Lecture 8

The B Method

Refining B Specifications - Data Refinement

Lecture Outline

- Refinement in B
- Arrays
- Sequencial Composition
- Local Variables
- Data Refinement
- Inheriting Static Information
- Including and Seeing Machines in Refinements

References

- [1] Abrial, J.-R., *The B Book - Assigning Programs to Meanings*, Cambridge University Press, 1996. (chapter 11)
- [2] Schneider, S., *The B-Method - An Introduction*, Palgrave Macmillan, Cornerstones of Computing series, 2001. (chapter 12)
- [3] Clearsy System Engineering, *AtelierB home page* `http://www.atelierb.eu/en/`
- [4] Clearsy System Engineering, *B Method home page* `http://www.methode-b.com/en/`

Refinement in B

- *Refinement* enables evolving an abstract system specification into an executable implementation
- B supports the concept of *stepwise refinement*
  - The implementation is developed gradually, through a series of *refinement steps*, each introducing a small number of design decisions
  - Intermediate steps between specification and implementation contain both abstract constructs and implementation details and are described by means of *refinement machines*
- Refinement topics
  - **Refinement of data**
  - Refinement of nondeterminism
  - Refinement proof obligations

Arrays

- An *array* is a named, indexed collection of values of a given type
- An array a of size n is a mapping (function) from $1\,..\,n$ to the type of values contained by the array
  - E.g. the array $a : 3 \;\; 15 \;\; 7 \;\; 19$ can be thought of as the function $a : 1\,..\,4 \;\; +\!\!-\!\!> \;\; \mathbb{N}$, $a = \{1 \;|\!\!-\!\!> 3,\; 2 \;|\!\!-\!\!> 15,\; 3 \;|\!\!-\!\!> 7,\; 4 \;|\!\!-\!\!> 19\}$
- Round brackets are used to provide access to elements of an array (as opposed to square brackets used in programming languages)
  - $a(i)$ = the i -th element of a, or the function a applied to i
- AMN allows assigning values to indexed elements of an array
  - $a(i) := e$, shorthand for $a := a \;<\!+\; \{i \;|\!\!-\!\!> e\}$

Sequencial Composition

- *Sequencial composition* allows one substitution to be executed after another
- Construct: $S;T$, where S and T are substitutions
- Definition (by post-condition establishment): $[S;T]P \Leftrightarrow [S]([T]P)$
- Any number of substitutions can be sequencially composed
- E.g.

$$
\begin{aligned}
[x := x+2; x := x+4](x > 9) &= [x := x+2](x+4 > 9) \\
&= x+2+4 > 9 \\
&= x > 3 \\
[t := x; x := y; y := t](x > 6 \wedge y < 4) &= [t := x; x := y](x > 6 \wedge t < 4) \\
&= [t := x](y > 6 \wedge t < 4) \\
&= y > 6 \wedge x < 4
\end{aligned}
$$

Local Variables

- In the context of sequencial composition, *local variables* are useful for acomplishing particular computations without imposing on the overall state space
- Construct: $\texttt{VAR}\ t\ \texttt{IN}\ S\ \texttt{END}$, where t is a variable and S is a substitution
- S should initialize t before using it
- E.g. $\texttt{VAR}\ t\ \texttt{IN}\ t := x; x := y; y := t\ \texttt{END}$
- Lists of several local variables can be described this way, if needed
- Definition (by post-condition establishment):
  $[\ \texttt{VAR}\ t\ \texttt{IN}\ S\ \texttt{END}\ ]\ P \Leftrightarrow \forall t \cdot [S]P$
- E.g. $[\ \texttt{VAR}\ t\ \texttt{IN}\ t := x; x := y; y := t\ \texttt{END}\ ]\ (x = A \wedge y = B)$

$$\begin{aligned} &= \forall t \cdot [t := x; x := y; y := t](x = A \wedge y = B) \\ &= \forall t \cdot (y = A \wedge x = B) \\ &= (y = A \wedge x = B) \end{aligned}$$

Data Refinement

- Abstract machines describe the state of the system in terms of abstract math structures (sets, relations, functions, sequences)
  - *Users* should only perceive the behavior of the system in terms of this specification
- System implementors though are concerned with data representation, since the abstract specification constructs are not directly supported by conventional programming languages
- Data representation is provided by means of a *refinement machine*, that captures:
  - the refined state
  - the connection between abstract and refined state by means of a linking invariant
  - the way that the initialisation and operations work with the new data representation
- A refinement machine has the same interface as the machine it refines (same operations with the same signatures)

Data Refinement (cont.)

- E.g.: specification of a football team
  - Keeps track of players in the field within a game
  - The team starts with the first 11 players
  - Operations are provided to replace a player on the field and query wether a particular team member is currently playing or not

MACHINE *Team*
SETS $ANSWER = \{in,\ out\}$
VARIABLES *team*
INVARIANT $team \subseteq 1\,..\,22 \wedge \mathbf{card}(team) = 11$
INITIALISATION $team := 1\,..\,11$
OPERATIONS

    $\mathbf{substitute}(pp, rr) \mathrel{\widehat{=}}$
    PRE $pp \in team \wedge rr \in 1\,..\,22 \wedge rr \notin team$
    THEN $team := (team - \{pp\}) \cup \{rr\}$
    END;

    $aa \leftarrow \mathbf{query}(pp) \mathrel{\widehat{=}}$
    PRE $pp \in 1\,..\,22$
    THEN
        IF $pp \in team$
        THEN $aa := in$
        ELSE $aa := out$
        END
    END
END

Data Refinement (cont.)

- Refinement for `Team`
  - Uses an array of size 11 to represent the state information
  - An abstract state may have several corresponding concrete states
  - Refined operations are assumed to work within their preconditions, as given in the abstract machine

REFINEMENT *TeamR1*
REFINES *Team*
VARIABLES *teamr*
INVARIANT $teamr \in 1 \,..\, 11 \rightarrowtail 1 \,..\, 22 \wedge$
$\qquad \mathrm{ran}(teamr) = team$
INITIALISATION $teamr := \mathrm{id}(1 \,..\, 11)$
OPERATIONS
$\quad$ substitute (pp , rr) $\widehat{=}$
$\quad$ **BEGIN**
$\qquad$ $\mathrm{teamr}(teamr^{-1}(pp)) := rr$
$\quad$ **END**;

$\quad$ $aa \leftarrow$ query (pp) $\widehat{=}$
$\quad$ **IF** $pp \in \mathrm{ran}(teamr)$
$\quad$ **THEN** $aa := in$
$\quad$ **ELSE** $aa := out$
$\quad$ **END**

END

Data Refinement (cont.)

- Alternative refinement for `Team`
  - Uses an array indexed by team members, rather than places in the team
  - Uses sequencial composition to define one the operations

```
REFINEMENT TeamR2
REFINES Team
VARIABLES teama
INVARIANT teama ∈ 1 .. 22 → ANSWER ∧
        team = (teama ⁻¹)[{in}]
INITIALISATION
   teama := (1 .. 11) × {in} ∪ (12 .. 22) × {out}
OPERATIONS
   substitute(pp,rr) ≙
   BEGIN
      teama(pp) := out;
      teama(rr) := in
   END;

   aa ← query(pp) ≙
   BEGIN
      aa := teama(pp)
   END

END
```

Inheriting Static Information

- A refinement has access to all the static information of the abstract machine it refines (parameters, sets, constants)
- A refinement may also access the sets and constants of any machine included in the abstract machine it refines
- A refinement has no default access to the information contained in machines seen/used by the abstract machine it refines; access to such information may be provided if a corresponding SEES clause is explicitely inserted in the refinement machine

Inheriting Static Information (cont.)

- E.g.: specification of a stack pile, used to track items of clothing to be ironed
  - Has a parameter, `limit`, which is the maximum number of items that can be piled
  - Has a given set, `ITEM`, defining the items that can be ironed

MACHINE $Ironing(limit)$
CONSTRAINTS $limit \in \mathbb{N}_1$
SETS $ITEM$
VARIABLES $pile$
INVARIANT $pile \in \mathbf{seq}(ITEM) \wedge \mathbf{size}(pile) \leq limit$
INITIALISATION $pile := []$
OPERATIONS

$\mathbf{put}(ii) \mathrel{\widehat{=}}$
PRE $ii \in ITEM \wedge \mathbf{size}(pile) < limit$
THEN $pile := pile \leftarrow ii$
END;

$ii \leftarrow \mathbf{take} \mathrel{\widehat{=}}$
PRE $pile \neq []$
THEN $pile := \mathbf{front}(pile) \mid\mid ii := \mathbf{last}(pile)$
END;

$bb \leftarrow \mathbf{query}(ii) \mathrel{\widehat{=}}$
PRE $ii \in ITEM$
THEN $bb := \mathbf{bool}(ii \in \mathbf{ran}(pile))$
END

END

Inheriting Static Information (cont.)

- Refinement machine for `Ironing`
  - Uses an array to store the items to be ironed
  - Uses a counter variable to indicate how much of the array corresponds to the abstract state

REFINEMENT $IroningR\ (limit)$
REFINES $Ironing$
VARIABLES $pilearr,\ counter$
INVARIANT $pilearr \in 1\,..\,limit \mapsto ITEM \wedge counter \in 0\,..\,limit \wedge$
$\quad 1\,..\,counter \lhd pilearr = pile$
INITIALISATION $pilearr := \emptyset \parallel counter := 0$
OPERATIONS
  put$(ii) \;\widehat{=}$
  BEGIN
   $counter := counter + 1;$
   pilearr$(counter) := ii$
  END;

  $ii \leftarrow$ **take** $\widehat{=}$
  BEGIN
   $ii := pilearr(counter);$
   $counter := counter$ - 1
  END;

  $bb \leftarrow$ **query**$(ii) \;\widehat{=}$
  IF $ii \in pilearr[1\,..\,counter]$
  THEN $bb :=$ **TRUE**
  ELSE $bb :=$ **FALSE**
  END

END

Including and Seeing Machines in Refinements

- Refinement machines may be incrementally developed using the structuring mechanisms provided by B
  - Abstract machines may be included/extended and operations promoted in refinements (similar to inclusion in other abstract machines)
  - Abstract machines can be seen by refinements
  - The `USES` clause cannot apear in a refinement (therefore just `INCLUDES`, `PROMOTES`, `EXTENDS` and `SEES`)
- The linking invariant of a refinement links all its state (native and included) with the corresponding abstract state
- The included state can be used in read mode in the description of operations of the refinement machine, but can only be modified through calls of operations of the included/extended machines (not by direct assignment)

Including and Seeing Machines in Refinements (cont.)

- E.g.: Specification of a ship control system in a port

MACHINE *Port*
SETS *SHIP*; *QUAI*
VARIABLES *waiting*, *docked*
INVARIANT $waiting \in \mathbf{iseq}(SHIP) \wedge docked \in QUAI \rightarrowtail\!\!\!\!\!\!+ \; SHIP \wedge$
$\quad \mathbf{ran}(waiting) \cap \mathbf{ran}(docked) = \emptyset$
INITIALISATION $waiting := [\,] \;||\; docked := \emptyset$
OPERATIONS

$\quad \mathbf{arrive}(ss) \mathrel{\widehat{=}}$
$\quad\quad$ **PRE** $ss \in SHIP \wedge ss \notin \mathbf{ran}(waiting) \wedge ss \notin \mathbf{ran}(docked)$
$\quad\quad$ **THEN** $waiting := waiting \leftarrow ss$
$\quad\quad$ **END**;

$\quad \mathbf{dock}(qq) \mathrel{\widehat{=}}$
$\quad\quad$ **PRE** $waiting \neq [\,] \wedge qq \in QUAI \wedge qq \notin \mathbf{dom}(docked)$
$\quad\quad$ **THEN** $waiting := \mathbf{tail}(waiting) \;||\; \mathbf{docked}(qq) := \mathbf{first}(waiting)$
$\quad\quad$ **END**;

$\quad qq \leftarrow \mathbf{leave}(ss) \mathrel{\widehat{=}}$
$\quad\quad$ **PRE** $ss \in SHIP \wedge ss \in \mathbf{ran}(docked)$
$\quad\quad$ **THEN** $docked := docked \rhd \{ss\} \;||\; qq := docked^{-1}(ss)$
$\quad\quad$ **END**;

$\quad nn \leftarrow \mathbf{numberwaiting} \mathrel{\widehat{=}}$
$\quad\quad$ **BEGIN**
$\quad\quad\quad nn := \mathbf{size}(waiting)$

$\quad\quad$ **END**

END

Including and Seeing Machines in Refinements (cont.)

- Machine `Port` will be refined by including two simpler machines
  - `List` - for handling the queue of waiting ships
  - `Map` - for handling the docking of ships to quays

MACHINE $List(ELEMENT)$
VARIABLES $list$
INVARIANT $list \in \mathrm{seq}(ELEMENT)$
INITIALISATION $list := []$
OPERATIONS
  $\mathrm{add}(ee) \mathrel{\widehat{=}}$
    PRE $ee \in ELEMENT$
    THEN $list := list \leftarrow ee$
    END;

  $ee \leftarrow \mathrm{take} \mathrel{\widehat{=}}$
    PRE $list \neq []$
    THEN $list := \mathrm{tail}(list) \mid\mid ee := \mathrm{front}(list)$
    END

END

MACHINE $Map(INDEX, ITEM)$
VARIABLES fun
INVARIANT $fun \in INDEX \nrightarrow ITEM$
INITIALISATION $fun := \emptyset$
OPERATIONS
  $\mathrm{insert}(ss1, ss2) \mathrel{\widehat{=}}$
    PRE $ss1 \in INDEX \wedge ss2 \in ITEM$
    THEN $\mathrm{fun}(ss1) := ss2$
    END;

  $\mathrm{remove}(ss1) \mathrel{\widehat{=}}$
    PRE $ss1 \in INDEX$
    THEN $fun := \{ss1\} \lhd fun$
    END;

  $ss2 \leftarrow \mathrm{query}(ss1) \mathrel{\widehat{=}}$
    PRE $ss1 \in \mathrm{dom}(fun)$
    THEN $ss2 := fun(ss1)$
    END

END

Including and Seeing Machines in Refinements (cont.)

- Refinement of `Port` with inclusion

```
REFINEMENT PortR
REFINES Port
INCLUDES List(SHIP), Map(SHIP, QUAI)
VARIABLES num
INVARIANT waiting = list ∧ docked ⁻¹ = fun ∧ num = size(waiting)
INITIALISATION num := 0
OPERATIONS
  arrive(ss) ≙
  BEGIN
    add(ss);
    num := num + 1
  END;

  dock(qq) ≙
  BEGIN
    VAR sh IN
      sh ← take;
      insert(sh,qq)
    END;
    num := num - 1
  END;

  qq ← leave(ss) ≙
  BEGIN
    qq ← query(ss);
    remove(ss)
  END;

  nn ← numberwaiting ≙
   nn := num

END
```