Safety Solutions Designer

AIX
LYON
PARIS
STRASBOURG

WWW.CLEARSY.COM

TRAINING B – Level 2
Practice B
CLEARSY

Licensed under the Creative Commons Attribution 4.0 Unported License. You may not use this file except in compliance with the License. You may obtain a copy of the License at http://creativecommons.org/licenses/by/4.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

order of presentation

Copyright © 2022 CLEARSY

Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

what is B

a method for

the construction of ‘correct’ pieces of software

the construction (or the formalization) of ‘correct’ systems

supported by

automated proof tools

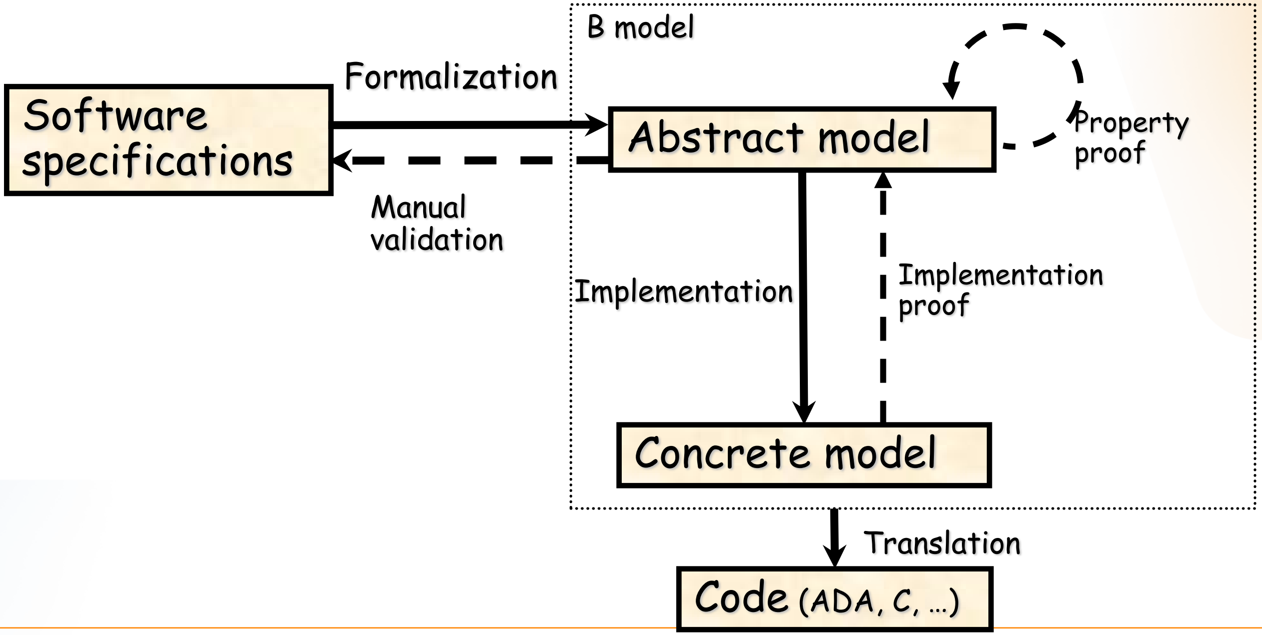
Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

elements of the B method

– mathematical basis	set-theory, predicate logic and substitutions
– structuring concept	modules: abstract machines, refinements
– development method	refinement, IMPORTS decomposition
– verification process	formal proof of correctness
– supporting tool	Atelier B: industry-oriented
– reference	*The B-Book*, by J-R Abrial

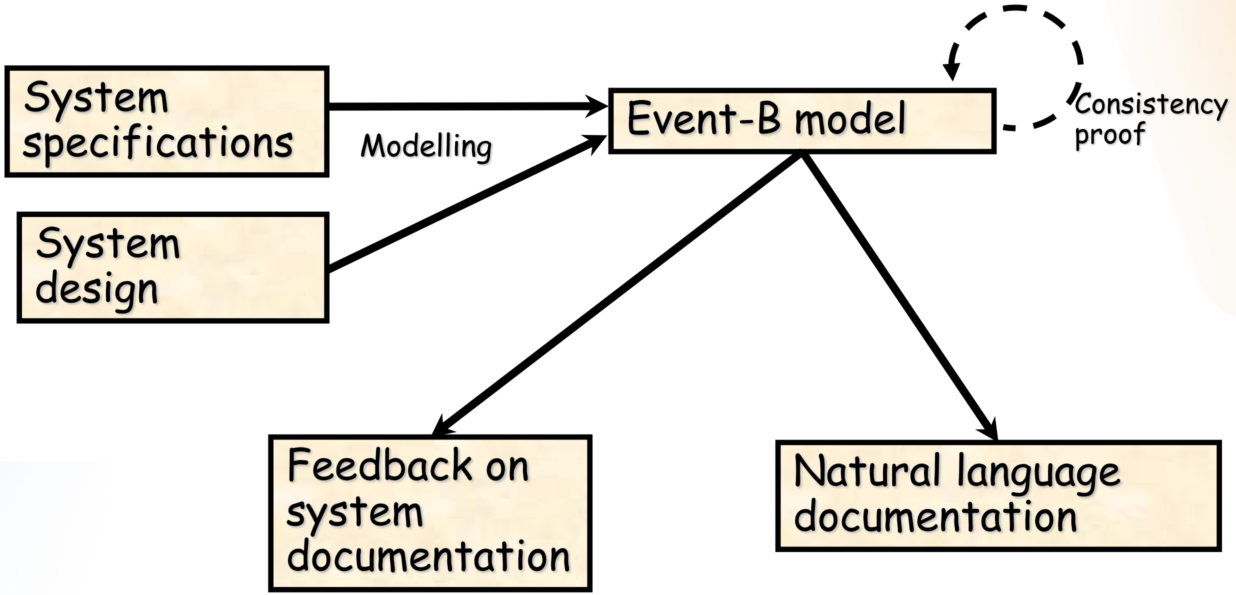
Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

software development with B



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

B-System



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

formalization process

- fully understanding the requirements stated and unstated!
  - document analysis and interviews help to:
  - clarify the requirements
  - understand what the system should really do
  - explicit unstated requirements
  - use any other formalism to understand what the system should do
  - natural language, functional analyses, statecharts, …
- building step by step a good quality B model
  - difficulty: it requires some experience
  - some criteria of model quality are given below
- iterate
  - the B Method: a process to adjust precisely a model during its construction

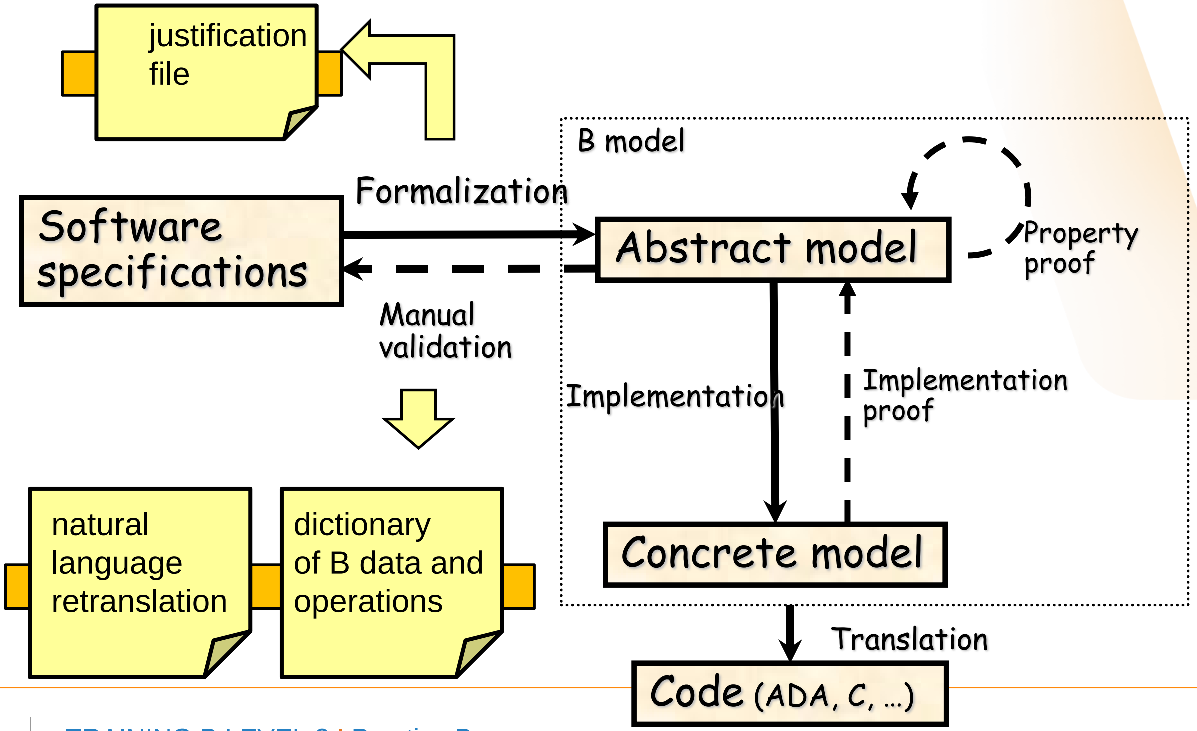
Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

criteria of model quality

- properties
  number and interest of proved properties,
  levels of detail, homogeneity,
  completeness
- modularity
  architecture: refinement, decomposition
  complexity of components
- Proof Obligations (PO)
  number, rate of coverage by the *automatic prover*
  complexity
- maintenance

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

B model traceability



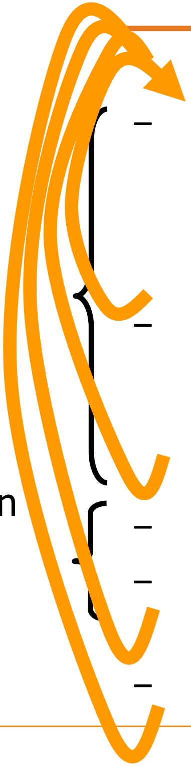
Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

B model traceability

- traceability is needed
  - because proof does not cover passing from non-formal to formal
- justification file
  - for each requirement
    - where it is formalized (precise or global)
    - why it is not formalized
- dictionary
  - precise definitions in natural language of every data / operation
- natural language retranslation
  - retranslation in natural language of the B specification, using the dictionary definitions

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

B development - an iterative process



building & adjusting the model

- modeling a sub-part
  - understanding the requirements, interviews
  - formalizing (making choices) in the current model
  - at some point: type check, PO examination
- when things go wrong, change the model
  - static error, visibility or architecture error
  - too many PO or too complex PO

'debug' & integration

- interactive PO demonstrations
- (software) translation, integration

evolution

- modification, evolution, precision of the requirements

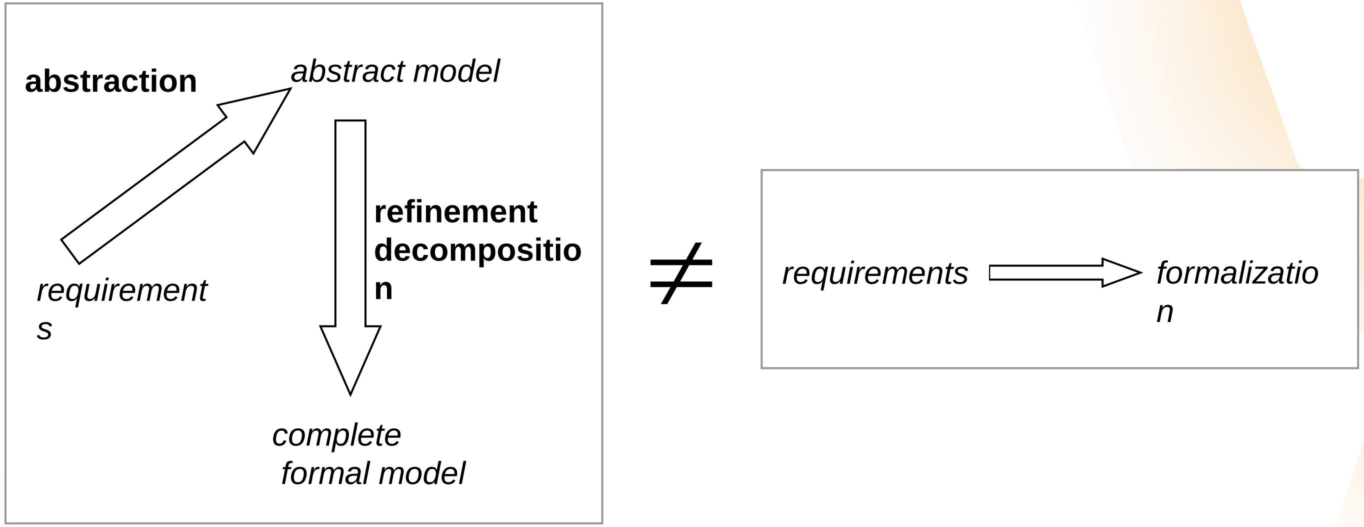
Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

formalization choices

- the notion of abstraction
- reusing pre-defined decomposition (in other formalism)
- modules and project
- building an architecture of modules: refinement and *imports* decomposition
- architecture rules
- static and dynamic properties
- some advice: choosing expressions, PO complexity, common pitfalls

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

abstraction of a B model



B approach

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

advantages and drawbacks of abstraction

- the requirements are
  - better understood
  - better expressed
- their formalization is
  - easier to track
  - easier to maintain

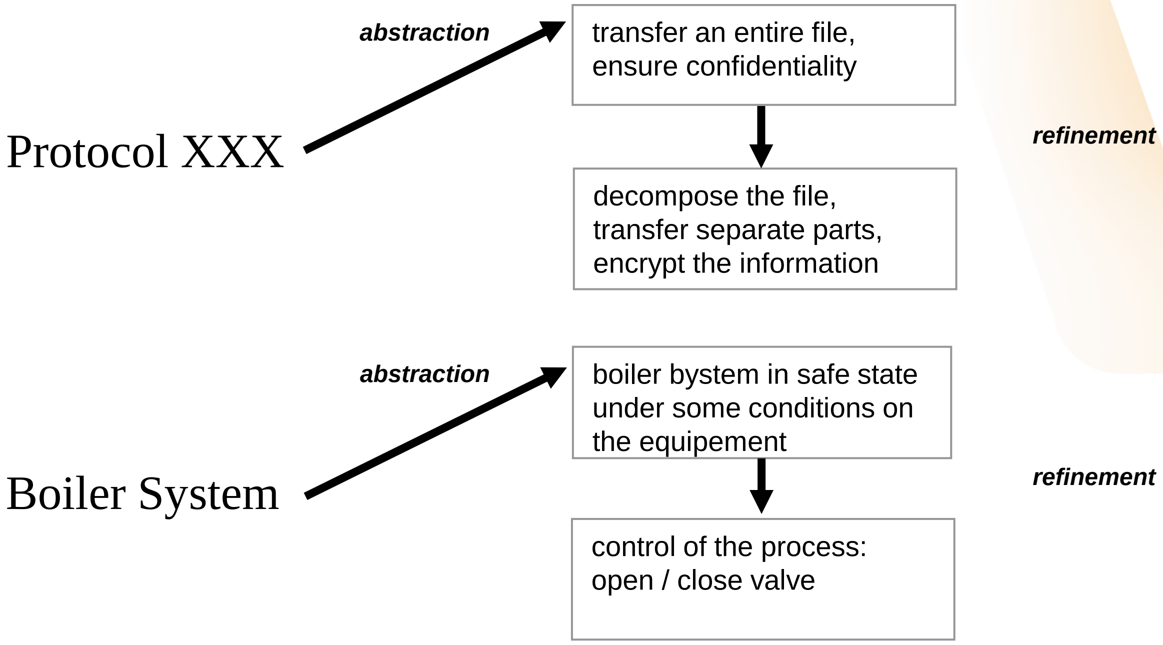
- *to abstract is difficult*
- *to abstract* efficiently *may be very difficult*

  a ‘good’ abstraction is *simple*

  ➔ but not simplistic …

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

examples of abstraction


abstraction
Protocol XXX
transfer an entire file, ensure confidentiality
decompose the file, transfer separate parts, encrypt the information
refinement
abstraction
Boiler System
boiler bystem in safe state under some conditions on the equipement
control of the process: open / close valve
refinement

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

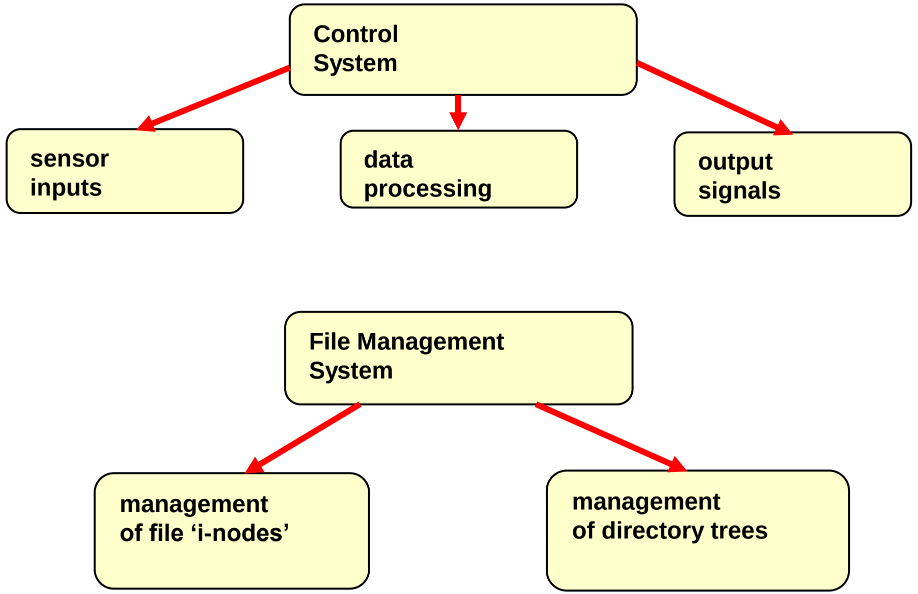
reusing pre-defined decomposition

an ‘architecture’ arises out of decomposing the specification

- reusing pre-defined decompositions
  - it may be interesting to reuse initial systems analysis: functional analysis, statecharts, …
  - it may also be misleading!
- decomposition principles in B
  - decompose the abstract model
  - aim to minimize complexity

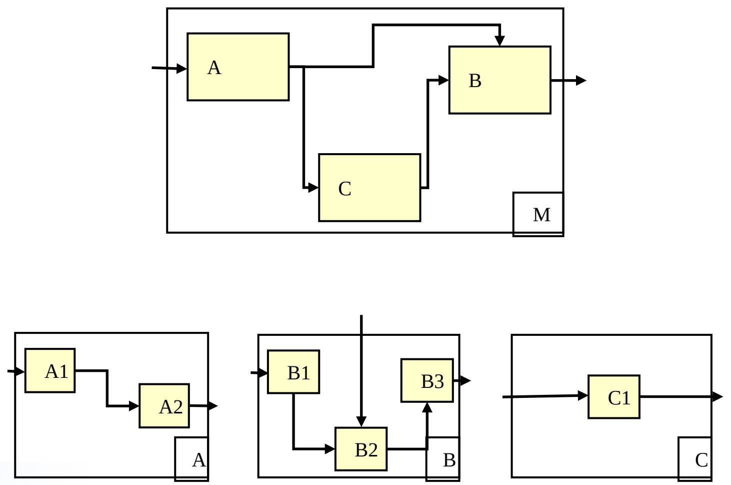
Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

pre-defined decomposition (standard architectures)



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

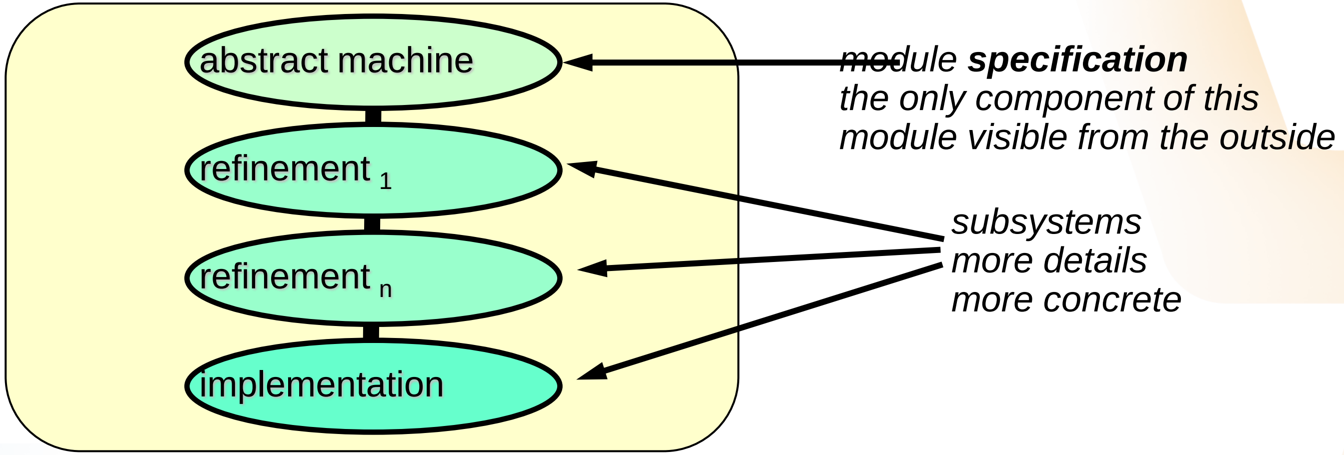
reusing pre-defined decomposition (from initial systems analysis)



$$\begin{tikzcd}
 & & & M \arrow[dlll] \arrow[d] \arrow[drr] & & \\
A \arrow[d] \arrow[dr] & & & B \arrow[dl] \arrow[d] \arrow[dr] & & C \arrow[d] \\
A1 & A2 & B1 & B2 & B3 & C1
\end{tikzcd}$$

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

components of a B module



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

B project and B modules

- a project is build with modules
  - different kinds of modules

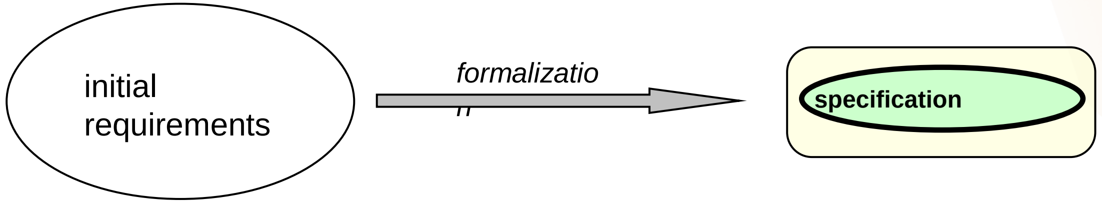
kind of module / characteristics	Abstract Module	Concrete Module	
		Refined Module	Basic Module
has an 'abstract machine' (its specification)	yes	yes	yes
has an implementation (and possibly, some intermediate refinements)	no	yes	no
has associated code (Ada, C++, Java, ...)	no	yes (by translation)	yes (user-supplied)

  - concrete modules are associated with concrete software modules
  - abstract modules are only used for *inclusion*

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

architecture - 1 abstract machine

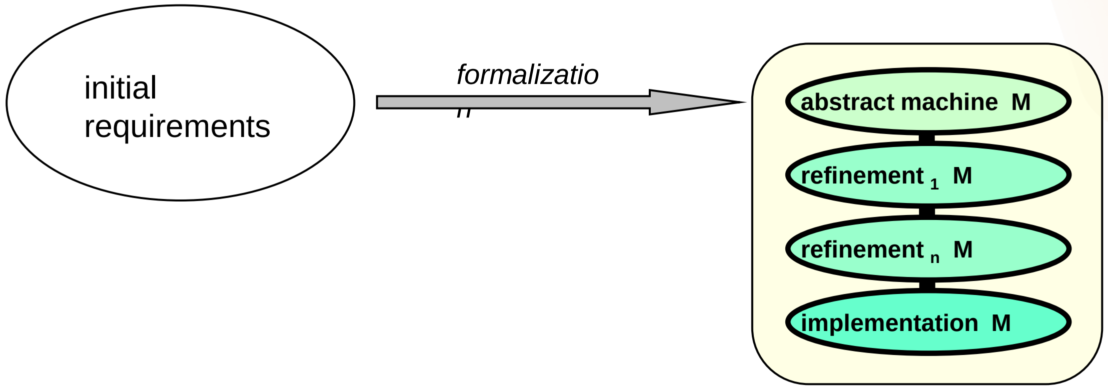
- minimal architecture: only 1 module with 1 abstract machine (no refinement)
- B system



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

architecture - 1 module with refinement

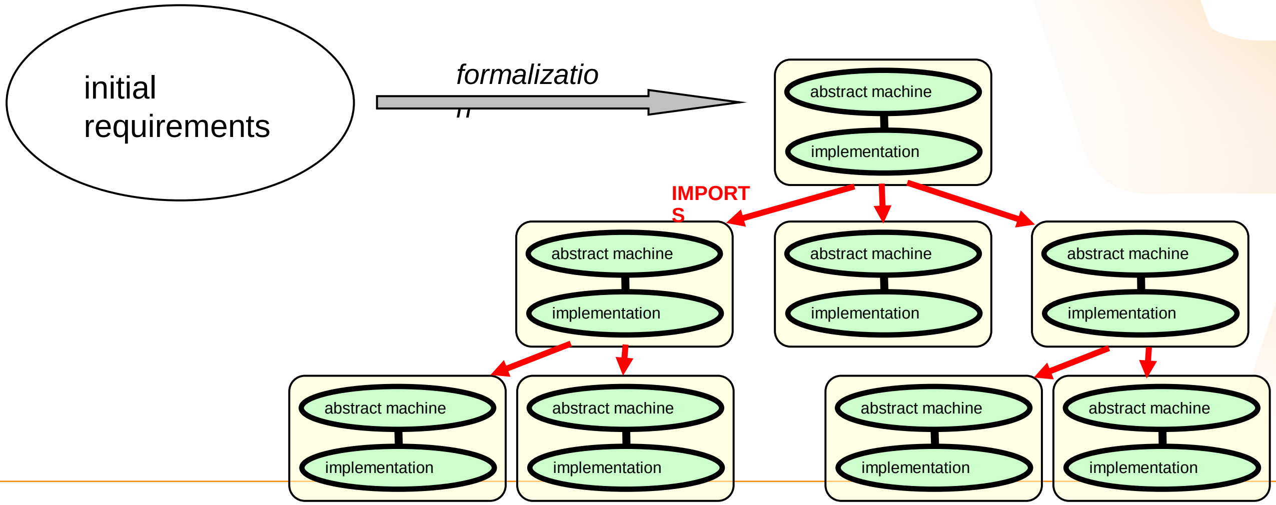
- 1 module with a column of refinements
- B system



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

architecture - decomposition of modules

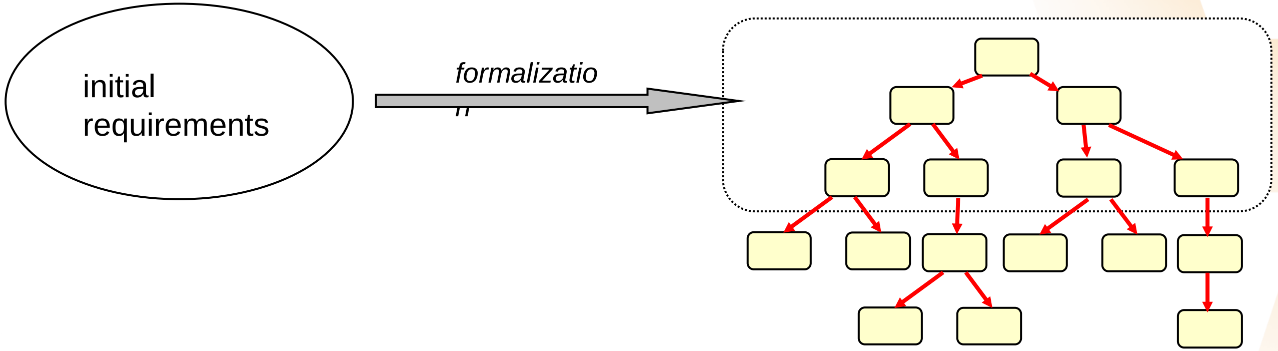
- N modules in an imports tree
- B software



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

decomposition - high / low levels

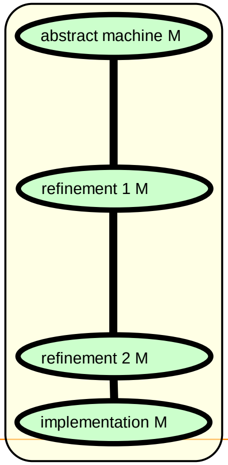
- a software project **may** be split into
  - *a high design level, to formalize requirements*
  - *a low design level, to implement the requirements*

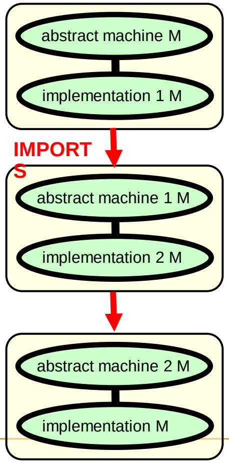


Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

architecture - refinement and decomposition

- comparison between refinement and imports decomposition
  - *intermediate refinement can also be expressed by* imports *decomposition*





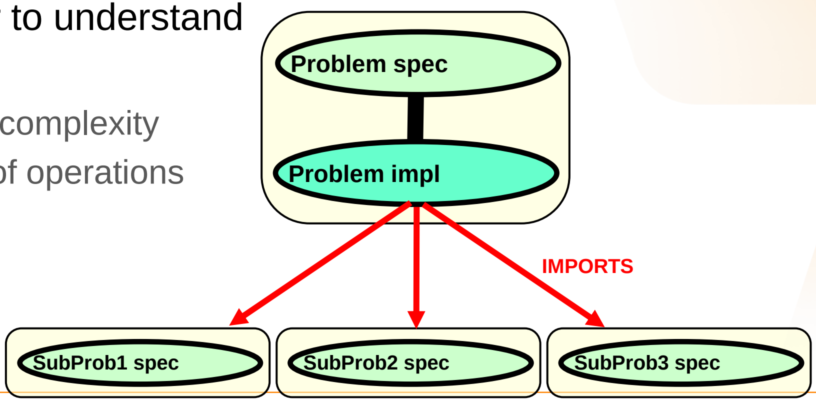
Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

modular decomposition

- imports decomposition provides
  - *decomposition into subsystems*
  - *module splitting (refinement does not)*
- advantages of decomposition
  - easier to read, easier to understand
  - easier to prove

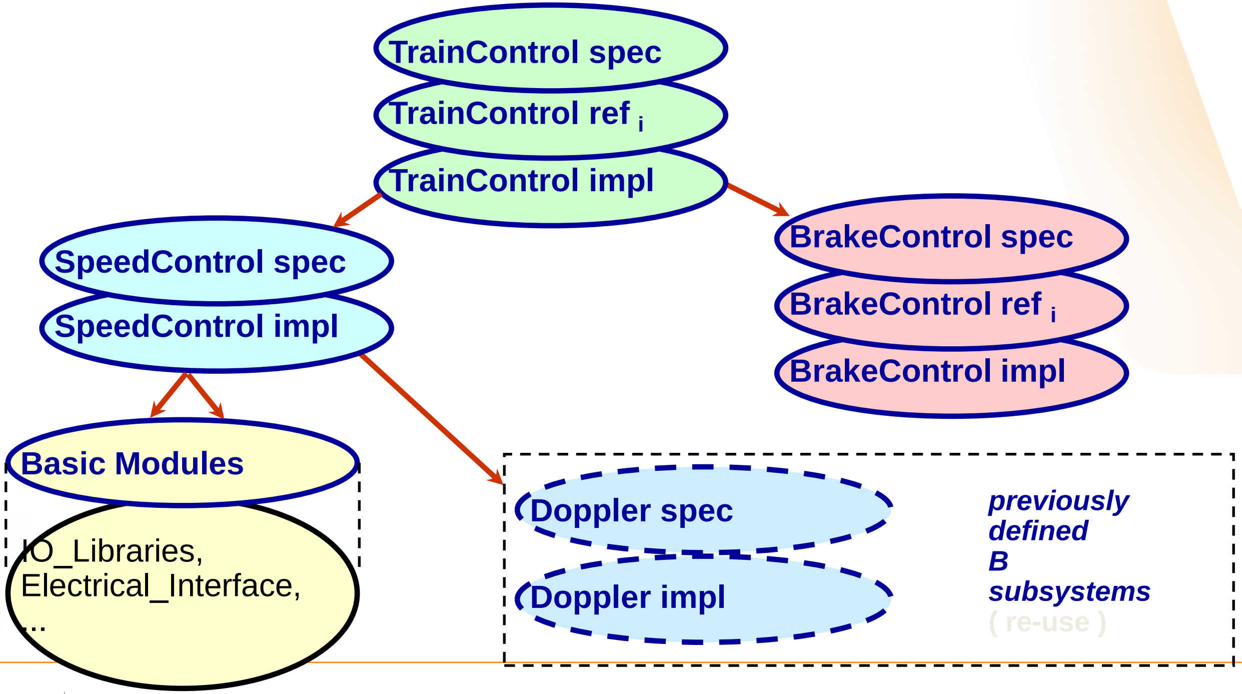
    breaking down proof complexity

    factorizing the proof of operations



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

example



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

modular decomposition - notion of graphs

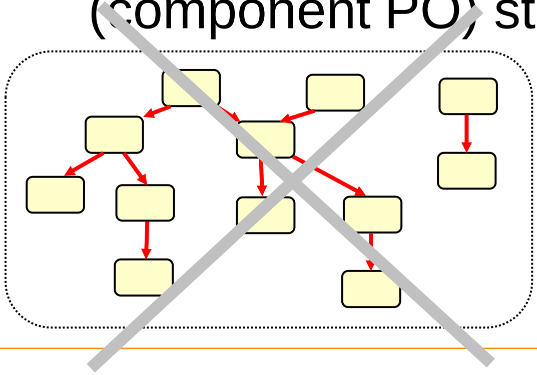
- dependency graph
  - graph made up of the project concrete **modules** and of the **IMPORTS** and **SEES** links between those modules
  - the links are oriented: MACHINE M SEES N, the link goes from module M to module N
  - the order of initialisation of the project (calling all the INITIALISATION procedures in a valid order) is determined by the dependency graph

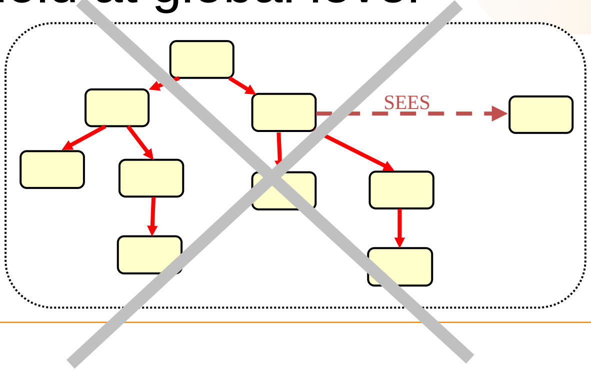
- IMPORTS graph
  - graph made up of the project concrete **modules** and of the **IMPORTS** links between those modules

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

modular decomposition - IMPORTS rules

- the imports graph must be a tree
  - each concrete module except the tree root must be *imported* in the project
  - to insure that the properties proved locally (component PO) still hold at global level

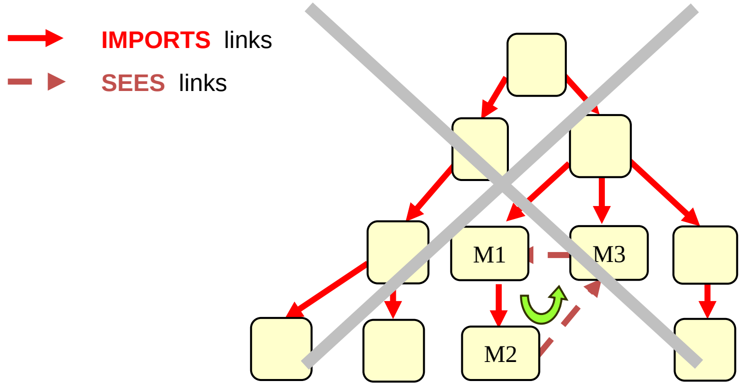




Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

modular decomposition - dependency rules

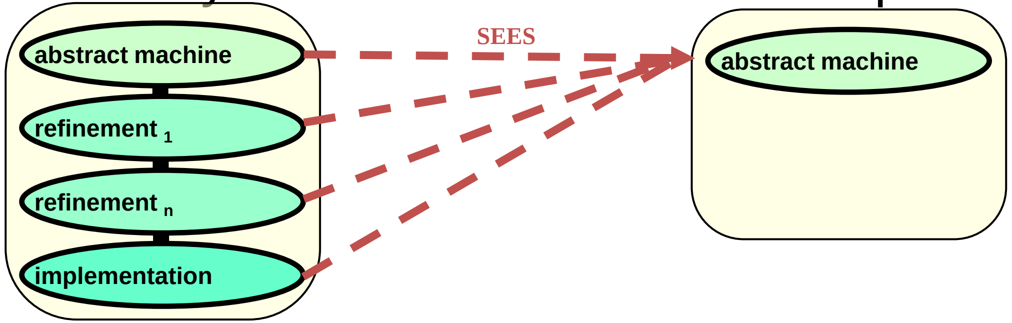
- the dependency graph must not have any ‘cycle’
  - there is no valid order of initialisation



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

modular decomposition – SEES rules (1)

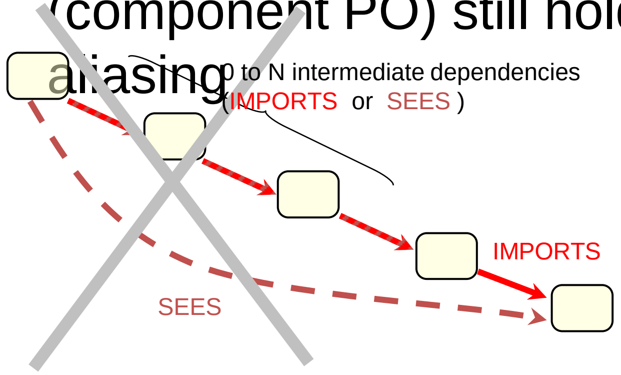
- a SEES clause provides *read-only access* to modules
- a module *seen* by a component must remain *seen* by the refinements of the component

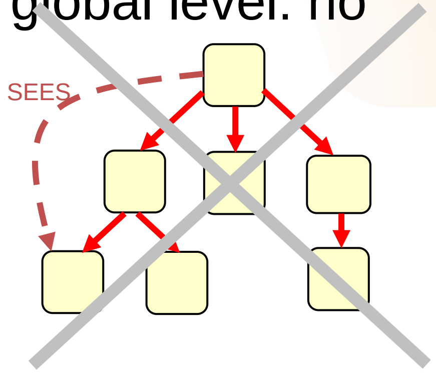


Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

modular decomposition - SEES rules (2)

- a component must not *see* a module *imported* by a transitively dependant module
- to insure that the properties proved locally (component PO) still hold at global level: no aliasing

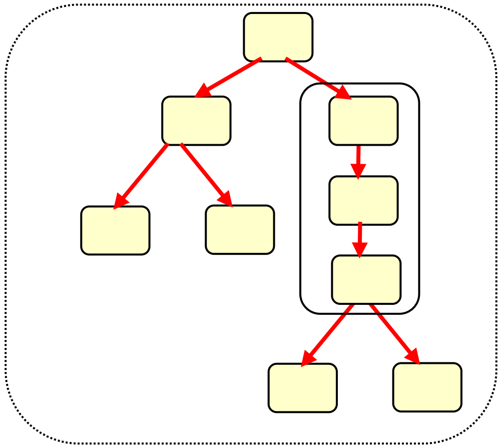




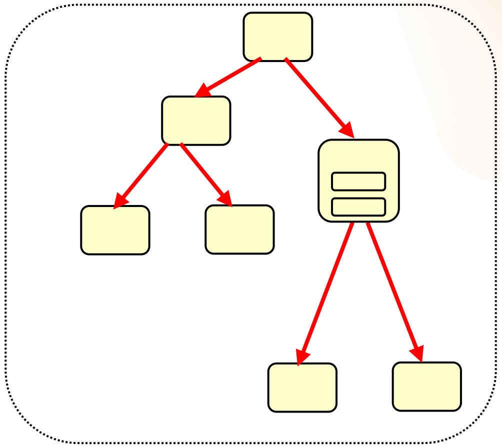
Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

architecture simplification

- use local operations to compact the imports tree
  - when a module has local modules only because of architecture rules:



IMPORTS tree



IMPORTS tree with local operations

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

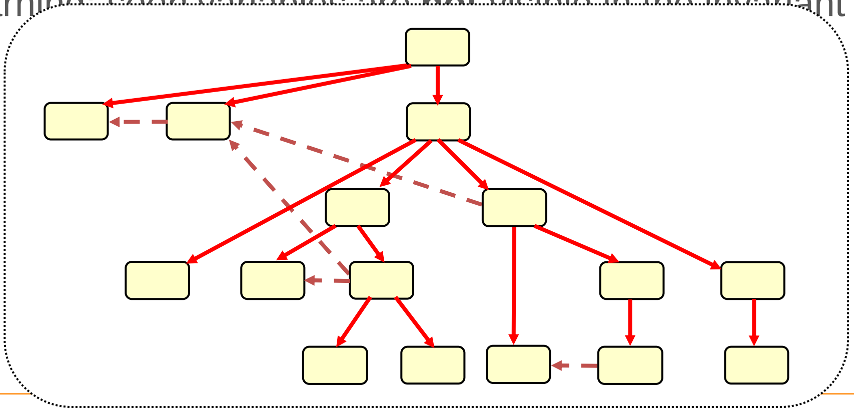
use of the SEES clause

- ***sees* on a stateless module**
- sees on a ‘brother’ or ‘cousin’ module

  read-only access to *seen* variables within substitutions

  call to *seen* operations that do not modify variables

  warning: seen variables are **not** visible in the invariant



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

stateless modules

- definitions file
  - scalar constants, concrete types with known values
  - referenced in the DEFINITIONS clause of any project component
  - source factorization, easier PO demonstration, PO maintenance may be difficult
- stateless module
  - **no variable**
  - SETS or constants (concrete or abstract)
  - purely functional operations:
  - *imported* at the highest level
  - can be *seen* from everywhere in the proje

```
$r \longleftarrow op(i_1, \ldots, i_N) \triangleq$
    PRE $P_{i1, \ldots, iN}$ THEN
        $r := f(i_1, \ldots,$
$i_N)$
    END
```

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

specifying the properties of a subsystem

- static properties
  - valid values for subsystem ‘state-variables’
  - these are specified by its **INVARIANT**

    expressed as a predicate over such values

- dynamic properties
  - valid ‘changes-of-state’ for the subsystem
  - these are specified by its **OPERATIONS**

    expressed in terms of substitutions

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

specifying static properties – the invariant (1)

– valid values of state-variables, from the same module

MACHINE
  MeasureLevel
VARIABLES
  LowWaterMeas, HighWaterMeas, …
INVARIANT /* must *always* hold (after initialisation) */

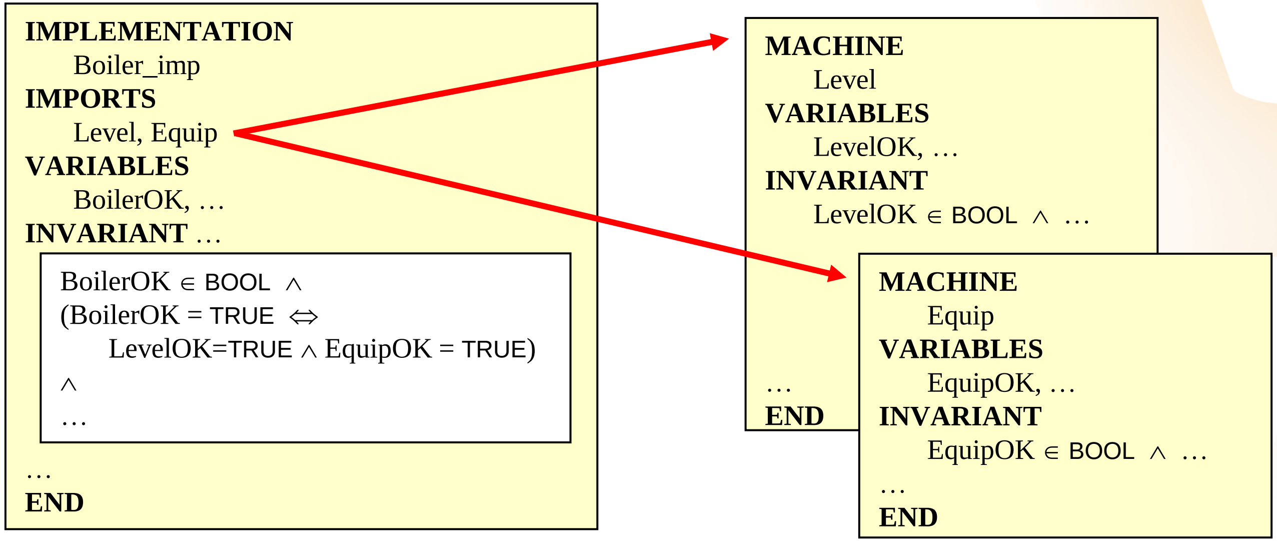
$\text{LowWaterMeas} \in \text{NAT} \land \text{HighWaterMeas} \in \text{NAT}_1 \land$
$\text{LowWaterMeas} < \text{HighWaterMeas} \land$
…

INITIALISATION /* must *establish* the invariant */
  …
OPERATIONS /* must *preserve* the invariant */
  UpdateValues $\triangleq$
    BEGIN … END; …
END

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

specifying static properties – the invariant (2)

➔ valid values of state-variables, from different modules



```
IMPLEMENTATION
    Boiler_imp
IMPORTS
    Level, Equip
VARIABLES
    BoilerOK, …
INVARIANT …
    BoilerOK ∈ BOOL ∧
    (BoilerOK = TRUE ⇔
        LevelOK=TRUE ∧ EquipOK = TRUE)
    ∧
    …
…
END
```

```
MACHINE
    Level
VARIABLES
    LevelOK, …
INVARIANT
    LevelOK ∈ BOOL ∧ …
…
END
```

```
MACHINE
    Equip
VARIABLES
    EquipOK, …
INVARIANT
    EquipOK ∈ BOOL ∧ …
…
END
```

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

specifying dynamic properties – the operations (1)

➔ valid changes-of-state

```
MACHINE
    Resources
VARIABLES
    available, in_use, faulty
INVARIANT
    available ⊆ RESOURCES ∧ …
OPERATIONS
    bb ⟵ AnyAvailable ≙
            bb := bool(available ≠ ∅)
    ;
    xx ⟵ AcquireResource ≙
        PRE available ≠ ∅ THEN
            ANY rr WHERE
            rr ∈ available
            THEN
            available := available - {rr} ||
            in_use := in_use ∪ {rr} ||
            xx := rr
            END
        END
END
```

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

specifying dynamic properties – the operations (2)

- *to specify an operation you have to choose between*
  - ➔ the weakest specification: the most indeterministic

    variables :(*Invariant*)
  - ➔ the strongest specification: completely deterministic

    variables := *values*
  - ➔ an intermediate specification

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

modeling - choosing the right expressions

- use
  - scalars (+, -, bool)
  - sets ($\cup$, $\cap$, -, $\times$, $\{x|P_x\}$)
  - relations and functions (dom, ran, r^{-1}, $\lhd$, ‘;’, $f(x)$, $f[X]$, λ)
- use carefully
  - special operators (card, closure), difficult to prove
  - sequences: difficult to prove, use functions instead
- do not use
  - records: inefficient in the current version of Atelier B
  - trees: inefficient

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

minimizing the number of PO

- ‘risky’ constructs
  - complex algorithmic refinements
  - sequences of conditional substitutions
  - large sequences of operations
  - conditional substitutions (IF, SELECT) in abstract machines
- possible solutions
  - decompose so that fewer or simpler proofs are required
  - introduce an additional level of refinement
  - improve the original abstraction
  - use local operations in implementations
  - in abstract machines, use a `becomes such that’ substitution instead of conditional substitutions (the caller gets fewer but harder PO)

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

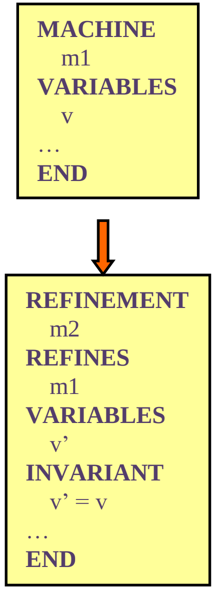
common pitfalls

- **using B like a programming language: having no abstraction**
- regarding B like an Object Oriented method
  - the analogy leads to severe drawbacks
- machine instanciation (IMPORTS or INCLUDES renamed machines)
- machine parameters are tempting because they look like genericity
  - they have drawbacks (type checking, translation)
  - instead a new genericity technique is rising: instanciation of stateless machines (cf. *Higher Order B*, J.R.-Abrial)
- too much (complex) properties
  - leading to too many or too complex PO
  - instead change the architecture, abstract, add refinement or *imports* decomposition, compromise (get rid of some properties)

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

sparing names : homonymy

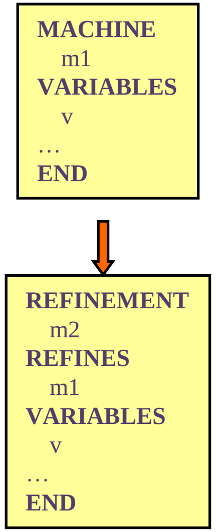
- *if you need this :*



```
MACHINE
   m1
VARIABLES
   v
…
END
```

```
REFINEMENT
   m2
REFINES
   m1
VARIABLES
   v'
INVARIANT
   v' = v
…
END
```

- *you can use homonymy :*



```
MACHINE
   m1
VARIABLES
   v
…
END
```

```
REFINEMENT
   m2
REFINES
   m1
VARIABLES
   v
…
END
```

- *implicit gluing invariant :*
  « v (in refinement) = v (in machine) »
- *added in the invariant of the refinement*

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

modeling tips - abstract constant functions

- *abstract constant functions* are the way to formalize in B programming language functions

  as operations are the way to formalize programming language procedures
- the specification of an *abstract constant function* is expressed in the PROPERTIES clause

  they have no side effect, as they are not related to variables
- an *abstract constant function* can be used everywhere an (abstract) expression can be used (invariant, precondition, …)
- it can be implemented as an operation

  input parameters correspond to the domain, output to the target set,
  its specification is: … ouput := f(input) …
  its implementation is a relevant algorithm

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

modeling tips - abstract constant functions

– example of *abstract constant function*

```
MACHINE
    …
ABSTRACT_CONSTANTS
    inc2
PROPERTIES
    inc2 ∈ ℤ → ℤ ∧
    ∀x.(x ∈ ℤ ⇒ inc2(x) = x + 2)
OPERATIONS
    y ⟵ Calc_inc2(x) ≙
    PRE x ∈ INT ∧ x+2 ∈ INT THEN
        y := inc2(x)
    END
…
END
```

```
IMPLEMENTATION
    …
OPERATIONS
    y ⟵ Calc_inc2(x) ≙
    BEGIN
        y := x + 2
    END
…
END
```

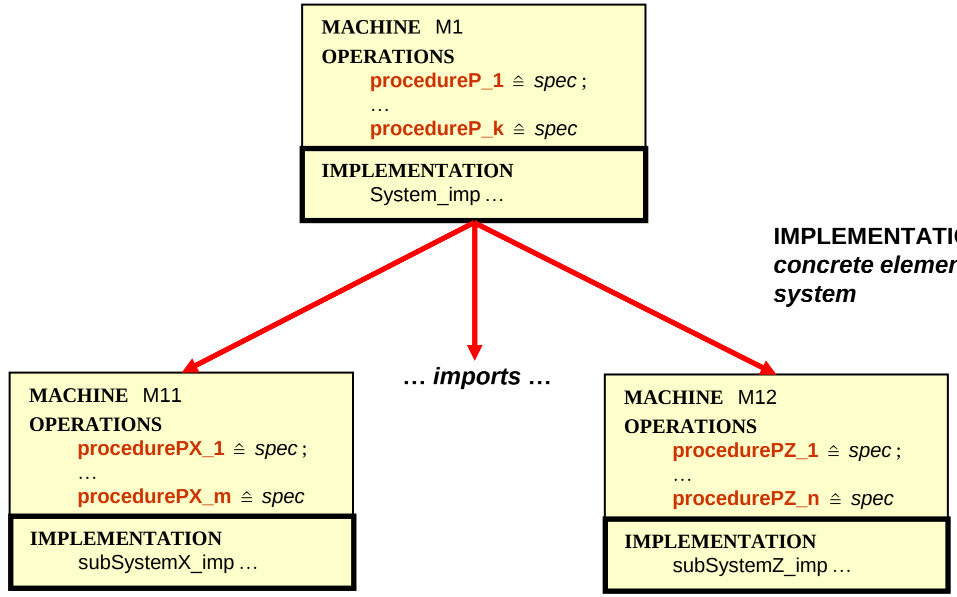
Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

‘event-driven’ B versus `procedural’ B

- 2 different interpretations of the same formalism …
- procedural B
  - operations ≈ ‘services’ that are *called* (in certain contexts)
  - modeling approach: description of abstract procedures
- event-driven B
  - operations ≈ ‘events’ that may *occur* (under certain conditions)
  - modeling approach: description of abstract events

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

procedural B (architecture)



IMPLEMENTATION components:
concrete elements of the final system

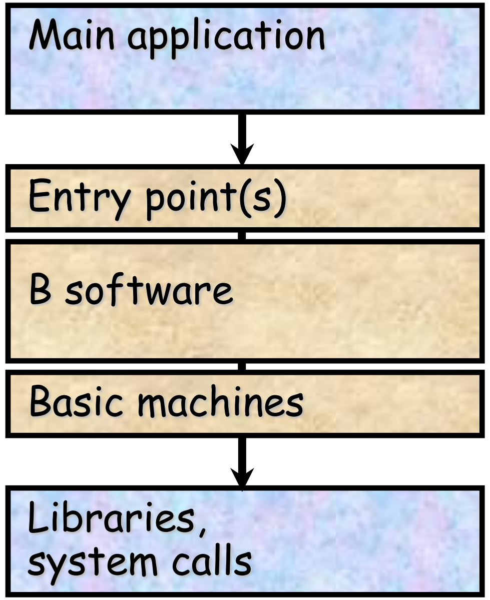
Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

procedural B - characteristics

- 99% cases : procedural B used to develop **software parts**
- the B procedures are used by the outside and may use non-B procedures through basic machines
- the notion of time
  - a sequence of `immediate’ procedure calls: the preconditions must hold, they are not checked at run-time
  - the B procedures may count cycles (it is called regularly from an outside “scheduler”)
  - the B procedures may access some clock outside B (through basic machines)

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

procedural B – integration (example : software system)



Ex: real time monitor, GUI

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

procedural B - architectural characteristics

- relatively few (intermediate) levels of refinement
  - usually to help the prover, not to add new specification
- modular decomposition based on IMPORTS
- procedures decomposed in terms of sub-procedures
  - operations call other operations in implementation
- implementations
  - ‘concrete’ body of components (constants values, initialisation and operations bodies)
  - often translated into a classic programming language (Ada, C, Java)

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

procedural B : operation precondition

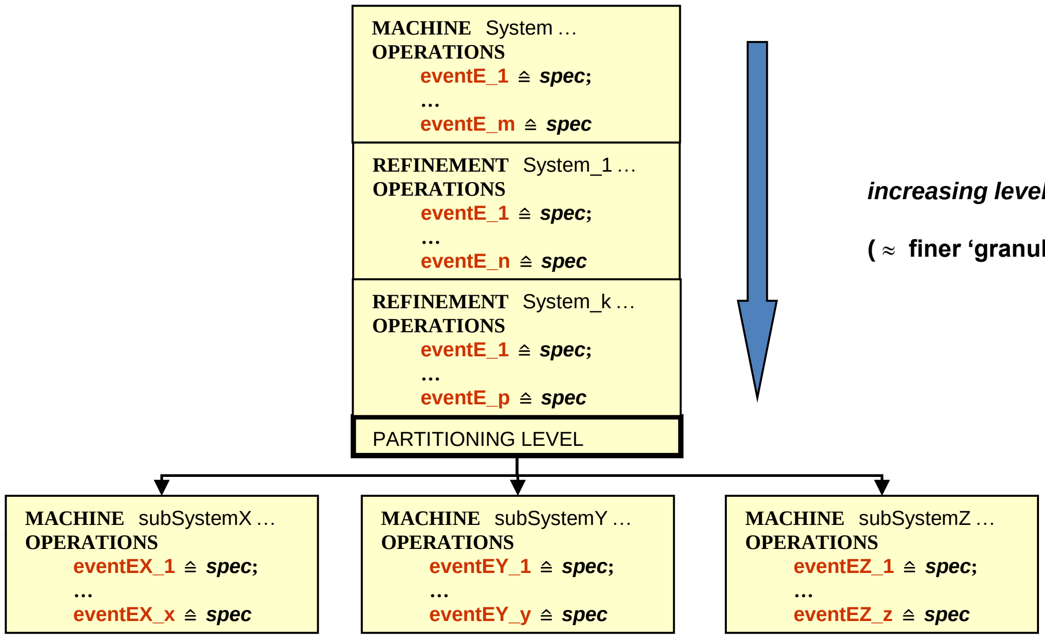
- they formalize the conditions under which the operation can be called
- the precondition has to be proved when calling the operation (the PO is part of the refinement PO of the operation that calls the operation)

```
IMPLEMENTATION
    …
IMPORTS
    Resources
OPERATIONS
    op ≙ …
    VAR IsAvlb IN
        IsAvlb ⟵ AnyAvailable ;
        IF IsAvlb = TRUE THEN
                res ⟵
AcquireResource
        END ;
    …
END
```

```
MACHINE
    Resources
VARIABLES
    …
INVARIANT
    …
OPERATIONS
    bb ⟵ AnyAvailable ≙
        bb := bool(available ≠ ∅)
    ;
    xx ⟵ AcquireResource ≙
    PRE available ≠ ∅ THEN …
END
END
```

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

event-driven modeling (architecture)



```
MACHINE  System …
OPERATIONS
        eventE_1 ≙ spec;
        …
        eventE_m ≙ spec
```

```
REFINEMENT  System_1 …
OPERATIONS
        eventE_1 ≙ spec;
        …
        eventE_n ≙ spec
```

```
REFINEMENT  System_k …
OPERATIONS
        eventE_1 ≙ spec;
        …
        eventE_p ≙ spec
```

PARTITIONING LEVEL

```
MACHINE  subSystemX …
OPERATIONS
        eventEX_1 ≙ spec;
        …
        eventEX_x ≙ spec
```

```
MACHINE  subSystemY …
OPERATIONS
        eventEY_1 ≙ spec;
        …
        eventEY_y ≙ spec
```

```
MACHINE  subSystemZ …
OPERATIONS
        eventEZ_1 ≙ spec;
        …
        eventEZ_z ≙ spec
```

increasing level of detail

(≈ finer ‘granularity’)

hardware elements ***software elements*** ***external elements*** ***(for example…)***

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

event-driven B - characteristics

- one B system project = model of a closed system
  - every part of the system is taken into account
  - the system may have heterogeneous parts: hardware, software, mechanics, human, …
- the notion of event
  - an event may only **occur** under some conditions: its guard
  - if a guard holds the event may be triggered at any time (but we do not know when)
  - two events cannot occur at the same time
  - we prefer the notion of **causality** to the notion of **time**
    - *phase variables insure the right causality*
    - *state / transition diagrams*
  - **if needed**, timers or clocks can be modeled

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

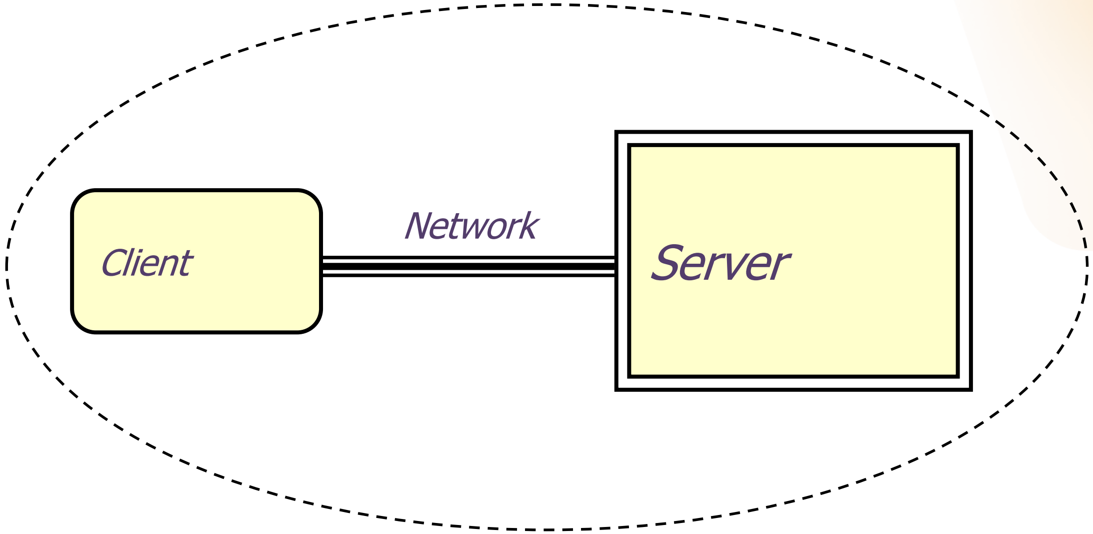
event-driven B - architectural characteristics

- description in terms of events
  - B operations with no input/output parameters and no precondition
- multiple levels of refinement
  - increasing level of detail for existing events
  - additional events (at a finer ‘granularity’)
  - subsystems identified by ‘partitioning’
  - modeling physical variables **AND** software variables
- the ‘terminal’ levels (e.g.)
  - hardware elements (behaviors as described by their ‘data sheets’)
  - software elements (procedures to be used with a separate ‘scheduler’)
  - external elements (contextual assumptions, limitations of usage etc…)

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

event-driven B - example (1)

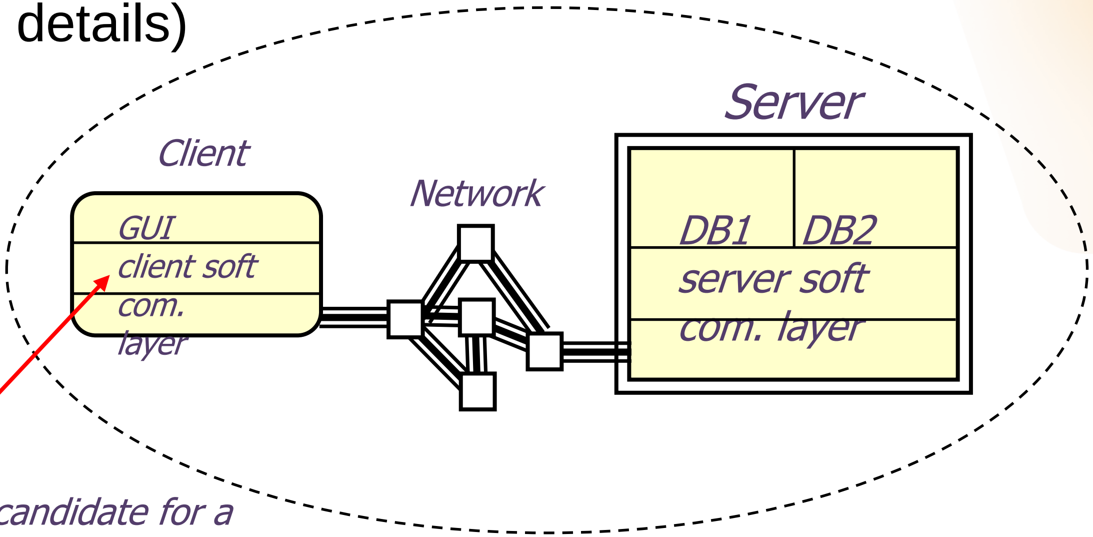
- a transaction system (first model)



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

event-driven B - example (2)

– a transaction system (a model with more details)



a good candidate for a B software development

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

event-driven B - form of event specifications

```
MACHINE
…
OPERATIONS …
    eventE_x ≙
        SELECT
            necessary conditions for this event to occur
        THEN
            substitution defining its state transformation
        END; …
    eventE_y ≙
        ANY variables WHERE
            necessary conditions, and value definitions
        THEN
            substitution defining its state transformation
        END; …
    eventE_z ≙
        BEGIN
            substitution defining its state transformation
        END; …
```

‘guard’ — **‘guarded event’: may only occur in certain conditions**

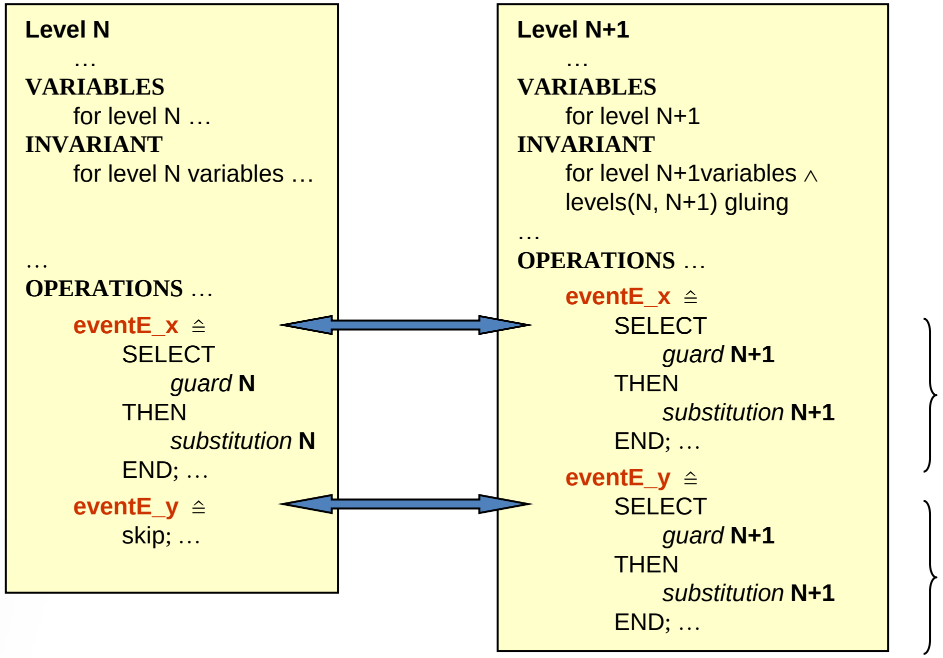
‘guard’ — **‘guarded event’ with (internal) parameters**

‘unguarded event’: may occur at any time (a demon)

NB: only the top-level SELECT (with a single branch) *or* ANY *specifies the guard for an event*

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

event-driven B - principles of refinement



```
Level N
      …
VARIABLES
      for level N …
INVARIANT
      for level N variables …


…
OPERATIONS …
      eventE_x ≙
            SELECT
                  guard N
            THEN
                  substitution N
            END; …
      eventE_y ≙
            skip; …
```

```
Level N+1
      …
VARIABLES
      for level N+1
INVARIANT
      for level N+1variables ∧
      levels(N, N+1) gluing
…
OPERATIONS …
      eventE_x ≙
            SELECT
                  guard N+1
            THEN
                  substitution N+1
            END; …
      eventE_y ≙
            SELECT
                  guard N+1
            THEN
                  substitution N+1
            END; …
```

refinement of an existing event: its guard at level N+1 must be *stronger* than that at level N

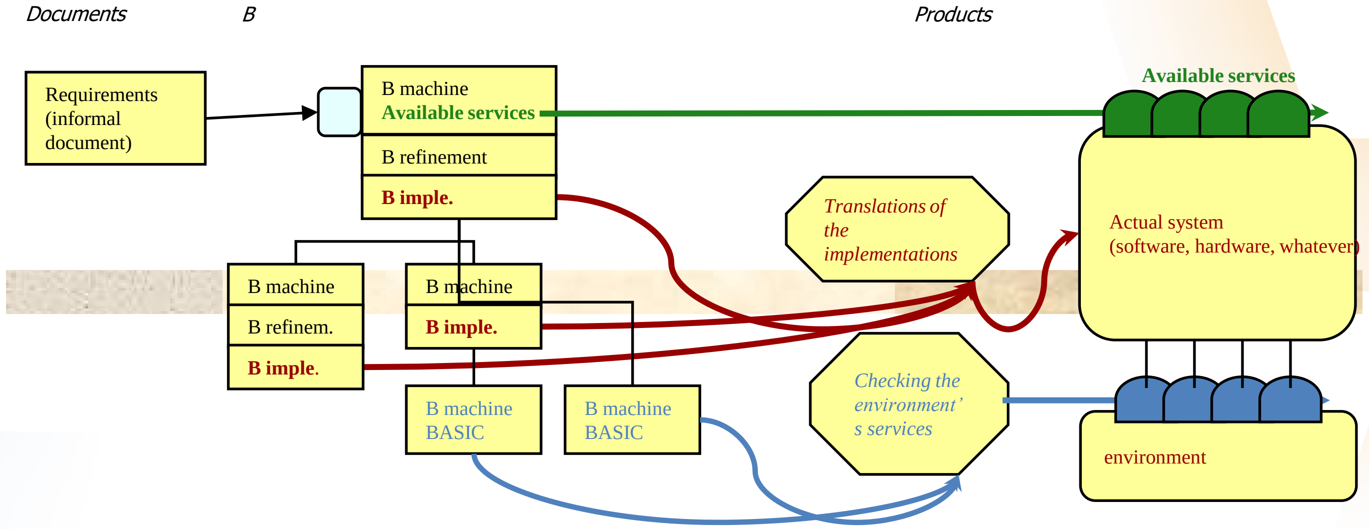
a ‘new’ event, not involved at level N, must refine `skip’

Copyright © 2022 CLEARSY

Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

procedural B : landmarks

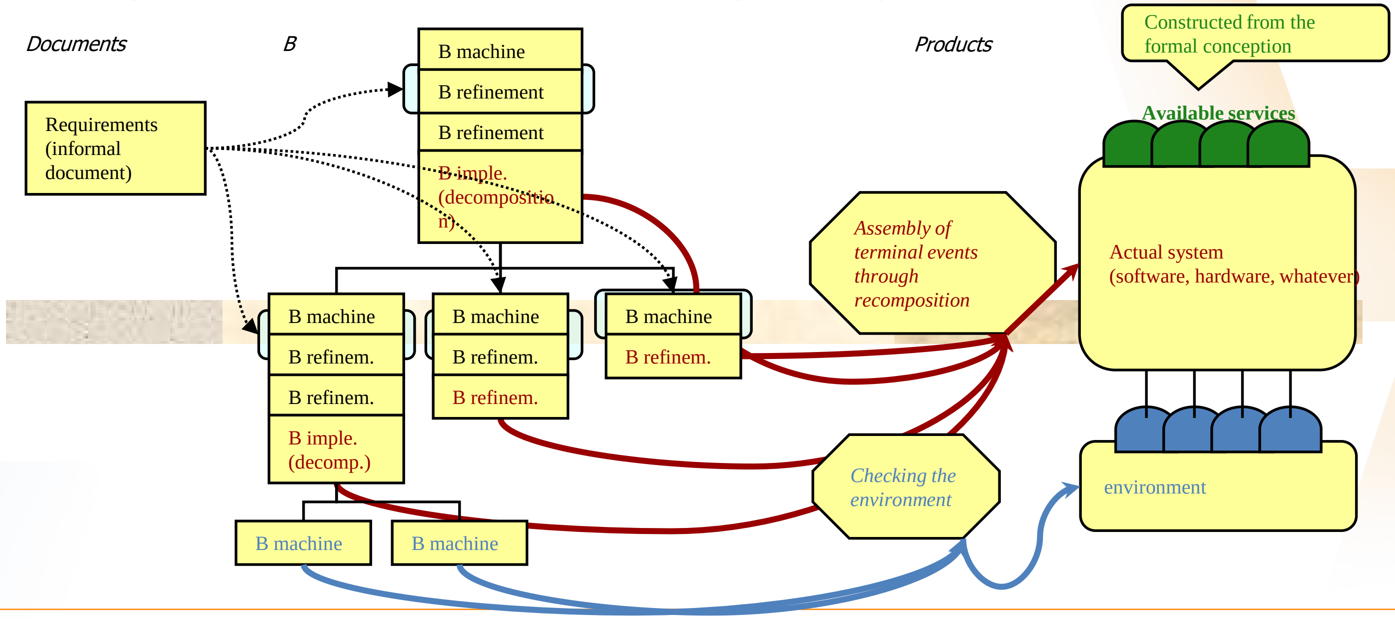
 *B operations = procedures to be called, **PRE** clause and operation parameters allowed*



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

event driven B : landmarks

 *B operations = events, **PRE** clause and operation parameters not allowed*



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

Correctness: Proof Obligation

the different kinds of PO,
overview of the proof process

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

proof obligations (PO)

- definition
  - a proof obligation is a logical formula that must be ***demonstrated*** in order for a B component to be semantically **correct**
- notation
  - in general, proof obligations have the form of implications,

    H
    $\Rightarrow$
    G

    where the 'hypothesis' *H* and the 'goal' *G* are *predicates*

  - the goal may involve *predicate transformers*,

    $[\,S\,]\;P$

    where *S* is a *substitution* and *P* is a *predicate*

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

purpose of the proof obligations

- internal consistency
  - to show that the invariant of an abstract machine holds for all *reachable* states

    e.g. for an operation Op of machine M,

    invariant of M $\land$ precondition of Op

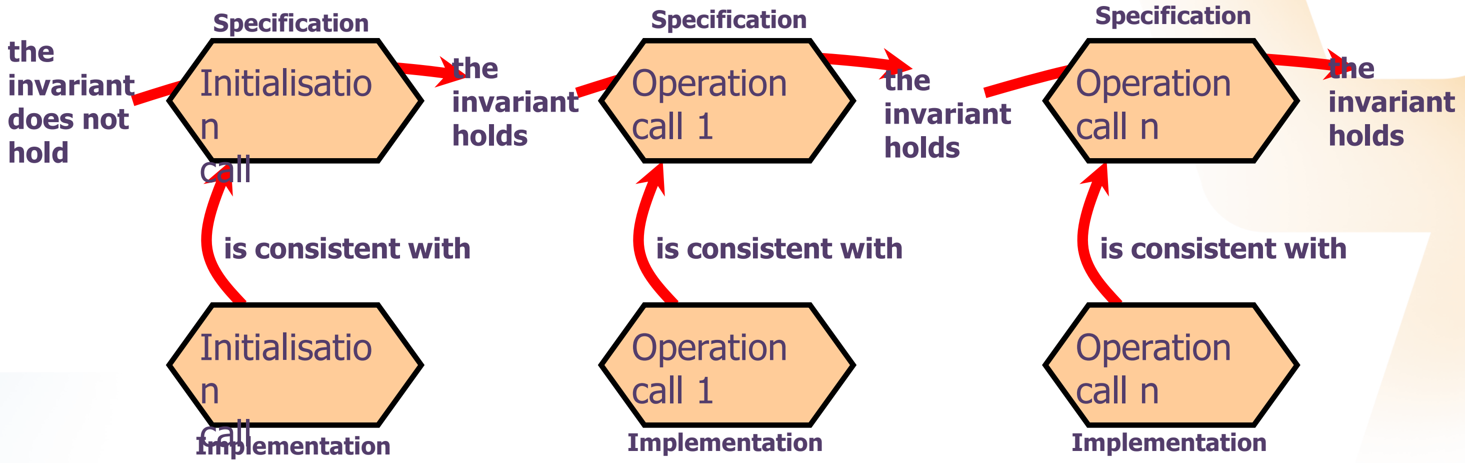
    $\Rightarrow$

    [substitution of Op] (invariant of M)
- correct development
  - to show that any refinement is consistent with its abstraction (regarding the refinement invariant)

    i.e. that the model for level n+1 preserves the properties specified at level n
  - to show that any decomposition maintains the properties required of the whole

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

what is proved?



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

abstract machine PO
- correctness of initialisation

- definition
  - *the initialisation of an abstract machine must* ***establish*** *its invariant*

- proof obligation

  static properties of the abstract machine, *excluding* its invariant

  $\Rightarrow$

  [includes initialisation ; machine initialisation] invariant

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

correctness of initialisation – example 1

```
MACHINE
    M
ABSTRACT_VARIABLES
    setv
INVARIANT
    setv ⊆ 1..10
INITIALISATION
    setv := ∅
…
END
```

PO for initialisation of M

$( )$

$\Rightarrow$

$[\ setv := \emptyset\ ]\ (\ setv \subseteq 1..10\ )$

that is,

$( )$

$\Rightarrow$

$\emptyset \subseteq 1..10$

Copyright © 2022 CLEARSY

Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

correctness of initialisation – example 2

```
MACHINE
    M
INCLUDES
    MA
ABSTRACT_VARIABLES
    setv
INVARIANT
    setv ⊆ 1..10
INITIALISATION
    setv := 1..natv
…
END
```

```
MACHINE
    MA
ABSTRACT_VARIABLES
    natv
INVARIANT
    natv ∈ 1..4
INITIALISATION
    natv :∈ 1..3
…
END
```

PO for initialisation of M

$$(\,)$$

$$\Rightarrow$$

$$[\ natv :\in 1..3\ ;\ \ setv := 1..natv\ ]\ \ (\ setv \subseteq 1..10\ )$$

Copyright © 2022 CLEARSY

Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

abstract machine PO
- correctness of operations

- definition
  - *any operation of an abstract machine must* ***preserve*** *its invariant*

- proof obligation

  static properties of the abstract machine $\wedge$
  precondition of the abstract operation
  $\Rightarrow$
  [abstract operation] (abstract invariant)

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

correctness of operations - example

```
MACHINE
    M
ABSTRACT_VARIABLES
    zz
INVARIANT
    zz ∈ ℕ  ∧  zz < 5
INCLUDES
    MA
OPERATIONS …
    Set ≙
        BEGIN  zz ← val  END;
    …
END
```

```
MACHINE
    MA
ABSTRACT_VARIABLES
    aa
INVARIANT
    aa ∈ 0..2
OPERATIONS …
    vv ← val  ≙
        BEGIN  vv := aa  END;
    …
END
```

PO for the abstract operation Set

$$
\begin{array}{l}
zz \in \mathbb{N} \ \land \ zz < 5 \ \land \\
aa \in 0..2 \\
\Rightarrow \\
[\, zz := aa \,] \ (\, zz \in \mathbb{N} \ \land \ zz < 5 \,)
\end{array}
$$

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

refinement PO - correctness of operations

- definition
  - *each operation of the refinement must **preserve** its invariant, without **contradicting** the corresponding abstract operation*

- proof obligation

  static properties of the abstract specification $\wedge$

  precondition of the abstract operation $\wedge$

  static properties of the refinement

  $\Rightarrow$

  [refined operation] $\neg$ [abstract operation] $\neg$ (refinement invariant and equality of result parameters)

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

correctness of operations - example

```
MACHINE
    M
ABSTRACT_VARIABLES
    va
INVARIANT
    va ∈ 0..10
OPERATIONS …
    nv ← newval ≙
        ANY vv WHERE
            vv > va
        THEN
            nv := vv
        END;
    …
END
```

```
REFINEMENT
    M_r
REFINES
    M
ABSTRACT_VARIABLES
    vr
INVARIANT
    vr ∈ ℕ ∧ vr > va
OPERATIONS …
    nv ← newval ≙
        BEGIN nv := vr END;
    …
END
```

PO for the refined operation newval

$va \in 0..10 \land vr \in \mathbb{N} \land vr > va$

$\Rightarrow$

$[\, nv' := vr \,] \neg\, [\, \text{ANY } vv \text{ WHERE } vv > va \text{ THEN } nv := vv \text{ END} \,] \neg\ (vr \in \mathbb{N} \land vr > va \land nv' = nv)$

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

refinement PO - correctness of initialisation

- definition
  - *the initialisation must **establish** its invariant, without **contradicting** the initialisation of its abstraction*

- proof obligation

  static properties of the refinement and all its abstractions,
  excluding their invariant

  ⇒

  [initialisation of includes / imports ; initialisation of the refinement]
  ¬ [abstract initialisation] ¬ (refinement invariant)

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

correctness of initialisation - example

```
MACHINE
    M
ABSTRACT_VARIABLES
    setv
INVARIANT
    setv ⊆ 1..10
INITIALISATION
    setv := ∅
…
END
```

```
REFINEMENT
    M_r
REFINES
    M
ABSTRACT_VARIABLES
    bitv
INVARIANT
    bitv ∈ 1..10 → BOOL ∧
    setv = bitv⁻¹ [{TRUE}]
INITIALISATION
    bitv := (1..10) × {FALSE}
…
END
```

PO for initialisation of M_r

$$( ) \Rightarrow [\, bitv := (1..10) \times \{FALSE\} \,] \neg \; [\, setv := \varnothing \,] \neg \, (bitv \in 1..10 \rightarrow BOOL \wedge setv = bitv^{-1} [\{TRUE\}])$$

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

implementation PO
- correctness of deferred values

- definition
  - *valuation of abstract sets and concrete constants must satisfy the module properties*
  - *there must exist possible values for the abstract constants of the module, that satisfy the properties*

- proof obligation

  visible properties of required machines

  $\Rightarrow$

  $\exists$ abstract constants · [values substitution] properties

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

correctness of deferred values - example

```
MACHINE
    M
SETS
    SS
ABSTRACT_CONSTANTS
    ac
CONCRETE_CONSTANTS
    cc
PROPERTIES
    ac ∈ SS  ∧  cc ∈ 1..10
…
END
```

```
IMPLEMENTATION
    M_i
REFINES
    M
VALUES
    SS = 0..255 ;
    cc = 1
…
END
```

PO for the deferred values

$$(\;) \Rightarrow \exists ac \cdot [SS := 0..255 ; cc := 1] (SS \in \mathbb{F}_1(\mathrm{INT}) \wedge ac \in SS \wedge cc \in 1..10)$$

)

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

component PO - correctness of assertions

- definition
  - *reminder: assertions are used to factorize properties*
  - *assertions of an abstract machine must **be deduced** from its static properties*
  - *assertions may be proved using previously proved assertions*

- proof obligation

  static properties of the abstract machine $\wedge$

  previous assertions (1..n-1)

  $\Rightarrow$

  assertion (n)

Copyright © 2022 CLEARSY

Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

correctness of assertions - example

```
MACHINE
    M
ABSTRACT_VARIABLES
    xx, yy, zz
INVARIANT
    xx ∈ ℤ ∧ yy ∈ ℕ ∧ zz ∈ ℤ ∧
    xx < 0 ∧ yy > 10 ∧
    zz = xx × yy
ASSERTIONS
    zz < 0
...
END
```

PO for the assertion:

$$
\begin{array}{l}
xx \in \mathbb{Z} \wedge yy \in \mathbb{N} \wedge zz \in \mathbb{Z} \wedge \\
xx < 0 \wedge yy > 10 \wedge \\
zz = xx \times yy \\
\Rightarrow \\
zz < 0
\end{array}
$$

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

component PO - correctness of instanciation

- definition
  - *actual parameters of includes or imports instantiation must* ***satisfy*** *the constraints of the formal parameters of the included or imported machine*

- proof obligation

  static properties of the component and its abstractions, *excluding* their invariant

  $\Rightarrow$

  [instantiation] (constraints of the imported machine)

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

correctness of imports - example

```
MACHINE
    M (pm)
CONSTRAINTS
    pm < 10
...
END
```

```
IMPLEMENTATION
    M_i (pm)
REFINES
    M
CONCRETE_CONSTANTS
    ci
PROPERTIES
    ci ∈ INT ∧ ci < pm/2
IMPORTS
    MA (ci)
...
END
```

```
MACHINE
    MA (pa)
CONSTRAINTS
    pa < 15
...
END
```

PO for the instantiation of MA

$pm < 10 \land$

$ci \in INT \land ci < pm/2$

$\Rightarrow$

$[\, pa := ci \,]\ (\, pa < 15 \,)$

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

correctness of integer arithmetic …

- overload of arithmetic operators in B0
  - in B0, arithmetic operators (+, -, /, mod, **) are restricted on INT
  - they overload the corresponding mathematical operators
  - INT is the subset of integers ($\mathbb{Z}$) assumed to be *representable* on the 'target machine', INT = MININT..MAXINT
  - in Atelier B 4.0, these pre-defined constants are:

    $\text{MAXINT} = 2^{31} - 1 \ \wedge\ \text{MININT} = - \text{MAXINT}$

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

component PO - well-defineness

- *in a component, expressions, predicates and substitutions must be well-defined. Examples:*

expression	*well-defineness condition*
a / b	*$b \neq 0$*
f(x)	*$x \in dom(f)$*
card(S)	*$S \in \mathbb{F}(S)$*

- *well-defineness has to be proved*
- *in B0 it has a special interpretation: no classic programming error (division by 0, arithmetic overflow, out of domain index)*
- consider the B0 assignment

  v0 := c1 + (v2/v3)
- the additional PO generated are

  $v2 \in INT \wedge v3 \in INT-\{0\} \wedge v2/v3 \in INT \wedge c1 \in INT \wedge c1+ (v2/v3) \in INT$

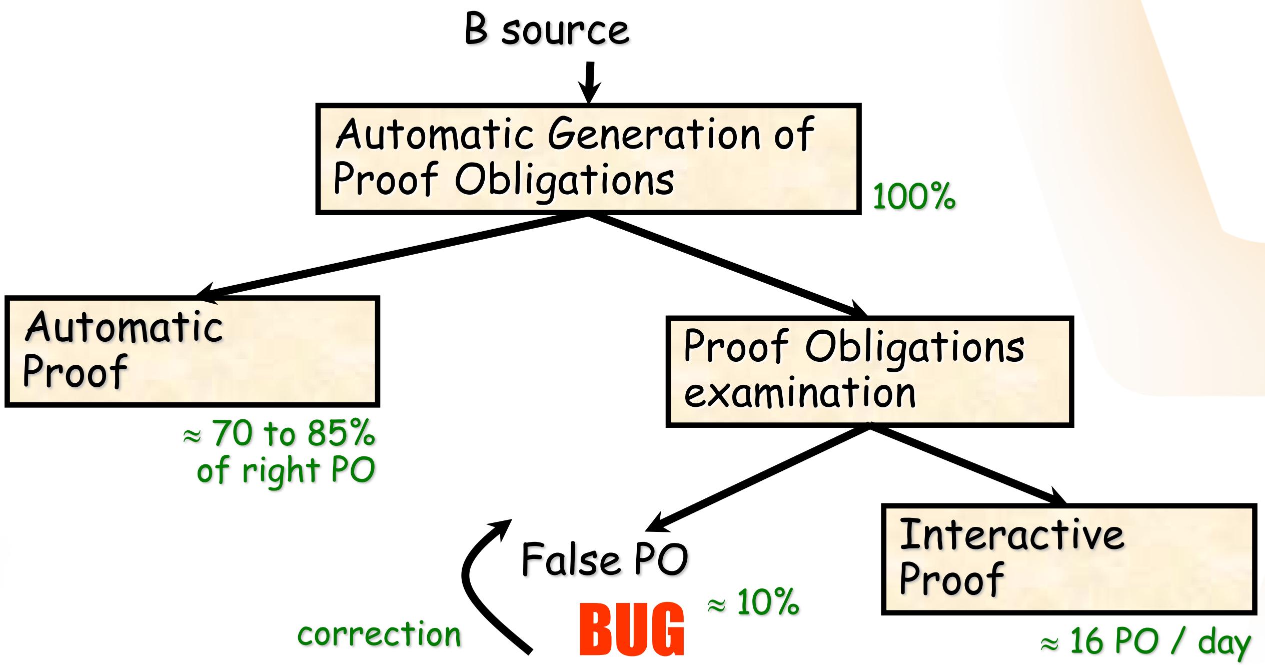
Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

special naming conventions in PO

- renaming principle
  - during symbolic transformations, the POG needs sometimes to create a ‘fresh’ variable name to distinguish variables having the same name but denoting different values: $x = 20 \wedge \forall x.(x \in 1..10 \Rightarrow x \leq 15)$
- interpretation of variable names in abstract machines PO
  - ***ident*** *a variable value before substitution*
  - ***ident$i*** (where i ≥ 0) a ‘fresh’ name
- interpretation of variable names in refinements PO
  - ***ident*** *a variable of an abstraction*
  - ***ident$1*** *a variable of the component or of an imported machine*
  - ***ident$i*** (where i ≥ 2) a ‘fresh’ name
  - ***ident$i*** (where i ≥ 7777) a ‘fresh’ variable (at the end of loops)

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

proof verification



Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

validity of proofs (user-rules)

- validation of user-supplied rules
  - the user-rules supplied during interactive proof must be ***validated***

    a rule that is ‘almost right’ is an *invalid* rule
  - such validations are carried out by

    specialized tools in Atelier B (using the ***Predicate Prover***)

    or ***manual demonstrations*** proving that the rules hold

‘Proof file’ for the B project

  - some general advice

    make sure that all possible cases are covered

    check that the variable names are unique

    attend B training level 3 (3 days)

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

Refinement and implementation issues

gluing invariants,
constructing correct loops

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

‘gluing invariants’ in a refinement

- the invariant of a refinement specifies
  - types and properties of the ‘new’ variables (introduced by the refinement)
  - links between abstraction variables and refinement variables: **‘gluing invariant’**

```
MACHINE
    AA
     ABSTRACT_VARIABLES
    SS
INVARIANT
    SS ⊆ NAT
...
END
```

```
REFINEMENT
    AA_r
REFINES
    AA
         ABSTRACT_VARIABLES
    mm
INVARIANT
    mm ∈ NAT  ∧  mm = max (SS ∪ {0})
…
END
```

  - special case: ‘implicit gluing’ (homonymy) abstraction variables and refinement variables having the same name are taken as ‘identical’

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

‘gluing invariants’ in an implementation

- the invariant of an implementation specifies
  - types and properties of the ‘new’ **concrete** variables (introduced by the implementation)
  - links between abstraction variables and, implementation or ***imported*** variables: **‘gluing invariant’**
  - special case: ‘implicit gluing’ (homonymy) abstraction variables and, implementation or ***imported*** variables with the same name are taken as ‘identical’

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

variables gluing in an implementation - example

```
MACHINE
    AA
ABSTRACT_VARIABLES
    v1, v2
CONCRETE_VARIABLE
S
    v3, v4
INVARIANT
    v1 ∈ NAT ∧
    v2 ∈ BOOL ∧
    v3 ∈ INT ∧
    v4 ∈ INT
…
END
```

```
IMPLEMENTATION
    AA_i
REFINES
    AA
IMPORTS
    BB
CONCRETE_VARIABLES
    v5
INVARIANT
    v5 ∈ NAT ∧
    v1 = v0   /*  gluing invariant */
 /*  implicit gluing:
                  *         v2 = v2$1
                  *         v3 = v3$1
                  *         v4 = v4$1
                        */
…
END
```

```
MACHINE
    BB
ABSTRACT_VARIABLES
    v0, v2
CONCRETE_VARIABLE
S
    v4
INVARIANT
    v0 ∈ NAT ∧
    v2 ∈ BOOL ∧
    v4 ∈ INT
…
END
```

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

‘missing links’ (the gluing invariant is too weak)

```
MACHINE
    MA
ABSTRACT_VARIABLES
    v1
INVARIANT
    v1 ∈ BOOL
INITIALISATION
    v1 := TRUE
OPERATIONS
    SET(vv) ≙
        PRE    vv  ∈  BOOL
THEN
            v1 := vv
        END;
    vv ⟵ GET ≙
        BEGIN  vv := v1  END
END
```

```
IMPLEMENTATION
    MA_i
REFINES
    MA
IMPORTS
    MB
OPERATIONS
    SET(vv) ≙
        BEGIN        BSET(vv)
END;
    vv ⟵ GET ≙
        BEGIN   vv  ⟵  BGET
END
END
```

the intended gluing,
v1 = v2
has been omitted …

```
MACHINE
    MB
ABSTRACT_VARIABLES
    v2
INVARIANT
    v2 ∈ BOOL
INITIALISATION
    v2 := TRUE
OPERATIONS
    BSET(vv) ≙
        PRE    vv  ∈  BOOL
THEN
            v2 := vv
        END;
    vv ⟵ BGET ≙
        BEGIN  vv := v2  END
END
```

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

‘missing links’ may lead to failed proof obligations

- the intended gluing (**v1 = v2**) is missing from the implementation of MA
  - if these variables had the same name, this gluing would have been implicit
- the problem is detected (here) by the PO for its implemented operations

  e.g. proof of the GET operation leads to:

  $v1 \in BOOL \quad \wedge$

  $v2\$1 \in BOOL$

  $\Rightarrow$

  $v1 = v2\$1$ ***false PO!***

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

‘missing links’ and proof obligations - revisited

– such missing links are not *always* detected by false proof obligations …

```
MACHINE
    MC
ABSTRACT_VARIABLES
    vc
INVARIANT
    vc ∈ BOOL
INITIALISATION
    vc := TRUE
…
END
```

```
IMPLEMENTATION
    MC_i
REFINES
    MC
CONCRETE_VARIABLES
    vi
INVARIANT
    vi ∈ BOOL
INITIALISATION
    vi := FALSE
…
END
```

the intended gluing invariant

vc = vi

has been omitted …

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

proof obligations cannot verify *unstated* intentions

– when the intended gluing invariant (vc = vi) is missing …

e.g. the PO for initialisation is:

$( )$

$\Rightarrow$

$[\ vi := FALSE\ ]\ \neg\ (\ [\ vc := TRUE\ ]\ \neg\ (\ vi \in BOOL\ )\ )$

that is,

$( )$

$\Rightarrow$

$[\ vi := FALSE\ ]\ \neg\ (\ \neg\ (\ vi \in BOOL\ )\ )$

that is,

$( )$

$\Rightarrow$

$FALSE \in BOOL$ **true PO!**

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

gluing invariants – some conclusions

- WARNING
  - a gluing invariant that is too weak may corrupt the B model with no detection by the proof obligations!
- advice
  - make sure that ***all*** intended properties are specified
  - use output operation parameters or concrete variables
  - a theorical solution to detect this problem arises

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

constructing correct loops

- objective
  - to suggest a systematic approach to the construction of **WHILE** loops, and in particular their associated *INVARIANT* and *VARIANT*, in order to facilitate the required proofs of correctness
- recall
  - a **WHILE** construct may only be used to *implement* an abstract operation (so its overall 'correctness' is formally *specified* by that abstraction)
  - the loop *INVARIANT* describes the properties that must be *established* when entering the loop, and that must be *preserved* by each iteration
  - the *VARIANT* of a loop must be a natural number strictly *decreasing* at each iteration, so as to ensure that the number of iterations is *finite*

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

the role of loop invariants

- building an *invariant* for a loop is the key to proving its correctness
- analogy between a loop invariant and an abstract machine invariant

```
MACHINE
    M
VARIABLES
    X
INVARIANT
    I
INITIALISATION
    S_0
OPERATIONS …
    rr ⟵ op ≙ S ;
    …
END
```

```
OPERATIONS …
    rr ⟵ op ≙
        VAR X IN S_0;
            WHILE P DO
                S
            INVARIANT
                I
            VARIANT
                V
            END; rr := …
        END;
    …
```

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

proof of correctness for loops

[WHILE *P* DO *S* INVARIANT *I* VARIANT *V* END] *R*

- the WHILE substitution can be split into 5 separate proof obligations
  - the loop invariant *I* holds on entry to the loop (PO1)
  - the body of the loop *S* preserves the invariant (PO2)
  - the loop variant *V* is a natural number (PO3)
  - the variant strictly decreases on each iteration (PO4)
  - the desired result *R* holds on exit from the loop (PO5)

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

formal correctness of a WHILE loop

- let X be the names of variables modified within the loop body S, and let n be a local variable (that is not otherwise used)

- correctness of the WHILE substitution is defined by conjunction:

  $[\ S_0\ ;\ \text{WHILE}\ \ P\ \ \text{DO}\ \ S\ \ \text{INVARIANT}\ \ I\ \ \text{VARIANT}\ \ V\ \ \text{END}\ ]\ R \quad \Leftrightarrow$

  $$\begin{array}{ll}
  [\ S_0\ ]\ I \quad \wedge & \text{(PO1)} \\
  \forall\ X \cdot (\ I \wedge P \ \Rightarrow\ [\ S\ ]\ I\ ) \ \wedge & \text{(PO2)} \\
  \forall\ X \cdot (\ I \ \Rightarrow\ V \in \mathbb{N}\ ) \ \wedge & \text{(PO3)} \\
  \forall\ X \cdot (\ I \wedge P \ \Rightarrow\ [\ n := V\ ;\ S\ ]\ (V < n)\ ) \ \wedge & \text{(PO4)} \\
  \forall\ X \cdot (\ I \wedge \neg\ P \ \Rightarrow\ R\ ) & \text{(PO5)}
  \end{array}$$

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

constructing correct loops - abstraction

- at the abstract level
  - a specification … that will be implemented by a

```
MACHINE
    MTAB
CONCRETE_VARIABLES
    Tab
INVARIANT
    Tab ∈ 1..n → NAT
OPERATIONS …
    m ⟵ maxTab ≙
      BEGIN  m := max ( ran (Tab) )  END
END
```

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

constructing correct loops - implementation

- building the loop ***invariant*** (difficult)
  - specify the local variables properties (loop index)
  - **generalize** the abstraction properties introducing local variables
  - add every property needed for proving inside the loop (to prove the preconditions of called operations, to prove well-defineness, …)
  - check that the invariant holds at the loop entrance
  - check that if you replace the loop by $X:(I \wedge \neg P)$ the refinement is correct
  - examine the PO and make the adjustments to prove then
- building the loop ***variant*** (easy)
  - a positive expression using the variables modified in the loop body, strictly decreasing after each iteration

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0

constructing correct loops – completed example

```
IMPLEMENTATION
    MTAB_i
REFINES
    MTAB
OPERATIONS …
    m ⟵ maxTab ≙
        VAR i, lm IN
            i := 1;  lm := Tab(i);
            WHILE i < n DO
                i := i + 1;
                IF Tab(i) > lm THEN lm := Tab(i) END

        INVARIANT
            i ∈ 1..n ∧ lm ∈ NAT ∧
            lm = max (ran((1..i) ◁ Tab))
        VARIANT
            n - i
            END;  m := lm
        END
END
```

Copyright © 2022 CLEARSY
Licensed under the Creative Commons Attribution 4.0 Unported License: Creative Commons — Attribution 4.0 International — CC BY 4.0