

SPARCL Tutorial, part 2

(Copyright 2002 by Lindsey Spratt. All Rights Reserved.)

2: SPARCL representation and meaning.

This section gives a systematic treatment of the representation and meaning of basic concepts of SPARCL, and introduces sets.

The topics included are:

- simple data objects (constants and variables)
- sets
- matching as the fundamental operation on objects
- declarative (or nonprocedural) meaning of a program
- procedural meaning of a program
- relation between the declarative and procedural meanings of a program, and
- altering the procedural meaning by "delay" specifications.

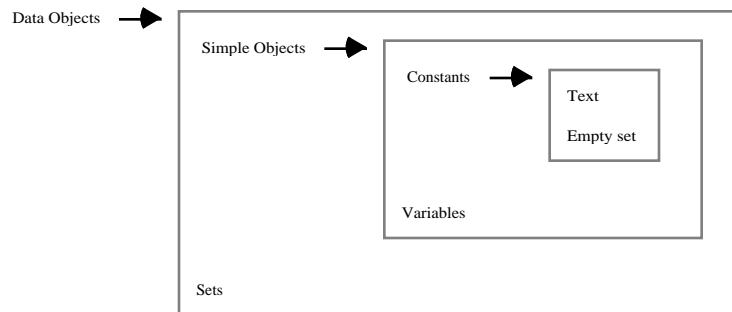


Figure 1: Classification of data objects.

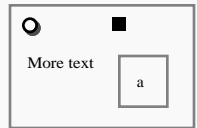
Object Type	Example Object
Text Constant	Some text
Empty set	■
Variable	○
Set	

Figure 2: Examples of some types of data objects (terms).

Most of these topics have already been reviewed in section 1. Here the treatment is more formal and detailed.

2.1: Data objects section

2.1.1: Term types discussion. Figure 1 shows a classification of data objects in SPARCL. The SPARCL system reflects the type of an object in its representation. There is a different kind of visual representation for each type of object. Figure 2 shows some examples of these data object (term) types.

We have already seen representations for text constants and variables in Chapter 1: text constants are represented by text, variables are represented by small circles. This is shown in the first and third entries in Figure 2. There are no explicit variable data type declarations in SPARCL programs. In SPARCL, text constants are used to represent numeric and non-numeric data.

SPARCL interprets a text constant as a number if a number-handling builtin predicate is being evaluated.

The second entry in Figure 2 shows the representation of an empty set. An empty set constant represents a set with no members.

Variables in SPARCL do not have names. Each small circle represents a distinct variable. If two variables are intended to refer to the same thing, then they are connected by a coreference link. Coreference links may connect any number of SPARCL terms (data objects). Most commonly they connect two variables. The terms connected by a single coreference link are all the same - they must unify with each other (unification is a special kind of matching). The terms joined by a coreference link must all be in the same clause. You have already seen some uses of coreference links in the example queries of Chapter 1.



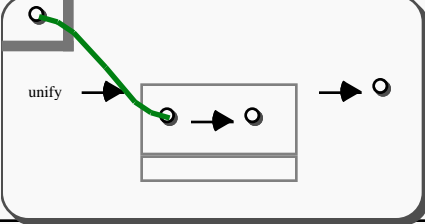

Set Type	Example Set										
Partitioned Set											
N-Tuple											
Intensional Set											
N-Tuple Table											
Function Table	<table border="1" data-bbox="917 829 1112 955"> <thead> <tr> <th>Column A</th> <th>Column B</th> </tr> </thead> <tbody> <tr> <td>a</td> <td>b</td> </tr> <tr> <td>1</td> <td>2</td> </tr> </tbody> </table>	Column A	Column B	a	b	1	2				
Column A	Column B										
a	b										
1	2										
Ordered, Factored Function Table	<table border="1" data-bbox="917 976 990 1186"> <tbody> <tr> <td>b</td> <td></td> </tr> <tr> <td>2</td> <td></td> </tr> <tr> <td>a</td> <td>c</td> </tr> <tr> <td>1</td> <td>3</td> </tr> <tr> <td>3</td> <td>1</td> </tr> </tbody> </table>	b		2		a	c	1	3	3	1
b											
2											
a	c										
1	3										
3	1										

Figure 3: Examples of some types of sets with specialized representations.

Sets are objects which contain any number of objects. The contained objects can, in turn, be sets. There are several representations of sets in SPARCL. Each representation is associated with a particular special set organization. We have already seen most of these set representations. They have been used to display other information in the examples of Chapter 1 and the earlier examples of this chapter.

Figure 3 shows most of the various types of set representations (some types of tables are not shown). The most general of the representations is the partitioned set. Any thing which can be represented using one of the other representations (e.g. N-Tuples) can be represented using a partitioned set, but the partitioned set representation may be much less concise.

The partitioned set is a collection of objects which is all in one part, or is divided into several parts. The example in the first entry of Figure 3 divides a set into three parts. The division means

that these three parts are all subsets of the whole set, that the union of these three sets is the entire set, and that no two parts "overlap"—every pair of parts has an empty intersection. This last constraint is the "pairwise disjoint" constraint. There are many uses of the single-part partitioned set in this section and in section 1.

The top part of the example partitioned set contains two terms, a constant and a variable. The variable could be bound to the same value as the constant, making it so that (after instantiation) this part contained only one distinct member. Or, the variable may be bound to a distinct term making this part contain two distinct members. The middle part contains no terms. It can be instantiated to a set of any number of terms; none (the empty set), 1 or many. This is, strictly speaking, another representation of a variable, where the variable *must* be bound to a set. The bottom part contains a single variable. Whatever the variable is instantiated to, this part must contain exactly one member. Due to the pairwise disjointness constraint on parts of a partitioned set, the variable in the top part and the variable in the bottom part can not be instantiated to the same term. This is an implicit "inequality" constraint.

There is an example of an N-tuple in the second entry of Figure 3. An N-tuple is a set of a particular construction:

- A 2-tuple (ordered pair) of $\langle X, Y \rangle$ is the set $\{\{X\}, \{X, Y\}\}$;
- An N-tuple of $\langle X_1, \dots, X_{(n-1)}, X_n \rangle$ equals $\langle \langle X_1, \dots, X_{(n-1)} \rangle, X_n \rangle$;
- An N-tuple of $\langle X \rangle$ equals X .

N-tuples have been used several times already in figures in this chapter and in Chapter 1.

The Intensional Set, or *IntenSet*, is a way of expressing "the set of all terms X such that X has property P"; symbolically this is " $\{X|P(X)\}$ ". An example of an *IntenSet* is shown in the third entry of Figure 3. The intensional set is a shorthand for the use of the "setof" builtin predicate: a way to specify the set of all terms such that some particular property (conjunction of literals) is true of those terms and no others. We will discuss it more in a later chapter.

Tables come in four basic varieties. Three examples are shown in the last three entries of Figure 3. Figure 3 is itself an example of a table (a "Function Table"). The basic types of tables are developed from combinations of two types of rows, N-tuple and function, and two types of row collections, set and N-tuple. Also, any kind of table may be "factored", where columns that are the same in all rows are extracted from the table and shown in a separate place. Finally, a table may have an initial row (a "0-th" row) which is treated as a single term instead of an N-tuple of terms. This last facility is particularly useful in defining a table which is a list of lists, where a list is an N-tuple with 'empty list' as its first element.

The N-Tuple Table (shown in entry four of Figure 3) is a set of rows, where each row is an N-tuple. All of the rows must be the same length. In this case, the rows are 2-tuples. The Function Table (shown in entry five of Figure 3) is a set of rows, where each row is a function. A

function is a set of ordered pairs (2-tuples), where the first element of the pair is the domain element and the second element of the pair is the range element. No two pairs in the function set can have the same domain element and different range elements. Such a set defines a function mapping from a domain to a range. All of the rows must have the same domains. In this case, the rows have the domain {"Column A", "Column B"}. The Ordered, Factored Function Table (shown in entry five of Figure 3) is an N-tuple of rows, where each row is a function, as for the Function Table. In this example, the ordered pair " $\langle b, 2 \rangle$ " has been factored out of the two rows.

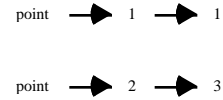


Figure 4: Two triples representing the points (1,1) and (2,3).

2.1.2: Discussion of simple geometry representation. The two most widely used set representations are partitioned sets and N-tuples. We can use these to conveniently represent many different kinds of

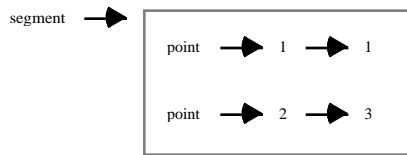


Figure 5: An ordered pair representing a line segment with end points at (1,1) and (2,3).

structures. In this section we see how some simple geometric objects can be represented.

The two 3-tuples in Figure 4 show a way to represent geometric points, one with $X = 1$ and $Y = 1$ and the other with $X = 2$ and $Y = 3$. An N-tuple is used here to provide a concise "naming" via element position to distinguish the X and the Y value.

The ordered pair (2-tuple) in Figure 5 represents a line segment with endpoints at (1,1) and (2,3). The 2-tuple is used here to "name" the endpoint data (telling us it defines a "segment"). The endpoint data is placed in a set, rather than in more elements of a N-tuple. This reflects the fact that there is no distinction between the two endpoints (this isn't a directed line).

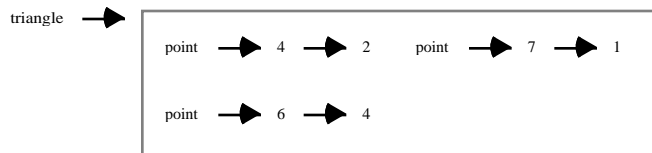


Figure 6: An ordered pair representing a triangle with corners at (4,2), (7,1), and (6,4).

The ordered pair in Figure 6 represents a triangle with three "corners" at (4,2), (6,4), and (7,1). The ordered pair is used to "name" the collection of points as describing a "triangle". The points are in a set since no ordering of the points is needed.

1. Exercise for section 2.1.

Develop a representation for rectangles, squares, and circles as structured SPARCL objects. Use an approach similar to that given in the "Geometry Example" program. Write single argument clauses which have example terms of each of these in their argument and use a comment in the argument to explain the elements of the terms (e.g. "a triangle is

represented by a set of three (X,Y) points which are the corners of the triangle"):

1. write a Rectangle clause of one argument which contains a term for a rectangle with diagonally opposite corners at (1,2) and (34,-5.8)
2. write a Circle clause of one argument which contains a term for a circle centered at (93,4) with a radius of 16
3. write a Square clause of one argument which contains a term for a square with upper left corner at (14, 23) and sides of length 123.

2.2:Term matching. In the previous section we have seen how terms can be used to represent complex data objects. The most important operation on terms is *matching*. A special kind of matching is used in SPARCL called *unification*. Matching alone can produce some interesting computation.

Given two terms, we say they *match* if:

- (1) they are identical
- (2) the variables in both terms can be instantiated

to objects in such a way that after the substitution of variables by these objects the terms become identical, and these instantiations don't violate any "current" partitioning constraints.

Using example predicates which define properties of line segments, we illustrate how matching alone can be used for interesting computation. Let us return to the simple geometric objects of the previous example and define a piece of program for recognizing horizontal and vertical line segments. The

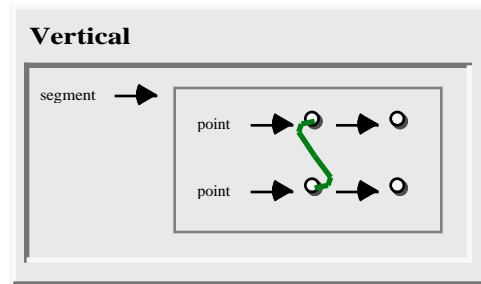


Figure 7: Clause defining the 'Vertical'/1 predicate.

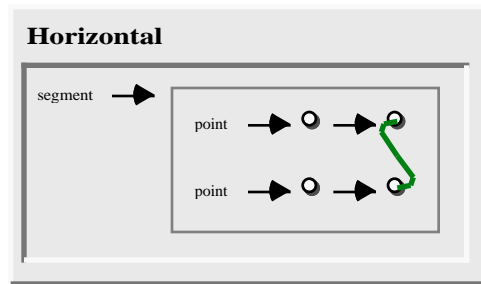


Figure 8: Clause defining the 'Horizontal'/1 predicate.

'Vertical'/1 predicate in Figure 7 is a property of segments, so it can be formalized in SPARCL as a unary relation. A segment is vertical if the x-coordinates of its end-points are equal, otherwise there is no other restriction on the segment. The property 'Horizontal'/1 is similarly formulated in Figure 8, with only the x and y interchanged.

The query in Figure 9 asks for the y value of a point such that there is a horizontal segment from (1,1) to that point with x = 2. The result in Figure 10 shows the desired y value to be 1.

Matching with the partitioned set term provides a very powerful tool. The 'Union'/3 predicate in Figure 11 relies entirely on this mechanism to relate two sets to the set which is their union.

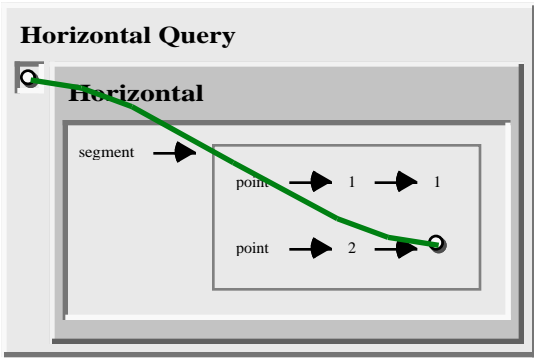


Figure 9: Clause for posing the query “What is the value of Y such that there is a horizontal segment from (1,1) to (2,Y).”

set C are connected and thus must be the same—must refer to the same set which we'll call Y. Since one of these parts is in set A, set Y must be a subset of set A. Since one of these parts is in set A with a part for set X (mentioned earlier), X and Y must have nothing in common (the pairwise disjointness constraint).

The bottom part of set B and the top part of set C are connected and thus must be the same—must refer to the same set which we'll call Z. Since one of these parts is in set B, set Z must be a subset of set B. Since one of these parts is in set B with a part for set X (mentioned earlier), X and Z must have nothing in common (the pairwise disjointness constraint).

Since Y and Z are referred to by these two parts within the same partition, Y and Z must not have any terms in common.

To summarize what we know about the sets A, B, C, X, Y, and Z:

$$A = X \cup Y$$

$$B = X \cup Z$$

$$C = X \cup Y \cup Z$$

$$X \cap Y = \emptyset \text{ \{from the partitioning of A\}}$$

$$X \cap Z = \emptyset \text{ \{from the partitioning of B\}}$$

$$Y \cap Z = \emptyset \text{ \{from the partitioning of C\}}$$

From these equations we can infer:

$$X = A \cap B$$

$$Y = A - B$$

$$Z = B - A$$

To infer the equation for X, suppose there is some term P which is common to A and B but is not in X, then this element must be in both Y and Z, but this

The bottom part of set A, the top part of set B, and the middle part of set C are connected and thus must be the same (i.e. they must unify with each other)--they must all "refer" to the same set which we'll call X. Since these parts are in sets A and B, X must be a set of terms common to A and B.

Horizontal Query → 1

Figure 10: Result of querying the clause in Figure 9.

The top part of set A and the bottom part of set C are connected and thus must be the same—must refer to the same set which we'll call Y. Since one of these parts is in set A, set Y must be a subset of set A. Since one of these parts is in set A with a part for set X (mentioned earlier), X and Y must have nothing in common (the pairwise disjointness constraint).

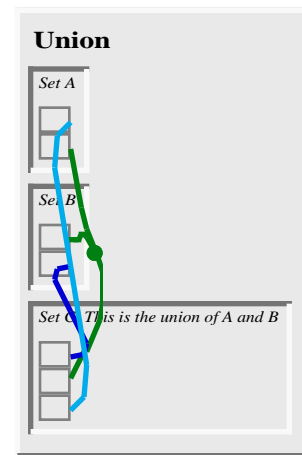


Figure 11: Clause defining the 'Union'/3 predicate.

violates the constraint that Y and Z are disjoint. The equation for Y derives from the equation for A as the union of X and Y and the equation for X as the intersection of A and B. The equation for Z is derived similarly to the equation for Y.

Finally, substituting the solutions for X, Y and Z into the equation for C:

$$C = (A \cap B) \cup (A - B) \cup (B - A)$$

which is equivalent to:

$$C = A \cup B$$

This is the desired result. The third argument of 'Union'/3 ("C") is the union of the first two arguments ("A" and "B").

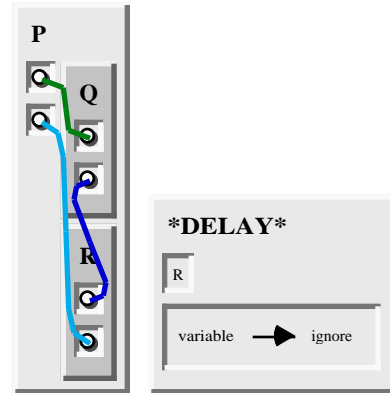


Figure 13: 'P'/2 example with two literals.

2.3: The declarative meaning of SPARCL. We have already seen in section 1 that SPARCL programs can be understood in two ways: declaratively and procedurally. In this and the next section we will consider a more formal definition of the declarative and procedural meanings of programs in basic SPARCL. But first let us look at the difference between the two meanings again.

Consider the clause in Figure 12. Some alternative declarative readings of this clause are:

P is true if Q and R are true.

From Q and R follows P.

Alternative procedural readings of this clause are:

To solve problem P, solve the subproblems Q and R.

To satisfy P, satisfy Q and R.



Figure 12: 'P'/0 example with two literals.

These procedural readings do not say the order in which to solve or satisfy Q and R. The information given (the lone clause for P) doesn't allow us to infer any particular ordering. However, SPARCL is a fundamentally sequential language. It will choose an order in which to solve Q and R. The programmer may affect this choice using '*DELAY*'/2 clauses, the 'ordered_disjunction'/2 builtin predicate, and the 'if'/3 builtin predicate.

To see the use of the '*DELAY*'/2 clause, consider the example in Figure 13. Predicates 'P'/2, 'Q'/2, and 'R'/2 each have two arguments and there is a '*DELAY*'/2 clause.

The 'P'/2 clause has a similar declarative reading as for the 'P'/0 clause in Figure 12:

P(X,Y) is true if Q(X,A) and R(A,Y) are true.

From $Q(X,A)$ and $R(A,Y)$ follows $P(X,Y)$.

The `*DELAY*/2` clause makes the procedural reading of the example in Figure 13 different from the procedural reading of the example in Figure 12. There is now a constraint on when the `'R'*/2` subgoal may be attempted:

To solve problem $P(X,Y)$, solve the subproblems $Q(X,A)$ and $R(A,Y)$. Delay any attempt to solve $R(A,Y)$ as long as A is an unbound variable.

Since the first argument of `'R'*/2`, in the clause for `'P'*/2`, can only be bound before attempting to solve `'R'*/2` if `'Q'*/2` has been solved, this `*DELAY*/2` constraint on `'R'*/2` has the effect of ordering attempts at solving the subgoals of `'P'*/2`.

Thus the difference between the declarative readings and the procedural ones is that the latter not only define the logical relations between the head of the clause and the goals in the body, but also (possibly) the *order* in which goals are processed.

2.3.1. A formalization of declarative meaning. The declarative meaning of programs determine whether a given goal is true, and if so, for what values of variables it is true. To precisely define the declarative meaning we need to introduce the concept of "instance" of a clause. An instance of a clause C is the clause C with each of its variables substituted by some term.

A goal G is true (that is, satisfiable, or logically follows from the program) if and only if

(1) there is a clause C in the program such that

(2) there is a clause instance I of C such that

(a) the head of I is identical to G ,

(b) all of the goals in the body of I are true,

(c) the instantiation of C to create I does not violate any partitioning constraints in C or I

This definition extends to SPARCL questions as follows. In general, a question to the SPARCL system is a set of goals. A set of goals is true if all of the goals in the set are true for the same instantiation of variables and that instantiation of variables does not violate any (implicit) partitioning constraints.

A set of goals, as just described, denotes the "conjunction" of those goals; they *all* have to be true. There is a builtin predicate in SPARCL which implements "disjunction"; `'ordered_disjunction'*/2`. This builtin is used in the clause for `'P'*/0` in Figure 14. The declarative reading of this is "P is true if Q or R is true." The procedural reading of this is ordered: To solve problem P, first try to solve subproblem Q. If this succeeds, then P is solved. Otherwise, if

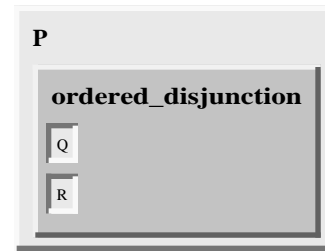


Figure 14: `'P'*/0 ordered_disjunction` example.

solving subproblem Q fails, then solve subproblem R. If this succeeds then P is solved.

The 'P'/0 clause with the 'ordered_disjunction'/2 literal in Figure 14 has the same declarative meaning as the two 'P'/0 clauses in Figure 15. The difference in meaning is only procedural. With these two clauses, there is no guarantee which ordering of the P clauses the SPARCL interpreter will use when attempting to solve the P problem.

2.4: The procedural meaning of SPARCL. The procedural meaning specifies *how* SPARCL answers questions. To answer a question means to try to satisfy a set of goals. They can be satisfied if the variable that occur in the goals can be instantiated in such a way that the goals logically follow from the program. Thus the procedural meaning of SPARCL is a procedure for executing a set of goals with respect to a given program. To "execute goals" means try to satisfy them.

Let us call this procedure "execute set". The inputs and the outputs for this procedure are:

input: a program, a goal set, and a set of partitioning constraints

output: a success/failure indicator and an instantiation of variables

The meaning of the two output results is as follows:

(1) The success/failure indicator is "yes" if the goals are satisfiable and "no" otherwise. We say that "yes" signals a successful termination and "no" a failure.

(2) An instantiation of variables is only produced in the case of a successful termination; in the case of failure there is no instantiation.

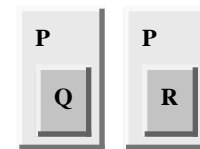


Figure 15: 'P'/0 example of disjunction implemented using two clauses.

In section 1, we discussed informally what procedure "execute set" does, under the heading "How SPARCL answers questions." What follows in this section is a more formal and systematic description of this process.

To execute a set of goals {G1, ..., Gm} with partitioning constraints P the procedure "execute" does the following:

- Order the goal set to create a list {G1', ..., Gm'}
- invoke "execute list".

To execute a LIST of goals [G1', ..., Gm'] with partitioning constraints P the procedure "execute list" does the following:

- If the goal list is empty then if the delayed goals list is empty terminate with "success", else terminate with "failure".
- If the goal list is not empty then divide the list into the first goal, G1', and the OtherGoals.

- If $G1'$ is the special "marker" goal 'end_body', then replace it with the DelayedGoalsList ([$DG1, \dots, DGk$]) to create [$DG1, \dots, DGk, G2', \dots, Gm'$] and recursively invoke `execute_list` with this new goal list and an empty delayed goals list. The result of the recursive invocation is the result of this invocation.
- Else, if $G1'$ is a goal which should be delayed according to the *DELAY* definitions in the program then add $G1'$ to the DelayedGoals and recursively invoke `execute_list` with the OtherGoals ([$G2', \dots, Gm'$]) and the extended DelayedGoals. The result of the recursive invocation is the result of this invocation.
- Otherwise (i.e. if GoalList is not empty, $G1'$ is not 'end_body', and $G1'$ does not need to be delayed) continue with (the following) operation called "SCANNING".
- SCANNING: Scan through the clauses in the program in any order until a clause, C , is found such that the head of C matches the first goal $G1'$ without violating the current partitioning constraints P . If there is no such clause then terminate with "failure".

If there is such a clause C with head H and body goals $\{B1, \dots, Bn\}$, then replace the variables of C with new variables (essentially, rename the variables of C) to obtain a variable C' of C , such that C' and the list $G1', \dots, Gm'$ have no common variables. Let C' have head H' and body $\{B1', \dots, Bn'\}$. Let $G1'$ match H' ; let the resulting instantiation of variables be S and extend partitioning constraints P to be P' .

Order the goal set $\{B1', \dots, Bn'\}$ to create the goal list

[$B1'', \dots, Bn''$].

In the goal list [$G1', \dots, Gm'$], replace $G1'$ with the list [$B1'', \dots, Bn''$], obtaining a new list

[$B1'', \dots, Bn'', G2', \dots, Gm'$].

(Note that if C is a fact then $n = 0$ and the new goal list is shorter than the original one; such shrinking of the goal list may eventually lead to the empty list and thereby a successful termination.)

Substitute the variables in this new goal list with new values as specified in the instantiation S , obtaining another goal list

[$B1''', \dots, Bn''', G2'', \dots, Gm''$]

- Execute (recursively with procedure "execute list") this new goal list. If the execution of this new goal list terminates with success then terminate the execution of the original goal list also with success. If the execution of the new goal list is not successful then abandon this new goal list and go back to SCANNING through the program. Continue the scanning with any untried clause and try to find a successful termination using some other clause.

This procedure can be written in a Pascal-like notation as shown in Figure 16. Several

additional remarks are in order here regarding the procedures "execute_set" and "execute_list" as presented. First, it was not explicitly described how the final resulting instantiation of variables is produced. It is the instantiation S which led to a successful terminate, and was possibly further refined by additional instantiations that were done in the nested recursive calls to "execute_list".

Whenever the recursive call within SCANNING to "execute_list" fails, the execution returns to SCANNING, continuing at the program that had been last used before. As the application of the clause C did not lead to a successful termination SPARCL has to try an alternative clause to proceed. What effectively happens is that SPARCL abandons this whole part of the

unsuccessful execution and backtracks to the point (clause C) where this failed branch of the execution was started. When the procedure backtracks to a certain point, all of the variable instantiations that were done after that point are undone. This ensures that SPARCL systematically examines all of the possible alternative paths of execution until one is found that eventually succeeds, or until all of them have been shown to fail.

The actual implementation of SPARCL adds many refinements to the execute procedures. One of them is to reduce the amount of scanning through the program clauses to improve efficiency. So SPARCL will not scan through all of the clauses of the program, but will only consider the clauses about the relation in the current goal.

```

procedure execute_set (Program, GoalSet, Constraints, Success)
begin
  OrderedGoalsList := order_goal_set(GoalSet);
  execute_list(Program, OrderedGoalList, [], Constraints, Success);
end;

procedure execute_list (Program, GoalList, DelayedGoals,
  Constraints, Success)
begin
  if empty(GoalList) then
  begin
    if empty(DelayedGoals) then
      Success := true
    else Success := false
    end
  else
  begin
    Goal := head(GoalList);
    OtherGoals := tail(GoalList);
    if Goal = end_body then
      begin
        NewGoals := append(DelayedGoals, OtherGoals);
        execute_list(Program, NewGoals, [], Constraints, Success);
      end
    else if delay(Goal, Program) then
      begin
        NewDelayedGoals := append(DelayedGoals, [Goal]);
        execute_list(Program, OtherGoals, NewDelayedGoals,
          Constraints, Success);
      end
    else
      begin
        Satisfied := false;
        while not Satisfied and "more clauses in program" do
          begin
            Let next clause in Program be
              head H and body {B1, ..., Bn}.
            Construct a variant of this clause
              head H' and body {B1', ..., Bn'}.
            match(Goal, H', Constraints, MatchOK, Instant,
              MatchConstraints);
            if MatchOK then
              begin
                OrderedBodyGoals := order_goal_set({B1', ..., Bn'});
                ExtendedBodyGoals := append(OrderedBodyGoals,
                  [end_body]);
                NewGoals := append(ExtendedBodyGoals, OtherGoals);
                NewGoals := substitute(Instant, NewGoals);
                execute_list(Program, NewGoals, MatchConstraints,
                  Satisfied);
              end
            end;
            Success := Satisfied
          end
        end
      end
    end;
  end
end;

```

Figure 16: execute_set and execute_list procedures.

Next section: 3. Presentation of an application of SPARCL