Shortest Path Algorithm

Formal Program Documentation

Dennis K. Peters
Software Engineering Research Group
Telecommunications Research Institute of Ontario
CRL, McMaster University, Hamilton, Ontario, Canada L8S 4K1

ABSTRACT

The display method is a technique for documenting the design of a program and presenting it such that it can be reviewed in small, independent pieces. This paper uses the display method to present a reasonably complicated program: an implementation of Dijkstra's 'shortest path algorithm'. The number of displays required to document a program can sometimes be reduced by using procedure specifications instead of program specifications, as is done in previous work using the display method. When writing procedure specifications care must be taken to describe the required behaviour under aliasing conditions. A discussion of specifications for C functions is given.

1.0 Introduction

In [6] Parnas et al. present a method for program documentation, known as the *display method*, in which a program is presented as a set of displays, which can each be independently verified. This paper illustrates how a variant of the display method can be used to understand and verify a reasonably complicated program. It also shows how the display method can be used with *procedure* specifications (as opposed to *program* specifications). The example used to illustrate these points is an implementation of Dijkstra's 'shortest path algorithm' as presented in [1] pp. 167-172.

2.0 The Display Method

In the Display Method a program is presented as a set of *displays* each of which contains three parts:

- P1:a specification of the portion of the program presented in this display,
- P2: the text of that part of the program, which may invoke other (sub-)programs, and
- P3: the specifications of all of the programs (other than that specified in P1) invoked in P2 that are not known[1].

Each display is a concise document, which can be used to determine the correctness of the portion of the program presented without needing to refer to other displays. The documen-

1. A known program is a program for which the semantics are assumed to be understood and hence a specification is not required.

tation also includes a lexicon, in which terms used in the documentation are defined. In the example presented in this paper terms that are defined in the lexicon are written in italics.

2.1 Specifications

As pointed out in [6], the display method is suitable for use with any program specification technique. In this paper, as in [6], programs are specified by giving the characteristic predicate of a Limited-Domain Relation (LD-Relation) describing the required behaviour. Briefly, an *LD-relation* is a pair, (relation, set), where the relation gives the state pairs (start state, stop state) that are considered to be acceptable behaviour of the program, and the set, known as the *competence set*, gives those start states for which the program is required to terminate. Usually in practice, as for all of the specifications in this paper, the competence set is the same as the domain of the relation, in which case it is omitted by convention. Also as in [6], tabular forms are used for representing LD-relations. For a more detailed discussion of LD-relations and their representation in tabular forms the reader is referred to [6].

2.2 Procedure Specifications

One unpleasant feature of the display method is that, if a procedure with arguments is used in a set of displays there will be a different specification for each invocation of the program, each of which must appear in P1 of a display in order for the set of displays to be complete. Thus, if the procedure is used more than once its body will appear in P2 of several displays: one for each time it is used. The reason for this is explained in [3]:

> The texts that we call "procedures with arguments" are not programs in the sense used …[in this paper]. In general, until the actual arguments are known (i.e. until we see the invocation) we do not know which actual elements of the data state will be affected by the execution nor do we know what effect the execution will have on the data structure. Consequently, procedures with arguments cannot be described by program functions (because determining the set of possible executions requires knowing the actual arguments used in an invocation).

In this paper, I illustrate how, in certain cases, it is possible to give a *specification* for a *procedure* in the form of an LD-relation and hence possibly reduce the number of displays required for a program (although this is not the case for the program presented in section 4.0). Procedure specifications must give the acceptable behaviour of all possible programs resulting from an invocation of the procedure.

The responsibilities of the inspectors of a set of displays are slightly different from the normal case when this approach is used. When inspecting a display in which P1 contains a procedure specification, the inspector must convince herself that any invocation of the procedure in P2 will satisfy the specification in P1 under the assumption that the programs invoked in P2 satisfy the specifications given in P3. Care must be taken in cases where P3 contains procedure specifications: the inspector must use knowledge of the parameter

binding semantics of the language to ensure that the procedures are specified to do the right thing for the invocations appearing in P2.

Writing procedure specifications is straightforward in instances where the parameter binding semantics are such that no aliasing could occur, but can be quite difficult if aliasing is a possibility. One solution that may be acceptable in many cases is to give the characteristic predicate of the procedure specification relation in the form "¬aliasing ⇒ normal behaviour", which means that any terminating behaviour is acceptable if aliasing occurs. If, in addition, the characteristic predicate of the competence set is given as "¬aliasing", then non-terminating behaviours are also acceptable where aliasing occurs. This has the advantage that both the programmer responsible for implementing the procedure, and the inspector charged with verifying it are relieved of the responsibility of detecting aliasing. Since detecting aliasing in general requires knowledge of the invocation (i.e. the actual variables passed as parameters) it is appropriate that the responsibility of avoiding aliasing be given to the programmer using a procedure. If a procedure is specified as described above, the programmer and inspector can make no assumptions about its behaviour if aliasing does occur.

2.2.1 Semantics of C Functions

The example presented in this paper is written using the C programming language. In this section I discuss the specifics of writing procedure specifications for C 'functions' (procedures).

> In C, all function arguments are passed "by value." This means that the called function is given the values of its arguments in temporary variables rather than the originals.[4] p. 27

It would seem from the above statement that aliasing should not be a problem in C since all formal arguments are local variables with their value set to equal the corresponding actual arguments at the time of invocation. However, since C allows 'pointer' data types (references) it is quite possible for aliasing to occur. Consider the following procedure, which compares two integers and replaces the largest with their difference.

```
void
gcdStep(int *x, int *y)
{
   if (*x > *y) {
      *x = *x - *y;
   } else {
      *y = *y - *x;
   }
}
```

In this example the variables `x` and `y` are pointers to other integer variables (i.e. in the calling procedure), and so the data structure for an invocation of `gcdStep` includes the variables pointed to by `x` and `y`, denoted `*x` and `*y`, as well as `x` and `y`. Note that since `x` and `y` are temporary variables (i.e. they only exist while the procedure is executing) it makes

no sense to place restrictions on their final values. One might be tempted to give the specification for gcdStep as follows[1].

void
gcdStep(int *x, int*y)

$R_{gcdStep}(,) =$

	'*x > '*y	'*y ≤ '*x
*x' =	'*x - '*y	'*x
*y' =	'*y	'*y - '*x

The procedure given does not satisfy this specification, however, since in the instance where `x` = `y` (i.e. `*x` and `*y` are the same variable) the condition that *x' = '*x will not be satisfied. However, if we are sure that we will not want to use this procedure in that situation, then it might be best to specify it as follows,.

void
gcdStep(int *x, int *y)

$R_{gcdStep}(,) = \neg('x = 'y) \Rightarrow$

	'*x > '*y	'*y ≤ '*x
*x' =	'*x - '*y	'*x
*y' =	'*y	'*y - '*x

This specification allows any terminating behaviour when `*x` and `*y` are the same variable. So, for example, we could convince ourselves that the following implementation of Euclid's greatest common divisor algorithm[2] is correct.

```
int
gcd(int x, int y)
{
   while (x != y) {
      gcdStep(&x, &y);
   }
   return(x);
}
```

One problem with this approach is that it may sometimes be difficult to state exactly when aliasing will occur. In the above example I have made the assumption (valid in C) that two non-equal pointers do not reference overlapping integer variables. For more complex data

1. The dereferencing operator "*" will be assumed to have a higher precedence than the " ' ", so that *x' is the value in the stopping state of the variable pointed to by x.

types pointer arithmetic, and a great deal of care, must be used. Some examples of how this is done are the predicates *noAliasing* and *overlapList* in the displays in section 4.0.

Since C allows a procedure to pass a value back to the caller using the `return` statement (which assigns a value to the accumulator register), this variable must also be treated as part of the data structure of the procedure. In this work, the name `value` is used to represent the value returned by a procedure.

3.0 Shortest Path Problem Description

The problem is to find the route between two nodes in a directed weighted graph such that it has the lowest total weight of all routes between those two nodes. The algorithm is to construct a spanning tree as follows: Add the starting node to the tree. Select the node, v, adjacent to the tree that has the lowest total weight of a path from the starting node to v and add it to the tree. Repeat this step until the destination node if found to be the nearest node adjacent to the tree.

The graph is represented by an array, `graph`, of linked lists, one for each node, of `EdgeNode` structures. Each list contains the weights of the edges leading from that node. The `path` array stores information about the candidate paths through the graph: `dist` is the weight of the shortest route found to each node, `status` indicates whether each node is in the tree, a fringe node, or has not been seen yet, `fringeLink` is the next node in the list of current fringe nodes and `parent` is the preceding node in the chosen path. `fringeList` is the first fringe node in the list.

4.0 Formal Description

4.1 Data Structure

```
typedef int VertexType;       /* Can't enforce range checking in C */

typedef struct EdgeNode * NodePointer;
typedef int BOOL;
typedef enum STATUS { INTREE, FRINGE, UNSEEN } StatusType;

struct EdgeNode {
   VertexType vertex;         /* Should be between 1...MAXVERTICES */
   int weight;
   NodePointer link;
};

typedef struct {
   StatusType status;   /* The nodes current status. */
   int dist;            /* The shortest path weight to this node */
   VertexType fringeLink;  /* the next node in the fringeList */
   VertexType parent;      /* the previous node in the path */
} PathElem;
```

4.2 Displays

DISPLAY 1

Specification

static VertexType
shortestPath(NodePointer graph[MAXVERTICES],
VertexType v, VertexType w,
PathElem path[MAXVERTICES])

$R_{shortestPath}$ = NCP(graph) ∧ (*noAliasing*(graph, path) ∧
(∀i,(0 ≤ i < MAXVERTICES) ⇒ ('path[i].'status = UNSEEN))) ⇒

	existsPath(graph, v, w)	¬*existsPath*(graph, 'v, 'w)
path' |	*validPath*(path', graph, 'v, 'w) ∧ (path[w].dist' = *weight*(path', graph, 'v, 'w)) ∧ ¬(∃p,*validPath*(p, graph, 'v, 'w) ∧ *(weight*(p, graph, 'v, 'w) < path['w].dist')))	***true***
value |	value = 'w	¬(value = 'w)

Program

```
{
VertexType fringeList;
VertexType x;
BOOL stuck;

initFirst(&path[v]);
fringeList = MAXVERTICES;
x = v;
stuck = FALSE;

while (x != w && !stuck) {
   fringeList = addNeighbours(graph[x], x, fringeList, path);
   if (fringeList == MAXVERTICES) {
      stuck = TRUE;
   } else {
      x = chooseShortest(path, &fringeList);
   }
}
return(x);
}
```

Subprogram Specifications

static void
initFirst(PathElem *first)

$R_{initFirst}$ = (first->staus' = INTREE) $\wedge$ (first->dist' = 0) $\wedge$
(first->parent' = MAXVERTICES)

static VertexType
addNeighbours(NodePointer adjList,
VertexType me, VertexType fringe
PathElem path[MAXVERTICES])

$R_{addNeighbours}$ = NCP(path[me], adjList) $\wedge$
$\neg$*overlapList*(adjList, path, MAXVERTICES*sizeof(PathElem)) $\Rightarrow$
(($\forall$i,(0 $\leq$ i < MAXVERTICES) $\Rightarrow$ (
(path[i].status' = INFRINGE $\Rightarrow$ *inFringe*(path', value, i)) $\wedge$
(*inFringe*('path, fringe, i) $\Rightarrow$ *inFringe*(path', value, i)) $\wedge$

	adjacent(adjList, i) $\wedge$ [(path[i].'status = UNSEEN) $\vee$ ((path[i].'status = INFRINGE) $\wedge$ (path[i].'dist > (*linkWeight*(adjList, i) + path[me].'dist)))]	$\neg$*adjacent*(adjList, i) $\vee$ (path[i].'status = INTREE)
path[i].dist' =	*linkWeight*(adjList, i) + path[me].'dist	path[i].'dist
path[i].parent' =	me	path[i].'parent
path[i].status' =	INFRINGE	path[i].'status

)))

static VertexType
chooseShortest(PathElem path[MAXVERTICES], VertexType *start)

$R_{chooseShortest}$ = ($\forall$i,(0 $\leq$ i< MAXVERTICES) $\Rightarrow$ NC(path[i].dist)) $\wedge$
($\neg$('*start = 0) $\wedge$
$\neg$(path-sizeof(VertexType) < start < path+MAXVERTICES*sizeof(PathElem))) $\Rightarrow$

	true
path', *start' |	$\neg$*inFringe*(path', *start', value) $\wedge$ ($\forall$i, *inFringe*(path', *start', i) $\Rightarrow$ *inFringe*('path, '*start, i)) $\wedge$ ($\forall$i, (*inFringe*('path, '*start, i) $\wedge$ (i $\neq$ value)) $\Rightarrow$ *inFringe*(path', '*start', i))
path[value].status' =	INTREE
value |	*inFringe*('path, '*start, value) $\wedge$ ($\forall$i, *inFringe*('path, '*start, i) $\Rightarrow$ (path[i].'dist $\geq$ path[value].'dist))

END OF DISPLAY 1

DISPLAY 2

Specification

static void initFirst(PathElem *first)
$R_{initFirst}$ = (first->staus' = INTREE) $\wedge$ (first->dist' = 0) $\wedge$ (first->parent' = MAXVERTICES)

Program

```
{
first->status = INTREE;
first->dist = 0;
first->parent = MAXVERTICES;
}
```

Subprogram Specifications

END OF DISPLAY 2

DISPLAY 3

Specification

static VertexType
addNeighbours(NodePointer adjList,
VertexType me, VertexType fringe
PathElem path[MAXVERTICES])

$R_{addNeighbours}$ = NCP(path[me], adjList) ∧
¬*overlapList*(adjList, path, MAXVERTICES*sizeof(PathElem)) ⇒
((∀i,(0 ≤ i < MAXVERTICES) ⇒ (
(path[i].status' = INFRINGE ⇒ *inFringe*(path', value, i)) ∧
(*inFringe*('path, fringe, i) ⇒ *inFringe*(path', value, i)) ∧

	adjacent(adjList, i) ∧ [(path[i].'status = UNSEEN) ∨ ((path[i].'status = INFRINGE) ∧ (path[i].'dist > (*linkWeight*(adjList, i) + path[me].'dist)))]	¬*adjacent*(adjList, i) ∨ (path[i].'status = INTREE)
path[i].dist' =	*linkWeight*(adjList, i) + path[me].'dist	path[i].'dist
path[i].parent' =	me	path[i].'parent
path[i].status' =	INFRINGE	path[i].'status

)))

Program

```
{
NodePointer ptr;

ptr = adjList;
while (ptr != NULL) { /* Traverse all edges leading from me. */
   if (newFringe(&path[ptr->vertex], ptr->weight, &path[me], me)) {
      path[ptr->vertex].fringeLink = fringe;
      fringe = ptr->vertex;
   }
   ptr = ptr->link;
}
return(fringe);
}
```

Subprogram Specifications

static BOOL
newFringe(PathElem *new, int weight, PathElem *me, VertexType my_num)

$R_{newFringe}$ = NCP(me) $\wedge$ $\neg$(new-sizeof(PathElem) < me < new+sizeof(PathElem)) $\Rightarrow$

	(new->'status = UNSEEN) $\vee$ ((new->'status = INFRINGE) $\wedge$ (new->dist > (me->'dist + 'weight)))	(new->'status = INTREE) $\vee$ ((new->'status = INFRINGE) $\wedge$ (new->dist $\leq$ (me->'dist + 'weight)))
new->parent' =	'my_num	new->'parent
new->status' =	INFRINGE	new->'status
new->dist' =	me->'dist + 'weight	new->'dist
value =	TRUE	FALSE

END OF DISPLAY 3

DISPLAY 4

Specification

static BOOL
newFringe(PathElem *new, int weight, PathElem *me, VertexType my_num)

$R_{newFringe}$ = NCP(me) $\wedge$ $\neg$(new-sizeof(PathElem) < me < new+sizeof(PathElem)) $\Rightarrow$

	(new->'status = UNSEEN) $\vee$ ((new->'status = INFRINGE) $\wedge$ (new->dist > (me->'dist + 'weight)))	(new->'status = INTREE) $\vee$ ((new->'status = INFRINGE) $\wedge$ (new->dist $\leq$ (me->'dist + 'weight)))
new->parent' =	'my_num	new->'parent
new->status' =	INFRINGE	new->'status
new->dist' =	me->'dist + 'weight	new->'dist
value =	TRUE	FALSE

Program

```
{
BOOL fringe = FALSE;

if (new->status == FRINGE && (me->dist + weight < new->dist)) {
   /* This is a new better route to new */
   new->parent = my_num;
   new->dist = me->dist + weight;
}
if (new->status == UNSEEN) {
   /* y is now on the fringe */
   new->status = FRINGE;
   new->parent = my_num;
   new->dist = me->dist + weight;
   fringe = TRUE;
}
return(fringe);
}
```

Subprogram Specifications

END OF DISPLAY 4

DISPLAY 5

Specification

static VertexType
chooseShortest(PathElem path[MAXVERTICES], VertexType *start)

$R_{chooseShortest}$ = ($\forall$i,(0 $\leq$ i< MAXVERTICES) $\Rightarrow$ NC(path[i].dist)) $\wedge$
($\neg$('*start = 0) $\wedge$
$\neg$(path-sizeof(VertexType) < start < path+MAXVERTICES*sizeof(PathElem))) $\Rightarrow$

	true
path', *start' |	$\neg$*inFringe*(path', *start', value) $\wedge$ ($\forall$i, *inFringe*(path', *start', i) $\Rightarrow$ *inFringe*('path, '*start, i)) $\wedge$ ($\forall$i, (*inFringe*('path, '*start, i) $\wedge$ (i $\neq$ value)) $\Rightarrow$ *inFringe*(path', '*start', i))
path[value].status' =	INTREE
value |	*inFringe*('path, '*start, value) $\wedge$ ($\forall$i, *inFringe*('path, '*start, i) $\Rightarrow$ (path[i].'dist $\geq$ path[value].'dist))

Program

```
{
VertexType min, m_prev = 0;
VertexType cur, prev;
min = *start;
prev = *start;
cur = path[*start].fringeLink;
while (cur != MAXVERTICES) {
   if (path[min].dist > path[cur].dist) {
      m_prev = prev;
      min = cur;
   }
   prev = cur;
   cur = path[cur].fringeLink;
}
if (min == *start) {
   *start = path[min].fringeLink;
} else {
   path[m_prev].fringeLink = path[min].fringeLink;
}
path[min].fringeLink = MAXVERTICES;
path[min].status = INTREE;
return(min);
}
```

Subprogram Specifications

END OF DISPLAY 5

LEXICON

A. Auxiliary Predicates

adjacent: NodePointer × VertexType

adjacent(list, v) $\stackrel{df}{=}$ (list->vertex = v) ∨ (¬(list->link = NULL) ∧ *adjacent*(list->link, v))

existsPath: array of NodePointer × VertexType × VertexType

existsPath(graph, start, end) $\stackrel{df}{=}$ (start = end) ∨
(∃v, 0 ≤ v < MAXVERTICES ∧ *adjacent*(graph[start], v) ∧ *existsPath*(graph, v, end))

inFringe: array of PathElem × VertexType × VertexType

inFringe(path, start, elem) $\stackrel{df}{=}$ (start = elem) ∨
(¬(path[start].fringeLink = 0) ∧ *inFringe*(path, path[start].fringeLink, elem))

noAliasing: array of NodePointer × array of PathElem

noAliasing(graph, path) $\stackrel{df}{=}$ (¬(graph-MAXVERTICES*sizeof(PathElem) < path < graph+MAXVERTICES*sizeof(NodePointer)) ∧
(∀i, (0 ≤ i < MAXVERTICES) ⇒
¬*overlapList*(graph[i], path, MAXVERTICES*sizeof(PathElem))

overlapList: NodePointer × void * × integer

overlapList(node, x, size) $\stackrel{df}{=}$

node = NULL	¬(node = NULL)
false	((node-size) < x < (node+ MAXVERTICES*sizeof(struct EdgeNode)) ∨ *overlapList*(node->link, x, size)

validPath: array of PathElem × array of NodePointer × VertexType × VertexType

validPath(path, graph, start, end) $\stackrel{df}{=}$ *adjacent*(graph[path[end].parent], end) ∧
((path[end].parent = start) ∨ *validPath*(path, graph, start, path[end].parent))

B. Auxiliary Functions

linkWeight: NodePointer × VertexType → integer

linkWeight(list, end) $\stackrel{df}{=}$

	list->vertex = end	¬(list->vertex = end)
value =	list->weight	*linkWeight*(list->link, end)

weight: array of PathElem × array of NodePointer × VertexType × VertexType → integer

weight(path, graph, start, end) $\stackrel{df}{=}$

	path[end].parent = start ∧ *adjacent*(graph[start], end)	¬(path[end].parent = start) ∧ *adjacent*(graph[path[end].parent], end)
value =	*linkWeight*(graph[start], end)	*linkWeight*(graph[path[end].parent], end) + *weight*(path, graph, start, path[end].parent)

Acknowledgements

This work was supported in part by the Government of Ontario through the Telecommunications Research Institute of Ontario (TRIO) and the Government of Canada through the Natural Sciences and Engineering Research Council (NSERC).

References

1. S. Baase, *Computer Algorithms Introduction to Design and Analysis*, second edition, Addison-Wesley Publishing Company, 1988.
2. E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.
3. M. Iglewski, J. Madey, D. L. Parnas and P. C. Kelly, “Documentation Paradigms”, *CRL Report No. 270*, Telecommunications Research Institute of Ontario (TRIO), July 1993, 45 pgs.
4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, second edition, Prentice Hall, 1988.
5. D. L. Parnas, “Predicate Logic for Software Engineering”, *IEEE Transactions on Software Engineering*, Vol. 19, No. 9, (September 1993), pp. 856-862.
6. D. L. Parnas, J.Madey and M. Iglewski, “Precise Documentation of Well-Structured Programs”, *IEEE Transactions on Software Engineering*, Vol. 20, No. 12, (December 1994), pp. 948 - 976.