Lecture 4

Finite Automata and Safe State Machines (SSM)

Daniel Kästner
AbsInt GmbH

2012

Initialization Analysis

- Is this node well initialized?

```
node init1() returns (out: int)
let
  out = 1 + pre( 1 -> (pre(out)));
tel
```

→ Initialization Error

- What does this node do?

```
node init2() returns (out: int)
let
  out = 1 + (1 -> pre(1 -> pre(out)));
tel
```

> `out = (2,2,3,3,4,4,…)`

Initialization Analysis

- $delay(c) = 0 \ \forall$ constants c
- $delay(X) = \begin{cases} 0, & if\ X \text{ is a signal} \\ \rho_i\,, & if\ X \text{ is input parameter } X_i \\ delay(E), & if\ X \text{ is local variable or} \\ & \text{output variable defined by } X = E \end{cases}$
- $delay(X \circ Y) = delay(X) \sqcup delay(Y)$ for combinatorial operators $\circ$
- $delay(pre\ E) = 1$
- $delay(X \rightarrow Y) = delay(X)$
- $delay(if\ E_1\ then\ E_2\ else\ E_3) = delay(E_1) \sqcup delay(E_1) \sqcup delay(E_3)$
- $delay(case\ E_1\ of\ E_2 \ldots E_k) = delay(E_1) \sqcup \cdots \sqcup delay(E_k)$

Initialization Analysis

- $delay(fby(E_1; d; E_2)) = delay(E_1) \sqcup delay(E_2)$
  - Note that this is different from the → operator.
- $delay(E_1 \; when \; E_2) = delay(E_1) \sqcup delay(E_2)$
- $delay\big(merge(h; E_1, \ldots, E_k)\big) = delay(h) \sqcup delay(E_1) \ldots \sqcup delay(E_k)$
- $delay(last\,'X) = \begin{cases} delay(X), & \text{if a last−value has been declared for X} \\ 1 & \text{, otherwise} \end{cases}$

Initialization Analysis

- $dcons^N(E_1 \rightarrow E_2, C^N) = C^N \cup dcons^N(E_1) \cup dcons^N(E_2)$

- $dcons^N(fby(E_1; d; E_2), C^N) = C^N \cup \{delay(E_1) = 0\}$
  $\cup \{delay(E_2) = 0\}$
- $dcons^N(E_1 \; when \; E_2, C^N) = C^N \cup \{delay(E_2) = 0\}$
- $dcons^N(merge(h; E_1 \; when \; B_1, \ldots, E_k when \; B_k), C^N)$
  $= C^N \cup \{delay(h) = 0\}$
  $\cup \{delay(E_1) = 0\} \cup \cdots$
  $\cup \{delay(E_k) = 0\}$
- $dcons^N(last \, 'X, C^N) = C^N \cup \{delay(X) = 0\}$
- $dcons^N(pre \; E, C^N) = C^N \cup \{delay(E) = 0\}$

Initialization Analysis

- At the beginning of the evaluation of the body of a node N: $C^N = \emptyset$.
- All constraints in C^N have to be simultaneously satisfied.
- Expressions contained in constraints in C^Ncan be decomposed according to the structure of E yielding new simplified constraints. Example: $delay(X \circ Y) = 0 \Leftrightarrow delay(X) = 0 \wedge delay(Y) = 0$
- After constraint simplification all constraints derived for input parameters are added to the node initialization type of N. The node initialization type is a function $\tau_1 \times \cdots \times \tau_m \rightarrow \sigma_1 \times \cdots \times \sigma_n$ where
  - $\tau_i = \begin{cases} 0, & \text{if input } X_i \text{ is constrained} \\ \rho_i, & \text{if input } X_i \text{ is unconstrained} \end{cases}$
  - $\sigma_i = delay(Y_i)$ for all output varibles Y_i

Initialization Analysis

- $delay(X_1, \dots, X_n = N(E_1, \dots, E_m)) =$
  $\sigma_1[\rho_i | delay(E_i)], \dots, \sigma_n[\rho_i | delay(E_i)]$

- $dcons^M\ (N(E_1, \dots, E_m), C^M) =$
  $C^M \cup \{delay(E_i) = 0 \mid \tau_i = 0 \text{ in node initialization type of } N\ \}$

Initialization Analysis

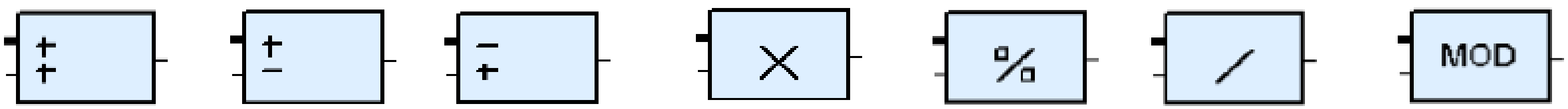
```
function f(clock h: bool; y,z: int) returns (o1: int; o2: bool)
let
  o1 = merge(h; y when h; z when not h);
  o2 = (y>z);
tel

node N(clock h: bool; y,z: int) returns (o1: int; o2: bool)
let
  o1,o2 = (1,true) -> f(h, pre y, pre z);
tel
```

Initialization Error: All the arguments of the merge operator must be well-initialized

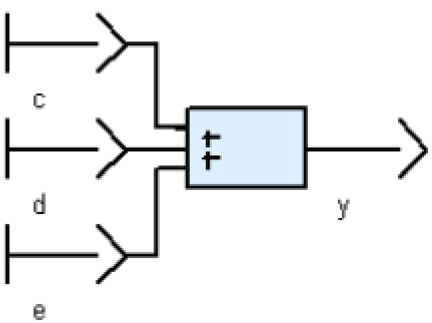
SCADE: The Graphical Language

- See SCADE Language Reference Manual.

- Arithmetic Operators:
  - "+" "-" "*" intdiv realdiv mod ...



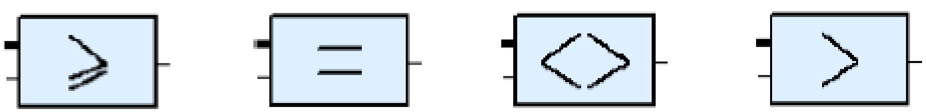
  - Example: y=c+d+e



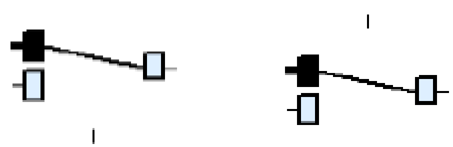
SCADE: The Graphical Language

- Logial Operators:
  - “or” “xor” “and” “not” ...

- Some Comparison Operators:

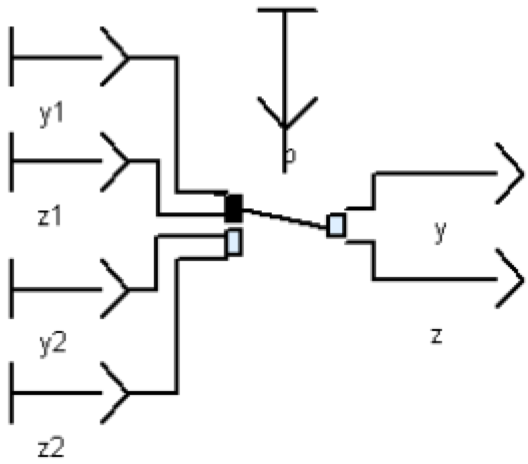


- Control Operators:
  - if ... then ... else ...



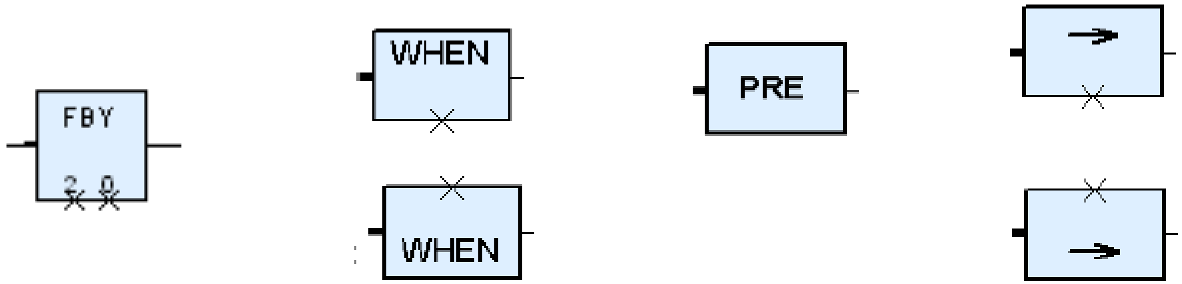
SCADE: The Graphical Language

- Example:
  - y,z=if b then (y1,z1) else (y2,z2)

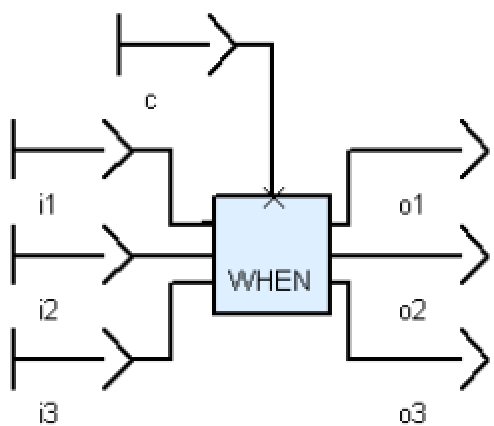


SCADE: The Graphical Language

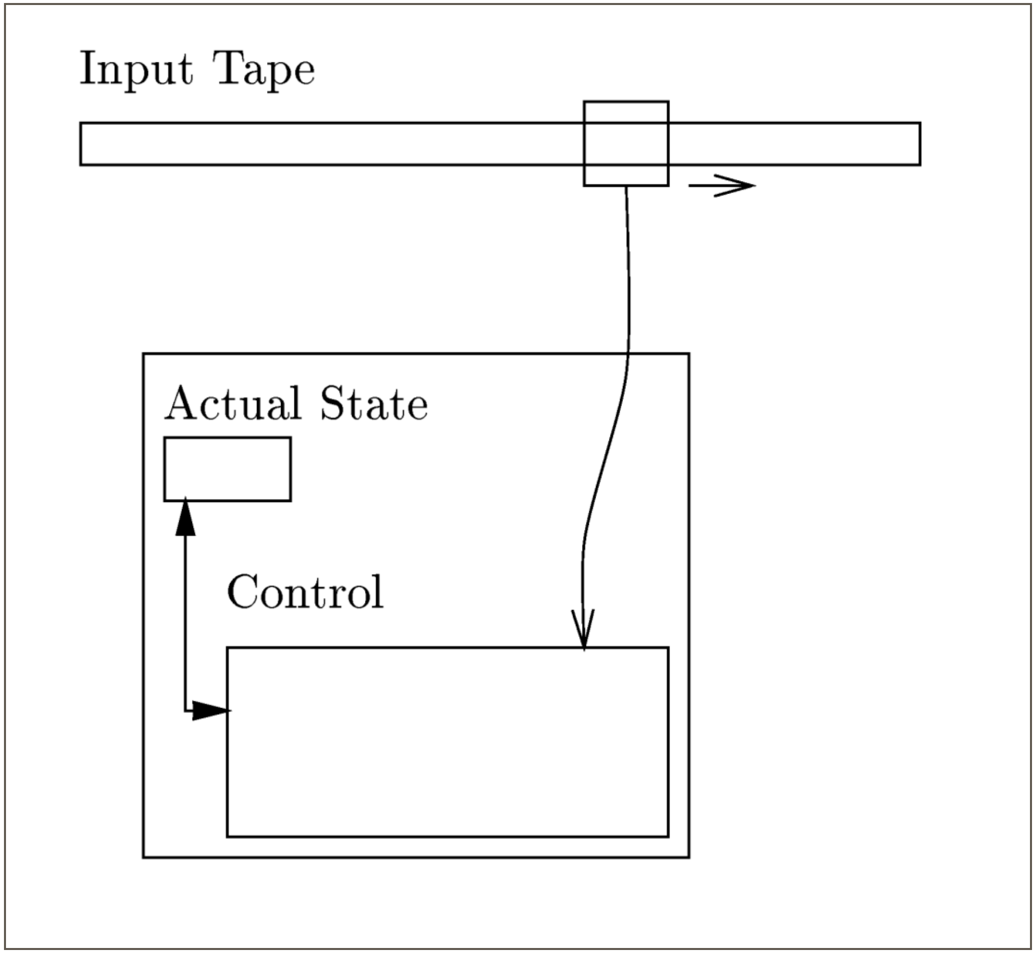
- Temporal Operators



- Example: o1,o2,o3 = (i1,i2,i3) when c



Finite Automata



Finite Automata

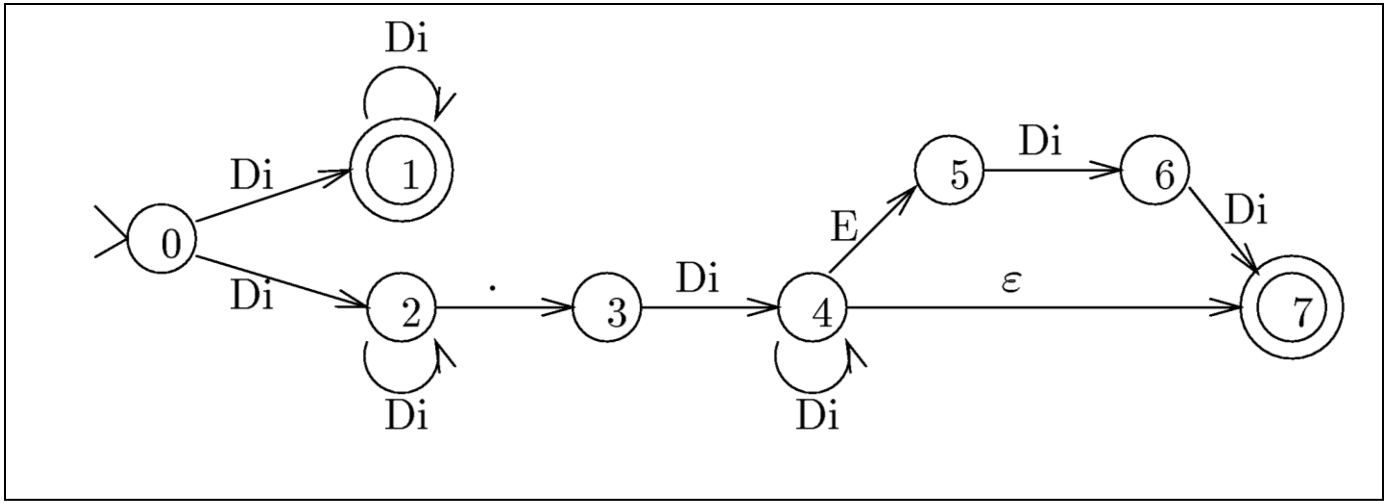
- Non-deterministic finite automaton (NFA):
  $M = (\Sigma, Q, \Delta, q_0, F)$ where
  - Σ: finite alphabet
  - Q: finite set of states
  - $q_0 \in Q$: initial state
  - $F \subseteq Q$: final states
  - $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$

- M is called a deterministic finite automaton, if Δ is a partial function

$$\delta : Q \times \Sigma \rightarrow Q$$

Simple State Transition Diagram

- Used to represent a finite automaton
- Nodes: states
- q_0 has special entry mark
- Final states are doubly circled
- An edge from *p* to *q* is labelled by *a* if $(p,a,q) \in \Delta$
- Example: integer and real constants:



Language Accepted by an Automaton

- $M = (\Sigma, Q, \Delta, q_0, F)$
- For $q \in Q$, $w \in \Sigma^*$: (q,w) is a configuration.
- Binary step relation $\vdash$ on configurations:

$$(q, aw) \vdash_M (p, w) \text{ iff } (q, a, p) \epsilon \Delta: Q \times \Sigma \to Q$$

- Reflexive transitive closure of $\vdash_M$ is denoted by $\vdash_M^*$
- Language accepted by M:

$$L(M) = \{w \in \Sigma^* \mid \exists q_f \in F: (q_0, w) \vdash_M^* (q_f, \epsilon)\}$$

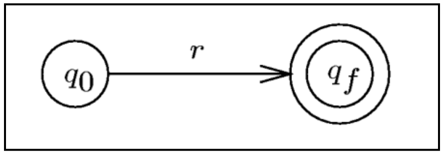
Regular Languages / Expressions

- Let Σ be an alphabet. The regular languages are defined inductively over Σ by:
  - $\emptyset$, $\{\varepsilon\}$ are regular languages over Σ
  - Forall a $\in \Sigma$, $\{a\}$ is a regular language
  - If R_1 and R_2 are regular languages over Σ, then also $R_1 \cup R_2$, R_1R_2, R_1^*.

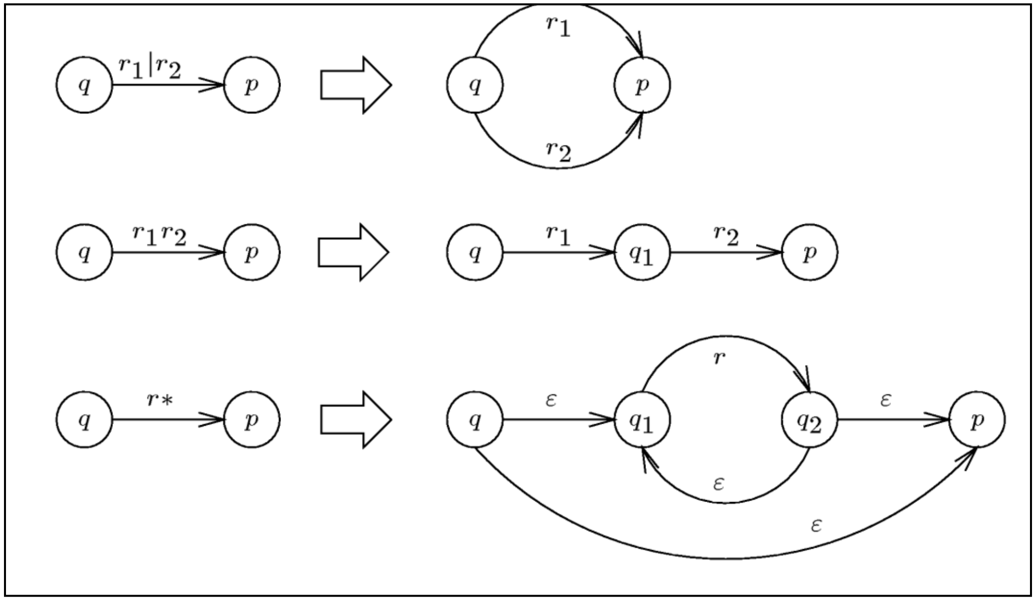
- Regular expressions over Σ are defined by:
  - $\emptyset$ is a regular expression and describes the language $\emptyset$
  - ε is a regular expression and describes the language $\{\varepsilon\}$
  - a (for a $\in \Sigma$) is a regular expression and denotes $\{a\}$
  - $(r_1|r_2)$ is a regular expression over Σ and denotes $R_1 \cup R_2$
  - (r_1r_2) is a regular expression over Σ and denotes R_1R_2
  - $(r_1)^*$ is a regular expression over Σ and denotes R_1^*.

Regular Expressions and FA

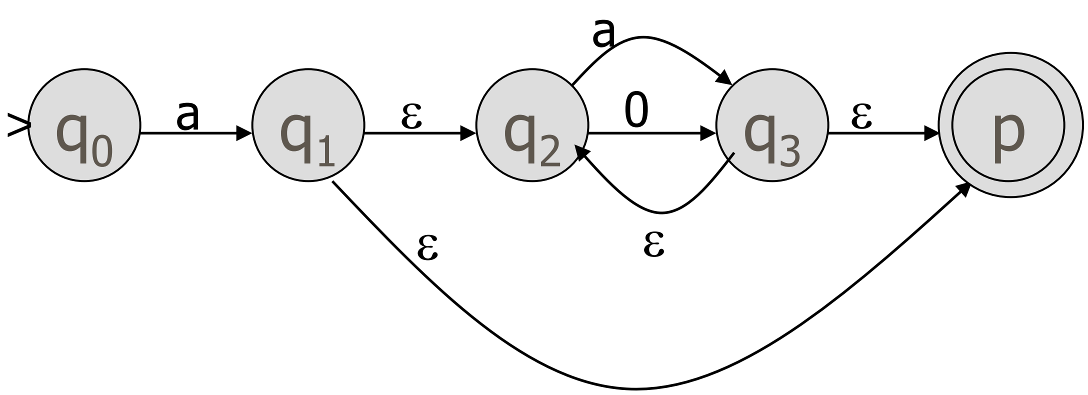
- For every regular language R, there exists an NFA M, such that L(M)=R.
- Constructive Proof (Subset Construction):
  - A regular language is defined by a regular expression r
  - Construct an NFA with one final state, q_f and the transition



  - Decompose r and develop the NFA according to the following rules -> until only transitions under single characters and ε remain.



Example: $a(a|0)^*$



Nondeterminism

- Sources of nondeterminism:
  - many transitions may be possible under the same character in a given state
  - ε-moves (next character is not read) may compete with non- ε-moves
- DFA:
  - No ε-transition
  - At most one transition from every state under a given character, ie for every $q \in Q$, $a \in \Sigma$:

$$|\{q' \mid (q,a,q') \in \Delta\}| \leq 1$$

NFA → DFA

- Let M=(Σ, Q, Δ, q_0, F) be an NFA and let q ∈ Q. The set of ε successor states of q, $\varepsilon SS(q)$, is
$$\varepsilon SS(q) = \{p | (q, \varepsilon) \vdash_M^* (p, \varepsilon)\}$$
or the set of all states p, including q, for which there exists an ε-path from q to p in the transition diagram for M.
- We extend εSS to sets of states S ⊆ Q:
$$\varepsilon SS(S) = \bigcup_{q \in S} \varepsilon SS(q)$$

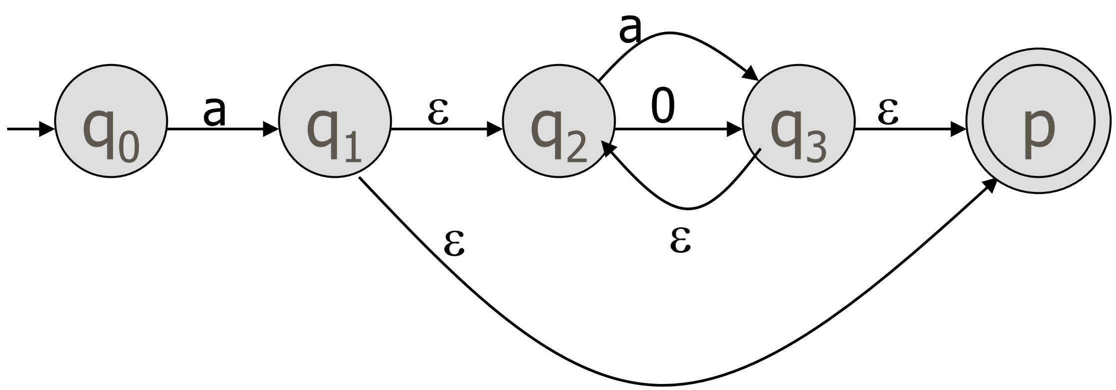
NFA → DFA

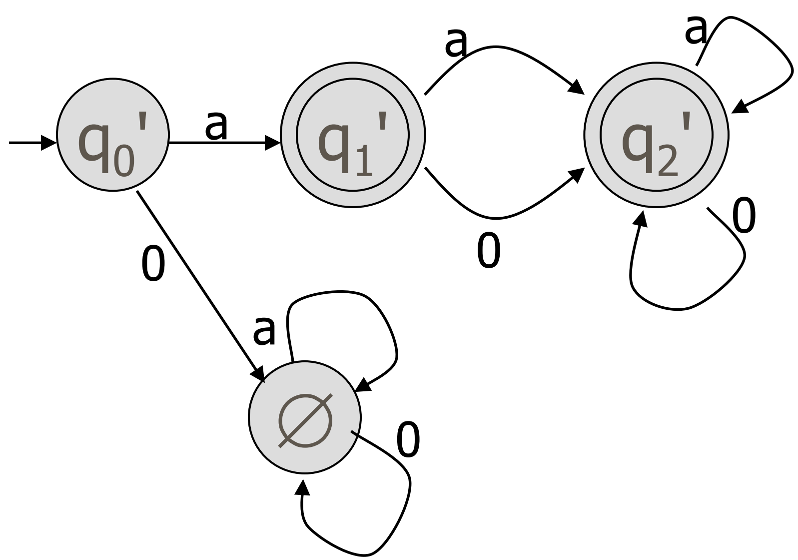
- If a language L is accepted by a NFA then there is also a DFA accepting L.
- Let $M=(\Sigma, Q, \Delta, q_0, F)$ be an NFA. The DFA associated with M, $M'=(\Sigma, Q', \delta, q_0', F')$ is defined by:
  - $Q' \subseteq \wp(Q)$
  - $q_0' = \varepsilon SS(q_0)$
  - $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$
  - $\delta(S,a) = \varepsilon SS(\{p \mid (q,a,p) \in \Delta \text{ for } q \in S\}) \text{ for } a \in \Sigma$
- Thus, the successor state of S under a character *a* in M' is obtained by combining the successor states of all states $q \in S$ under *a* and adding the ε successor states.

Algorithm NFA->DFA

$$
\begin{array}{l}
q'_0 := \varepsilon SS(q_0);\ Q' := \{q_0'\}; \\
\text{marked}(q_0') := \text{false};\ \delta := \emptyset; \\
\text{while } \exists\ S \in Q' \text{ and marked}(S) = \text{false do} \\
\quad \text{marked}(S) := \text{true}; \\
\quad \text{foreach } a \in \Sigma \text{ do} \\
\qquad T := \varepsilon SS(\{p \in Q \mid (q,a,p) \in \Delta \text{ and } q \in S\}); \\
\qquad \text{if } T \notin Q' \\
\qquad\qquad Q' := Q' \cup \{T\};\ \text{// new state} \\
\qquad\qquad \text{marked}(T) := \text{false} \\
\qquad \delta := \delta \cup \{(S,a) \to T\};\ \text{// new transition} \\
\quad \text{od} \\
\text{od}
\end{array}
$$

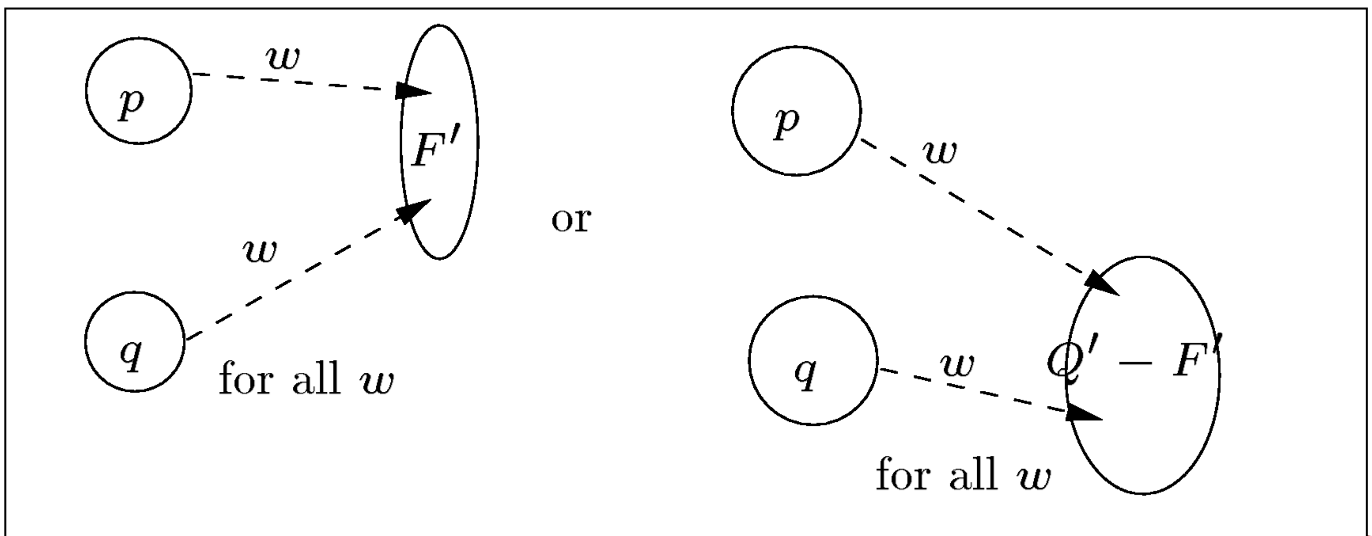
Example: $a(a|0)^*$





DFA Minimization

- After NFA->DFA the DFA need not have minimal size, ie minimal number of states and transitions.
- p and q are undistinguishable, iff for all words w both (q,w) and (p,w) lead by $\vdash_M^*$ into either F' or Q'-F'.
- Undistinguishable states can be merged.



DFA Minimization

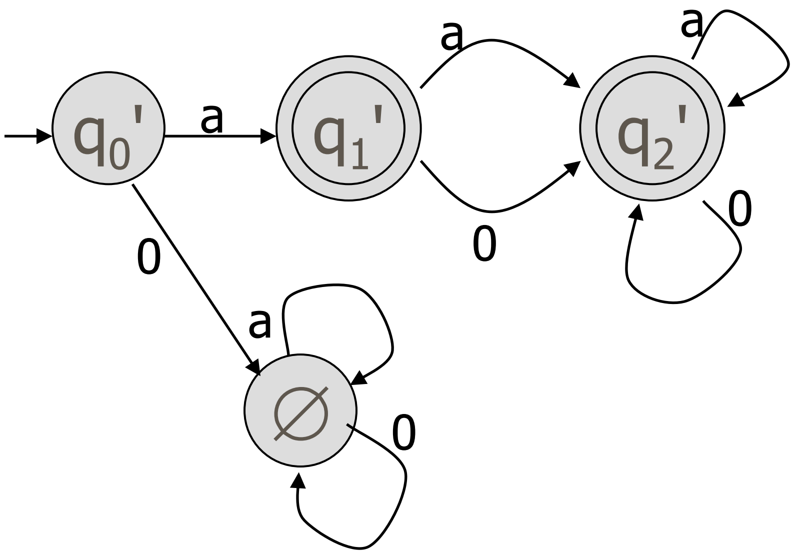
- Input: DFA $M=(\Sigma, Q, \delta, q_0, F)$
- Output: DFA $M_{min}=(\Sigma, Q_{min}, \delta_{min}, q_{0min}, F_{min})$ with $L(M)=L(M_{min})$ and Q_{min} minimal.
- Iteratively refine a partition of the set of states where each set S in the partition consists of states so far undistinguishable.
- Start with the partition $\Pi=\{F, Q-F\}$
- Refine the current Π by splitting sets $S\in\Pi$ into S_1, S_2 if there exist q_1, $q_2 \in S$ such that
  - $\delta(q_1,a) \in S_1$
  - $\delta(q_2,a) \in S_2$
  - $S_1 \neq S_2$
- Merge sets of undistinguishable states into a single state.

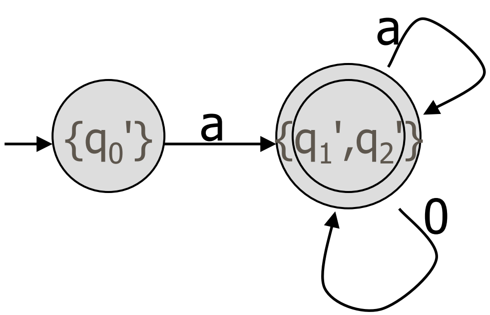
Algorithm minDFA

Π := {F, Q-F}
do changed := false
    $\Pi' := \Pi$;
    foreach K in Π do
        $\Pi' := (\Pi' - \{K\}) \cup \{\{K_i\}_{1 \le i \le n}\}$ with K_i maximal such that
        $K = \bigcup_{1 \le i \le n} K_i$ and $\forall a \in \Sigma \; \forall q \in K_i \; \exists K_i' \in \Pi : \delta(q,a) \in K_i'$
        if $n > 1$ then changed := true fi
    od
    $\Pi := \Pi'$;
until not changed;

- $Q_{min} = \Pi$ - (Dead $\cup$ Unreachable);
- q_{0min}: Class of Π containing q_0
- F_{min}: Classes containing an element of F
- $\delta_{min}(K,a)=K'$ if $\delta(q,a)=p$ with $a \in \Sigma$ and $p \in K'$ for one (ie for all) $q \in K$
- $K \in$Dead, if K is not final state and contains only transitions to itself
- K Unreachable, if there is no path from the initial state to K

Example: a(a|0)*





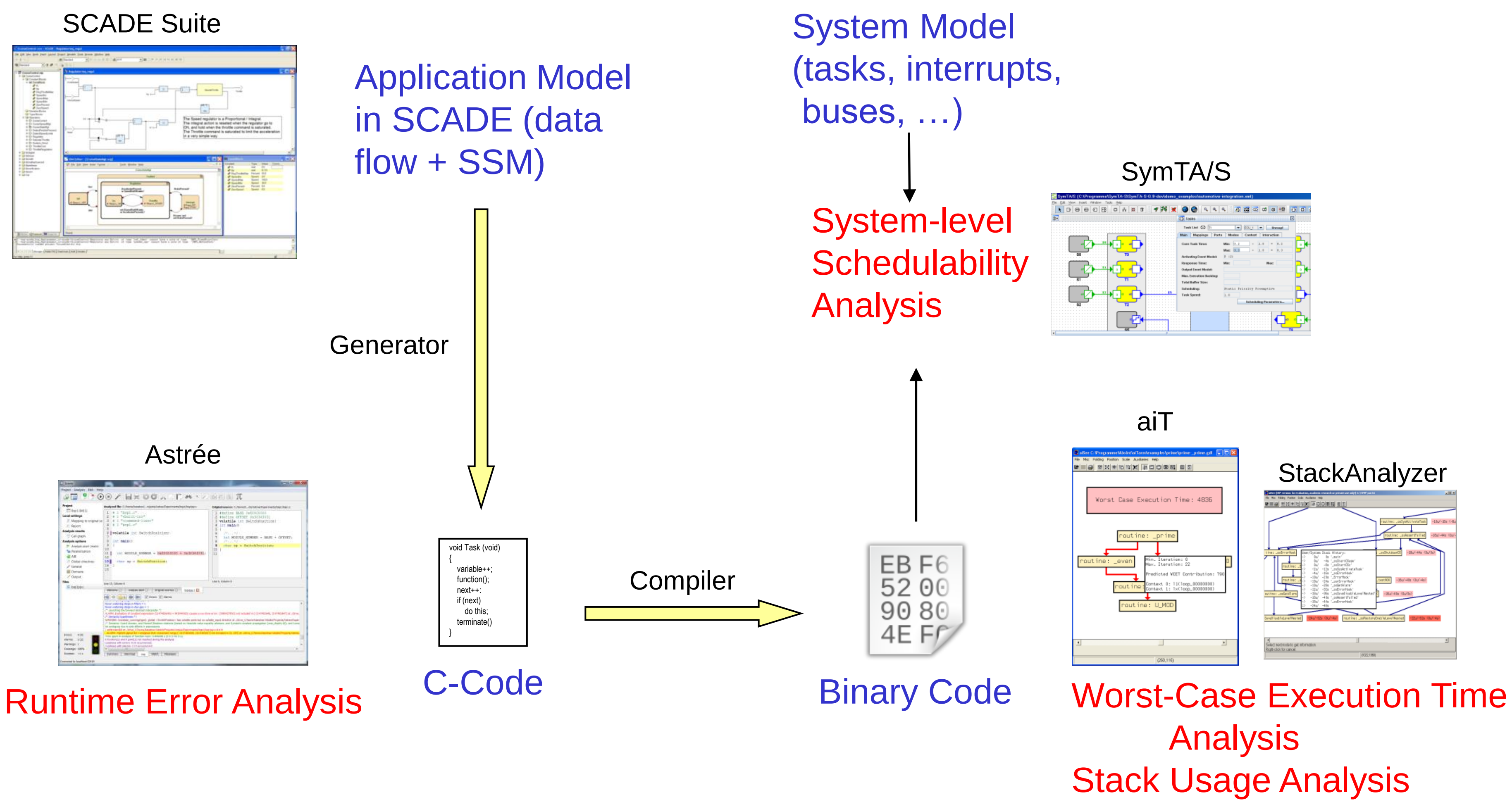
Mealy Automata

- Mealy automata are finite-state machines that act as transducers, or translators, taking a string on an input alphabet and producing a string of equal length on an output alphabet.
- A machine in state q_j, after reading symbol sigma σ_k writes symbol λ_k; the output symbol depends on the state just reached and the corresponding input symbol.
- A Mealy automaton is a six-tuple $M_E=(Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ where
  - Q is a finite set of states
  - Σ is a finite input alphabet
  - Γ is a finite output alphabet
  - δ: $Q \times \Sigma$ -> Q is the transition function
  - λ: $Q \times \Sigma$ -> Γ is the output function
  - q_0 is the initial state

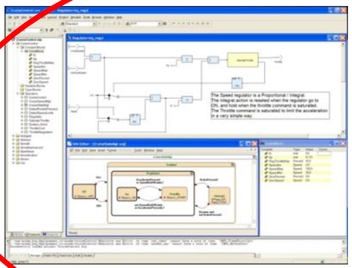
Moore Automata

- Moore automata are finite-state machines that act as transducers, or translators, taking a string on an input alphabet and producing a string of equal length on an output alphabet.
- Symbols are output after the transition to a new state is completed; output symbol depends only on the state just reached.
- A Moore automaton is a six-tuple $M_O=(Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ where
  - Q is a finite set of states
  - Σ is a finite input alphabet
  - Γ is a finite output alphabet
  - $\delta: Q \times \Sigma \rightarrow Q$ is the transition function
  - $\lambda: Q \rightarrow \Gamma$ is the output function
  - q_0 is the initial state

Model-based Software Development



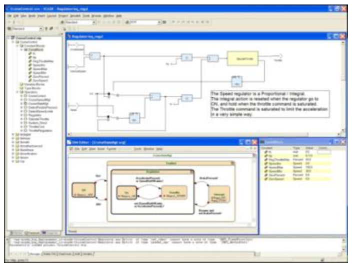
Model-based Software Development



Application Model in SCADE (data flow + SSM)

SyncCharts/SSM as Enhancement to FSA

Model-based Software Development



Application Model in SCADE (data flow + SSM)

Generator

Automata Minimization

```
void Task (void)
{
   variable++;
   function();
   next++:
   if (next)
      do this;
   terminate()
}
```

C-Code

Compilation of SCADE Programs

- Two alternatives:
  - Single-loop Code
  - Automaton Code
- Single-loop code
  - produce an infinite loop whose body implements the computations of each basic cycle of the node.
  - Expand all nodes and functions.
  - Sort the statements according to data dependences (acyclicity ensures that ordering exists)
  - If needed, introduce new variables for `pre` expressions.
  - Execute the code in an infinite loop
  - The resulting code will be suboptimal:
    - The choice of a good evaluation order is difficult.
    - All equations are computed in each step.

Example

```
node N(I:bool) returns (O:bool)
var X:bool;
let
    O = false->pre(X) and I;
    X = false->pre(I);
tel
```

→

```
init = true;
while (true) {
    read(I);
    if (init) then {
        O=false; X=false; init=false;
        PRE_I = I;
    }
    else {
        O = X and I;
        X = PRE_I;
        PRE_I = I;
    }
    write (O);
}
```

Example

```
init = true;
while (true) {
     read(I);
     if (init) then {
          O=false;
          X=false;
          init=false;
          PRE_I = I;
     }
     else {
          O = X and I;
          X = PRE_I;
          PRE_I = I;
     }
     write (O);
}
```

```
void N_reset(outC_N *outC)
{
  outC->init = kcg_true;
}

void N(inC_N *inC, outC_N *outC)
{
  if (outC->init) {
    outC->O = kcg_false;
  }
  else {
    outC->O = outC->X & inC->I;
  }
  if (outC->init) {
    outC->X = kcg_false;
  }
  else {
    outC->X = outC->rem_I;
  }
  outC->rem_I = inC->I;
  outC->init = kcg_false;
}
```

Automata Code

- Two observations:
  - In SCADE, imperative control structures are represented by conditional and temporal expressions.
  - If a conditional or temporal expression depends on a Boolean variable computed at previous cycles, specialized code could be generated for each value of the variable.
- Automata Generation:
  - Choose a set of state variables:
    - Boolean expressions resulting from `pre` operators
    - Auxiliary variables like _init_C for a clock C to allow the evaluation of -> operators.
  - For each possible value of the state define a node associated with the sequential code that would be executed with the corresponding variable setting.

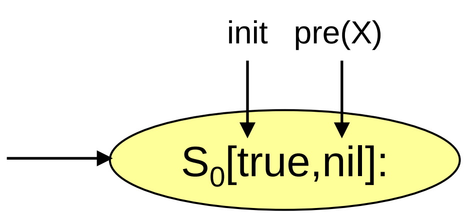
Automata Code

```
node EDGE(X:bool) returns (Y:bool)
let
    Y = false->(X and not pre(X))
tel
```

Single-Loop

```
init = true;
while (true) {
    read(X);
    if (init) then {
        Y=false; init=false;
        PRE_X = X;
    }
    else {
        Y = X and not PRE_X;
        PRE_X = X;
    }
    write (Y);
}
```

- Choose state variables init and pre(X).
- In the initial state: *init=true* and *pre(X)=nil*.
- Create a state S_0 with the tuple *init, pre(X):*



S0-Code:

```
Y=false;
```

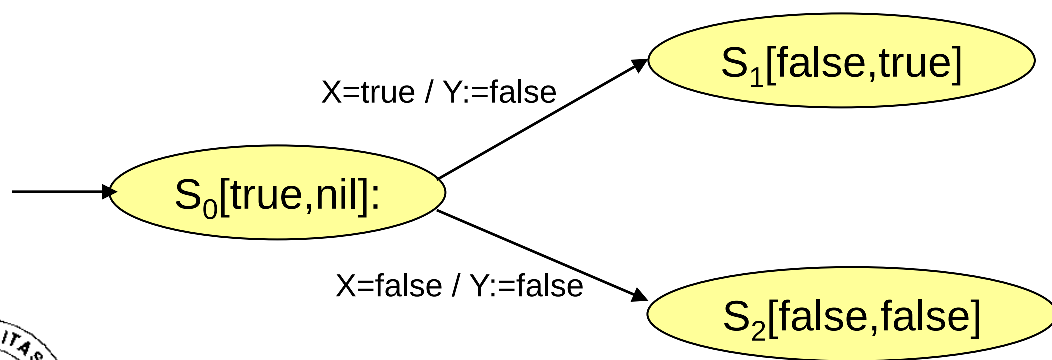
Automata Code

```
node EDGE(X:bool) returns (Y:bool)
let
    Y = false->(X and not pre(X))
tel
```

Single-Loop

```
init = true;
while (true) {
    read(X);
    if (init) then {
        Y=false; init=false;
        PRE_X = X;
    }
    else {
        Y = X and not PRE_X;
        PRE_X = X;
    }
    write (Y);
}
```

- State variables: init, pre(X)
- In the next step, init is false.
- pre(X) must be set correctly for each value of *X*.



S1-Code:

```
Y = X and false = false;
```

S2-Code:

```
Y = X and true = X;
```

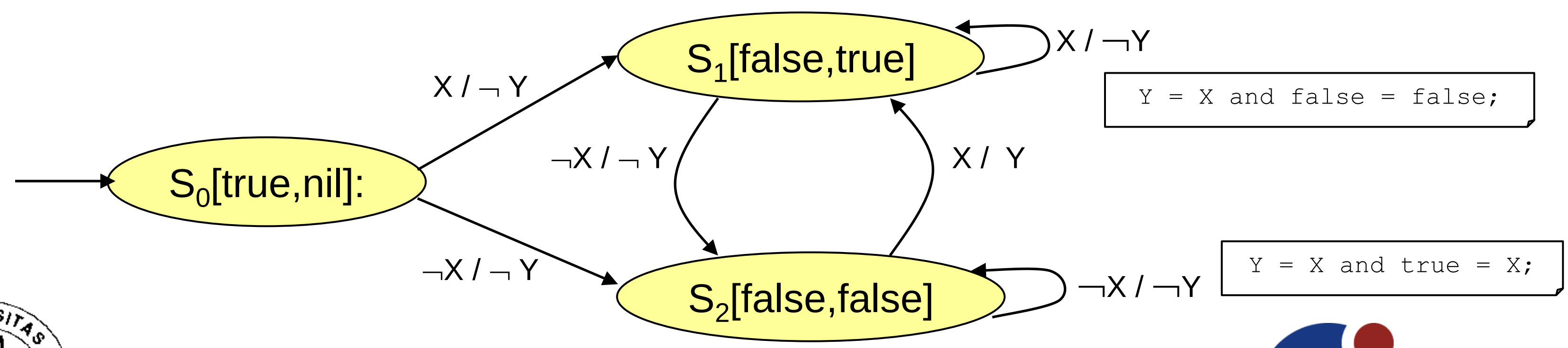
Automata Code

```
node EDGE(X:bool) returns (Y:bool)
let
    Y = false->(X and not pre(X))
tel
```

Single-Loop

```
init = true;
while (true) {
    read(X);
    if (init) then {
        Y=false; init=false;
        PRE_X = X;
    }
    else {
        Y = X and not PRE_X;
        PRE_X = X;
    }
    write (Y);
}
```

- *init* is never true again, so control moves between S_1 and S_2.
- Note: Behavior of automaton in S_0 and S_1 is the same.



Improving Code Efficiency

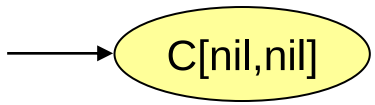
- Code generation is fast, but:
- The generated automaton usually is not minimal.
- Possible improvements:
  - Apply standard minimization algorithms (minDFA). However: whole automaton has to be constructed once, possibly involving exponential expansion of the code size.
  - Directly generate the minimal automaton, according to algorithm MinDFA => Demand-driven automata.
    But: compilation is slow.

Demand-Driven Automata

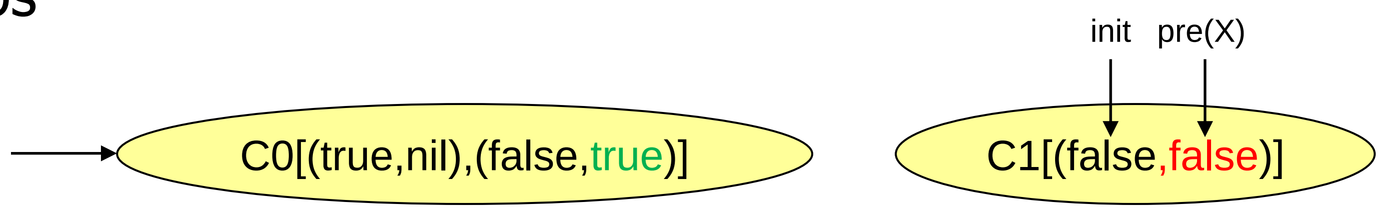
- Equivalence class of states (see powerset construction):
  - Two states are equivalent as long they have not been shown to be different
  - Two states are different, if they produce different outputs or lead to states which already have been shown to be different, in response to the same input.
- Algorithm:
  1. Start with one equivalence class, containing the whole program.
  2. Choose one equivalence class C and compute its outputs and successors. If this involves some unknown state information then split C into two states and compute for each predecessor to which of the two states it leads.
  3. If new states have been added goto 2. Otherwise return.

Example: Demand-Driven Construction

- Initially nothing is known



- Must split class C to allow computation of outputs in subsequent steps



- Compute outputs