Designing a Routine by Deriving It from the MID

SFWR ENG 2B03

2003

Robert L. Baber

Designing from the MID

MID = Module Internal Design

A MID contains

- the specification of each routine and
- the main overall design decisions for each routine.

Use this information

- to your benefit and
- to the benefit of others.

Why Design from the MID?

Use the information in the MID *systematically*

- to save yourself work,
- to reduce or avoid errors and
- to save others time, frustration and annoyance.

Don’t start all over again and “design by intuition”.

Only hackers do it that way.

Engineers are not hackers.

Postcondition

To form the postcondition:

- start with the input/output relation from the routine semantics part of the MID *
- remove all apostrophes ' *after* variable names (but *not* those *before* variable names)
- combine the modified input/output relation and the state invariant * with $\wedge$ (logical and)
- the result is the postcondition

  * formulation with *concrete* state variables

Names Preceded by an Apostrophe '

Names preceded by an apostrophe '

- are *specification parameters* (constants), not program variables.
- represent values of the corresponding program variables immediately before calling the routine in question.
- are *not* to be removed from the postcondition (leave them in the postcondition).

Precondition

To form the precondition:

- combine the domain and the state invariant (with *concrete* state variables) with $\wedge$ (logical and)
- remove prefixed apostrophes, then
- for every identifier x prefixed with an apostrophe ' ('x) in the *postcondition*, append the term x='x to the *precondition* with $\wedge$
- the result is the precondition

Design Procedure

The aim of the design procedure is

- to transform the postcondition into the precondition (or into a condition implied by the precondition, i.e. into a weaker condition)
- by a series of steps, where
- each step corresponds to the application of a proof rule.

The statements corresponding to the proof rules become the routine being designed.

Proof Rule: Assignment Statement

$$V \Rightarrow P^{x}_{e}$$

$\Rightarrow$ [because $\{P^{x}_{e}\}$ x:=e $\{P\}$]

$$\{V\}\ x:=e\ \{P\}$$

where x is a variable name and e is an expression

Design strategy: find an x and an e such that substituting e for x in the postcondition leads to V (or a condition more like V and repeat)

Design Example: Stack Routine “Push”

V: size$\in$ Z $\wedge$ 0 $\leq$ size < MaxDepth $\wedge$ x$\in$ A

$\wedge$ ($\wedge$ i : i$\in$ Z $\wedge$ 0$\leq$i$\leq$'size-1 : stack[i]='stack[i])

$\wedge$ size = 'size

P: size$\in$ Z $\wedge$ 0 $\leq$ size $\leq$ MaxDepth

$\wedge$ ($\wedge$ i : i$\in$ Z $\wedge$ 0$\leq$i$\leq$'size-1 : stack[i]='stack[i])

$\wedge$ size = 'size+1 $\wedge$ stack['size] = x

Step 1: substitute size+1 for size in P

Step 2: substitute x for stack[size] in result of (1)

Design Example: Assignment Statements

Step 1: substitute size+1 for size in P

Step 2: substitute x for stack[size] in result of (1)

I.e.

$$V \Rightarrow [P^{size}_{size+1}]^{stack[size]}_{x}$$

Therefore, the program is:

stack[size] := x

size := size+1

Note the order of the statements!

Proof Rule: While Loop

$$\{V\}\ init\ \{I\} \wedge \{I \wedge B\}\ S\ \{I\} \wedge (I \wedge \neg B \Rightarrow P)$$

$$\Rightarrow$$

$$\{V\}\ init;\ \text{while } B \text{ do } S \text{ endwhile } \{P\}$$

To design a loop for V and P, determine

- a suitable loop invariant I
- init so that $\{V\}\ init\ \{I\}$
- B so that $I \wedge \neg B \Rightarrow P$
- S so that $\{I \wedge B\}\ S\ \{I\}$ (and progress toward $\neg B$)

Design Example: While Loop, Step 1

Step 1: determine a suitable loop invariant I

- initial situation is a special case of I
- final situation (P) is a special case of I
- I is a generalization of the initial and final situations

Approaches: generalize P to I

- so that I is easy to initialize
- by introducing a new variable if necessary

Design Example: While Loop, Step 1

V: $n \in Z \wedge 0 \leq n$

P: $n \in Z \wedge 0 \leq n \wedge sum = \sum_{j=1}^{n} X(j)$

Note: n may not be modified

Initialization must eliminate the $\sum$ series, but neither 1 nor n may be modified.

Therefore, a new variable must be introduced:

I: $n \in Z \wedge i \in Z \wedge 0 \leq i \leq n \wedge sum = \sum_{j=1}^{i} X(j)$

Design Example: While Loop, Step 2

Step 2: determine init so that {V} init {I}

V: $n \in Z \wedge 0 \leq n$

I: $n \in Z \wedge i \in Z \wedge 0 \leq i \leq n \wedge sum = \sum_{j=1}^{i} X(j)$

If i=0, the $\sum$ series is empty (and = 0). I.e.,

$V \Rightarrow [I^{i}_{0}]^{sum}_{0}$

Therefore, init is:

sum := 0; i := 0

Design Example: While Loop, Step 3

Step 3: determine B so that $I \wedge \neg B \Rightarrow P$

I: $n \in Z \wedge i \in Z \wedge 0 \leq i \leq n \wedge sum = \sum_{j=1}^{i} X(j)$

P: $n \in Z \wedge 0 \leq n \wedge sum = \sum_{j=1}^{n} X(j)$

If i=n (or if $i \geq n$), then I reduces to P:

$I \wedge i \geq n \Rightarrow P$

Therefore, a suitable choice for B is $\neg(i \geq n)$, i.e.

- $i < n$

Design Example: While Loop, Step 4

Step 4: determine S so that

- $\{I \wedge B\}\ S\ \{I\}$ and
- S makes progress toward $\neg B$, P, termination

$I \wedge B : n \in Z \wedge i \in Z \wedge 0 \leq i < n \wedge sum = \sum_{j=1}^{i} X(j)$

$I: n \in Z \wedge i \in Z \wedge 0 \leq i \leq n \wedge sum = \sum_{j=1}^{i} X(j)$

Increasing i by 1 makes progress toward $\neg B$. But

- $\{I^{i}_{i+1}\}$ i:=i+1 $\{I\}$

Design Example: While Loop, Step 4b

so we need Step 4b: determine S2 so that

- $\{I \wedge B\}\ S2\ \{I^{i}_{i+1}\}$

$I \wedge B$: $n \in Z \wedge i \in Z \wedge 0 \leq i < n \wedge sum = \sum_{j=1}^{i} X(j)$

I^{i}_{i+1}: $n \in Z \wedge i \in Z \wedge 0 \leq i+1 \leq n \wedge sum = \sum_{j=1}^{i+1} X(j)$

Note that

- $I \wedge B \Rightarrow [I^{i}_{i+1}]^{sum}_{sum+X(i+1)}$

Therefore, S is: sum:=sum+X(i+1); i:=i+1

Design Example: While Loop

Combining the above, the while loop with initialization becomes:

```
sum := 0; i := 0
while i<n do
sum:=sum+X(i+1); i:=i+1
endwhile
```

The obvious alternative body of the loop can also be derived in this way.

Proof Rule: If Statement

$\{V \wedge B\}\ S1\ \{P\} \wedge \{V \wedge \neg B\}\ S2\ \{P\}$

$\Rightarrow$

$\{V\}$ if B then S1 else S2 endif $\{P\}$

Alternate form for deriving a precondition:

$\{V1\}\ S1\ \{P\} \wedge \{V2\}\ S2\ \{P\}$

$\Rightarrow$

$\{V1 \wedge B \vee V2 \wedge \neg B\}$
if B then S1 else S2 endif $\{P\}$

Designing an If Statement

If you

- try to design a program segment S for precondition V and postcondition P,
- but find a program segment S1 with a stronger precondition ($V \wedge B$) instead,
- then embed S1 in an if statement and
- design S2 for $\{V \wedge \neg B\}$ S2 $\{P\}$.

Then: {V} if B then S1 else S2 endif {P}

Summary

- The precondition and the postcondition together with the proof rules suggest the structure of the program.
- Most expressions in a program can be derived algebraically from the precondition, the postcondition and the relevant proof rules.
- Don’t guess, derive mathematically. You will get it correct and quicker that way.
- “Let the symbols do the work.” (Dijkstra)

References

See list of references in 2B03 course outline

For a short introduction to designing programs from their specifications, see especially *Error Free Software: Know-How and Know-Why of Program Correctness*

Dijkstra: EWD 1041-7