Training for THE B METHOD

Level 1
overview

- **Introduction to B**
  - a formal method...
  - ...with proofs
  - the usage of B
  - foundations
  - benefits

- **Concepts of B**
  - B modules
  - B components
    - abstract machines
    - refinements
    - implementations
  - B projects

- **The B Language**
  - predicate logic
  - set theory
  - substitutions
  - data typing
  - form of components
  - Modular decomposition
introducing B

- **a formal method ...**
  - specification method based on a mathematical formalism to build models

- **... with proofs**
  - to prove that a model is consistent (in every possible case)

- **used for:**
  - systems specification
  - software development
B and Atelier B: a formal method for

- **B for Systems**
  - Goal: help to understand, specify, design, verify a system development
    - not a method to create a system, but to check it
    - requires contacts with the system creators to deeply understand the system
  - a B-System model formalizes:
    - the system (hardware and software)
    - its environment (other systems, infrastructure, procedures handled by operators)
  - covers functional logical angle of the system, not digital calculus, not real-time requirements
B and Atelier B: a formal method for

**B for developing (safety-critical) Software**

- Goal: to develop a code that complies with its specification and to be sure of it (to know exactly what is proved)
- a B-Software model formalizes the software itself, through a modules break down
- covers a subpart of the software with functional logical procedures, only for one task or thread, not low-level Operating System features, no direct input/output
B-Software: Industrial References

- **KVB: Alstom**
  Automatic Train Protection for the French railway company (SNCF), installed on 6,000 trains since 1993
  - 60,000 lines of B; 10,000 proofs; 22,000 lines of Ada

- **SAET METEOR: Siemens Transportation Systems**
  Automatic Train Control: new driverless metro line 14 in Paris (RATP), 1998. 3 safety-critical software parts: onboard, section, line
  - 107,000 lines of B; 29,000 proofs; 87,000 lines of Ada

- **Roissy VAL: ClearSy (for STS)**
  Section Automatic Pilot: light driverless shuttle for Paris-Roissy airport (ADP), 2006
  - 28,000+155,000 lines of B; 43,000 proofs; 158,000 lines of Ada
**B-System: Industrial References**

- **Peugeot Automobiles**
  - Model of the functioning of subsystems (lightings, airbags, engine, ...) for Peugeot aftersales service
  - Goal: Understanding precisely the functioning of cars to build tools to diagnose breakdowns

- **RATP (Paris Transportation)**
  - Model of automatic platform doors to equip an existing metro line
  - Goal: Verifying consistency of System Specification
B-System References

- **EADS**
  - Model of tasks scheduling of the software controlling stage separation of Ariane rocket

- **Study of a Communication Protocol**
  - Proof that the algorithm of a communication protocol complies with its requirements

- **INRS (French Institute for Workers Safety)**
  - Model of a mechanical press complying with safety requirements (protection of the hands of the press operator)
  - Building the software specification of the press controller
basic concepts

→ B is a method for specification (and possibly for programming)
  ● B formalized system properties, static description, dynamic description
→ B is based on a mathematical language
  ● predicates, Booleans, sets, relations, functions
→ B is structured
  ● the notion of module
  ● the notion of refinement
→ B is a framework for development, validated by proofs
  ● proof validation: systematic debugging
B structuring

- **the notion of modules**

  ➔ to break down a large system or software into smaller parts
  ➔ a module has a specification, where to formalize:
    - system properties
    - static description of requirements
    - dynamic description of requirements
B structuring

- **the notion of refinement**
  - a module specification is refined: it is reexpressed with more information:
    - adding some requirements
    - refining abstract notions with more concrete notions (design choice)
    - for B-software, getting to implementable code level
  - a refinement must be consistent with its specification (this should be proved)
  - a refinement may also be refined (refinement column)
  - for B-software, the final refinement is called the implementation
B structuring

module specification
module 1\textsuperscript{st} refinement
module 2\textsuperscript{nd} refinement
module 3\textsuperscript{rd} refinement

B module
B specification
Implementation or refinement
validation by the proof

**B source**

- **Automatic Generation of Proof Obligations**
  - 100%
  - 100% up to 97% of provable PO

- **Automatic Proof**
  - ≈ up to 97% of provable PO

- **False PO**
  - ≈ 10%
  - BUG correction

- **Proof Obligations examination**

- **Interactive Proof**
  - ≈ 16 PO / day

**Note:**
- The diagram illustrates the process of validation by proof, starting from the B source and involving automatic generation, automatic proof, false PO handling, and interactive proof examination. The process includes a correction for false proofs and a note on the number of provable obligations per day.
B-Software

Software requirements

Formalization

Verification

Abstract model

Implementation

Well-implementation proof

Concrete model

Translation

Code (ADA, C, ...)

Consistency proof

B model
B-System

- System requirements
- System design
- B model
  - Formalization
  - Consistency proof
  - Remarks on system doc.
  - Natural language model reformulation
B-Software structure
benefits of B-Software

- **The Abstract Model**
  - Requirements are formalized into B specifications module by module
    - Non-formal and formal specification are very close (they both express what the software should do) to minimize errors
  - Some software properties are formalized into B
    - They strengthen the B model, since we must prove that they remain true when the modules are put together

- **The Concrete Model**
  - We must prove that the concrete model complies with its specification (the abstract model)
benefits of B-Software

- **The whole Model**
  - NO classic programming error in the code (overflow, division by 0, out of range index, infinite loop, aliases)
  - A healthy program architecture
  - Unit Test are no longer used
  - Early detection of errors
  - These benefits remain even after some modifications/evolutions
traditional development cycle

- System analysis
- Specification
- Design
- Coding
- Physical modelling
- Unit tests
- Validation tests
- Integration tests
the B development cycle

- System analysis
- Physical modelling
- Specification
- Design
- Coding
- Unit tests
- Validation tests
- Integration tests
- No corrective maintenance
comparison with other languages

- **Assembly Language**
  - no static control

- **C Language**
  - limited *static* controls (e.g. typing for data size)

- **Ada Language**
  - extended *static* controls (e.g. strong typing)

- **B Method**
  - static controls + controlling the program meaning
    (by *proving* that the B specification is consistent *and* by *proving* that the B code complies with its specification)
benefits of B-System

B-System model
- The bottom line is to deeply understand the system through the B model construction
- A work in cooperation with system creators
- Early detections of errors, at system design level, producing better Software Specification

Remarks on the system
- most questions on the system arise during creation of the B model
- a few inconsistencies may also be detected through model proof (since the model should be consistent in every possible case)

Produces
- interesting remarks on the system
- a natural language reformulation of the model giving a sharp, concise and highly structured system description
The Tools

- **Atelier B (ClearSy)**
  - created to develop industrial B-Software projects
  - a set of tools integrated into a project manager tool
    - static checkers
    - automatic proof obligation generator
    - automatic provers and interactive prover
    - code translators: Ada, C, ...
  - it is also used for B-System

- **B4free (www.b4free.com)**
  - free but restricted to academic users and owners of Atelier B
  - the core tools of Atelier B + a new x-emacs interface

- **Rodin platform (September 2007)**
  - a new open platform dedicated to B-System (in construction)
ClearSy: activities related to B

- uses B-System internally to help understand, specify, verify a system development

- uses B-Software internally to develop safety-critical software (and also to finish up proof or validate proof of B-software projects)

- is part of the B community and tries to create useful processes based on B

- training sessions for the B Language and Atelier B

- development, distribution and support of Atelier B
B in education

- 30 universities/research labs currently active
- 300 graduates per year with some experience
conclusion

- $B$ is a language
- $B$ is a development method
- $B$ ensures correct systems and software
- $B$ is used successfully by industry
- $B$ is supported by a tool: Atelier $B$
- $B$ brings concrete benefits to its users
concepts of B: modules

- a B module corresponds to a subsystem model (eventually software)
- each B module manages its own data space: “data encapsulation”
  - cf. classes (object-oriented languages)
    - abstract data types
    - packages (ADA)
- a fully developed B module consists of several B components
  - an abstract machine (the module specification)
  - some possible refinements (of its specification)
  - an implementation (final refinement: B0 code)
- these components are maintained within a single B project
concepts of B: components

- **static aspect**
  - definition of the subsystem state space: *sets, constants, variables*
  - definition of static properties for its state variables: *invariant*

- **dynamic aspect**
  - definition of the initialisation phase (for the state variables)
  - definition of operations for querying or modifying the state

- **proof obligations**
  - the static properties are consistent
  - they are *established* by the initialisation
  - they are *preserved* by all operations
concepts of B: abstract machines

- an abstract machine is the **formal specification** of a software module
- it defines a **mathematical model** of the subsystem concerned
  - an abstract description of its state space and possible initial states
  - an abstract description of operations to query or modify the state
- this model establishes the **external interface** for that module
  - every implementation will conform to its specification
  - this guarantee is assured by *proves* that has to be done during the formal development process
concepts of B: an abstract machine

MACHINE
machine name

SETS
set names

CONSTANTS
constant names

PROPERTIES
predicate

VARIABLES
variable names

INVARIANT
predicate

INITIALISATION
substitution

OPERATIONS
operation definitions

END

general form

static aspect

dynamic aspect
concepts of B: refinements

- components that refine an abstract machine (or its most recent refinement)
- they add new properties to the previous math. model (more detailed properties) and make it more concrete
  - data refinement
    - introduction of new variables to represent the state variables for the refined component, with their *linking invariant*
  - algorithmic refinement
    - transformation of the operations for the refined component
- correctness of development
  - each refinement *has to* preserve the properties of the component it refines
## concepts of B: refinements

### an intermediate refinement

<table>
<thead>
<tr>
<th>general form</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>REFINEMENT</strong></td>
</tr>
<tr>
<td>machine name_n</td>
</tr>
<tr>
<td><strong>REFINES</strong></td>
</tr>
<tr>
<td>machine name</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td><strong>VARIABLES</strong></td>
</tr>
<tr>
<td>variable names</td>
</tr>
<tr>
<td><strong>INVARIANT</strong></td>
</tr>
<tr>
<td>predicate</td>
</tr>
<tr>
<td><strong>INITIALISATION</strong></td>
</tr>
<tr>
<td>initialisation refinement</td>
</tr>
<tr>
<td><strong>OPERATIONS</strong></td>
</tr>
<tr>
<td>operation refinements</td>
</tr>
<tr>
<td><strong>END</strong></td>
</tr>
</tbody>
</table>

- data for the refined component  
  (sets and constants are preserved)

- new variables with their own properties + linking invariant

- it is not possible to introduce new operations here
concepts of B: implementations

*a final refinement containing B0:*
*the B code, that can be executed*

**general form**

```
IMPLEMENTATION
  machine name_n
REFINES
  machine name
VALUES
  valuations
CONCRETE_VARIABLES
  variable names
INVARIANT
  predicate
INITIALISATION
  initialisation implementation
OPERATIONS
  operation implementations
END
```

{ values for fixed sets and constants 
  implementation variables with 
  their invariant properties 
  + linking invariant }
concepts of B: projects

- A B project is a set of linked B modules.
  - Each module is formed of components: an abstract machine (its specification), possibly some refinements and an implementation.
- The principal dependencies links between modules are:
  - **IMPORTS** links (forming a *modular decomposition* tree).
  - **SEES** links (read-only transversal visibility).
- Sub-projects may be grouped into *libraries*.
- A B project supports formal development of software (translation in Ada, C, C++)
  - Software developed in B may integrate or may be integrated with traditionally developed code.
Concepts of B: what is proved?

Initialisation call

Operation call 1

Operation call n

Implementation

Specification

the invariant does not hold

the invariant holds

is consistant with
the B language

- order of presentation
  - predicate logic
  - set theory (B expressions)
  - substitutions
  - data typing
  - form of components
  - modular decomposition
the B language: predicate logic

**Predicates**

- the way to express properties
- a predicate is a logical formula, which may or may not *hold* *(is true or is false)*
- equations, inequalities and membership of a set are simple predicates
  
  *e.g.*
  
  \[
  x = 3 \\
  5 < 2 \\
  x \in \{1, 2, 3\}
  \]

- Simple predicates may be combined by negation, conjunction or disjunction
  
  *e.g.*
  
  \[x + y = 0 \land x < y\]
the B language: predicate logic

propositions

- \( \neg P \): negation of \( P \) (logical NOT)
- \( P \land Q \): conjunction of \( P \) and \( Q \) (logical AND)
- \( P \lor Q \): disjunction of \( P \) and \( Q \) (logical OR)
- \( P \Rightarrow Q \): logical implication: \( \neg P \lor Q \)
- \( P \Leftrightarrow Q \): logical equivalence: \( P \land Q \lor \neg P \land \neg Q \)

quantified predicates

- \( \forall x . ( P_x ) \): universal quantification
- \( \exists x . ( P_x ) \): existential quantification: \( \exists x . ( \forall x . ( P_x ) ) \)
## the B language: predicate logic

**equality predicates**
- Let $x$ and $y$ be two expressions
  - $x = y$  \( x \) equal to \( y \)
  - $x \neq y$  non equality: \( n \) \( ( x = y ) \)

**inequality predicates**
- Let $x$ and $y$ be two integer expressions
  - $x < y$  \( x \) strictly less than \( y \)
  - $x \leq y$  \( x \) less than or equal to \( y \)
  - $x > y$  \( x \) strictly greater than \( y \)
  - $x \geq y$  \( x \) greater than or equal to \( y \)

<table>
<thead>
<tr>
<th>on paper</th>
<th>at the keyboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=$</td>
<td>$=$</td>
</tr>
<tr>
<td>$d$</td>
<td>$/$</td>
</tr>
<tr>
<td>$&lt;$</td>
<td>$&lt;$</td>
</tr>
<tr>
<td>$&lt;$</td>
<td>$&lt;=$</td>
</tr>
<tr>
<td>$&gt;$</td>
<td>$&gt;$</td>
</tr>
<tr>
<td>$&gt;=$</td>
<td>$&gt;=$</td>
</tr>
</tbody>
</table>
the B language: predicate logic

- set predicates
  - let $x$ be an element, and let $X$ and $Y$ be two sets
  - $x : X$ membership: $x$ is an element of $X$
  - $x / X$ non membership
  - $X ( Y$ inclusion: $X$ is a subset of $Y$
  - $X - Y$ non inclusion
  - $X \in Y$ strict-inclusion: $X \in Y \cap Y \setminus X$
  - $X _ Y$ non strict-inclusion
the B language

- set theory (B expressions)
  - sets
  - subsets
  - Boolean set
  - numeric sets
  - sets of maplets
  - relations
  - functions
  - sequences
the B language: set theory

- **explicit sets**
  - empty set: 0
  - finite set defined in extension: \{ x_1, x_2, \ldots, x_n \}
    - e.g. \{a, b, c\}
    - \{1, x+1, y-2\}
  - set of integers between \( x \) and \( y \) (interval):
    - \( z : (x..y) \)  e  \( z > x \& z < y \)
    - e.g. 1..3 = ...
    - 3..2 = ...

- **sets defined in comprehension**
  - \( \{ x \mid P_x \} \) subset of \( X \) such that the predicate \( P \) holds
  - e.g. \( \{ x \mid x : 1..5 \& x \mod 2 = 0 \} = ... \)
the B language: set theory

- **set expressions**

  - let $X$ and $Y$ be two sets
  
  - $X \cup Y$ union of $X$ and $Y$:
    
    $z : (X \cup Y)$ e $z : X \cup z : Y$
    
    e.g. $\{1, 3, 5\} \cup 1..3 = ...$

  - $X \cap Y$ intersection of $X$ and $Y$:
    
    $z : (X \cap Y)$ e $z : X \cap z : Y$
    
    e.g. $\{1, 3, 5\} \cap 1..3 = ...$

  - $X - Y$ set deference of $X$ and $Y$:
    
    $z : (X - Y)$ e $z : X \cap z / Y$
    
    e.g. $\{1, 3, 5\} - 1..3 = ...$
    
    $1..3 - \{1, 3, 5\} = ...$
subset types

- let \( X \) be a set

\( \mathcal{P}(X) \) the set (type) of subsets of \( X \):
\[
\forall x : \mathcal{P}(X) \in X \quad \mathcal{P}(X)
\]

\( \mathcal{P}_1(X) \) the set (type) of non-empty subsets of \( X \):
\[
\mathcal{P}_1(X) = \mathcal{P}(X) - \{\emptyset\}
\]
\( \text{e.g.} \quad \mathcal{P}(\{1,2,3\}) = \ldots \)
\[
\mathcal{P}_1(\{1,2,3\}) = \ldots
\]

\( \mathcal{F}(X) \) the set (type) of finite subsets of \( X \)
\[\mathcal{F}(X) = \mathcal{F}(X) - \{\emptyset\}\]

\( \mathcal{F}_1(X) \) the set (type) of finite non-empty subsets of \( X \):
\[
\mathcal{F}_1(X) = \mathcal{F}(X) - \{\emptyset\}
\]
the B language: set theory

- **Boolean constants**
  - TRUE, FALSE predefined constants

- **Boolean expressions**
  - BOOL predefined Boolean set
    
    \[
    BOOL = \{TRUE, FALSE\}
    \]

- **Boolean expressions**
  - let \( P \) be a predicate
    
    the value of \( \text{bool}(P) \) is TRUE if \( P \) holds, otherwise FALSE
    
    e.g. \( \text{bool}(P) = \text{bool}(Q) \) e ...
the B language: set theory

**numeric expressions**

- `card (X)` cardinal of $X$: its number of elements
  - *e.g.* `card ( \{1,3,5\} ) = ...`

- `max (X), min (X)` maximum, minimum of $X$
  - *e.g.* `max ( \{1,3,5\} ) = ...`
  - `min ( \{1,3,5\} ) = ...`

- `let x and y be integers, m be a non-zero natural number and n be a natural number`
  - `pred (x), succ (x)` predecessor, successor
  - `x + y, x - y` addition, subtraction
  - `x * y, x / m` multiplication, integer division
  - `x mod m` modulo
  - `x ^ n` power

*on paper* $x^n$  *at the keyboard* $x^{**n}$
the B language: set theory

- **Sets of integers**
  - $\mathbb{Z}$: set of relative integers ($>0$ and $<0$)
  - $\mathbb{N}$: set of natural integers ($>0$)
  - $\mathbb{N}_1$: set of positive natural integers ($>0$)

- **Numeric constants**
  - MAXINT: the largest implementable relative integer
  - MININT: the smallest implementable relative integer
  - INT: predefined set of implementable relative integers
    - $\text{INT} = \text{MININT} .. \text{MAXINT}$
  - NAT: predefined set of implementable natural numbers
    - $\text{NAT} = 0 .. \text{MAXINT}$
  - $\text{NAT}_1$: predefined set of strictly positive implementable natural numbers
    - $\text{NAT}_1 = 1 .. \text{MAXINT}$

<table>
<thead>
<tr>
<th>on paper</th>
<th>at the keyboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{Z}$</td>
<td>INTEGER</td>
</tr>
<tr>
<td>$\mathbb{N}$</td>
<td>NATURAL</td>
</tr>
<tr>
<td>$\mathbb{N}_1$</td>
<td>NATURAL1</td>
</tr>
</tbody>
</table>
the B language: set theory

- **maplet expression**
  - let \( x \) and \( y \) be two elements
  - \( x, y \) ordered couple or
  - \( x \_m \_y \) ‘maplet’ (\( x \) associated to \( y \))
  - \( x \_m \_y = x, y \)

- **cartesian product types**
  - let \( X \) and \( Y \) be two sets
  - \( X \_* \_Y \) cartesian product of \( X \) and \( Y \)
  - the set (type) of maplets \( x, y \) such that \( x : X \) et \( y : Y \)
  - e.g. \( \{0, 1\} \_* \_\{a, b, c\} = \ldots \)
the B language: set theory

- **relations**
  - definition: a relation from a source set \( X \) into a target set \( Y \) is a subset of the cartesian product \( X \times Y \), that is a set of maplets where the first element belongs to \( X \) and the second to \( Y \).
  - consequence: set expressions may also be applied to relations.
  - such a relation is denoted: \( R : X \rightarrow Y \).

- **relation types**
  - the set of relations from \( X \) into \( Y \): \( X \rightarrow Y = \mathcal{P}(X \times Y) \).
the B language: set theory

- **Explicit relations**
  
e.g.: 0  
  (empty relation)
  
\{2 \_\_ a,  
2 \_\_ c,  
3 \_\_ d,  
4 \_\_ b, 
4 \_\_ d\}
the B language: set theory

relational operations

→ let $R$ be a relation from $X$ into $Y$

$\text{dom} ( R )$ domain of the relation $R$ (a subset of $X$)
$X : \text{dom} ( R ) \text{ e } # y . ( y : Y \& x m y : R )$

\textit{e.g.} $\text{dom} ( \{2 m a, 2 m c, 3 m d, 4 m b, 4 m d\} ) = ...$

$\text{ran} ( R )$ codomain or range of the relation $R$ (a subset of $Y$)
$y : \text{ran} ( R ) \text{ e } # x . ( x : X \& x m y : R )$

\textit{e.g.} $\text{ran} ( \{2 m a, 2 m c, 3 m d, 4 m b, 4 m d\} ) = ...$

→ let $R$ be a relation from $X$ into $Y$, and $S$ be a subset of $X$

$R [ S ]$ image of the set $S$ through the relation $R$
$y : R [ S ] \text{ e } # s . ( s : S \& s m y : R )$

\textit{e.g.} $\{2 m a, 2 m c, 3 m d, 4 m b, 4 m d\} [ \{1,2,3\} ] = ...$
the B language: set theory

- relational expressions
  - let $X$ be a set
    
    \[
    \text{id}(X) \quad \text{identity on } X \text{ (the set of maplets } x \cdot m \cdot x, \text{ for } x : X) :
    \]
    \[
    x \cdot m \cdot x : \text{id}(X) \quad e \quad x : X
    \]
    
    e.g. $\text{id}(\{a, b, c\}) = ...$

  - let $R$ be a relation from $X$ into $Y$
    
    \[
    R^{-1} \quad \text{converse relation (inverse maplets, from } Y \text{ into } X) :
    \]
    \[
    y \cdot m \cdot x : R^{-1} \quad e \quad x \cdot m \cdot y : R
    \]
    
    e.g. $\{2 \cdot m \cdot a, 2 \cdot m \cdot c, 3 \cdot m \cdot d, 4 \cdot m \cdot b, 4 \cdot m \cdot d\}^{-1} = ...$
the B language: set theory

**composition of relational expressions**

composition of the relations $R_1$ and $R_2$: $R_1 ; R_2$

$x m z : (R_1 ; R_2)$ e $y$. ($y : Y$ & $x m y : R_1$ & $y m z : R_2$)

e.g. \{2 m a, 2 m c, 3 m d, 4 m b, 4 m d\} ; \{a m 0, c m 1, d m 1, e m 3, e m 4\}

= ...
the B language: set theory

■ filtering relational expressions

- \( S \circ R \) restriction to the set \( S \) over the domain of relation \( R \) (keeping only maplets with first elements belonging to \( S \))
  \[
  S \circ R = \text{id}(S) ; R
  \]
  e.g. \( \{1, 2, 3\} \circ \{2 \text{ m a, 2 } \text{ m c, 3 } \text{ m d, 4 } \text{ m b, 4 } \text{ m d}\} = ... \)

- \( R \circ S \) restriction to the set \( S \) over the codomain of relation \( R \) (keeping only maplets with second elements belonging to \( S \))
  \[
  R \circ S = R ; \text{id}(S)
  \]
  e.g. \( \{2 \text{ m a, 2 } \text{ m c, 3 } \text{ m d, 4 } \text{ m b, 4 } \text{ m d}\} \circ \{b, c, d\} = ... \)
the B language: set theory

- **filtering relational expressions (cont.)**
  - \( S \setminus R \) exclusion of the set \( S \) from the domain of relation \( R \)
    (removing maplets with first elements belonging to \( S \))
    \[
    S \setminus R = \text{id}(\text{dom}(R) - S) ; R
    \]
    *e.g.* \( \{1, 2, 3\} \setminus \{2 \text{ m a}, 2 \text{ m c}, 3 \text{ m d}, 4 \text{ m b}, 4 \text{ m d}\} = ... 
  
  - \( R \setminus S \) exclusion of the set \( S \) from the codomain of relation \( R \)
    (removing maplets with second elements belonging to \( S \))
    \[
    R \setminus S = R ; \text{id}(\text{ran}(R) - S)
    \]
    *e.g.* \( \{2 \text{ m a}, 2 \text{ m c}, 3 \text{ m d}, 4 \text{ m b}, 4 \text{ m d}\} \setminus \{b, c, d\} = ... 

the B language: set theory

- relational expressions (cont.)

Let $R_1$ and $R_2$ be two relations from $X$ into $Y$. The sum $R_1 + R_2$ is obtained by overloading of relation $R_1$ by relation $R_2$ to obtain the relation with maplets from $R_1$ where their first elements do not belong to $\text{dom}(R_2)$, together with all maplets from $R_2$.

$$R_1 + R_2 = (\text{dom}(R_2) \ x \ R_1) \ u \ R_2$$

E.g. $\{0 \ m \ 1, 1 \ m \ 1\} + \{0 \ m \ 0\} = ...$

$\{2 \ m \ a, 2 \ m \ c, 3 \ m \ d, 4 \ m \ b, 4 \ m \ d\} + \{1 \ m \ a, 2 \ m \ b, 2 \ m \ c, 3 \ m \ e\} = ...$
the B language: set theory

- **functions**
  - reminder
    - definition: a *relation* from a source set \( X \) into a target set \( Y \) is a subset of the cartesian product \( X \times Y \), that is a set of maplets where the first element belongs to \( X \) and the second to \( Y \)
    - consequence: set operations also apply to relations
  - special case
    - definition: a *function* from a source set \( X \) into a target set \( Y \) is a relation from \( X \) into \( Y \), such that each element of \( X \) is associated to *at most one* element of \( Y \) (but in general, the inverse of a function is not itself a function)
    - consequence: relational expressions also apply to functions
the B language: set theory

**function applications**

- let $F$ be a function, and $x$ be an element of $\text{dom}(F)$
  
  $F(x)$ the (unique) value of function $F$ for $x$
  the associated element from $\text{ran}(F)$, where $x \mapsto F(x) : F$

* e.g. $f = \{ \text{0 ma, 1 mb, 2 ma} \}$
  
  $f(1) = ...$

**functions defined by expressions**

- let $x$ be a name, $X$ be a set and $E$ be an expression (in $x$)
  
  $x . ( x : X | E(x) )$ explicit definition in the form of a $\%$-expression
  the function consisting of maplets $x \mapsto E(x)$, for $x : X$

* e.g. $\text{plus2 = } \% z . ( z : z | z + 2 )$

  $\text{plus2}(1) = ...$
the B language: set theory

function types

- definition: in general, a function from the set \( X \) into the set \( Y \) is called a **partial function**
- such a function is denoted: \( F : X \rightarrow Y \)
- \( F : X \rightarrow Y \) e \( F : X \rightarrow Y \) & \((F^{-1} ; F)(id(Y))\)

- definition: a **total function** is a function from \( X \) into \( Y \) where the domain is equal to \( X \)
- such a function is denoted: \( F : X \rightarrow Y \)
- \( F : X \rightarrow Y \) e \( F : X \rightarrow Y \) & \( \text{dom}(F) = X \)
the B language: set theory

**injective functions**

- Definition: a **partial injection** is a function from $X$ into $Y$ where each range element has one and only one antecedent.
- Such a function is denoted: $F : X \rightarrow Y$

\[
F : X \_4 \ Y \quad \text{and} \quad F^-1 : Y \_2 \ X
\]

- Definition: a **total injection** is an injection from $X$ into $Y$ where the domain is equal to $X$.
- Such a function is denoted: $F : X \rightarrow Y$

\[
X_5 Y = X_4 Y \_i \ X_3 Y
\]
the B language: set theory

- **surjective functions**

  - definition: a **partial surjection** is a function from $X$ into $Y$ where the range is equal to $Y$
  - such a function is denoted: $F : X \rightarrow Y$
    \[ F : X \rightarrow Y \quad \text{and} \quad \text{ran} (F) = Y \]
  - definition: a **total surjection** is a surjection from $X$ into $Y$ where the range is equal to $X$
  - such a function is denoted: $F : X \rightarrow Y$
    \[ X \rightarrow Y = X \rightarrow Y \quad \text{and} \quad X_3 Y \]
the B language: set theory

**bijective functions**

- definition: a **partial bijection** is a function from $X$ into $Y$ that is injective and surjective;
- such a function is denoted: $F : X \rightarrow Y$
  
  $$X \rightarrow Y = X^4 Y \upharpoonright X^6 Y$$

- definition: a **total bijection** is a total function from $X$ into $Y$ that is injective and surjective;
- such a function is denoted: $F : X \rightarrow Y$
  
  $$X \rightarrow Y = X^5 Y \upharpoonright X^7 Y$$

- the inverse of a partial bijection is ...
the B language: set theory

**sequences**

- definitions: a sequence of ‘elements’ belonging to a set $\mathcal{X}$ is a total function from an interval $1..n$ into $\mathcal{X}$, for $n : \mathbb{N}$

  the sequence then correspond to the second elements of the maplets of this function, *ordered* by their first elements

  *e.g.* $\{ 1 \, m \, a, 2 \, m \, b, 3 \, m \, c \}$

  ➔ consequence: function expressions also apply to sequences

**explicit sequences**

- $\langle \rangle$ the empty sequence

- $[x_1,..x_n]$ sequence of $\mathcal{X}$ defined by enumeration

  *e.g.* $[a, b, c] = \{1 \, m \, a, 2 \, m \, b, 3 \, m \, c\}$
the B language: set theory

**sequence operations**

- **size (S)** length of the sequence S
  - e.g. size ([ ] ) = 0
  - size ([a, b, c] ) = 3

- **first (S)** first element of S: first (S) = S(1)
  - e.g. first ([a, b, c] ) = a

- **last (S)** last element of S: last (S) = S(size(S))
  - e.g. last ([a, b, c] ) = c

- **rev (S)** reversal of the sequence S
  - e.g. rev ([a, b, c] ) = [c, b, a]

- **x_\text{k} S** insertion of x before the sequence S
  - e.g. a_\text{k} [b, c] = [a, b, c]

- **S_\text{j} x** insertion of x after the sequence S
  - e.g. [a, b]_\text{j} c = [a, b, c]
the B language: set theory

- **sequence expressions**

  - $S_1 \circ S_2$ concatenation of sequences $S_1$ and $S_2$
    
    *e.g.* $[ a, b, c ] \circ [ c, b, a ] = [ a, b, c, c, b, a ]$

  - $S_q n$ sequence comprising the first $n$ elements of $S$ at most, or $S$ itself when $n > \text{size}(S)$
    
    *e.g.* $[ a, b, c ]_q 2 = [ a, b ]$

  - $S_w n$ sequence obtained by removing the first $n$ elements of $S$
    
    *e.g.* $[ a, b, c ]_w 2 = [ c ]$

  - $\text{tail}(S)$ sequence obtained by removing the first element of $S$
    
    *e.g.* $\text{tail}([ a, b, c ]) = [ b, c ]$

  - $\text{front}(S)$ sequence obtained by removing the last element of $S$
    
    *e.g.* $\text{front}([ a, b, c ]) = [ a, b ]$
the B language: set theory

**sequence types**

- `seq ( X )` the set of sequences of `X`
- `seq1 ( X )` the set of non-empty sequences of `X`
- `iseq ( X )` the set of injective sequences of `X`
- `iseq1 ( X )` the set of non-empty injective sequences of `X`
- `perm ( X )` the set of bijective sequences of `X` (permutations on `X`)

**Example**

```
perm({a,b,c}) = \{[a,b,c], [a,c,b], [b,a,c], [b,c,a], [c,a,b], [c,b,a]\}
```
substitutions represent the transformation of data by programs

so they change the state of a system

they concern some list of variables

substitutions are mathematically defined as *predicate transformers*

the application of a substitution $S$ to a predicate $P$ is noted: $[ S ] P$

e.g. $[x := 2] (x > 1) \Leftrightarrow (2 > 1)$
the B language substitutions

- these substitutions are used in the specifications (abstract machine and its possible refinements) and also in the code (implementation) of a module.

- in specifications [Spec]: a substitution describes abstract properties of operations, they may be non-deterministic. e.g.“becomes such that” substitutions.

- in implementations [B0 Code]: only classical programming language constructs are allowed (":=", ";", IF, CASE, WHILE, procedure calls, null statement).
the B language
substitutions

➔ null substitution [Spec, B0 code]
   skip      the variables keep their values (what variable list?)

➔ "becomes equal to" substitution [Spec, B0 code]
   \( \nu := E \) the value of \( E \) is assigned to \( \nu \)
   e.g.  \( \nu := 0 \)
   \( x := y + 1 \)
   \( a, b := c, d \)
   \( f(i) := m \)
   \( r'b := n \)

➔ "becomes element of" substitution [Spec]
   \( \nu : \mathcal{X} \) an element of \( \mathcal{X} \) is assigned to \( \nu \)
   e.g.  \( \nu : (1..3) \)
the B language substitutions

“becomes such that” substitution [Spec]

- $x : (P_x)$
- the variable $x$ is assigned a value which satisfies the predicate $P_x$

  e.g. $x : (x : \text{NAT} \land x \mod 3 = 1)$

  (results of dividing $n$ by $m$, where $n : \text{N}$ and $m : \text{N}_1$)

  $q, r : (q : \text{N} \land r : \text{N} \land n = (m \cdot q) + r \land r < m)$

- the previous value of $x$ can be referenced in $P_x$ by $x$0
  e.g. $x : (x > x$0)
the B language substitutions

- **simultaneous substitutions** [Spec]
  \[ S_1 || S_2 \]
  applies the substitutions \( S_1 \) and \( S_2 \) simultaneously
  the variables modified in \( S_1 \) and \( S_2 \) must be *distinct*
  e.g.
  \[
  \begin{align*}
  x &:= 1 || y := 2 \\
  x &:= y || y := x
  \end{align*}
  \]

- **sequential substitutions** [B0 code]
  \[ S_1 ; S_2 \]
  applies the substitution \( S_1 \) *and then* the substitution \( S_2 \)
  e.g.
  \[
  \begin{align*}
  x &:= 1 ; y := 2 \\
  x &:= y ; y := x + 1
  \end{align*}
  \]
the B language substitutions

- "BEGIN" substitution (block substitution) [Spec, B0 code]
  -> BEGIN S END used to parenthesize substitutions
    e.g. BEGIN x := y || y := x END
    BEGIN x := y ; y := x + 1 END

- "VAR" substitution (block of local variables) [B0 code]
  -> VAR v₁,..., vₙ IN S END
    introduction of local variables v₁,...vₙ that may be used in substitution S
    e.g. VAR t IN t := x; x := y; y := t END
the B language substitutions

- **pre-condition** [Spec]
  - \( \text{PRE } P \text{ THEN } S \text{ END} \)
  - a substitution that may only be used when the predicate \( P \) holds
  - used to specify the properties that have to hold when calling an operation
  - *e.g.* \( \text{PRE } x : \text{NAT}_1 \text{ THEN } x := x - 1 \text{ END} \)

- **assertion** [Spec, B0 code]
  - \( \text{ASSERT } P \text{ THEN } S \text{ END} \)
  - similar to a precondition, but used to simplify the proof, by factorizing a property
"ANY" substitution [Spec]

- ANY  \( x \) WHERE  \( P_x \) THEN  \( S \) END

  apply the substitution \( S \) in which, the variables \( x \) that satisfy \( P \), can be used (in read only)

  e.g.  ANY  \( x \) WHERE  \( x : \text{NAT} \land x < 10 \) THEN  \( y := x + 1 \) END

- Note: the "ANY" substitution is very versatile

  skip  ANY  \( x \) WHERE  \( x = y \) THEN  \( y := x \) END

  \( y := a \)  ANY  \( x \) WHERE  \( x = a \) THEN  \( y := x \) END

  \( y := E \)  ANY  \( x \) WHERE  \( x : E \) THEN  \( y := x \) END

  \( y : (P_y) \)  ANY  \( x \) WHERE  \( P_x \) THEN  \( y := x \) END
the B language substitutions

- **“CHOICE” substitution** [Spec]
  - CHOICE $S_1$ OR $S_2$ ... OR $S_n$ END
  - apply one of the substitutions $S_1$, $S_2$, ..., $S_n$
  - *e.g.* CHOICE $x := x + 1$ OR $x := x - 1$ OR skip END

- **“SELECT” substitution** [Spec]
  - SELECT $P_1$ THEN $S_1$ WHEN $P_2$ THEN $S_2$ ... ELSE $S_n$ END
  - defines several branches of substitutions $S_i$ “guarded” by $P_i$
  - a substitution may be applied if its guard holds
  - if no guard holds, then the ELSE substitution is applied
  - *e.g.* SELECT $x > 10$ THEN $x := x - 10$
    WHEN $x < 10$ & $x > 0$ THEN $x := 2 \cdot (x - 10)$
    ELSE $x := x - 1$
    END
the B language substitutions

- **"IF" substitution** [Spec, B0 Code]
  
  \[
  \begin{align*}
  &\text{IF } P_1 \text{ THEN } S_1 \text{ ELSIF } P_2 \text{ THEN } S_2 \ldots \text{ ELSE } S \text{ END}
  
  &\text{the substitution applied is:}
  
  &S_i \text{ if } P_i \text{ holds and the previous predicates do not hold}
  
  &S \text{ if no predicate } P_i \text{ holds (by default } S \text{ is skip)}
  
  \text{e.g.} \quad \begin{align*}
  &\text{IF } x > 10 \text{ THEN } \\
  &\quad x := x - 10 \\
  &\text{ELSIF } x = 0 \text{ THEN } \\
  &\quad x := x + 1 \\
  &\text{ELSE } \\
  &\quad x := 1 \\
  &\text{END}
  \end{align*}
  \]
the B language substitutions

"CASE" substitution [Spec, B0 Code]

CASE \( V \) OF EITHER \( V_1 \) THEN \( S_1 \) OR \( V_2 \) THEN \( S_2 \) ... ELSE \( S_n \) END END

the substitution applied is:

- \( S_i \) if \( V \) belongs to the list of literals \( V_i \) (the \( V_j \) have to be distinct)
- \( S \) otherwise (by default \( S \) is skip)

\( e.g. \) CASE \( x \) OF

- EITHER 0, 1, 2 THEN \( x := x + 1 \)
- OR 3, 4 THEN \( x := x - 1 \)
- OR 10 THEN skip
- ELSE \( x := x + 10 \)
- END

END
the B language substitutions

- "WHILE" substitution [B0 Code]
  \[ \text{WHILE } P \text{ DO } S \text{ INVARIANT } I \text{ VARIANT } V \text{ END} \]
  
  - while loop, or iterative behaviour: while the predicate \( P \) holds, the "loop body" \( S \) is applied
  
  - the negation of \( P \) is the "exit condition" from the loop
  
  - the loop INVARIANT parts gives the properties that hold just before the loop, and after every iteration. It should give a recurrence relation on the variables modified inside the loop
  
  - the VARIANT clause defines a decreasing positive expression, in order to prove that the number of iterations is finite, and so that the loop terminates
the B language substitutions

- *operation calls* [Spec, B0 Code]
  - application of the substitution specified for the operation *op*, with replacement of its formal parameters by the actual parameters “call by value”
    - e.g.
      - `op1`
      - `op2 ( y - 1 )`
      - `x c op3`
      - `x, y c op4 ( x + 1, TRUE )`
the B language
data typing

- **data typing principles**
  - every data item must be typed before being used
  - types within the B language are based on the set theory
  - predicates, expressions and substitutions have to respect *typing rules*
  - such rules avoid obviously meaningless constructs ("don’t mix apples and oranges")

  e.g. $2 = \text{TRUE}$

- **Definition**
  - the type of a data item is the largest B set to which it belongs
the B language
data typing

- **B types**
  - every type is described in terms of *basic types* and *type constructors*
    - the basic types are
      - BOOL
      - Z
      - fixed or enumerated sets (see the **SETS** clause)
        - *e.g.* TRUE : BOOL
          - 2 : Z
    - the type constructors are
      - subsets $\mathcal{P}(T)$
      - Cartesian products $T_1 \times T_2$
        - *e.g.* \{1, 3, 5\} : $\mathcal{P}(Z)$
        - (0 m FALSE) : $Z \times BOOL$
the B language
data typing

- **how are data items typed?**
  - In general, data items are typed by *typing predicates*, which are particular predicates of the form

    \[
    \text{untyped\_data\_item} \quad \text{typing\_operator} \quad \text{typed\_expression}
    \]

    where the *typing\_operators* are `=`, `:`, `et`, `|`

    e.g. \(x: 1..10 \& y: \text{BOOL} \& z = x + 1 \& S(\text{INT})\)

    such typing predicates must be at the highest syntactic level within a conjunction list.

  - Local variables and result parameters of an operation are instead typed by *typing substitutions*.

    e.g. \(\text{VAR } t \text{ IN } ... t := x + 1 ; ... \text{ END}\)

    \(r \leftarrow \text{op}(p) = \text{PRE } p : \text{NAT}_1 \text{ THEN } ... r := p - 1 \text{ ... END}\)
the B language

- **B components (reminder)**
  - **static aspect**
    - definition of the subsystem state space: *sets, constants, variables*
    - definition of static properties for its state variables: *invariant*
  - **dynamic aspect**
    - definition of the initialisation phase (for the state variables)
    - definition of operations for querying or modifying the state
  - **proof obligations**
    - the static properties must be mutually consistent
    - they must be *established* by the initialisation
    - they must be *preserved* by all operations
the B language
form of components

- **static aspect**
  - set definitions (SETS clause)
  - constant definitions (CONSTANTS, PROPERTIES clauses)
  - variable definitions (VARIABLES, INVARIANT clauses)
  - set and constant values (VALUES clause)
  - machines with parameters (CONSTRAINTS clause)
  - textual abbreviations (DEFINITIONS clause)
  - supplementary assertions (ASSERTIONS clause)

- **dynamic aspect**
  - initialisation phase (INITIALISATION clause)
  - operation definitions (OPERATIONS clause)
form of B components: static aspect

- **set definitions**

  SETS  \( S_1; \ldots ; S_n \)

  this clause introduces new base types into a component

  ➔ a fixed set is defined by its name \( X_i \)

     *e.g.* SETS  \( STUDENTS \)

     its *value* is not yet defined, it will be given in the implementation

     eventually its value is a non-empty implementable interval

  ➔ an enumerated set is defined by its name and the list of its elements: \( X_i = \{ x_1, \ldots, x_m \} \)

     *e.g.* SETS  \( COLOR = \{ \text{Red, Green, Blue} \} \)
form of B components: static aspect

- **constant definitions**
  - \( \text{ABSTRACT\_CONSTANTS } x_1, \ldots, x_n \) (CONCRETE\_)\text{CONSTANTS } x_1, \ldots, x_n
  - these clauses introduce new constants into a component
  - a constant may be read but not modified
  - a concrete constant is directly implementable (scalar, interval, array),
    - it is automatically preserved through refinement,
    - it has to be *valued* in the implementation
  - an abstract constant is a constant of any arbitrary type,
    - it is not automatically preserved through the refinement,
    - it is not allowed in implementations
constant definitions

PROPERTIES $P_{x_1,\ldots,x_n}$
the PROPERTIES clause defines the types and other properties of the constants

e.g.
CONSTANTS $c_1$, $c_2$
ABSTRACT_CONSTANTS $c_3$
PROPERTIES
$c_1 : 0..10$ &
$c_2 : 0..10$ &
$c_1 + c_2 < 15$ &
c3 : N 3 0..15
form of B components: static aspect

- **variable definitions**
  - (ABSTRACT_)VARIABLES \( v_1, \ldots, v_n \)
  - CONCRETE_VARIABLES \( v_1, \ldots, v_n \)
  - these clauses introduce new variables into a component
  - an abstract variable is a data item of any arbitrary type, it is not automatically preserved through the refinement, it is not allowed in implementations
  - a concrete variable is a variable directly implementable (scalar or array), it is automatically preserved through refinement
form of B components: static aspect

- **variable definitions**
  
  ➔ INARIANT \( P_{v_1,...,v_n} \)
  
  *e.g.* VARIABLES

  \[
  A, B
  \]

  INARIANT

  \[
  A \ ( T \& \\
  B \ ( T \& \\
  \text{card}(A \cup B) = 1
  \]

  ➔ the INARIANT clause defines the types and other properties of the variables

  ➔ after the module initialisation, these properties remain invariant after any operation call
form of B components: static aspect

**Example**

```plaintext
MACHINE
  Register
SETS
  STUDENTS
CONCRETE_CONSTANTS
  max_students
PROPERTIES
  max_students : NAT
ABSTRACT_VARIABLES
  Enrolled
INVARIANT
  Enrolled ( STUDENTS &
        card (Enrolled) < max_students
... END
```

} machine name

} fixed set

} constant type

} variable type,
} and properties
form of B components: static aspect

- **values of fixed sets and concrete constants**
  
  > e.g. VALUES
  
  \[
  \begin{align*}
  STUDENTS &= 0..255; \\
  max\_students &= 255; \\
  transfer &= \{0 \text{ m FALSE}, 1 \text{ m TRUE}, 2 \text{ m FALSE}\}; \\
  default\_grade &= (0..255) \ast \{0\}
  \end{align*}
  \]

  > the VALUES clause is only allowed in implementation

  > it should give a value to every
    - fixed sets of the B module
    - concrete constants of the B module

  > fixed sets are eventually valued with implementable intervals
form of B components: static aspect

- **textual abbreviations**
  - the DEFINITIONS clause defines textual abbreviations, which may then be used as expressions in the rest of the current component (similar to `#define` in C language)
  - definitions may have parameters and may be factorized in definition files
    - e.g. DEFINITIONS
      - NMAX == 255 ;
      - NMAXm1 == NMAX - 1 ;
      - no(b) == bool(b = FALSE) ;
      - "mydef.def"
form of B components: dynamic aspect

- **initialisation phase**

  INITIALISATION  $S$

  this clause defines the initial values of the component variables
  initialisation has to *establish* the invariant

  *ex. : ABSTRACT_VARIABLES*

  Enrolled

  INARIANT

  Enrolled $\leq$ STUDENTS

  INITIALISATION

  Enrolled := 0
form of B components: dynamic aspect

- **operation definitions**
  - the OPERATIONS clause defines operations (B procedures or functions)
  - each operation defined in an abstract machine has to be redefined in the refinements of the abstract machine
  - it is not possible to introduce new operations within refinements (or implementations)
  - operations may have input and output parameters defined in the operation header, that are implementable
  - properties of input parameters that have to be proved when calling the operation are defined in the precondition (useful only for abstract machines)
  - output parameters are typed in the substitution of the operation specification
  - an operation is a substitution that defines how all of the component variables and the output parameters are modified
form of B components: dynamic aspect

- **operation definitions**

  ➔ syntax of operations

  \[
  op = S \\
  op(p_1,\ldots,p_n) = S \\
  r_1,\ldots,r_m \circ op = S \\
  r_1,\ldots,r_m \circ op(p_1,\ldots,p_n) = S
  \]

  ➔ operations that change some state variables are *modifying operations* (otherwise *querying operations* or *read-only*)

  ➔ operations have to *preserve* the invariant

  ➔ operation refinements or implementations have to *be consistent* with their specifications
form of B components: dynamic aspect

example (version 2 completed)

MACHINE
  Register
SETS
  STUDENTS
ABSTRACT_VARIABLES
  Enrolled
INVARIANT
  Enrolled (STUDENTS)

/* dynamic */
INITIALISATION
  Enrolled := 0

OPERATIONS
  num_enrolled =
    n := card (Enrolled);
  is_enrolled (s) =
    PRE s : STUDENTS
    THEN b := bool (s : Enrolled)
    END;
  enrol_student (s) =
    PRE s : STUDENTS - Enrolled
    THEN Enrolled := Enrolled \cup \{s\}
    END;
  withdraw_student (s) =
    PRE s : Enrolled
    THEN Enrolled := Enrolled - \{s\}
    END
END
form of B components: dynamic aspect

- **local operation**
  - a new feature to avoid too much levels of module in a project
  - to make the proof of implementations easier
  - they may be defined only in implementations
  - they may be called from implementation operations (global or local)
  - they are specified in the LOCAL_OPERATIONS clause
    (abstract specifications on the visible variables)
  - their invariant is made up by the typing of the concrete variables
  - they are implemented in the OPERATIONS clause
    (like global operations)
form of B components: dynamic aspect

local operation example

IMPLEMENTATION
...
LOCAL_OPERATIONS
\[ r \in GetMax(x, y) = \]
\[
\text{PRE } x : \text{INT} \ & \ y : \text{INT THEN}
\]
\[
r := \text{max} \left( \{x, y\} \right)
\]
END

OPERATIONS
\[ r \in GetMax(x, y) = \]
\[
\text{IF } x < y \ \text{THEN } r := y
\]
\[
\text{ELSE } r := x
\]
END;

\[ \ldots \]
\[
op =
\]
\[
\ldots \in GetMax(z, 10);
\]
\[
\ldots
\]
END
the B language

- **modular construction**
  - the B language supports modularity: breaking down large sub-systems, building up small sub-systems into larger ones

- **modular mechanisms**
  - importing abstract machines at implementation level (IMPORTS clause)
  - read-only visibility of other abstract machines (SEES clause)
the B language: decomposition

- importing of other machines
  
  e.g. IMPLEMENTATION
  
  $A_i$
  
  REFINES
  
  $A$
  
  IMPORTS
  
  $B, C$

  - the IMPORTS clause may only appear in implementations
  - an implementation *imports* other abstract machines in order to implement data and operations with lower level machines: this is the main breaking down mechanism in B
  - variables of an imported machine may be modified in the implementation by operation calls
the B language: decomposition

- visibility of other machines
  
  e.g. MACHINE
       A
       SEES
       B, C

  ➔ when a component sees an abstract machine $M$, the data of $M$ may be accessed in read-only, modification operations of $M$ can not be called
  ➔ the SEES link is not transitive
B-Software links

Abstract model

Concrete model

- B module
- B specification
- Implementation or refinement
- Decomposition link (IMPORTS)
- Read-only link (SEES)
the B language

B0

- **B0 is the part of B that may be translated**
  - the name of the abstract machine and the links in the implementation
  - the concrete data of the module: sets, concrete constants, concrete variables, the valuation of sets and concrete constants
  - implementation initialisation
  - operations: operation parameters and the substitutions of implementation operations

- **the data must be concrete**
  - scalar data: INT, BOOL, sets (fixed and enumerated sets)
  - sub-intervals of INT
  - implementable arrays
B Training Sessions

- **Level I: understanding B**
  - overview of the B method and the B language
  - tutorials and practical with Atelier B: specification, writing a program that is consistent with its specification, notion of proof

- **Level II: applying B**
  - advanced notions of the B method, B for systems
  - tutorials and practical with Atelier B: building a large program, Proof Obligations

- **Level III: proving**
  - learning to use Atelier B to prove a project
  - practical: automatic and interactive proof, proof tools, user rules