

Rodin and Refinement

Deepak D'Souza

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

19 February 2026

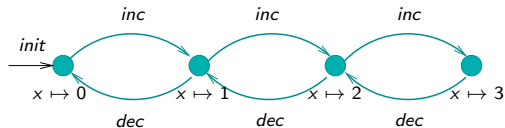
Specifying ADTs logically or declaratively

• States

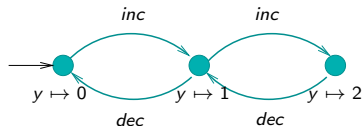
- Variables: Eg. $\{x\}$, $\{y\}$, $\{len, beg, end\}$.
- Invariants:
Eg. $x \in \mathbb{N}$, $x \leq 3$.
Eg. $0 \leq beg \leq len$.

• Operations (Eg. *inc*, *dec*, *enq*)

- Parameters: Eg. *newval*
- Guards: Eg. $x < 2$, $y < 2$, *newval* $\in \mathbb{N}$
- Update (or Action): Eg.
 $x' = x + 1$, $y' = y - 1$



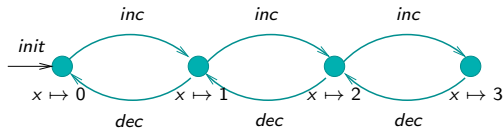
ADT 3counter



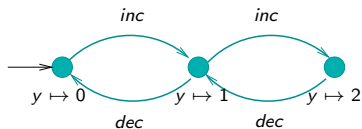
ADT 2counter

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*:

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

For *inc*:

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

Proof obligations

Refinement conditions:

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

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

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

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

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

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

Rodin tool

- Provides an environment for developing a system design by successive 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)
 - Event parameters
 - Modelling return values
 - List operations, relational operators.

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

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

- Events (guard **weakening**)

$$(A \wedge I \wedge J \wedge \textcolor{red}{G}) \Longrightarrow \textcolor{red}{H}.$$

- Events (invariant preservation)

$$(A \wedge I \wedge J \wedge \textcolor{red}{G} \wedge U) \Longrightarrow J[v'/v, w'/w].$$

Assert these as **theorems**.

A C implementation of a queue

```

1: typedef struct queue { 12: void task enq(task t){ 1: task resched(
2:   task A[MAXLEN];      13:   if (q->len == MAXLEN)           task cur){
3:   int begin, end, len;  14:   assert(0); /*exception*/ 2:   task t;
4: } queue;               15:   q->A[q->end] = t;                3:   enq(cur);
5:                       16:   if (q->end < MAXLEN-1) 4:   t = deq();
6: queue q;              17:   q->end++;                    5:   return t;
                       18:   else                                6: }
7: void init() {         19:   q->end = 0;
8:   q->begin = 0;        20:   q->len++;
9:   q->end = 0;          21: }
10:  q->len = 0;          22:
11:}                      23: task deq() { ... }

```

(b)



A high-level specification of the queue functionality

$QADT_k$

$$\begin{aligned} QADT_k &= (Q, U, E, \{\circ_n\}_{n \in QType}) \text{ where} \\ Q &= \{\epsilon\} \cup \bigcup_{i=1}^k \mathbb{B}^i \cup \{E\} \\ \circ_{init}(q, a) &= \begin{cases} (\epsilon, \succ) & \text{if } q \neq E \\ (E, e) & \text{otherwise.} \end{cases} \\ \circ(q, a) &= \begin{cases} (q \cdot a, \succ) & \text{if } q \neq E \text{ and } |q| < k \\ (E, e) & \text{otherwise.} \end{cases} \\ \circ(q, a) &= \begin{cases} (q', b) & \text{if } q \neq E \text{ and } q = b \cdot q' \\ (E, e) & \text{otherwise.} \end{cases} \end{aligned}$$

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