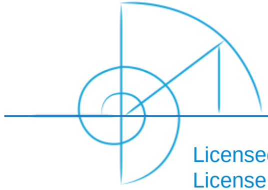
Safety Solutions Designer

AIX
LYON
PARIS
STRASBOURG

WWW.CLEARSY.COM

TRAINING B – Level 1
Understand B

CLEARSY



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 feature 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

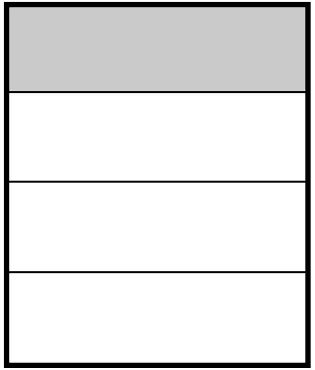
B structuring

- 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 implementatio

B structuring



module specification

module 1st refinement

module 2nd refinement

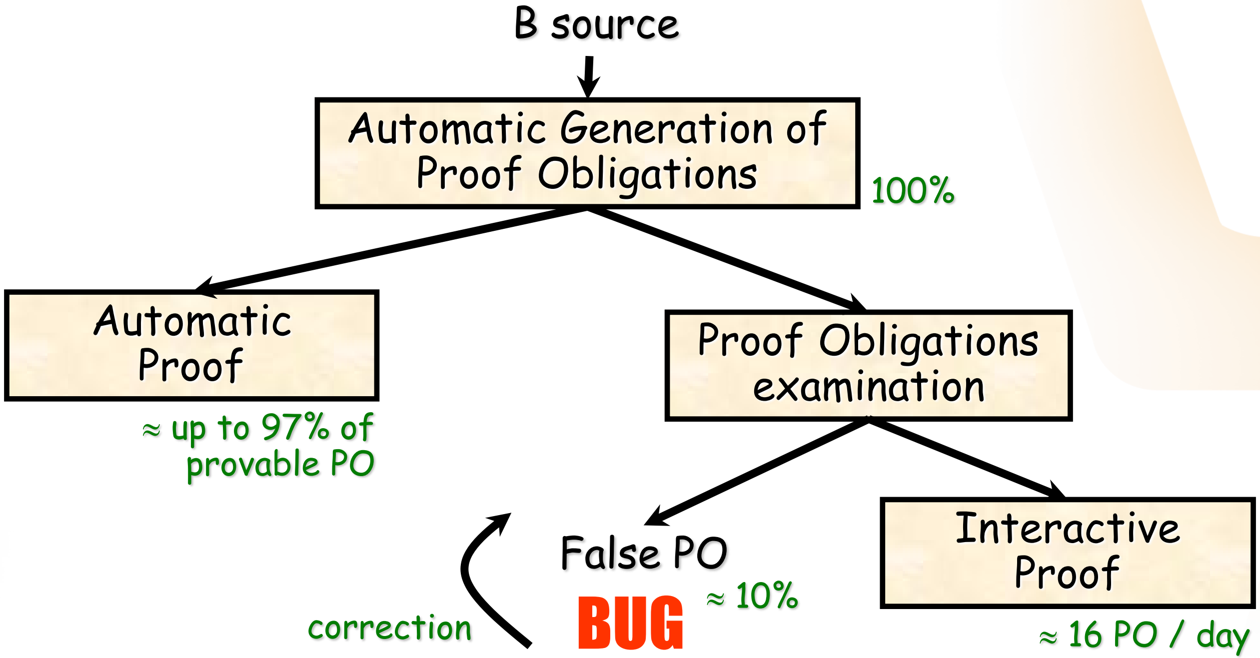
module 3rd refinement

▭ **B module**

▬ **B specification**

▭ **Implementation or refinement**

validation by the proof



B source

Automatic Generation of Proof Obligations

100%

Automatic Proof

≈ up to 97% of provable PO

Proof Obligations examination

False PO

BUG

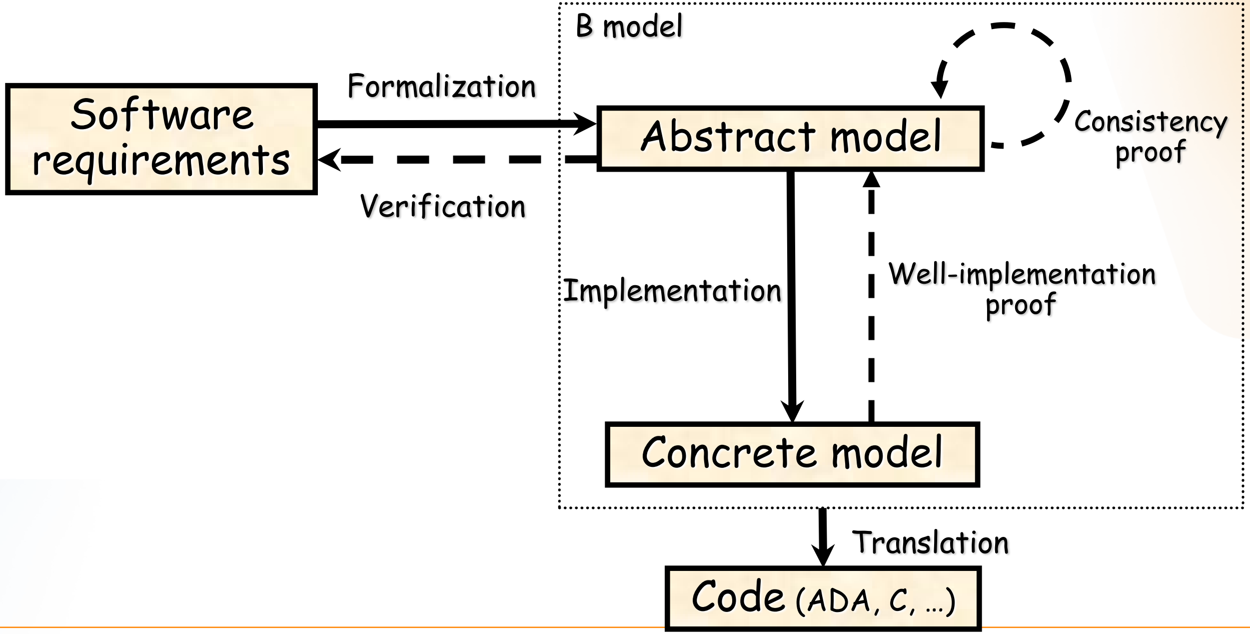
≈ 10%

correction

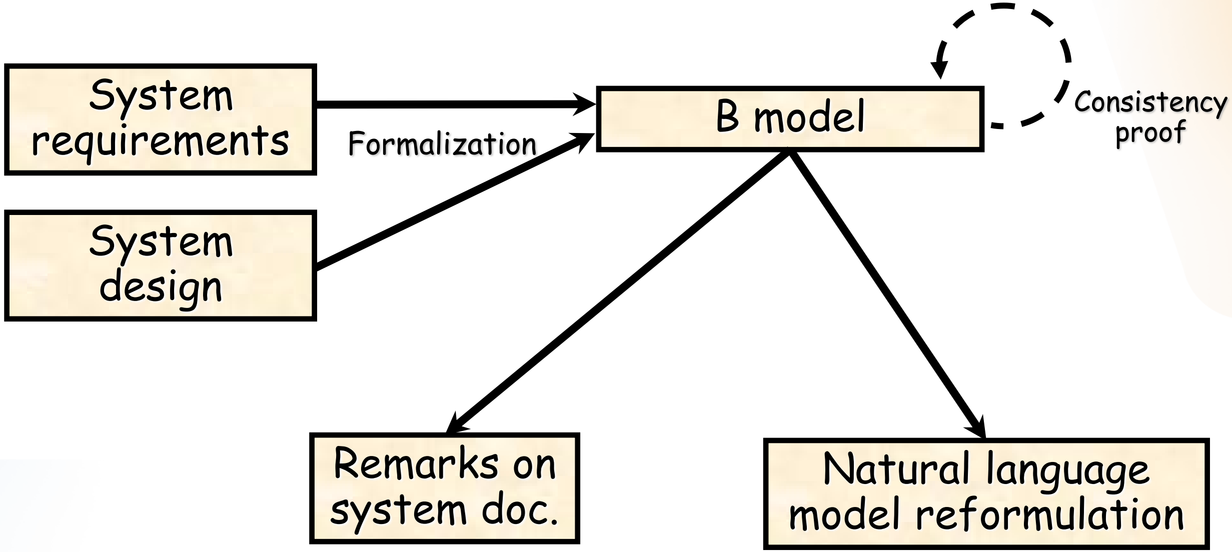
Interactive Proof

≈ 16 PO / day

B-Software


B model
Software requirements
Formalization
Verification
Abstract model
Consistency proof
Implementation
Well-implementation proof
Concrete model
Translation
Code (ADA, C, …)

B-System



System requirements

System design

Formalization

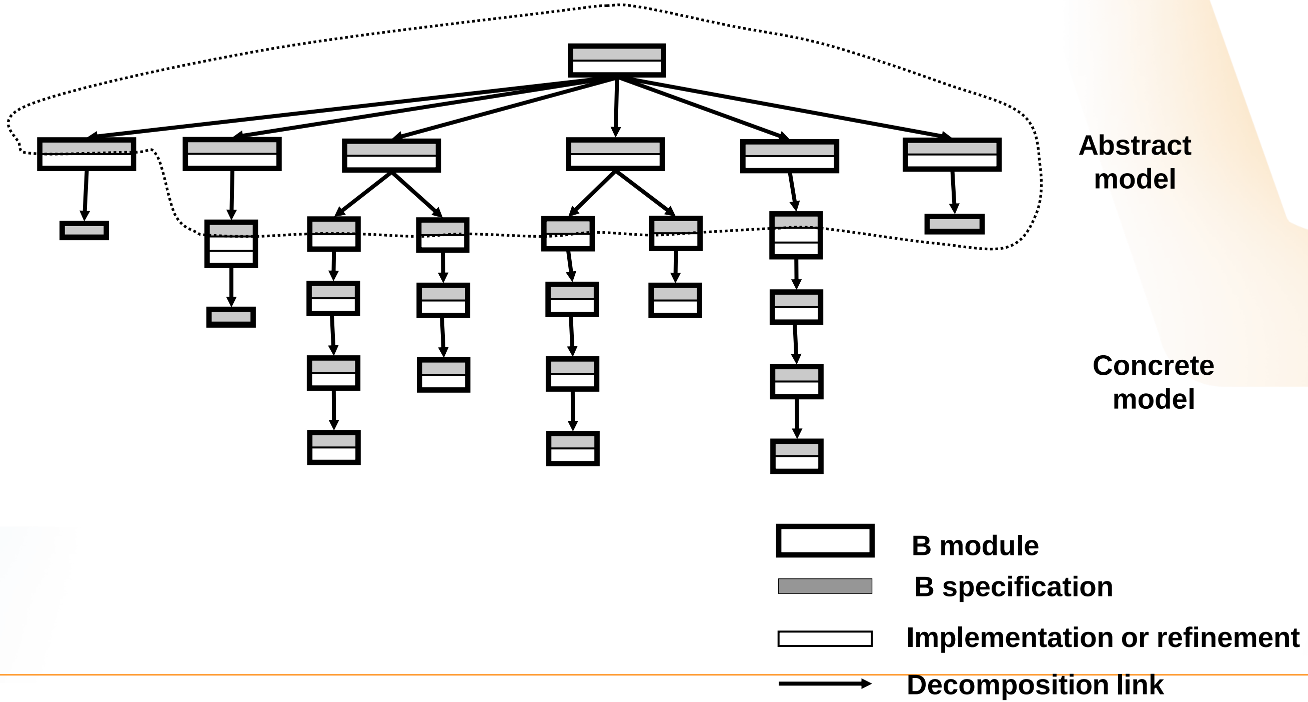
B model

Consistency proof

Remarks on system doc.

Natural language model reformulation

B-Software structure



Abstract model

Concrete model

B module

B specification

Implementation or refinement

Decomposition link

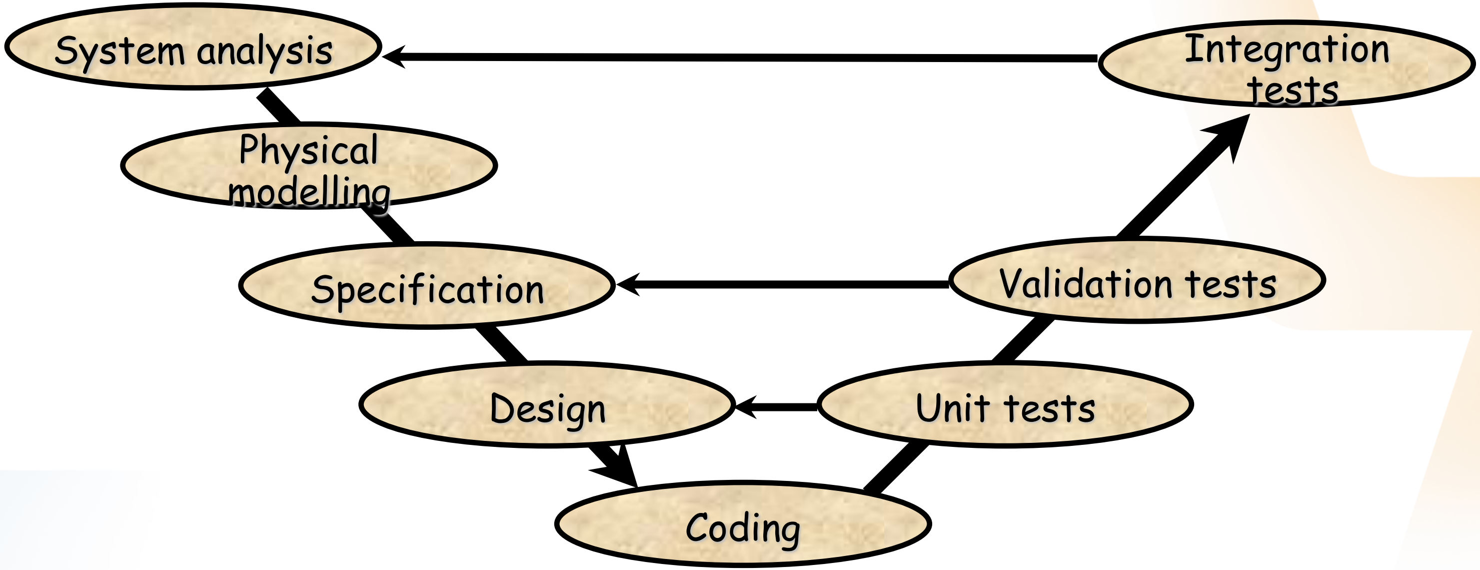
benefits of B-Software

- 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)

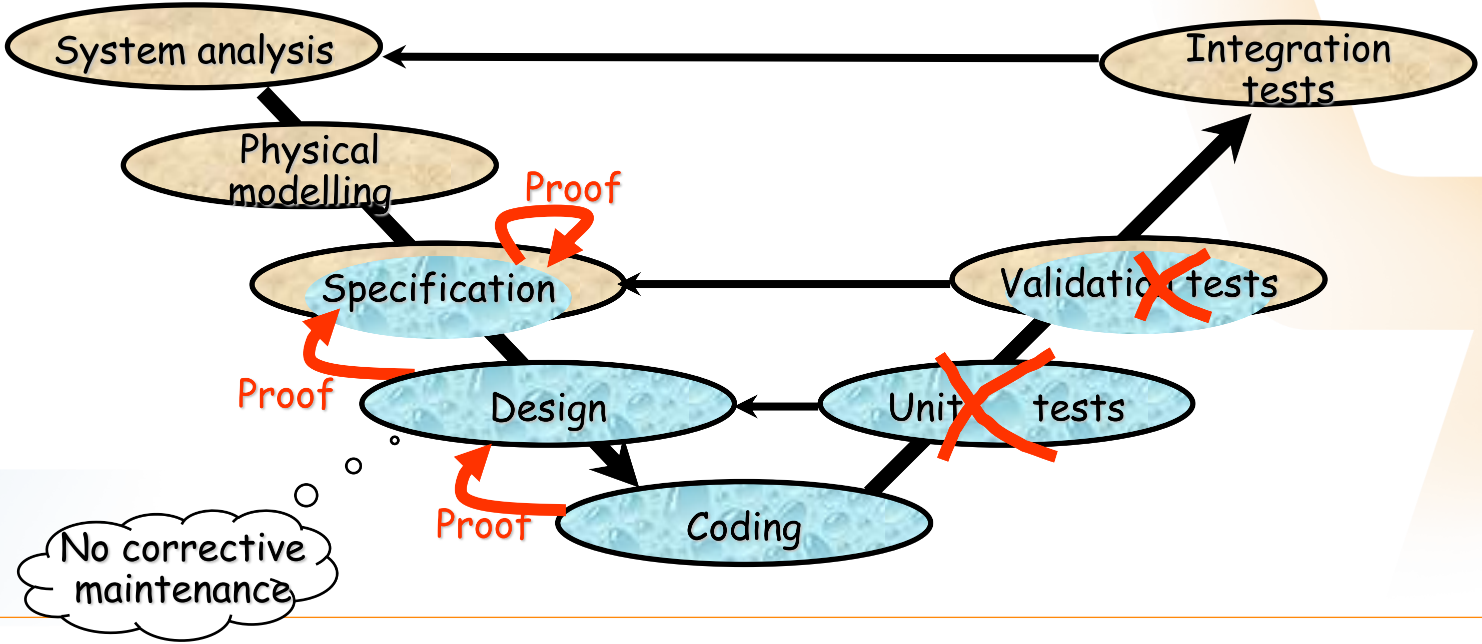
benefits of B-Software

- 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



the B development cycle



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 component**

  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

general form

```
MACHINE
            machine name
SETS
            set names
CONSTANTS
            constant names
PROPERTIES
            predicate
VARIABLES
            variable names
INVARIANT
            predicate
INITIALISATION
            substitution
OPERATIONS
            operation definitions
END
```

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

general form

```
REFINEMENT
            machine name_n
REFINES
            machine name
...

VARIABLES
            variable names
INVARIANT
            predicate
INITIALISATION
            initialisation refinement
OPERATIONS
            operation refinements
END
```

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
```

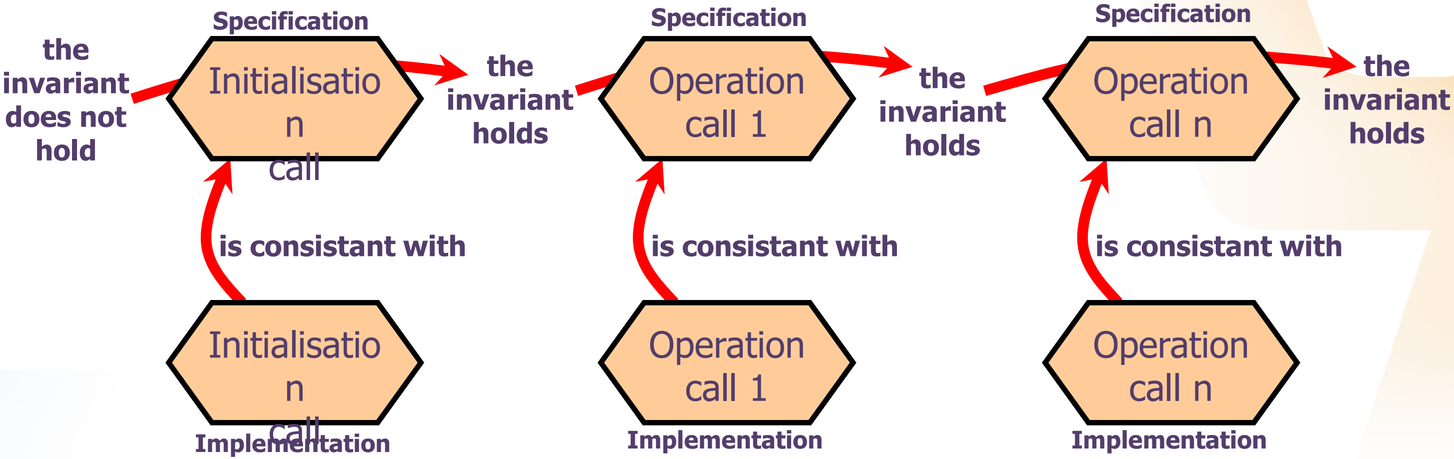
values for fixed sets
and constants

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?



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. $x + y = 0 \ \wedge \ x < y$

the B language: predicate logic

on paper	at the keyboard
$\neg$	`not`
$\wedge$	`&`
$\vee$	`or`
$\Rightarrow$	`=>`
$\Leftrightarrow$	`<=>`
$\forall$	`!`
$\exists$	`#`
$\cdot$	`.`

propositions

$\neg P$	negation of P (logical NOT)
$P \wedge Q$	conjunction of P and Q (logical AND)
$P \vee Q$	disjunction of P and Q (logical OR)
$P \Rightarrow Q$	logical implication: $\neg P \vee Q$
$P \Leftrightarrow Q$	logical equivalence: $P \Rightarrow Q \wedge Q \Rightarrow P$

quantified predicates

$\forall x \cdot (P_x \Rightarrow Q_x)$ universal quantification

$\exists x \cdot (P_x)$ existential quantification: $\neg (\forall x \cdot (\neg P_x))$

the B language: predicate logic

on **paper**	at the **keyboard**
=	=
≠	/=
<	<
≤	<=
>	>
≥	>=

equality predicates

➔ let x and y be two expressions

$x = y$	x equal to y
$x \neq y$	non equality: $\neg (x = y)$

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

the B language: predicate logic

set predicates

➔ let x be an element, and let X and Y be two sets

$x \in X$	membership: x is an element of X
$x \notin X$	non membership
$X \subseteq Y$	inclusion: X is a subset of Y
$X \not\subseteq Y$	non inclusion
$X \subset Y$	strict-inclusion: $X \subset Y \Leftrightarrow X \subseteq Y \wedge X \neq Y$
$X \not\subset Y$	non strict-inclusion

on paper	at the keyboard
$\in$	`:`
$\notin$	`/:`
$\subseteq$	`<:`
$\not\subseteq$	`/<:`
$\subset$	`<<:`
$\not\subset$	`/<<:`

the B language

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

the B language: set theory

on paper	at the keyboard
∅	{ }

explicit sets

- ➔ empty set: ∅
- ➔ 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):

$$z \in (x\ ..\ y) \quad \Leftrightarrow \quad z \geq x \ \wedge \ z \leq y$$

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. $\{ x \mid x \in 1..5 \ \wedge \ x \bmod 2 = 0 \} = \ldots$

the B language: set theory

on paper	at the keyboard
$\cup$	\/
$\cap$	/\

set expressions

➔ let X and Y be two sets

$X \cup Y$ union of X and Y :

$z \in (X \cup Y) \Leftrightarrow z \in X \vee z \in Y$

e.g. $\{1, 3, 5\} \cup 1..3 = ...$

$X \cap Y$ intersection of X and Y :

$z \in (X \cap Y) \Leftrightarrow z \in X \wedge z \in Y$

e.g. $\{1, 3, 5\} \cap 1..3 = ...$

$X - Y$ set deferrence of X and Y :

$z \in (X - Y) \Leftrightarrow z \in X \wedge z \notin Y$

e.g. $\{1, 3, 5\} - 1..3 = ...$

$1..3 - \{1, 3, 5\} = ...$

the B language: set theory

subset types

on paper	at the keyboard
$\mathbb{P}$	POW
$\mathbb{P}_1$	POW1
$\mathbb{F}$	FIN
$\mathbb{F}_1$	FIN1

➔ let X be a set

$\mathbb{P}(X)$ the set (type) of subsets of X :

$x \in \mathbb{P}(X) \Leftrightarrow x \subseteq X$

$\mathbb{P}_1(X)$ the set (type) of non-empty subsets of X :

$\mathbb{P}_1(X) = \mathbb{P}(X) - \{\varnothing\}$

e.g. $\mathbb{P}(\{1,2,3\}) = \ldots$

$\mathbb{P}_1(\{1,2,3\}) = \ldots$

$\mathbb{F}(X)$ the set (type) of finite subsets of X

$\mathbb{F}_1(X)$ the set (type) of finite non-empty subsets of X :

$\mathbb{F}_1(X) = \mathbb{F}(X) - \{\varnothing\}$

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 bool (*P*) is TRUE if *P* holds, otherwise FALSE

  e.g. bool(*P*) = bool(*Q*) ⇔ ...

the B language: set theory

numeric expressions

on paper	at the keyboard
$\times$	*
x^n	x ** n

- ➔ 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

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)

on <u>paper</u>	**at the <u>keyboard</u>**
$\mathbb{Z}$	`INTEGER`
$\mathbb{N}$	`NATURAL`
$\mathbb{N}_1$	`NATURAL1`

numeric constants

MAXINT	the largest implementable relative integer
MININT	the smallest implementable relative integer
INT	predefined set of implementable relative integers INT = MININT .. MAXINT
NAT	predefined set of implementable natural numbers NAT = 0 .. MAXINT
NAT_1	predefined set of strictly positive implementable natural numbers NAT_1 = 1 .. MAXINT

the B language: set theory

on paper	at the keyboard
↦	|->
×	*

maplet expression

➔ let x and y be two elements

x, y ordered couple or

$x \mapsto y$ 'maplet' (x associated to y)

$$x \mapsto y = x, y$$

cartesian product types

➔ let X and Y be two sets

$X \times Y$ cartesian product of X and Y

the set (type) of maplets x, y such that $x \in X$ et $y \in Y$

e.g. $\{0, 1\} \times \{a, b, c\} = ...$

the B language: set theory

on paper	at the keyboard
↔	`<->`

relations

- ➔ definition: a **relation** from a source set X into a target set Y is a subset of the cartesian product X × 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: $R \in X \leftrightarrow Y$

relation types

- ➔ the set of relations from X into Y :

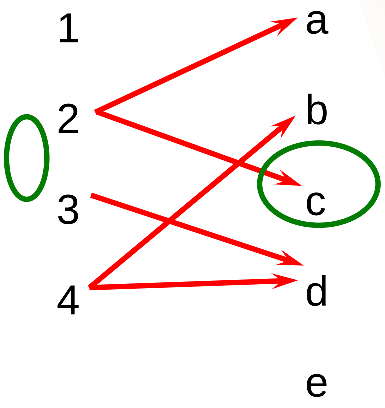
$$X \leftrightarrow Y \;=\; \mathbb{P}\,(X \times Y)$$

the B language: set theory

Explicit relations

e.g. : $\varnothing$ (empty relation)

$\{2 \mapsto a,$
$2 \mapsto c,$
$3 \mapsto d,$
$4 \mapsto b,$
$4 \mapsto d\}$



the B language: set theory

- relational operations

➔let R be a relation from X into Y

$\mathrm{dom}\,(R)$ domain of the relation R (a subset of X)

$x \in \mathrm{dom}\,(R) \Leftrightarrow \exists y \cdot (y \in Y \wedge x \mapsto y \in R)$

e.g. $\mathrm{dom}\,(\{2 \mapsto a, 2 \mapsto c, 3 \mapsto d, 4 \mapsto b, 4 \mapsto d\}) = \ldots$

$\mathrm{ran}\,(R)$ codomain or *range* of the relation R (a subset of Y)

$y \in \mathrm{ran}\,(R) \Leftrightarrow \exists x \cdot (x \in X \wedge x \mapsto y \in R)$

e.g. $\mathrm{ran}\,(\{2 \mapsto a, 2 \mapsto c, 3 \mapsto d, 4 \mapsto b, 4 \mapsto d\}) = \ldots$

➔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 (a subset of Y)

$y \in R[S] \Leftrightarrow \exists s \cdot (s \in S \wedge s \mapsto y \in R)$

e.g. $\{2 \mapsto a, 2 \mapsto c, 3 \mapsto d, 4 \mapsto b, 4 \mapsto d\}[\{1,2,3\}] = \ldots$

the B language: set theory

on paper	at the keyboard
R^{-1}	R ~

relational expressions

➔ let X be a set

id (X) identity on X (the set of maplets $x \mapsto x$, for $x \in X$):

$$x \mapsto x \in id(X) \quad \Leftrightarrow \quad x \in X$$

e.g. id ({a, b, c}) = ...

➔ let R be a relation from X into Y

R^{-1} converse relation (inverse maplets, from Y into X):

$$y \mapsto x \in R^{-1} \quad \Leftrightarrow \quad x \mapsto y \in R$$

e.g. $\{2 \mapsto a, 2 \mapsto c, 3 \mapsto d, 4 \mapsto b, 4 \mapsto d\}^{-1} = ...$

the B language: set theory

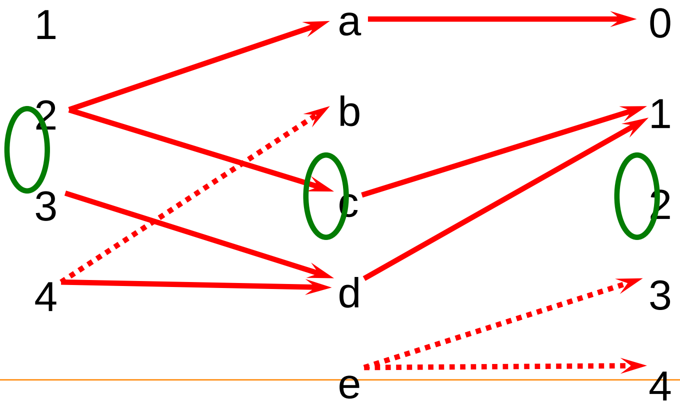
composition of relational expressions

composition of the relations R_1 and R_2: $R_1 \mathbin{;} R_2$

$$x \mapsto z \in (R_1 \,;\, R_2) \Leftrightarrow \exists y.\ (y \in Y \wedge x \mapsto y \in R_1 \wedge y \mapsto z \in R_2)$$

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

$= \ldots$



the B language: set theory

filtering relational expressions

<table>
<tr><th>on paper</th><th>at the keyboard</th></tr>
<tr><td>$\lhd$</td><td><|</td></tr>
<tr><td>$\rhd$</td><td>|></td></tr>
</table>

➔ $S \lhd R$ restriction to the set S over the domain of relation R (keeping only maplets with first elements belonging to S)

$$S \lhd R \quad = \quad id (S) ; R$$

e.g. $\{1, 2, 3\} \lhd \{2 \mapsto a, 2 \mapsto c, 3 \mapsto d, 4 \mapsto b, 4 \mapsto d\} = ...$

➔ $R \rhd S$ restriction to the set S over the codomain of relation R (keeping only maplets with second elements belonging to S)

$$R \rhd S \quad = \quad R ; \; id (S)$$

e.g. $\{2 \mapsto a, 2 \mapsto c, 3 \mapsto d, 4 \mapsto b, 4 \mapsto d\} \rhd \{b, c, d\} = ...$

the B language: set theory

on paper	at the keyboard
⩤	<<|
⩥	|>>

filtering relational expressions (cont.)

➔ $S ⩤ R$ exclusion of the set S from the domain of relation R (removing maplets with first elements belonging to S)

$S ⩤ R \;=\; id\,(\,dom\,(R) - S\,)\,;\,R$

e.g. $\{1, 2, 3\} ⩤ \{2 \mapsto a, 2 \mapsto c, 3 \mapsto d, 4 \mapsto b, 4 \mapsto d\}$

= ...

➔ $R ⩥ S$ exclusion of the set S from the codomain of relation R (removing maplets with second elements belonging to S)

$R ⩥ S \;=\; R\,;\,id\,(\,ran\,(R) - S\,)$

e.g. $\{2 \mapsto a, 2 \mapsto c, 3 \mapsto d, 4 \mapsto b, 4 \mapsto d\} ⩥ \{b, c, d\}$

= ...

the B language: set theory

on paper	at the keyboard
$\mathbin{<\!\!\!+}$	<+

relational expressions (cont.)

➔let R_1 and R_2 be two relations from X into Y

$R_1 \mathbin{<\!\!\!+} R_2$ 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 dom (R_2), together with all maplets from R_2

$$R_1 \mathbin{<\!\!\!+} R_2 \;=\; (\mathrm{dom}\,(R_2) ⩤ R_1) \cup R_2$$

e.g. $\{0 \mapsto 1, 1 \mapsto 1\} \mathbin{<\!\!\!+} \{0 \mapsto 0\} = \ldots$

$\{2 \mapsto a, 2 \mapsto c, 3 \mapsto d, 4 \mapsto b, 4 \mapsto d\} \mathbin{<\!\!\!+} \{1 \mapsto a, 2 \mapsto b, 2 \mapsto c, 3 \mapsto e\}$

$= \ldots$

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

on paper	**at the keyboard**
λ	%

➔ let F be a function, and x be an element of dom (F)

F (x)   the (unique) value of function F for x

the associated element from ran (F), where $x \mapsto F(x) \in F$

e.g. $f = \{ 0 \mapsto a, 1 \mapsto b, 2 \mapsto a \}$

$f(1) = \dots$

functions defined by expressions

➔ let x be a name, X be a set and E be an expression (in x)

$\lambda x \cdot (x \in X \mid E(x))$ explicit definition in the form of a λ-expression

the function consisting of maplets $x \mapsto E(x)$, for $x \in X$

e.g. $plus2 = \lambda z \cdot (z \in \mathbb{Z} \mid z + 2)$

$plus2(1) = \dots$

the B language: set theory

on paper	at the keyboard
⇸	+->
→	-->

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: $F \in X ⇸ Y$
- ➔ $F \in X ⇸ Y \quad \Leftrightarrow \quad F \in X \leftrightarrow Y \ \wedge\ (F^{-1} ; F) \subseteq id (Y)$
- ➔ definition: a **total function** is a function from X into Y where the domain is equal to X
- ➔ such a function is denoted: $F \in X \rightarrow Y$
- ➔ $F \in X \rightarrow Y \quad \Leftrightarrow \quad F \in X ⇸ Y \ \wedge\ dom (F) = X$

the B language: set theory

on paper	at the keyboard
⤔	>+>
↣	>->

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: $F \in X ⤔ Y$

$$F \in X ⤔ Y \quad \Leftrightarrow \quad F \in X ⇸ Y \wedge F^{-1} \in Y ⇸ X$$

- ➔ definition: a **total injection** is an injection from X into Y where the domain is equal to X
- ➔ such a function is denoted: $F \in X \rightarrowtail Y$

$$X \rightarrowtail Y \quad = \quad X ⤔ Y \cap X \rightarrow Y$$

the B language: set theory

on paper	at the keyboard
⤀	`+->>`
↠	`-->>`

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: $F \in X ⤀ Y$

$$F \in X ⤀ Y \quad \Leftrightarrow \quad F \in X ⇸ Y \;\wedge\; \mathrm{ran}\,(F) = Y$$

- ➔ definition: a **total surjection** is a surjection from X into Y where the range is equal to X
- ➔ such a function is denoted: $F \in X \twoheadrightarrow Y$

$$X \twoheadrightarrow Y \quad = \quad X ⤀ Y \;\cap\; X \rightarrow Y$$

the B language: set theory

on paper	at the keyboard
⤗	>+>>
⤖	>->>

bijective functions

- ➔ definition: a **partial bijection** is a function from X into Y that is injective and surjective;
- ➔ such a function is denoted: $F \in X ⤗ Y$

$$X ⤗ Y \quad = \quad X ⤔ Y \;\cap\; X ⤀ Y$$

- ➔ definition: a **total bijection** is a total function from X into Y that is injective and surjective;
- ➔ such a function is denoted: $F \in X ⤖ Y$

$$X ⤖ Y \quad = \quad X \rightarrowtail Y \;\cap\; X \twoheadrightarrow Y$$

- ➔ the inverse of a partial bijection is ...

the B language: set theory

- sequences

  ➔definitions: a sequence of ‘elements’ belonging to a set *X* is a total function from an interval 1..n into *X* , for n $\in \mathbb{N}$
  the sequence then correspond to the second elements of the maplets of this function, *ordered* by their first elements

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

  ➔consequence: function expressions also apply to sequences

- explicit sequences

  ➔[ ] the empty sequence

  ➔$[x_1,\ldots x_n]$ sequence of *X* defined by enumeration

  e.g. $[a, b, c] = \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$

the B language: set theory

on paper	at the keyboard
←	<-
→	->

- 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* → *S* insertion of *x* before the sequence *S*
e.g. a → [b, c] = [a, b, c]

➔*S* ← *x* insertion of *x* after the sequence *S*
e.g. *[a, b] ← c = [a, b, c]*

the B language: set theory

<table>
<tr><th>on
<u>paper</u></th><th>at the
<u>keyboard</u></th></tr>
<tr><td>⌢</td><td>^</td></tr>
<tr><td>↑</td><td>/|\</td></tr>
<tr><td>↓</td><td>\|/</td></tr>
</table>

- sequence expressions

➔ $S_1 \frown S_2$ concatenation of sequences S_1 and S_2

e.g. [a, b, c] ⌢ [c, b, a] = [a, b, c, c, b, a]

➔ $S \uparrow n$ sequence comprising the first n elements of S at most, or S itself when $n >$ size (S)

e.g. [a, b, c] ↑ 2 = [a, b]

➔ $S \downarrow n$ sequence obtained by removing the first n elements of S

e.g. [a, b, c] ↓ 2 = [c]

➔ tail (S) sequence obtained by removing the first element of S

e.g. tail ([a, b, c]) = [b, c]

➔ front (S) sequence obtained by removing the last element of S

e.g. 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)

e.g. perm({a,b,c}) = {[a,b,c], [a,c,b], [b,a,c], [b,c,a], [c,a,b], [c,b,a]}

the B language
substitutions

- 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 proper
  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]

  $v := E$ the value of E is assigned to v

  e.g. $v := 0$

  $x := y + 1$

  $a, b := c, d$

  $f(i) := m$

  $r'b := n$

- ➔ “becomes element of” substitution [Spec]

  $v :\in X$ an element of X is assigned to v

  e.g. $v :\in (1..3)$

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. $x :(x \in \mathrm{NAT} \wedge x \bmod 3 = 1)$

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

  $q, r : (q \in \mathbb{N} \wedge r \in \mathbb{N} \wedge n = (m \times q) + r \wedge r < m)$

  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]

  ➔$S_1 \parallel S_2$
  applies the substitutions S_1 and S_2 simultaneously
  the variables modified in S_1 and S_2 must be *distinct*
  e.g. $x := 1 \parallel y := 2$
  $x := y \parallel y := x$

- sequential substitutions [B0 code]

  ➔$S_1 ; S_2$
  applies the substitution S_1 *and then* the substitution S_2
  e.g. $x := 1 ; y := 2$
  $x := y ; y := x + 1$

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 $v_1, \dots , v_n$ IN *S* END

  introduction of local variables $v_1,\dots v_n$ 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]

  ➔PRE *P* THEN *S* 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. $PRE\ x \in NAT_1\ THEN\ x := x - 1\ END$

- assertion [Spec, B0 code]

  ➔ASSERT *P* THEN *S* END

  similar to a precondition, but used to simplify the proof, by factorizing a property

the B language substitutions

- "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 $x \in$ NAT $\wedge$ $x \leq 10$ 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
$y :\in E$	ANY x WHERE $x \in$ 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 $x \geq 10$ THEN $x := x - 10$
  WHEN $x \leq 10 \wedge x \geq 0$ THEN $x := 2 \times (x - 10)$
  ELSE $x := x - 1$
  END

the B language substitutions

- "IF" substitution [Spec, B0 Code]

  IF P_1 THEN S_1 ELSIF P_2 THEN S_2 … ELSE S END

  the substitution applied is:

  S_i if P_i holds **and** the previous predicates do not hold

  S if no predicate P_i holds (by default S is skip)

  e.g.

```
IF  x > 10  THEN
        x := x - 10
ELSIF  x = 0  THEN
        x := x + 1
ELSE
        x := 1
END
```

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]
  - WHILE *P* DO *S* INVARIANT *I* VARIANT *V* 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

on **paper**	at the **keyboard**
⟵	`<--`

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 ⟵ op3
x, y ⟵ 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 = ~~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 constructor*

  ➔the basic types are
  - *BOOL*
  - $\mathbb{Z}$
  - *fixed or enumerated sets (see the* ***SETS*** *clause)*

  $$e.g.\ \mathrm{TRUE} \in \mathrm{BOOL}$$
  $$2 \in \mathbb{Z}$$

  ➔the type constructors are

subsets	$\mathbb{P}(T)$
Cartesian products	$T_1 \times T_2$

  $$e.g.\ \{1, 3, 5\} \in \mathbb{P}(\mathbb{Z})$$
  $$(0 \mapsto \mathrm{FALSE}) \in \mathbb{Z} \times \mathrm{BOOL}$$

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

    untyped_data_item typing_operator typed_expression

    where the *typing_operators* are '=', '$\in$' et '$\subseteq$'

    e.g. $x \in 1..10 \land y \in BOOL \to INT \land z = x + 1 \land S \subseteq INT$

    such typing predicates must be at the highest syntactic level within a conjunction list
  - local variables and result parameters of an operation are inste typed by *typing substitutions*

    e.g. $VAR\ t\ IN\ \dots t := x + 1\ ;\ \dots\ END$

    $r \longleftarrow op(\ p\ ) \triangleq\ PRE\ p \in NAT_1\ THEN \dots r := p - 1 \dots\ END$

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; … ; 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: $X_i = \{x_1, \dots\ x_m\}$

    e.g. SETS COLOR = { Red, Green, Blue}

form of B components: static aspect

- constant definitions

➔ABSTRACT_CONSTANTS $x_1,\dots,x_n$
(CONCRETE_)CONSTANTS $x_1,\dots,x_n$

➔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

form of B components: static aspect

- constant definitions

PROPERTIES $P_{x1,\dots,xn}$

the PROPERTIES clause defines the types and other properties of the constants

e.g. CONSTANTS *c1*, *c2*
ABSTRACT_CONSTANTS *c3*

PROPERTIES
$c1 \in 0..10 \wedge$
$c2 \in 0..10 \wedge$
$c1 + c2 < 15 \wedge$
$\mathrm{c3} \in \mathbb{N} \rightarrow 0..15$

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

➔INVARIANT $P_{v1,\ldots,vn}$

e.g. VARIABLES

A, B

INVARIANT

$A \subseteq T \wedge$

$B \subseteq T \wedge$

$\mathrm{card}(A \cap B) = 1$

➔the INVARIANT 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

```
MACHINE
        Register                                    } machine name
SETS
        STUDENTS                                    } fixed set
CONCRETE_CONSTANTS
        max_students
PROPERTIES
        max_students ∈ NAT                          } constant type
ABSTRACT_VARIABLES
        Enrolled
INVARIANT
        Enrolled ⊆ STUDENTS  ∧                      } variable type,
        card (Enrolled) ≤  max_students             } and properties
...
END
```

form of B components:
static aspect

- values of fixed sets and concrete constants

➔*e.g.* VALUES

STUDENTS = 0..255;
max_students = 255;
transfer = {0 ↦ FALSE, 1 ↦ TRUE, 2 ↦ FALSE};
default_grade = (0..255) × {0}

➔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 definiti 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
        INVARIANT
                Enrolled ⊆ STUDENTS
        INITIALISATION
                Enrolled := ∅
```

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

$$op \;=\; S$$
$$op\;(\,p_1,\ldots,p_n\,)\;=\;S$$
$$r_1,\ldots,r_m \longleftarrow op\;=\;S$$
$$r_1,\ldots,r_m \longleftarrow op\;(\,p_1,\ldots,p_n\,)\;=\;S$$

  - ➔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 :=  ∅
...
```

```
...
OPERATIONS
        n ⟵ num_ enrolled  =
                n :=  card (Enrolled) ;
        b ⟵ is_ enrolled ( s ) =
                PRE  s ∈ STUDENTS
                THEN  b :=  bool ( s ∈ Enrolled )
                END ;
        enrol_student ( s ) =
                PRE  s ∈ STUDENTS – Enrolled
                THEN Enrolled :=  Enrolled ∪ {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 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 ⟵ GetMax(x, y) =
                PRE x ∈ INT ∧ y ∈ INT THEN
                r := max ({x, y})
                END
OPERATIONS
        r ⟵ GetMax(x, y) =
                IF x ≤ y THEN r := y
                ELSE r := x
                END ;
        …
        op =
                …
                v ⟵ GetMax(z, 10);
                …
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*

```
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

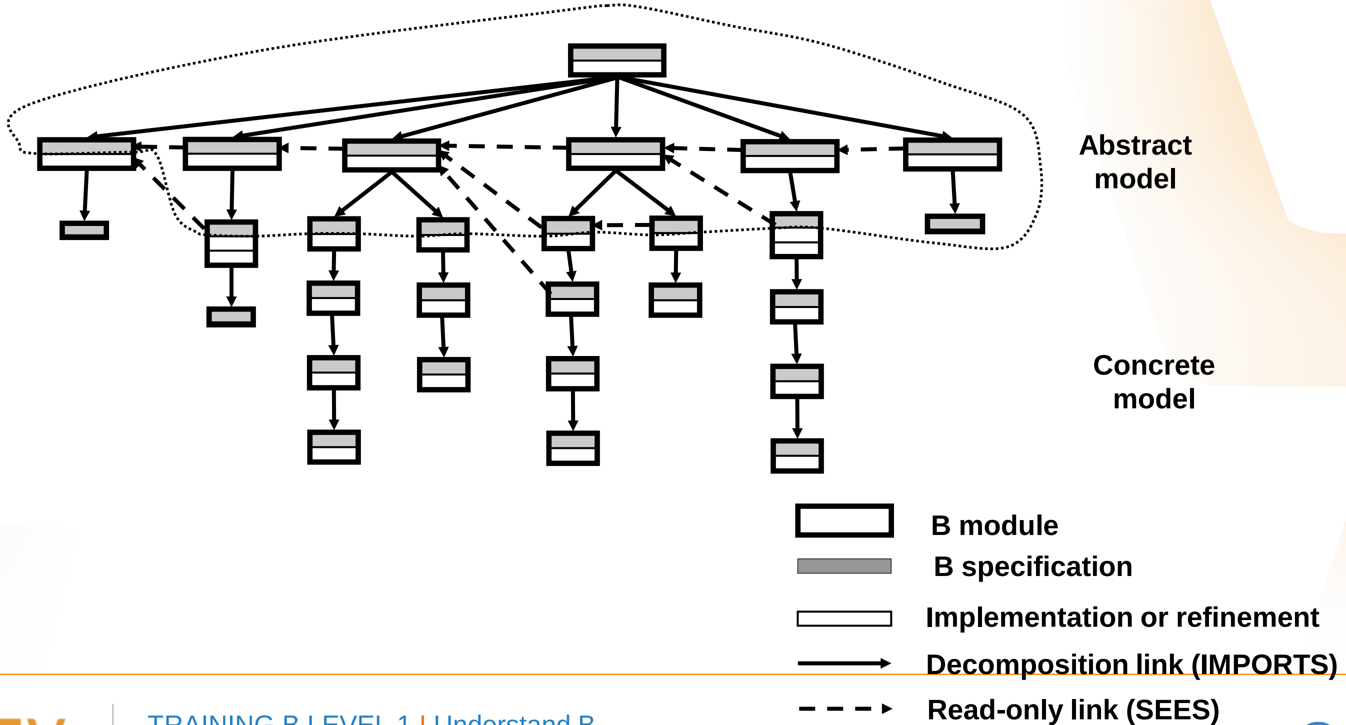
e.g. MACHINE
A

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



▭ **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