# Specification Languages for Temporal Contracts

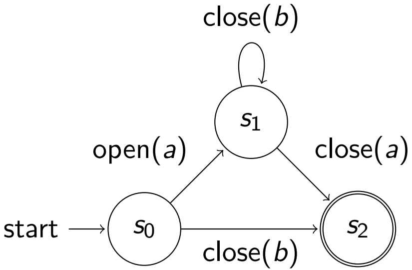
Christian Lidström, Dilian Gurov,
Paula Herber, Marieke Huisman

June 23, 2021
VTLTC seminar

# Finite State Machines

Here based on finite traces of *events*.

```
proc foo is
    if n = 0 then
        close b
    else
        open a;
        while (n > 0)
            n := n − 1;
            close b;
        close a;
```



# FSMs as Symbolic Transition Systems (in NuSMV style)

```
MODULE main

VAR
  action : {open, close};
  state  : {s0, s1, s2};

ASSIGN
  init(state) := s0;
  next(state) := case
                   state = s0 & action = open    : s1;
                   state = s0 & action = close   : s2;
                   state = s1 & action = close   : s2;
                 esac;
```

# Regular contracts

Also based on finite traces of *events*.

Syntax:

$$C ::= \mathrm{open}(L) \mid \mathrm{close}(L) \mid C^* \mid C_1 \; C_2 \mid C_1 \text{ or } C_2 \mid \text{not } C$$
$$L ::= l \mid \#$$

Example:

$$(\mathrm{open}(a) \; \mathrm{close}(b)^* \; \mathrm{close}(a)) \text{ or } \mathrm{close}(b)$$

# Context-free Grammars

```
proc even is
    open e;
    if n = 0 then
        r := 1
    else
        (n := n − 1; call odd);
    close e;
```

```
proc odd is
    open o;
    if n = 0 then
        r := 0
    else
        (n := n − 1; call even);
    close o;
```

$$E \leftarrow \text{open}(e)\ \text{close}(e) \mid \text{open}(e)\ O\ \text{close}(e)$$
$$O \leftarrow \text{open}(o)\ \text{close}(o) \mid \text{open}(o)\ E\ \text{close}(o)$$

# Interval Temporal Logic

Based on finite sequences of states $\sigma_1, \ldots, \sigma_n$.

Syntax:

$$f ::= p \mid \mathsf{false} \mid \neg f \mid f_1 \vee f_2 \mid \mathsf{skip} \mid \bigcirc f \mid f_1 {}^\frown f_2 \mid f^*$$

Examples:

$$(n \bmod 2 = 0)^\frown(\mathsf{skip}^*)^\frown(ret = 1) \;\vee\; (n \bmod 2 = 1)^\frown(\mathsf{skip}^*)^\frown(ret = 0)$$

$$(n \bmod 2 = 0)^\frown(\bigcirc(n \bmod 2 = 1)^\frown\bigcirc(n \bmod 2 = 0))^{*\frown}\bigcirc(n = 0) \;\vee\; \ldots$$

# Timed CSP

Let $\Sigma$ be an alphabet, $\mathcal{V}$ a set of process variables and $a \in \Sigma$, $A \subseteq \Sigma$, $X \in \mathcal{V}$.

$$P := \mathsf{STOP} \mid \mathsf{SKIP} \mid X \mid a \to P \mid P \,\Box\, P \mid P \sqcap P \mid$$
$$P; P \mid P \setminus P \mid P \underset{A}{\|} P \mid P \overset{d}{\triangleright} P \mid P \triangle_d P$$

Examples:

$$\mathrm{TimedPrinter} = (\mathsf{accept} \to \mathsf{print} \to \mathsf{STOP}) \overset{300}{\triangleright} \mathsf{shutdown} \to \mathsf{STOP}$$
$$\mathrm{Offer} = \mathsf{recommendation} \to ((\mathit{reject} \to \mathsf{STOP}) \overset{7}{\triangleright}$$
$$\mathsf{sendbook} \to \mathsf{payment} \overset{30}{\to} \mathit{Offer})$$

# Model-based specification in VerCors (1/2)

```
class Future {
  int x;

  modifies x;
  ensures x == \old(x) + 2;
  process incr();

  modifies x;
  ensures x == \old(x) + 4;
  process OG() = incr() || incr();
}
```

# Model-based specification in VerCors (2/2)

```
class Program {
  ensures \result == x + 4;
  int main(int x) {
    Future model = new Future();
    model.x = x;
    assert Perm(model.x, 1);

    create model, model.OG(); // initialise model
    assert Future(model, 1, model.OG())
        ** HPerm(model.x, 1);
    split model, 1\2, model.incr(), 1\2, model.incr();
    assert Future(model, 1\2, model.incr())
        ** Future(model, 1\2, model.incr())
        ** HPerm(model.x, 1);

    invariant inv(HPerm(model.x, 1)) //;
    {
      assert Future(model, 1\2, model.incr())
          ** Future(model, 1\2, model.incr());

      // fork and join threads, distribute model
      par Thread1()
        requires Future(model, 1\2, model.incr());
        ensures Future(model, 1\2, empty);
      {
        atomic (inv) {
          action(model, 1\2, empty, model.incr())
            { model.x = model.x + 2; }
        }
      }
      and Thread2()
        requires Future(model, 1\2, model.incr());
        ensures Future(model, 1\2, empty);
      {
        atomic (inv) {
          action(model, 1\2, empty, model.incr())
            { model.x = model.x + 2; }
        }
      }

      assert Future(model, 1\2, empty)
          ** Future(model, 1\2, empty);
      // After both threads have terminated, we may
      // merge the two models back into one again
      merge model, 1\2, empty, 1\2, empty;
      assert Future(model, 1, empty);
    }

    assert Future(model, 1, empty)
        ** HPerm(model.x, 1);
    destroy model; // finalise the model

    return model.x;
  }
}
```