Logic Specification

Logic specifications

- A formula of First Order Theory(FOT) is an expression involving variables, numeric constants, functions, predicates, and parentheses.
- logical connectives used : and, or, not, in, implies, and ≡ (logical equivalance)
- Result must be boolean
- May use quantifiers (e.g exists, for all)

Logic specifications

Examples of first-order theory (FOT) formulas:

1. $x > y$ **and** $y > z$ **implies** $x > z$
2. $x = y \equiv y = x$
3. **for all** x, y, z ($x > y$ **and** $y > z$ **implies** $x > z$)
4. $x + 1 < x - 1$
5. **for all** x (**exists** y ($y = x + z$))
6. $x > 3$ **or** $x < -6$

Logic specifications

- *free* variable: if variable is not quantified
- *bound* variable: the variable which is not free
- *closed* formula:
    - If all variables are quantified.
    - Is always either true or false
- *Closure* of a formula: Can be obtained by applying *for all* quantifier for all of its free variables
- In some cases, the truth of a formula depends on the domain chosen for its variables.

Example

Example – Saving addresses

// name must not be empty

// state must be valid

// zip must be 5 numeric digits

// street must not be empty

// city must not be empty

Rewriting to logical expression

name != “” ∧ state in stateList ∧ zip >= 00000 ∧ zip <= 99999 ∧ street != “” ∧ city != “”

Specifying complete programs

A *property*, or *requirement,* for P is specified as a formula of the type

$$\{Pre\ (i_1, i_2,\ldots, i_n)\ \}$$

$$P$$

$$\{Post\ (o_1, o_2,\ldots, o_m, i_1, i_2,\ldots, i_n)\}$$

Pre: precondition
Post: postcondition

Specifying complete programs

- PRE: FOT formula having $i_1,i_2,\ldots,i_n$ as free variables
- POST: FOT formula having $o_1,o_2,\ldots,o_m$, and possibly $i_1,i_2,\ldots,i_n$ as free variables
- PRE :Precondition of P
- POST :Post condition of P
- *The preceding formula is intended to mean that if PRE holds for the given input values before P's execution, then, after P finishes executing, POST must hold for the output and input values*

Examples

- Simple requirement of the division

$$\{\text{exists } z\ (i_1 = z * i_2)\}$$
$$P$$
$$\{o_1 = i_1/i_2\}$$

Examples(cont.)

- Stronger requirement of the division

$$\{i_1 > i_2\}$$

$$P$$

$$\{i_1 = i_2 * o_1 + o_2 \textbf{ and } o_2 >= 0 \textbf{ and } o_2 < i_2\}$$

- Imposes more constraints on output values less on input values
- A precondition {true} does not place any constraint on input values

Examples(cont.)

- Requires that P produce greater of i_1 and i_2

  {true}

  P

  {($o = i_1$ **and** $o>=i_2$) | | ($o = i_2$ **and** $o>=i_1$)}

- Program to compute sum of the input sequence

  {n > 0}

  P

  $$\{ O = \sum_{k=1}^{n} i_k \}$$

Exercise

- Write a Program specification to compute greatest common divisor
- Write a Program specification to produce reverse of the input sequence

Exercise : solution

- Program to compute greatest common divisor

  $\{i_1 > 0$ **and** $i_2 > 0\}$

  P

  $\{($**exists** $z_1, z_2\ (i_1 = o * z_1$ **and** $i_2 = o * z_2)$

  and not (**exists** h

  (**exists** $z_1, z_2\ (i_1 = h * z_1$ **and** $i_2 = h * z_2)$ **and** $h > o))$

- Program to produce reverse of the input sequence

  $\{n > 0\}$

  P

  $\{$ **for all** $i\ (1 <= i <= n)$ **implies** $(o_i = i_{n-i+1})\ \}$

Exercise

1. *Give a logic specification for a program that reads a sequence of n+1 values and checks whether the first value also appears in the next input n values*

2. *Give a logic specification for a program that first reads two words(i.e. two sequences of alphabetic characters, separated by a blank and terminated by '#'). The second word may be null; the first must not. Then, the program reads a sequence of other words, separated by blanks and terminated by '#', and rewrites the sequence, substituting all occurrences of the first word by the second.*

Specifying procedures

- To check whether element exists in table

  {n > 0} -- n is a constant value
  procedure search (table: in integer_array; n: in integer;
  element: in integer; found: out Boolean);
  {found $\equiv$ (exists i ($1 \leq i \leq n$ and table (i) = element))}

- To reverse the content of an array of integers

  {n > 0 }
  procedure reverse (a: in out integer_array; n: in integer);
  {for all i ($1 \leq i \leq n$) implies (a (i) = *old*_a (n - i +1))}

Specifying procedures(cont.)

- To sort a given list of integers

  {n > 0}

  procedure sort(a: in out integer_array; n: in integer);
  {sorted(a,n) },

  sorted(a,n) $\equiv$ (for all i(1 <= i < n) implies a(i) <= a(i+1)}

Specifying classes

- Defining properties of the state of program execution, rather than just I/O relations
- More important for OO languages
- *Invariant* predicates
  - Invariant defines a property that characterizes the object from its creation, throughout its lifetime
  - Must be preserved by the operations
- Example of invariant specifying an array implementing ADT SET

for all i, j ($1 \leq i \leq$ length **and** $1 \leq j \leq$ length **and** $i \neq j$)
implies $IMPL[i] \neq IMPL[j]$
(no duplicates are stored)

Specifying classes: precondition and postcondition with invariant

- Suppose an operation DELETE is defined to delete an element x from set. Then a precondition for DELETE could be,

  exists i (1 <= i <= length **and** IMPL[i] = x)

The post condition would be,

  for all i(1 <= i <= length **implies** IMPL[i] ≠ x) **and**

  for all i((1 <= i <= old_length **and** old_IMPL[i] ≠ x) **implies**

  exists j(1<=i<=length **and** IMPL[i] = old_IMPL[i])

Specifying classes

- In General, let us assume that INV is an invariant predicate for a class.
- for each operation op_i, the complete specification for the code implementing operation opi defined by the class, may be given as,

{INV and pre_i} program fragment for op_i{INV and $post_i$}

- A constructor operation cstr provided by a class,

  {true} program fragment for cstr {INV}

A case-study using logic specifications

- We outline the elevator example
- Elementary predicates
  - at (E, F, T)
    - E is at floor F at time T
  - start (E, F, T, up)
    - E left floor F at time T moving up
- Rules
  - (at (E, F, T) and on (EB, F_1, T) and $F_1 > F$) implies start (E, F, T, up)

States and events

- Elementary predicates are partitioned into
  - states, having non-null duration
    - standing(E, F, T1, T2)
  - events
    - instantaneous (caused state change at same time)
    - represented by predicates that hold only at a particular time instant
      - arrived (E, F, T)
- For simplicity, we assume
    - zero decision time
    - no simultaneous events

Events (1)

- arrival (E, F, T)
  - E in [1..n], F in [1..m], $T \geq t_0$, (t_0 initial time)
    - does not say if it will stop or will proceed, nor where it comes from
- departure(E, F, D, T)
  - E in [1..n], F in [1..m], D in {up, down}, $T \geq t_0$
- stop (E, F, T)
  - E in [1..n], F in [1.. m], $T \geq t_0$
    - specifies stop to serve an internal or external request

Events (2)

- new_list (E, L, T)
  - E in [1..n], L in [1.. m], $T \geq t_0$
    - L is the list of floors to visit associated with elevator (scheduling is performed by the control component of the system)
- call(F, D, T)
  - external call (with restriction for 1, N)
- request(E, F, T)
  - internal reservation

States

- moving (E, F, D, T1, T2)
- standing (E, F, T1, T2)
- list (E, L, T1, T2)
    - We implicitly assume that state predicates hold for any sub- interval (i.e., the rules that describe this are assumed to be automatically added)
        - Nothing prevents that it holds for larger interval

Rules relating events and states

R_1:
When E arrives at floor F, it continues to move if there is no request for service from F and the list is not empty.
If the floor to serve is higher, it moves upward; otherwise it moves downward.

arrival (E, F, T_a) and
list (E, L, T, T_a) and
first (L) > F
implies
departure (E, F, up, T_a)

A similar rule describes downward movement.

R2:
Upon arrival at F, E stops if F must be serviced (F appears as first of the list)

arrival (E, F, T_a) and
list (E, L, T, T_a) and
first (L) = F
implies
stop (E, F,T_a)

R3: E stops at F if it gets there with an empty list

arrival (E, F, T_a) and
list (E, empty, T, T_a)
implies
stop (E, F, T_a)

R4:
Assume that elevators have a fixed time to service a floor.
If the list is not empty at the end of such interval, the elevator leaves the floor immediately.

R5:
If the elevator has no floors to service, it stops until its list becomes nonempty.

R4:
Assume that elevators have a fixed time to service a floor.
If the list is not empty at the end of such interval, the elevator leaves the floor immediately.

$$
\begin{array}{l}
\textit{stop}\ (E, F, T_a)\ \textit{and} \\
\textit{list}\ (E, L, T, T_a + Dt_s)\ \textit{and} \\
\textit{first}\ (L) > F, \\
\qquad\qquad \textit{implies} \\
\textit{departure}\ (E, F, up, T_a + Dt_s)
\end{array}
$$

R5:
If the elevator has no floors to service, it stops until its list becomes nonempty.

$$\begin{array}{l} \textit{stop}\ (E, F, T_a)\ \textit{and list}\ (E, L, T_p, T)\ \textit{and} \\ T_p > T_a + Dt_s\ \textit{and list}\ (E, \textit{empty}, T_a + Dt_s, T_p)\ \textit{and} \\ \textit{first}\ (L) > F \\ \qquad\qquad \textit{implies} \\ \textit{departure}\ (E, F, \textit{up}, T_p) \end{array}$$

R6: Assume that the time to move from one floor to the next is known and fixed. The rule describes movement.

departure (E, F, up, T)
implies
arrival (E, F + 1, T + Dt)

R7: The event of stopping initiates standing for at least Dts.

stop (E, F, T)
implies
standing (E, F, T, T + Dt_s*)*

R8: At the end of the minimum stop interval Dts, E remains standing if there are no floors to service.

stop (E, F, T_s) and
list (E, empty, T_s + Dt_s, T)
implies
standing (E, F, T_s, T)

R9: Departure causes moving.

departure (E, F, D, T)
implies
moving (E, F, D, T, T + Dt)

R11: Effect of arrival of E at floor F

arrival (E, F, T_a) *and list* (E, L, T, T_a) *and*
$F = first (L)$ *and* $L_t = tail (L)$
implies
new_list (E, L_t, T_a)

R12: How list changes

new_list (E, L, T_1) *and not (new_list* (E, L, T_2) *and*
$T_1 < T_2 < T_3)$
implies
list (E, L, T_1, T_3)

Verifying specifications

- The system can be simulated by providing a state (set of facts) and using rules to make deductions

  standing (2, 3, 5, 7) elevator 2 at floor 3 at least from instant 5 to 7 (as a consequence of stop(2,3,5)

  list(2, empty, 5, 7)

  request(2, 8, 7)

  new_list(2, {8}, 7)

Verifying specifications

- Properties can be stated and proved via deductions

$$new\_list\ (E, L, T)\ and\ F \in L$$

$$implies$$

$$new\_list\ (E, L_1, T_1)\ and\ F \notin L_1\ and\ T_1 > T_2$$

(all requests are served eventually)