying IF-THEN-ELSE substitution*)

$(X < Y \Rightarrow [MIN := X]\ (MIN = X)) \wedge (X \geq Y \Rightarrow [MIN := Y]\ (MIN = X)) \equiv$

(*Applying assignment substitutions*)

$(X < Y \Rightarrow X = X) \wedge (X \geq Y \Rightarrow X = Y) \equiv$

(*Replacing* X = X *by* **true**)

$(X < Y \Rightarrow \textbf{true}) \wedge (X \geq Y \Rightarrow X = Y) \equiv$

(*Replacing* P ⇒ **true** *by* **true**)

$\textbf{true} \wedge (X \geq Y \Rightarrow X = Y) \equiv$

(*Replacing* **true** ∧ P *by* P)

$X \geq Y \Rightarrow X = Y \equiv$

(*Replacing* P ⇒ Q *by* ¬P ∨ Q)

$X < Y \vee X = Y \quad \equiv X \leq Y$

Exercise

Compute the following expression

$$Y = \Sigma_{i \in 0..X-1}\, i \wedge X \neq N \Rightarrow [Y:=Y+X;\ X:=X+1\ ]\ Y = \Sigma_{i \in 0..X-1}\, i$$

and check that it is true

Solution

$Y = \Sigma_{i\in 0..X-1}\, i \wedge X \neq N \Rightarrow [Y{:=}Y+X;\ X{:=}X+1\ ]\ Y = \Sigma_{i\in 0..X-1}\, i \equiv$

$Y = \Sigma_{i\in 0..X-1}\, i \wedge X \neq N \Rightarrow [Y{:=}Y+X]\ Y = \Sigma_{i\in 0..X+1-1}\, i \equiv$

$Y = \Sigma_{i\in 0..X-1}\, i \wedge X \neq N \Rightarrow [Y{:=}Y+X]\ Y = \Sigma_{i\in 0..X}\, i \equiv$

$Y = \Sigma_{i\in 0..X-1}\, i \wedge X \neq N \Rightarrow Y+X = \Sigma_{i\in 0..X}\, i \equiv$

$Y = \Sigma_{i\in 0..X-1}\, i \wedge X \neq N \Rightarrow Y = \Sigma_{i\in 0..X}\, i - X \equiv$

$Y = \Sigma_{i\in 0..X-1}\, i \wedge X \neq N \Rightarrow Y = \Sigma_{i\in 0..X-1}\, i \equiv$

true

Proof obligations

We write INV for the invariant of the state machine

- The substitution INIT which initialises the variables must **establish the state invariant**

  $\Rightarrow$ proof obligation [INIT] INV

  Analogy in Hoare logic: { true } INIT { INV }

- Each operation

  $$\textit{outputs} \leftarrow \text{op}\ (\textit{inputs}) = \textbf{PRE}\ \text{P}\ \textbf{THEN}\ \text{S}\ \textbf{END}$$

  must **preserve the state invariant**

  $\Rightarrow$ proof obligation $(\text{INV} \wedge \text{P}) \Rightarrow [\ \text{S}\ ]\ \text{INV}$

  Analogy in Hoare logic: $\{\ \text{INV} \wedge \text{P}\ \}\ \text{S}\ \{\ \text{INV}\ \}$

Proof obligation examples (1/2)

PROPERTIES $cap \in N$

INVARIANT

$(onboard \subseteq PASSENGERS) \wedge (card\ (onboard) \leq cap)$

INITIALISATION

$onboard := \varnothing$

Proof obligation (initialisation) : [INIT] INV

$$\underbrace{[\ onboard := \varnothing\ ]}_{\text{INIT}}\ \underbrace{(onboard \subseteq PASSENGERS) \wedge (card\ (onboard) \leq cap\ )}_{\text{INV}}$$

Computing and proving the obligation

$[\ onboard := \varnothing\ ]$

$(onboard \subseteq PASSENGERS) \wedge (card\ (onboard) \leq cap\ )$

$=$

$(\varnothing \subseteq PASSENGERS) \wedge (card\ (\varnothing) \leq cap)$

$\Leftrightarrow$

true ✓

Proof obligation examples (2/2)

Boarding (p) =

PRE	$(p \in PASSENGERS) \wedge (p \notin onboard)$	(P)
THEN	$onboard := onboard \cup \{ p \}$	(S)
END		

Proof obligation: $INV \wedge P \Rightarrow [S]\ INV$

$(onboard \subseteq PASSENGERS) \wedge (card\ (onboard) \leq cap) \wedge$	(INV)
$(p \in PASSENGERS) \wedge (p \notin onboard) \wedge$	(P)
$\Rightarrow$	
$[onboard := onboard \cup \{ p \}]$	(S)
$((onboard \subseteq PASSENGERS) \wedge (card\ (onboard) \leq cap))$	(INV)

Computing and proving the obligation

$[onboard := onboard \cup \{ p \}]$
$((onboard \subseteq PASSENGERS) \wedge (card\ (onboard) \leq cap))$ =
$(onboard \cup \{ p \} \subseteq PASSENGERS) \wedge$
$(card\ (onboard \cup \{ p \}) \leq cap)$

Thus, assuming the hypothesis
$(onboard \subset PASSENGERS) \wedge (card\ (onboard) \leq cap) \wedge$
$(p \in PASSENGERS) \wedge (p \notin onboard)$
we must show
$(onboard \cup \{ p \} \subseteq PASSENGERS) \wedge$
$(card\ (onboard \cup \{ p \}) \leq cap)$

This does not hold when $card\ (onboard) = cap$
Conclusion: The specification is incorrect!
Missing precondition: $card\ (onboard) < cap$

B example: Voting machine (1/2)

- We specify in B a simplified voting machine that records the votes for 2 candidates, represented by the numbers 1 and 2.
- The variables votes1 and votes2 hold the number of votes for each of the candidates
- The variable cast holds the voters who have already voted

B example: Voting machine (2/2)

MACHINE Voting_Machine
SETS VOTERS
VARIABLES votes1 votes2 cast
INVARIANT $votes1 \in \mathbb{N} \land votes2 \in \mathbb{N} \land cast \subseteq VOTERS \land$
$card\ (cast) = votes1 + votes2$
INITIALISATION $votes1 := 0;\ votes2 := 0;\ cast := \emptyset$
OPERATIONS
Vote (e, n) = **PRE** $e \in VOTERS \land n \in \{1, 2\}$ **THEN**
$cast := cast \cup \{e\};$
IF $n = 1$ **THEN**
$votes1 := votes1 + 1$
ELSE
$votes2 := votes2 + 1$
END
END
END

Question n°1

- What is the proof obligation that would allow to guarantee that the initialisation establishes the invariant?
- Explain the main steps of the computation
- Explain (even informally) why this proof obligation is true or false

Response to question n°1

- For an invariant INV and an initialisation substitution INIT, the proof obligation is [INIT] INV. Here:

  $[votes1 := 0; votes2 := 0; cast := \varnothing]\ (votes1 \in \mathbb{N} \wedge votes2 \in \mathbb{N}$

  $\wedge\ cast \subseteq VOTERS \wedge card\ (cast) = votes1 + votes2)$

  $\equiv 0 \in \mathbb{N} \wedge 0 \in \mathbb{N} \wedge \varnothing \subseteq VOTERS \wedge card\ (\varnothing) = 0 + 0$

- This property is true because:
  - 0 is indeed a natural number
  - The empty set is indeed a subset of VOTERS (it is a subset of any set)
  - The cardinal of the empty set is indeed 0

Question n°2

- What is the proof obligation that would allow to guarantee that the operation Vote preserves the invariant?
- Explain the main steps of the computation
- Explain (even informally) why this proof obligation is true or false

Response to question n°2 (1/2)

- For an invariant INV, a precondition PRE and a substitution SUB, the proof obligation is INV $\wedge$ PRE $\Rightarrow$ [SUB] INV. Here:

  votes1 $\in \mathbb{N} \wedge$ votes2 $\in \mathbb{N} \wedge$ cast $\subseteq$ VOTERS $\wedge$ card (cast) = votes1 + votes2 $\wedge$ e $\in$ VOTERS $\wedge$ n $\in$ {1, 2}

  $\Rightarrow$

  [cast := cast $\cup$ {e}; **IF** n = 1 **THEN** votes1 := votes1 + 1 **ELSE** votes2 := votes2 + 1 **END**] (votes1 $\in \mathbb{N} \wedge$ votes2 $\in \mathbb{N} \wedge$ cast $\subseteq$ VOTERS $\wedge$ card (cast) = votes1 + votes2)

- The right-hand-side [SUB] INV of the implication yields:

  (n = 1 $\Rightarrow$ (votes1 + 1) $\in \mathbb{N} \wedge$ votes2 $\in \mathbb{N} \wedge$ cast $\cup$ {e} $\subseteq$ VOTERS $\wedge$ card (cast $\cup$ {e}) = votes1 + 1 + votes2) $\wedge$

  ($\neg$(n = 1) $\Rightarrow$ votes1 $\in \mathbb{N} \wedge$ (votes2 + 1) $\in \mathbb{N} \wedge$ cast $\cup$ {e} $\subseteq$ VOTERS $\wedge$ card (cast $\cup$ {e}) = votes1 + votes2 + 1)

Response to question n°2 (2/2)

- The proof obligation INV $\wedge$ PRE $\Rightarrow$ [SUB] INV cannot be proven
- One should indeed always have:
  card (cast $\cup$ {e}) = card (cast) + 1
- But if e $\in$ cast then cast $\cup$ {e} = cast, i.e., card (cast $\cup$ {e}) = card (cast) !
- For the invariant to be preserved, one should for instance strengthen the precondition of the Vote operation, to ensure that the voter has not yet cast his/her vote:
  e $\in$ VOTERS $\wedge$ e $\notin$ cast $\wedge$ n $\in$ {1, 2}

ASM refinement

- **Goal**: Transform the formal specification to executable code
- Successive manual modifications of the specification
  - Suppression of **non-executable elements**: preconditions, simultaneity, nondeterminism
  - Introduction of **control structures**
  - Transformation of **abstract data structures** (sets, relations, ...) into **programmable data structures** (arrays, files, ...)

Example : Refinement of the plane (1)

Refinement of the plane in which a seat is assigned to each passenger admitted on board

MACHINE Plane_seats

REFINES Plane

SEES Bool_TYPE

SETS SEATS = 1..cap

VARIABLES assign

assign: partial function that assigns a passenger to each occupied seat

INVARIANT

$(assign \in SEATS \mathrel{+\!\!\!\rightarrow} PASSENGERS) \wedge (onboard = rng\ (assign))$

INITIALISATION assign := ∅

initially, no seat is attributed

rng : image of a function (here, the set of passengers on board)

Example : Refinement of the plane (2)

Set of values on which the partial function is defined

OPERATIONS

Boarding (p) =

PRE (p $\in$ PASSENGERS) $\wedge$ (dom (assign) $\subset$ SEATS) $\wedge$ (p $\notin$ rng (assign))

THEN

ANY x WHERE x :$\in$ (SEATS \ dom (assign))

THEN assign (x) := p

END

END;

...

END

Extension of the function

Proof obligations of refinement

- The B method defines new proof obligations
  - To prove that the initialisation of the refined machine is compatible with the initialisation of the original machine
  - To prove that each operation of the refined machine is compatible with the corresponding operation of the original machine
  - And hence that the refined invariant is preserved by the refined machine
- Precise definition of these proof obligations is out of the scope of this course

Software tools for the B method

- Atelier B (http://www.atelierb.eu)
  - Commercial software developed by the company ClearSy
  - Several tools
    - Syntax analyser
    - Type controler
    - Proof obligation generator
    - Automated prover
    - Interactive prover
    - Translator into several programming languages
- Free software: JBTools, B4Free, ABTools, ProB, ...

Conclusion

- There exist formal methods to help developing reliable transformational programs
  - Set-based (or *model-based*) methods
  - Algebraic (or *property-based*) methods
- Formal methods provide many advantages : early error detection, quality and reliability, utilisability of formal specifications in the next steps of the software lifecycle (test, evolution)
- The knowledge of formal methods is **a plus in your practice of programming**

To go further (1/3)

- **Hoare logic**
  - Original article by Tony Hoare: An axiomatic basis for computer programming. CACM, 1969.
  - Wikipedia : http://en.wikipedia.org/wiki/Hoare_logic
- **VDM**
  - Cliff B. Jones. Systematic software development using VDM. Prentice Hall, 1986.
  - Wikipedia : http://en.wikipedia.org/wiki/Vienna_Development_Method
- **The Z notation**
  - J. M. Spivey. The Z notation (2nd edition). Prentice Hall, 1998. 168 pages.
  - David Lightfoot. Formal specification using Z (2nd edition). Palgrave, 2000. 176 pages.
  - Wikipedia : http://en.wikipedia.org/wiki/Z_notation

To go further (2/3)

- **The B method**
  - J.-R. Abrial. The B-Book, assigning programs to meanings. Cambridge University Press, 1996.
  - Much information and resources on B : http://www-lsr.imag.fr/B/Bsite-pages.html
  - Wikipedia : http://en.wikipedia.org/wiki/B-Method
- **Algebraic specifications**
  - H. Ehrig, B. Mahr. Fundamentals of algebraic specification. Springer, 1985. 321 pages.
  - Wikipedia : http://en.wikipedia.org/wiki/Algebraic_specification
- **Synthesis on formal methods**
  - Marc Frappier, Henri Habrias (editors). Software specification methods: an overview using a case study. Springer, 2000. 312 pages. http://www.dmi.usherb.ca/~spec
  - Wikipedia : http://en.wikipedia.org/wiki/Formal_Methods

To go further (3/3)

Tools based on higher-order languages and logics

- **PVS** (SRI, California, USA)
  - http://pvs.csl.sri.com
- **LCF** (Edinburgh, Scotland and Stanford, California, USA)
  - Ancestor of Isabelle and HOL
- **Isabelle** (Cambridge, UK and Munich, Germany)
- http://www.cl.cam.ac.uk/research/hvg/Isabelle
- **HOL** (University of Pennsylvania, USA)
  - Acronym of *Higher Order Logic*
  - https://www.cs.ox.ac.uk/tom.melham/res/hol.html
- **Coq** (INRIA, France)
  - http://coq.inria.fr

Competence and Knowledge which will be evaluated

- be able to
  - understand simple **algebraic specifications**, simple **abstract state machines** and **operations**
  - carry on simple **algebraic proofs**, derive and prove simple **proof obligations**
- know
  - the general notions of **precondition**, **postcondition**, **loop variant**, **loop invariant** and **state invariant**
  - **reason rigorously** on a transformational programs