FINITE-STATE MODELING OF SOFTWARE

Sukhamay Kundu

Computer Science Dept.
Louisiana State University, Baton Rouge, USA
(kundu@bit.csc.lsu.edu)

Outline:

- Simple behavior-patterns and finite-state models
- Finite-state model of a program
- ‡ Some common problems in an FSM from design perspective
- Simplification of an FSM by eliminating problem-transitions and states
- ‡ Automatic code generation from FSM
- Data-centric notion of a state
- ‡ A complex design problem starting from requirements: elevator controller design
- ‡ Deriving FSMs for subsystems from the global FSM
- Several FSMs interacting with each other

SOFTWARE BEHAVIOR

- Performs certain desired sequences of computations, including input/output activities.

  The number of valid activity sequences can be very large, even infinite, if there are infinitely many input situations.

- The "pattern" in these activity-sequences determine the software structure.
- A software is a compact description which incorporates each of these activity-sequences by taking advantage of their common parts.
- Understanding of the (structure) *pattern* of the activity-sequences is therefore a prerequisite to a good software model and a good design.

behavior ⟶ model ⟶ design

Jackson-Method:

Design the software-structure to fit the desired behavior pattern.

EXAMPLES

OF

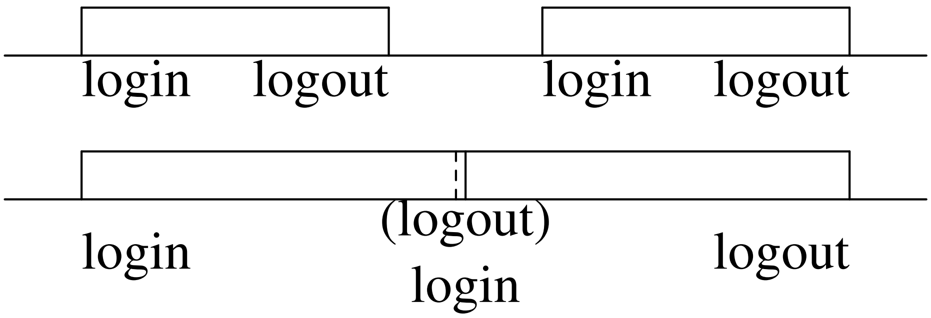
BEHAVIOR-PATTERNS
(Activity-Sequences)

AND

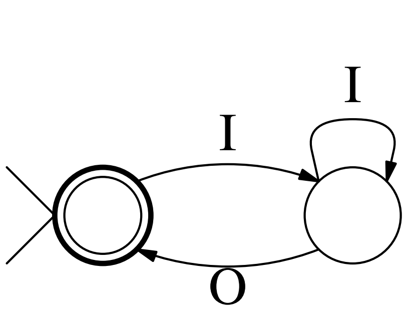
FINITE-STATE MODELS

PATTERN IN ACTIVITY SEQUENCES FOR LOGIN-LOGOUT OPERATIONS

Example 1. The pattern of simple login (I) - logout (O):



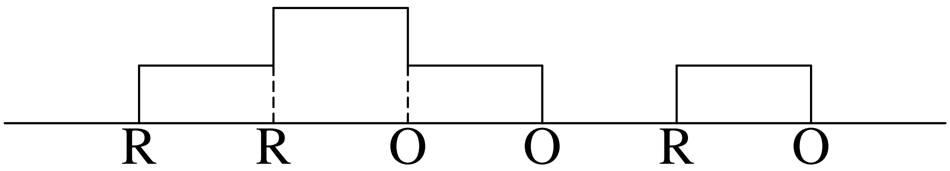
Valid sequences:	λ, IO, IIO, IOIO, IIIO, $\cdots$
Invalid sequences:	I, O, II, OO, OI, OOO, $\cdots$
Not modeled:	Activities between login and logout
Key properties:	(a) Must begin with I and end with O (b) An O can only be followed by I



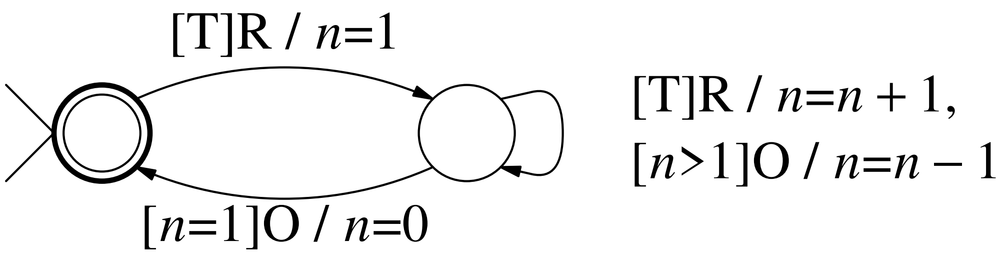
A finite-state model with two states.

A MORE COMPLEX PATTERN FOR RLOGIN-LOGOUT

Example 2. The pattern of rlogin (R) - logout (O):



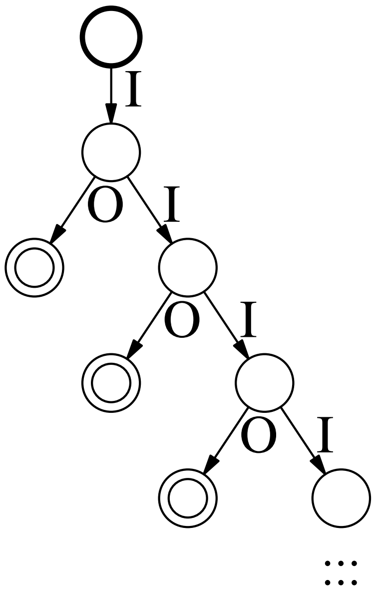
Valid sequences:	λ, RO, RROO, RORO, $\cdots$
Invalid sequences:	R, O, OO, OR, OOO, OOR, $\cdots$
Not modeled:	Activities between rlogin and logout
Key properties:	Balanced string of R and O (a) Must begin with R and end with O (b) #(R) $\geq$ #(O) for each initial part (c) #(R) = #(O) for the whole string



The guards [n > 1] and [n = 0] disambiguates the two transitions at the state on right.

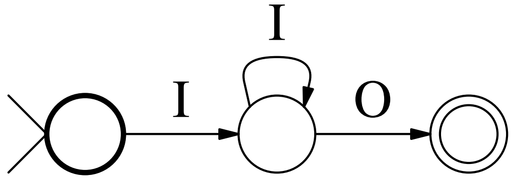
BEHAVIOR-TREES

- All valid input or action sequences, arranged in the form of a tree, to indicate their common initial parts.
- It helps in better understanding of those sequences.

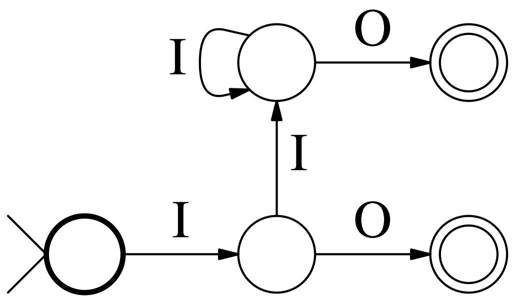


An infinite-state model: one state for each node

Tree of all valid login-logout behavior sequences for one complete session.

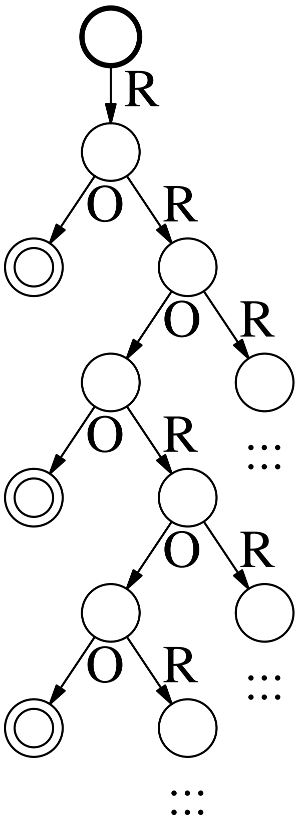


A modified FSM for a single session.

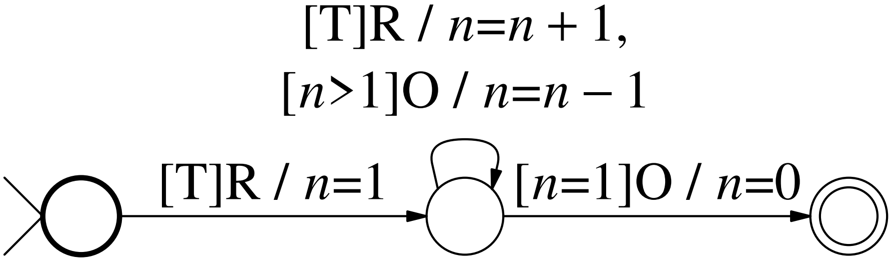


An alternate FSM with two extra states.

BEHAVIOR-TREES(Contd.)

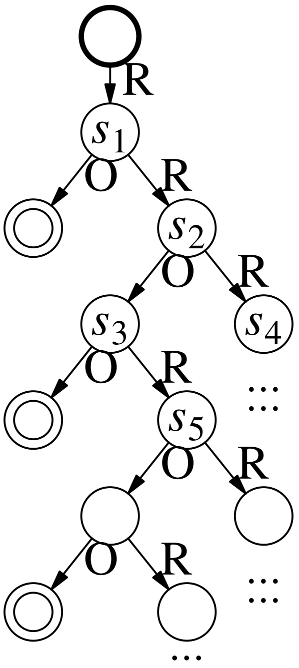


Tree of all valid rlogin-logout behavior sequences for one complete session.

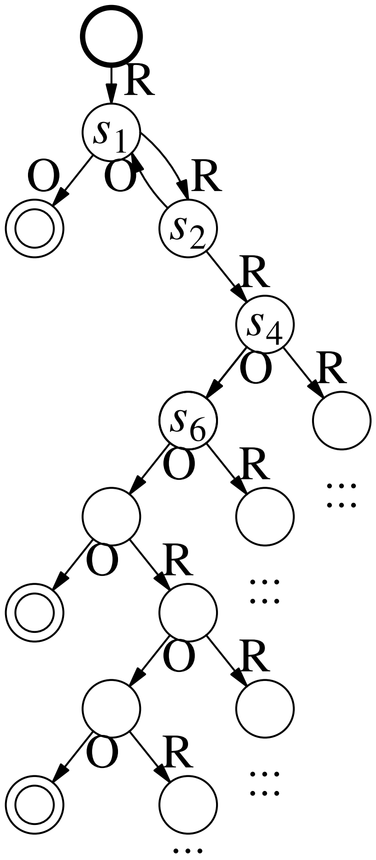


Question: Are there other FSMs in this case?

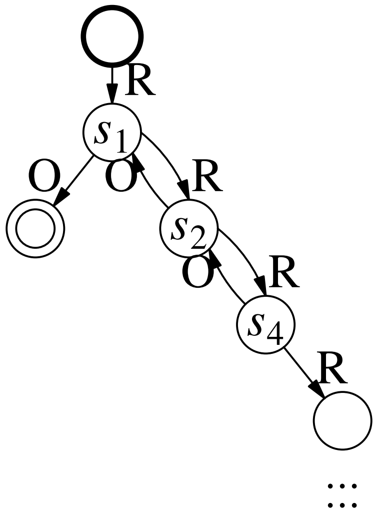
FROM BEHAVIOR-TREE TO FSM: A COMPLEX PROCESS OF STATE-REDUCTION

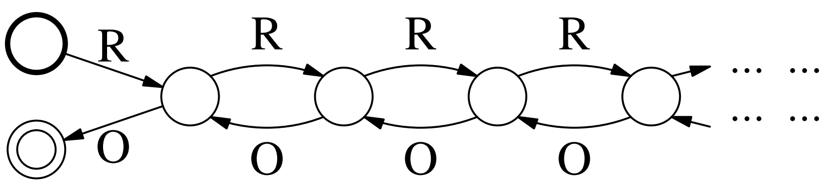


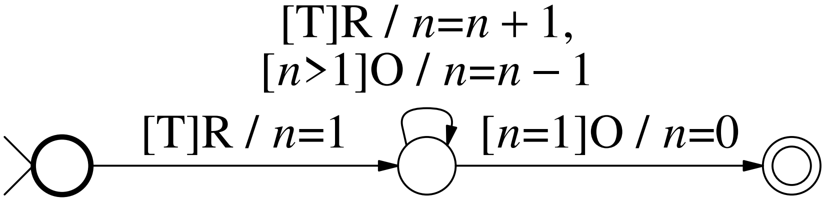
The subtrees at s_1 and s_3 are identical, as are those at s_2 and s_5.



The subtrees at s_2 and s_6 are identical.





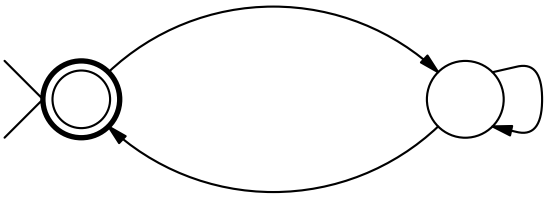


THE PATTERN OF ACTIVITIES IN WEB-NAVIGATION

Example 3. Consider the basic activities:

Start	Close	Goto
Back	Forward	

[T]Start /
create root with
homeAddress and let
curPosition = root



[T]Close /
root = NULL

[hasParent(curPosition)]Back /
curPosition = parent(curPosition),

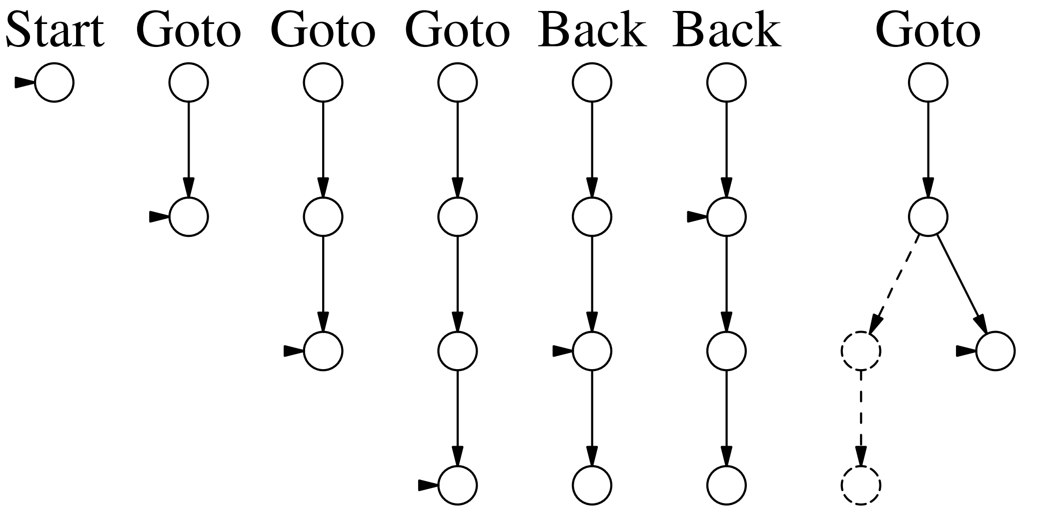
[hasChild(curPosition)]Forward /
curPosition = next(curPosition),

[addr(curPosition) $\neq$ addr(Goto)]
Goto /
Remove child and descendents (if any), add child(curPosition) with addr(Goto)

- The guards also determine which actions are applicable at a state.
- Removing the guard for processing "Goto" would amount to changing the associated action to

  if (addr(curPosition) $\neq$ addr(Goto)) then remove child and descendents $\cdots$

NAVIGATION-PATH DATA MAINTAINED BY THE ACTIONS



- These are not parts of the behavior-tree.

Questions:

(1) What is the behavior-tree for the web-navigation?

(2) How to transform the behavior-tree to the FSM shown earlier?

FINITE-STATE MODEL

OF

A PROGRAM

WHY MAKE A FINITE-STATE MODEL

- FSM is a *finite* description of a potentially *infinite* number of behavior-sequences. (The same is true for a program.)
- Can easily build different abstractions at higher levels and build subsystem-models.
- Can generate semi-automated code from the model.

FSM is programming language independent

Remarks:

(1) Converting a program P to an FSM $M(P)$ can be automated.

(2) Building an FSM from the problem-statement, without the software, cannot be fully automated, and it remains mostly a human-centered activity.

A SAMPLE PROGRAM

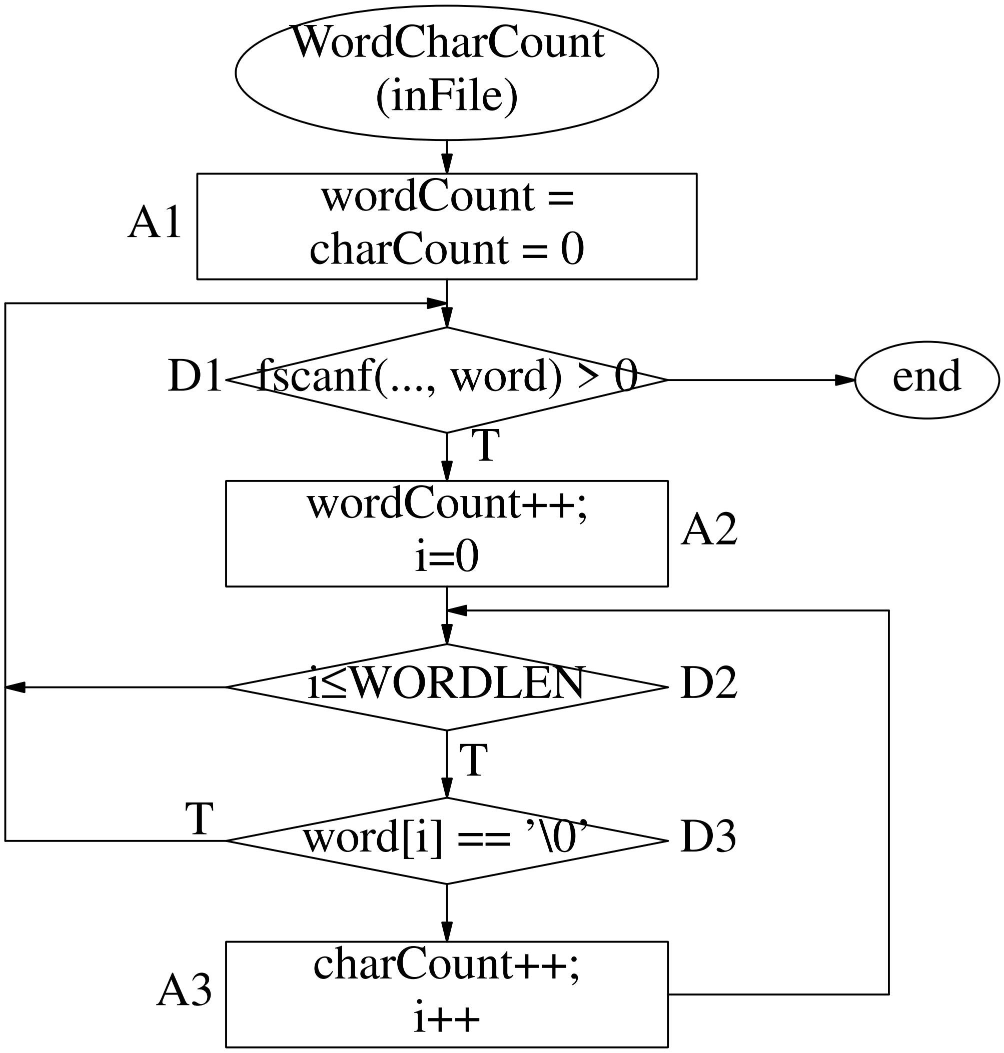
```
//Scan input textfile and set values of
//global vars wordCount and charCount;
//blanks, newlines, etc. are not counted.
//Assume each word has length <= WORDLEN.

//inFile ≠ NULL
void WordCharCount(FILE *inFile)
{int i;
 char word[WORDLEN+1]; //1 for end of string

 wordCount = charCount = 0;
 while (fscanf(inFile, "%s", word) > 0) {
       wordCount++;
       for (i=0; i<=WORDLEN; i++)
           if (word[i] == '\0') break;
           else charCount++;
 }
}
```

Behavior Patterns:

Patterns of read, update, and test operations.

Question: What is the behavior-tree here?

FLOWCHART OF WordCharCount



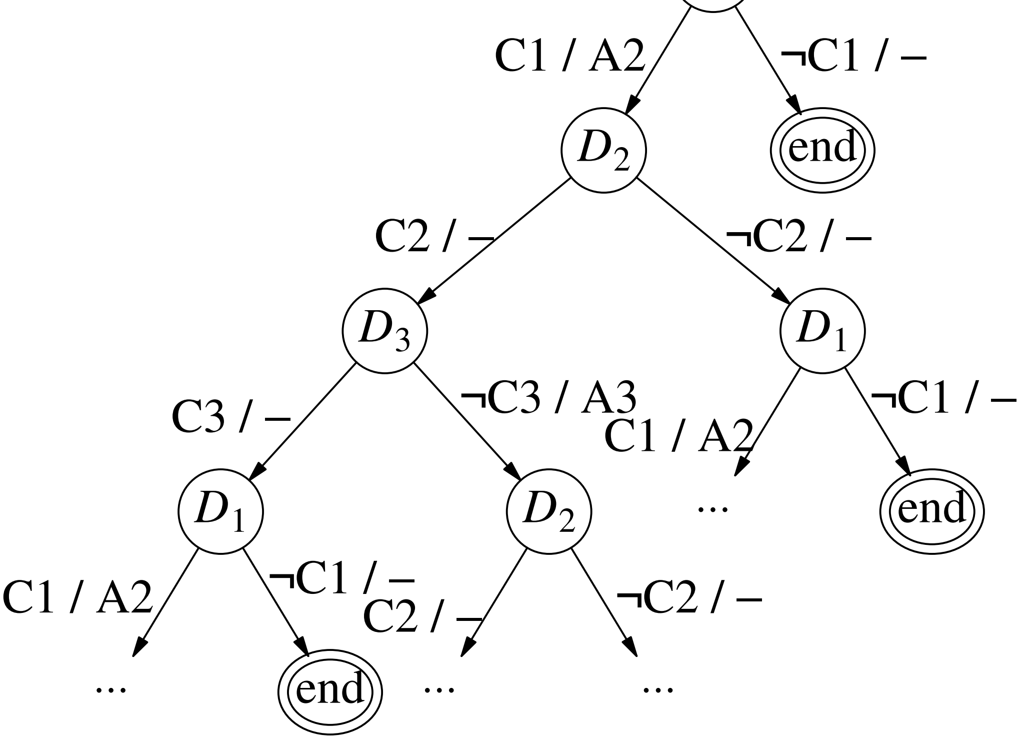
Decision to decision path (DD-path):

The "chunk" of activities between branch-tests.

- Start → A1
- D1 → A2 (→ D2); D1 → end;
- D2 (→ D1); D2 (→ D3)

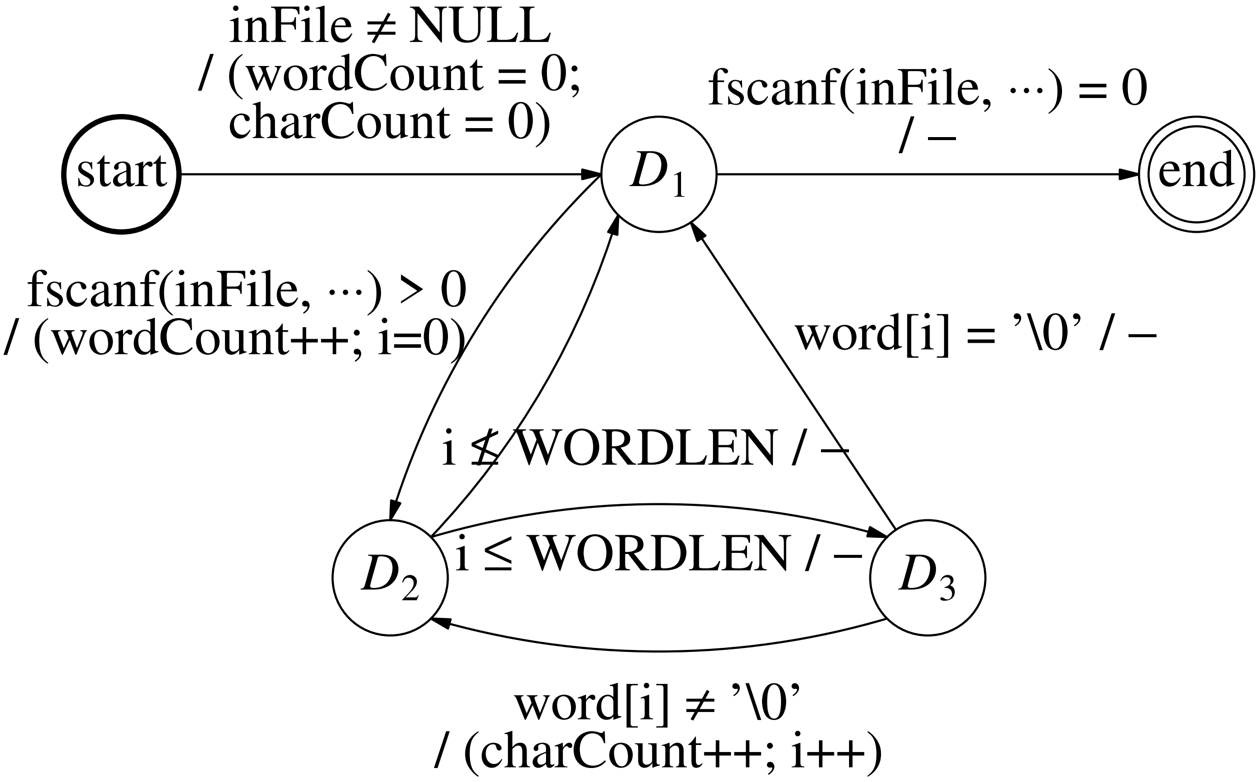
BEHAVIOR-TREE FOR THE FLOWCHART OF WordCharCount

C0: inFile ≠ NULL
C1: fscanf(inFile, ··) > 0
C2: i ≤ WORDLEN
C3: word[i] == '\0'



Some of these paths may not be executable and thus do not represent valid activity-sequence.

FINITE-STATE MODEL FROM DD-PATHS



Conditions and actions: c_{ij}/a_{ij}.

- Each node, other than the start and end nodes, has two transitions from it.
- Cycles in the flowchart give cycles in the FSM.

Each decision-node plus the values of local/global vars. equals an history-abstraction that determines the future.

DEFINITION OF AN FSM

(1) Finite number of states s_i
- Two special states: start and end

(2) Finite number of transitions $t_{ij} = (s_i, s_j)$ between two states:
- condition $c_{ij} = c(s_i, s_j)$
- action $a_{ij} = a(s_i, s_j)$
- c_{ij} and a_{ij} can be arbitrarily complex

(3) No transition to start-node, and no transition from end-node.

(4) The transitions from each state s_i are mutually disjoint:
- $c_{ij} \wedge c_{ij'} = \text{False}$ for $j \neq j'$

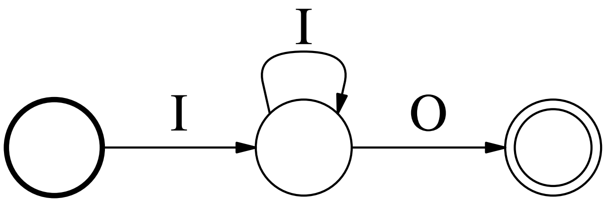
(5) The transitions from each state s_i are complete:
- $\bigvee_{all\ j} c_{ij} = \text{True}$

Computation:

A path $\pi = \langle \text{start}, s_{i_1}, s_{i_2}, \cdots, s_{i_n}, \text{end} \rangle$ such that each condition $c_{i_k i_{k+1}}$ is satisfied by the current values of variables.

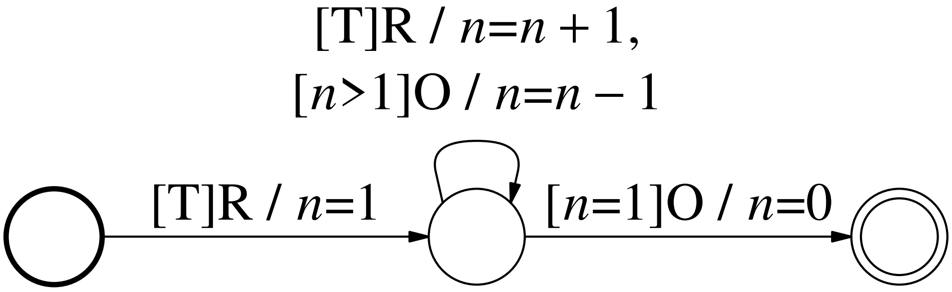
IMPACT OF CONDITION-GUARDS

Finite-state Automata: Transitions without guards.



- Actions at a state s_i DOES NOT depend on the past, i.e., how one arrives at s_i. (Each transition from s_i is applicable every time we are at s_i.)

Finite-State Models: Transitions with guards.



- Actions at a state s_i DEPENDS on the past, which determines values of variables in a guard.

Deterministic:.

- Only one transition from s_i is applicable for any action-class each time we are at s_i, which may be different for different arrivals at s_i.

SUMMARY

$\boxed{\text{All computations can be modeled, at any desired level, by FSMs with condition-guards.}}$

Remarks:

- This is basically a restatement of the Church-Turing hypothesis:

  Each computation/algorithm can be modeled by a Turing Machine.

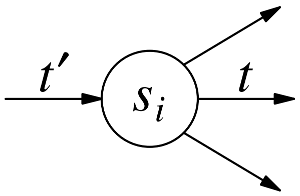
- Construction of an FSM, without a program in hand, may not be easy.

SIMPLIFICATION OF

AN FSM

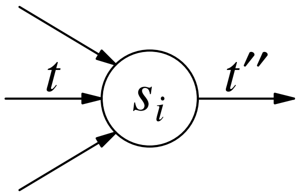
COMMON PROBLEMS IN A FINITE STATE MODEL

- **Unused transitions:**
  Not used in any computation-path.
- **Precedence dependency**:
  Transition t is precedent dependent on t' if t' must be used *before* t.



A trivial case:
t' is the only transition to s_i.

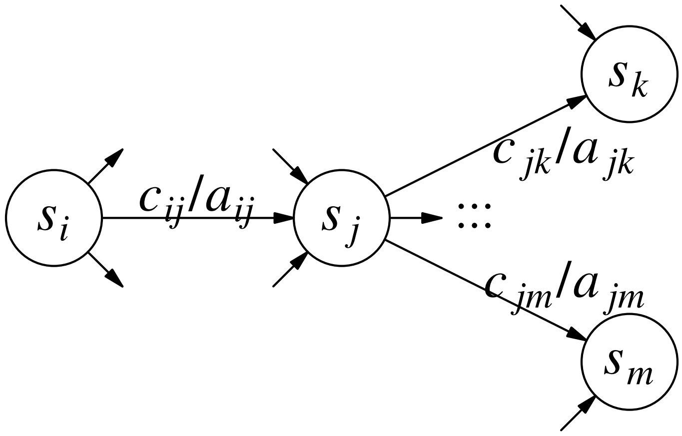
- **Successor dependency**:
  Transition t is successor dependent on t'' if t'' must be used *after* t.



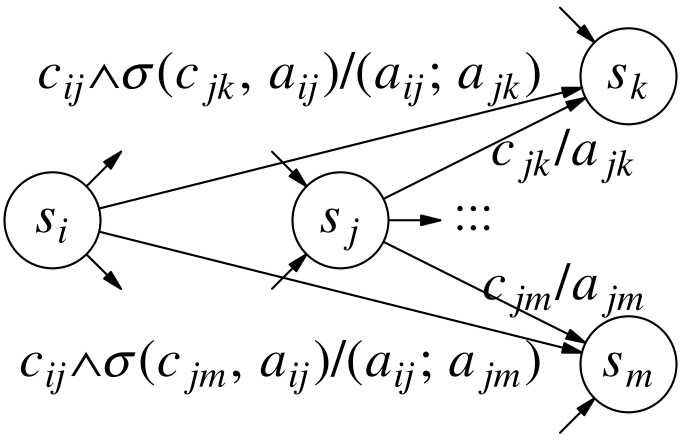
A trivial case:
t'' is the only transition from s_i.

Dependencies may exist between transitions that do not follow each other immediately (the head of one transition $\neq$ the tail of the other).

TRANSITION ELIMINATION



Before elimination of (s_i, s_j).

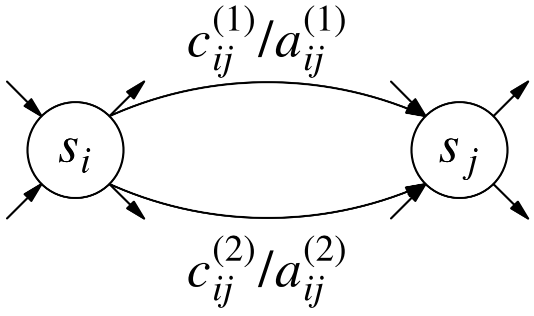


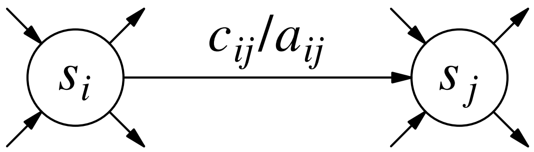
After elimination of (s_i, s_j).

$$
\begin{array}{rl}
a_{ij}: & (x = x + y;\ y = f(x, y, z)) \\
c_{jk}: & x > y \\
\hline
\sigma(c_{jk}, a_{ij}): & (x + y) > f(x + y, y, z)
\end{array}
$$

The modified form $\sigma(c_{jk}, a_{ij})$ of c_{jk}.

ELIMINATION OF PARALLEL TRANSITIONS





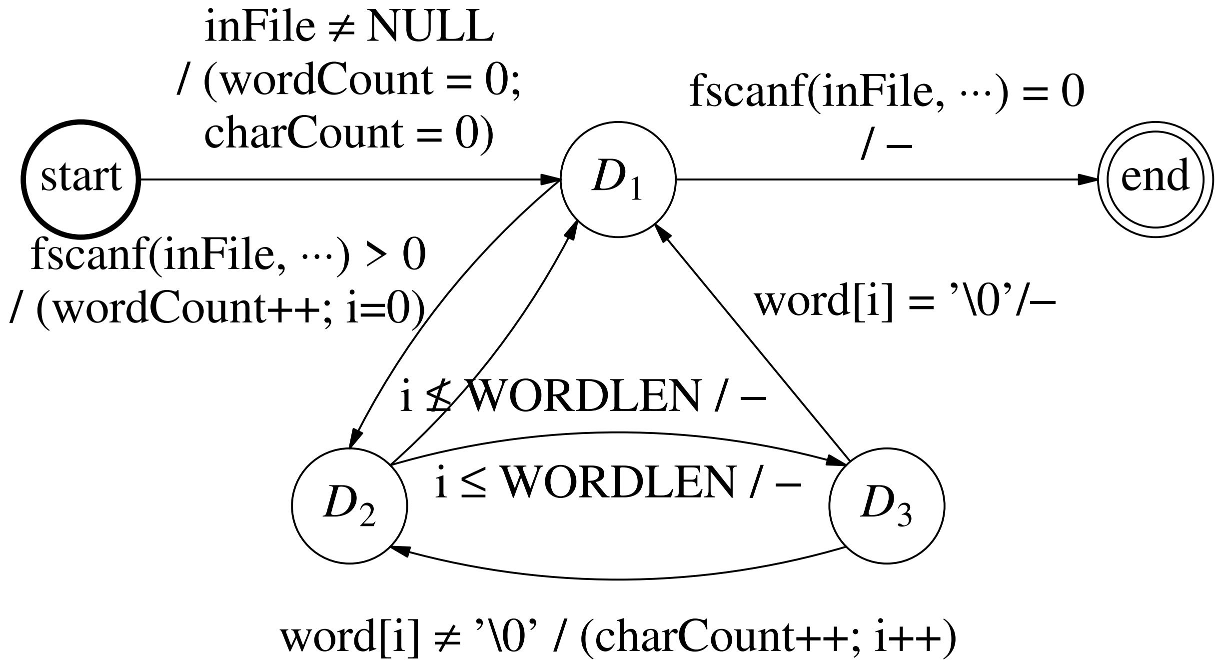
$$
\begin{aligned}
c_{ij} &= c_{ij}^{(1)} \vee c_{ij}^{(2)} \quad \text{[disjoint]} \\
a_{ij} &= \text{If } (c_{ij}^{(1)}) \text{ then } a_{ij}^{(1)} \text{ else } a_{ij}^{(2)} \\
&= \text{If } (c_{ij}^{(2)}) \text{ then } a_{ij}^{(2)} \text{ else } a_{ij}^{(1)}
\end{aligned}
$$

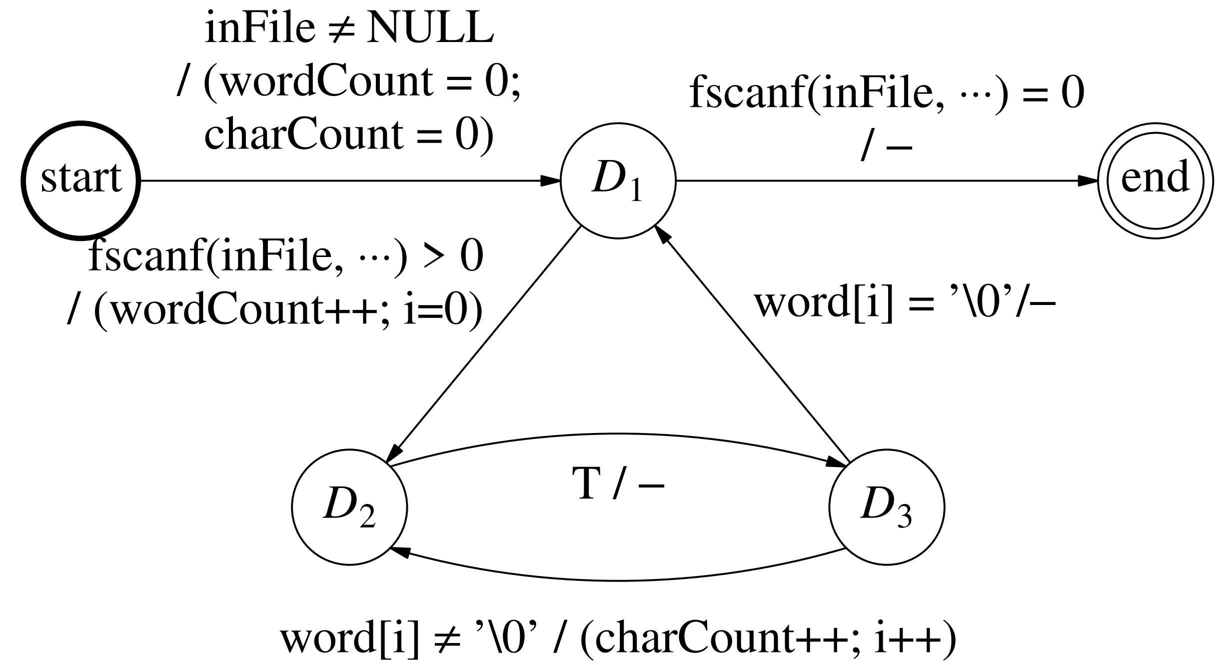
$$\boxed{a_{ij} \text{ is more complex than both } a_{ij}^{(1)} \text{ and } a_{ij}^{(2)}}$$

Remarks:

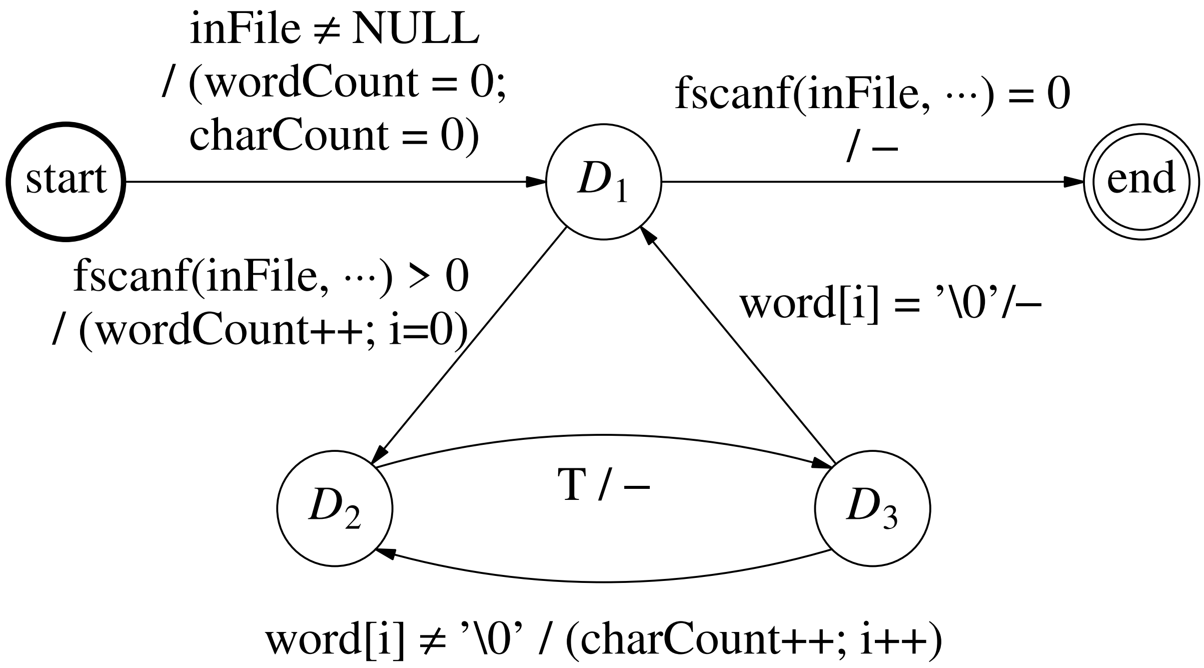
- If $a_{ij}^{(1)} = a_{ij}^{(2)} = a$, then $a_{ij} = a$.
- All other transitions from s_i are disjoint from c_{ij}:
$$c_{ij} \wedge c_{ik} = [c_{ij}^{(1)} \wedge c_{ik}] \vee [c_{ij}^{(2)} \wedge c_{ik}] = \text{False}$$
- c_{ij} together with all other c_{ik} from s_i are complete:
$$c_{ij} \vee [\bigvee_k c_{ik}] = \text{True}$$

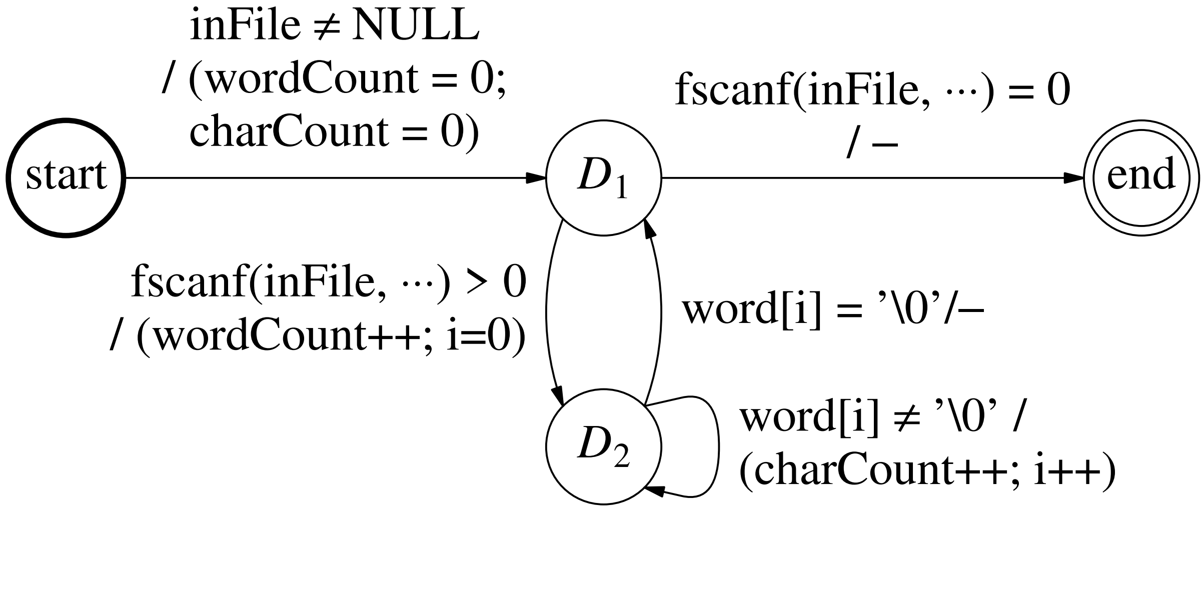
ELIMINATION OF UNUSED TRANSITION





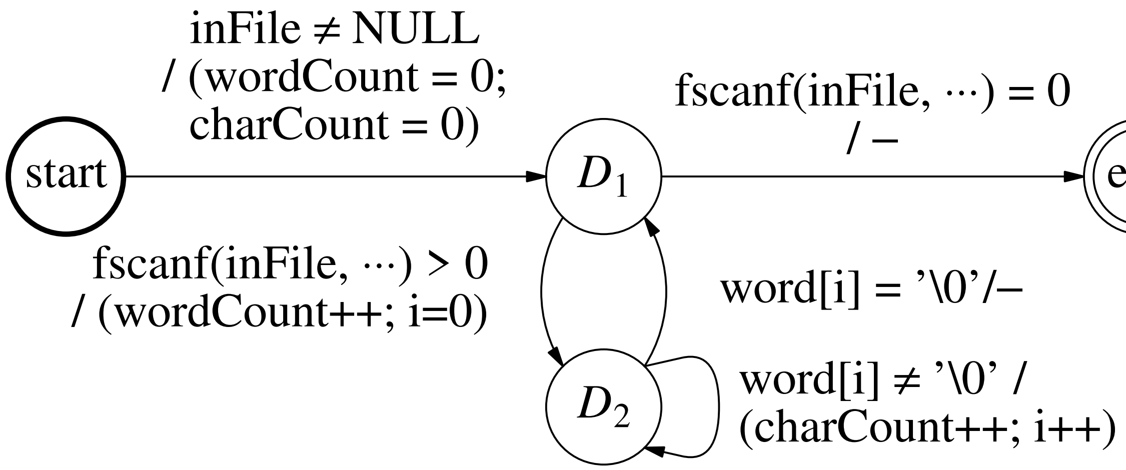
ELIMINATION OF PRECEDENT-DEPENDENT TRANSITION

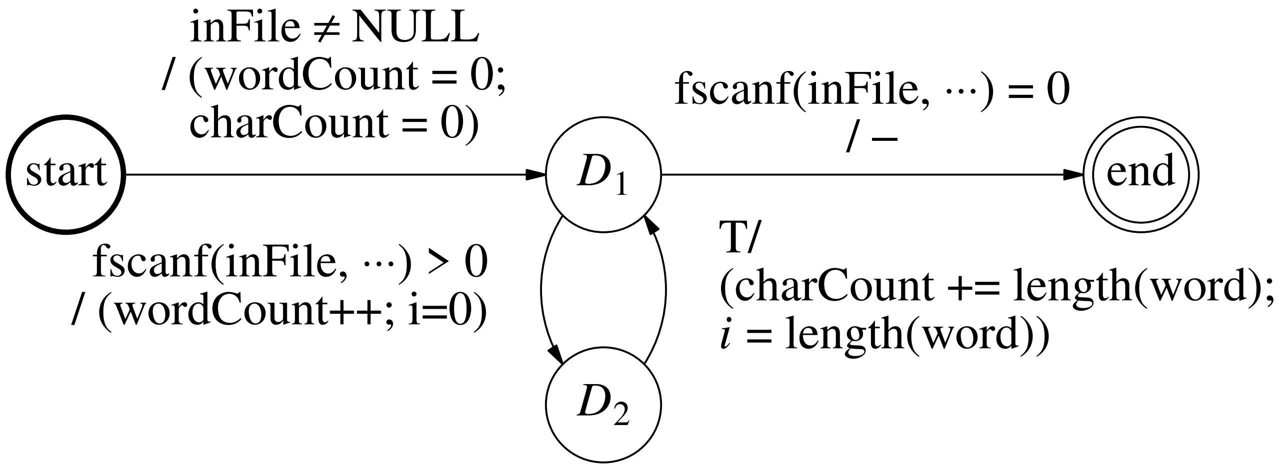


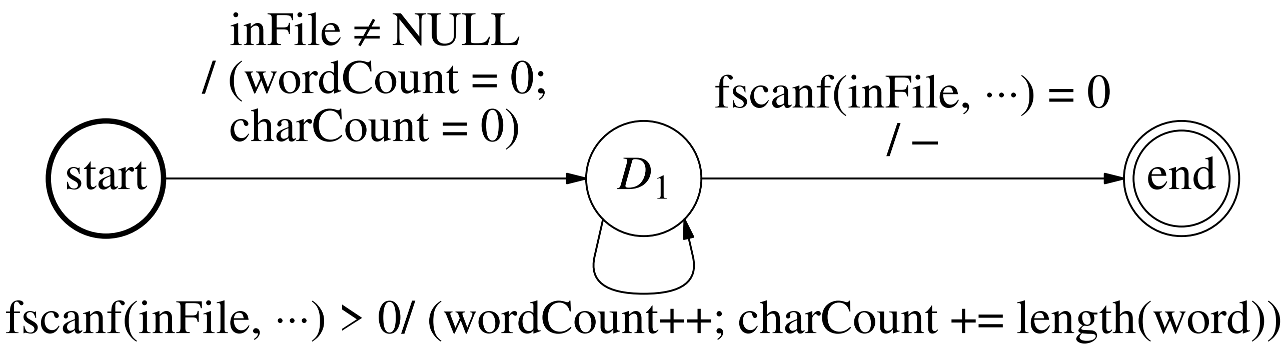


State D_3 is also eliminated.

ELIMINATION OF LOOP







After elimination of i, operations on it, and D_2.

THE NEW PROGRAM

New Program:

```
void WordCharCount(FILE *inFile)
{char word[WORDLEN+1];

 wordCount = charCount = 0;
 while (fscanf(inFile, "%s", word) > 0) {
       wordCount++;
       charCount += length(word);
 }
}
```

The Original Program:

```
void WordCharCount(FILE *inFile)
{int i;
 char word[WORDLEN+1];

 wordCount = charCount = 0;
 while (fscanf(inFile, "%s", word) > 0) {
       wordCount++;
       for (i=0; i<=WORDLEN; i++)
           if (word[i] == '\0') break;
           else charCount++;
 }
}
```

THE PROGRAMS FOR OTHER INTERMEDIATE FSMs

New Program (top FSM on page 25):

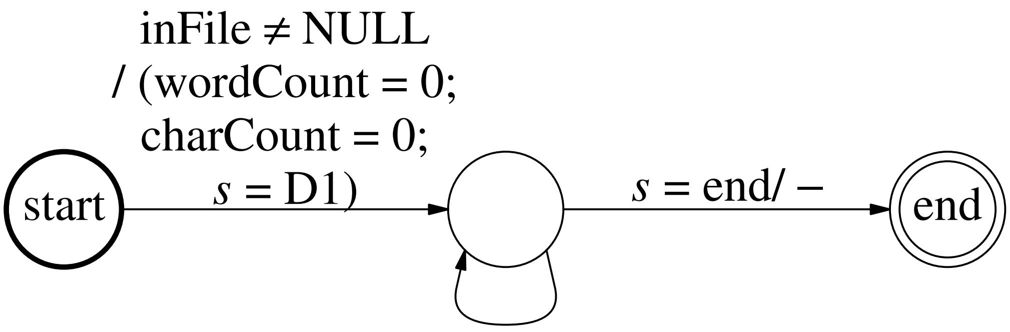
```
void WordCharCount(FILE *inFile)
{char word[WORDLEN+1];

 wordCount = charCount = 0;
 while (fscanf(inFile, "%s", word) > 0) {
       wordCount++;
       for (i=0; word[i]!='\0'; i++)
           charCount++;
 }
}
```

After elimination of succ-dependency of (D_1, D_2) on (D_2, D_2):

```
void WordCharCount(FILE *inFile)
{char word[WORDLEN+1];

 wordCount = charCount = 0;
 while (fscanf(inFile, "%s", word) > 0) {
       wordCount++; charCount++;
       for (i=1; word[i]!='\0'; i++)
           charCount++;
 }
}
```

Keep Program Logic as much clean as possible.

AN EQUIVALENT 3-STATE FSM



D1D2: [s = D1] ∧ [fscanf(inFile, ⋯) > 0]/
(wordCount++; i=0; s = D2)
D2D1: [s = D2] ∧ [$i \not\leq$ WORDLEN] / s = D1
D2D3: [s = D2] ∧ [$i \leq$ WORDLEN] / s = D3
D3D1: [s = D3] ∧ [word[i] = ’\0’] / s = D1
D3D2: [s = D3] ∧ [word[i] ≠ ’\0’]/
(charCount++; i++; s = D2)
D1End: [s = D1] ∧ [fscanf(inFile, ⋯) = 0] /s = end

Remarks:

- The state variable s keeps track of the current-state in the original FSM.
- It has identical computation sequences, save the actions/tests involving s.

– This state reduction does not help understanding/analysis of the FSM.

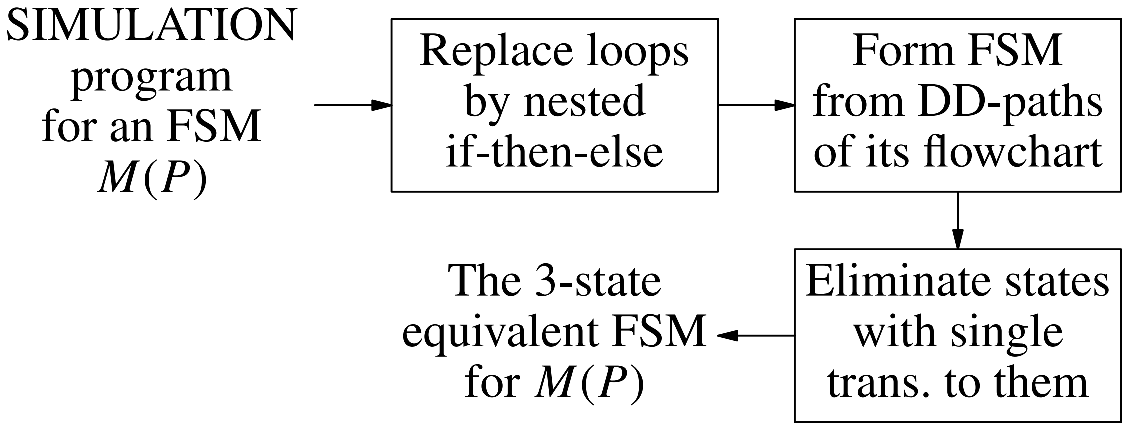
– It is similar to structuring an unstructured code by introducing additional variables and tests.

AUTOMATIC CODE GENERATION

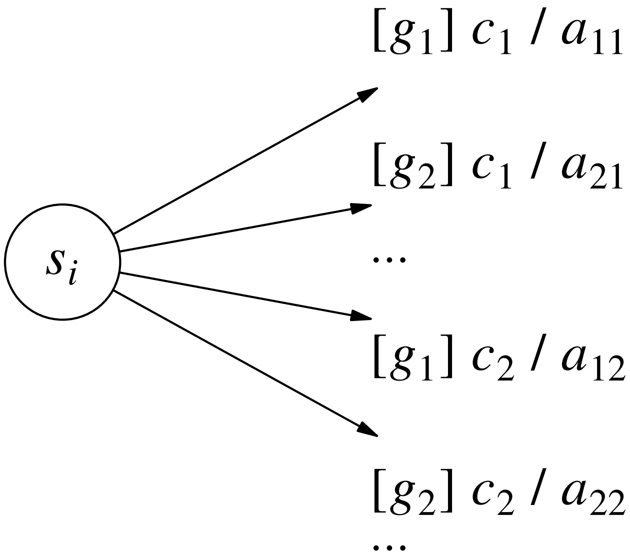
Algorithm FSM-SIMULATOR:

Input: An FSM for a program P and an input data.
Output: The output of P for that input.

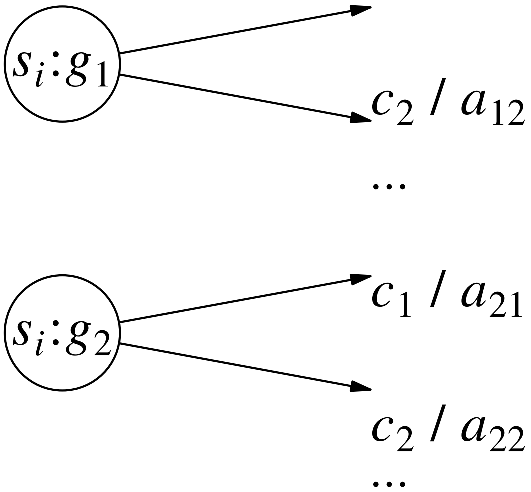
1. Let s = start-state of the finite-state model.
2. Do the actions for the unique transition from start-state and let s be the next state.
3. While ($s \neq$ end-state) do the following:
   For (each state s_i) do the following:
   If ($s = s_i$), then do the following:
   (3a) For (each transition $t(s_i, c_{ij}, a_{ij}, s_j)$ from state s_i) do the following:
   If (c_{ij} holds for the current values of variables), then break for-loop (3a).
   (3b) Do the corresponding action a_{ij}, let $s = s_j$, and break the (outer) for-loop.



WILL THE FOLLOWING METHOD TO ELIMINATE GUARDS WORK



Method: For each guard g_j associated with the transitions from s_i, create a state $s_i{:}g_j$.

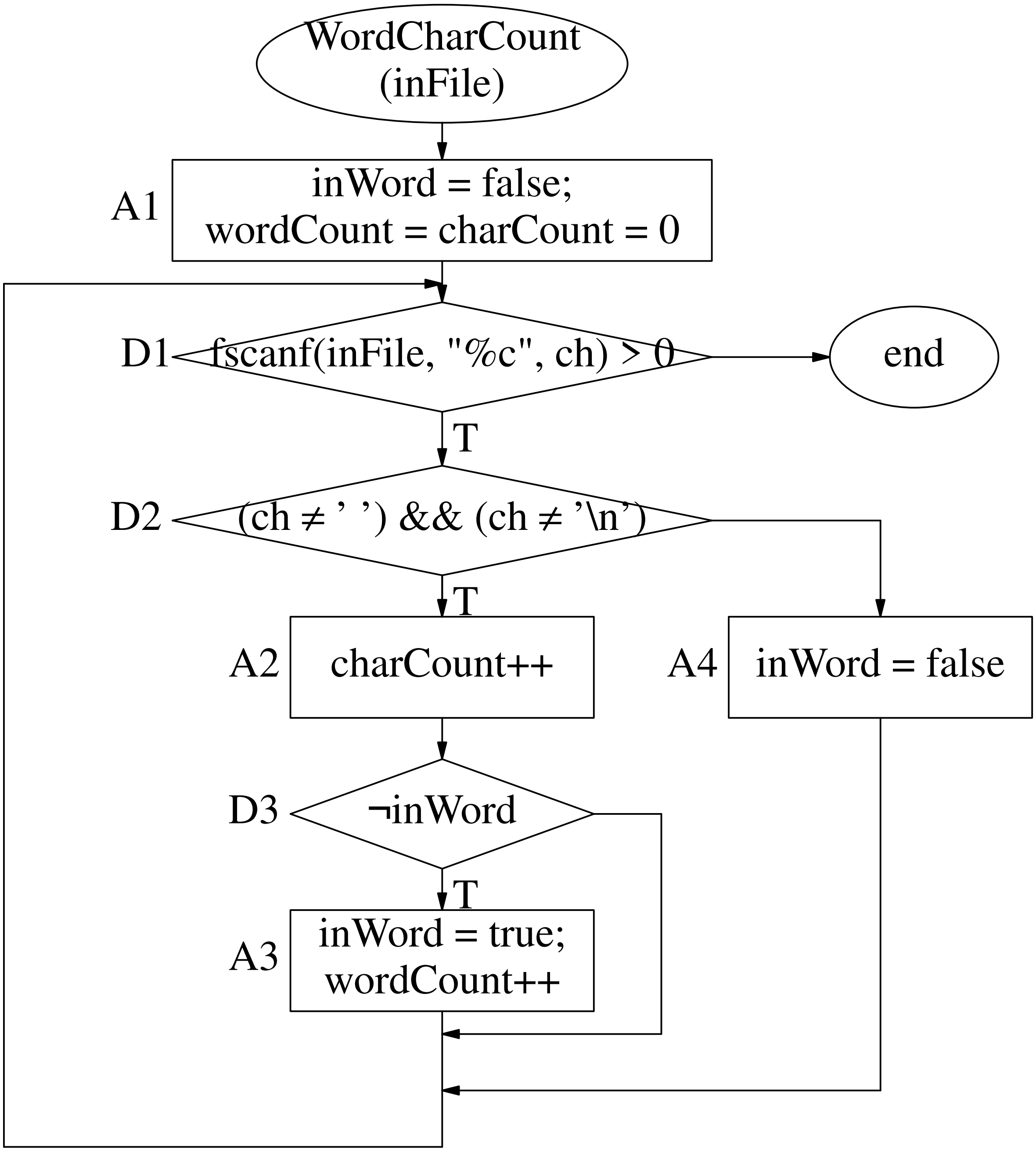


DATA-CENTRIC NOTION

OF

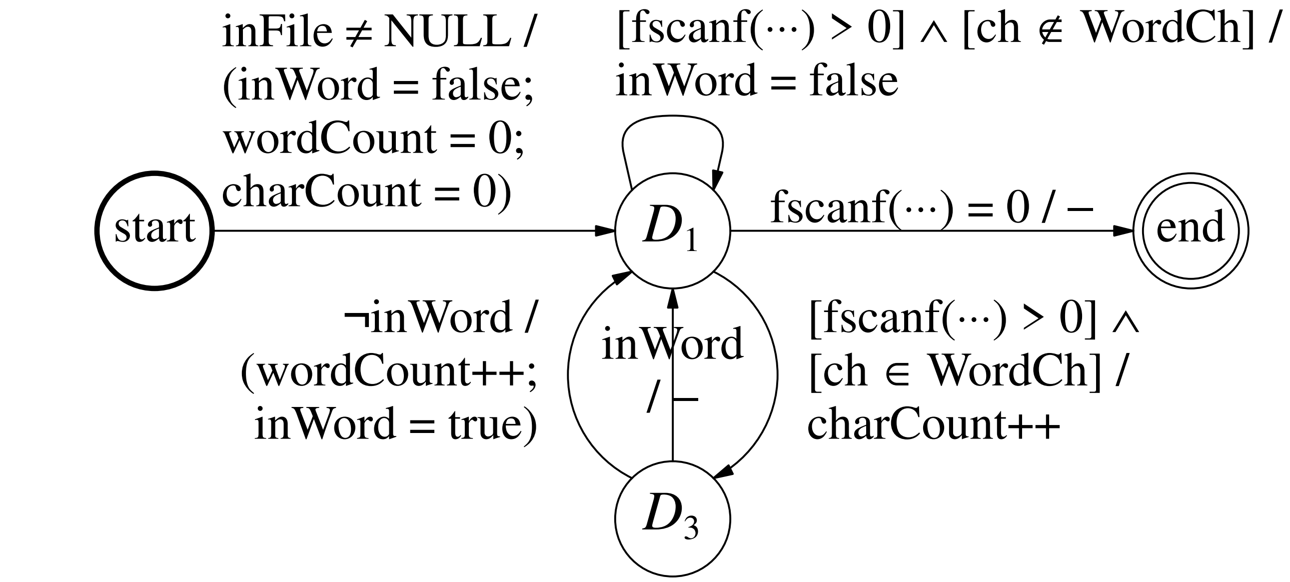
A STATE

FLOWCHART OF A NEW WordCharCount FUNCTION

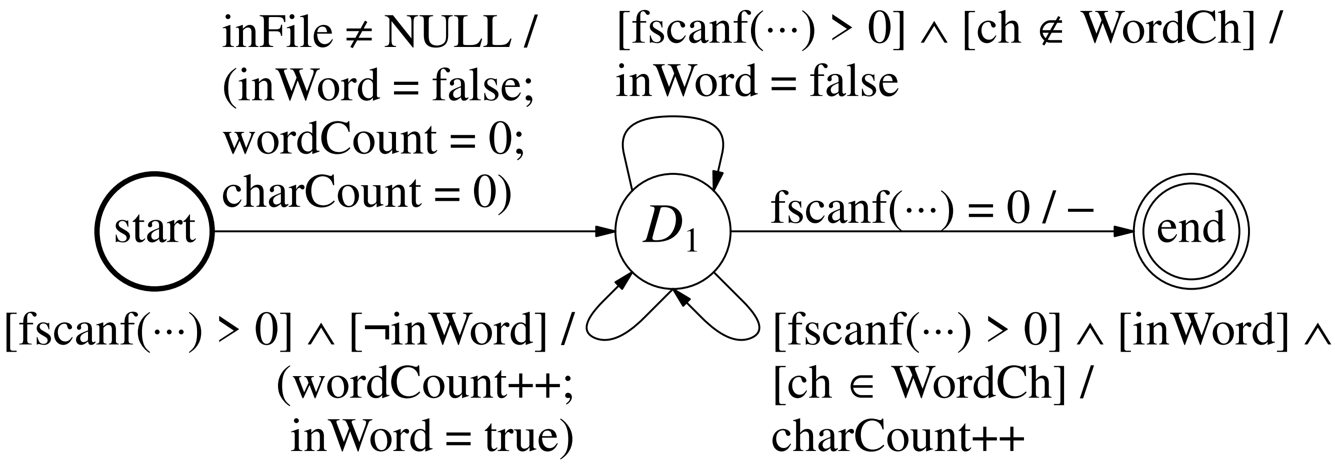


Reads one character at a time.

FSM FOR THE NEW FLOWCHART

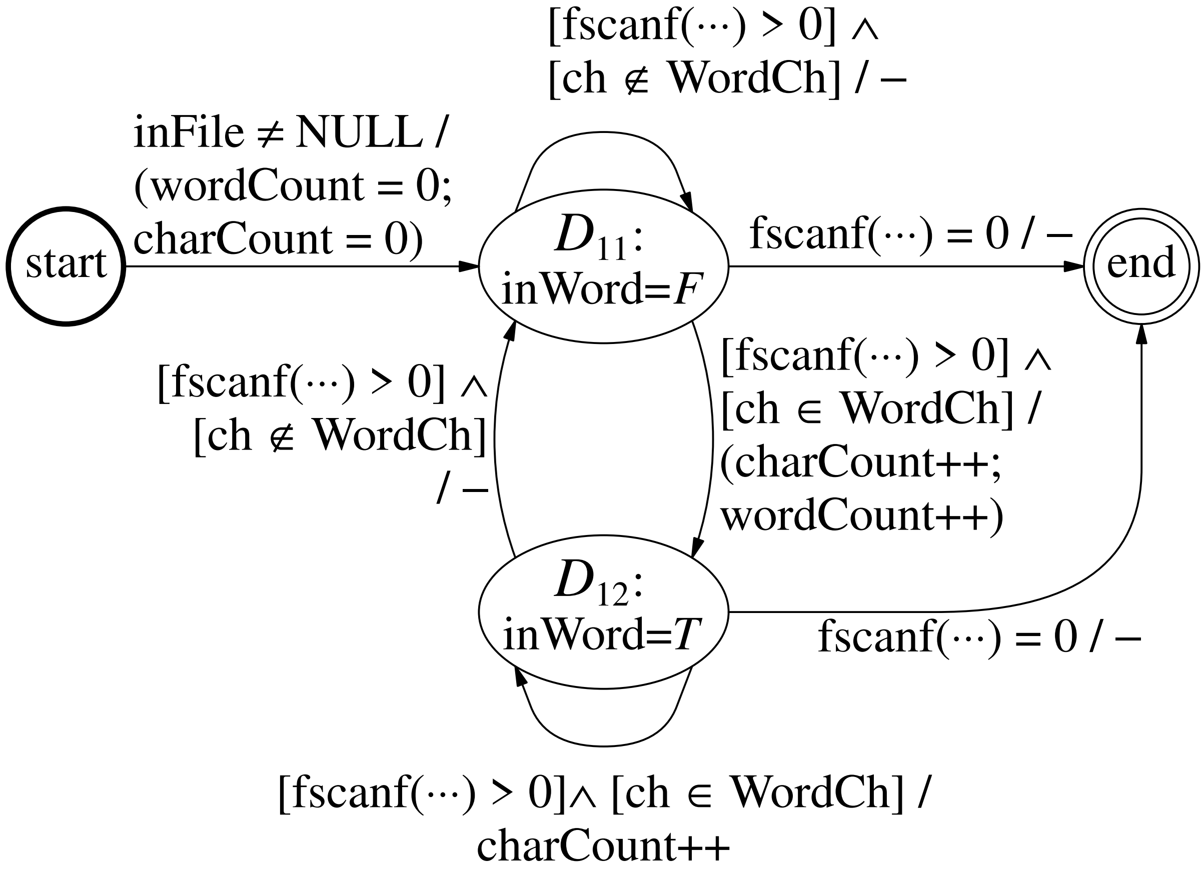


FSM#1: After elimination of the transition for the *dd*-path (D_1, D_2) which has the null-action.



FSM#2: After elimination of transition (D_1, D_3) which precede both the transitions from D_3 to D_1.

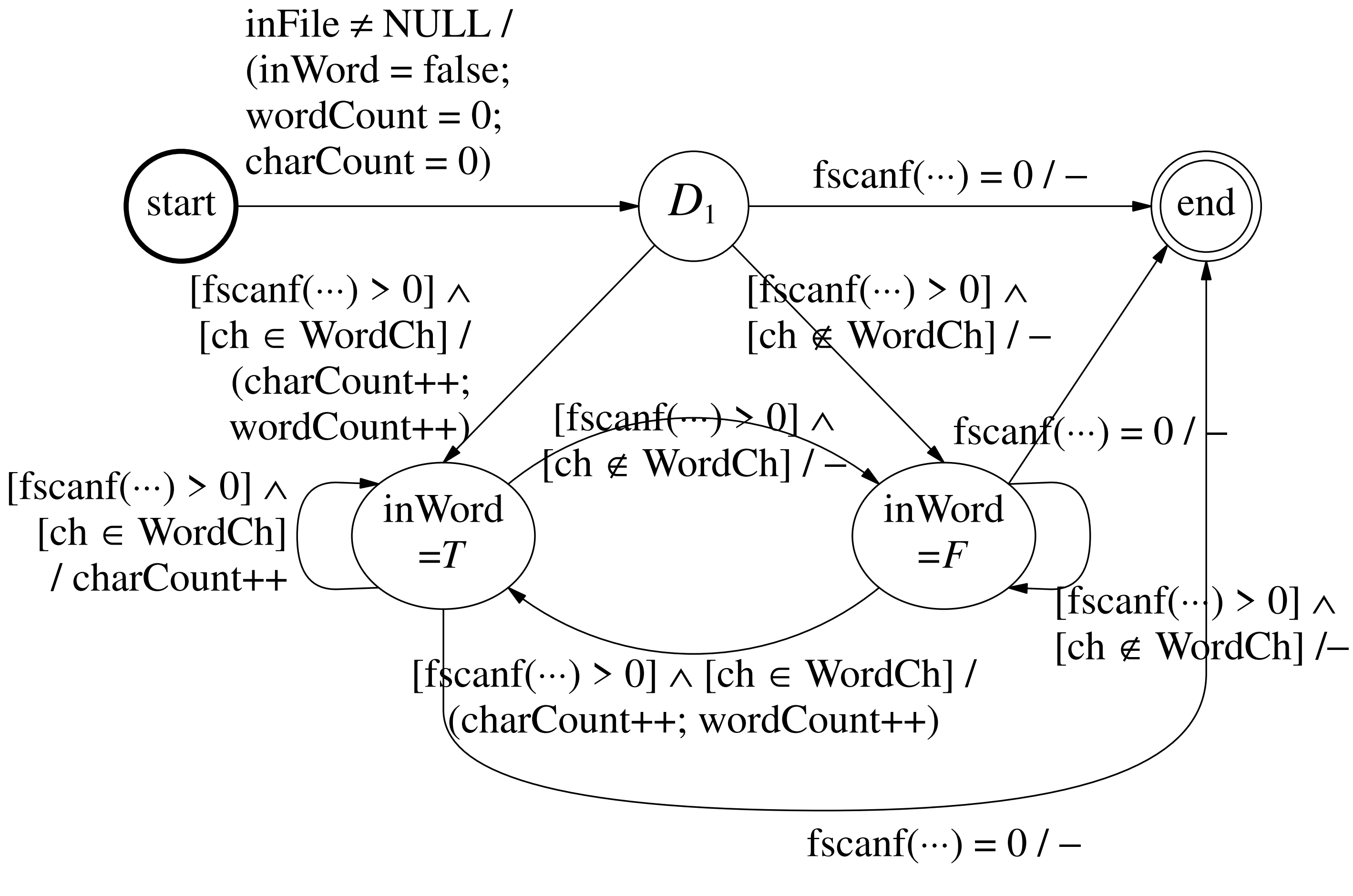
FSM FOR THE NEW FLOWCHART (Contd.)



FSM#3: The result of splitting D_1 into two states based on values of "inWord".

Question: Is it possible to eliminate the loop at D_{12}? If so, what is the new FSM?

FSM FOR THE NEW FLOWCHART (Contd.)



FSM#4: The result of splitting D_3 into two states based on values of "inWord".

A PROGRAM FOR THE MODEL FSM#3

```
//inFile ≠ NULL
void WordCharCount(FILE *inFile)
{char ch;
 wordCount = charCount = 0;

 while (fscanf(inFile, "%c", ch) > 0)
       if ((ch != ' ') && (ch != '\n')) {
          charCount++;
          wordCount++;
          //read 2nd and other chars of word
          while (fscanf(inFile, "%c", ch) > 0)
                if ((ch != ' ') && (ch != '\n'))
                   charCount++;
                else break;
       }
}
```

Remarks:

- The role of artificial variable "inWord" is replaced by the inner while-loop. (The effect of artificial variables can be just as bad as goto's.)
- The extra-loop is not to be considered an increase in complexity.
- The new code has fewer dependencies among the different parts.

FROM REQUIREMENTS

TO

FINITE-STATE MODEL

DESIGNING A FINITE-STATE MODEL FROM REQUIREMENTS

- Create the first few levels of the behavior-tree.

  Paths in behavior-tree = Possible "scenarios"

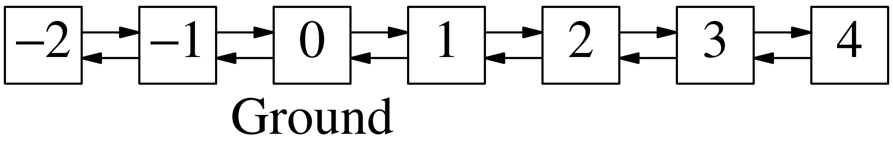
- Analyze the behavior-tree.
  - Determine control information needed to distinguish the branches/links at a node.
  - Determine the actual conditions for the links.
  - Determine other variables related to actions.

    Some actions may involve acquiring new inputs and updating control information.

- Determine states and transitions.
- Verify the design.
  - With additional example scenarios.
  - Rebuilding part of the behavior-tree from the model and comparing with the original tree.

Remark: One may initially consider only some of the requirements, and then extend the model to include other requirements.

A COMPLEX DESIGN PROBLEM: AN ELEVATOR CONTROLLER



Requirements:

(1) The elevator always stops at floor #0 on each upward/downward movement.

(2) A customer visits only one destination floor, which is fixed when boarding the elevator at floor #0, before returning to floor #0; no floor-button inside elevator.

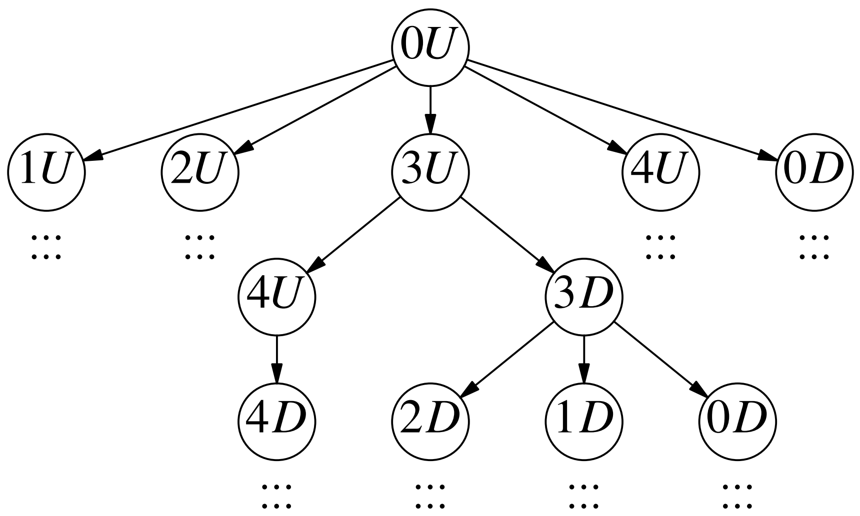
(3) To return from floor #k, $k > 0$, to floor #0, a customer enters elevator at floor #k only when the elevator is on a downward trip; similarly for $k < 0$.

(4) The elevator changes move-direction from UP to DN at floor #k, $k > 0$, only if there are no pending exit/entry-requests for floors above k; similarly for move-direction change from DN to UP.

(5) The elevator has unlimited capacity and there are no special service-priority for any floor.

Remark: Removing the restriction (2) is a future consideration.

BEHAVIOR-TREE



We pretend U = initial move-direction at floor #0.

- Node label = Label of the link to node (except root).
- The move (iU, iD) indicates change of direction.
- Without floor #, it is not possible to determine conditions for parent-to-child links (U/D-part of a node-label is redundant).

Condition for ($0U$, $3U$):

(1) There are no exit-requests for floors #1, #2 (there may be entry request for floors #1, #2), AND

(2) [There is either an exit-request for floor #3] OR [there are no exit-requests for floor #3 and floor #4, there is no entry-request for floor #4, and there is an entry-request for floor #3].

CONTROL-VARIABLES AND ACTIONS

Control Vars:

(1) Presence of entry/exit requests for each floor #k.

- entry-requests (onward journeys) at floor #0

  entry-requests (return journeys) at floor #k, $k \neq 0$
- exit-requests of customers inside the elevator for floor #k, $k \neq 0$ (onward journeys)

  exit-requests of customers inside the elevator for floor #0 (return journeys)

(2) Current move-direction and floor#.

This is all that is needed from the past to determine future behavior, which also depends on future requests.

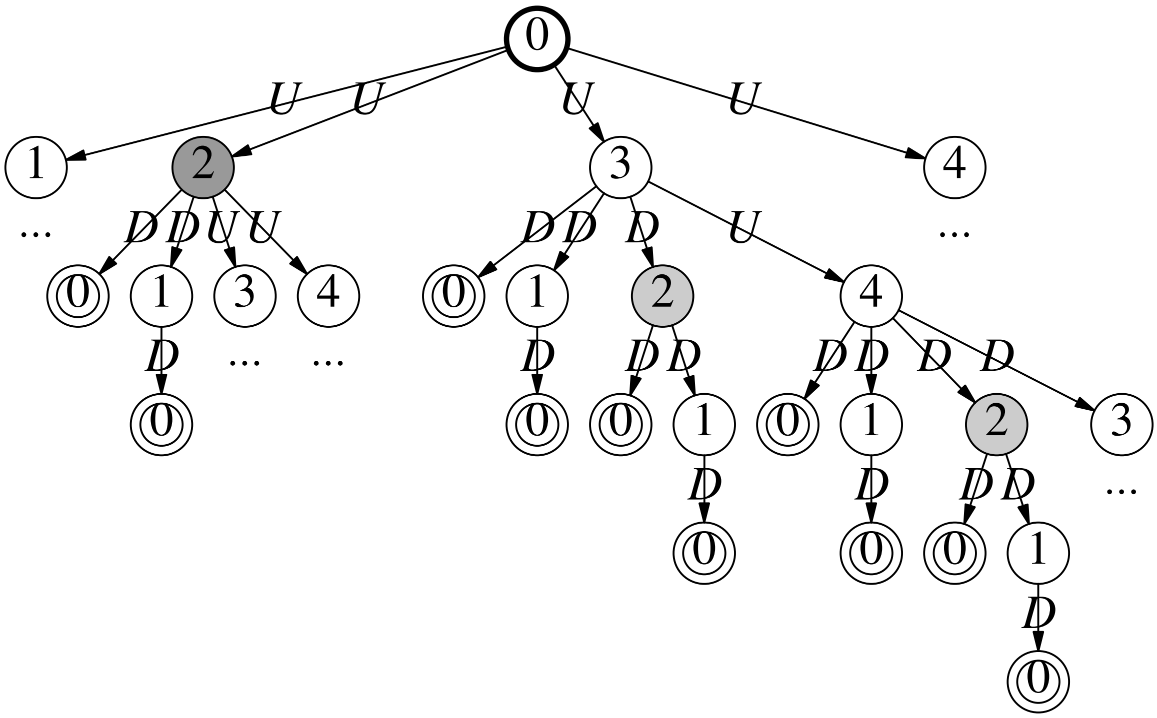
Actions:

(1) Unloading/Loading of customers and updating pending exit/entry requests (update takes place only on stopping at a floor).

(2) Determining move-direction and next-stop.

PROBLEM IN USING ONLY *U/D*

Trips from floor #0 to floor #0 starting with *U*

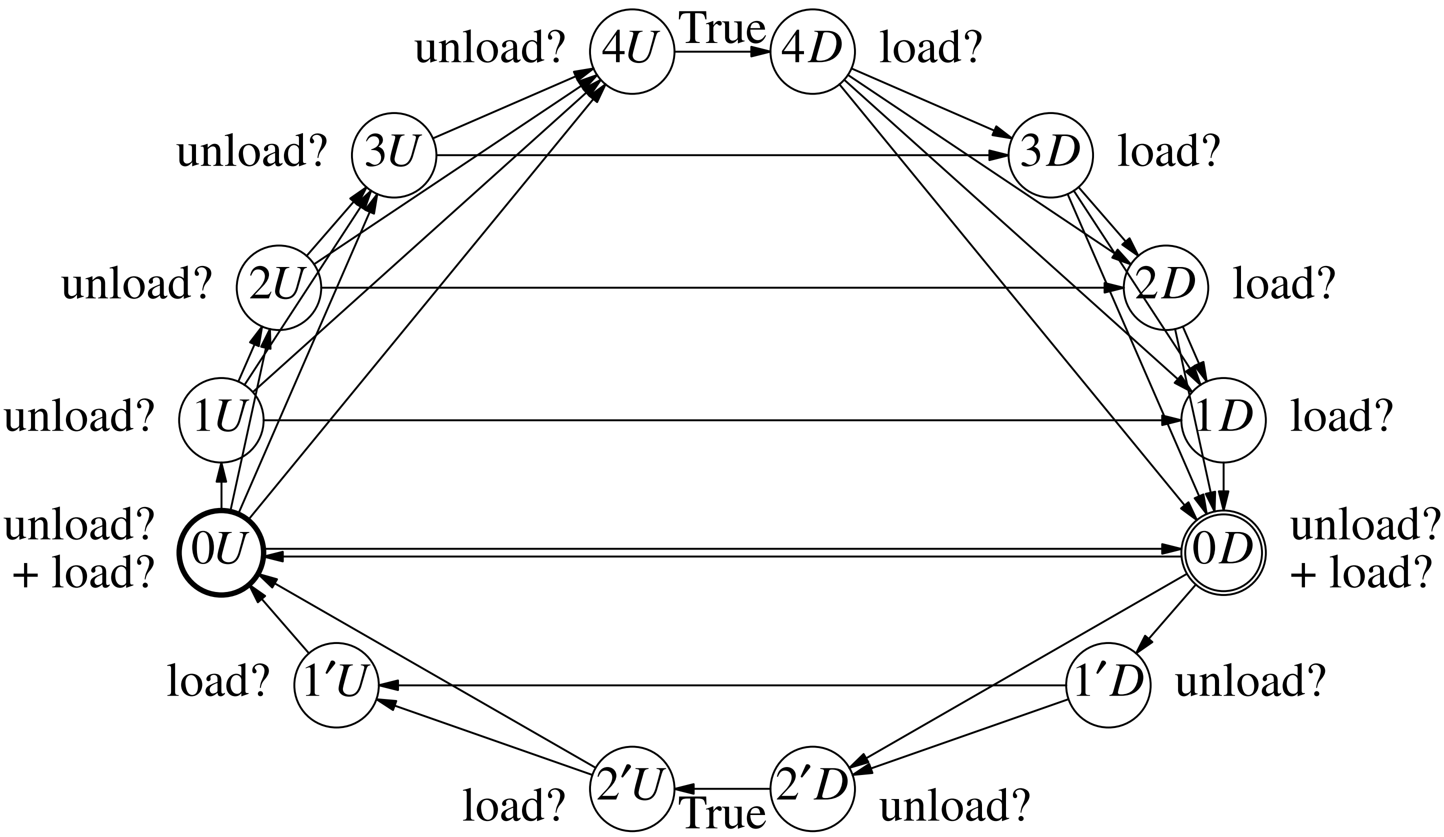


Not all nodes marked 2 are equivalent.
Valid sequences: $U^m D^n$, $1 \leq m \leq 4$ and $1 \leq n \leq 4$.

- We cannot determine the conditions to distinguish the *U*-links from root without the children's floor #.
  - A child's floor # cannot be determined from the path to the child (can be determined from the longest path with *D*-links from the child).
- Similarly for other nodes with > 1 children.

Conclusion: Need floor # as additional information in modeling the behavior and the states.

THE ELEVATOR-FSM



Remarks:

(1) All transitions from a state have the same actions, and hence they are assigned to the state itself.

(2) The unload-operation at a state may or may not occur. Similarly for load-operation.

(3) Easy to eliminate transition $(4U, 4D)$ and state $4D$.

(4) Similar elimination of state $3D$ makes new transitions overly complicated and hides important actions.

STATE = (FLOOR #, MOVE-DIRECTION)

- This still gives finite number of states.
- The start-state = (0, U).

Control Information in State 2U:

- Exit-requests for floor #3 and floor #4.

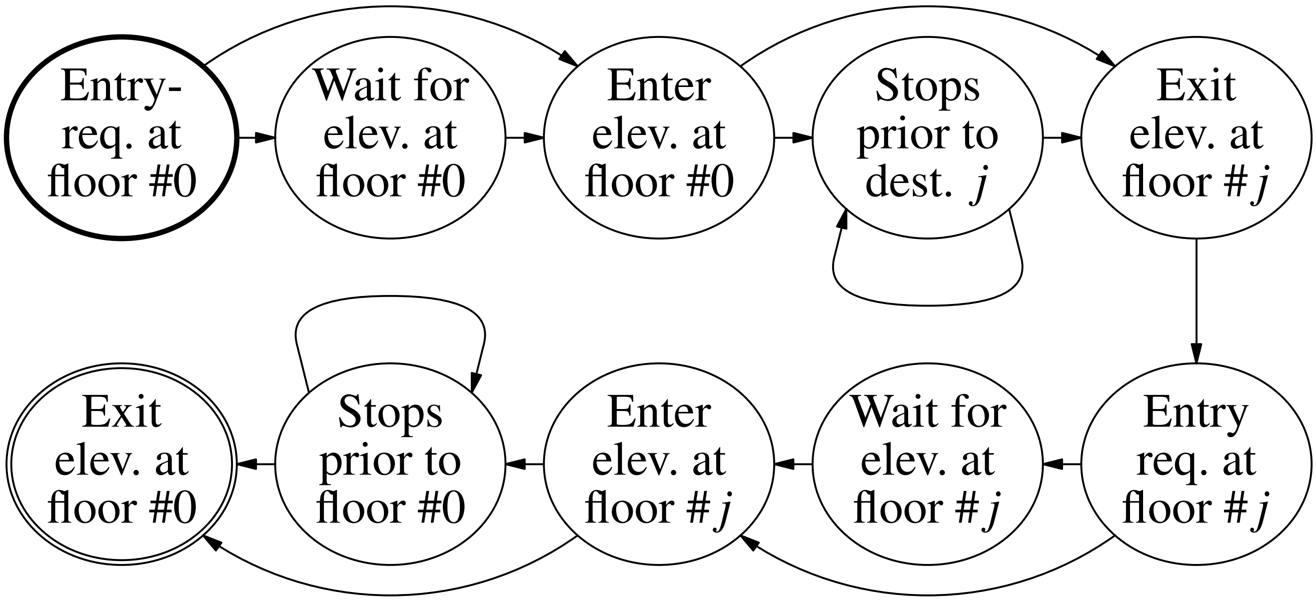
  There are no exit-requests for floor #k, $k < 2$.

Operational Information in State 2U:

- Entry-requests for floor #k, $k \leq 2$ or $k > 2$; these may influence future control information.

Different states use different information for control and action.

FSM FOR A SUBSYSTEM FROM THE SYSTEM-FSM



The FSM of the elevator
from an individual's point of view.

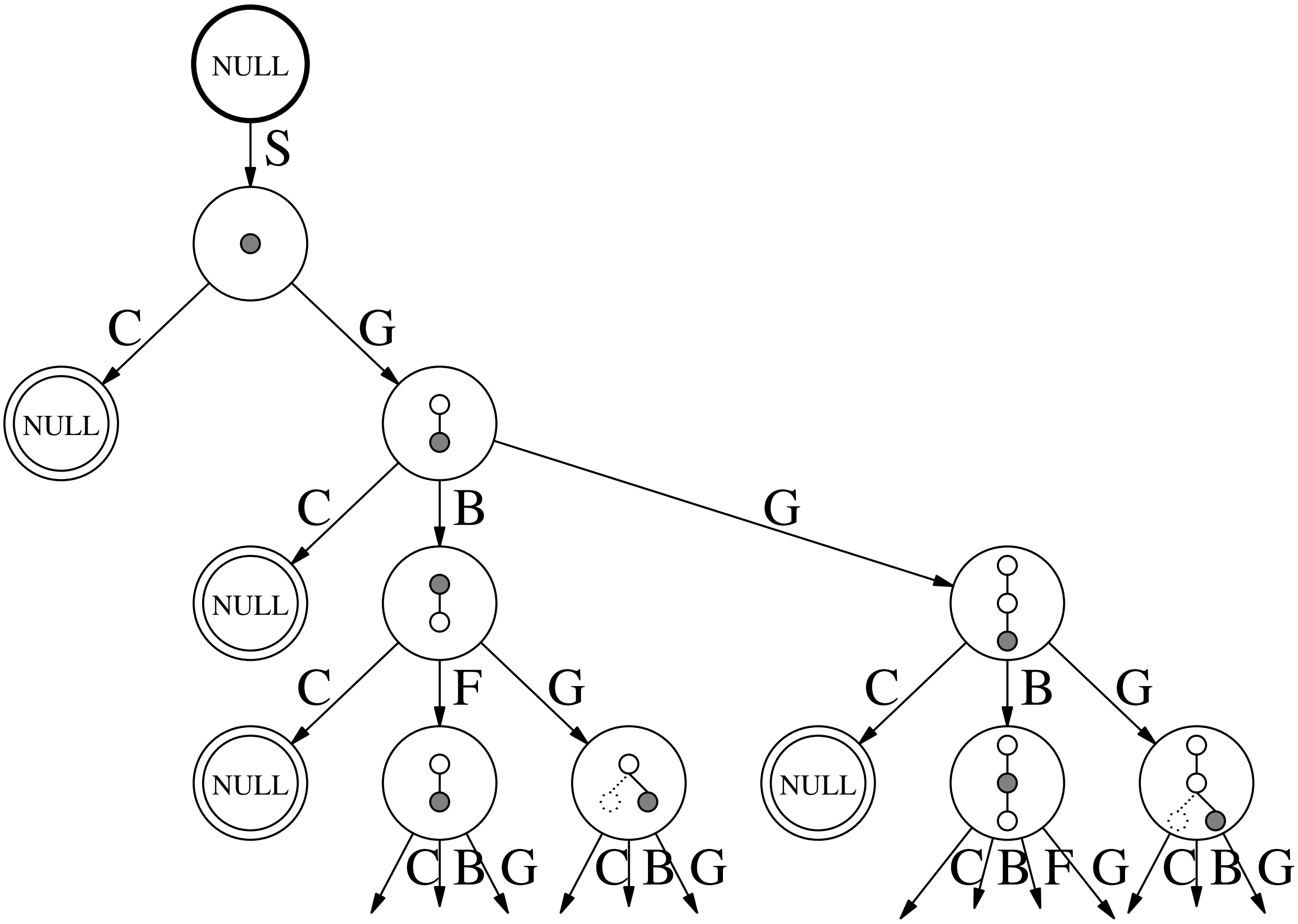
ANOTHER LOOK AT

BEHAVIOR-TREE FOR WEB-NAVIGATION

AND ITS

FINITE-STATE MODEL

BEHAVIOR-TREE FOR ACTIVITIES IN WEB-NAVIGATION



C = Close, B = Back, F = Forward, G = Goto

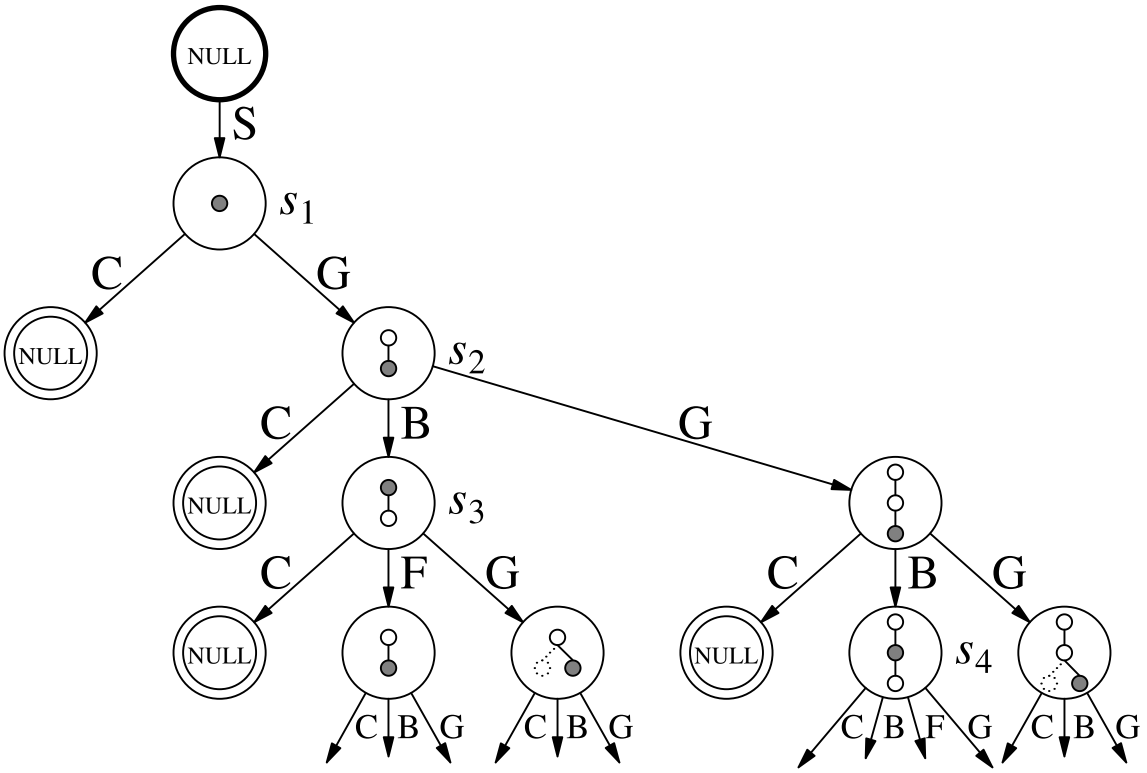
- Shown at each node of the behavior-tree is the associated navigation-path data.
- The solid node in the navigation-path is the current node.

FROM BEHAVIOR-TREE TO FSM

Method:

(1) Introduce condition-guards, with supporting data, to make two nodes process the same input-information in the same way.

(2) Merge two nodes if they have identical subtrees.



- Guard [hasParent(curPosition)] for processing "Back" equates links at s_1 and s_2.
- Guard [hasChild(curPosition)] for processing "Forward" equates links at s_3 and s_4.
- This also makes the subtrees at node s_1 and s_2 identical and merging them gives the FSM obtained earlier.

NAVIGATION-HISTORY DATA ABSTRACTION TO CREATE FSM-STATES

The (finite) past completely determines all future possibilities.

Equivalent Nodes:

Nodes s_i and s_j in a behavior tree are equivalent if the subtrees at them are equivalent, except for the missing links.

- #(equivalent classes of nodes) must be finite.

Navigation History Data for a Node:

Abstraction of the path from root to the node, and hence consists of a finite amount of information.

- This node-information, including conditions for the missing links, must be described in terms of a finite number of vars and relationships (e.g., child, parent).
- The history data (involving lists, trees, etc.) for equivalent nodes need not be *bounded*.
- Equivalent nodes may have different abstractions.

BUILDING PATH-ABSTRACTIONS TO CREATE FSM-STATES

(1) For each action at each node in behavior-tree, determine the *control* information needed to decide its applicability and the (additional) *operational* inputs needed to carry out the action.

- Control inputs may come from:
  - information at ancestor nodes.
- Operational inputs may come from:
  - information at ancestor nodes.
  - fresh external inputs acquired at the node.

(2) If certain information about the path grows without bounds, keep such information in the form of values of variables, which may take the form of lists/trees.

- It must be possible to update these variables and data-structures as we move from a node of behavior-tree to its children.

(3) Use other information, for which there are only finite number of alternatives, to distinguish nodes in the behavior-tree, giving rise to different states.

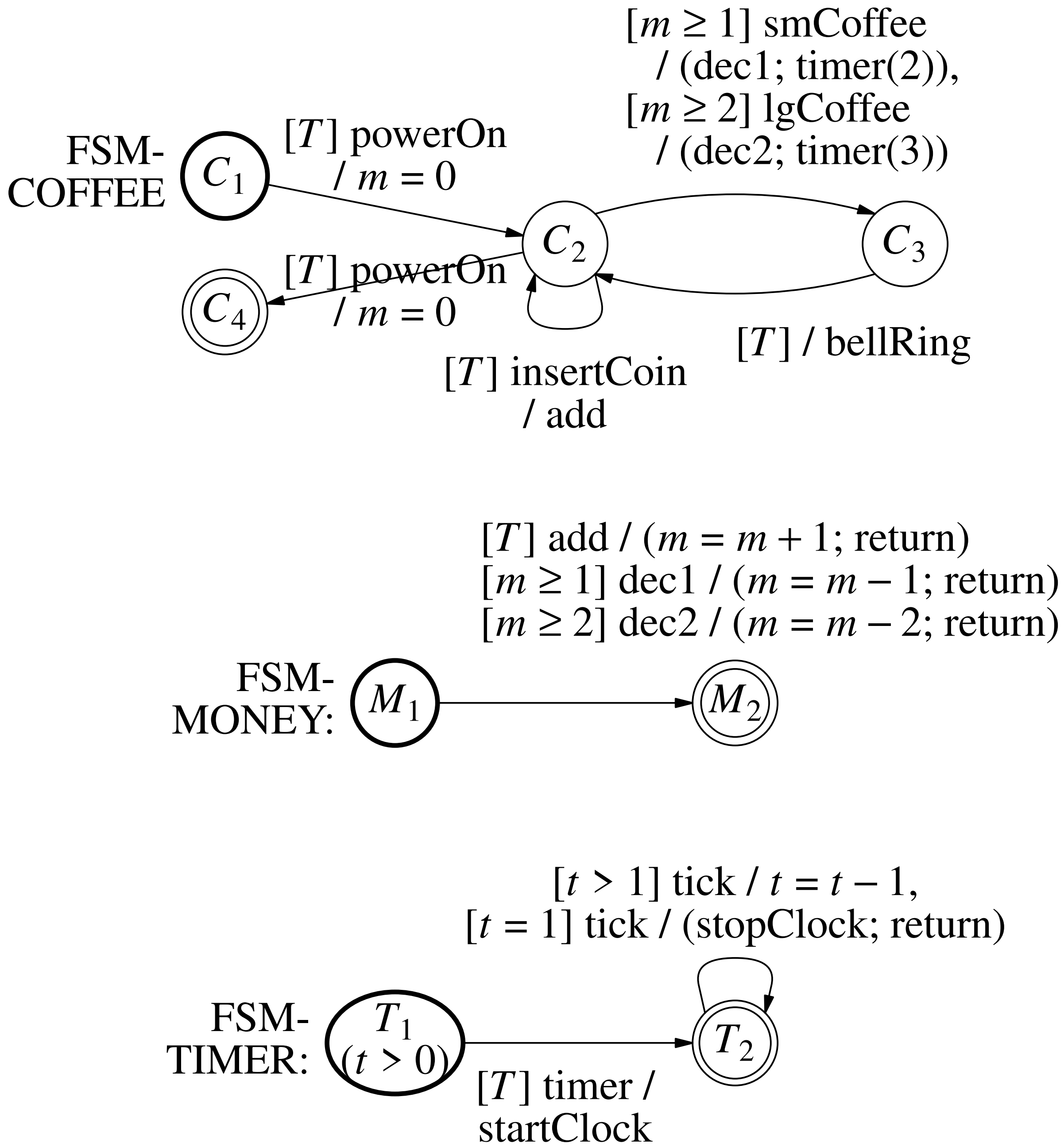
- Such information will not appear in condition-guards and is not control information.

A FAMILY OF INTERACTING FSMS

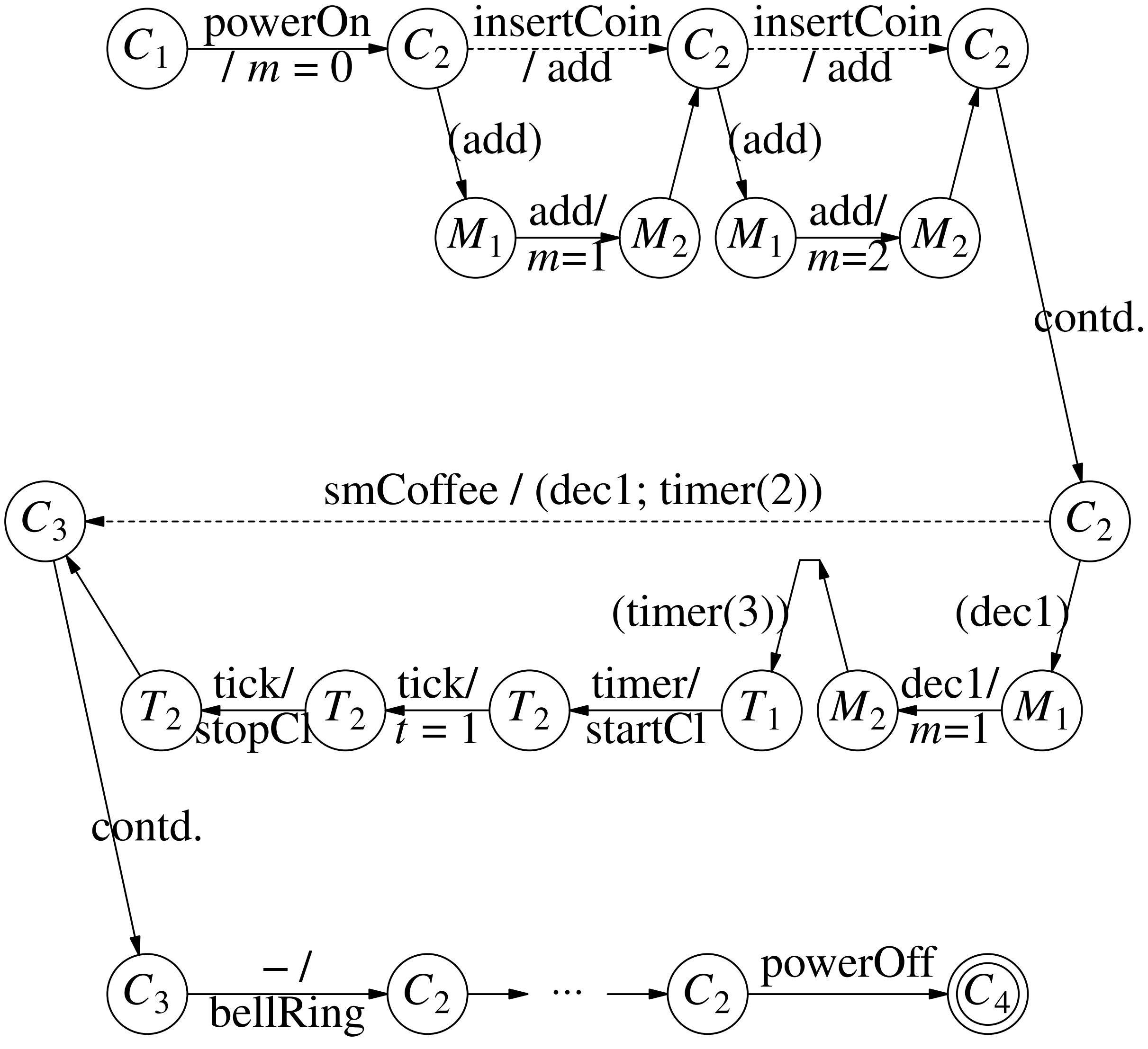
Advantages:

- Models function-calls in a program, including recursion.
- Keeps the size of FSM small.
- Helps modular design.

THREE INTERACTING FSMs



AN EXAMPLE BEHAVIOR-SEQUENCE WITH CALLS AMONG THE FSMs



CONCLUSION

Design your software with FSM.

- Build your FSM from requirements.
  - Identify behavior-tree upto a certain level so that each node at that level is equivalent to some other node.
  - Identify control and other operational data.
  - Identify conditions for each link in behavior tree.
  - Build your states and transitions.
  - Simplify the model, if necessary.
- Build your software from FSM.
  - by a semi-automatic code generation scheme, or
  - by a direct implementation.
- Advantages
  - More reliable software