Lecture 4

Finite Automata and Safe State Machines (SSM)

Daniel Kästner
AbsInt GmbH
2012
Initialization Analysis

- Is this node well initialized?

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

- What does this node do?

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

- out = (2, 2, 3, 3, 4, 4,...)
Initialization Analysis

- \( \text{delay}(c) = 0 \ \forall \ \text{constants} \ c \)

- \( \text{delay}(X) = \begin{cases} 0, & \text{if } X \text{ is a signal} \\ \rho_i, & \text{if } X \text{ is input parameter } X_i \\ \text{delay}(E), & \text{if } X \text{ is local variable or output variable defined by } X = E \end{cases} \)

- \( \text{delay}(X \circ Y) = \text{delay}(X) \sqcup \text{delay}(Y) \) for combinatorial operators \( \circ \)
- \( \text{delay}(\text{pre } E) = 1 \)
- \( \text{delay}(X \rightarrow Y) = \text{delay}(X) \)
- \( \text{delay}(\text{if } E_1 \text{ then } E_2 \text{ else } E_3) = \text{delay}(E_1) \sqcup \text{delay}(E_1) \sqcup \text{delay}(E_3) \)
- \( \text{delay}(\text{case } E_1 \text{ of } E_2 \ldots E_k) = \text{delay}(E_1) \sqcup \cdots \sqcup \text{delay}(E_k) \)
Initialization Analysis

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

- \( \text{dcons}^N(E_1 \rightarrow E_2, C^N) = C^N \cup \text{dcons}^N(E_1) \cup \text{dcons}^N(E_2) \)

- \( \text{dcons}^N(fby(E_1; d; E_2), C^N) = C^N \cup \{\text{delay}(E_1) = 0\} \cup \{\text{delay}(E_2) = 0\} \)

- \( \text{dcons}^N(E_1 \text{ when } E_2, C^N) = C^N \cup \{\text{delay}(E_2) = 0\} \)

- \( \text{dcons}^N(\text{merge}(h; E_1 \text{ when } B_1, \ldots, E_k \text{ when } B_k), C^N) = C^N \cup \{\text{delay}(h) = 0\} \cup \{\text{delay}(E_1) = 0\} \cup \ldots \cup \{\text{delay}(E_k) = 0\} \)

- \( \text{dcons}^N(\text{last }'X, C^N) = C^N \cup \{\text{delay}(X) = 0\} \)

- \( \text{dcons}^N(\text{pre } E, C^N) = C^N \cup \{\text{delay}(E) = 0\} \)
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 $C_N$ can be decomposed according to the structure of E yielding new simplified constraints. Example: $delay(X \circ Y) = 0 \iff delay(X) = 0 \land delay(Y) = 0$
- 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 $\tau_1 \times \cdots \times \tau_m \to \sigma_1 \times \cdots \times \sigma_n$ 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 variables $Y_i$
Initialization Analysis

- \( \text{delay}(X_1, \ldots, X_n = N(E_1, \ldots, E_m)) = \sigma_1[\rho_i|\text{delay}(E_i)], \ldots, \sigma_n[\rho_i|\text{delay}(E_i)] \)

- \( \text{dcons}^M (N(E_1, \ldots, E_m), C^M) = C^M \cup \{\text{delay}(E_i) = 0 | \tau_i = 0 \text{ in node initialization type of } N \} \)
Initialization Analysis

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

**Initialization Error:** All the arguments of the merge operator must be well-initialized
SCADE: The Graphical Language


- Arithmetic Operators:
  - “+”       “-”    “*”       intdiv  realdiv  mod ...

- Example: y = c + d + e
SCADE: The Graphical Language

- **Logical Operators:**
  - "or"
  - "xor"
  - "and"
  - "not"
  ...  

- **Some Comparison Operators:**

- **Control Operators:**
  - if ... then ... else ...
SCADE: The Graphical Language

- Example:
  - \( y, z = \text{if } b \text{ then } (y_1, z_1) \text{ else } (y_2, z_2) \)
SCADE: The Graphical Language

- Temporal Operators

- Example: \( o_1, o_2, o_3 = (i_1, i_2, i_3) \) when \( c \)
Finite Automata
Finite Automata

- **Non-deterministic** finite automaton (NFA):
  \[ M = (\Sigma, Q, \Delta, q_0, F) \] where
  - \( \Sigma \): 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 \( \Delta \) is a partial function

  \[ \delta : Q \times \Sigma \to Q \]
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:
  $$(q, aw) \vdash_M (p, w) \text{ iff } (q, a, p) \epsilon \Delta: Q \times \Sigma \rightarrow Q$$
- Reflexive transitive closure of $\vdash_M$ is denoted by $\vdash^*_M$
- Language accepted by $M$:
  $$L(M) = \{w \in \Sigma^* | \exists q_f \in F: (q_0, w) \vdash^*_M (q_f, \epsilon)\}$$
Regular Languages / Expressions

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

- Regular expressions over $\Sigma$ are defined by:
  - $\emptyset$ is a regular expression and describes the language $\emptyset$
  - $\varepsilon$ 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 $\Sigma$ and denotes $R_1 \cup R_2$
  - $(r_1r_2)$ is a regular expression over $\Sigma$ and denotes $R_1R_2$
  - $(r_1)^*$ is a regular expression over $\Sigma$ 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 \( \varepsilon \) 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, i.e., for every \( q \in Q, a \in \Sigma \):
    \[
    | \{ q' \mid (q, a, q') \in \Delta \} | \leq 1
    \]
NFA → DFA

- Let $M=(\Sigma, Q, \Delta, q_0, F)$ be an NFA and let $q \in Q$. The set of $\varepsilon$ 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 $\varepsilon$-path from $q$ to $p$ in the transition diagram for $M$.

- We extend $\varepsilon SS$ to sets of states $S \subseteq Q$:
  $$\varepsilon SS(S) = \bigcup_{q \in S} \varepsilon SS(q)$$
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:

- $Q' \subseteq \mathcal{P}(Q)$
- $q_0' = \varepsilon \text{SS}(q_0)$
- $F' = \{ S \subseteq Q \mid S \cap F \neq \emptyset \}$
- $\delta(S, a) = \varepsilon \text{SS}(\{ p \mid (q, a, p) \in \Delta \text{ for } q \in S \})$ 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 $\varepsilon$ successor states.
Algorithm NFA->DFA

$q'_0 := \varepsilon S(q_0); Q' := \{q'_0\};$
marked($q'_0$) := false; $\delta := \emptyset$

while $\exists S \in Q'$ and marked($S$) = false do
    marked($S$) := true;
    foreach $a \in \Sigma$ do
        $T := \varepsilon S\{p \in Q| (q,a,p) \in \Delta \text{ and } q \in S\};$
        if $T \notin Q'$
            $Q' := Q' \cup \{T\};$ // new state
            marked($T$) := false
            $\delta := \delta \cup \{(S,a) \rightarrow T\};$ // new transition
        od
    od
DFA Minimization

- After NFA->DFA the DFA need not have minimal size, i.e., 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 $M_{\text{min}} = (\Sigma, Q_{\text{min}}, \delta_{\text{min}}, q_{0\text{min}}, F_{\text{min}})$ with $L(M) = L(M_{\text{min}})$ and $Q_{\text{min}}$ 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 $\Pi$ 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

\[ \Pi := \{F, Q-F\} \]
do changed := false
\[ \Pi' := \Pi; \]
foreach K in \(\Pi\) do
\[ \Pi' := (\Pi' - \{K\}) \cup \{\{K_i\}_{1 \leq i \leq n}\} \] with \(K_i\) maximal such that
\[ K = \bigcup_{1 \leq i \leq n} K_i \]\(\text{ 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;

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

The diagram illustrates a nondeterministic finite automaton (NFA) with states $q_0'$, $q_1'$, $q_2'$, and a special state $\emptyset$. Transitions include:
- $a$ from $q_0'$ to $q_1'$
- $a$ from $q_1'$ to $q_2'$
- $a$ from $q_2'$ (loop)
- $0$ from $\emptyset$ to $q_1'$
- $0$ from $q_2'$ to $\emptyset$
- $a$ from $\emptyset$ to $q_0'$
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 $\sigma_k$ writes symbol $\lambda_k$; 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
  - $\Sigma$ is a finite input alphabet
  - $\Gamma$ is a finite output alphabet
  - $\delta: Q \times \Sigma \rightarrow Q$ is the transition function
  - $\lambda: Q \times \Sigma \rightarrow \Gamma$ 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
  - $\Sigma$ is a finite input alphabet
  - $\Gamma$ 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

SCADE Suite

Application Model in SCADE (data flow + SSM)

Generator

Astrée

System Model (tasks, interrupts, buses, …)

System-level Schedulability Analysis

SymTA/S

Generator

C-Code

Compiler

C-Code

Compiler

Runtime Error Analysis

Runtime Error Analysis

Worst-Case Execution Time Analysis

Stack Usage Analysis

SCADE Suite

System Model (tasks, interrupts, buses, …)

System-level Schedulability Analysis

SymTA/S

Generator

C-Code

Compiler

C-Code

Compiler

Runtime Error Analysis

Runtime Error Analysis

Worst-Case Execution Time Analysis

Stack Usage Analysis

SCADE Suite

Application Model in SCADE (data flow + SSM)

Generator

Astrée

System Model (tasks, interrupts, buses, …)

System-level Schedulability Analysis

SymTA/S

Generator

C-Code

Compiler

C-Code

Compiler

Runtime Error Analysis

Runtime Error Analysis

Worst-Case Execution Time Analysis

Stack Usage Analysis

SCADE Suite

Application Model in SCADE (data flow + SSM)

Generator

Astrée

System Model (tasks, interrupts, buses, …)

System-level Schedulability Analysis

SymTA/S

Generator

C-Code

Compiler

C-Code

Compiler

Runtime Error Analysis

Runtime Error Analysis

Worst-Case Execution Time Analysis

Stack Usage Analysis

SCADE Suite

Application Model in SCADE (data flow + SSM)

Generator

Astrée

System Model (tasks, interrupts, buses, …)

System-level Schedulability Analysis

SymTA/S

Generator

C-Code

Compiler

C-Code

Compiler

Runtime Error Analysis

Runtime Error Analysis

Worst-Case Execution Time Analysis

Stack Usage Analysis

SCADE Suite

Application Model in SCADE (data flow + SSM)

Generator

Astrée

System Model (tasks, interrupts, buses, …)

System-level Schedulability Analysis

SymTA/S

Generator

C-Code

Compiler

C-Code

Compiler

Runtime Error Analysis

Runtime Error Analysis

Worst-Case Execution Time Analysis

Stack Usage Analysis
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

C-Code

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

```c
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);
}
```

```c
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 `-o` 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.
Choose state variables init and pre(X).
In the initial state: init=true and pre(X)=nil.
Create a state S₀ with the tuple init, pre(X):

\[
\text{S₀}[\text{true}, \text{nil}]:
\]

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);
}
Automata Code

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

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

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);
}

Single-Loop

\[ S_0[true,nil] \quad \rightarrow \quad S_1[false,true] \quad \rightarrow \quad S_2[false,false] \]

S1-Code:
\[ Y = X \text{ and false} = \text{false}; \]

S2-Code:
\[ Y = X \text{ and true} = X; \]
Automata Code

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

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