Rodin and Refinement

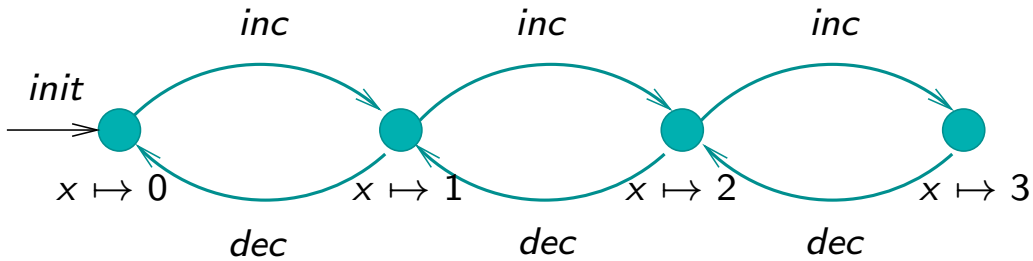
Deepak D'Souza

Department of Computer Science and Automation
Indian Institute of Science, Bangalore.

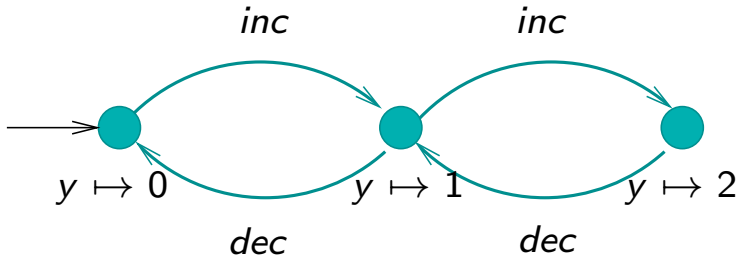
20 February 2024

Specifying ADTs logically or declaratively

- States
  - Variables: Eg. $\{x\}$, $\{y\}$, $\{len, beg, end\}$.
  - Invariants:
    Eg. $x \in \mathbb{N}$, $x \leq 3$.
    Eg. $0 \leq beg \leq len$.
- Operations (Eg. *inc*, *dec*, *enq*)
  - Parameters: Eg. *newval*
  - Guards: Eg. $x < 2$, $y < 2$, $newval \in \mathbb{N}$
  - Update (or Action): Eg. $x' = x + 1$, $y' = y - 1$



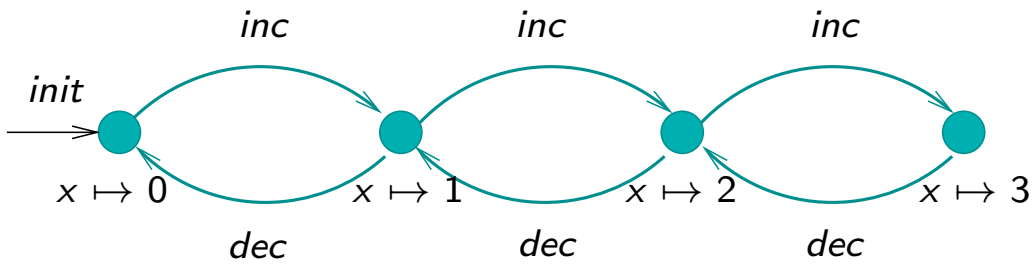
ADT 3counter



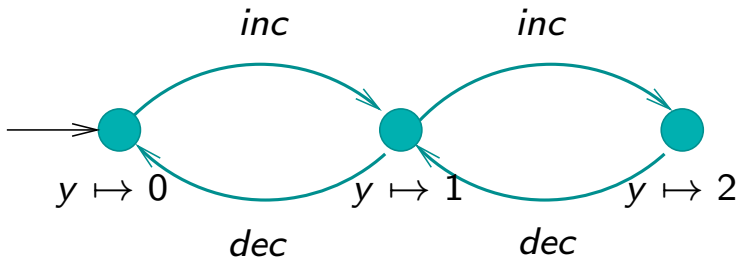
ADT 2counter

Specifying abstraction relations

- Abstraction relation (or gluing relation)
  - Constraint on joint state of abstract and concrete ADTs. Eg. $x = y$



ADT 3counter



ADT 2counter

Proof obligations

Operations are well-defined

- Actions restore invariants.
  Eg. The following formulas should be logically valid:
  For *init*:

$$(\underbrace{x' = 0}_{action} \Rightarrow (\underbrace{x' \in \mathbb{N} \land x' \leq 3}_{invariant}).$$

  For *inc*:

$$(\underbrace{x \in \mathbb{N} \land x \leq 3}_{invariant} \land \underbrace{x < 3}_{guard} \land \underbrace{x' = x + 1}_{action}) \Rightarrow (\underbrace{x' \in \mathbb{N} \land x' \leq 3}_{invariant}).$$

Proof obligations

Refinement conditions:

- Init (gluing relation should hold initially): Eg. The following formula should be valid:

$$(\underbrace{x' = 0 \wedge y' = 0}_{\textit{abs and conc actions}}) \Rightarrow \underbrace{x' = y'}_{\textit{gluing inv}} .$$

- Guard strengthening (conc guard stronger than abs guard): Eg. Following formula should be valid:

$$(\underbrace{x \in \mathbb{N} \wedge x \leq 3}_{\textit{abs invariant}} \wedge \underbrace{y \in \mathbb{N} \wedge y \leq 2}_{\textit{conc invariant}} \wedge \underbrace{x = y}_{\textit{gluing inv}} \wedge \underbrace{y < 2}_{\textit{conc guard}}) \Rightarrow \underbrace{x < 3}_{\textit{abs guard}} .$$

- Sim (gluing inv is restored): Eg. Following formula should be valid:

$$(\underbrace{x \in \mathbb{N} \wedge x \leq 3}_{\textit{abs invariant}} \wedge \underbrace{y \in \mathbb{N} \wedge y \leq 2}_{\textit{conc invariant}} \wedge \underbrace{x = y}_{\textit{gluing}} \wedge \underbrace{y < 2}_{\textit{conc guard}} \wedge \underbrace{x' = x + 1}_{\textit{abs action}} \wedge \underbrace{y' = y + 1}_{\textit{conc action}}$$

$$\Rightarrow \underbrace{x' = y'}_{\textit{gluing inv}} .$$

Rodin tool

- Provides an environment for developing a system design by succcessive refinement.
- Uses Event-B modelling language.
- Provides Features
  - Checking consistency of models.
    - Are expressions well-defined. For example if `x := y/z` then is `z` non-zero? As another example, if `x < y` then are both `x` and `y` of type integer?
    - Does the initialization event always result in a state satisfying the state invariants?
    - Does an event always restore the state invariants?
  - Checking refinement between models.
    - $\mathcal{B}$ refines $\mathcal{A}$ iff there exists a gluing relation by which $\mathcal{A}$ can simulate $\mathcal{B}$.
    - Generates proof obligations to check if one machine $\mathcal{B}$ refines another $\mathcal{A}$.

Demo in Rodin

- Counter model (counter.zip) demonstrating
  - Proof obligations generated by consistency checks
  - Proof obligations generated by refinement conditions
  - Using the Prover perspective to check proof obligations
- Byte counter (TwoByteCounter.zip)
- Queue (queue.zip)

Proof obligations generated by Rodin

```
CONTEXT ctx1

 CONSTANTS

  red
  green

 SETS

  COLOURS

 AXIOMS

   type: partition(COLOURS, {red}, {green})
   ... A ...
```

```
MACHINE counter2

REFINES counter

SEES ctx1

VARIABLES count2

INVARIANTS ...J...

EVENTS

INITIALIZATION ...T_init...

Event inc2
any param
when H_inc2
then ...T_inc2...

Event inc2
any param'
when H_inc2 then ...T_inc2...
```

Main proof obligations generated by Rodin

Here concrete invariant J includes gluing invariant ρ.

- Initialization

$$(A \wedge T_{init}) \implies J.$$

- Events (guard strengthening)

$$(A \wedge I \wedge J \wedge H) \implies G.$$

- Events (invariant preservation)

$$(A \wedge I \wedge J \wedge H \wedge U) \implies J[v'/v, w'/w].$$

Abstract Machine

Variables:	V
Invariants:	I
Event Op:	
Guard:	G
Action:	T

Concrete Machine

Variables:	W
Invariants:	J
Event Op:	
Guard:	H
Action:	U

Proof obligations generated by Rodin for `theorems`

- In Axioms (A_{thm}), where A_b is axioms appearing before A_{thm}:

$$A_b \implies A_{thm}.$$

- In event guards (H_{thm}), where H_b is guards appearing before H_{thm}:

$$(A \wedge I \wedge J \wedge H_b) \implies H_{thm}.$$

- In invariants (J_{thm}), where J_b is invariants appearing before J_{thm}:

$$(A \wedge I \wedge J_b) \implies J_{thm}.$$

Proof obligations for Z refinement

- Initialization

$$(A \wedge T_{init}) \implies J.$$

- Events (guard weakening)

$$(A \wedge I \wedge J \wedge G) \implies H.$$

- Events (invariant preservation)

$$(A \wedge I \wedge J \wedge G \wedge U) \implies J[v'/v, w'/w].$$

Assert these as `theorems`.

A C implementation of a queue

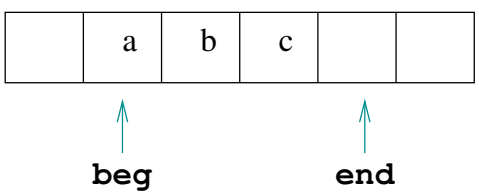
```
1: typedef struct queue {
2:   task A[MAXLEN];
3:   int begin, end, len;
4: } queue;
5:
6: queue q;

7: void init() {
8:   q->begin = 0;
9:   q->end = 0;
10:  q->len = 0;
11:}
```

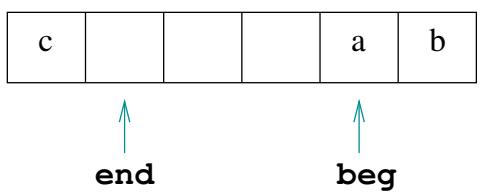
```
12: void task enq(task t){
13:   if (q->len == MAXLEN)
14:     assert(0); /*exception*/
15:   q->A[q->end] = t;
16:   if (q->end < MAXLEN-1)
17:     q->end++;
18:   else
19:     q->end = 0;
20:   q->len++;
21: }
22:
23: task deq() { ... }
```

(a)

```
1: task resched(
                task cur){
2:   task t;
3:   enq(cur);
4:   t = deq();
5:   return t;
6: }
```

(b)





A high-level specification of the queue functionality

$QADT_k$

$$
\begin{array}{lcl}
QADT_k & = & (Q, U, E, \{op_n\}_{n \in QType}) \text{ where} \\
Q & = & \{\epsilon\} \cup \bigcup_{i=1}^{k} \mathbb{B}^i \cup \{E\} \\
op_{init}(q, a) & = & \begin{cases} (\epsilon, ok) & \text{if } q \neq E \\ (E, e) & \text{otherwise.} \end{cases} \\
op_{enq}(q, a) & = & \begin{cases} (q \cdot a, ok) & \text{if } q \neq E \text{ and } |q| < k \\ (E, e) & \text{otherwise.} \end{cases} \\
op_{deq}(q, a) & = & \begin{cases} (q', b) & \text{if } q \neq E \text{ and } q = b \cdot q' \\ (E, e) & \text{otherwise.} \end{cases}
\end{array}
$$

Would like to argue that C implementation provides the same functionality as abstract queue specification.