Science and Engineering Research Council
**Rutherford Appleton Laboratory**
CHILTON, DIDCOT, OXON, OX11 0QX

RAL-84-068

# Formal Specification and Graphics Software

D A Duce E V C Fielding L S Marshall

August 1984

# Formal Specification and Graphics Software

*D A Duce, E V C Fielding*

Rutherford Appleton Laboratory, Chilton, Didcot, Oxon OX11 0QX

*L S Marshall*

Department of Computer Science, University of Manchester, Manchester M13 9PL

## 1. Introduction

This paper examines the application of one particular technique for formal specification, the Vienna Development Method [1,2], to a small part of GKS [3,4], the description of implicit regeneration. From the specification, properties of implicit regeneration are formulated and proved, showing that the behaviour conforms to that expected intuitively.

## 2. Formal Specification

The purpose of a specification is to state what a system is to do, not how it is to work. One starting point is to consider a system as a "black box" - whose internal workings cannot be observed and to describe the system in terms of the relationship between input and output. There are, however, limitations to this approach - often the output depends not only on the last input received, but also on the internal state of the system when that input arrives. It is not always realistic to specify the output sequence in terms of all possible input sequences. The solution is to introduce the notion of internal state, but in a carefully controlled way so that only the essential characteristics of the state are modelled. This is done through abstract data types, a data type characterized only by the operations allowed over it.

The axiomatic or algebraic methods are one approach to the definition of such data types [5,6]. The aim of an algebraic specification is to define abstract data types without saying anything about representations of the associated classes of objects. This is done by giving a set of axioms which the operations over the data type must satisfy. Such specifications are termed property-oriented specifications. A good introduction to this method of specifying abstract data types appears in [7]. To give the flavour of this approach, we consider the specification of a LIFO stack.

type *Stack*

operators

*Newstack* : → *Stack*

*Push* : *Stack* × *Element* → *Stack*

*Inspect* : *Stack* → *Element*

*Remove* : *Stack* → *Stack*

*Is_Empty* : *Stack* → *Boolean*

axioms

*Remove* (*Push* (*st* , *el*)) = *st*

*Inspect* (*Push* (*st* , *el*)) = *el*

*Is_Empty* (*Push* (*st* , *el*)) = *false*

. . .

end type

The operations for manipulating a stack are: *Newstack* which produces an instance of the empty stack, *Push* which adds an element to a stack and returns a new stack, *Inspect* which returns the top element of a stack (without altering the stack), *Remove* which discards the top element of a stack and returns the resulting stack, and *Is_Empty* which tests if a stack is empty. The types of these operations are shown in the section of the specification headed operators. Note that the possibility of errors is ignored completely.

Axioms such as those under the heading axioms can completely characterize *Stack* without committing to a model. Note that the axiom set above is not complete.

An alternative approach is to provide a model for the abstract data type in terms of mathematically tractable entities, for example, sets, lists and mappings. This is termed the constructive or model-based approach. This paper describes and applies the constructive method known as the Vienna Development Method (VDM). The method and applications are described in detail in [1, 2, 8].

Continuing with the LIFO stack example, a stack could be modelled as a list of elements. In VDM this is written as:

*Stack* = list of *Element*

which defines the data type *Stack* to be a list of entities of type *Element* .

Operations then have the general form:

*Inputs* × *State* → *Outputs* × *State'*

i.e. operations can both modify the state and produce outputs. The effect of operations is described in terms of their inputs, outputs and effect on the underlying state. VDM conveniently describes these effects by two predicates, a pre-condition and a post-condition. The former is a predicate over *Inputs* and *State* which defines the conditions under which the operation will produce a valid result. In other words, the pre-condition restricts the domain of the operation, thus making it a partial function. The post-condition is a predicate over *Inputs*, *Outputs*, *State* (the initial state) and *State'* (the final state)

defining the effect of the operation. Now consider the specification of the *Push* operation for *Stack* in this approach.

*Push* : *element* : *Element*

**ext wr** *stack* : *Stack*

**post** *stack'* = *element* :: *stack*

The operation takes a single argument, *element*, of type *Element*. The **ext** statement is used to indicate what components of the state are accessed and perhaps altered by the operation. **rd** indicates read-only access and **wr** write access. Here the state has only one component, a variable named *stack* of type *Stack*, which is modified by the *Push* operation. **post** is the post-condition for the operation which says that the effect of the operation is to add *element* to the head of *stack*. (In this context '::' is used in infix form to denote the operator *cons*.)

The definitions of the other operations will not be described here, as they follow fairly readily. A more detailed explanation of the notation used in VDM is given in section 4.2.

The specification of the data type provides a way of determining what the effects of its operations will be, by showing how to derive them. It is likely that a realization built in the same way would be inefficient in that the operations would take a long time to compute, except on hardware directly supporting the data types (e.g. lists, sets and maps) used in the specification.

Parallels exist between axiomatic and constructive approaches in mathematics. A good example is group theory. Abstract groups can be elegantly described by a small number of axioms. For some parts of group theory, the most convenient exposition is in terms of the axiomatic description. An alternative way to discuss the structure of an abstract group is in terms of its representation by, say, matrices. For some areas of the theory, representations rather than axioms are the most convenient exposition. The same is true in the definition of abstract data types - some are more conveniently discussed in an algebraic style, others in a constructive style. Each has its own insights to offer and limitations to bear.

This paper uses the constructive approach. Previous work on the specification of graphics software seems to have been based on the algebraic approach [9,5]. The use of the constructive approach is explored here and it is felt that it has produced a clear, readable specification of the problem which serves well as a basis from which to reason about the behaviour of the system. It is hoped that good insights into the structure of the problem have been offered and a description produced which is intuitively appealing to graphics workers.

## 3. The Problem

The specification of a simplified version of implicit regeneration in GKS is now considered.

The appearance of output primitives in GKS is determined by their aspects. For the remainder of this paper we will restrict ourselves to one output primitive, polyline. The aspects of a polyline are:

linetype
linewidth scale factor
polyline colour index

The values of these aspects are determined by a single attribute, the polyline index. Polyline index is in fact an index into a table, the polyline bundle table, each entry in which specifies values for each of the aspects.

The physical output device on which the pictures will be displayed is presented to the applications programmer through a workstation. A workstation represents zero or one display surfaces and zero or more input devices. The point of interest here is that each workstation has its own polyline bundle table. Thus the appearance of a given polyline can be different on different workstations, in that the values of the aspects may be set differently.

GKS provides the function:

SET POLYLINE REPRESENTATION( WKID, PLI, LTYPE, LWIDTH, COLI )

where the arguments are:

| | |
|---|---|
| WKID | workstation identifier |
| PLI | polyline index |
| LTYPE | linetype |
| LWIDTH | linewidth scale factor |
| COLI | colour index |

Consider the program fragment:

```
SET POLYLINE REPRESENTATION( 1, 1, 1, 1.0, 1 )
SET POLYLINE INDEX( 1 )
POLYLINE( N, X, Y )
```

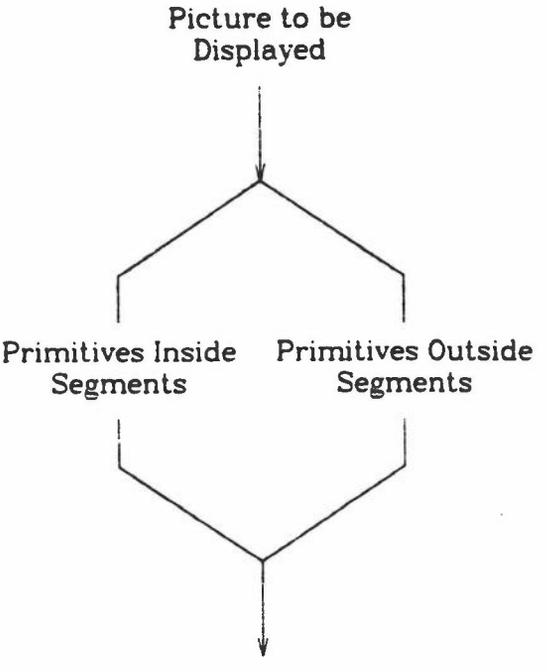
The polyline defined by the N data points in the arrays X, Y will be drawn with polyline index 1, which corresponds to linetype 1, linewidth scale factor 1.0 and colour index 1 on workstation 1.

What happens if the representation of polyline index 1 on workstation 1 is subsequently changed?

Before this question is answered, two digressions are made, the first to discuss picture structuring in GKS and the second to discuss the classes of device hardware that GKS has to address.

### 3.1. Picture Structure

Output primitives may be grouped together in segments. Pictures may thus consist of primitives and segments. This leads to the situation depicted below.



There are cases in GKS where primitives inside segments are treated differently from primitives outside segments; an example of this follows shortly.

Some mention needs to be made here of coordinate systems. Three coordinate systems are discernible in GKS:

world coordinates
normalized device coordinates
device coordinates

World coordinates are the cartesian coordinate system in which the coordinates of output primitives are specified. These are transformed to an intermediate cartesian coordinate system called normalized device coordinates by a normalization transformation, which is a standard window-to-viewport transformation. Normalized device coordinates are restricted to the unit square, $[0, 1] \times [0, 1]$. The mapping from normalized device coordinates to the coordinate system of the output device (device coordinates) is controlled by a second window-to-viewport transformation, the workstation transformation.

Each workstation has a segment store in which primitives in segments are held. The coordinates of these primitives are stored in normalized device coordinates.

### 3.2. Device Hardware

For the present purposes, two classes of graphics device are distinguished:

1. those which allow the representation of a component of a picture to be changed dynamically;
2. those which do not.

Consider changing the colour of a polyline. If the output device is, say a colour raster scan display, the colour of the polyline may be changed dynamically by changing the colour of the pixels that comprise the polyline. If the output device is, say, a pen plotter, the colour can only be 'changed' by advancing the frame and redrawing the polyline using a different pen. The idea should be clear, even if the example is slightly contrived.

### 3.3. Implicit Regeneration

We return to the question posed at the beginning of section 3. If a polyline representation is changed, what happens to the appearance of polylines already drawn with the old representation? The GKS model stipulates that their appearance should change.

For devices of class (1) above, this poses no problems since appearance may be changed dynamically. For devices of class (2) appearance can only be altered by redrawing the complete picture from a stored representation. The only stored representation in GKS, is the representation of the picture in segment storage. Since, however, not all of the picture may be stored in segment storage (primitives outside segments are not so stored), parts of the picture may be lost when this is done. When a change to the picture is requested, the workstation checks if the change can be made dynamically, if so, the change is made. If not, an implicit regeneration is signalled. GKS provides control over when the regeneration will occur, through the setting of implicit regeneration mode. The possible values are:

| | | |
|---|---|---|
| ALLOWED | - | regeneration will be performed immediately |
| SUPPRESSED | - | regeneration will be postponed until one of the functions REDRAW ALL SEGMENTS ON WORKSTATION, UPDATE WORKSTATION or CLOSE WORKSTATION is invoked. |

For each possible change that can potentially require a regeneration each workstation has a flag to indicate whether the change may be performed immediately (IMM) or requires a regeneration (IRG).

The next section addresses the problem of how implicit regeneration can be specified.

## 4. Overview of the Specification of a Simplified Model of Regeneration

### 4.1. Simplifications

For illustrative purposes the following simplifications are made:

1. The only output primitive is polyline.
2. Only a single workstation is considered.
3. Only changes in polyline representation are considered. Thus for example, the workstation transformation is regarded as fixed.
4. The polyline bundle is restricted to linetype and linewidth scale factor. Colour is ignored.
5. Segment attributes are not considered.
6. The functions UPDATE WORKSTATION and CLOSE WORKSTATION are not considered.
7. Buffering of picture changes and deferral mode are ignored.
8. Initialization of the state is not described, as this is straightforward.
9. World coordinates, individual attributes and clipping are ignored.

For this example only REDRAW ALL SEGMENTS ON WORKSTATION need be considered.

### 4.2. VDM

Before the specification is described in detail, the notation used in the following sections is summarized.

A VDM specification has three components:

(a) a model of the state;

(b) invariants on the state;

(c) operations over the state.

#### 4.2.1. Notation used in describing the state

A particular state is modelled by a structured object, whose components may be basic objects (integers, reals etc.), or other structured objects ultimately built using objects of these types. Sets, lists and mappings may also be used in the construction process. A mapping is similar to a function except that it has a (possibly sparse) finite domain and the pairing of domain and range elements is constructed rather than being defined by an expression. Sets and lists should not require further explanation. For further details, see [1].

The first section of a VDM specification describes the structure of the class of objects representing the state. If *X* is some class of objects, objects belonging to this class are said to have type *X*.

In the simplest case, a state has only one component. VDM then uses the equality sign '='. For example:

*NDC_Picture* = list of *Component*

defines the type *NDC_Picture* to be a list, each of whose elements is an object of type *Component*. The definition of the type *Component* might be:

*Component* = *Segment* | *NDC_Polyline*

meaning that *Component* is either ('|' denotes alternation) of type *Segment* or *NDC_Polyline*.

More often a state is a composite object, consisting of the Cartesian product of several components. VDM uses the sign '::' to define a composite object. For example:

*GKS* :: *ndc_picture* : *NDC_Picture*

*dc_picture* : *DC_Picture*

*polyline_bundle_table* : *Polyline_Bundle_Table*

defines an object of type *GKS* to have three components, named *ndc_picture* (of type *NDC_Picture*), *dc_picture* (of type *DC_Picture*) and *polyline_bundle_table* (of type *Polyline_Bundle_Table*). As can be seen, component definitions take the form:

*component_name* : *Component_Type*

The convention adopted is that type names have initial upper case letters and the names of components of composite objects and of specific instances of objects are the lower case equivalents of their type names.

An example of how to construct composite objects and access their components is given later.

The type *Polyline_Bundle_Table* is defined as:

*Polyline_Bundle_Table* = **map** *Polyline_Index* **to** *Bundle*

which introduces the VDM notation for a mapping.

### 4.2.2. Notation used in describing the invariants

Data type invariants are constraints on the components of the state which must be preserved by the operations. They restrict the class of states to a valid subset.

The invariants which are necessary over the state *GKS* are listed in section 5.2. The notation used in expressing these predicates will not be discussed as it is self-explanatory.

### 4.2.3. Notation used in describing the operations

To illustrate the notation for defining operations consider the definition of *add_polyline*. This is:

*add_polyline* : (*ndc_points* : *NDC_Points*) × (*polyline_index* : *Polyline_Index*)

**ext rd** *polyline_bundle_table* : *Polyline_Bundle_Table*

**wr** *ndc_picture* : *NDC_Picture*

wr *dc_picture* : *DC_Picture*

pre *polyline_index* ∈ dom *polyline_bundle_table*

post *ndc_picture'* = < *ndc_points* , *polyline_index* > :: *ndc_picture* ∧

let *e* = < *transform* (*ndc_points* ), *polyline_index* ,

*polyline_bundle_table* (*polyline_index* ) >

in *dc_picture'* = *e* :: *dc_picture*

The first line of the definition states that the arguments of *add_polyline* are of types *NDC_Points* and *Polyline_Index*. The next three lines define the components of the state used by the operation. The state here is actually the state *GKS* introduced in section 4.2.1; when referring to a component of *GKS* the notation that should strictly be used is:

*GKS.ndc_picture*

However, this becomes cumbersome and so the first component of the name for the top level components is dropped and this is written as:

*ndc_picture*

Only the components of the state (and their types) actually used in the definition of the operation need to be introduced in the ext statement. rd indicates read only access and wr indicates write access. Thereafter, *component_name* may be used to refer to that component of the state.

The pre-condition says that the polyline index supplied as argument must be valid, i.e. within the domain of the polyline bundle table mapping. The operator dom applied to a mapping or function yields its domain.

The post-condition says that the effect of the operation is to add the list of points and polyline index to the *ndc_picture*, and the representation of the polyline (linewidth, linetype and list of points) to the *dc_picture*. '::' is used in infix form to denote the operator *cons*. The infix operator '||' denotes *append*.

*polyline_bundle_table* (*polyline_index* ) illustrates the application of the mapping of type *Polyline_Bundle_Table* to an object from its domain, of type *Polyline_Index*, which produces a result of type *Bundle*. This follows function application.

An example of the notation for constructing a composite object is:

<*ndc_points* , *polyline_index* >

The constructed object is of type *NDC_Polyline* having two components: *ndc_points* of type *NDC_Points*, and *polyline_index*, of type *Polyline_Index*.

An example of the notation for accessing the components of a composite object is the let clause:

let <*ndc_points* , *polyline_index* > = *ndc_polyline*

. . .

Here the names *ndc_points* and *polyline_index* are declared to refer to the two components of the object *ndc_polyline*.

Finally, an example of the construction of a mapping is shown. Mappings are constructed using the operator '+'. Thus:

*polyline_bundle_table* = *polyline_bundle_table* + [*polyline_index* → *bundle* ]

adds [*polyline_index* → *bundle* ] to the mapping, overriding any previous value

associated with *polyline_index*.

### 4.3. Description of the State

The operations that are defined later, rely on the concepts of:

1. the picture in NDC space;
2. the segment store;
3. the picture in DC space;
4. the polyline bundle table.

Following the description of picture structure in section 3.1 the following types are defined:

*NDC_Picture* = list of *Component*

*Component* = *Segment* | *NDC_Polyline*

*Segment* = list of *NDC_Polyline*

A polyline in NDC-space is represented as a list of components and a polyline index (a natural number). This is written:

*NDC_Polyline* :: *ndc_points* : *NDC_Points*
*polyline_index* : *Polyline_Index*

*NDC_Points* = list of *NDC_Point*

*NDC_Point* :: *x* : R
*y* : R

*Polyline_Index* : N

The segment store is represented as:

*Segment_Store* = list of *Segment*

At the DC picture level, the appearance of a polyline is bound to the polyline. Thus at the DC level a polyline is described by a list of points, a linetype and linewidth. DC pictures are represented as:

*DC_Picture* = list of *DC_Polyline*

*DC_Polyline* :: *dc_points* : *DC_Points*
*polyline_index* : *Polyline_Index*
*bundle* : *Bundle*

*DC_Points* = list of *DC_Point*

*DC_Point* :: *x* : R
*y* : R

*Bundle* :: *linetype* : *Linetype*
*linewidth* : *Linewidth*

*Linetype* : N

*Linewidth* : R

This DC picture description has been designed so that it is possible to compare the equality of pictures at the DC level. It is a representation of the essential

features of the displayed picture (list of points, linetype, linewidth). Two pictures are said to be equal if they contain exactly the same polylines (in turn implying that corresponding polylines have equal lists of points, linetypes and linewidths).

Note that this design is not concerned with modelling particular physical display devices; some devices for example would actually represent the segment structure at the device level (e.g. a refresh display driven from a display file), others would have no stored representation whatsoever, other than the picture on the physical display surface (e.g. a pen plotter). For our present purposes it is not necessary to discuss the segment structure of a picture below NDC level. For other purposes (e.g. discussion of input), it will probably be necessary to model device level segment storage. However, the analysis presented here will not be affected, since a segmented DC picture model can be mapped down to the non-segmented model used here.

The polyline bundle table is modelled as a mapping from a polyline index to a bundle:

*Polyline_Bundle_Table* = map *Polyline_Index* to *Bundle*

Finally, two flags are required,

*Implicit_Regeneration* = {*ALLOWED*, *SUPPRESSED*}

which specifies the setting of implicit regeneration mode, and

*Bundle_Modification_Flag* = {*IMM*, *IRG*}

which specifies whether or not dynamic modification is possible for a bundle representation change.

The complete state is listed in section 5.1.

### 4.4. Operations

The following operations over this state are considered:

*add_polyline*
*add_segment*
*redraw_all_segments*
*set_polyline_representation*

The first two operations do not correspond directly to any one of the GKS functions. They are, rather, abstractions of GKS functions which help to clarify our presentation. The operation *add_polyline* is roughly equivalent to:

```
SET POLYLINE INDEX( ... )
POLYLINE( ... )
```

and *add_segment* to:

```
CREATE SEGMENT( ... )
SET POLYLINE INDEX( ... )
POLYLINE( ... )
. . .
SET POLYLINE INDEX( ... )
POLYLINE( ... )
. . .
CLOSE SEGMENT( ... )
```

Full specifications of the operations are given in section 5.3.

## 5. The Complete Specification

### 5.1. GKS State

The complete state is:

*GKS* :: *ndc_picture* : *NDC_Picture*
*dc_picture* : *DC_Picture*
*segment_store* : *Segment_Store*
*polyline_bundle_table* : *Polyline_Bundle_Table*
*bundle_modification_flag* : *Bundle_Modification_Flag*
*implicit_regeneration* : *Implicit_Regeneration*

*NDC_Picture* = **list of** *Component*
*Component* = *Segment* | *NDC_Polyline*
*Segment* = **list of** *NDC_Polyline*
*NDC_Polyline* :: *ndc_points* : *NDC_Points*
*polyline_index* : *Polyline_Index*
*NDC_Points* = **list of** *NDC_Point*
*NDC_Point* :: *x* : **R**
*y* : **R**
*Polyline_Index* : **N**

*DC_Picture* = **list of** *DC_Polyline*
*DC_Polyline* :: *dc_points* : *DC_Points*
*polyline_index* : *Polyline_Index*
*bundle* : *Bundle*
*DC_Points* = **list of** *DC_Point*
*DC_Point* :: *x* : **R**
*y* : **R**
*Bundle* :: *linetype* : *Linetype*
*linewidth* : *Linewidth*
*Linetype* : **N**
*Linewidth* : **R**

*Segment_Store* = list of *Segment*

*Polyline_Bundle_Table* = map *Polyline_Index* to *Bundle*

*Bundle_Modification_Flag* = {*IRG*, *IMM*}

*Implicit_Regeneration* = {*ALLOWED*, *SUPPRESSED*}

### 5.2. Invariants

**Auxiliary Definitions**

The functions defined below are used in the specification of the invariants. The symbol '$\triangleq$' denotes 'is defined to be'.

*flatten* : list of (*list* | *atom*) → list

*flatten*(*l*) $\triangleq$ if *l* = < > then < >

else if *atom*(hd *l*) then hd *l* :: *flatten*(tl *l*)

else hd *l* || *flatten*(tl *l*)

*is_sublist_of* : list × list → *Boolean*

$l_1$ *is_sublist_of* $l_2 \triangleq$ len $l_1 \leq$ len $l_2 \wedge$

elems $l_1 \subseteq$ elems $l_2 \wedge$

$(\forall i, j)(i, j \in \{1 \,..\, \text{len } l_1\} \wedge i \neq j)$

$((\exists m, n)(m, n \in \{1 \,..\, \text{len } l_2\} \wedge m \neq n)$

$(l_1(i) = l_2(m) \wedge l_1(j) = l_2(n) \wedge$

$i < j \Rightarrow m < n \wedge i > j \Rightarrow m > n ))$

*map_to_NDC* : *DC_Picture* → list of *NDC_Polyline*

*map_to_NDC*(*dc_picture*) $\triangleq$

if *dc_picture* = < > then < >

else let <*dc_points*, *polyline_index*, *bundle*> = hd *dc_picture*

and *e* = <*transform*$^{-1}$(*dc_points*), *polyline_index*>

in *e* :: *map_to_NDC*(tl *dc_picture*)

*transform* : *NDC_Points* → *DC_Points*

In addition invariant (2) uses the infix operator '⌈', which given a list and a set produces a list whose members are those elements of the original list (in the same order) which are contained in the set, i.e. it is a restriction operator.

⌈ : *List* × *Set* → *List*

*l* ⌈ *s* $\triangleq$ if *l* = <> then <>

else if hd *l* ∈ *s* then hd *l* :: ( tl *l* ⌈ *s* )

else tl *l* ⌈ *s*

**Invariant (1)**

All polyline indices used in the NDC picture must be within the domain of the polyline bundle table:

$$(\forall polyline\_index)(< , polyline\_index > \in \textbf{elems}\ flatten(ndc\_picture)) \quad (1)$$
$$(polyline\_index \in \textbf{dom}\ polyline\_bundle\_table)$$

**Invariant (2)**

All segments in the NDC picture must be stored in the segment store:

$$segment\_store = ndc\_picture \lceil \{c \mid c \in \textbf{elems}\ ndc\_picture \wedge Segment(c)\} \quad (2)$$

**Invariant (3)**

All point-list (after inverse transformation) and polyline index pairs which are in the *dc_picture* are also in the *ndc_picture*:

$$map\_to\_NDC(dc\_picture)\ is\_sublist\_of\ flatten(ndc\_picture) \quad (3)$$

**Invariant (4)**

All point-list (after transformation) and polyline index pairs in *segment_store* are also contained in the *dc_picture*:

$$flatten(segment\_store)\ is\_sublist\_of\ map\_to\_NDC(dc\_picture) \quad (4)$$

**Invariant (5)**

NDC coordinates are in the range $[0, 1] \times [0, 1]$:

$$\textbf{let}\ <x, y> = NDC\_Point\ \textbf{in}$$
$$0.0 \leq x \leq 1.0 \wedge 0.0 \leq y \leq 1.0 \quad (5)$$

### 5.3. Operations

$$add\_polyline : (ndc\_points : NDC\_Points) \times (polyline\_index : Polyline\_Index)$$
$$\textbf{ext rd}\ polyline\_bundle\_table : Polyline\_Bundle\_Table$$
$$\textbf{wr}\ ndc\_picture : NDC\_Picture$$
$$\textbf{wr}\ dc\_picture : DC\_Picture$$
$$\textbf{pre}\ polyline\_index \in \textbf{dom}\ polyline\_bundle\_table$$
$$\textbf{post}\ ndc\_picture' = < ndc\_points, polyline\_index > :: ndc\_picture \wedge$$
$$\textbf{let}\ e = < transform(ndc\_points), polyline\_index,$$
$$polyline\_bundle\_table(polyline\_index) >$$
$$\textbf{in}\ dc\_picture' = e :: dc\_picture$$

```
add_segment : segment : Segment
ext rd polyline_bundle_table : Polyline_Bundle_Table
     wr ndc_picture : NDC_Picture
     wr segment_store : Segment_Store
     wr dc_picture : DC_Picture
pre (∀polyline_index)(< , polyline_index > ∈ elems segment )
                        (polyline_index ∈ dom polyline_bundle_table )
post ndc_picture' = segment :: ndc_picture ∧
      segment_store' = segment :: segment_store ∧
      dc_picture' = applypbt (segment , polyline_bundle_table ) || dc_picture

where applypbt : Segment × Polyline_Bundle_Table → DC_Picture
       applypbt (segment , polyline_bundle_table ) ≜
                if segment = < > then < >
                else let <ndc_points , polyline_index > = hd segment
                     and e = <transform (ndc_points ), polyline_index ,
                                polyline_bundle_table (polyline_index )>
                     in e :: applypbt (tl segment , polyline_bundle_table )

redraw_all_segments
ext rd segment_store : Segment_Store
     rd polyline_bundle_table : Polyline_Bundle_Table
     wr dc_picture : DC_Picture
post dc_picture' = regenerate ( segment_store , polyline_bundle_table )

where regenerate : Segment_Store × Polyline_Bundle_Table → DC_Picture
       regenerate (segment_store , polyline_bundle_table ) ≜
                  if segment_store = < > then < >
                  else applypbt ( hd segment_store , polyline_bundle_table ) ||
                                regenerate (tl segment_store , polyline_bundle_table )
```

*set_polyline_representation* : (*polyline_index* : *Polyline_Index*) × (*linetype* : *Linetype*) × (*linewidth* : *Linewidth*)

ext rd *segment_store* : *Segment_Store*

rd *bundle_modification_flag* : *Bundle_Modification_Flag*

rd *implicit_regeneration* : *Implicit_Regeneration*

wr *dc_picture* : *DC_picture*

wr *polyline_bundle_table* : *Polyline_Bundle_Table*

post *polyline_bundle_table'* = *polyline_bundle_table* +

[*polyline_index* → <*linetype*, *linewidth*>] ∧

( *bundle_modification_flag* = *IMM* =>

*dc_picture'* = *recreate*(*dc_picture*, *polyline_bundle_table'*)) ∧

( *bundle_modification_flag* = *IRG* ∧ *implicit_regeneration* = *ALLOWED* =>

*dc_picture'* = *regenerate*(*segment_store*, *polyline_bundle_table'*)) ∧

( *bundle_modification_flag* = *IRG* ∧ *implicit_regeneration* = *SUPPRESSED* =>

*dc_picture'* = *dc_picture* )

where *recreate* : *DC_Picture* × *Polyline_Bundle_Table* → *DC_Picture*

*recreate*(*dc_picture*, *polyline_bundle_table*) ≜

if *dc_picture* = < > then < >

else let <*dc_points*, *polyline_index*, *bundle*> = hd *dc_picture*

and *e* = <*dc_points*, *polyline_index*,

*polyline_bundle_table*(*polyline_index*)>

in *e* :: *recreate*(tl *dc_picture*, *polyline_bundle_table*)

**6. Behaviour**

The behaviour of the system which has been specified is now examined and compared with our intuitive understanding.

Suppose there is one workstation which can support dynamic modification for changes to polyline bundle representations and a second which requires a picture regeneration. Suppose that for the second workstation, implicit regeneration mode is ALLOWED (i.e. regeneration takes place immediately). Then if the same picture is drawn on each workstation with the same polyline representations and the representation of the same polyline index is changed in each in the same way, we would expect to end up with the same picture on each, if the picture consisted only of primitives inside segments. If the picture contained primitives outside segments, we would not expect to get the same picture, because such primitives would be lost in the regeneration of the picture on the second workstation, but retained on the first.

If regeneration mode for the second workstation were SUPPRESSED, we would expect to achieve the effect described above after the function REDRAW ALL SEGMENTS has been invoked, (the effect of which is to perform the regeneration).

It will now be shown that the specification does indeed conform to this intuitive behaviour. Because the model does not incorporate multiple workstations, instead three GKS systems with appropriate settings of the dynamic modification and implicit regeneration mode flags are considered.

The theorem below is a statement of the intuitive behaviour described above. It is phrased in terms of relationships between states; in order to do this, the initial and final states of each operation are explicitly included as arguments of the operation.

**Abbreviations**

To aid readability the following abbreviations are used in the statement and proof of the theorem.

| | |
|---|---|
| *dcp* | *dc_picture* |
| *pbt* | *polyline_bundle_table* |
| *ss* | *segment_store* |
| *lt* | *linetype* |
| *lw* | *linewidth* |
| *pl* | *polyline* |
| *pi* | *polyline_index* |
| *s* | *segment* |

**Theorem:**

Consider states $gks_0$, $gks_1$ and $gks_2$, such that:

$$gks_0.bundle\_modification\_flag = IMM$$

$$gks_1.bundle\_modification\_flag = IRG \wedge gks_1.implicit\_regeneration = ALLOWED$$

$$gks_2.bundle\_modification\_flag = IRG \wedge gks_2.implicit\_regeneration = SUPPRESSED$$

but are otherwise identical. Then, (apart from the settings of *bundle_modification_flag* and *implicit_regeneration*, which will remain unchanged):

$$post\_set\_polyline\_representation(gks_0, pi, lt, lw, gks_0') \equiv$$
$$post\_set\_polyline\_representation(gks_1, pi, lt, lw, gks_1') \quad (6)$$

and

$$post\_set\_polyline\_representation(gks_0, pi, lt, lw, gks_0') \equiv$$
$$post\_redraw\_all\_segments(gks_2', gks_2'')$$
$$\text{where } post\_set\_polyline\_representation(gks_2, pi, lt, lw, gks_2') \quad (7)$$

iff $ndc\_picture = segment\_store$

**Proof:**

Consider the post-condition of *set_polyline_representation*. For state $gks_0$, where $bundle\_modification\_flag = IMM$, this takes the form:

$$pbt' = pbt + [pi \rightarrow <lt, lw>] \wedge dcp' = recreate(dcp, pbt')$$

For state $gks_1$, where

$$bundle\_modification\_flag = IRG \wedge implicit\_regeneration = ALLOWED$$

it becomes:

$$pbt' = pbt + [pi \rightarrow <lt, lw>] \wedge dcp' = regenerate(ss, pbt')$$

For the state $gks_2$, where

$$bundle\_modification\_flag = IRC \wedge implicit\_regeneration = SUPPRESSED$$

it has the form:

$$pbt' = pbt + [pi \rightarrow <lt, lw>] \wedge dcp' = dcp$$

hence, $post\_redraw\_all\_segments(gks_2', gks_2'')$ also has the form:

$$pbt' = pbt + [pi \rightarrow <lt, lw>] \wedge dcp' = regenerate(ss, pbt')$$

Thus the truth of equation (7) follows immediately from the truth of equation (6).

To show the equivalence of the final states and hence the truth of (6), it is sufficient to consider the equivalence of the DC picture components of the state. The effect on the polyline bundle table is the same in each case and both *ndc_picture* and *segment_store* are unaffected by the operation *set_polyline_representation*.

There are 2 cases to consider.

**Case 1**

$$ndc\_picture = segment\_store$$

The truth of (6) can be shown by structural induction on the data type list.

Basis:

$$ndc\_picture = segment\_store = < >$$

In this case $dcp = < >$ by invariant (3) and it follows from the definitions of *recreate* and *regenerate* that:
$recreate(< >, pbt') = regenerate(< >, pbt') = < >$

Assume:

$$recreate(dcp, pbt') = regenerate(ss, pbt') \qquad (8)$$

for the case where neither *ndc_picture* nor *segment_store* is empty.

To Prove:

As by invariant (2) and the definitions of the operations, *ndc_picture* and *segment_store* are created in this case by the same sequence of *add_segment* operations. So denoting the segment that is added by $s$, what must be shown is:

$$recreate(applypbt(s, pbt') \parallel dcp, pbt') = regenerate(s :: ss, pbt') \qquad (9)$$

The RHS of (9) expands to:

$$applypbt(\mathrm{hd}(s :: ss), pbt') \parallel regenerate(\mathrm{tl}(s :: ss), pbt')$$

$$= applypbt(s, pbt') \parallel regenerate(ss, pbt')$$

$$= applypbt(s, pbt') \parallel recreate(dcp, pbt') \quad by\ (8)$$

= LHS of (9)

The last step can be shown formally by proving the lemma:

$$recreate(applypbt(s, pbt'), pbt') = applypbt(s, pbt')$$

This is not presented here; the truth of the lemma can be seen informally from the function definitions since *recreate* applies the same polyline bundle table as *applypbt*.

Case 2

$$ndc\_picture \neq segment\_store$$

There are 3 sub-cases to consider:

Case 2a:

$$segment\_store = <> \wedge ndc\_picture \neq <>$$

In this case invariant (2) and the definitions of the operations show that *ndc_picture* is created by a sequence of *add_polyline* operations. Consider:

$$dcp' = recreate(dcp, pbt')$$

It follows from the definition of *add_polyline* that $dcp \neq <>$ and hence, from the definition of *recreate* that $dcp' \neq <>$. Now consider:

$$dcp' = regenerate(ss, pbt')$$

Since $ss = <>$ it follows that $dcp' = <>$.

Case 2b:

$$segment\_store \neq <> \wedge ndc\_picture \neq <>$$

In this case we know from the invariants and the definitions of the operations that *ndc_picture* is created by a sequence of both *add_segment* and *add_polyline* operations. Suppose that *ndc_picture* = *segment_store*. They both start empty and suppose that a sequence of *add_segment* operations only has been performed. From case 1 we know:

$$recreate(dcp, pbt') = regenerate(ss, pbt') \qquad (10)$$

Now suppose that an *add_polyline* operation is performed. If the polyline added to *dcp* is denoted by $<pl, pi, b>$, after this operation, we have:

$$dcp' = <pl, pi, b> :: dcp \wedge ss' = ss$$

Adding this polyline has added another point-list, polyline-index pair to *ndc_picture* and *dcp*, but has not altered *segment_store*.

Then:

$$recreate(<pl, pi, b> :: dcp, pbt')$$

$$= <pl, pi, pbt'(pi)> :: recreate(dcp, pbt')$$

By (10)

$$= <pl, pi, pbt'(pi)> :: regenerate(ss, pbt')$$

$$\neq regenerate(ss', pbt')$$

Case 2c:

$$segment\_store \neq < > \wedge ndc\_picture = < >$$

This case is ruled out by invariants (4) and (3).

**Acknowledgements**

We wish to thank Bob Hopgood, Dale Sutcliffe and Julian Gallop for their many useful comments on an earlier draft of this paper, and Alan Kinroy for his assistance with proof-reading.

**References**

1. C. B. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall, Englewood Cliffs, NJ (1980).
2. C. B. Jones, *Systematic Program Development*, Department of Computer Science, University of Manchester (1984).
3. F. R. A. Hopgood, D. A. Duce, J. R. Gallop, and D. C. Sutcliffe, *Introduction to the Graphical Kernel System (GKS)*, Academic Press (1983).
4. *Graphical Kernel System (GKS) 7.2 Functional Description, ISO/DIS 7942*, Information Processing (4 November 1982).
5. R. Gnatz, "An Algebraic Approach to the Standardization and the Certification of Graphics Software," *Computer Graphics Forum* 2(2/3) (1983).
6. J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," in *Current Trends in Programming Methodology Volume IV*, ed. R. T. Yeh, Prentice-Hall (1978).
7. J. V. Guttag, E. Horowitz, and D. R. Musser, "The Design of Data Type Specifications," in *Current Trends in Programming Methodology Volume IV*, ed. R. T. Yeh, Prentice-Hall (1978).
8. Elizabeth Fielding, "The Specification of Abstract Mappings and their Implementation as B+-Trees," *Technical Monograph PRG-18*, Oxford University Computing Laboratory, Programming Research Group (September 1980).
9. G. S. Carson, "A Formal Specification of the Programmer's Minimal Interface to Graphics," ANSI X3H34 Working Document (1982).