Software Engineering

Lecture 04: The B Specification Method

Peter Thiemann

University of Freiburg, Germany

SS 2014

The B specification method

- B-Method: formal approach to specification and development of software systems
- Developed by Jean-Raymond Abrial, late 1980es
- Definitive reference: The B-Book, Cambridge University Press
- Supports all phases of software development
- Emphasis on simplicity
- Amenable to formal verification
- Tool support: Atelier-B, B-Toolkit
- Industrial use
- Syntax `http://www.stups.uni-duesseldorf.de/ProB/index.php5/Summary_of_B_Syntax`

Abstract Machines

Central concept: Abstract Machine

Example: The Ticket Dispenser

Ticket Dispenser in B

Abstract Machine Notation (AMN)

```
MACHINE Ticket
VARIABLES serve, next
INVARIANT serve : NAT & next : NAT & serve <= next
INITIALISATION serve, next := 0, 0
OPERATIONS
  ss <-- serve_next =
    PRE  serve < next
    THEN ss, serve := serve + 1, serve + 1
    END ;
  tt <-- take_ticket =
    PRE  true
    THEN tt, next := next, next + 1
    END
END
```

MACHINE, VARIABLES, INVARIANT

`MACHINE` *name*

- uniquely names a machine in a project

`VARIABLES` *name*, ...

- components of local *machine state space*
- all distinct names

`INVARIANT` *formula*

Conjunction of

- type of each variable, e.g., `serve : NAT`
- relations between variables, e.g., `serve <= next`

OPERATIONS

List of operation definitions

```
output, ... <-- name (input, ...) =
  PRE  precondition
  THEN statement
  END
```

- *name*: name of operation
- *input*, *outputs*: names of input and output parameters
- PRE *precondition*
  - Formula that must be true to invoke
  - May be dropped if `true`
- THEN *statement*: *body* of the operation
  - *Must* specify each output variable
  - *May* update the machine state

Statement / Assignment

Simple Assignment

```
name := expression
```

Multiple Assignment

```
name, ... := expression, ...
```

- all distinct names on left hand side
- simultaneous assignment — evaluate all right hand sides, then assign to left hand sides all at once

INITIALISATION

`INITIALISATION` *statement*

- defines a set of possible initial states
- all variables of the machine state must be assigned

Sets and Logic

Sets

- B builds on *typed* set theory
- Standard mathematical notation for set operations is ok, but we use the syntax of the tools
- Predefined sets:
  - `BOOL = { TRUE, FALSE }`
  - `INT`, `NAT`, `NAT1` machine integers and natural numbers (without 0)
  - `STRING` with elements of the form `"string content"`
- Types of variables are defined by predicates

`v:S`	the value of `v` is an element of set `S`
`v<:S`	the value of `v` is a subset of set `S`

Set Formation

SETS *declaration*; ...

- another MACHINE clause
- declaration can be
  - *set-name*: set with unspecified elements
  - *set-name* = { *element-name*, ...}: set with named elements
- example

  ```
  SETS COLOR = {red, green, blue}; KEY; PERSON
  ```

Set Expressions (Part I)

Excerpt

If S and T are sets, then so are . . .

{}, {E}, {E,...}	empty set, singleton set, set enumeration
{x|P}	comprehension (set of all x such that P holds)
S\/T, S/\T, S-T	set union, set intersection, set difference
S*T	Cartesian product (set of all pairs (s,t) where s:S and t:T)
POW(S), POW1(S)	power set, set of non-empty subsets
union(S)	generalised union over sets of sets
inter(S)	generalised intersection over sets of sets

Set Expressions (Part II)

Excerpt

Properties of sets

`E:S`, `E/:S`	element of, not element of
`S<:T`, `S/<:T`	subset of, not subset of
`S<<:T`, `S/<<:T`	strict subset of, not strict subset of
`card(S)`	cardinality

Types for Set Expressions

- Every type in B is a set, but not vice versa
- Types ensure consistency and the absence of paradoxa
- Types are generated by the grammar

$$A, B ::= \mathbb{N} \mid \mathtt{M} \mid A \times B \mid \mathbb{P}(A) \mid \textsc{Prop}$$

  - `M` is an abstract set defined in a `SETS` clause
  - $\textsc{Prop}$ is the type of propositions (outcomes of predicates)
- Write $E :: A$ to indicate that set expression E has type A

Rules for Typed Set Expressions (Part I)

$$1 :: \mathbb{N} \qquad \mathtt{NAT} :: \mathbb{P}(\mathbb{N}) \qquad \frac{\mathtt{SETS\ M} \ldots}{\mathtt{M} :: \mathbb{P}(\mathtt{M})} \qquad \frac{\mathtt{SETS\ M} = \{x_1, \ldots, x_n\}}{x_i :: \mathtt{M}}$$

$$\{\} :: \mathbb{P}(A) \qquad \frac{E_i :: A}{\{E_1, \ldots\} :: \mathbb{P}(A)} \qquad \frac{[x :: A]\ P :: \textsc{Prop}}{\{x \mid P\} :: \mathbb{P}(A)}$$

$$\frac{E :: A \qquad S :: \mathbb{P}(A)}{E : S :: \textsc{Prop}} \qquad \frac{S :: \mathbb{P}(A) \qquad T :: \mathbb{P}(A)}{S <: T :: \textsc{Prop}} \qquad \frac{S :: \mathbb{P}(A)}{\mathtt{card}(S) :: \mathbb{N}}$$

Rules for Typed Set Expressions (Part II)

$$\frac{S :: \mathbb{P}(A) \qquad T :: \mathbb{P}(A)}{S \cup T :: \mathbb{P}(A) \qquad S \cap T :: \mathbb{P}(A) \qquad S \setminus T :: \mathbb{P}(A)}$$

$$\frac{S :: \mathbb{P}(A) \qquad T :: \mathbb{P}(B)}{S * T :: \mathbb{P}(A \times B)} \qquad \frac{S :: \mathbb{P}(A)}{\mathtt{POW}(S) :: \mathbb{P}(\mathbb{P}(A))} \qquad \frac{S :: \mathbb{P}(A)}{\mathtt{POW1}(S) :: \mathbb{P}(\mathbb{P}(A))}$$

$$\frac{S :: \mathbb{P}(\mathbb{P}(A))}{\mathtt{union}(S) :: \mathbb{P}(A) \qquad \mathtt{inter}(S) :: \mathbb{P}(A)}$$

Examples

- $\{1,2,3\} :: \mathbb{P}(\mathbb{N})$
  because $1,2,3 :: \mathbb{N}$
- $\{1,2,\{\}\}$
  not well typed because $1,2 :: \mathbb{N}$ but $\{\} :: \mathbb{P}(A)$
- $1 : \{\}$
  well typed expression because $\{\} :: \mathbb{P}(\mathbb{N})$ / wrong proposition
- $\mathtt{NAT} \cup \{\} :: \mathbb{P}(\mathbb{N})$
- $\mathtt{NAT} \cup \mathtt{M}$
  not well typed because $\mathtt{NAT} :: \mathbb{P}(\mathbb{N})$ but $\mathtt{M} :: \mathbb{P}(\mathtt{M})$
- $\mathtt{NAT} \cup S * T$
  not well typed
- $\mathtt{union}(\mathtt{POW}(\{1,2,3\})) :: \mathbb{P}(\mathbb{N})$

First-Order Predicate Logic

- Atoms are expressions of type PROP
- Standard connectives

`P & Q`	conjunction
`P or Q`	disjunction
`P => Q`	implication
`P <=> Q`	equivalence
`not P`	negation
`!(x).(P => Q)`	universal quantification
`#(x).(P & Q)`	existential quantification

- In quantification, predicate `P` must fix the type of `x`
- Example

```
!(m).(m:NAT => #(n).(n:NAT & m < n))
```

Weakest Preconditions

State Space

- State space of a B machine = type of its variables restricted by invariant I
- Specification of operation = relation on state space
- Questions
  1. Is an operation executable?
  2. Does an operation preserve the invariant?

State Space

- State space of a B machine = type of its variables restricted by invariant I
- Specification of operation = relation on state space
- Questions
  1. Is an operation executable?
  2. Does an operation preserve the invariant?
- Formalized for operation `PRE` P `THEN` S `END`
  1. Executable: I `&` P
  2. Preservation: if executable, does I hold after S?

State Space

- State space of a B machine = type of its variables restricted by invariant I
- Specification of operation = relation on state space
- Questions
  1. Is an operation executable?
  2. Does an operation preserve the invariant?
- Formalized for operation `PRE` P `THEN` S `END`
  1. Executable: I `&` P
  2. Preservation: if executable, does I hold after S?
- Tool: *Weakest Precondition* (WP) $[S]Q$ (a predicate)
  - If $[S]Q$ holds before executing S, then Q holds afterwards
  - For all R that hold before S and guarantee that Q holds afterwards, R `=>` $[S]Q$

State Space

- State space of a B machine = type of its variables restricted by invariant `I`
- Specification of operation = relation on state space
- Questions
  1. Is an operation executable?
  2. Does an operation preserve the invariant?
- Formalized for operation `PRE P THEN S END`
  1. Executable: `I & P`
  2. Preservation: if executable, does `I` hold after `S`?
- Tool: *Weakest Precondition* (WP) `[S]Q` (a predicate)
  - If `[S]Q` holds before executing `S`, then `Q` holds afterwards
  - For all `R` that hold before `S` and guarantee that `Q` holds afterwards, `R => [S]Q`
- WP can be calculated for each statement of the AMN

Example

```
VARIABLES x, y
INVARIANT x:{0,1,2} & y:{0,1,2}
OPERATIONS
  f =
    y := max { 0, y - x }
END
```

Weakest precondition

```
    [ y := max { 0, y - x } ] (y > 0)
<=>
       (y = 1) & (x = 0)
    or (y = 2) & (x = 0)
    or (y = 2) & (x = 1)
```

Calculation of the Weakest Precondition

WP for Assignment

$$[x := E]P = P[E/x]$$

Example

```
    [ y := max { 0, y - x } ] (y > 0)
<=>
    (max { 0, y - x } > 0)
<=>
    (y - x > 0)
<=>
       (y = 1) & (x = 0)
    or (y = 2) & (x = 0)
    or (y = 2) & (x = 1)
```

Calculation of the Weakest Precondition

WP for skip

$$[\texttt{skip}]P = P$$

The `skip` statement has no effect on the state.

Calculation of the Weakest Precondition

WP for conditional

- Syntax: `IF` E `THEN` S `ELSE` T `END`
  for statements S and T
- Weakest precondition

$$[\texttt{IF } E \texttt{ THEN } S \texttt{ ELSE } T \texttt{ END}]P = (E\&[S]P) \texttt{ or } ((\texttt{not } E)\&[T]P)$$

Example

```
    [IF x<5 THEN x := x+4 ELSE x := x-3 END] (x < 7)
<=>
    (x < 5) & [x := x+4] (x < 7)
 or not (x < 5) & [x := x-3] (x < 7)
<=>
    (x < 5) & (x+4 < 7)
 or (x >= 5) & (x-3 < 7)
<=>
    (x < 3)
 or (x >= 5) & (x < 10)
```

Machine Consistency

INVARIANT and INITIALISATION

Objectives

1. The state space must not be empty
2. Initialization must be successful

`INVARIANT` I

State space is non-empty if `#(v).(`I`)`

`INITIALISATION` T

Success if $[T]I$

INVARIANT and INITIALISATION

Objectives

1. The state space must not be empty
2. Initialization must be successful

`INVARIANT` I

State space is non-empty if `#(v).(`I`)`

`INITIALISATION` T

Success if `[`T`]`I

Example: Ticket Dispenser

1. For `serve = 0` and `next = 0`, `serve <= next` holds
2. `[serve, next := 0, 0]`$I =$ `0:NAT & 0:NAT & 0<=0`

Proof Obligation for Operations

Consider

- `INVARIANT` I
- operation `PRE` P `THEN` S `END`

Consistent if

I `&` P `=>` `[`S`]`I

Proof Obligation for Operations

Consider

- `INVARIANT` I
- operation `PRE` P `THEN` S `END`

Consistent if

```
I & P => [S]I
```

Example: Ticket Dispenser `serve_next`

```
(serve:NAT & next:NAT & serve <= next) & (serve < next) =>
[serve := serve + 1] (serve:NAT & next:NAT & serve <= next)
<=>
(serve:NAT & next:NAT & serve < next) =>
(serve:NAT & next:NAT & serve + 1 <= next)
```

Relations

Printer Permissions

```
MACHINE Access
SETS USER; PRINTER; OPTION; PERMISSION = { ok, noaccess }
CONSTANTS options
PROPERTIES
  options : PRINTER <-> OPTION &
  dom( options ) = PRINTER & ran( options ) = OPTION
VARIABLES access
INVARIANT access : USER <-> PRINTER
INITIALISATION access := {}
OPERATIONS
  add (uu, pp) =
    PRE  uu:USER & pp:PRINTER
    THEN access := access \/ { uu |-> pp }
    END ;
  ...
```

New Machine Clauses

`CONSTANTS` *name*, ...

- *name* is a fixed, but unknown value
- Type determined by `PROPERTIES`

`PROPERTIES` *formula*

- Describes conditions that must hold for `SETS` and `CONSTANTS`
- *Must* specify the types of the constants
- *Must not* refer to `VARIABLES`

About clauses

- Clauses must appear in the same order as in the example!
- No forward references allowed

Relational Operations

- Binary relation between S and T
  $S\ \texttt{<->}\ T = \texttt{POW}\ \texttt{(}S\texttt{*}T\texttt{)}$
- Elements of a relation $R\ \texttt{:}\ S\ \texttt{<->}\ T$ are pairs, written as $uu\ \texttt{|->}\ pp$, where $uu\texttt{:}S\ \texttt{\&}\ pp\texttt{:}T$
- Predefined symbols for domain and range of a relation

  $\texttt{dom}\ \texttt{(}R\texttt{)} = \{\texttt{s | s:}S\ \texttt{\& \#(t).(t:}T\ \texttt{\& s |-> t :}R\texttt{)}\ \}$

  $\texttt{ran}\ \texttt{(}R\texttt{)} = \{\texttt{t | t:}S\ \texttt{\& \#(s).(s:}S\ \texttt{\& s |-> t :}R\texttt{)}\ \}$

Relational Operations

- Binary relation between S and T
  $S\ \texttt{<->}\ T = \texttt{POW}\ (S\texttt{*}T)$
- Elements of a relation $R\ \texttt{:}\ S\ \texttt{<->}\ T$ are pairs,
  written as $uu\ \texttt{|->}\ pp$, where $uu\texttt{:}S\ \texttt{\&}\ pp\texttt{:}T$
- Predefined symbols for domain and range of a relation

  $\texttt{dom}\ (R) = \{\texttt{s}\ \texttt{|}\ \texttt{s:}S\ \texttt{\&}\ \texttt{\#(t).(t:}T\ \texttt{\&}\ \texttt{s}\ \texttt{|->}\ \texttt{t}\ \texttt{:}R\texttt{)}\ \}$

  $\texttt{ran}\ (R) = \{\texttt{t}\ \texttt{|}\ \texttt{t:}S\ \texttt{\&}\ \texttt{\#(s).(s:}S\ \texttt{\&}\ \texttt{s}\ \texttt{|->}\ \texttt{t}\ \texttt{:}R\texttt{)}\ \}$
- Example:

```
PRINTER = {PL, PLDUPLEX, PLCOLOR}
options = { PL |-> ok, PLCOLOR |-> noaccess }
dom (options) = {PL, PLCOLOR}
ran (options) = {ok, noaccess}
```

Printer Permissions (Cont'd)

```
MACHINE Access ...
OPERATIONS ...
  ban (uu) =
    PRE  uu:USER
    THEN access := { uu } <<| access
    END ;
  nn <-- printnumquery (pp) =
    PRE  pp:PRINTER
    THEN nn := card (access |> { pp })
    END ;
```

Relational Operations II

Domain and range restriction

Let `R:S<->T`

Domain restriction: Remove elements from `dom (R)`

- Keep domain elements in `U`:
  `U <| R = { s |-> t | (s |-> t):R & s:U }`
- Drop domain elements in `U` (anti-restriction, subtraction):
  `U <<| R = { s |-> t | (s |-> t):R & s/:U }`

Range restriction: Remove elements from `ran (R)`

- Keep range elements in `U`:
  `R |> U = { s |-> t | (s |-> t):R & t:U }`
- Drop range elements in `U`:
  `R |>> U = { s |-> t | (s |-> t):R & t/:U }`

Relational Operations III

Further Relational Operations

`id(S)`	identity relation
`R-`	inverse relation
`R[U]`	relational image
`(R1;R2)`	relational composition
`R1<+R2`	relational overriding

Relational Operations III

Further Relational Operations

`id(S)`	identity relation
`R-`	inverse relation
`R[U]`	relational image
`(R1;R2)`	relational composition
`R1<+R2`	relational overriding

Overriding . . .

- `R1<+R2` means `R2` overrides `R1`
- Union of `R1` and `R2`, but in the intersection of `dom (R1)` and `dom (R2)`, the elements of `R2` take precedence
- `R1<+R2` = `(dom (R2) <<| R1) \/ R2`

Functions

Functions

- In B, a function is an unambiguous relation (i.e., a set of pairs)
- Shorthand notation to indicate properties of functions

`S+->T`	partial function	`S-->T`	total function
`S+->>T`	partial surjection	`S-->>T`	total surjection
`S>+>T`	partial injection	`S>->T`	total injection
`S>+>>T`	partial bijection	`S>->>T`	total bijection

- Using functions
  - `f (E)` function application
  - `%x.(P|E)` lambda abstraction, `P` gives type of `x`

Example: Reading Books / Declarations

```
MACHINE Reading
SETS READER; BOOK; COPY; RESPONSE = { yes, no }
CONSTANTS  copyof
PROPERTIES copyof : COPY -->> BOOK
VARIABLES  hasread, reading
INVARIANT
  hasread : READER <-> BOOK &
  reading : READER >+> COPY &
  (reading ; copyof) /\ hasread = {}
INITIALISATION
  hasread := {} || reading = {}
```

Example: Reading Books / Operations (Excerpt)

OPERATIONS (excerpt)

```
start (rr, cc) =
  PRE
    rr:READER & cc:COPY & copyof (cc)/:hasread(rr) &
    rr/:dom (reading) & cc/:ran (reading)
  THEN
    reading := reading \/ { rr |-> cc }
  END
;
bb <-- currentbook (rr) =
  PRE
    rr:READER & rr:dom (reading)
  THEN
    bb := copyof (reading (rr))
  END
```

Sequences and Arrays

Sequences

- A sequence over set S is a *total* function from an initial segment of `NAT1` to S
- `seq (S) = (1..N --> S)`, where `N:NAT`
- Notation for manipulating sequences:
  formation, concatenation, first, last, etc

Arrays

- An array over set S is a *partial* function from an initial segment of `NAT1` to S
- `(1..N +-> S)`, where `N:NAT`
- Notation for updating arrays
  $\boxed{\texttt{a (i) := E}} = \boxed{\texttt{a := a <+ \{ i |-> E \}}}$

Nondeterminism

Nondeterminism in Specifications

- Up to now: high-level programming with sets
  - deterministic machines
  - abstraction from particular data structures
  - abstraction from realization of operations
- Further abstraction
  - specification may allow a range of acceptable behaviors
  - specification describes possible choices
  - subsequent refinement narrows down towards an implementation
- This section
  - AMN operations that exhibit nondeterminism

Example: Jukebox / Declarations

```
MACHINE Jukebox
SETS TRACK
CONSTANTS limit
PROPERTIES limit:NAT1
VARIABLES credit, playset
INVARIANT credit:NAT & credit<=limit & playset<:TRACK
INITIALISATION credit, playset := 0, {}
```

```
OPERATIONS
  pay (cc) =
    PRE cc:NAT1
    THEN credit := min ( {credit + cc, limit}) END ;
```

Example: Jukebox / Operations (excerpt)

OPERATIONS

```
tt <-- play =
  PRE  playset /= {}
  THEN ANY tr WHERE tr:playset
       THEN tt := tr || playset := playset - {tr}
       END
  END
;
select (tt) =
  PRE  credit>0 & tt:TRACK
  THEN playset := playset \/ {tt}
    || CHOICE credit := credit - 1
       OR skip
       END
  END
```

ANY statement

`ANY` x `WHERE` Q `THEN` S `END`

- x fresh variable, only visible in Q and S
- Q predicate; type of x; other constraints
- S the body statement
- executes S with an arbitrary value for x fulfilling Q

Examples

Assume `total:NAT`

1. `ANY n WHERE n:NAT1 THEN total := total*n END`
2. `ANY t WHERE t:NAT & t<=total & 2*t>= total`
   `THEN total := t END`

ANY weakest precondition

$$\boxed{\texttt{[ANY }x\texttt{ WHERE }Q\texttt{ THEN }S\texttt{ END]}P = \texttt{!(}x\texttt{).(}Q\texttt{ => [}S\texttt{]}P\texttt{)}}$$

Examples

1. Example:

   ```
   [ANY n WHERE n:NAT1 THEN total := total*n END] (total > 1)
   = !(n).(n:NAT1 => [total := total*n] (total > 1))
   = !(n).(n:NAT1 => (total*n > 1))
   = (total > 1)
   ```

2. Example:

   ```
   [ANY t WHERE t:NAT & t<=total & 2*t>= total ...]  (total > 1)
   = !(t).(t:NAT & t<=total & 2*t>= total => [total := t](total > 1))
   = !(t).(t:NAT & t<=total & 2*t>= total => (t > 1))
   = (total > 2)
   ```

CHOICE statement

`CHOICE` S_1 `OR` S_2 `OR ... END`

- choice between unrelated statements S_1, S_2, ...

Example

Outcome of a driving test

```
CHOICE result := pass || licences := licences \/ {examinee}
OR result := fail
END
```

CHOICE weakest precondition

$$\boxed{[\texttt{CHOICE } S \texttt{ OR } T \texttt{ END}]P = [S]P \mathbin{\&} [T]P}$$

Example

Check that all licenced persons are old enough.

$$\left[\begin{array}{l}\texttt{CHOICE result := pass ||}\\ \qquad\quad\texttt{licences := licences \textbackslash/ \{examinee\}}\\ \texttt{OR result := fail}\\ \texttt{END}\end{array}\right](\texttt{licences<:ofAge})$$

$$= \begin{array}{l}\left[\begin{array}{l}\texttt{result := pass ||}\\ \texttt{licences := licences \textbackslash/ \{examinee\}}\end{array}\right](\texttt{licences<:ofAge})\\ \texttt{\& [result := fail] (licences<:ofAge)}\end{array}$$

$$= \begin{array}{l}\texttt{[licences := licences \textbackslash/ \{examinee\}](licences<:ofAge)}\\ \texttt{\& (licences<:ofAge)}\end{array}$$

$$= \texttt{(licences<:ofAge) \& examinee:ofAge}$$

Refinement

Refinement

- Refinement formalizes design decisions
- Refinement transforms specification towards implementation
- Refinement comes with proof obligations that relate the participating machines

Data refinement

- Formalizes change of data representation
- Usually from abstract to concrete
- Example: set $\rightarrow$ list or array

Refinement of nondeterminism

- Formalizes selection of particular behavior from a nondeterministic specification
- Refined operations are “more deterministic”

Example: Jukebox / Declarations

```
REFINEMENT JukeboxR
REFINES Jukebox
CONSTANTS freefreq
PROPERTIES freefreq:NAT1
VARIABLES creditr, playlist, free
INVARIANT
  creditr:NAT & creditr = credit &
  playlist:iseq(TRACK) & ran (playlist) = playset &
  free:0..freefreq
INITIALISATION
  creditr:=0 ; playlist:= [] ; free:=0
```

Example: Jukebox / Operations (excerpt)

```
select (tt) =
  BEGIN
    IF tt/:ran (playlist) THEN playlist := playlist <- tt END ;
    IF free = freefreq
    THEN CHOICE free := 0 OR creditr := creditr-1 END
    ELSE free := free+1 ; creditr := creditr-1
    END
  END
;
tt <-- play =
  PRE playlist /= []
  BEGIN tt := first (playlist) ;
        playlist := tail (playlist)
  END
```

Proof Obligation for Refinement

- INVARIANT of the REFINEMENT specifies the *linking invariant* between state spaces of original and refinement
- Let `INVARIANT I` in original and `INVARIANT IR` in refinement
- For `INITIALISATION T` in original and `INITIALISATION TR` in the refinement, it must hold that

```
[TR] (not [T] (not IR))
```

Proof Obligation for Refinement

- INVARIANT of the REFINEMENT specifies the *linking invariant* between state spaces of original and refinement
- Let `INVARIANT I` in original and `INVARIANT IR` in refinement
- For `INITIALISATION T` in original and `INITIALISATION TR` in the refinement, it must hold that

  ```
  [TR] (not [T] (not IR))
  ```
- For operation `PRE P THEN S END` in original and `PRE PR THEN SR END` in refinement, it must hold that

  ```
  I & IR & P => [SR] (not [S] (not IR))
  ```

Summary

- B — an industrial strength formal method that supports all phases of software development
- Approach:
  - start with high-level spec
  - apply refinement steps until level of implementation reached
  - (code generation tools exist)
- Each refinement step results in proof obligations that must be discharged
- Tools: ProB, Rodin
- Omitted from lecture
  - structuring: machine parameters, inclusion, extension, state and type export
  - implementation machines, loops, library machines
  - more notation . . .