FINITE-STATE MODEL OF OPERATIONS IN A SOFTWARE

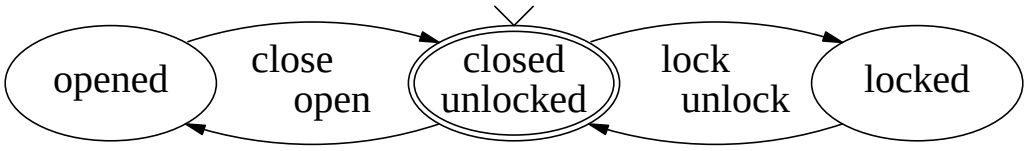
Why Make a Finite-State Model (FSM):

- It shows which high-level, user-operations can apply following other operations.
- A finite-state model of a program P is an abstraction of it.
  - It highlights the operations and their valid application sequences.
  - It is programming language independent.
  - One can generate semi-automated code from the finite-state model (FSM).
  - One can easily build subsystem-models and higher level (architecture) models from FSM.
- In practice, we build the model $M(P)$ of P first and then build the software P.
  - Converting a P to an $M(P)$ can be automated.
  - Building an FSM from the problem-statement or requirements (without having the software), *cannot* be fully automated, and remains very much a human-centered activity.
    + The requirements use terms/concepts that cannot be automatically converted to code.
    + The requirements must be, however, stated in a form and with sufficient details to allow us build an FSM (not necessarily automatically though).

FINITE-STATE MODEL FOR A DOOR WITH LOCK

Assumptions: Four operations: open, close, lock, unlock.

- Door can be opened if it is closed and unlocked, and it can be closed if it is opened.
- Door can be locked only if it is closed and unlocked, and it can be unlocked only if it is locked.
- Initially, the door is closed and unlocked.

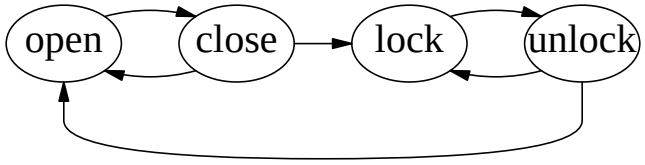


The meaning of states determines the transitions among them.

Question:

•? What would be the model if there are two locks and they can be locked and unlocked in either order?

•? What is wrong with the model below, which has more states and links? What does an arrow indicate here? Can we take "open" and "lock" as start-states and "close" and "unlock" as final-states?

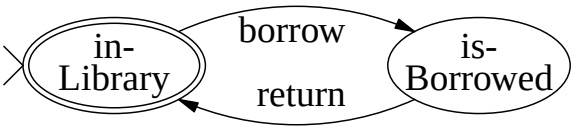


Two reasons for not using this kind of model is that typically, it will have: (1) multiple states with the same operation-name, and (2) more states and links, making the diagram more complex.

FINITE-STATE MODEL OF A LIBRARY BOOK

A Simple Case: Two operations "borrowed" and "returned".

- The FSM shows that the two operations must alternate.
- The start-operation is borrow (why?) and we assume that a book once borrowed will be returned at some point.



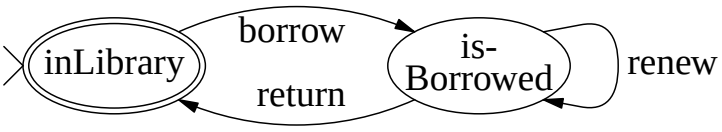
- Shows the applicable operations in each state.
- We need two states (why?).

Question:

•? State another situation where we have two operations "b..." and "r..." which also alternate but the starting operation is "r...".

More Complex FSM: Adding renew-operation.

- A book can be renewed only if it is borrowed but not yet returned.
- There is no limit on the number of renewals.



Questions

•? What is the similarity between return and renew operations? What distinguishes them and how is it reflected in the FSM? Do you see any shortcoming in this model?

•? Show the new FSM if we assume that a book can be renewed at most 2 times. (Is there a need for such a restriction?)

•? How to model the fact that the person borrowing the book is the person renewing it? (Is this restriction necessary?)

MODEL-BASED MENU FOR DISPLAYING APPLICABLE OPERATIONS

Example of Applicable Operations for Different Books:

Book	Borrow	Return	Renew
Book #1	×		
Book #2	×		
Book #3		×	×
...	...	...	

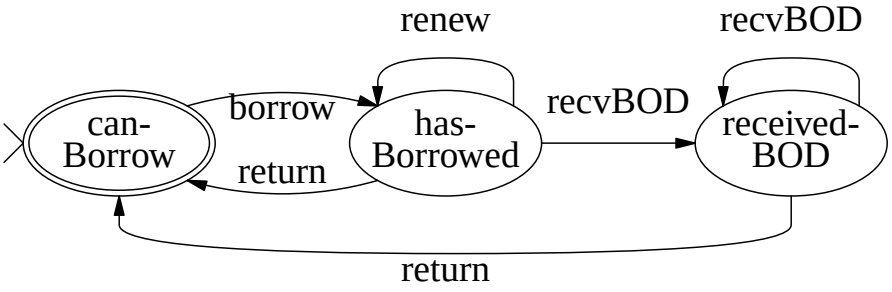
General Case:

- We can create a menu of applicable operations based on the current "state" of software.
- Such a menu would be less cluttered and therefore more user-friendly.
- This is particularly important when there are many applicable operations but only a few apply at any time (i.e., at any state).

FINITE-STATE MODEL OF A PERSON FOR LIBRARY EXAMPLE

Assumptions:

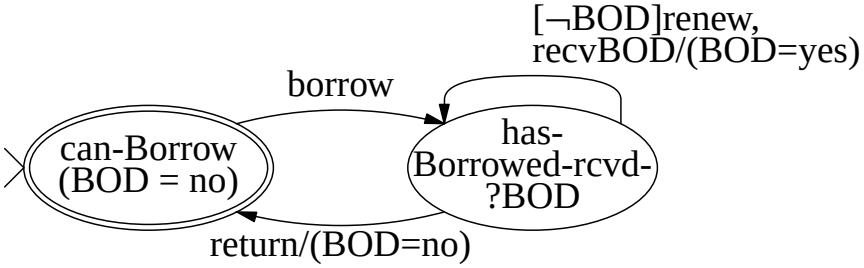
- At most one book may be borrowed at any time.
- If a person doesn't return a book by the due date, then he is sent a book-overdue notice (BOD) periodically till the book is returned.
- A person can return a borrowed book; it can be also renewed if no BOD-notice has been sent (= received).



This is also a model for a book (why?) with 5 operations borrow, return, etc

A Simplified Form Using Guards:

- Initially, variable BOD = "no"; not modifi ed by borrow-operation.
- The recvBOD-operation sets it to 'yes' and the return-operation resets it 'no'.



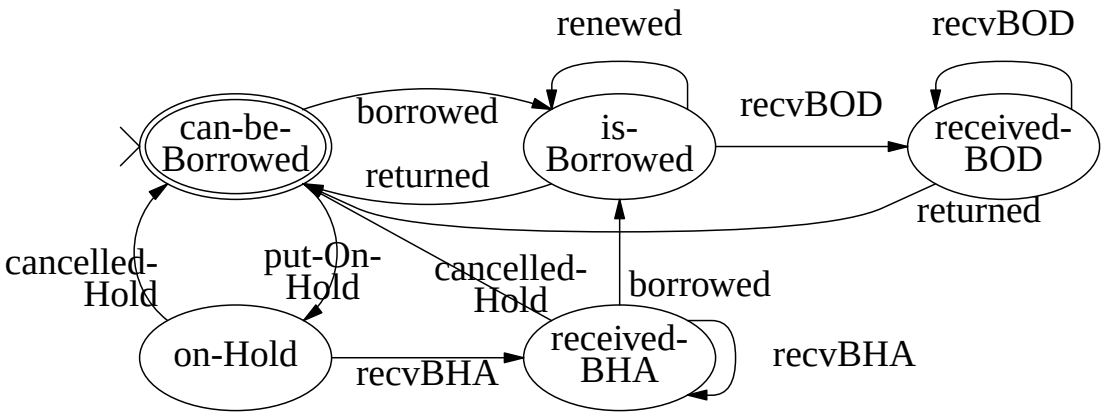
EXERCISE

1. Add another variable to create a model with just one state.

EXERCISE

1. Modify the FSM of a person for the additional assumptions:
   (a) The person can put a hold on a book (borrowed by someone else) if he does not have another book borrowed.
   (b) He can put at most one book on hold at any time (just as he can borrow at one book at a time).
   (c) He may borrow the book on which he has put a hold after he gets a Book-on-Hold-Available (BHA) notice. He may cancel the hold after or before getting BHA-notice.
   (d) The BHA-notice is sent periodically until the person borrows the book or cancels the "hold". (Assume no limit on the number of BHA-notice.)

2. Shown below is a finite-state model of a book which allows hold-operation and BHA-notice-operation. State all restrictions implied by this model. Why should we not merge the states "on-Hold' and "received-BHA"?

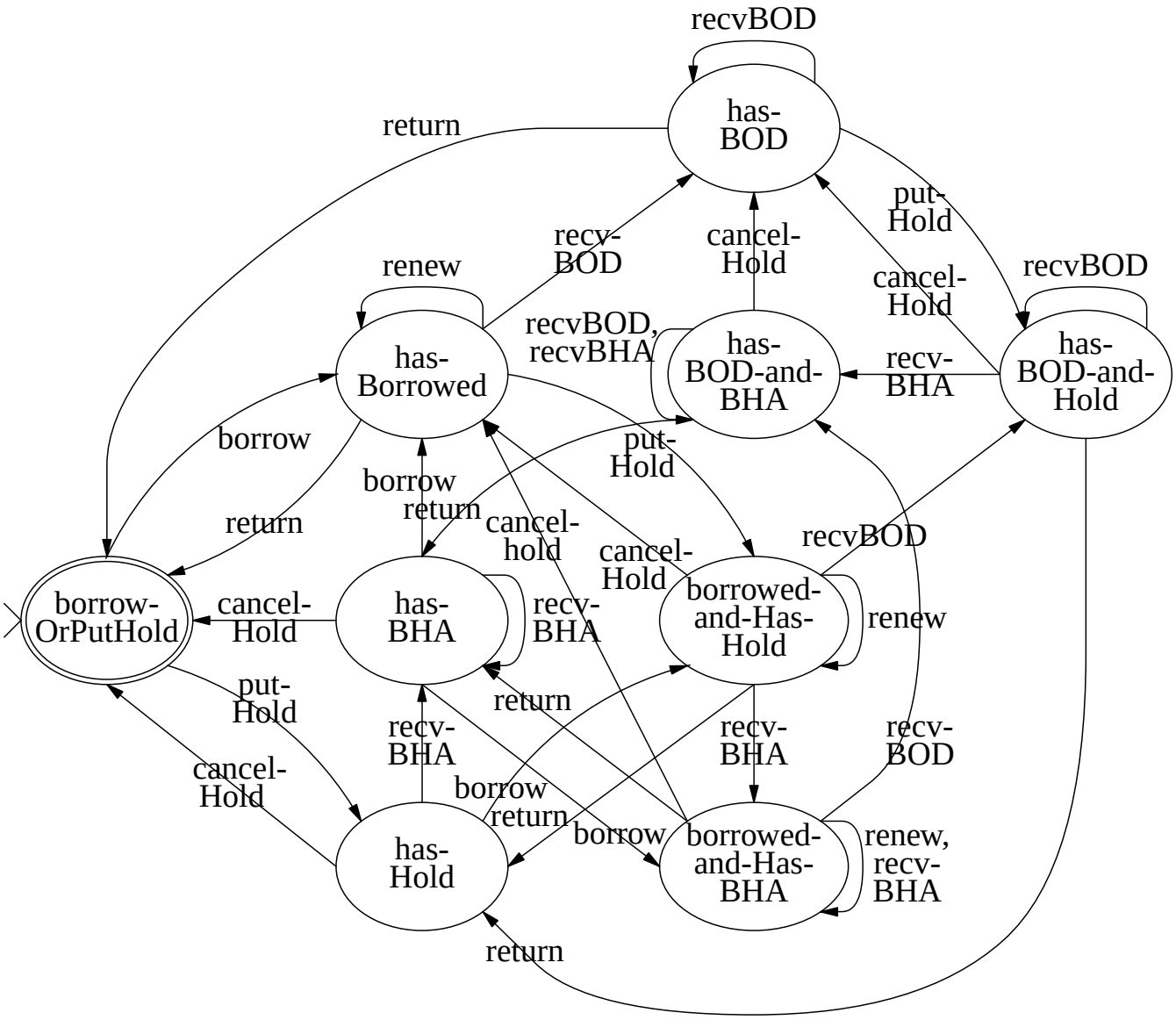


Generalize this model when a book may also be put on hold if it is borrowed and may or maynot have received BOD. (The person putting the hold must be different from the one having the book.) Can we now merge the states "on-Hold' and "received-BHA"? Show the new model after merging.

A MORE COMPLEX MODEL OF A PERSON FOR THE LIBRARY EXAMPLE

A More General Case:

- A person can borrow at most one book at a time and place an hold on at most one book other than the one he has borrowed.
- Other restrictions apply as before.

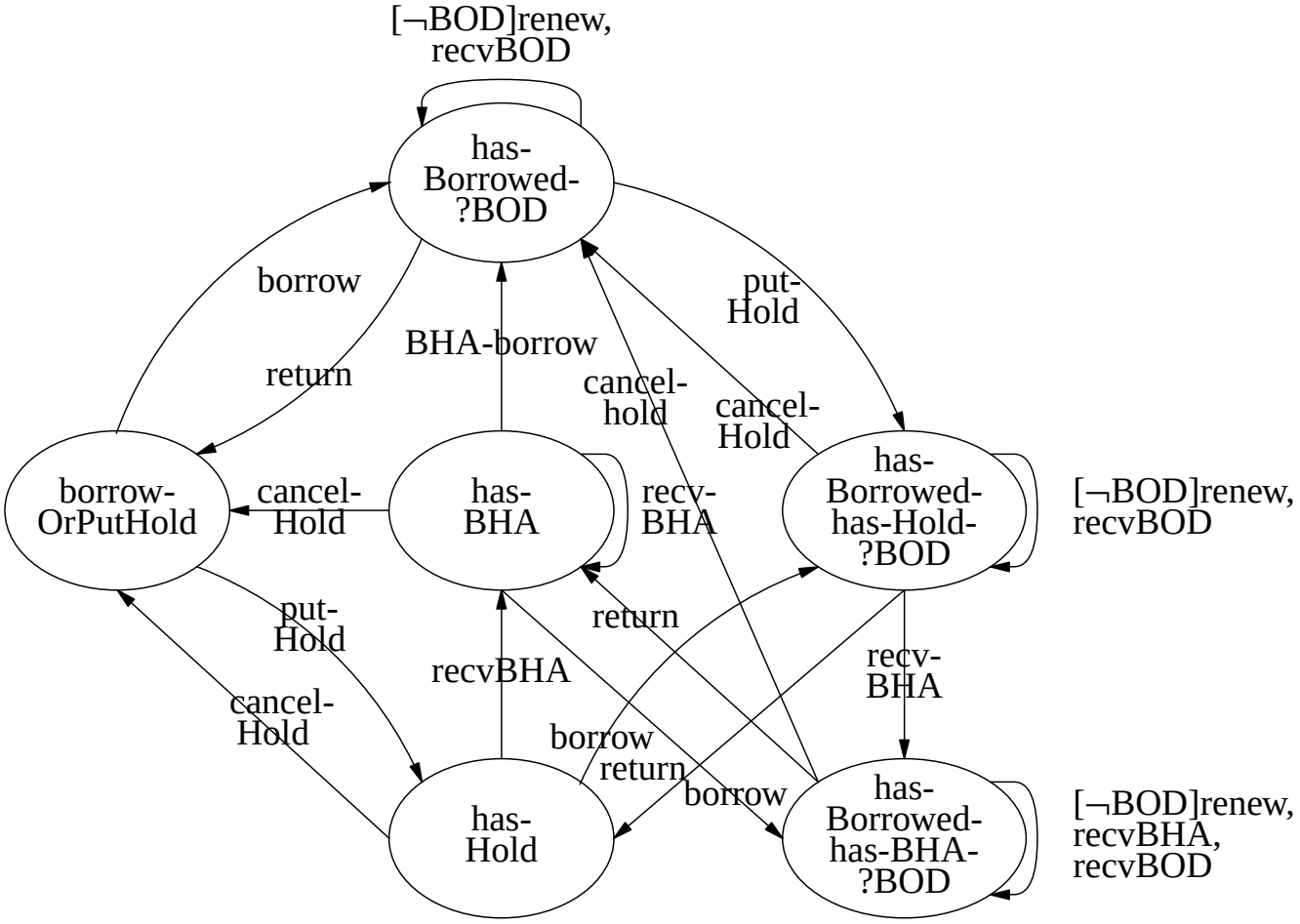


•? Should we have a "borrow" transition from "has-BHA" to "borrowed-and-Has-Hold"? How would its meaning differ from the existing borrow-transtion from "has-BHA"?

A SIMPLIFIED MODEL USING GUARDS

Following state-pairs are merged:

- {hasBorrowed, hasBOD}, {borrowedAndHasHold, hasBODand-Hold}, and {borrowedAndHasBHA, hasBODandBHA}.
- The part "?BOD" in a state-name means BOD-notice may or maynot have been received.



Question:

•? Show the state-diagram if we merge the states has-borrowed-has-Hold-?BOD and has-borrowed-has-BHA-?BOD, (Hint: some more transitions will now have new guards.) Should we also merge has-hold and has-BHA at the same time?

•? Modify the model by adding "search" operation for books.

GOING FROM A PROGRAM *P* TO ITS FINITE-STATE MODEL *M*(*P*)

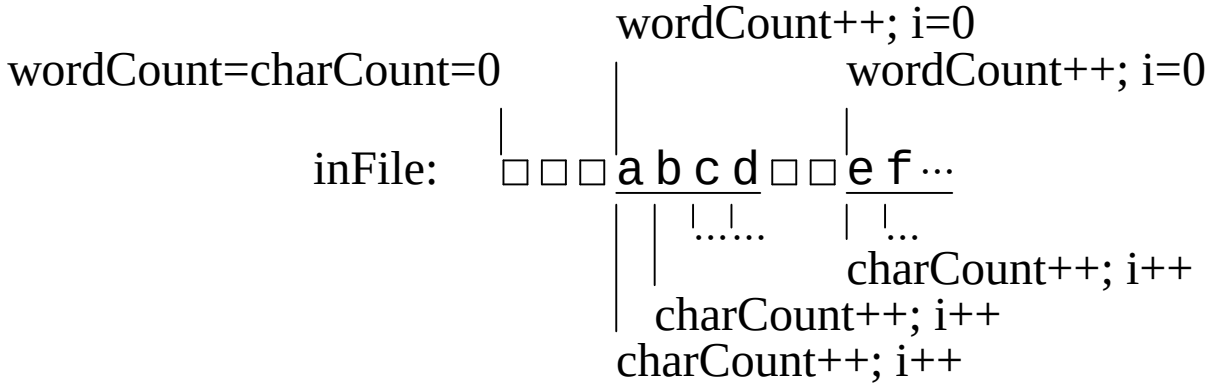
The WordCharCount program:

```
#include <stdio.h>
#define WORDLEN 20

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 ('\0' == word[i]) break;
           else charCount++;
 }
}
```

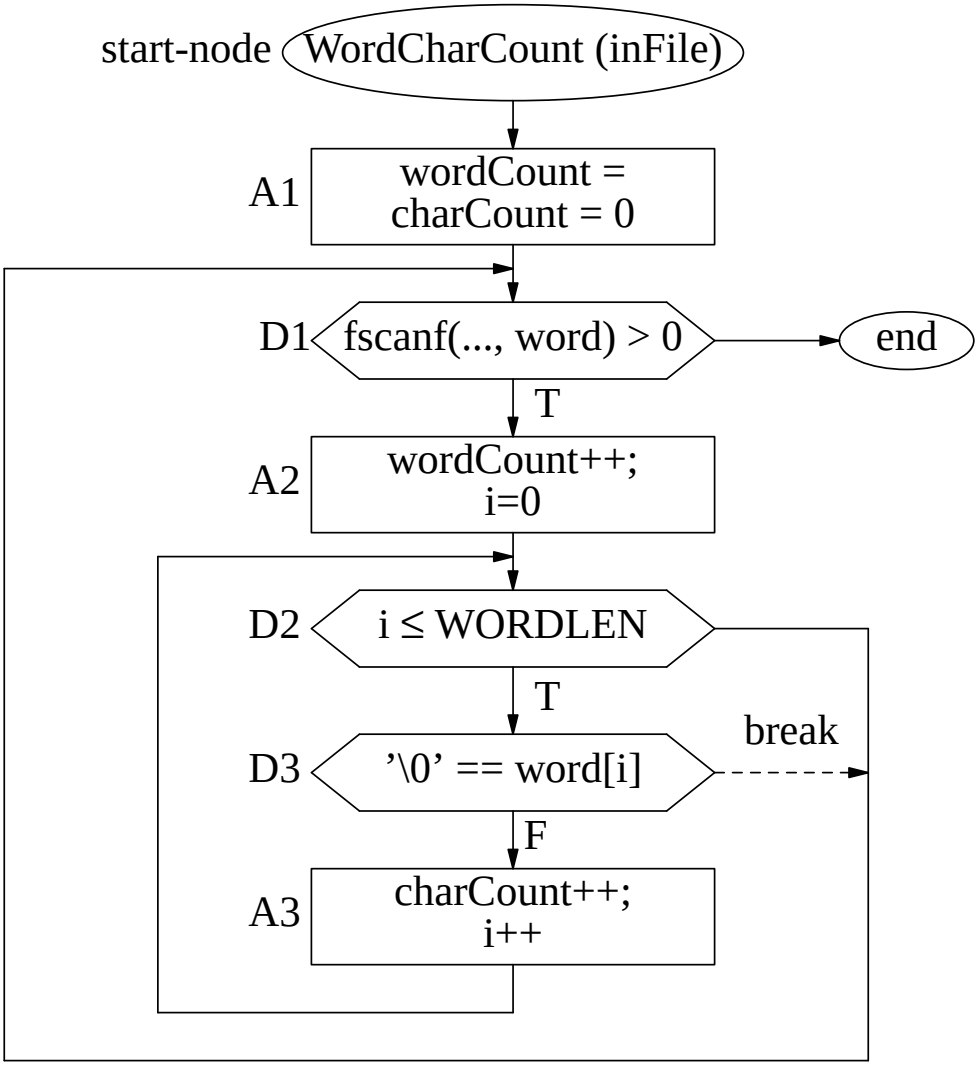
Example Input And Activities:



Behavior Patterns:

Patterns of read, test, and update operations.

FLOWCHART OF WordCharCount



Decision to decision path (DD-path):

The "chunk" of activities, if any, between two consecutive branch-points on a path from start to end.

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

Question: What is the relationship between the #(DD-paths) and #(decision-points) in the program? (Assume that each decision is two-way: true and false.)

EXERCISE

1. Show all possible improvements to the logic and efficiency of the following pseudocode (assume input file has no error); in particular, is it possible to require that the data in the input file be in a certain form that would allow simplifying the pseudo- code? Show the flowchart of the new pseudo-code; how many DD-paths are there in the flowchart? [7+6+2].

```
    function ClassifyTriangles(inputFile of one
                                integer-triplet per line)
    { int a, b, c;
 1.   while (not end of input file) {
 2.         read-integers(a, b, c); //positive numbers
 3.         print("input lengths:");
 4.         print-integers(a, b, c);
 5.         if ((a < b + c) && (b < a + c) && (c < a + b))
 6.         then isTriangle = true;
 7.         else isTriangle = false;
 8.         if (isTriangle)
 9.         then if ((a == b) && (b == c))
10.              then print(" an equilateral triangle");
11.              else if ((a ≠ b) && (a ≠ c) && (b ≠ c))
12.                   then print(" scalene triangle");
13.                   else print(" an isosceles triangle");
14.         else print(" not a triangle");
15.   }
    }
```

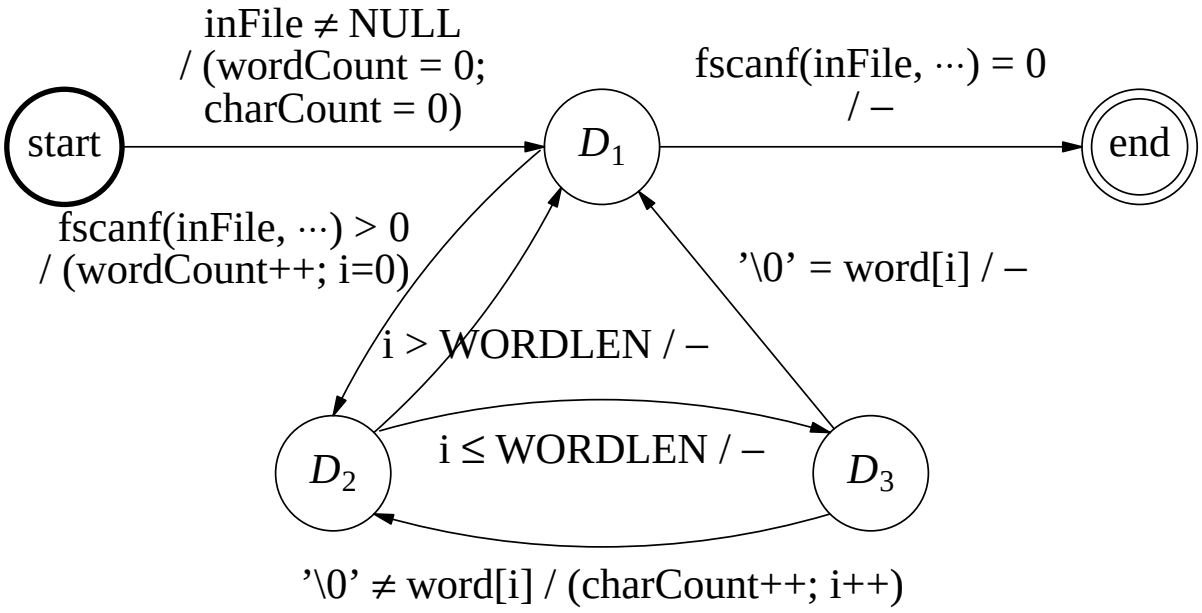
2. What is the general formula for the number of DD-paths and the number of decision-points (assume each decision is two-way: true and false)?

3. Assume initially ClassifyTriangles-function simply distinguished "triangular" and "non-triangular" triplets. What would be the successive refinements (extension) of the function that would result in the final improved form.

FINITE-STATE MODEL FROM DD-PATHS

Link Label:

- Conditions and actions: c_{ij}/a_{ij}.
- Conditions c_{ij} for the links from each node D_i are disjoint ($c_{ij} \wedge c_{ij'} = F$) and complete ($\vee c_{ij}$ o ver $j = T$).



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

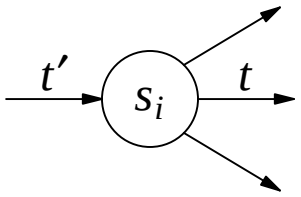
Abstracton of History of Computation:

- Decision-node + current values of all local/global vars.
- This determines the future computations from that point on.

All computations can be modeled, at any desired level, by FSMs using condition-guards on the transitions.

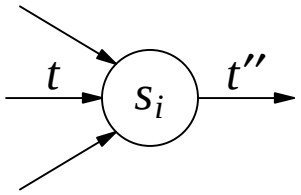
COMMON PROBLEMS IN SPECIFYING AN FSM

- **Unused transitions:**
  Not used in any computation-path.
- **Predecessor dependency**:
  Transition t is predecessor 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.



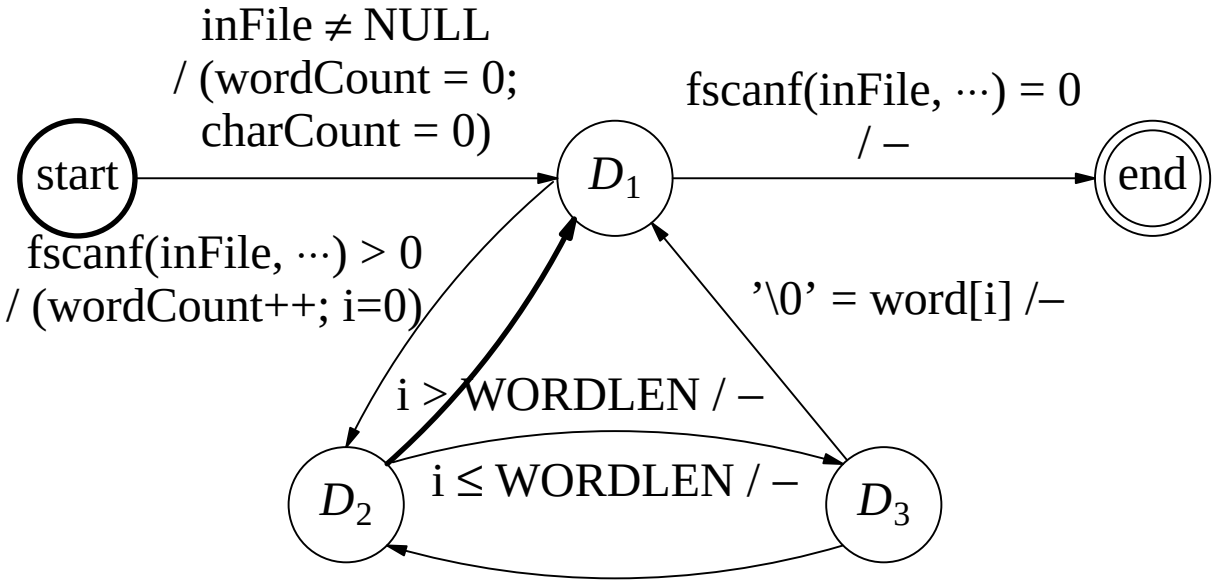
A trivial case:
t'' is the only transition from s_i.

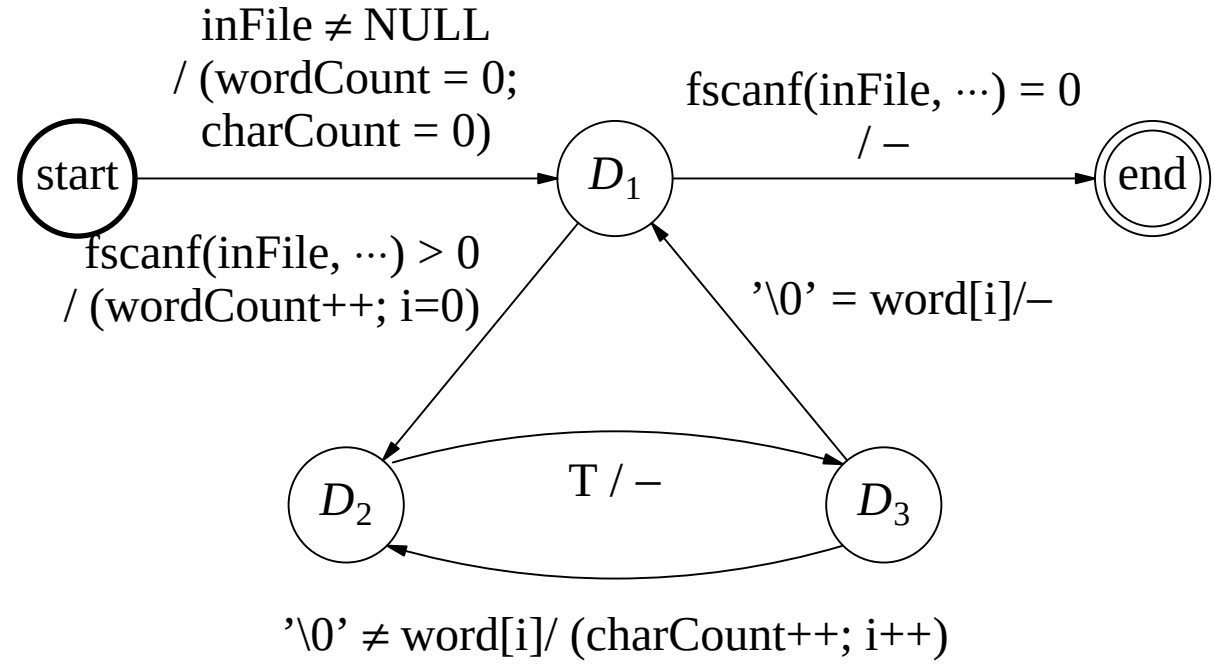
- Elimination of unused transitions and the dependencies can simplify the FSM and hence the program.

More Complex Dependencies:

- Dependencies between transitions that do not follow each other immediately (head of one transition ≠ tail of the other).

ELIMINATION OF UNUSED TRANSITION





THE NEW PROGRAM

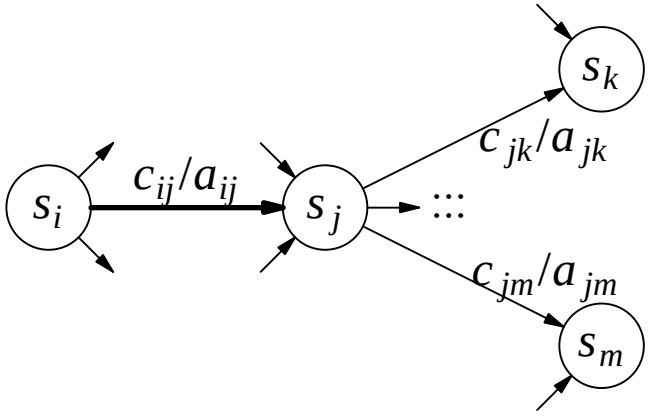
The Original Program:

```
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 ('\0' == word[i]) break;
            else charCount++;
  }
}
```

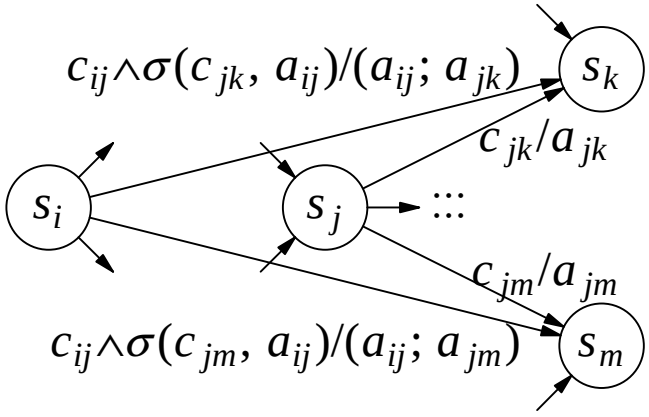
The New Program: Save the loop-test "i ≤ WORDLEN".

```
void WordCharCount(FILE *inFile)
{ int i;
  char word[WORDLEN+1];
  wordCount = charCount = 0;
  while (fscanf(inFile, "%s", word) > 0) {
        wordCount++;
        for (i=0; '\0'≠word[i]; i++)
            charCount++;
  }
}
```

TRANSITION ELIMINATION



Before elimination of (s_i, s_j).



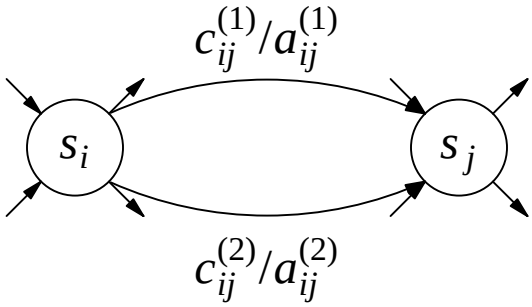
After elimination of (s_i, s_j).

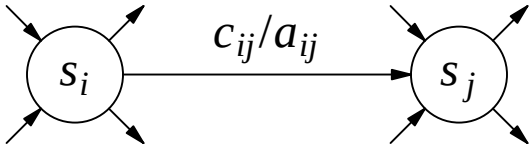
The modified form $\sigma(c_{jk}, a_{ij})$ of c_{jk}:

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

- If there are no transitions left to s_j, eliminate s_j.

ELIMINATION OF PARALLEL TRANSITIONS





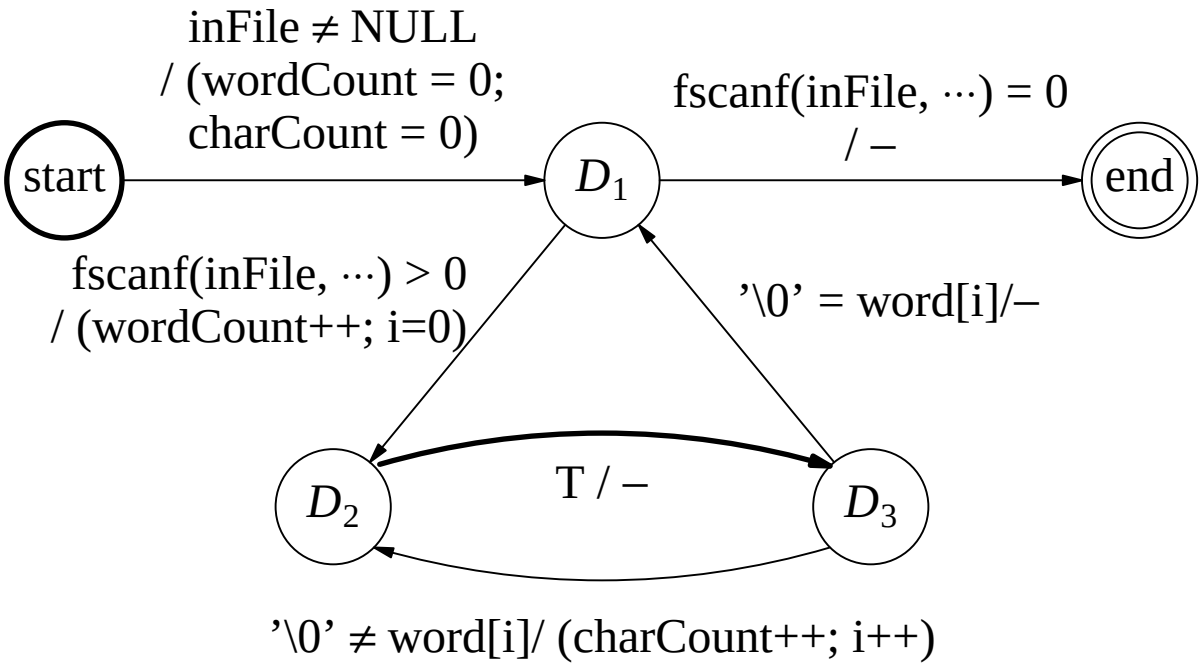
$$c_{ij} = c_{ij}^{(1)} \vee c_{ij}^{(2)} \quad \text{[disjoint]}$$
$$a_{ij} = \text{If } (c_{ij}^{(1)}) \text{ then } a_{ij}^{(1)} \text{ else } a_{ij}^{(2)}$$
$$= \text{If } (c_{ij}^{(2)}) \text{ then } a_{ij}^{(2)} \text{ else } a_{ij}^{(1)}$$

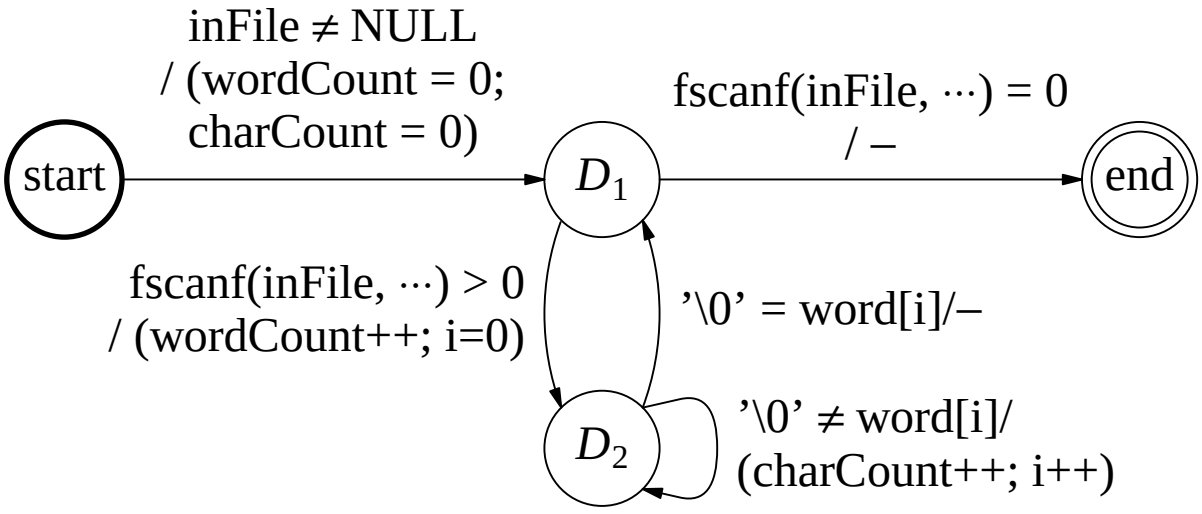
a_{ij} is more complex than both $a_{ij}^{(1)}$ and $a_{ij}^{(2)}$

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:
$$c_{ij} \vee [\bigvee_k c_{ik}] = \text{True}$$

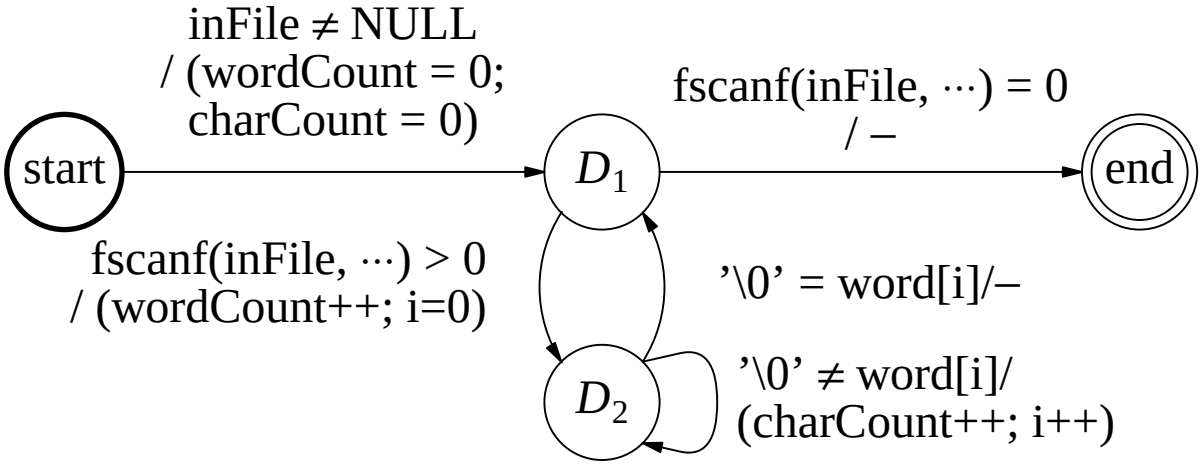
ELIMINATION OF PRECEDENT-DEPENDENT TRANSITION

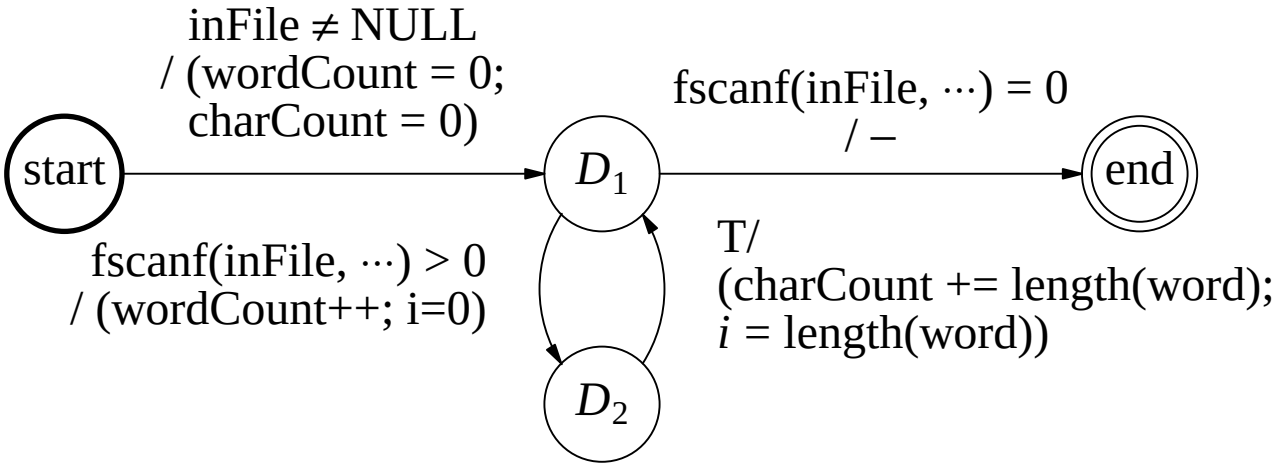


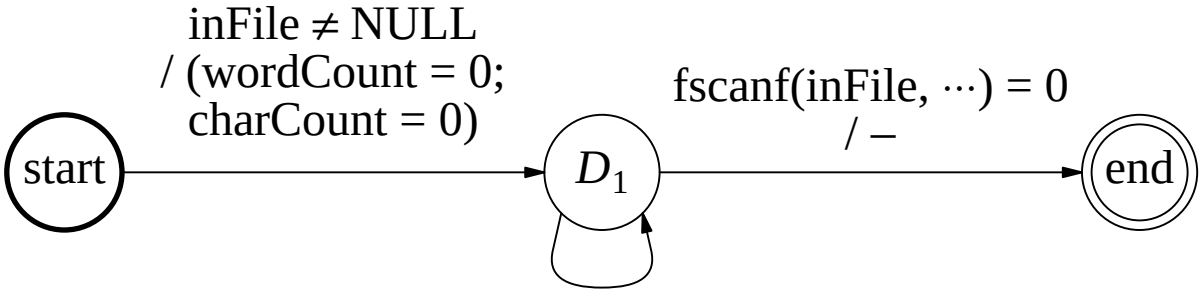


State D_3 is also eliminated.

ELIMINATION OF LOOP







fscanf(inFile, ···) > 0/ (wordCount++; charCount += length(word))

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

THE NEW PROGRAM

The final Program: //Slightly more effi cient than the previous one.
//Does not use the variable *i*, and charCount
//is incremented once in outer while-loop.

```
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 ('\0' == word[i]) break;
            else charCount++;
  }
}
```

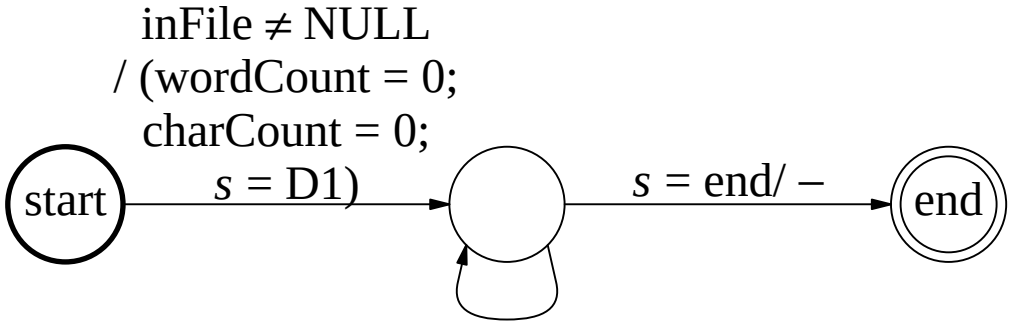
Keep Program Logic as clean as possible.

EXERCISE

1. Shown below is a variation of the WordCharCount-function, where there is no restriction on word-length; it assumes that the only word separators are the blanks and new-lines. Show the fbwchart (draw it properly), list the DD-paths, show the initial FSM (with condition-guards) based on the DD-paths, and then simplify the FSM as much as possible using transition-elimina-tions and state-elimination. Show intermediate steps.

```
void WordCharCounts(FILE *inFile)
{ char ch;
  wordCount = charCount = 0;
  while (fscanf(inFile, "%c", ch) > 0)
        if ((ch != ' ') && (ch != '\n')) {
           charCount++; wordCount++;
           while (fscanf(inFile, "%c", ch))
                 if ((ch != ' ') && (ch != '\n'))
                    charCount++;
                 else break;
        }
}
```

AN EQUIVALENT 3-STATE FSM



D1D2: [s = D1] ∧ [fscanf(inFile, ···) > 0]/
(wordCount++; i=0; s = D2)
D2D1: [s = D2] ∧ [i > WORDLEN] / s = D1
D2D3: [s = D2] ∧ [i ≤ WORDLEN] / s = D3
D3D1: [s = D3] ∧ [’\0’ = word[i]] / s = D1
D3D2: [s = D3] ∧ [’\0’ ≠ word[i]]/
(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.

SUMMARY

All computations can be modeled,
at any desired level, by FSMs
using condition-guards on the transitions.

Remarks:

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

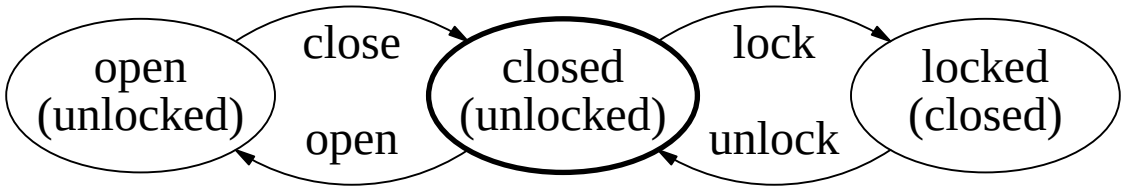
  Each computation/algorithm can be
  modeled by a Turing Machine.

- Construction of an FSM, without having a program in hand, is often a non-trivial task.

MORE EXAMPLES OF FSM

Window with a Lock:

- Four operations: open, close, lock, and unlock.
- Constraints:
  - can be opened only if it closed and unlocked.
  - can be closed only if it opened.
  - can be locked only if it is closed and unlocked.
  - can be unlocked only if it is locked.
- Initially closed and unlocked.



Proper state-names help us to easily identify
the applicable actions at a state.

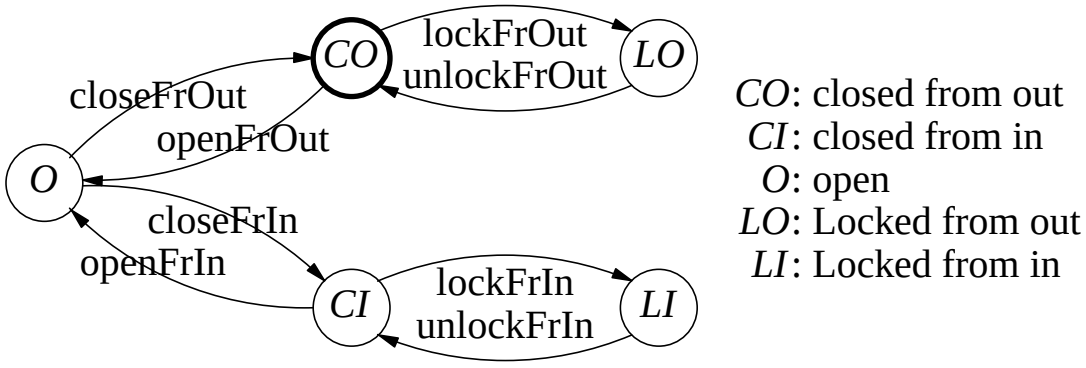
- There are no condition-guards for the transitions here (why?).
- There are no final states here because the operations can be continued for ever, without termination.
- We allowed transitions to the start-state to keep the number of states small.

MORE EXAMPLES OF FSM

Door With Two-sided Lock:

- Eight operations: openFromIn, closeFromIn, lockFromIn, and unlockFromIn, and similar operations from out.

  Imagine a person moving in and out of the room when the door is open; the person's moves are not modeled.
- Initially, the door is closed from out and unlocked.
- Constraints:
  - Similar to those for the window for the operations from in and for the operations from out.
  - "Inside" operations can occur only after the operation openFromOut, and likewise for "outside" operations.
- The door can be closed/locked from one side at a time.

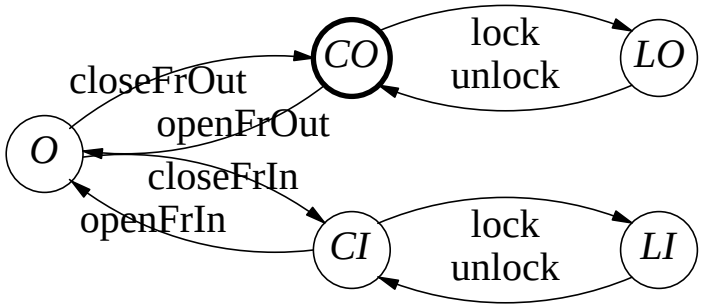


EXERCISE

1. Show the new FSM after we add the operations goIn and goOut to model the person's move.

THE CHOICE OF OPERATION-NAMES CAN AFFECT THE FSM

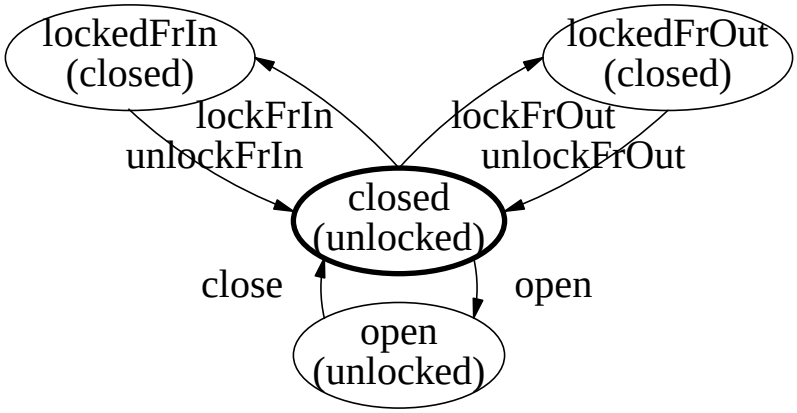
- Imagine a single lock-function that takes into account the state of the door-with-two-sided-lock and performs the appropriate operation lockFrIn or lockFrOut as needed.
- Similarly for the unlock-function.



Cannot use the same name "close" for both closeFrIn and closeFrOut because it causes non-determinism.

EXERCISE

1. Show the new FSM when we use "close" both for closeFrIn and closeFrOut and similarly for open, but keep different names lockFrIn and lockFrOut. (Hint: Following FSM is no good - why? Start with the FSM with goIn and goOut operations, replace them by λ-moves, and finally convert the FSM to a deterministic form if necessary.)



EXERCISE

1. How can a state-diagram fail to represent a proper FSM?
2. An elevator control problem. Assume that:

   G1. The elevator-use of a person is controlled by his id-card, which is validated by a security office at the ground level, for the following:

   (a) one groundFloor-entry and one destination-exit,
   (b) one destination-entry and one groundFloor-exit.

   G2. One registers the elevator-use request on his current fbor by inserting his id-card in a slot next to the elevator. (The elevator has no fbor-buttons inside or outside, and no fbor-indicator light inside; the id-card has a sound/light indicator to show that a person can exit the the elevator when it stops.)

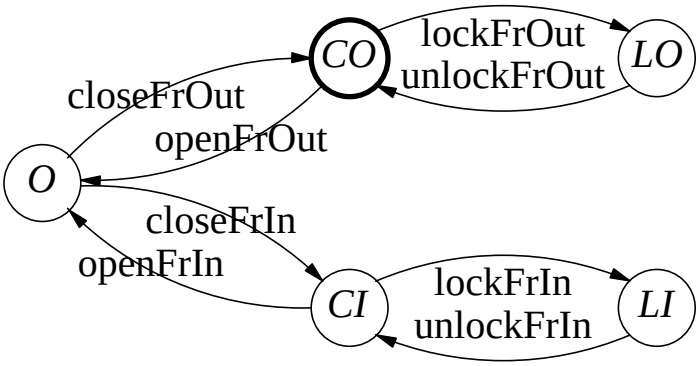
   A1 The elevator moves when an entry/exit-request (by persons outside/inside the elevator) is pending.

   A2 The load/unload operations are restricted so that no one goes to a fbor beyond his destination.

   Show the FSM from a person's view point. Also, show the FSM for elevator's movement, assuming that the fbors are {0=ground, 1, 2, 3, 4}. Here, use the states *Uj* (resp. *Dj*) = arriving at fbor *j* going-up (resp. going-down). A transition from *Uj* to *Dj* indicates a change of move-direction. Let *D*0 = startState. Do not label the transitions now.
   A complete system-state will consist for pending requests for people inside/outside the elevator, the elevator's current fbor, and its move direction.

DECOMPOSING AN FSM

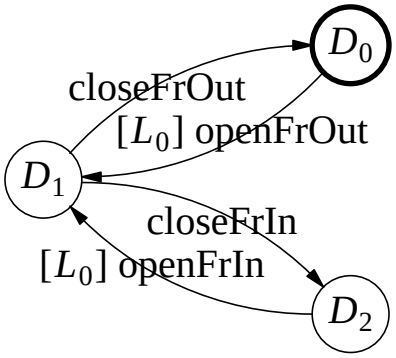
The FSM for Door With Two-sided Lock:



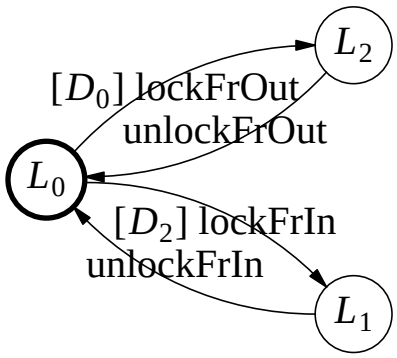
CO: closed from out
CI: closed from in
O: open
LO: Locked from out
LI: Locked from in

Decomposition into Two FSMs:

- We use guards to coordinate the interaction between them.
- The composition $M(D)\times M(L)$ gives the original FSM.



The fi nite-state model $M(D)$ for door.

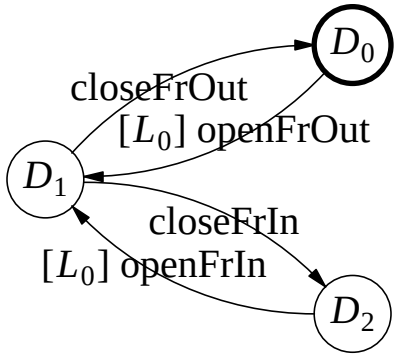


The fi nite-state model $M(L)$ for two-sided-lock.

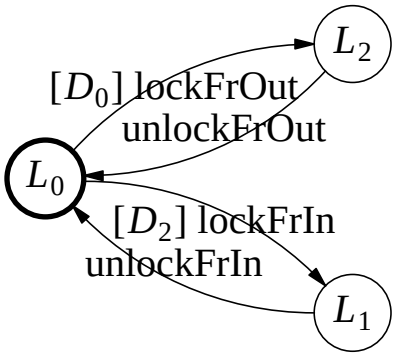
Question: What do the states in $M(D)$ and $M(L)$ look like in terms of the states in the original FSM?

FORMING THE COMPOSITION $M(D)\times M(L)$

Starting FSMs $M(D)$ and $M(L)$:



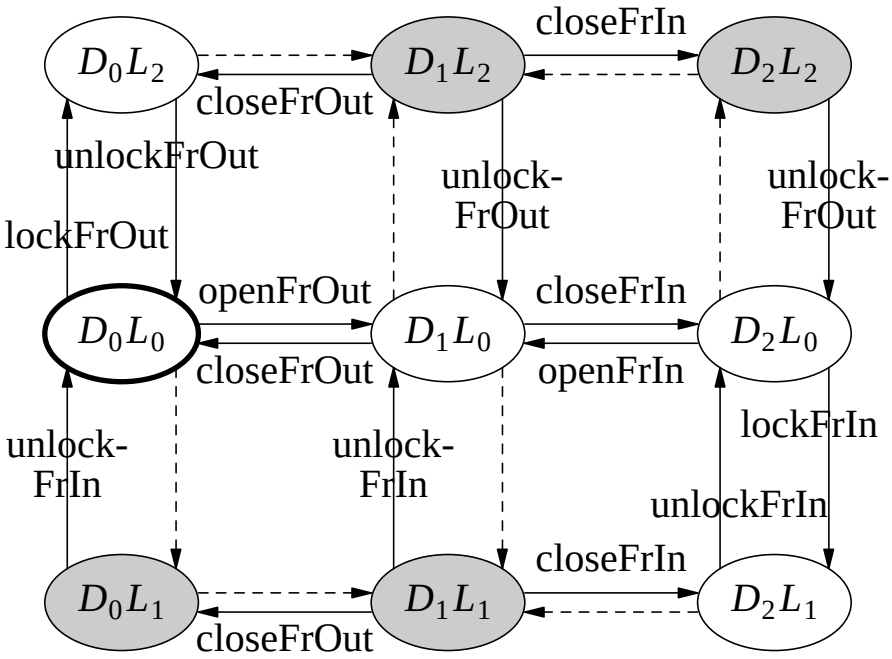
$M(D)$ for door.



$M(L)$ for two-sided-lock.

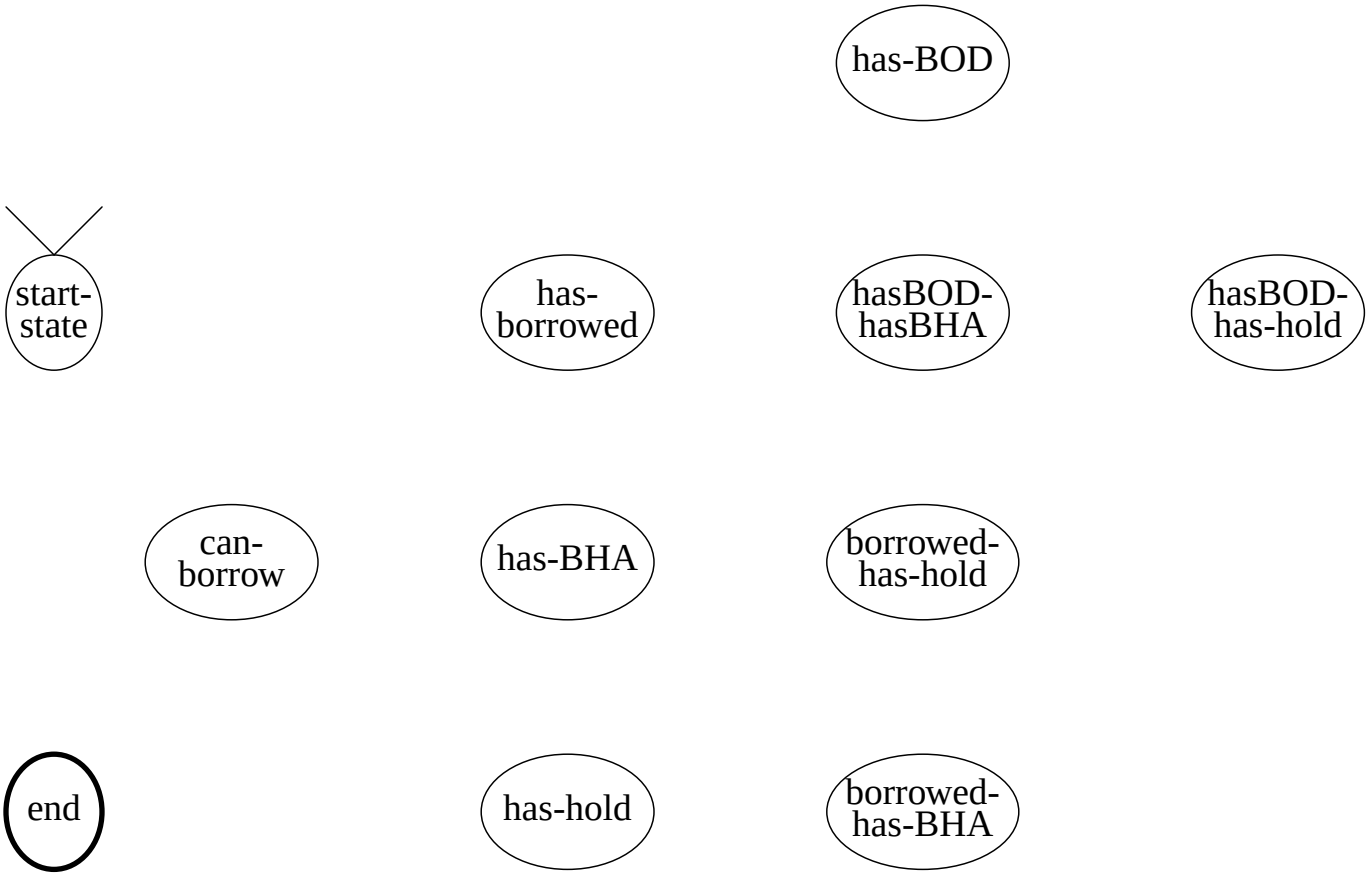
Composition $M(D)\times M(L)$:

- The dashed transitions are not present due to guards.
- The shaded states are not there because they are not reachable from the start-state D_0L_0.



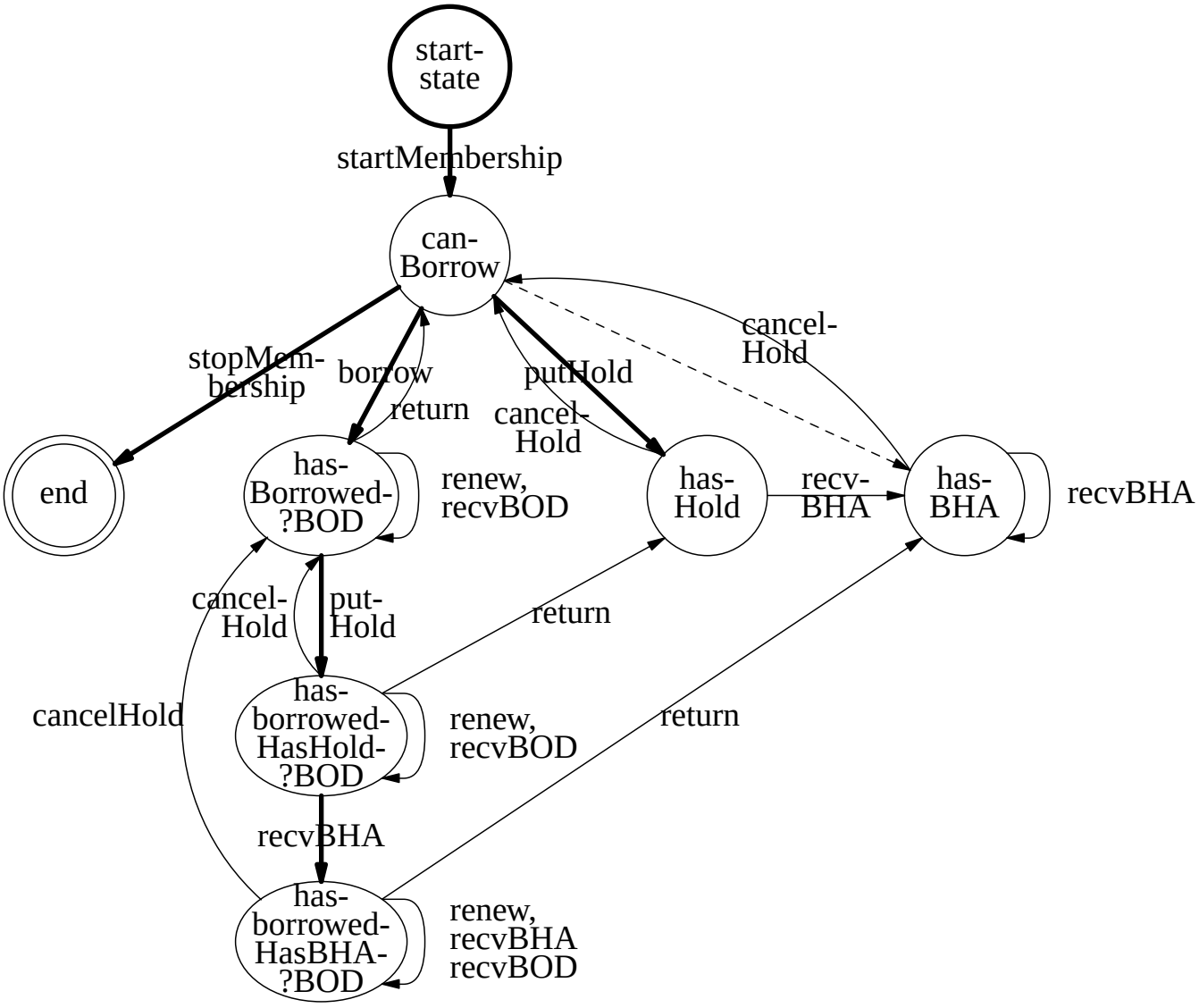
Simplifying Assumptions:

- At most one book can be borrowed at any time. In addition, at most one hold can be placed at any time.
- No limit on how many book-overdue or book-on-hold-available notices are received before memberhsip is cancelled.



Show the transitions among the states. The operations are: startMembership, stopMembership, borrow, renew, return, putHold, cancelHold, recvBOD, and recvBHA. Assume people are responsible and non-mischievous.

DOMAIN-TREE BASED DEVELOPMENT PLAN



Development (and Test Plan):

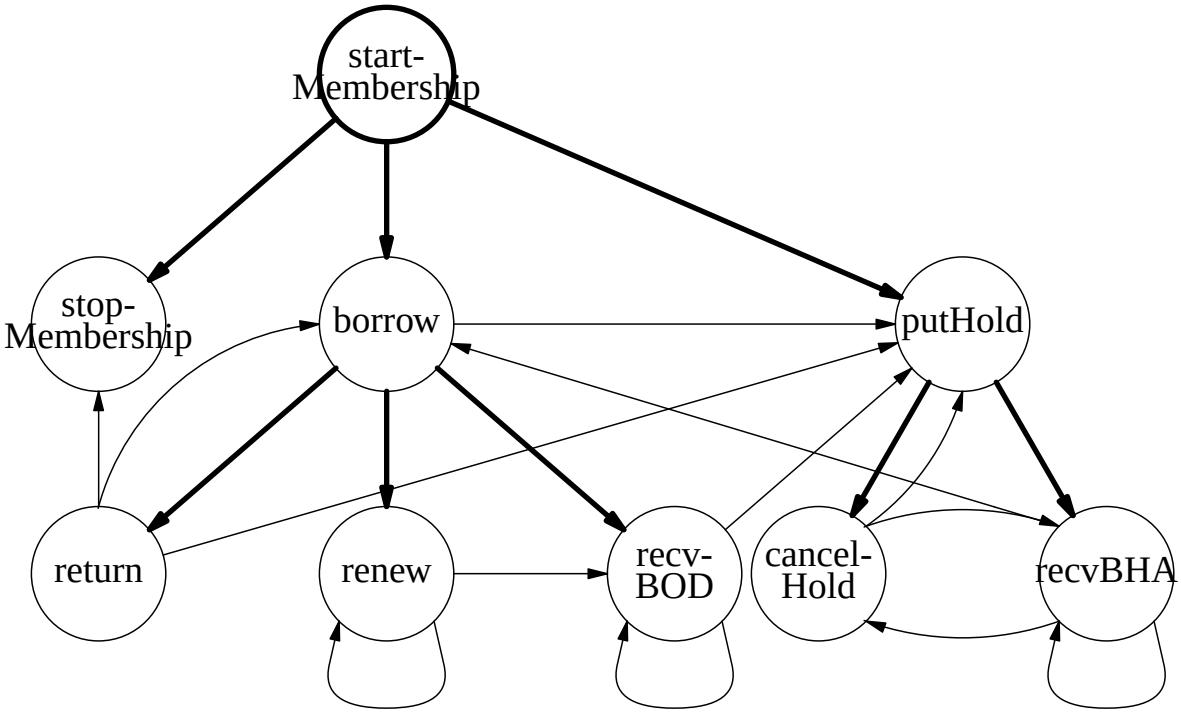
(1) startMembership, stopMembership (test them together)

(2) borrow, return (test borrow + return); recvBOD, renew (test them together, and also test them with borrow and return)

(3) putHold, cancelHold (test putHold + cacnelHold; also test them with borrow + ··· + return)

(4) recvBHA (test recvBHA + cancelHold; also test return + recvBHA + cacnelHold), etc.

DOMINATION-TREE FOR THE OPERATIONS



Development and Test Plan:

(1) startMembership, stopMembership (test them together)

(2) borrow, return (test borrow + return); recvBOD, renew (test them together, and also each with ? + return)

(3) putHold, cancelHold (test putHold + cacnelHold; also test each with borrow + and test borrow + putHold + return + ?)

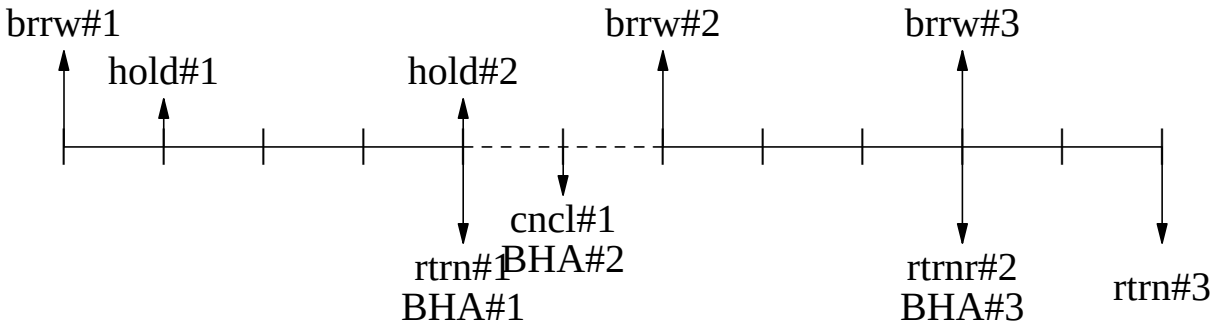
(4) recvBHA (test recvBHA + cancelHold; also test return + recvBHA + cacnelHold)

Question:

- Why can’t we merge some of the states above?
- Show the new FSM if we model a book that is not returned within a given number of reminders as "lost-notReturned"; the book may still be returned by the borrower at a later date.
- What is the new FSM if we allow multiple-holds on a book? What are the constraints for each new transition?
- What is the new FSM if we also add an event bookLost?

EXERCISE

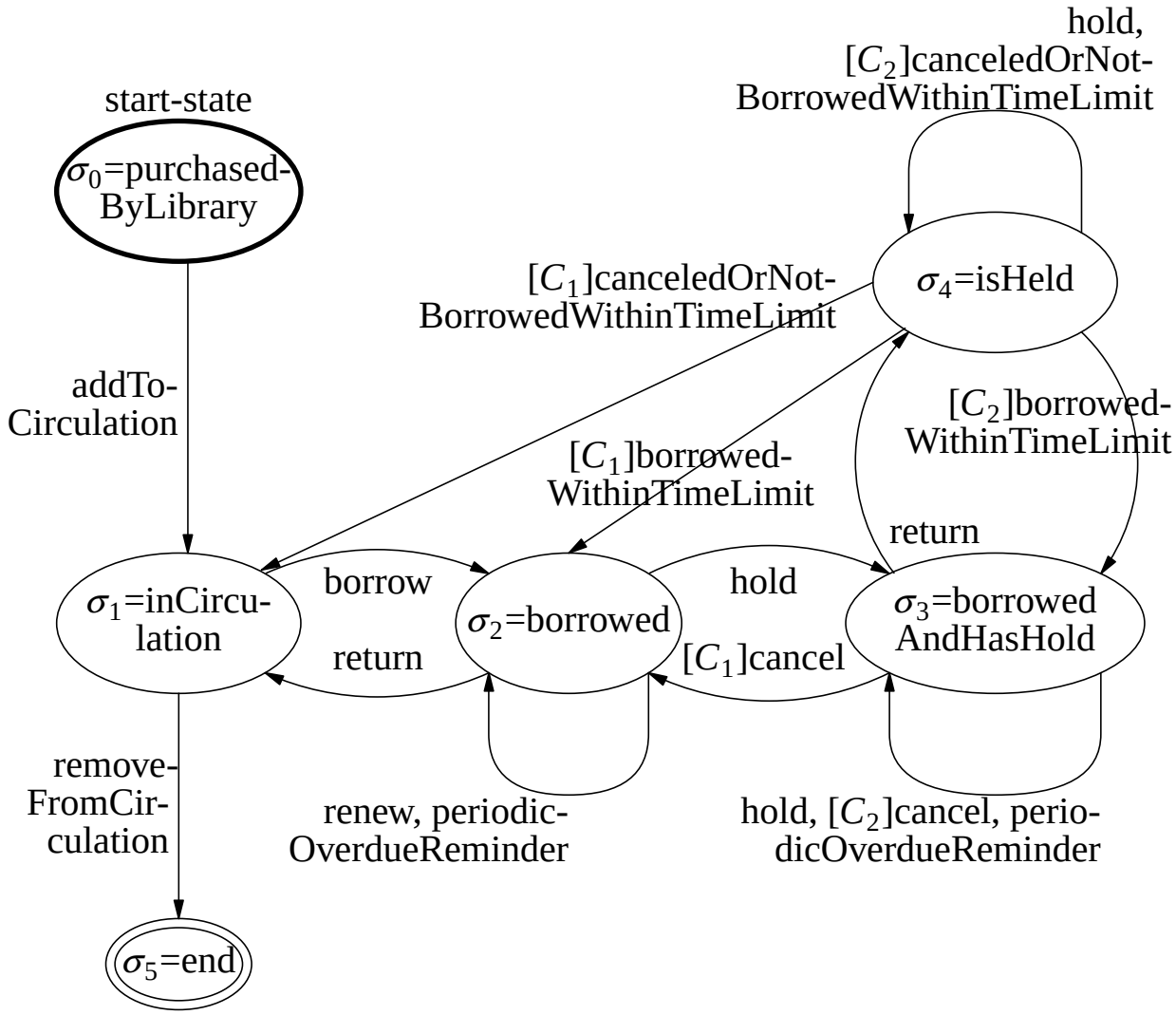
1. Show the guards for each transition in the finite-state diagram for a book with at most one hold-request.
2. Repeat Problem 1 for the modified FSM with multiple holds on a book.
3. Consider a simulation program for the library operation, where all events occur at the discrete time points 0, 1, 2, ··· and all requests are processed instantaneously at those discrete time points. In particular, all borrow and hold periods start and end at those discrete time points, and two distinct borrow periods can be juxtaposed without any gap between them. Give a pseudocode to describe how multiple borrow requests will be processed at a time point t. Do the same for the hold-requests. In which order will you process the borrow, hold, cancel, return requests at time t?
4. Shown below is the borrowed states (solid line) and isHeld states (dashed line) of a particular book for a period of 11 units of time.



Assume that only three persons p_1, p_2, and p_3 were involved during this time. One of these persons, who was very patient, had put a hold but canceled the hold after a long period of time. Assume that the max. period that a book is held for any customer is one unit of time. Show the times of each borrow-

request, hold-request, and cancel-request (use a horizontal line to show the start and end of a borrow-period and the start and end of a hold-request before cancellation, and so on). How long the person waited before he canceled his hold? Note that "borrow#*i*" is simply the *i*th borrow-activity and it need not be related to p_i.

EXTENSION OF BOOK-FSM TO THE CASE OF MULTIPLE HOLDS



Guards:

- C_1 = Has one hold and C_2 = Has ≥ 2 holds.

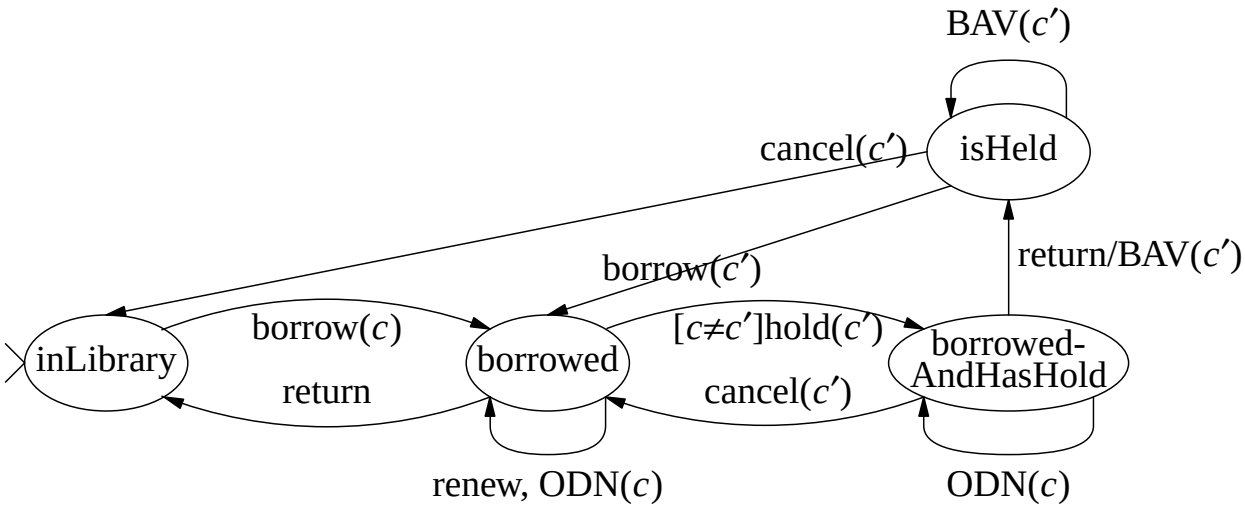
Notes:

- The transition $\sigma_3 \rightarrow \sigma_4$ for return involves addtional action of informing the fi rst hold-requester.
- The transition $\sigma_4 \rightarrow \sigma_4$ for cancel by the isHeld.person involves additional action of informing the next hold-requester.

FURTHER MODIFICATION OF THE BOOK-FSM

New Operations:

- Putting "hold" on a book currently checked-out, and canceling the "hold". Assume at most one hold on a book.
- Periodic overdue-notice (ODN) to the borrower for overdue books (including need-for-early-return when a "hold" is put on the book).
- Informing a customer who has a "hold" on a book that is available (BAV).
- ODN and BAV are automatically generated by the system.



Question:

•? Why do we need a separate state "isHeld", i.e., why can't we merge it with some other state?

•? How to extend this model using guards and other-actions if *m* = max #(ODN or BAV) sent?