Inferring Flow of Control in Program Synthesis by Example

Stefan Schrödl and Stefan Edelkamp

Institut für Informatik
Albert-Ludwigs-Universität, Am Flughafen 17, D-79110 Freiburg, Germany,
e-mail:{schroedl,edelkamp}@informatik.uni-freiburg.de

Abstract. We present a supervised, interactive learning technique that infers control structures of computer programs from user-demonstrated traces. A two-stage process is applied: first, a minimal deterministic finite automaton (DFA) M labeled by the instructions of the program is learned from a set of example traces and membership queries to the user. It accepts all prefixes of traces of the target program. The number of queries is bounded by $O(k \cdot |M|)$, with k being the total number of instructions in the initial example traces. In the second step we parse this automaton into a high-level programming language in $O(|M|^2)$ steps, replacing jumps by conditional control structures.

1 Introduction

1.1 Program Synthesis from Examples

The ultimate goal of program synthesis from examples is to teach the computer to infer general programs by specifying a set of desired input/output data pairs. Unfortunately, the class of total recursive functions is not identifiable in the limit [7]. For tractable and efficient learning algorithms either the class has to be restricted or more information has to be provided by a cooperative teacher.

Two orthogonal strains of research can be identified [5]. Until the late 1970s, the focus was on inferring functional (e.g., Lisp) programs based on traces. Since the early 1980s the attention shifted towards model-based and logic approaches.

All functional program synthesis mechanism are based on two phases: *trace generation* from input/output examples, and *trace generalization* into a recursive program. Biermann's *function merging mechanism* [3] takes a one-parameter Lisp function whose only predicate is *atom*, and decomposes the output in an algorithmic way into a set of nested basic functions. Subsequently, they are merged into a minimal set that preserves the original computations by introducing discriminant predicates. These mechanisms perform well on predicates that involve structural manipulation of their parameters, such as list concatenation or reversal. However, their drawbacks are two-fold. The functional mapping between input and output terms cannot be determined in this straightforward way for less restrictive applications; on the other hand, manually feeding the inference algorithm with example traces can be a tedious and error-prone task. Secondly, the merging algorithms require exponential time in general.

The second direction of research (frequently called *Inductive Logic Programming*) is at the intersection between empirical learning and logic programming. A pioneering work was Shapiro's *Model Inference System* [11] as a mechanism for synthesizing Prolog programs from positive and negative facts. The system explores the search space of clauses using a configurable strategy. The subsumption relation assists in specializing incorrect clauses implying wrong examples, and in adding new clauses for uncovered ones. The critical issues are the undecidability of subsumption in the general case, the large number of required examples, and the huge size of the search space.

1.2 Programming in the Graphical User Interface

The last decades have seen a revolutionary change in human-computer interfaces. Instead of merely typing cryptic commands into a console, the user is given the illusion of moving around objects on a "desktop" he already knows from his everyday-life experience. Users can refer to an action by *simply performing* the action, something they already know how to do. Therefore, they can more easily handle *end user programming* tools.

Many spreadsheet programs and telecommunication programs have built-in *macro recorders*. Similarly to a tape recorder, the user presses the "record" button, performs a series of keystrokes or mouse clicks, presses "stop", and then invokes "play" to replay the entire sequence. Frequently, the macro itself is internally represented in a higher programming language (such as Excel macros in Visual Basic).

Moreover, the current trends in software development tools show that even programming can profit from graphical support. "Visual computing" aims at relieving conventional programming from the need of mapping a visual representation of objects being moved about the screen into a completely different textual representation of those actions. In an ideal general-purpose programming scenario, we could think of a domain-independent graphical representation for standard data structures, such as arrays, lists, trees, etc. which can be visually manipulated by the user.

Cypher gives an overview of current approaches [4]. Lieberman's *Tinker* system permits a beginning programmer to write Lisp programs by providing concrete examples of input data, and typing Lisp expressions or providing mouse input that directs the system how to handle each example. The user may present multiple examples that serve to incrementally illustrate different cases in conditional procedures. The system subsequently prompts the user for a distinguishing test. However, no learning of program structures takes place.

Based on these observations, we argue that program synthesis from traces could regain some attraction. The burden of trace generation can be greatly alleviated by a graphical user interface and thus becomes feasible.

In this paper, we propose an efficient interactive learning algorithm which solves the complexity problem of the merging algorithm in functional program synthesis. Contrary to the latter approach, we focus on imperative programming languages. They also reflect more closely the iterative nature of interaction with graphical user interfaces. The flow of control in imperative languages is constituted by conditional branches and loops; their lack in most current macro recorders is an apparent limitation.

2 Editing a First Example Trace

Figure 1 shows our prototypical graphical support. The user generates a first example trace by performing a sequence of mouse selections, mouse drags, menu selections, and key strokes.

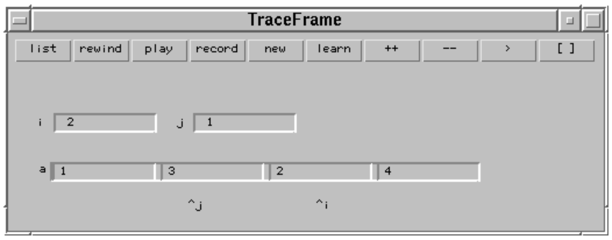

Fig. 1. Trace Frame.

Throughout the paper, we will exemplify the inference mechanism with the well-known *bubble-sort* algorithm. The user might start with the sample array `a` = [2, 1] of length `n` = 2. A variable `i` is introduced to hold the number of remaining iterations, and is initialized to one (`int i=n-1`). Then he states that the end is not yet reached (`i`>0). Subsequently he initializes another variable `j` to zero, meant as an index for traversing the array (`int j=0`). Now the array element with index 0 is compared to its successor (`a[j]>a[j+1]`). Since the comparison 1>2 fails (`F`) he swaps the elements (`swap(j,j+1)`). For ease of exposition, we assume that the `swap`-procedure has already been programmed to interchange two values in the array. The user increases `j` (`j++`) and then observes that the array has been traversed up to position `i` (`j<i; F`) in which case `i` is decremented (`i--`). The next iteration starts. But since `i` now has reached the left border (`i>0; F`) the sorting is accomplished and the procedure stops (`return`). In summary, the example generated by the end user is given as follows: `i=0;i>0;T;j=0;a[j]<a[j+1];F;swap(j,j+1);j++;j<i;F;i--;i>0;F;return`.

3 The *ID*-Algorithm

Grammar inference is defined as the process of learning an unknown grammar given a finite set of labeled examples. An important, widely used subset of formal languages are *regular grammars*, which can be generated and recognized by deterministic finite automata (DFA). However, given a finite set of positive examples and a finite, possibly empty set of negative examples, the problem of learning a minimum state DFA equivalent to the target is *NP*-hard [8]. Hence, the learner's task has to be simplified by imposing certain desired criteria on the examples (like structural completeness, characteristic samples), or by providing the learner with access to sources of additional information, like a knowledgeable teacher (oracle) who responds to queries generated by the learner.

Our algorithm is based on Angluin's *ID*-algorithm which is briefly recalled in this section. It may be skipped in a first reading.

Let Σ be the set of symbols, Σ^* be the set of strings, and λ be the empty string. Furthermore, let $M = (Q, \delta, \Sigma, q_0, F)$ be a DFA according to the usual quintuple definition and $L(M)$ be the language accepted by M. A state q in M is *alive* if it can be reached by some string α and left with some string β such that $\alpha\beta \in L(M)$. In a minimal DFA there is only one state d_0 that is not alive. A set of strings P is said to be *live-complete* w.r.t. M if for every live state q in M there exists a string $\alpha \in P$ such that $\delta(q_0, \alpha) = q$. Therefore, $P' = P \cup \{d_0\}$ represents all states in M. In order to find a string representation of the state reached on reading an input b from the state represented by α we define a function $f : P' \times \Sigma \to \Sigma^* \cup \{d_0\}$ by $f(d_0, b) = d_0$ and $f(\alpha, b) = \alpha b$. The *transition set* T' denotes the set of all elements of P', together with all elements $f(\alpha, b)$ for all $(\alpha, b) \in P \times \Sigma$. Analogously to P we define $T = T' - \{d_0\}$.

Input: a live complete set P and a teacher to answer membership queries
Output: a description of the canonical DFA M for the target regular grammar

$i = 0; v_i = \lambda; V = \{\lambda\}$, $T = P \cup \{f(\alpha, b) | (\alpha, b) \in P \times \Sigma\}$; $T' = T \cup \{d_0\}$, $E_0(d_0) = \emptyset$;
for each $\alpha \in T$
  if $(\alpha \in L)$ $E_0(\alpha) = \{\lambda\}$ **else** $E_0(\alpha) = \emptyset$**;**
while $(\exists \alpha, \beta \in P'$ and $b \in \Sigma$ such that $E_i(\alpha) == E_i(\beta)$ but $E_i(f(\alpha, b)) \neq E_i(f(\beta, b)))$
  let $\gamma \in E_i(f(\alpha, b)) \oplus E_i(f(\beta, b))$
  let $v_{i+1} = b\gamma$
  let $V = V \cup \{v_{i+1}\}$ and $i = i + 1$
  for each $\alpha \in T$
    if $(\alpha v_i \in L)$ $E_i(\alpha) = E_{i-1}(\alpha) \cup \{v_i\}$; **else** $E_i(\alpha) = E_{i-1}(\alpha)$;
Extract the automaton M for L from the sets E_i and T (see text)

Fig. 2. Angluin's *ID*-algorithm.

The goal of the *ID* algorithm (Figure 2) is to construct a partition of T' that places all the equivalent elements in one state [2]. The equivalence relation is the Nerode relation such that the resulting DFA will be minimal [1]. The algorithm starts with an initial partition of one accepting and one non-accepting state and refines it successively. In each step i of *ID* a string v_i is drawn such that for any two states q and q' there exists a $j \leq i$ with $\delta(q, v_j) \in F$ and $\delta(q', v_j) \notin F$ or vice versa. Thus, we define the i-th partition E_i as follows: $E_i(d_0) = \emptyset$ and $E_i(\alpha) = \{v_j | j \leq i, \alpha v_j \in L(M)\}$. Then for every two strings $\alpha, \beta \in T$ with $\delta(q_0, \alpha) = \delta(q_0, \beta)$ we have $E_j(\alpha) = E_j(\beta)$ for all $j \leq i$. For each i the algorithm searches for a separating pair α, β and a symbol b such that $E_i(\alpha) = E_i(\beta)$ but $E_i(f(\alpha, b)) \neq E_i(f(\beta, b))$. Let γ be any string that is either in $E_i(f(\alpha, b))$ and not in $E_i(f(\beta, b))$ or vice versa. Then we define $v_{i+1} = b\gamma$ and construct the $(i+1)$-th partition as follows. For each $\alpha \in T$ we query the string αv_{i+1}. If $\alpha v_{i+1} \in L(M)$ we set $E_{i+1} = E_i \cup \{v_{i+1}\}$; otherwise, we let $E_{i+1} = E_i$ unchanged.

We iterate until no separating pair α, β exists and extract M from the sets E_i and the transition set T as follows. The states of M are the sets $E_i(\alpha)$, for $\alpha \in T$. The initial state of M is $E_i(\lambda)$. The accepting states of M are the sets $E_i(\alpha)$, where $\alpha \in T$ and $\lambda \in E_i(\alpha)$. If $E_i(\alpha) = \emptyset$ then we add self loops on the state $E_i(\alpha)$ for all $b \in \Sigma$; else we set the transition $\delta(E_i(\alpha), b) = E_i(f(\alpha, b))$ for all $\alpha \in P$ and $b \in \Sigma$.

Angluin proved that *ID* asks no more than $n \cdot |\Sigma| \cdot |P|$ queries, where n is the number of states in M: the algorithm iterates through the *while*-loop at most n times, since each time at least one set E_i (corresponding to a state) is partitioned into two subsets. It asks $|T|$ questions, where T contains no more than $|\Sigma| \cdot |P|$ elements.

4 Customizing *ID* for Program Traces

4.1 Naive Approach

A simple strategy to apply the *ID*-algorithm to the problem of program inference from traces goes as follows. The alphabet Σ consists of all program lines occurring in the examples. More precisely, we partition Σ into $\Gamma \cup \Lambda \cup \Delta \cup \{\texttt{return}\}$, where Γ is the set of non-branching instructions (e.g. assignments), Λ is the set of (boolean) tests (e.g. numerical comparisons), $\Delta = \{\texttt{T}, \texttt{F}\}$ is the set of boolean values, and `return` signals the end of the procedure. The language L to be learned is regular and consists of all prefixes of valid execution traces. Programs are represented as finite state machines, where transitions are labeled with the respective instructions. Let $Pr(\alpha)$ be the set of all prefixes to α. The live-complete set P for the *ID*-algorithm can now be fixed as $P = Pr(S) \cup \{\lambda\}$, with S being the example trace.

For the initial examples in P, the user is free to choose any data, such as the array $[2, 1]$ in our case. As a heuristic guideline, the first examples are supposed both not to be overly lengthy (in order to reduce the number of subsequent questions), but at the same time cover all states of the automaton (in order to specify P). However, this requirement is not compulsory: in the version *IID* of the algorithm [10], the initial set of examples need not to be *live-complete*; the user is allowed to incrementally refine the automaton structure by presenting additional (positive or negative) examples later on.

Using this scheme, the number of queries (2158) asked for our bubble-sort case is clearly inacceptably high. Fortunately, the majority of them can immediately be answered by the system itself.

4.2 Pruning

We make the following general assumptions to hold for all execution traces α in Σ^*.

1. If $\alpha a \in L$ for some $a \in \Sigma$ then also $\alpha \in L$. In words: every prefix of a word in L is itself in L.
2. If $\alpha ab \in L$ and $\alpha ac \in L$ where $a \in \Gamma \cup \Delta$ and $b, c \in \Sigma$, then $b = c$. In words: There is only one instruction that follows a non-branching instruction or a boolean.
3. If $\alpha ab \in L$ and $a \in \Lambda$ then $b \in \Delta$. In words: A test is only followed by a boolean denoting its outcome.
4. We have $\alpha ab \notin L$ for $a =$ `return` and all $b \in \Sigma$. In words: No instruction may follow the end statement.

If condition 3. or 4. is violated, the trace is malformed and is hence rejected.

According to condition 1., we can efficiently store both the example traces and the query traces confirmed by the user in a *trie* data structure [9]. The bold path in Figure 3 corresponds to the first example trace of Section 2. Given a query string $\alpha b \gamma$, we tentatively insert it into the trie. If it is already contained, the answer is "yes". If the new trie forks at a non-branching instruction, condition 2. is violated and thus the answer is "no". Otherwise, the user is prompted. Unless his response is positive, the query string is removed.

For example at branch (3) in Figure 3 the system asks: `int i=n-1; i>0;T; int j=0; a[j]>a[j+1]; T; swap(j,j+1); j++; j<i; T; int i=n-1;` $\in L$? The user will answer "no".

In the further course of the session, the system will eventually "guess" all possible instructions as $b\gamma$ until the correct one `a[j]>a[j+1]` is found. As a further simplification, we can allow the user to edit the question and to immediately type in the right continuation.

4.3 Selection of Example Data

Ideally, the system should present its queries by animating a sequence of instructions for a suitable instantiation of the variables. Given only the raw code fragments, it might be difficult for the user to find the correct continuation.

This raises the question of how to select data which is consistent with a given trace, i.e., how to find an assignment to the variables that makes one choice point true and another one which makes it false. Two options are conceivable: the user could be asked to give a pool of examples independently of (prior or alternating to) the learning process, from which the system can choose some appropriate one. Alternatively, he can provide a specification to generate random data. E.g.,

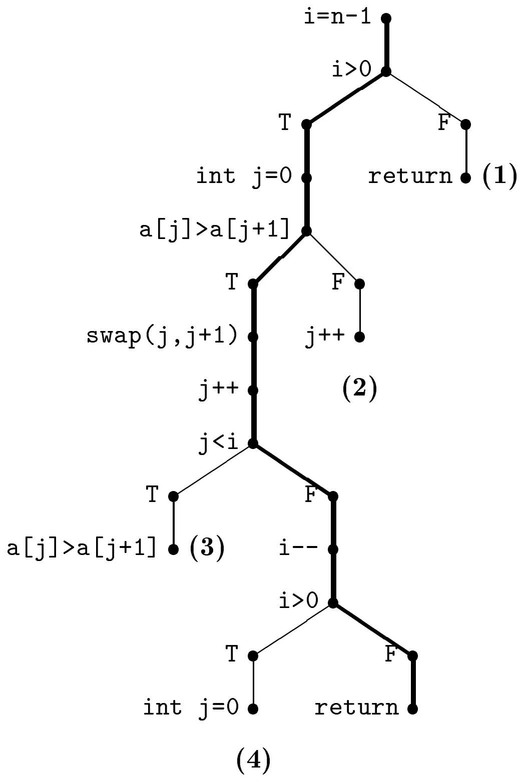

Fig. 3. Trie of example traces (bold edges) and query results (thin).

the bubble-sort algorithm should sort every permutation of the array elements, which we w.l.o.g. fix to be $[1, 2, \ldots, n]$. For instance, the array $a = [3, 1, 2]$ of length $n = 3$ leads to the following instantiation for question (3): `int i=2; 2=i>0; T; int j=0; 2=a[0]>a[1]=1; T; swap(0,1); j++; 1=j<i=2; T; i=2;` $\in L$? The user responds by replacing `i=2` by the next step which compares $3 = a[1] > a[2] = 2$, i.e., the test `a[j]>a[j+1]`.

Figure 4 depicts the finite state machine for the bubble-sort program inferred by the *ID*-algorithm. All states are accepting, and all omitted transitions lead to the dead state d_0.

4.4 Query Complexity

Every affirmatively answered membership question and every edited answer string inserts at least one node into the trie. Incrementally extending the trie in this way contributes to reduce the number of user questions. The total number is bounded by the size of the final trie minus that of the the initial one. In our bubble-sort example, this bound corresponds to the number of thin edges in Figure 3. Actually, the user is asked four instead of 2158 times.

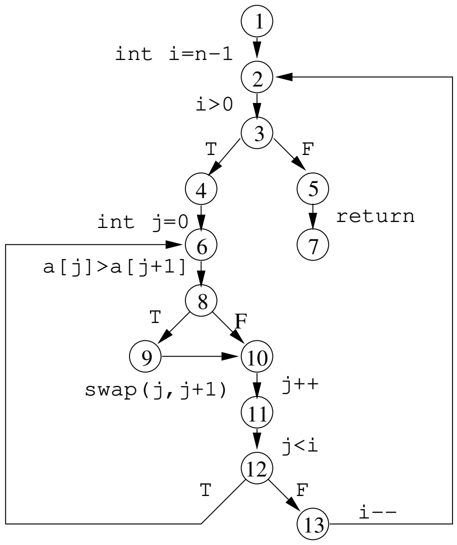

Fig. 4. DFA for bubble-sort.

Due to the restrictions on well-formed traces, we can specify a tighter upper bound on the number of user questions, compared to that of Angluin. If $\alpha \in T$ violates one of conditions 2. - 4., then $\alpha v_i \notin L$ for all distinguishing strings v_i. We count the number of the remaining valid elements $\tilde{T}$.

We assume that all given examples in P are complete traces, i.e. end with a `return` statement. Therefore, extensions $f(\alpha, b) \in T$ to $\alpha \in P$ are only available at proper prefixes of elements in the example set. However, if α ends with a non-branching instruction, restriction 2. constrains $f(\alpha, b)$ to be in the set P. In case α ends with a test instruction, condition 3. leaves us with two choices `T` and `F` for b. With k denoting the total number of tests in the example set, we have that $|\tilde{T}|$ is bounded by $k + |P|$. Finally, we conclude that the total number of membership queries is bounded by $n \cdot (k + |P|) = O(n \cdot |P|)$.

5 Transforming Automata into Structured Programs

It is straightforward to write down any generated automaton as a program using some form of jumps (e.g., *goto*-statements).

For more complex algorithms such flow charts quickly become confusing. In most current high-level programming languages, jump statements are either strongly discouraged (e.g., in C), or do not exist at all (e.g., in Pascal). Instead, high-level constructs are available for conditional branching and looping.

Therefore, we do not regard the automaton generated by the *ID*-algorithm as the final output, but rather apply a transformation in order to replace jumps by control structures.

Our algorithm transforms the automaton graph step by step by repeatedly collapsing a subgraph into a new edge, for which we keep track of extra information: its type (e.g., simple, test, sequence, *while*-loop, etc.), possibly its subcomponents, and the set of its successors.

Two adjacent edges labeled with arbitrary instructions other than tests or booleans can be merged into a *sequence* if they are the only inward or outward edges of the enclosed node. Connected tests can be merged into (compound) *conditions* containing boolean operators depending on the role of their `T`- and `F`-edges in the obvious way. For example, if test t_1 is connected to test t_2 via its `T`-edge, and the `F`-edges of both t_1 and t_2 point to the same node v, then a compound condition $t_1 \wedge t_2$ is formed whose `T`-edge leads to the same node pointed to by the `T`-edge of t_2, and whose `F`-successor-node is v.

The more interesting cases are the instances where control structures are inferred: a (simple or compound) condition whose `T`-edge leads to a non-test edge with successor node v, and whose `F`-successor-node is also v, can be merged into an *if-then*-statement pointing to v. Similarly, if the `T`- and the `F`-edge lead to different edges with the same successor node, then the resulting conditional statement additionally contains an else-part. A *while*-loop is a condition-edge c whose `T`-successor leads to an edge (i.e., the repeated block) which has, in turn, c as its successor. The resulting edge points to the destination of the `F`-successor-edge of c. If the two edges are interchanged, the condition in the generated *while*-statement is negated. In *do-while*-loops, the condition follows the edge for the repeated block.

First, the algorithm initializes the in-degrees of all nodes (in linear time). Then all n nodes are repeatedly checked for applicable transformation rules. If none is found, we are done; otherwise the automaton is altered accordingly, and the degree of affected nodes is adjusted. Both these operations require constant time. Since each transformations removes at least one node, at most n iterations are performed, giving an overall worst-case complexity of $O(n^2)$.

Note that, in principle, it is not always possible to transform jumps into control structures without reasoning about the semantics of a program or changing the set of variables (Fig. 5 sketches a critical loop structure). In these cases, the system should at least try to minimize the number of remaining *goto*s. Such graceful degradation is not covered by our algorithm and left as a topic for further research.

```
      int i=10, j=20;
l1 :  i--;
l2 :  if (j==0) return;
      j--;
      if (i > 0 && j>0) goto l1;
      goto l2;
```

Fig. 5. Without semantic information unfolding is impossible.

For our example, Figure 6 show the sequence of transformations applied to the original automaton of Figure 4. First, the edges $(6,8)$, $(8,9)$, $(8,10)$, and $(9,10)$ are collapsed into an *if-then*-statement (a). In the next step, the edge labeled `j++` is appended to form a *sequence* edge (b). Now we create a *do-while*-loop, since the test edge $(11,12)$ appears after the repeated block (c). The two next steps summarizes it, together with the edges with respective labels `int j=0` and `i--`, into a sequence (d). We create the outer *while*-loop (e), and then concatenate `int i=n-1` and `return` to it, such that only one edge is left corresponding to the final program (f).

6 Conclusion and Discussion

We have presented a supervised, interactive learning algorithm which infers control structures of computer programs from example traces and queries.

First, a deterministic finite automaton is learned by a customized version of the *ID*-algorithm for regular language inference. By exploiting the syntactical form of programs and allowing the user to incrementally type in instructions, the number of questions is reduced from an infeasible to a moderate scale. An upper bound of $O(n \cdot |P|)$ membership queries is given. Secondly, the resulting automaton is rewritten in a high-level language with control structures using an $O(n^2)$ algorithm.

An early precursor of this work similar in spirit is presented by Gaines [6]. His approach infers a DFA by exhaustive and exponential search until an automaton is found that is consistent with the given traces.

End users without programming knowledge can take benefit from inference of control structures. More powerful customization tools (e.g., macro recorders) are able to support them in solving more of the repetitive routine work which often needs elementary conditional branching and looping.

For the experienced programmer, the proposed inference mechanism might support the process of software development, mainly in view of integrity and incremental extensibility.

The final set of execution traces (as depicted by the resulting trie) uniquely determines the structure of the automaton. All source fragments in the generated program have been exercised in at least one example. Therefore, no untested code can arise. For recorded execution traces on concrete sample data, differences between the intended and the actual meaning of the program will occur by far more infrequently than bugs in programs developed without the control of explicit variable instantiations. In a way, both stages in the software development cycle, coding and testing, are performed more efficiently in parallel rather than in the usual alternating way.

A sequence of examples should start with simple examples and build to more complex and exceptional cases. Recursive and conditional procedures can be developed incrementally by starting with simple, “incorrect” definitions, and later adding more instances to handle more complicated and special purpose situations. Maintaining all used examples and only adding to this set ensures

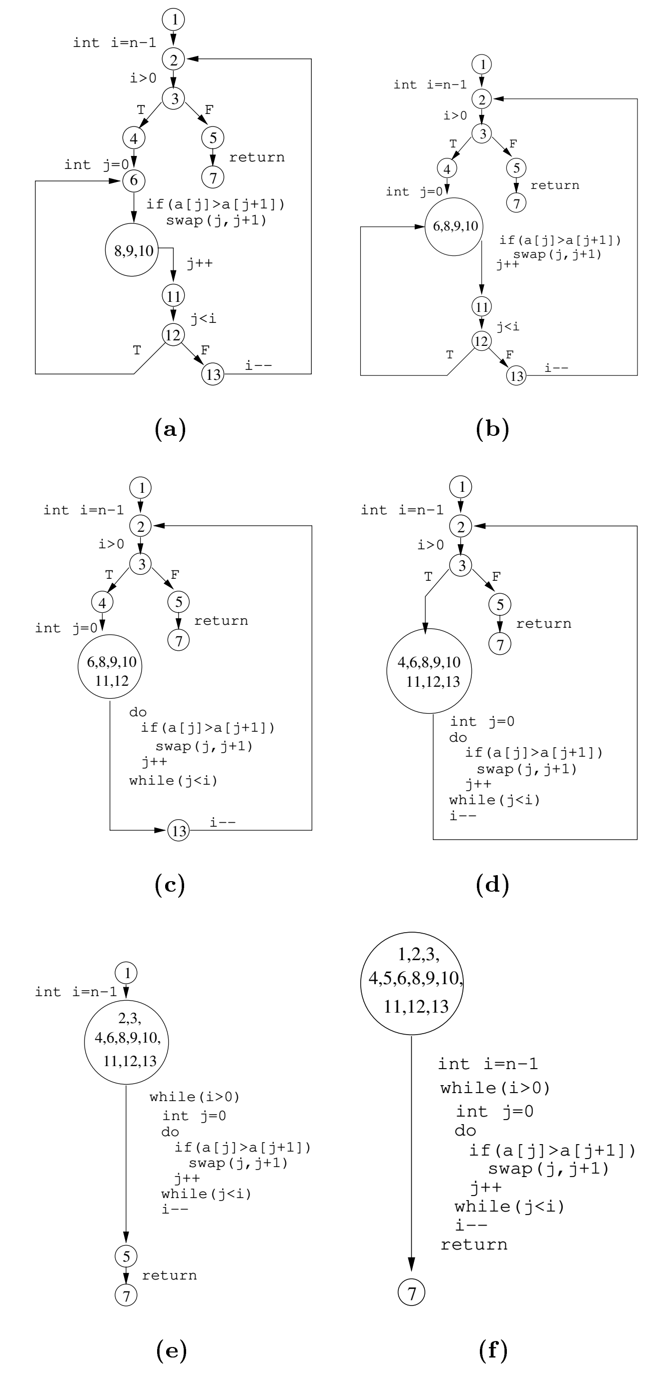

Fig. 6. Transformation of the DFA into a structured program.

that previous examples are still covered and that with growing complexity, no new bugs are introduced for cases which have already been successfully treated.

References

1. A. V. Aho, J. E. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms*. Addison–Wesley, 1974.
2. D. Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51(1):76–87, 1981.
3. A. W. Biermann. The inference of regular lisp programs from examples. *IEEE Trans. on Systems, Man, and Cybernetics*, 8(8):585–600, 1978.
4. A. Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.
5. P. Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Publishers, 1995.
6. B. Gaines. Behaviour/structure transformations under uncertainty. *International Journal of Man-Machine Studies*, 8(3):337–365, 1976.
7. E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.
8. E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.
9. D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, Reading, 1973.
10. R. Parekh, C. Nichitiu, and V. Honovar. A polynomial time incremental algorithm for regular grammar inference. Technical Report 97-03, Department of computer science, Iowa State University, 1997.
11. E. Y. Shapiro. *Algorithmic Program Debuggging*. PhD thesis, Yale University, 1983. Published under the same title by MIT press.