Finite-State Modeling in Software Design: Some Fundamental Techniques

Sukhamay Kundu
Computer Science Dept, Louisiana State University
Baton Rouge, LA 70803, USA (email: kundu@bit.csc.lsu.edu)

Abstract

Although finite-state models have been used in software modeling for quite some time, a general method for building and manipulating such models which directly relates to a program's structure is not readily available. We fill this gap by constructing a canonical finite-state model $M(P)$ from the flowchart of a program P. Next, we present several methods for simplifying $M(P)$ which correspond to creating higher level models for P and to improving P by eliminating its design flaws. Finally, we show that states based on data-values and their abstractions give us a greater flexibility in creating finite-state models that can be used in practice to build the models from requirements.

1. Introduction

Finite-state models have several important applications in software design and development: (1) the verification of software requirements [4, 12, 14], which is often done in a semi-formal fashion because of the difficulties in purely formal verification methods, (2) the semi-automatic generation of a correct program from the model, and (3) the development of test-plans and high-level test-cases [3, 7, 8, 12] from the model. The ability to build finite-state models is also a prerequisite for building more complex models such as statecharts [5, 6] and petri-nets [4, 12]. A finite-state model (FSM) is basically a high-level description of the software which facilitates the tasks (1)-(3) above. An FSM is more general than a dataflow model since the latter does not show the decision points. The abstract nature of an FSM compared to a program-code makes an FSM more suitable for design analyses. Although the basic notion of an FSM is quite simple, the design of a suitable FSM for a given set of software requirements is far from so. The main difficulty lies in conceptualizing the abstract notion of a state into a concrete form that fits the software to be developed. Typically, there are too many choices for the states that are available to a designer, and selecting a proper notion for the states is a non-trivial task. We know from the theory of computation [11, 13] that in designing finite-state automata, which are structurally simpler than finite-state models for softwares, an improper choice of states may make it impossible to define the transitions and complete the model or make the model unnecessarily complex. A proper choice of states in an FSM likewise plays a critical role in successfully modeling a software design.

We present here some fundamental methods for building and refining an FSM. We begin by constructing a canonical finite-state model $M(P)$ from a program P to get an insight into the relationship between a program and its finite-state models, specifically the ways in which certain design flaws in P manifest themselves in the model $M(P)$. This leads us to the notion of states based on the data-values and their abstractions. This form of states is most suitable in practice for building an FSM from software requirements, which must precede the software development. The level at which we represent the data and the operations in such an FSM determines the level at which the model represents the software.

2. Basic concepts

A *finite-state* model [4, 6, 12, 14] consists of a finite number of states and a finite number of *transitions* among them with the following properties (1)-(4).

(1) There are two special states, called the *start-state* and the *end-state*. There are no transitions to the start-state and no transition from the end-state.

(2) Each transition $t(s_i, c_{ij}, a_{ij}, s_j)$ from a state s_i to a state s_j, where s_j may be the same as s_i, has associated with it a *condition* $c_{ij} = c(t)$ which evaluates to true(T)/false(F) and an *action* $a_{ij} = a(t)$ which modifies zero (as in a null action or $x = x$) or more variables in the program.

(3) The conditions for the transitions from a state s_i ($\neq$ end-state) are mutually disjoint, i.e., $c_{ij} \wedge c_{ik} = F$ for $j \neq k$. Moreover, the conditions c_{ij} are *complete* (exhaustive) in the sense that one of them always holds, i.e., their union or disjunction equals T, except perhaps for s_i = start-state.

(4) Each state s_i belongs to at least one path from the start-state to the end-state.

We sometimes refer to a transition $t(s_i, c_{ij}, a_{ij}, s_j)$ simply as $t(s_i, s_j)$ or $(s_i,\ s_j)$, when no confusion is likely. Note that there can be more than one transition from a state s_i to a state s_j. We assume, without loss of generality, that there is a unique transition from the start-state; the condition associated with it describes the valid inputs, and its associated action describes the initialization operations, if any. A *computation* in an FSM corresponds to a path, i.e., a sequence of transitions, from the start-state to the end-state. Each transition $t(s_i, c_{ij}, a_{ij}, s_j)$ in the path represents a computation *step*, where the condition c_{ij} holds for the current values of variables. The disjointness and the completeness of c_{ij}'s for the transitions at a state s_i ($\neq$ start-state) imply that exactly one transition at s_i is applicable at any point, which may depend on the preceding computation steps in the path. One can view a state s_i as representing a "phase" in the computation, where one of a few selected actions (which may depend on s_i) are performed. One can also view a state as a representation of some properties of the variables involved in the computations represented by the FSM. Note that a computation may follow different transitions at a state s_i if it arrives there multiple times. The complete *state-of-computation* at any point is given by the current state in the FSM together with the current values of all variables.

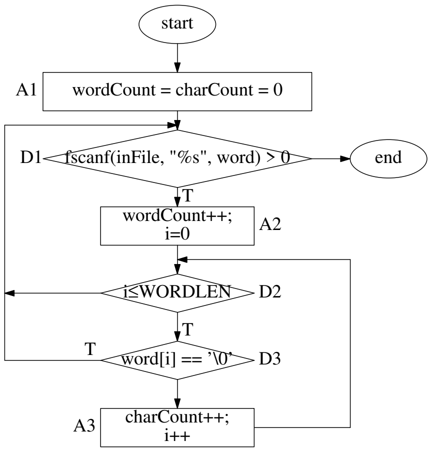
2.1. Building an FSM from a flowchart

The first step in constructing the finite-state model $M(P)$ for a program P, called the *canonical* FSM for P, is to form the flowchart of P. Next, we decompose the flowchart into larger "chunks", called the *decision-to-decision* paths (*dd*-path, in short). Each *dd*-path starts at the start-node or a branch-node of the flowchart and ends in a branch-node or the end-node, without going through any intermediate branch-node [1, 2]. We define the states of $M(P)$ to be the decision-nodes of the flowchart plus the start-node and the end-node (we may think of these two nodes as representing the start and end nodes of the flowchart). Each dd-path now gives a transition in $M(P)$ between the corresponding states in $M(P)$; the branch-condition of the dd-path corresponds to the condition-part of the transition and all the actions in the dd-path together corresponds to the action-part of the transition. The empty else-part of an if-statement gives a transition with the null-action. Note that each state in $M(P)$, save the start and end states, has exactly two transitions from it and their conditions are disjoint and complete.

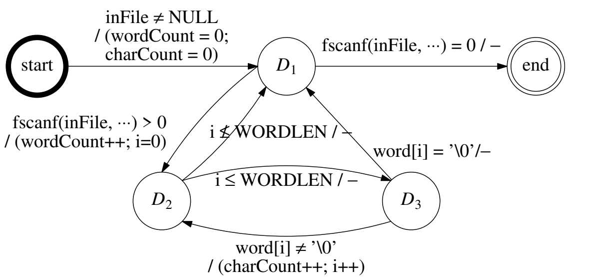
Example 1. Fig. 1(i) shows a simple function for counting words in a textfile and counting the characters in those words. Figs. 1(ii)-(iii) show the flowchart and the finite-state model for this function based on the *dd*-paths. The break-statement in the for-loop in Fig. 1(i), which makes the program unstructured, can be regarded as a design flaw which is easily corrected by replacing the exit-condition "`i <= WORDLEN`" of the loop with "`word[i] ≠ '\0'`" and replacing the if-statement of the loop-body by "`charCount++`". (This also makes the code slightly more efficient.) As we remarked earlier, each state in Fig. 1(iii) represents a computation phase where we select an action from a fixed set of alternatives (depending on the state) and perform that action before updating the state. The alternatives at D_1, for example, are: the termination of computation because of end-of-file, and the preparation for counting characters in the current `word` after updating `wordCount`. □

```
//Scan input textfile and set values of global vars.
//wordCount and charCount; blanks, newlines, etc. are
//not counted.  Assume each word has length <= WORDLEN.
void WordCharCount(FILE *inFile) //inFile ≠ NULL
{int i;
 char word[WORDLEN+1]; //1 for end of string '\0'
 wordCount = charCount = 0;
 while (fscanf(inFile, "%s", word) > 0) {
        wordCount++;
        for (i=0; i<=WORDLEN; i++)
             if (word[i] == '\0') break;
             else charCount++;
 }
}
```

(i) A function for counting the words in a textfile and counting the characters in those words.



(ii) The flowchart of the function in (i)



(iii) A finite-state model based on *dd*-paths; the conditions and actions are shown as c_{ij}/a_{ij}.

Figure 1. Construction of the finite-state model $M(P)$.

The above method for building $M(P)$ works equally well if P involves recursion; each recursive call simply becomes part of an action a_{ij}. The following theorem is easily proved.

Theorem 1. The canonical finite-state model $M(P)$ based on the *dd*-paths in a program P has a cycle if and only if the flowchart of P has a cycle. □

If a program P consists of several functions P_i, then the finite-state model $M(P)$ consists of several $M(P_i)$, one for each P_i, which would interact in a fashion similar to the way the functions call each other [10].

2.2. Two characteristics of a poor FSM

A transition t_i is said to be *precedent* dependent on another transition t_j if every computation-path which uses t_j also uses t_i prior to t_j. It is not necessary, however, that every use of t_j has to be preceded by a distinct use of t_i. The notion of *successor* dependency is defined in a similar way, A trivial case of the precedence dependency arises if t_i is the only transition to state s_k and t_j is a transition from s_k; in this case, every use of t_j is (immediately) preceded by a distinct use of t_i. A similar case of successor dependency arises if t_j is the only transition from s_k and t_i is a transition to s_k. A precedence (successor) dependency may occur also between two transitions to (from) the same state. The presence of dependent transitions makes the analysis of an FSM more difficult and is therefore undesirable. An *unused* transition, i.e., one which does not participate in any computation has a similar effect. To eliminate an unused transition t_{ij} from s_i, we simply delete it; we may also replace the condition-part c_{ik} of one or more of the other transitions from s_i by $c_{ik} \wedge \neg c_{ij}$ to simplify it. The complete elimination of dependent transitions by eliminating one of the two transitions from every dependent pair may not be always possible. The presence of dependent transitions and unused transitions in $M(P)$ arises directly from the corresponding flaws in P. In Fig. 1(iii), (D_2, D_1) is an unused transition and (D_2, D_3) is trivially precedent dependent on both (D_3, D_1) and (D_3, D_2).

If a program P contains paths which do not represent valid computations, then the same is true for $M(P)$. One way to eliminate such paths in an FSM is to keep the model at a sufficiently high level, and this can be used as a guideline for determining a suitable level at which to build a model. We can also consider other methods for building a finite-state model without using the *dd*-paths (see Section 5).

3. Improving an FSM

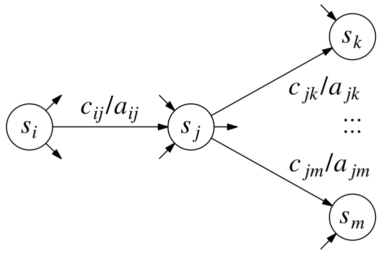
We now present several techniques for improving an FSM. For the FSM $M(P)$, some of these techniques have direct implications on improving the design of P.

3.1. Eliminating a transition

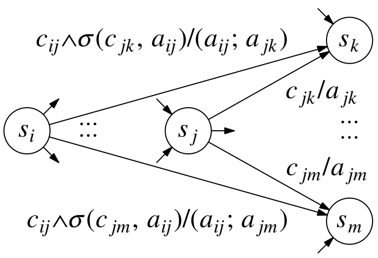
Fig. 2 illustrates the technique for eliminating a transition (s_i, s_j). Here, we assume that there are no loop-transitions at state s_j; the elimination of loops is discussed later. The elimination of the transition (s_i, s_j) introduces a new transition (s_i, s_k) from s_i for each transition (s_j, s_k) from s_j. The condition and the action for the new transition (s_i, s_k) are shown below, where $\sigma(c_{jk}, a_{ij})$ represents the modified form of c_{jk} expressed in terms of the values of variables prior to the action a_{ij} as illustrated in Fig. 2(iii). The *compound* action $(a_{ij}; a_{jk})$ means the actions a_{ij} are followed by actions a_{jk}.

$$c_{ik} = c_{ij} \wedge \sigma(c_{jk}, a_{ij}) \quad \text{and} \quad a_{ik} = (a_{ij}; a_{jk})$$

Although the total number of transitions may actually go up in the above process, the meaning of the states remain unchanged. However, if (s_i, s_j) is the only transition to s_j then the elimination of (s_i, s_j) makes s_j unreachable from the start-state and hence we can eliminate s_j and all transitions from it, reducing both the number of states and the number of transitions by one. A similar situation arises if (s_j, s_k) is the only transition from s_j and we eliminate all transitions to s_j and hence the state s_j as well.



(i) Before elimination of (s_i, s_j).



(ii) After elimination of (s_i, s_j).

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

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

Figure 2. The addition of new transitions due to the elimination of a transition (s_i, s_j).

Example 2. Fig. 3(i) shows the result of eliminating the unused transition (D_2, D_1) in Fig. 1(ii). Fig. 3(ii) shows the result of eliminating the transition (D_2, D_3), which is precedence-dependent on both the transitions (D_3, D_1) and (D_3, D_2); the state D_3 is also eliminated in the process. The new FSM corresponds to replacing the for-loop's exit condition "`i <= WORDLEN`" in Fig. 1(i) by "`word[i] != '\0'`", eliminating the break-operation (see Example 1), which may be regarded as a correction-patch for the exit-condition design error. □

Suppose we construct an FSM $M'(P)$ from P using each node in the flowchart as a state in $M'(P)$. A branch node in P then gives a state with two transitions, each with the null-action, and a non-branch node in P gives a state with one transition with $c_{ij} = T$ and a_{ij} = the action at that node. If we now eliminate all transitions to the states in $M'(P)$ resulting from the non-branch nodes, then the final result is the canonical FSM $M(P)$. The following theorems are easily proved [10]. Theorem 2 follows directly from $c_{ij} = \vee\{c_{ij} \wedge \sigma(c_{jk}, a_{ij})$: transitions (s_j, s_k) at $s_j\}$, where the terms $c_{ij}\wedge\sigma(c_{jk}, a_{ij})$ are mutually disjoint and are also disjoint from the conditions c_{im} for other transitions from s_i.

Theorem 2. If the transitions at both s_i and s_j are initially complete, then the new transitions at s_i after the elimination of $t(s_i, c_{ij}, a_{ij}, s_j)$ are also complete. □

Theorem 3. If P is a semi-structured program, i.e., P does not contain goto's, but may contain break, continue, and return statements, then each strongly connected component in $M(P)$ has a single entry node. □

3.2. Eliminating parallel transitions

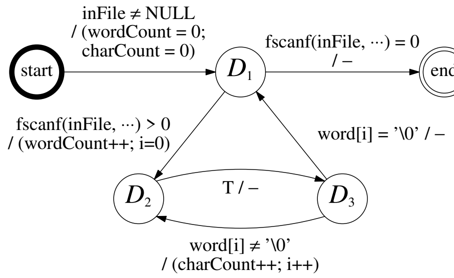
Two transitions which start from the same state s_i and which end in the same state s_j are called *parallel*. Parallel transitions can arise in the process of eliminating a transition. They also arise from *dd*-paths of a simple if-then-else statement that does not contain nested blocks and goto-like operations (such as break, continue, and return). We can merge two parallel transitions $t_1(s_i, c_{ij}^{(1)}, a_{ij}^{(1)}, s_j)$ and $t_2(s_i, c_{ij}^{(2)}, a_{ij}^{(2)}, s_j)$ into a single transition $t_{12}(s_i, c_{ij}^{(1)}\vee c_{ij}^{(2)}, a_{ij}^{(12)}, s_j)$, where

$$\begin{aligned} a_{ij}^{(12)} &= \text{if } (c_{ij}^{(1)}) \text{ then } a_{ij}^{(1)} \text{ else } a_{ij}^{(2)} \\ &= \text{if } (c_{ij}^{(2)}) \text{ then } a_{ij}^{(2)} \text{ else } a_{ij}^{(1)}. \end{aligned}$$

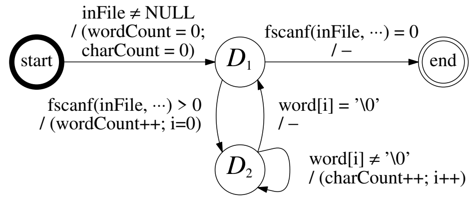
Here, $a_{ij}^{(12)}$ has a complex nature and that is permissible; in fact, an action may be arbitrarily complex. The above merging operation is particularly useful when $a_{ij}^{(1)} = a_{ij}^{(2)} = a_{ij}$, in which case we also have $a_{ij}^{(12)} = a_{ij}$. Merging two parallel transitions may sometimes give rise to new precedence (successor) dependencies.

3.3. Eliminating a loop

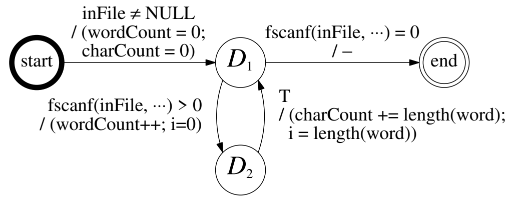
A loop-transition has the form $t(s_i, c_{ii}, a_{ii}, s_i)$, and its elimination is more difficult compared to the two previous operations. We now need to determine: (1) the condition for the *loop-termination*, and (2) the *loop-invariant* or, more precisely, the loop-termination action which describe the total effect of the computations of the loop on its termination. The elimination of $t(s_i, c_{ii}, a_{ii}, s_i)$ requires the information in both (1) and (2) to be filtered into each transition (s_i, s_j), $s_j \neq s_i$.



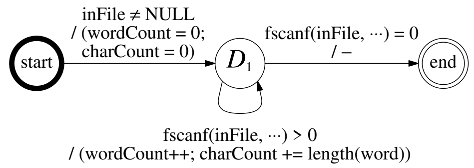
(i) After eliminating the unused transition (D_2, D_1) and "$i \leq$ WORDLEN" from the condition-part of (D_2, D_3).



(ii) After eliminating the transition (D_2, D_3) and the state D_3.



(iii) After eliminating the loop-transition (D_2, D_2) which is precedent-dependent on (D_2, D_1).



(iv) After eliminating the variable i, the transition (D_1, D_2), and the state D_2.

Figure 3. Simplifying the FSM in Fig. 1(iii) by transition-elimination.

For the loop at D_2 in Fig. 3(ii), the loop-termination condition is "`word[length(word)] = '\0'`", which always holds, i.e., the loop always terminates. The total computation of the loop on termination is given by "`charCount = charCount + length(word)`" and "`i = length(word)`". This gives the FSM in Fig. 3(iii). We can further simplify it by eliminating "`i = 0`" from the action-part of (D_1, D_2) since this definition of `i` is not used in the condition or action of any transition that can be reached from D_2; likewise, we can eliminate "`i = length(word)`" from the action-part of (D_2, D_1). The FSM in Fig. 3(iv) is obtained now by eliminating transition (D_1, D_2), which is precedence-dependent on (D_2, D_1). Shown below is the corresponding modified form of the program in Fig. 1(i); it has no precedence-dependencies or unused transitions, and it does not model the detailed processing of the individual characters in `word`.

```
void WordCharCount(FILE *inFile) //inFile ≠ NULL
{char word[WORDLEN+1];
 wordCount = charCount = 0;
 while (fscanf(inFile, "%s", word) > 0) {
        wordCount++;
        charCount += length(word);
 }
}
```

If we eliminate the loop in Fig. 3(iv), the action-part of the new transition (D_1, end) gives the program's output specification: wordCount = #(words in inFile) and charCount = the sum of lengths of those words.

3.4. Minimizing states

The notion of minimizing the number of states in an FSM is not meaningful, unlike that for a finite-state automata in Theory of Computation [10, 12]. This is because one can always create an equivalent FSM with at most three states, i.e., with at most one state other than the start and end states, which performs the same sequence of actions on the original variables for each input as in the given FSM. Fig. 4 shows such an equivalent FSM corresponding to Fig. 1(iii); it uses one additional variable s to represent the current state (other than start-state) of the FSM in Fig. 1(iii). The transitions in Fig. 4 are labeled as D_iD_j, where D_j may equal end-state, for the ease of comparison with corresponding transitions (D_i, D_j) in Fig. 1(iii), except for the transition from start-state and the transition to end-state. The condition-part of each transition from the unique intermediate-state in Fig. 4 has an additional term of the form "$s = D_i$", and the action-part has a similar term $s = D_j$.

This form of state reduction generally does not simplify either the understanding of the FSM or the verification of its correctness. In particular, all precedence-dependencies in the original FSM also hold among the corresponding transitions in the reduced model. The use of the variable s here is comparable to the use of artificial variables in structuring an unstructured program.

We can likewise eliminate the end-state as well in Fig. 4, giving a 2-state model, where we assume that a computation stops when s = "end" since no transitions apply at that point. Note that the completeness property of the transitions at the non start-state no longer holds. The start-state can also be eliminated in the same way and make the unique intermediate state the start-state, if we assume that initially s = start-state, add the condition "[s = start-state]" to the condition-part of the transition corresponding to the transition from the original start-state, and do not require that there are no transition to start-state.

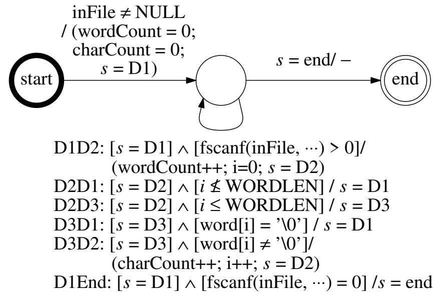


Figure 4. The 3-state equivalent FSM for Fig. 1(iii).

3.5. Merging equivalent states

We define two states s_i and s_j to be (strongly) *equivalent* if for each path from s_i to end-state there is a corresponding path from s_j to end-state having the same sequence of computations and condition-checkings, and vice-versa. This means for each given initial values of the variables, we go through the same sequence of computations and condition-checkings by starting at s_i and s_j. In particular, if for one set of initial values the computation starting at s_i terminates, i.e., reaches end-state, then the same is true for the computation starting at s_j for those initial values; moreover, both cases will give the same final value for each variable. Note that no state can be equivalent to end-state and likewise no state can be equivalent to start-state. It is easy to see that the strong equivalence defines an equivalence relation on the states of an FSM.

A simple way to verify the strong equivalence of two states s_i and s_j is that for each state s_k and for each transition from s_i to s_k there is a corresponding transition from s_j to s_k having the same condition $c_{ik} = c_{jk}$ and the same action $a_{ik} = a_{jk}$, and vice-versa. We can slightly relax the above requirement when $s_k = s_i$ or s_j, namely, the corresponding transition from s_j may go to either s_i or s_j. For $s_k \neq s_i$ and s_j, it actually suffices that corresponding to each transition from s_i to s_k, there is a transition from s_j to some $s_{k'}$ having the same condition and action, where $s_{k'}$ is known to be equivalent to s_k.

It is clear that if we merge two strongly equivalent states s_i and s_j into a single state (and eliminate duplicate transitions created by the merging and replace the transition between s_i and s_j, if any, by a loop-transition at the merged state), then we do not alter the computations modeled by the FSM. Sometimes, merging of one equivalent state-pair may make it easier to detect other equivalent state-pairs and thus the merging process may be continued further. In the finite-state model $M(P)$ for a program P, equivalent states arises, for example, when a chunk of identical code (say, a loop) is duplicated at the end of both "then" and "else" parts of an if-statement. The merging of the states arising from these duplicated codes amounts to filtering the duplicated code out of the if-statement and putting one copy of it immediately after the if-statement. If the duplicated code consists of a simple assignment statement, then replacing them by one copy after the if-statement does not alter $M(P)$.

Suppose now that the then and else parts of an if-statement begin with the same assignment statement and that the assignment operation does not affect any of the variables in the condition-part of the if-statement. Let s_i be the state in $M(P)$ corresponding to the decision node for the if-statement. The action-parts of the two transitions from the state s_i will have a common initial action component a_i corresponding to the assignment statement. Replacing the two copies of the assignment statements by one copy before the if-statement means each transition to state s_i will now have its action-part augmented by the additional action a_i following its existing actions; also, the part a_i will be eliminated from the action of each transition from s_i. If a_i modifies one or more variables in the if-test, one can still carry out the filtering operation provided the action a_i is reversible, i.e., a_i^{-1} exists. In that case the condition-part c_{ij} of each transition (s_i, s_j) from s_i is to be replaced by $\sigma(c_{ij}, a_i^{-1})$ (see Section 3.1). The situation is somewhat different when the initial duplicate code in the then and else parts is more complex, say, a loop. In this case, the modification to $M(P)$ is also complex and includes merging of states and modifying transitions both from s_i and to s_i.

4. Automatic code generation

An important use of an FSM is the semi-automatic generation of a correct program from the model. If we instantiate the FSM-SIMULATOR algorithm below with the states s_i and the condition-action pairs c_{ij}/a_{ij} for the transitions of an FSM, then it becomes the pseudocode of a program for that model. The correctness of that program follows directly from the correctness of FSM-SIMULATOR, which is easily proved. Note that we can eliminate step (2) in FSM-SIMULATOR and let the first iteration of the while-loop in step (3) take care of step (2). However, we prefer to keep step (2) as shown here because otherwise one needs to understand that the while-loop will iterate at least once. If we resolve this by replacing the while-loop by a repeat-until loop with the exit-condition "s = end", one still needs to realize that s never becomes start-state again.

Algorithm FSM-SIMULATOR:

Input: A finite-state model of 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.

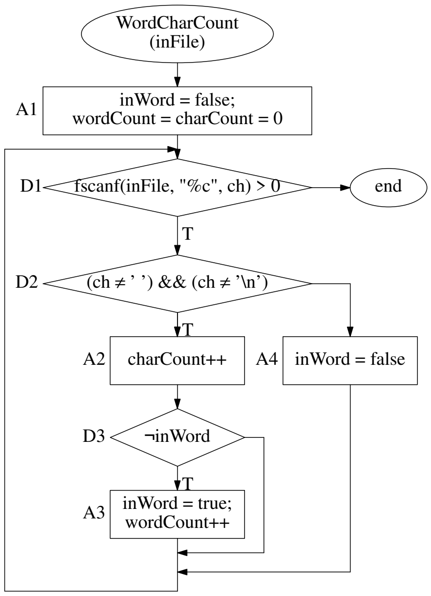
Remark: Suppose we instantiate the above algorithm with, say, the states and transitions of the FSM in Fig. 1(iii), and then we unfold the outer for-loop into a series of nested if-then-else, one for each state $s = s_i$, $s_i \neq$ start-state, and finally for each s_i we replace step (3a) by a series of nested if-then-else based on the conditions c_{i1}, c_{i2}, $\cdots$. Next, we form the flowchart from it and obtain the FSM using *dd*-paths as in Section 2.1. If we now eliminate the intermediate states which have a single incoming transition, then the reduced finite-state model becomes the same as that in Fig. 4.

5. States based on data-values

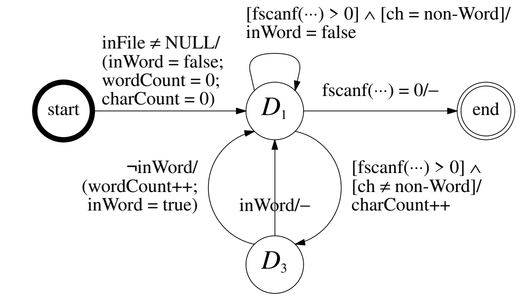
The presence of precedence and successor dependencies often causes major difficulties in understanding an FSM. The method given in Section 3.1 cannot help to eliminate two such dependent transitions if, for example, they are to the same state s_i or they are from the same state s_i. Sometimes, the direct use of data-values resulting from the actions of these transitions provides a way out by decomposing the state s_i into two or more states in terms of those data-values. We illustrate this technique by considering a slightly different situation for the wordCharCount problem considered earlier; we assume now that the words in the input textfile can have arbitrarily large length. This means we must process one character at a time from the input file to detect the end-of-word for each word.

Example 3. Fig. 5(i) shows a flowchart of the new WordCharCount function, where we assume for simplicity that the words are separated by two possible non-word characters: blanks and newlines. The variable `inWord`, which is set `true` when we encounter the first character of a word, controls the updates to `word-Count`. Fig. 5(ii) shows the FSM based on the *dd*-paths in Fig. 5(i) after eliminating the transition (D_1, D_2), which has the null-action and which is precedent-dependent on both (D_2, D_3) and (D_2, D_1); this also eliminates the state D_2. In Fig. 5(ii), leftside transition (D_3, D_1) is precedent-dependent on the center transition (D_3, D_1); also (D_1, D_3) is precedent dependent on both of them. The first character of a word is processed by the leftside transition (D_3, D_1) after going through (D_1, D_3), and it sets `inWord` = true; each remaining character of that word is processed by one traversal of (D_1, D_3) followed by the center transition (D_3, D_1). If we eliminate the only transition (D_1, D_3) to D_3, we get the FSM in Fig. 5(iii); here, the above precedent-dependency is carried by the corresponding loop-transitions (D_1, D_1) on the bottom side of D_1. On the other hand, merging the two parallel transitions (D_3, D_1) in Fig. 5(ii) into a single transition does not solve the problem completely; moreover, one has to unfold the merged transition in order to show that the FSM correctly computes `wordCount` and `charCount`. The problem lies in the design of the underlying algorithm in Fig. 5(i), and the solution requires a redesign of the algorithm.

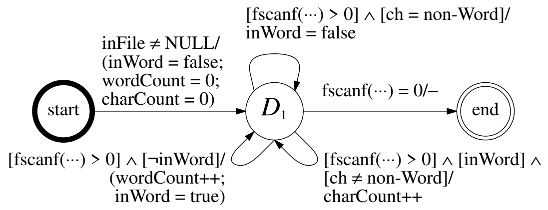
We obtain the first solution in Fig. 6(i) by decomposing the state D_1 in Fig. 5(iii) into two states D_{11}: "`inWord` = false" and D_{12}: "`inWord` = true", based on the values of `inWord` which controls the selection of the transition from D_3 to D_1 in Fig. 5(ii). We can think of D_{11} as "D_1 and `inWord` = false" or simply as "`inWord` = false" and D_{12} as "`inWord` = true". The FSM in Fig. 6(i) has no non-trivial precedent-dependency; also, the variable `inWord` is no longer needed as such and it is used in the state-names D_{11} and D_{12} only to help us conceptualize these states and determine the transitions. Fig. 7 shows a program P' which has a different flowchart from Fig. 5(i) and whose $M(P')$ is given by Fig. 6(i) after the usual simplifications, including the elimination of a transition (D_{12}, D_{11}) with the condition-part "fscanf(⋯) = 0" (not shown in Fig. 6(i)) which is both successor-dependent on (D_{11}, end) and precedent-dependent on (D_{11}, D_{12}). The elimination of that transition with (D_{11}, end) gives transition (D_{12}, end) in Fig. 6(i). The flowchart in Fig. 5(i) has a single while-loop and the program P' in Fig. 7 has two while-loops, but P' has one less variable; also, we can eliminate the break-statement in P' by building its associated if-test into the exit-condition of the inner while-loop.



(i) The flowchart of a new WordCharCount function.



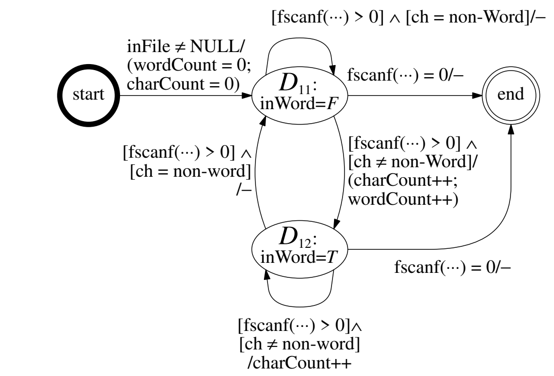
(ii) After elimination of the transition for the *dd*-path (D_1, D_2) which has the null-action.



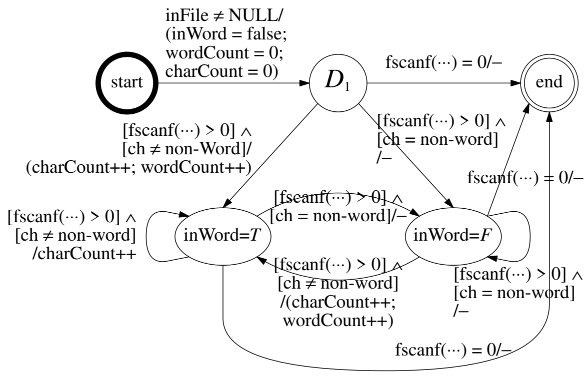
(iii) After elimination of the transition (D_1, D_3) which precede both the transitions from D_3 to D_1.

Figure 5. The flowchart of a new WordCharCount algorithm, its canonical FSM, and a simplified form of it.

A second solution is shown by the FSM in Fig. 6(ii), which is obtained by splitting the state D_3 in Fig. 5(ii) as before. However, since the states D_1 and "`inWord` = false" are now strongly equivalent, they may be merged, giving the FSM in Fig. 6(i). □



(i) The result of splitting D_1 in Fig. 5(iii) into two states based on values of "inWord".



(ii) An alternate FSM obtained by replacing D_3 in Fig. 5(iii) by two states based on the values of `inWord`.

Figure 6. Elimination of precedence-dependency in Fig. 5(iii) by using data-value based states.

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

Figure 7. A program corresponding to the FSM in Fig. 5(iv).

6. Conclusion

We have presented a simple method for constructing a finite-state model $M(P)$ from a program P and have shown that certain design flaws in P can be detected and corrected by analyzing $M(P)$. We have also shown that defining the states in terms of data-values (and their abstractions) is more powerful in eliminating design flaws related to precedent and successor dependencies among the transitions in the model. Building states based on data-value is also a better approach in practice where one must build the model from the requirements before the software is developed. We need many additional techniques for identifying and eliminating design flaws in a finite-state model. We also need techniques for refinements of a finite-state model similar to those for a dataflow diagram. We believe that a generalization of the notion of path-closed subsets in [8] would be useful for this purpose.

7. References

1. Aho, A. and Ullman, J.D., *Principles of Compiler Design,* Addison-Wesley Publ. Co., Mass., 1977.
2. Beizer, B. *Software testing techniques,* Van Nostrand Reinhold Comp, New York, 1983.
3. Chow, T., Testing software design modeled by finite-state machines, *IEEE Trans. Soft. Engg,* 4(1978), pp. 178-187.
4. Ghezzi, C., Jazayeri, M., and Mandrioli, D., *Fundamentals of software engineering,* Prentice-Hall, Inc., USA, 1991.
5. Gurevitch, Y., Sequential abstract state machines capture sequential algorithms, ACM Trans. Computational Logic, 1(2000), pp. 77-111.
6. Harel, D., Statecharts: a visual formalism for complex systems, *Science of Computer Program.,* 8(1987), pp. 231-274.
7. Heimdahl, M.P.E. and Whalen, M.W., Reduction and slicing of hierarchical state machines, *Proc. 5th ACM SIGSOFT Symp. on Foundations of Soft. Engg.,* 1997.
8. Kim, Y.G., Hong, H.S., Bae, D.H., and Cha, S.D., Test case generation from UML state diagrams, *IEE Proc. of Software,* 146(1999), pp. 187-192.
9. Kundu, S., The concept of path-closed subsets and its application to software functional design, *Proc. 7th Asia Pacific Software Engg. Conf, APSEC-2000, Singapore, Dec 5-8, 2000, pp. 112-119.*
10. Kundu, S., Finite state modeling in software design: some fundamental techniques, Tech. Report, Computer Sc. Dept., Louisiana State University, 2002.
11. Lewis, H.L. and Papadimitriou, C.H., *Elements of the theory of computation*, Prentice-Hall, 1998.
12. Peled, D.A., *Software reliability methods,* Springer-Verlag, 2001.
13. Savage, J.E., *Models of computation,*, Addison-Wesley, 1998.
14. Sommerville, I., *Software engineering* (5th ed.), Addison-Wesley Publ., England, 1995.