Forward-Chained Rules for Specifying Algorithms

Zeki O. Bayram

Computer Engineering Department
Bogaziçi University
Bebek 80815/Istanbul-Turkey
e-mail: bayram@boun.edu.tr

Abstract

Data-Directed programming, where the logic of computation is encoded using forward-chained rules, has been used primarily for expert system applications. In such applications, human expertise in some domain is expressed in the form of if-then rules and an inference engine is used to deduce new facts from existing ones. In this paper we demonstrate that forward-chained rules need not be limited to such a role and can be used to specify the logic of algorithms in a way that mimic the human understanding of these algorithms. If this idea is extrapolated into specifying software systems in general, the result is *self-documenting* and *executable* specifications.

Keywords: Expert system, forward chaining, context, heuristic, inference, algorithm, specification, executable

1. Introduction

We humans tend to think of algorithms in an operational way, often as transitions from one state to the next, with some start state and some ending state in mind. Consider someone describing how to get from one place to the next, or someone describing how to make tomato soup. These descriptions can very naturally be mapped onto state diagrams with transitions from one state to the next.

We take this idea and apply it to the specification of software algorithms. We use working memory states to represent states of the algorithm and forward-chained rules to represent the transitions. This works because a forward chaining inference engine implicitly traverses a dynamically constructed state machine, where the elements in the working memory at any time is the current state and the next state will be achieved through the application of a relevant rule to the current state.

2. A Language for Specifying Forward-Chained Rules

In [[1]] we described a probabilistic forward chained expert system shell with backtracking (SSST- State Space Search Tool). A non-probabilistic version of SSST has now been developed, which we call *Discrete SSST*, or DSSST, and we shall use DSSST to encode algorithms in the ensuing sections. First we give a brief introduction to DSSST.

2.1 Features of DSSST

DSSST features include

- a *context mechanism* for narrowing down the non-determinism at every transition point
- *rule priorities* which determine the order in which rules will be applied during normal execution and upon backtracking in case of more than one rule matching the current working memory
- *backtracking* in case a dead-end is reached in the search process
- *explicit success conditions* which specify when the search can terminate
- *explicit failure conditions* which specify forbidden working-memory states, causing immediate backtracking to take place in case they are satisfied by the working-memory state reached

These features allow us to specify the search space of working memory states declaratively, hence the name of the tool DSSST.

2.2 Syntax of DSSST

We informally give the form of DSSST syntactic elements as follows.

Rules:

```
rule( rule_name, rule_priority, list_of_contexts_in_which_the_rule_is_applicable,
      left_hand_side_conditions
      →
      right_hand_side_actions).
```

Success Conditions:

```
end_goal( success_condition_name,
          list_of_contexts_in_which_the_condition_is_applicable,
          conditions).
```

Failure Conditions:

```
fail_condition( fail_condition_name,
                list_of_contexts_in_which_the_condition_is_applicable,
                conditions).
```

The semantics of the constructs in DSSST are fairly intuitive and we do not elaborate on them much further. Conditions on the left hand side are patterns that match facts in the working memory, n-ary connectors "not_true", "one_true" and "not_true", as well as the built-in predicate "evaluate" which calls the underlying Prolog [[4]] interpreter to prove a goal (actually a constraint-solving variant of Prolog, CLP(R) [[7]] was used to implement DSSST).

Actions on the right hand side include **remove** for removing working memory elements, **make** for adding working memory elements, **modify** for modifying working memory elements, **add_context** for adding a context to the currently active contexts, and **remove_context** for making a currently active context inactive.

2.3 Declarations and actions needed to initialize DSSST

Working memory elements are also called "facts." Before a fact can be added to the working memory, its template must be made known to the system through the **literalize** command. As an example, if we wish to place information about a car into the working memory and that a car has the attributes **owner**, **color** and **age,** the command

```
?- literalize(car(owner,color,age)).
```

must first be given to the system. Similarly, any contexts that will be used must be declared at the start of a program run.

Working memory is initialized through the addition of facts by using the make command as in the following example.

```
?- make( car(owner george, color green, age 5)).
```

3. Specifying Algorithms using forward chained rules

In this section we specify algorithms for two different kinds of computational problems using forward chained rules. Note how in each case the algorithm specification mimics the way we would normally describe the workings of the algorithm.

3.1 Prim's Minimal Spanning Tree Algorithm expressed as a DSSST program

A *minimal spanning tree* for a weighted graph is a subgraph of the original graph that is a tree (i.e. contains no cycles and is connected) and contains all the nodes of the original graph. Prim's algorithm [[2]] starts from any node, and gradually builds a tree of minimum cost. At every iteration, an arc is included if it is the lowest cost arc among the

arcs which connect a node in the tree to a node outside the tree. The algorithm terminates when all the nodes of the original graph are included in the tree.

Below is the DSSST program that describes this algorithm. We will explain each part as we go along.

```
context(all).
?- add_context(all).
```

Here we are declaring "all" to be a context, and making it active.

```
?- literalize( arc(from,to,dist, status)).
/* status = in , out */

?- literalize( node(name, status)).
/* status = visited, not_visited */
```

We declare to DSSST that "arc" facts will have four attributes, and "node" facts will have two, with the names shown.

```
?- make(node(name a, status not_visited)).
?- make(node(name b, status visited)).
.........

?- make(arc( from a, to b, dist 3, status out)).
?- make(arc( from b, to c, dist 4, status out)).
.........
```

Here we initialize the working memory with facts regarding nodes and the arcs connecting them. The algorithm will start from node b.

```
end_goal( eg1,
        [all],
        not_true( node( status not_visited) ) ).
```

Here we specify when the algorithm should end: when it is not true that there exists a node that is not visited, i.e. when all nodes have been visited.

```
rule( compute, 10, [all],
   n1 := node(name N1, status visited) and
   n2 := node(name N2, status not_visited)  and
   a1 := arc(from N1, to N2, dist Dist, status out)  and
   not_true( node(name N1a, status visited) and
           node(name N2a, status not_visited) and
           arc(from N1a, to N2a, dist Dist_a,status out)  and
           evaluate( Dist_a<Dist))
   -->
   modify(n2, status visited) and
   modify(a1, status in)).
```

Finally, we have the rule (notice that only one rule is sufficient in this case) that builds the tree. Here is what it says: if there is a node **n1** that is visited (i.e. in the tree generated so far), a node **n2** that is not visited, an arc **a1** connecting **n1** and **n2**, and it is not true that there is another arc of lesser weight which connects **n1** and **n2** (i.e. **a1** is minimal), then modify **n2** so that its status field becomes **visited** (i.e. it is included in the spanning tree) and modify **a1** so that its status field becomes **in** (i.e. it also is included in the tree). When the inferencing stops, all arcs that have their status field set to **in** will be considered as included in the minimum spanning tree.

3.2 Specifying the generation of prime numbers in DSSST

Prime numbers are those integers that are divisible only by themselves and 1. Suppose we want to generate all primes up to a certain number n. One way to do it would be to start from 2, and check each odd number from 3 up to n (or n-1 if n is even) to see if it is divisible by an already generated prime number. If it is not, then we can include it as a prime number, and move on to the next odd number. Otherwise, we move directly on to the next odd number.

Here is the DSSST program that implements this algorithm.

```
context(all).
?- add_context(all).
```

Only one context is needed, and we call it **all**.

```
?- literalize( prime(number)).
?- literalize( globals(current, last)).

?- make(prime(number 2)).
?- make( globals( current 3, last 20)).
```

Here we declare that 2 is a prime number, that the current number to be tested is 3 and that we shall generate prime numbers up to 20.

```
end_goal( eg1,
        [all],
        globals( current X, last Y) and
        evaluate(X>Y) ).
```

Here we specify the termination condition: stop when the "current" number X is greater than the "last" number Y.

```
/* X=A*Y+R */
divide(X,A,Y,R):- X<Y, A=0, R=X,!.
divide(X,A,Y,R):- divide(X-Y,A2,Y,R),A=A2+1.
```

Here we define the **divide** predicate in Prolog which returns the quotient and remainder of a division operation. We will call this predicate as a left hand side condition to check for divisibility.

```
rule( found_prime, 10, [all],
   g := globals( current X) and
   not_true( prime(number Y) and
         evaluate( (divide(X,A,Y,R),R=0) ) )
   -->
   modify(g, current X+2) and
   make( prime(number X))).
```

Finally, we have the rules which compute the prime numbers. The above rule handles the case when the current number is indeed prime. It says this: If the number that is being currently considered (X) is not divisible by any previously generated prime number, register it as a prime number and move on to the next odd number. The rule below handles the case when X *is* divisible by a previously generated prime number and moves on to the next odd number without registering X as a prime number.

```
rule( not_prime, 10, [all],
   g := globals( current X) and
   prime(number Y) and
   evaluate( (divide(X,A,Y,R),R=0))
   -->
   modify(g, current X+2)).
```

4. Discussion

In both examples above, we note the closeness of the rule-based specifications of the algorithms to the verbal ones. We could in fact very easily generate textual descriptions of the algorithms just by reading out the rules. As such, algorithm specification using rules is self-documenting.

The reader should not be misled into thinking that algorithm specification using rules works only for small, insignificant problems. We have in fact used forward-chained rules to specify many kinds of graph algorithms, including breadth-first, depth-first, topological sort and Dijkstra's single-source-shortest path algorithms [[2]], AI problems

such as man-wolf-cabbage-goat, 8-queens [[5],[6]], as well as a queue simulation algorithm [[8]]. In all these cases, the rule-based specification could readily be translated into a verbal description of the algorithm and vice versa.

5. Related Work

Rules-based systems have mainly been used for capturing the knowledge in a specific domain traditionally requiring human expertise. [[3],[5],[6]] discuss many different rule-based systems. A survey of literature has failed to find a usage of rules for describing algorithms in the manner that was presented in this paper.

6. Conclusions and Future Work

We showed that rule-based systems can be used to describe algorithms in a way that mimic their natural language description. Furthermore, specifications specified in the form of rules are *executable*, hence increasing the designer's confidence that the specification represents a correct solution for the problem under consideration.

Further work in this area would be in the direction of using rules for specifying software systems in general. Executable specifications that are also self documenting would be very valuable form a software engineering point of view.

REFERENCES

[1] Bayram, Z. *SSST (State-Space Search Tool): A Probabilistic Forward-Chained Expert System Shell With Full Backtracking*, Proceedings of The Eleventh Symposium on Computer and Information Sciences, 1996

[2] Data Structures and Algorithms,.....

[3] Alty, J.L., Coombs,M.J. *Expert Systems*,NCC Publications, 1984

[4] Clocksin,W.F., Mellish,C.S., *Programming in Prolog*, Springer-Verlag, 1981

[5] Hayes-Roth,F., Waterman A.D., Lenat,D.B. (editors) *Building Expert Systems*, Addison Wesley, 1983

[6] Nilsson, N. J., *Principles of Artificial Intelligence*, Morgan Kauffman Publishers, 1986
[7] CLP(R) manual
[8] Dale, Data Structures in Pascal