Rodin and refinement

Deepak D'Souza

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

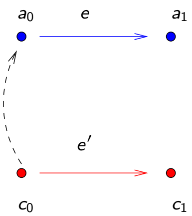
17 February 2020

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 $\mathcal{A}$.

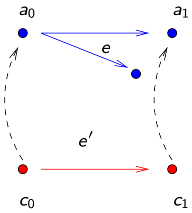
Refinement conditions in Rodin

Guard strengthening:



If a concrete event is enabled in a concrete state then the corresponding abstract event is also enabled in the abstract representation of the state.

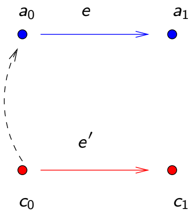
Simulation:

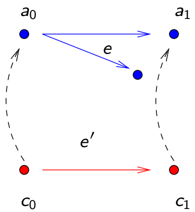


If a concrete event e' takes us from c_0 to c_1, then there should be a transition from the abstract representation of c_0 to the abstract representation of c_1, on the corresponding abstract event.

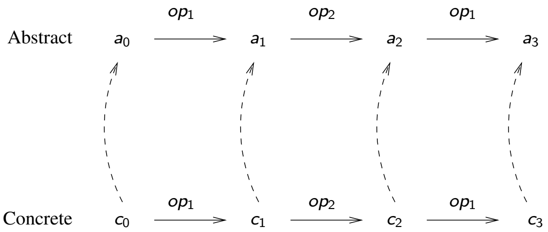
Refinement conditions imply simulation property

and





clearly implies that the abstract can simulate the concrete:



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

- Initialization

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

- Events (guard strengthening)

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

- Events (invariant preservation)

$$(A \wedge I \wedge J \wedge H \wedge T) \implies J[v'/v, w'/w].$$

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 our notion of refinement

- Initialization

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

- Events (guard weakening)

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

- Events (invariant preservation)

$$(A \wedge I \wedge J \wedge G \wedge T) \implies J[v'/v, w'/w].$$

Assert these as `theorems`.

Demo in Rodin

- Counter example demonstrating
  - Proof obligations generated by consistency checks
- Counter models demonstrating
  - Proof obligations generated by Rodin's notion of refinement
  - Theorems that assert our notion of refinement.
  - Using the Prover perspective to help Rodin complete a proof.

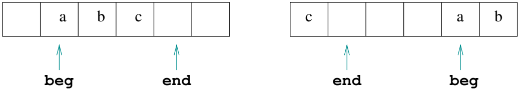
A C implementation of a queue

```
typedef struct queue {
  task A[MAXLEN];
  int begin, end, len;
} queue;

queue q;

void init() {
  q->begin = 0;
  q->end = 0;
  q->len = 0;
}
void task enq(task t){
  if (q->len == MAXLEN)
    assert(0); /*exception*/
  q->A[q->end] = t;
  if (q->end < MAXLEN-1)
    q->end++;
  else
    q->end = 0;
  q->len++;
}

task deq() { ... }
```

(a)

```
task resched(
           task cur){
  task t;
  enq(cur);
  t = deq();
  return t;
}
```

(b)



A high-level specification of the queue functionality

$QADT_k$

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

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