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

RAL-85-051

Formal Specification - A Comparison of Two Techniques

D A Duce and E V C Fielding

July 1985

Formal Specification - A Comparison of Two Techniques

D. A. Duce and E. V. C. Fielding

Rutherford Appleton Laboratory, Chilton, Didcot OXON OX11 OQX

1. Introduction

In previous papers [1, 2, 3] the applicability of a constructive specification technique to the problem of formally specifying graphics software has been investigated by the development of a formal specification of a small part of the Graphical Kernel System (GKS) [4, 5], the ISO graphics software standard. Other authors [6, 7, 8] have investigated the application of algebraic specification techniques to graphics software. This paper compares and contrasts these two approaches to specification by presenting specifications of the same example from GKS using both an algebraic technique and a constructive technique. An attempt is made to explain the insights that each technique provided.

The algebraic technique described is OBJ [9, 10, 11] and the constructive technique which is used is based on the Vienna Development Method (VDM) [12], The example chosen for the comparison is attribute handling in GKS. The next two sections describe the specification methods and the example which is tackled, after which the two specifications are presented and discussed.

2. Formal Specification

The purpose of a formal specification is to state **what** a system should to do without prescribing **how** it is to do it. A formal specification defines a system in an implementation independent way by describing its internal state in terms of **abstract data types** which are characterized only by the operations allowed over them.

The two main approaches to specifying abstract data types which this paper explores are: the **algebraic** approach, which specifies the operations of an abstract data type implicitly by relating them to each other using, for example, algebraic equations; and the **constructive** approach, which specifies the operations of an abstract data type explicitly in terms of some precise discipline such as set theory, for example. OBJ has been used as an example of an algebraic approach, and a subset of VDM has been used as a representative of the constructive approach. These two specification techniques are described and are then applied to the same example to reveal the strengths and differences in the two approaches.

2.1. The Algebraic Approach using OBJ

OBJ is an algebraic specification technique in which equations are used to define abstract data types. This enables abstract data types to be defined independently of any representation and OBJ specifications have an algebraic formal semantics based on equational logic. OBJ specifications also have an operational semantics based on interpreting the equations as rewrite rules, which makes it possible to validate OBJ specifications by testing and to use the specifications as prototypes to explore and to confirm the correctness of design decisions.

OBJ has powerful facilities for parameterization, modularity and data abstraction. However, the specification given here uses only a subset of OBJ for which an interpreter was available in the U.K. at the time of writing this paper [9]. The advantages gained by mechanized syntax- and type-checking were considered to be sufficiently great as to outweigh the disadvantages caused by restriction to this executable subset of OBJ. The subset excludes parameterization, mixfix operators, overloaded operators and operator attributes. Mixfix operators are defined using underbars

to indicate argument positions, for example, '**if** _ **then** _ **else** _'. This allows customized operator syntax with prefix, postfix, infix etc. orders as well as generally distributed fix operators with keywords and arguments in any desired order. Overloaded operators are distinct operators with the same syntax but with different argument types, for example, the operator '_+_' defined for both real and integer addition. The interpreter supports only prefix operators, however, for readability, the specification is presented using mixfix and overloaded operators. Operator attributes have only been used where necessary. They are described later in this section.

An OBJ specification has a modular structure with basic components called **objects**. An object is a module defining an abstract data type by introducing new sorts of data and operations on the data. These sorts (identifiers for carrier sets) and operations (symbols for functions) with their associated functionality, together constitute the **signature** of the abstract data type. The idea of an object corresponds to the class concept of Simula or the package concept in Ada, while the idea of a sort is similar to that of a type in Pascal (a named domain of values).

An object in OBJ can be defined as an enrichment of a previously defined sort. Objects in OBJ then form a dependency hierarchy which is an acyclic graph where one object is higher in the hierarchy than another if it imports the lower object. This ensures that everything that is used has been defined. At the lowest level are 'built-in' objects of the implementation, such as *Bool* and *Nat*.

These ideas will be illustrated and new terms that have been introduced will be defined by looking at the OBJ specifications of three abstract data types: *STATE*, *NDC_POINT* and *NDC_POINTS*, starting with the specification of the abstract data type *NDC_POINT*. In OBJ this is:

```
obj   NDC_POINT
sorts NDC_Point
ops   mk_ndc_point: R R → NDC_Point
      get_x_coordinate: NDC_Point → R
vars  x, y: R
eqns  ( get_x_coordinate(mk_ndc_point(x, y )) = x )
jbo
```

Each object is introduced by the keyword **obj**, followed by the name of the object, in this case *NDC_POINT*. The end of an object definition is denoted by **jbo**. An object may introduce a number of sorts; each sort (e.g. *NDC_Point*) is an identifier that denotes a set of values, called the **carrier set** of the sort. A carrier set is simply the domain of values of elements of the sort. Note that *NDC_POINT* is used to denote the object as a whole, including the carrier sets, signatures and operators of the object, while *NDC_Point* denotes only the carrier set of *NDC_POINT*. Objects may also introduce operators and their signatures, following the keyword **ops**. The **signature** of an operator declares its functionality by: a **form** which indicates the distribution of keywords and arguments (using underbars to indicate argument places for mixfix operators); an **arity** which lists argument (domain) sorts; and a **coarity** which is the sort of the range of the operator. The form and the arity of an operator are separated by ':', and the arity is separated from the coarity by the symbol '→'. The object *NDC_Point* introduces two operators: *mk_ndc_point*, which takes two arguments of sort **R** (real) and returns an object of sort *NDC_Point*; and *get_x_coordinate*, which takes an argument of sort *NDC_Point* and returns an object of sort **R**.

The definition of *NDC_POINT* illustrates how to define objects whose sorts are Cartesian products of component sorts, in this case the object is an ordered pair.

The operators of a data type can be classed into two groups: **generator operators** which produce objects of the sort of interest; and **selector** or **enquiry operators** which produce objects of other types. The smallest subset of the operators which allows all the values of the carrier of the sort to be reached (ie. can generate all the objects of the sort) is called the **constructors** or **primitive generators** of the sort.

The operator *mk_ndc_point* is the constructor for the sort *NDC_Point*, ie:

$\forall\ pt \in NDC\_Point \stackrel{=}{\exists} x, y \in \mathbf{R}\ .\ pt = mk\_ndc\_point(x, y)$

Most objects of interest have equations, defined after the keyword **eqns**. The equations define the relationships (identities) between expressions containing the operators, free variables (universally quantified), and conditional tests. The variables appearing in the equations are sort-constrained and must be declared after the keyword **vars** preceding the equation definitions.

The equations relate the other operators of the sort to the constructors. So the equation given in the definition of the object *NDC_POINT* expresses the selector operator *get_x_coordinate* in terms of an identity involving *mk_ndc_point*.

A more familiar form of this equation might be:

$\forall\ x, y \in \mathbf{R}\ .\ get\_x\_coordinate(mk\_ndc\_point(x, y)) = x$

Having defined the data type *NDC_POINT*, the definition of the data type *NDC_POINTS* can now be given. It uses the sort *NDC_Point*, so its definition illustrates the hierarchical structure of an OBJ specification.

```
obj   NDC_POINTS / NDC_POINT
sorts NDC_Points
ops   empty_ndc_points: → NDC_Points
      _ :: _ : NDC_Point NDC_Points → NDC_Points
jbo
```

The object *NDC_POINTS* refers to the object *NDC_POINT* (by the part of the definition: / *NDC_POINT*). Several previously defined objects may be imported in this way. Two operators are defined for the sort *NDC_Points*: *empty_NDC_Points* and '::', both of which are constructors. The operator *empty_ndc_points* defines a constant of the sort *NDC_Points* as it has no arity. The definition of the operator '::' is an example of the OBJ notation for defining mixfix operators in which arguments are distributed through the form, using the symbol '_' as a place marker to indicate argument positions. Both the number and order of place markers must agree with the number and order of domain sorts (sorts on the left hand side of the signature).

That the two operators defined are both constructors can be seen from the fact that the object *NDC_POINTS* has no equations. Two constructors are necessary because there is a distinguished value or constant in the carrier of the sort which can be reached only by the constructor *empty_ndc_points*. In other words:

$\stackrel{=}{\exists}\ empty\_ndc\_points \in NDC\_Points\ s.t.$
$\quad \forall\ pt \in NDC\_Point \wedge ndcpts \in NDC\_Points$
$\quad empty\_ndc\_points \neq pt :: ndcpts$

It is also true that:

$\forall\ ndcpts \in NDC\_Points\ .\ ndcpts \neq empty\_ndc\_points$
$\quad \stackrel{=}{\exists}\ n_1 \in NDC\_Points \wedge p \in NDC\_Point$
$\quad$ s.t. $ndcpts = p :: n_1$

The operator '::' defines how objects of sort *NDC_Points* are built up from objects of sort *NDC_Point* and is equivalent to an infix form of the *cons* list constructor. The definition of *NDC_POINTS* thus illustrates how to define objects which have a list structure in OBJ.

A data type *STATE* is now defined as follows.

```
obj   STATE/ NDC_POINTS
sorts State
ops   mk_state: NDC_Points R → State
      add_point: NDC_Point State → State
vars  pt: NDC_Point
      x, y: R
      ndc_pts: NDC_Points
eqns  ( add_point(pt , mk_state(ndc_pts, x_min)) =
                  mk_state(pt::ndc_pts, x_min) if (get_x_coordinate(pt) ≥ x_min) )
      ( add_point(pt , mk_state(ndc_pts, x_min)) =
                  mk_state(pt::ndc_pts, get_x_coordinate(pt)) if (get_x_coordinate(pt) < x_min) )
jbo
```

It is assumed that the ordering operators '≥' and '<' have been defined in the data type **R**.

The equations above illustrate the use of conditional expressions to limit the applicability of an equation. They define the effect of the operation *add_point* on *State*, which is to add a new point to the list of points in the first component of *State* and to update the second component so that it remains equal to the minimum of all the **x** coordinates of the points in the list. (Questions of initialization are ignored.)

OBJ has powerful pattern matching facilities which aid conciseness in defining operations by recursion equations.

It has already been mentioned that OBJ specifications have a well defined mathematical semantics. This is true of algebraic specifications in general, and in the case of OBJ these semantics are based on an initial algebra semantics. The formal semantics of OBJ specifications do not concern us in this paper. However the operational semantics which may be given to algebraic specifications through term rewriting is of interest and the way in which it operates is as follows. Each equation is treated as an ordered pair

< left hand side > → < right hand side >

and used as a rule for replacing instances of the left hand side of the pair by the right hand side. Computation of an expression then proceeds by repeatedly matching instances of left hand sides of rules and replacing them by their corresponding right hand sides until an expression is obtained in which no further matches are possible. The resulting expression is then said to be **reduced** or in **normal form** and is the meaning of the original expression.

The equation defining commutativity of an operator, for example:

$$(i + j = j + i)$$

if used directly as a rewrite rule causes the process of expression rewriting to become non-finite terminating, because a commutative equation may always be applied to the result of a previous application to yield the original expression. Partly for this reason, some properties of operators may be specified as attributes, for example, the operator '+' may be given the PERMUTING attribute:

_+_: *Nat Nat* → *Nat* (PERMUTING)

the effect of which is that the OBJ interpreter then remembers intermediate values arising in the evaluation of an expression involving '+' and will not apply a rule which would produce an expression already obtained, thus avoiding looping.

Other attributes permitted are ASSOC, COMM, and ID. ASSOC defines an operator to be associative, COMM defines it to be commutative and ID declares the identity element for an operator. Thus '+' could be written:

_ + _: *Nat Nat* → *Nat* (ASSOC COMM ID:0)

These attributes indicate to the expression rewriting system to take special action to avoid non-terminating sequences of rewrites and they also effectively introduce additional equations: the equations for associativity, commutativity and the identity element.

Examples of the use of attributes will be seen later in this paper.

As an example of the rewriting process, consider the equation in *NDC_Point*. Expressed as a rewrite rule this becomes:

get_x_coordinate(*mk_ndc_point*(*x*, *y*)) → *x*
where *x*, *y* ∈ **R**

Consider the expression:

mk_ndc_point(*get_x_coordinate*(*mk_ndc_point*(0.0, 0.1), 0.2)

Applying the rewrite rule above we obtain:

mk_ndc_point(0.0, 0.2)

which is the meaning of the expression as it contains only a constructor and so cannot be reduced further.

The constraints which must be satisfied for the application of rewrite rules to an expression to converge finitely are called the Church Rosser property (every terminating sequence of rewrites stops at a unique reduced form) and the termination property (every rewrite sequence terminates after a finite number of steps). In practice these properties are usually easy to satisfy. They imply that two expressions have the same meaning only if they reduce to the same normal form.

The OBJ interpreter allows specifications to be 'executed', in the sense that expressions may be evaluated by exhaustively using the equations as rewrite rules. Examples of the use of this means of testing or exploring the behaviour of an OBJ specification are shown later as are other features of OBJ.

2.2. The Constructive Approach using VDM

VDM is a constructive or model-based specification technique, which is denotational in approach. The mathematical foundations underlying denotational semantics are discussed in relation to VDM in [13] but are not the concern of this paper. The specifications given here use a limited subset of VDM for which a simplified notation can be described. It is possible to give an operational semantics to a subset of VDM. This has been done for a variant of VDM, called **me too**[14], which provides the operational semantics of LispKit for such constructive specifications.

A VDM specification has three components,

1) a model of the state;
2) invariants on the state;
3) operations over the abstract data type comprising the state.

The state definition describes the structure of the class of objects representing the state in terms of familiar basic types (natural number, real number, boolean etc.) and more complex data types built from these basic types, and constructors such as set, tuple, list and mapping. The constructed types are assumed to have implicitly defined with them constructor functions and other associated operators that are 'intuitively obvious', for example, **hd** and **tl** with **list**. These constructed types could be thought of as being already defined and available abstract data types, and could indeed be defined algebraically for completeness. So unlike an OBJ specification, a VDM specification does not define a multi-level hierarchy of abstract data types, with each abstract data type being defined in a uniform way and packaged into a separate module. Rather, in VDM, a single global definition is given of the abstract data type comprising the state, defined in terms of implicitly predefined abstract data types, and the hierarchy of types, although it exists, is not

explicitly brought out.

The invariants limit the objects within the class of objects representing the state to those that represent valid states. Invariants are thus constraints that must be preserved by the operations. Invariants are useful not only during the design process, but also during later modifications to the specification, for they represent properties of a design which must always hold.

The operations are defined implicitly by predicates which allows relations, and thus non-determinacy to be specified. However, the operations given in this paper do not require this generality, and in fact, reduce to functions, which allows the possibility of translation into **me too** with the benefits of 'executability'. The definitions of operations given here have the general form (or signature):

$$Inputs \times State \rightarrow State$$

There are two predicates defining each operation: a **pre-condition** and a **post-condition.** The former is a predicate over initial state and inputs and defines the conditions under which the operation produces valid results. The latter is a predicate over the initial state, inputs and final state, which defines the effect of the operation.

To illustrate these ideas, the data types *NDC_Points* and *NDC_Point* are considered as they would be defined and used in a VDM specification. Suppose that *NDC_Points* is the first component of some state *State*, which has a second component of type **R** (real). The state definition is as follows:

The state

$$State = NDC\_Points \times X\_Min$$

$$NDC\_Points = \textbf{list of } NDC\_Point$$
$$NDC\_Point = \mathbf{R} \times \mathbf{R}$$

$$X\_Min = \mathbf{R}$$

This defines an object of type *State* to consist of an ordered pair with first component of type *NDC_Points* and second component of type *X_Min*. Objects of type *NDC_Points* are modelled as a list of objects of type *NDC_Point*. The type *NDC_Point* is again an ordered pair and each of its components has the basic type **R**, which is also the type of *X_Min*. So the data type *NDC_Points* is constructed from the data type **R** using the constructors Cartesian product and **list of**, as is the data type *State*.

This state is intended to model a list of points and the minimum value of the x coordinates of the points in the list.

Invariant

$$\forall\ mk\_state(ndc\_points, x\_min) \in State$$
$$ndc\_points \neq empty\_list \Rightarrow$$
$$x\_min = min(\{\ get\_x\_coordinate(pt) \mid pt \in elems\ ndc\_points\})$$

The invariant states that for a non-empty list of points, the *x_min* component of the state is the minimum value of the x coordinates of the points in the list. The operator *elems* maps a list onto the set of elements contained in it.

The operation

```
let mk_state(ndc_points, x_min) = state in

add_point: NDC_Point × State→ State
add_point(state, pt, state') ≜
pre true
post ndc_points' = pt :: ndc_points ∧
     x_min' = if get_x_coordinate(pt) ≥ x_min then x_min
              else get_x_coordinate(pt)

get_x_coordinate: NDC_Point → R
get_x_coordinate(pt) ≜ let mk_ndc_point(x, y) = pt in x
```

The operation *add_point* is defined over objects of type *State*, and it adds a new object of type *NDC_Point* to the *NDC_Points* component of the state and ensures that the *X_Min* component reflects the minimum of all the x-coordinate values in the state.

The **let** clause preceding the operation definition holds over all subsequent operation definitions (in this case only one), and names the initial state and its components. *mk_state* is an example of a **constructor function** and it creates an object of type *State*. The convention that is used is that type names have capitalized initial letters and constructor function names are always, and instance names are usually, the lower case equivalents of their types (prefixed by *mk_* in the case of constructors). The names of the final state and its components which result from operations are obtained by decorating the initial state and component names with prime (′). This should be done by a second **let** clause of the same form which names and constructs the final state, but this has been omitted as its absence should not cause any confusion.

The first line of the operation definition is its signature and defines the types of the arguments and the result of the operation. The second line names the objects whose types are given in the signature. So the initial state, *state*, is of type *State*; the argument *pt* is of type *NDC_Point*; and the final state resulting from the operation, *state'* is also of type *State*.

The pre-condition being **true** implies that for all values of inputs and initial state the operation will produce a valid result. True pre-conditions are omitted in the specifications that follow.

The post-condition predicate describes the effect of the operation by relating the objects describing the initial and final states. It expresses the effect of the operation *add_point* as follows. The value of the *NDC_Points* component of the final state, *ndc_points'*, is obtained by adding the argument *pt* to the head of the list *ndc_points* by the use of the polymorphic list constructor '::'. This symbol denotes an infix form of the list operator *cons*, which is implicitly available for use with objects defined by the **list of** construct. The value of the *X_Min* component of the final state, *x_min'*, is expressed as the minimum of the result of applying an auxiliary function, *get_x_coordinate* to the argument *pt* and the initial value *x_min*. All the components of the state have been explicitly mentioned in the post-condition. By convention, any components of the state which are not mentioned in the post-condition of an operation are assumed to be unchanged by the operation.

The use of the projection function, *get_x_coordinate*, in the post-condition of the operation *add_point* illustrates how subsidiary functions are defined and used. The first line of its definition is its signature which shows that it is defined over the data type *NDC_Point* and returns a result of type **R**. The next line names the argument *pt* and defines *get_x_coordinate* to be a selector function which returns the first component of an object constructed by the constructor *mk_ndc_point*. In general, subsidiary functions will not just be operators for data types, but this example is interesting as it illustrates the difference in packaging between OBJ and VDM.

3. The Example

GKS provides a functional interface between an application program and a configuration of graphical input and output devices at a level of abstraction that hides the peculiarities of device hardware. It achieves device independence by means of the concepts of abstract input, abstract output and abstract workstations. The concept of abstract input will not be discussed, but the specifications attempt to capture the other two concepts.

Firstly, the concept of abstract output is described. In GKS, pictures are constructed from a number of basic building blocks, called **output primitives**, which are abstractions of the basic actions that a graphical output device can perform (eg. drawing a line). There are six output primitives in GKS: polyline, polymarker, text, fill area, cell array and generalized drawing primitive (GDP); each of which has associated with it a set of **parameters**, which defines a particular instance of the primitive. This paper considers the polyline primitive, which draws a connected sequence of line segments and has the coordinates of its vertices as parameters.

The concept of an abstract **workstation** in GKS is an abstraction from physical device hardware and maps abstract output primitives to physical output primitives and physical input primitives to abstract input primitives. It represents zero or one display surfaces and zero or more input devices as a configuration of abstract devices. An application program may direct output to more than one workstation simultaneously; however, the specification will aim to model only a system with a single workstation.

The application program specifies coordinate data in the parameters of an output primitive in **world coordinates** (WC), a Cartesian coordinate system. World coordinates are then transformed to a uniform coordinate system for all workstations, called **normalized device coordinates** (NDC) by a window to viewport mapping termed a **normalization transformation**. A second window to viewport mapping, called the **workstation transformation** accomplishes the transformation to the **device coordinates** (DC) of the display surface.

In order to simplify the specification, WC coordinate space will be ignored and it will be assumed that polyline coordinate data are supplied in NDC coordinates. It is also assumed that the single workstation transformation is fixed. Primitives can optionally be clipped to the boundary of the viewport of the normalization transformation, but clipping too, will be ignored.

An application often requires the capability of structuring a picture, such as the ability to define a graphical object, for example, a tree by a sequence of polyline primitives, and the ability to re-use this definition. GKS allows output primitives to be grouped together into units termed segments which are stored, conceptually at the NDC level, and which may be manipulated in certain ways as a single entity. Segments may not be nested.

So pictures are constructed from both primitives outside segments and segments. The specifications will attempt to capture this picture structure and the creation and storage of segments, but not any further segment manipulations.

The appearance of a primitive displayed on a workstation is determined by its parameters and additional data termed **aspects**. The aspects of a polyline are: linetype, which in GKS may be solid, dashed, dashed-dotted or an implementation-dependent type; linewidth scale factor, which is applied to the nominal linewidth provided by the workstation to give a value which is then mapped to the nearest available linewidth; and polyline colour index which is an index into a workstation dependent colour table. For simplicity, it will be assumed in the specifications that follow that the value for linewidth can be specified directly, rather than as the product of a scale factor and a nominal width. It is also assumed that the workstation supports any linetype and linewidth requested, as, although it is a simple matter to map the requested value onto the nearest available value, this adds needless complexity for the present purposes. Colour will also not be considered.

The values of aspects are determined by **attributes**. There are two basic schemes for specifying aspects, termed **individual specification** and **bundled specification**. In the individual scheme the value of each aspect is determined by a different attribute; the linetype aspect by the linetype attribute and the linewidth scale factor aspect by the linewidth scale factor attribute. For each of the

attributes there is an operation to set its value. In this scheme, the setting of the value of an attribute, such as linetype, applies to all subsequently created polyline primitives until it is reset. The individual scheme will not be further considered here.

In the bundled mode of specifying polyline aspects, the values of all the aspects are determined by a single attribute, called the **polyline index**. A polyline index defines a position in a table, the **polyline bundle table**, each entry in which is termed a **bundle** and specifies the values for each of the aspects. The bundle corresponding to a particular polyline index is termed the **representation** of the index. There is an operation which sets the value of polyline index modally, as well as an operation to set the representation of a bundle index. When a polyline primitive is created, the current value of the polyline index is bound to the primitive and cannot subsequently be changed. Bundles are bound to primitives when they are displayed. If a representation of a particular polyline index has not been defined, the representation for polyline index 1 is used instead. GKS is initialized such that there will always be a representation defined for polyline index 1.

In GKS each workstation has its own polyline bundle table, which allows the application to control the appearance of polylines with the same polyline index independently on each workstation on which they are displayed, using the capabilities of the workstation. If a representation of a polyline index is changed, the appearance of polylines already created with that polyline index may also be changed to the new representation. Thus although the value of the polyline index with which a polyline is created cannot subsequently be changed, the representation with which the polyline is displayed can be changed. GKS admits that some workstations are able to perform changes dynamically (for example a colour table can be dynamically changed on most raster devices affecting the appearances of all primitives displayed), whilst for other devices the picture has to be redrawn to effect the change (for example changing colour on a pen plotter). This act of redrawing the picture is termed **regeneration**. Regeneration is performed from segment storage since this is the only stored representation of the picture in GKS.

Associated with each workstation in GKS is a workstation description table which describes the capabilities of the workstation. Amongst other entries, it contains flags for each possible picture change that may require a regeneration, for example:

dynamic modification accepted for polyline bundle modification

If a flag has the value IMM then the corresponding change can be performed immediately by the workstation; if it has the value IRG an implicit regeneration is required. When a regeneration is signalled, another flag - **implicit regeneration mode** is consulted. This flag can be set by the application program and has two possible values: ALLOWED - the regeneration is performed immediately; SUPPRESSED - the regeneration is postponed until one of the GKS functions REDRAW ALL SEGMENTS ON WORKSTATION, UPDATE WORKSTATION, or CLOSE WORKSTATION is invoked.

In order to specify this simplified GKS system, the concepts that must be captured are:

1) the concept of a picture in NDC space in which polylines are created and where attributes are bound to them;
2) the concept of a segment store;
3) the concept of a workstation, which can be represented by specifications of the concepts of: a picture in DC space which displays polylines with their representations bound to them; a polyline bundle table; a bundle modification flag; and an implicit regeneration mode flag.

The GKS functions relevant to this simplified system are:

POLYLINE
generate a polyline defined by points in world coordinates;

SET POLYLINE INDEX
select a polyline index to be bound to subsequently created polylines;

CREATE SEGMENT
create the specified segment: subsequently created primitives are stored in this segment until it is closed;

CLOSE SEGMENT
close the open segment;

REDRAW ALL SEGMENTS ON WORKSTATION
redraw all visible segments stored on the specified workstation;

SET POLYLINE REPRESENTATION
define the representation of the specified polyline index on the specified workstation.

To model these functions directly introduces some complexity unnecessary for the present purposes, and hence the following operations, which are abstractions of the GKS functions are used instead. The following abstractions of GKS functions need to be specified:

add_polyline: *NDC_Points* × *GKS* → *GKS*
add_segment: *Segment* × *GKS* → *GKS*
redraw_all_segments: *GKS* → *GKS*
set_polyline_representation: *Polyline_Index* × *Linetype* × *Linewidth* × *GKS* → *GKS*

The operation *add_polyline* is equivalent to:

SET POLYLINE INDEX(· · ·)
POLYLINE(· · ·)

and *add_segment* to:

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

4. The Specifications

The OBJ specification is presented first, followed by the VDM specification. Some descriptive commentary and comparison is given with the presentations of the specifications and in the next section the merits of the two techniques are further discussed.

4.1. The OBJ Specification

This is presented in a 'bottom-up' order. Although it does not necessarily have to be developed this way; this order, where things have to be defined before they are used, is required by the OBJ interpreter, and is a logical way to describe the OBJ specification. Firstly, some fundamental objects are defined, on which the objects capturing the GKS concepts are based.

4.1.1. Fundamental Objects

OBJ has the built-in object *TRUTH* which defines the sort *BOOL* and (distinguished) constants *T* and *F* for true and false. The object *TRUTH* is implicitly accessible from any user defined object, however, the usual boolean operators are not built-in to OBJ and must be defined.

```
obj   BOOLEAN
ops   not _: BOOL → BOOL
      _ and _, _ or _: BOOL BOOL → BOOL
vars  b: BOOL
eqns  ( not T = F )
      ( not F = T )
      ( T or b = T )
      ( b or T = T )
      ( F or b = b )
      ( b or F = b )
      ( b or b = b )
      ( T and b = b )
      ( b and T = b )
      ( F and b = F )
      ( b and F = F )
      ( b and b = b )
jbo
```

As there is no built-in implementation of the sort *Real*, the built-in sort *Nat* is used, without too much loss of generality, in places where **R** is used in the VDM specification as is required by GKS.

The next objects to be defined are the polylines in NDC and DC space which are used to specify the GKS function *polyline*. These objects are *NDC_POLYLINE* and *DC_POLYLINE*. NDC polylines are represented by the list of vertices (of sort *NDC_Points*) of the polyline (in NDC coordinates), and the polyline index (of sort *Nat*) with which the polyline is created. The definition of *NDC_POLYLINE* uses the objects *NDC_POINTS* defined earlier. The definition is not repeated here.

```
obj   NDC_POLYLINE | NDC_POINTS
sorts NDC_Polyline
ops   mk_ndc_polyline: NDC_Points Nat → NDC_Polyline
jbo
```

Next the object *DC_POLYLINE* is defined. The sort *DC_Polyline* represents polylines as displayed on the workstation display surface. Objects of this sort are tuples; the first field is the list of vertices of the polyline, the second the polyline index with which the polyline was created and the third the bundle, (linetype, linewidth) pair, with which it is displayed. *DC_POLYLINE* uses the objects *BUNDLE* and *DC_POINTS*. The latter defines lists of points in DC coordinates, and mirrors the object *NDC_POINTS*.

The object *BUNDLE* defines bundles. Bundles are represented by Cartesian products: pairs, whose first component represents linetype and second represents linewidth. Linetype and linewidth are both represented by natural numbers.

The object *TRANSFORM* defines a function *t* which converts an object of type *NDC_Points* to an object of type *DC_Points* by transforming each of the component objects of type *NDC_Point* to a corresponding objects of type *DC_Point*. As the details of how the actual transformation is accomplished are not of interest, they are not given here.

```
obj   DC_POINT
sorts DC_Point
ops   mk_dc_point: Nat Nat → DC_Point
jbo

obj   DC_POINTS | DC_POINT
sorts DC_Points
ops   empty_dc_points: → DC_Points
      _ :: _: DC_Point DC_Points → DC_Points
jbo

obj   TRANSFORM | NDC_POINTS DC_POINT
ops   t: NDC_Points → DC_Points
jbo

obj   BUNDLE
sorts Bundle
ops   mk_bundle: Nat Nat → Bundle
jbo

obj   DC_POLYLINE | DC_POINTS BUNDLE
sorts DC_Polyline
ops   mk_dc_polyline: DC_Points Nat Bundle → DC_Polyline
jbo
```

4.1.2. Objects Capturing GKS Concepts

At this point, all the objects have been defined which are necessary to enable the GKS concepts of a segment, segment storage, the NDC and DC pictures, as well as a polyline bundle table, to be defined.

Firstly the objects *SEGMENT* and *SEGMENT_STORE* are specified.

```
obj   SEGMENT | NDC_POLYLINE
sorts Segment
ops   empty_segment: → Segment
      _ :: _: NDC_Polyline Segment → Segment
jbo

obj   SEGMENT_STORE | SEGMENT
sorts Segment_Store
ops   empty_ss: → Segment_Store
      _ :: _: Segment Segment_Store → Segment_Store
jbo
```

The object *SEGMENT* uses the object *NDC_POLYLINE* as segments are comprised of NDC polylines. The definitions of *SEGMENT* and *SEGMENT_STORE* resemble the definitions of both *NDC_POINTS* and *DC_POINTS* as they are all list-structured objects and all define the overloaded operator '::'. (If the parameterization facilities of OBJ were being exploited, a single object could be given which defines the list operators and is parameterized on the type of objects comprising the list. This object could then be instantiated with these four types giving a more concise description.)

Now the concepts of a NDC picture and a DC picture can be specified by the objects *NDC_PICTURE* and *DC_PICTURE*. Again, these two objects have a list structure, but they

have richer operators than those seen previously.

```
obj   NDC_PICTURE / NDC_POLYLINE SEGMENT
sorts NDC_Picture
ops   empty_ndc_picture: → NDC_Picture
      _ :: _: NDC_Polyline NDC_Picture → NDC_Picture
      _ :: _: Segment NDC_Picture → NDC_Picture
jbo

obj   DC_PICTURE / DC_POLYLINE
sorts DC_Picture
ops   empty_dc_picture: → DC_Picture
      _ :: _: DC_Polyline DC_Picture → DC_Picture
      _ || _: DC_Picture DC_Picture → DC_Picture
vars  dc_pic1, dc_pic2: DC_Picture
      dc_pl: DC_Polyline
eqns  ( empty_dc_picture || dc_pic2 = dc_pic2 )
      ( (dc_pl :: dc_pic1) || dc_pic2 = dc_pl :: (dc_pic1 || dc_pic2) )
jbo
```

As can be seen from the two definitions of the polymorphic operator '::' in the object *NDC_PICTURE*, a picture in NDC space is comprised of both NDC polylines and segments. The object *DC_Picture* consists of DC polylines only and calls for some comment. The operator '||' combines DC pictures and the equations linking this operation to the constructors *empty_dc_picture* and '::' illustrate the use and power of pattern matching in OBJ. Patterns are used in the left hand sides of the equations to select the cases in which they are applicable. The equations given here are equivalent to:

dc_pic1 || *dc_pic2* = **if** *dc_pic1* = *empty_dc_picture* **then** *dc_pic2*
else hd *dc_pic1* :: (**tl** *dc_pic1* || *dc_pic2*)

where **hd** and **tl** are the usual head and tail operators for lists.

The next part of the specification characterizes a polyline bundle table. Before defining the object *POLYLINE_BUNDLE_TABLE*, we need to define sets of natural numbers.

```
obj   SET_OF_NAT / BOOLEAN
sorts Set_Of_Nat
ops   Ø: → Set_Of_Nat
      {_}: Nat → Set_Of_Nat
      _ ∪ _: Set_Of_Nat Set_Of_Nat → Set_Of_Nat (ASSOC COMM ID: Ø)
      _∈_: Nat Set_Of_Nat → BOOL
vars  i, j : Nat
      s: Set_Of_Nat
eqns  ( s ∪ s = s )
      ( i ∈ Ø = F )
      ( i ∈ {j} = (i == j) )
      ( i ∈ (s1 ∪ s2) = (i ∈ s1) or (i ∈ s2) )
jbo
```

The attributes of '∪' define the operator to be associative:

$$(s_1 \cup s_2) \cup s_3 = s_1 \cup (s_2 \cup s_3)$$

commutative:

$$s_1 \cup s_2 = s_2 \cup s_1$$

and have identity $\emptyset$ (the empty set):

$$\emptyset \cup s = s \cup \emptyset = s$$

It takes a little time to get accustomed to these definitions. It is worth noting that the set:

$$\{a, b, c, d\}$$

is represented by the following expression in the algebra:

$$\{a\} \cup \{b\} \cup \{c\} \cup \{d\}$$

The first equation in *SET_OF_NAT* removes duplicate elements when forming the cup of two sets. The next three equations define the membership relation, the second of which defines membership of a singleton set to be element equality, ie:

$$i \in \{j\} \text{ if } (i == j)$$

The commutativity and associativity of '$\cup$' ensures that the sets:

$$\{a,b,c,d\} \text{ and } \{b,a,c,d\}$$

are equal, because:

$$\{a\} \cup \{b\} \cup \{c\} \cup \{d\} = \{b\} \cup \{a\} \cup \{c\} \cup \{d\}$$

As mentioned earlier, the OBJ interpreter which was available does not currently support attributes. To make the specification given above executable, the operator '$\cup$' was declared without attributes and the first equation was removed. This effectively defines a list of *Nat* rather than a set of *Nat*, but the difference does not affect the integrity of the overall specification.

The object *POLYLINE_BUNDLE_TABLE* can now be defined.

```
obj   POLYLINE_BUNDLE_TABLE | BUNDLE SET_OF_NAT BOOLEAN
sorts Polyline_Bundle_Table
ops   empty_pbt: → Polyline_Bundle_Table
      [_ → _]: Nat Bundle → Polyline_Bundle_Table
      _ + _: Polyline_Bundle_Table Polyline_Bundle_Table → Polyline_Bundle_Table
            (PERMUTING)
      dom _: Polyline_Bundle_Table → Set_Of_Nat
      bundle: Polyline_Bundle_Table Nat → Bundle
vars  i, j: Nat
      b1, b2: Bundle
      pbt: Polyline_Bundle_Table
      s: Set_Of_Nat
eqns  ( (pbt + [i→b1]) + [i→b2] = pbt + [i→b2] )
      ( (pbt + [i→b1]) + [j→b2] = (pbt + [j→b2]) + [i→b1] if not (i==j) )
      ( dom empty_pbt = ∅ )
      ( dom (pbt + [i→b1])= {i} ∪ (dom pbt) )
      ( bundle(pbt + [i→b1], i) = b1 )
      ( bundle(pbt + [j→b1], i) = bundle(pbt, i) if (not(i==j) and i ∈ (dom pbt)) )
      ( bundle(pbt, i) = bundle(pbt, 1) if not(i ∈ (dom pbt)) )
jbo
```

These definitions call for some comment. The operators '[_→_]' and '_ + _' in combination are used to add new entries to a polyline bundle table. The first two equations state that if an entry for the polyline index already exists in the table, its representation will be replaced with the new representation. The second equation also ensures that the sort behaves well with respect to equality, so that the polyline bundle tables represented by the expressions:

$$(empty\_pbt + [1 \rightarrow b\,1]) + [2 \rightarrow b\,2]$$

and

$$(empty\_pbt + [2 \rightarrow b\,2]) + [1 \rightarrow b\,1]$$

are equal. The equations can be thought of as defining a finite function.

The operator '*dom*' returns the set of indices defined in a table.

The operator '*bundle*' provides the representation of a specified polyline index, delivering the representation of polyline index 1 if the table does not contain a representation for the actual index specified. This is the default mechanism that GKS requires. As noted earlier, GKS is initialized so that a representation for polyline index 1 will be defined, though initialization is not actually specified here.

It is instructive to digress slightly at this point. The specification for *POLYLINE_BUNDLE_TABLE* just given cannot be executed on the OBJ interpreter available to us, because without the PERMUTING attribute, the second equation causes the interpreter to loop (it is similar to a commutator equation). To get round this problem an alternative a definition was used, given below.

$$((pbt + [i \rightarrow b1]) + [i \rightarrow b2] = pbt + [i \rightarrow b2])$$
$$((pbt + [i \rightarrow b1]) + [j \rightarrow b2] = (pbt + [j \rightarrow b2]) + [i \rightarrow b1] \text{ if } (not(i == j) \text{ and } j \in (dom\ pbt)))$$

The second equation is a restricted form of the earlier equation. The restriction $j \in (dom\ pbt)$ ensures that infinite rewrite sequences cannot arise. To see this, consider:

$$((pbt + [1 \rightarrow b1]) + [2 \rightarrow b2]) + [1 \rightarrow b3]$$

Applying this equation yields:

$$((pbt + [1 \rightarrow b1]) + [1 \rightarrow b3]) + [2 \rightarrow b2]$$

The equation cannot now be applied to give the original expression back because:

$$2 \notin dom\ ((pbt + [1 \rightarrow b\,1]) + [1 \rightarrow b\,3])$$

The restriction, then, ensures that rewriting makes progress. The price paid for this is that equality is lost, in so far as:

$$(empty\_pbt + [1 \rightarrow b\,1]) + [2 \rightarrow b\,2]$$

and

$$(empty\_pbt + [2 \rightarrow b\,2]) + [1 \rightarrow b\,1]$$

do not represent the same object, although they ought to. However, since equality over this sort is not used in the specification, this does not pose any problems.

These definitions again illustrate the power of pattern matching, though it does take some practice to read and write such equations easily.

The next object, *TO_DC*, describes the operator *to_dc* which given a segment delivers the corresponding sequence of DC polylines, in the same order as the NDC polylines in the segment.

```
obj   TO_DC | SEGMENT POLYLINE_BUNDLE_TABLE DC_PICTURE TRANSFORM
ops   to_dc: Segment Polyline_Bundle_Table → DC_Picture
vars  ndc_pl: NDC_Polyline
      pts: NDC_Points
      pbt: Polyline_Bundle_Table
      dcp: DC_Picture
      index: Nat
      s: Segment
eqns  ( to_dc(empty_segment, pbt) = empty_dc_picture )
      ( to_dc(mk_ndc_polyline(pts, index) :: s, pbt) =
            mk_dc_polyline(t(pts), index, bundle(pbt, index)) :: to_dc(s, pbt)) )
jbo
```

Again, a recursive definition is given.

Lastly, two objects define the sorts: *Bundle_Modification_Flag* and *Implicit_Regeneration*. Each has two values which are distinguished.

```
obj   BUNDLE_MODIFICATION_FLAG
sorts Bundle_Modification_Flag
ops   IRG, IMM: → Bundle_Modification_Flag
jbo

obj   IMPLICIT_REGENERATION
sorts Implicit_Regeneration
ops   ALLOWED, SUPPRESSED: → Implicit_Regeneration
jbo
```

4.1.3. The Top Level - GKS and its Operators

Finally, the object *GKS* is defined. It defines the operations, *add_polyline*, *add_segment*, *redraw_all_segments* and *set_polyline_representation* on the sort *GKS*. The constructor of sort *GKS* is *mk_gks*, and it constructs elements of this sort from the sorts *NDC_Picture*, *DC_Picture*, *Segment_Store*, *Polyline_Bundle_Table*, *Bundle_Modification_Flag* and *Implicit_Regeneration*. *GKS* embodies the concepts identified as characterizing the simplified GKS system described in section 3.

```
obj   GKS | NDC_PICTURE DC_PICTURE SEGMENT_STORE POLYLINE_BUNDLE_TABLE TO_DC
          BUNDLE_MODIFICATION_FLAG IMPLICIT_REGENERATION
sorts GKS
ops   mk_gks: NDC_Picture DC_Picture Segment_Store Polyline_Bundle_Table
       Bundle_Modification_Flag Implicit_Regeneration → GKS
      add_polyline: NDC_Points Nat GKS → GKS
      add_segment: Segment GKS → GKS
      redraw_all_segments: GKS → GKS
      recreate: DC_Picture Polyline_Bundle_Table → DC_Picture
      regenerate: Segment_Store Polyline_Bundle_Table → DC_Picture
      set_polyline_representation: Nat Nat Nat GKS → GKS
vars  ndcp: NDC_Picture
      dcp: DC_Picture
      ndc_pts: NDC_Points
      dc_pts: DC_Points
      ss: Segment_Store
      bmf: Bundle_Modification_Flag
      ir: Implicit_Regeneration
      ndc_pl: NDC_Polyline
      s: Segment
      pbt: Polyline_Bundle_Table
      i, pi, lt, lw: Nat
      b: Bundle
eqns  ( add_polyline(ndc_pts, pi, mk_gks(ndcp, dcp, ss, pbt, bmf, ir)) =
            mk_gks(mk_ndc_polyline(ndc_pts, pi) :: ndcp,
            mk_dc_polyline(t(ndc_pts), pi, bundle(pbt, pi)) :: dcp, ss, pbt, bmf, ir) )
      ( add_segment(s, mk_gks(ndcp, dcp, ss, pbt, bmf, ir)) =
            mk_gks(s :: ndcp, to_dc(s, pbt) :: dcp, s :: ss, pbt, bmf, ir) )
      ( redraw_all_segments(mk_gks(ndcp, dcp, ss, pbt, bmf, ir)) =
            mk_gks(ndcp, regenerate(ss, pbt), ss, pbt, bmf, ir) )
      ( recreate(empty_dc_picture, pbt) = empty_dc_picture )
      ( recreate(mk_dc_polyline(dc_pts, pi, b) :: dcp, pbt) =
            mk_dc_polyline(dc_pts, pi, bundle(pbt, pi)) :: recreate(dcp, pbt) )
      ( regenerate(empty_ss,pbt) = empty_dc_picture )
      ( regenerate(s :: ss, pbt) = to_dc(s, pbt) || regenerate(ss, pbt) )
      ( set_polyline_representation(i, lt, lw, mk_gks(ndcp, dcp, ss, pbt, bmf, ir)) =
            mk_gks(ndcp, recreate(dcp, pbt + [i→mk_bundle(lt, lw)]), ss, pbt + [i→mk_bundle(lt, lw)], bmf, ir)
                  if bmf == IMM )
      ( set_polyline_representation(i, lt, lw, mk_gks(ndcp, dcp, ss, pbt, bmf, ir)) =
            mk_gks(ndcp, regenerate(ss, pbt + [i→mk_bundle(lt, lw)]), ss,
                        pbt + [i→mk_bundle(lt, lw)], bmf, ir)
                  if (bmf == IRG) and (ir == ALLOWED) )
      ( set_polyline_representation(i, lt, lw, mk_gks(ndcp, dcp, ss, pbt, bmf, ir)) =
            mk_gks(ndcp, dcp, ss, pbt + [i→mk_bundle(lt, lw)], bmf, ir)
                  if (bmf == IRG) and (ir == SUPPRESSED) )
jbo
```

It is worth looking at the *add_polyline* operator. The equation defining the operator describes its effect on a state represented by the object:

mk_gks(ndcp, dcp, ss, pbt, bmf, ir)

The effect of *add_polyline* is to create a new NDC polyline in the NDC picture and display the corresponding DC polyline in the DC picture. The result is thus modelled by an object of sort

GKS, whose NDC picture component is that of the initial state with the addition of the new NDC polyline, and whose DC picture component is that of the initial state with the addition of the new DC polyline. The other components of the object are unchanged. This is what the first equation above states.

The *add_segment* operation is similar, but also adds the segment, supplied as argument, to the segment store. The operator *to_dc* is used to generate the DC polylines corresponding to the NDC polylines in the segment. The operation *redraw_all_segments* generates a new DC picture from the segment store, *recreate* redisplays a DC picture with new representations for the polyline indices used in its creation. The operators *regenerate* and *recreate* are really hidden operators within the data type *GKS*. The modularization of the specification is not ideal, but is adequate for this presentation. An alternative modularization might define '*recreate*' in the object *DC_PICTURE* and '*regenerate*' in the object *SEGMENT_STORE*.

The operation *set_polyline_representation* adds the representation for the specified index to the bundle table, and depending on the values of the flags bundle modification flag and implicit regeneration mode, may also cause the DC picture to be changed.

That concludes the algebraic specification, and is a good point from which to consider the corresponding VDM specification. The VDM specification consists of a definition of the GKS state, followed by the operations on this state, and this corresponds closely with the top level of the OBJ specification: the object *GKS*.

4.2. The VDM Specification

The VDM specification and the OBJ specification mirror each other. The VDM state definition specifies the following GKS components:

NDC picture
DC picture
segment store
polyline bundle table
and bundle modification and implicit regeneration flags,

and these definitions may be contrasted with the corresponding OBJ object specifications. Firstly, the VDM GKS state definition is given and then the operations over this state are defined.

4.2.1. The state

GKS = *NDC_Picture* × *DC_Picture* × *Segment_Store* × *Polyline_Bundle_Table*
× *Bundle_Modification_Flag* × *Implicit_Regeneration*

NDC_Picture = **list of** *Component*
Component = *NDC_Polyline* | *Segment*
Segment = **list of** *NDC_Polyline*

NDC_Polyline = *NDC_Points* × *Polyline_Index*
NDC_Points = **list of** *NDC_Point*
NDC_Point = **R** × **R**

Polyline_Index = **N**

Segment_Store = **list of** *Segment*

DC_Picture = **list of** *DC_Polyline*
DC_Polyline = *DC_Points* × *Polyline_Index* × *Bundle*

DC_Points = **list of** *DC_Point*
DC_Point = **R** × **R**

Bundle = *Linetype* × *Linewidth*
Linetype = **N**
Linewidth = **R**

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

Bundle_Modification_Flag = {*IRG, IMM*}
Implicit_Regeneration = {*ALLOWED, SUPPRESSED*}

The state *GKS* consists of the Cartesian product of components of type *NDC_Picture*, *DC_Picture*, *Polyine_Bundle_Table*, *Bundle_Modification_Flag* and *Implicit_Regeneration*. This compares with the OBJ object *GKS*, which is constructed from sorts of the same names, and this shows how Cartesian product, or its equivalent is expressed in the two notations. The NDC picture is modelled as a list of *Component*, which may be either of type *NDC_Polyine* or of type *Segment*. The OBJ specification of *NDC_PICTURE* expressed this by defining two operators for constructing elements of the sort *NDC_Picture* - one operator using the sort *NDC_Polyline* and the other the sort *Segment*.

Segments and segment store are modelled as a list of NDC polylines and a list of segments, respectively, which again corresponds to the OBJ specifications of these objects, as does the characterization of *NDC_Polyline* by a list of its vertices and a polyline index.

The type *DC_Picture* is modelled as a list of objects of type *DC_Polyline*, which in turn consists of three components: a list of vertices (the points in DC space obtained by transforming the vertices of the corresponding NDC polyline); the polyline index with which the polyline was created; and the bundle (linetype, linewidth pair) with which it is displayed. This again follows the OBJ definitions closely.

The polyline bundle table definition illustrates how a mapping, or finite function, is expressed in VDM, as compared with the OBJ definition. A mapping is not defined by an expression, but is constructed by pairing domain and range elements explicitly. The definition states the types of the domain and range of the mapping.

The comparison of the definitions of the two flags is straightforward.

4.2.2. The invariants

The state is constrained by the following invariants.

let *mk_gks(ndcp, dcp, ss, pbt, bmf, ir)* = *gks* **in**

Invariant (1)

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

$ss = ndcp \lceil \{c \mid c \in elems\ ndcp \wedge Segment(c)\}$

Invariant (2)

All point-list (after transformation back to NDC coordinates) and polyline index pairs which are in the DC picture are also in the NDC picture:

$map\_to\_NDC(dcp)\ is\_sublist\_of\ flatten(ndcp)$

Invariant (3)

All point-list (after transformation) and polyline index pairs in segment store are also contained in the DC picture:

flatten(*ss*) *is_sublist_of map_to_NDC*(*dcp*)

The auxiliary functions used in these definitions are not further defined here. Full definitions are given in [1]. The intentions of the invariants should be clear.

4.2.3. The operations

The VDM operation definitions correspond closely with the equations given in the object *GKS* of the OBJ specification. For example, the operation *add_polyline* creates a new polyline in the NDC picture and displays it by adding it to the DC picture with a representation taken from the polyline bundle table. From both the VDM operation definition and the equation for *add_polyline* in the OBJ specification, it can be seen that for every polyline added to the NDC picture a corresponding polyline is also added to the DC picture.

The other operations, *add_segment* and *redraw_all_segments*, mirror corresponding OBJ equations and do not require more detailed comparison. The operation *set_polyline_representation* in the VDM specification splits into three separate equations in the OBJ specification and requires the use of conditional equations. The definition of a mapping and the notation for updating it, which are illustrated in this operation and the definition of the type *Polyline_Bundle_Table*, are considerably more succinct in VDM than that of the corresponding OBJ specification for *Polyline_Bundle_Table* and its operators.

There are a couple of further interesting points to note. As everything must be encapsulated in an object in OBJ, it was necessary to package the functions *t* and *to_dc* in objects: *TRANSFORM* and *TO_DC* respectively. They need not necessarily have been made separate objects and might have been included in other objects that had been defined. The idea of defining an object which defines only a function is interesting and something to get accustomed to. Also of interest is the way in which the recursive function definitions of *regenerate* and *recreate* become operators of the sort *GKS* and are expressed using pattern-matching in OBJ.

```
let mk_gks(ndcp, dcp, ss, pbt, bmf, ir) = gks in

add_polyline: NDC_Points × Polyline_Index × GKS → GKS
add_polyline(pts, pi, gks, gks') ≜
post ndcp' = mk_ndc_polyline(pts, pi) :: ndcp ∧
     dcp' = mk_dc_polyline(t(pts), pi, bundle(pbt, pi)) :: dcp

bundle: Polyline_Bundle_Table × Polyline_Index → Bundle
bundle(pbt, pi) ≜ if pi ∈ dom pbt then pbt(pi)
                  else pbt(1)

t: NDC_Points → DC_Points
add_segment: Segment × GKS→ GKS
add_segment(s, gks, gks') ≜
post ndcp' = s :: ndcp ∧
     ss' = s :: ss ∧
     dcp' = to_dc(s, pbt) || dcp
```

to_dc: *Segment* × *Polyline_Bundle_Table* → *DC_Picture*
to_dc(*s*, *pbt*) ≜ **if** *s* = < > **then** < >
else let *mk_ndc_polyline*(*pts*, *pi*) = **hd** *s* **in**
mk_dc_polyline(*t*(*pts*), *pi*, *bundle*(*pbt*, *pi*)) :: *to_dc*(**tl** *s*, *pbt*)

redraw_all_segments: *GKS* → *GKS*
redraw_all_segments(*gks*, *gks'*) ≜
post *dcp'* = *regenerate*(*ss*, *pbt*)

regenerate: *Segment_Store* × *Polyline_Bundle_Table* → *DC_Picture*
regenerate(*ss*, *pbt*) ≜ **if** *ss* = < > **then** < >
else *to_dc*(**hd** *ss*, *pbt*) || *regenerate*(**tl** *ss*, *pbt*)

set_polyline_representation: *Polyline_Index* × *Linetype* × *Linewidth* × *GKS* → *GKS*
set_polyline_representation(*pi*, *lt*, *lw*, *gks*, *gks'*) ≜
post *pbt'* = *pbt* + [*pi* → *mk_bundle*(*lt*, *lw*)] ∧
(*bmf* = *IMM* ⟹ *dcp'* = *recreate*(*dcp*, *pbt'*))
∧
(*bmf* = *IRG* ∧ *ir* = *ALLOWED* ⟹ *dcp'* = *regenerate*(*ss*, *pbt'*))
∧
(*bmf* = *IRG* ∧ *ir* = *SUPPRESSED* ⟹ *dcp'* = *dcp*)

recreate: *DC_Picture* × *Polyline_Bundle_Table* → *DC_Picture*
recreate(*dcp*, *pbt*) ≜ **if** *dcp* = < > **then** < >
else let *mk_dc_polyline*(*dpts*, *b* ,*pi*) = **hd** *dcp* **in**
mk_dc_polyline(*dpts*, *pi*, *bundle*(*pbt*, *pi*)) :: *recreate*(**tl** *dcp*, *pbt*)

5. Executable Specifications

As has been mentioned, the OBJ specification given here may be executed by using the data type equations as rewrite rules. This gives a useful check on the behaviour of the system specified. Execution was used to explore the specification and this revealed syntactic errors and also increased confidence levels that the specification did behave as intended and expected.

In order to illustrate the kind of test that was carried, an example follows. Suppose we define:

gks_0 = *mk_gks*(*empty_ndc_picture*, *empty_dc_picture*, *empty_ss*, *empty_pbt*, *IRG*, *ALLOWED*)

Then we may ask what is the effect of:

set_polyline_representation(1, 1, 1, *set_polyline_representation*(2, 2, 2, gks_0))

By applying the equations describing *set_polyline_representation* we obtain:

mk_gks(*empty_ndc_picture*, *empty_dc_picture*, *empty_ss*,
((*empty_pbt* + [2 → *mk_bundle*(2, 2)]) + [1 → *mk_bundle*(1, 1)]), *IRG*, *ALLOWED*)

This is the term representing the state after both operations have been performed. A variety of tests of this nature were performed and produced expected results.

This same example has also been formulated in **me too**, the method for obtaining executable constructive specifications for rapid prototyping. This, as well as details of the tests that were carried out, is described in [15] The same tests have also been executed on the specification described here using the OBJ interpreter.

6. Discussion and Conclusions

The most obvious disparity in the two specifications is the difference in their lengths. However, one of the reasons for this is that VDM relies heavily on intuitively obvious operators being implicitly available for data types, whereas in OBJ all operators used have to be defined. OBJ has far more powerful data structuring facilities than VDM, with its modularity and the packaging of data types with their operators, which ensures that everything has been defined and leaves no room for misconceptions due to differing interpretations. So there are tradeoffs between brevity and complete definition.

It is worth noting that parameterization of the OBJ specification would have reduced its length considerably. The specification would also have been shortened if the operators implicitly available in VDM were assumed to have been defined in a library of parameterized data types. In practice the provision of such libraries would be the normal approach in any large specification project.

An algebraic specification in OBJ closely resembles a functional program. As the VDM specification was written largely using recursive functions, it is not surprising that the one specification can fairly readily be directly translated into the other. These similarities are made more apparent by the use of mixfix, overloaded operators. VDM does also allow the definition of relations rather than functions in the post-conditions of operations, as well as more implicit and less constructive operation definitions, and were this exploited, a closely corresponding OBJ specification may not be derivable.

Reading and writing algebraic specifications initially took more practice than reading and writing a constructive specification. This could be because of greater previous exposure to constructive approaches. However, the effort of writing the OBJ specification was effort well spent, for it forced us to consider the structure of the problem more thoroughly and this in turn led us to see how the specification could be built incrementally. The module structure of OBJ is a very useful structuring tool, which constructive notations seem to lack (however Z is addressing these problems [16]).

The benefits of being able to execute the OBJ specification were considerable. In particular, the availability of a type checker for OBJ was invaluable. Most of the errors in the first version of the constructive specification were type errors which a type checker would have found.

The two approaches each lend their own insights to a problem. VDM encourages a more 'top-down' approach to viewing a problem, while OBJ may be used in a more 'bottom-up' style which gives fresh ideas on how to partition the problem and how to structure the specification. The overall experience was that the two methods complemented each other.

7. Acknowledgements

We are grateful to Derek Coleman and Robin Gallimore for introducing us to OBJ and for suggesting several clarifications and improvements to this paper; and to Peter Henderson for pointing out how exploiting the power of pattern matching in OBJ allows more concise formulation of equations.

References

1. D. A. Duce, E. V. C. Fielding, and L. S. Marshall, "Formal Specification and Graphics Software," RAL-84-068, Rutherford Appleton Laboratory, Chilton, Didcot, OXON OX11 0QX, U.K. (1984).
2. D. A. Duce and E. V. C. Fielding, "Better Understanding through Formal Specification," RAL-84-128, Rutherford Appleton Laboratory, Chilton, Didcot, OXON OX11 0QX, U.K. (1984).
3. D. A. Duce and E. V. C. Fielding, "Formal Specification - A Simple Example," *ICL Technical Journal*, to appear (1985).
4. *Graphical Kernel System (GKS) 7.2 Functional Description, ISO/DIS 7942,* Information

Processing (4 November 1982).

5. F. R. A. Hopgood, D. A. Duce, J. R. Gallop, and D. C. Sutcliffe, *Introduction to the Graphical Kernel System (GKS)*, Academic Press (1983).
6. R. Gnatz, "An Algebraic Approach to the Standardization and the Certification of Graphics Software," *Computer Graphics Forum* 2(2/3) (1983).
7. G. S. Carson, "The Specification of Computer Graphics Systems," *IEEE Computer Graphics and Applications*, pp. 27-41 (September 1983).
8. W. R. Mallgren, "Formal Specification of Graphic Data Types," *ACM Transactions on Programming Languages and Systems* 4(4), pp. 687-710 (October 1982).
9. D. Coleman and R. M. Gallimore, "Software Engineering Using Executable Specifications," Dept. of Computation, UMIST (1984).
10. K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of OBJ2," *Proceedings of the 1985 Symposium on Principles of Programming Languages* (1985).
11. J. Goguen and J. Meseguer, "Rapid Prototyping in the OBJ Executable Specification Language," *ACM Sigsoft Software Engineering Notes* 7(5), p. 75 (1982).
12. C. B. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall, Englewood Cliffs, NJ (1980).
13. D. Bjorner and C. B. Jones, *Formal Specification and Software Development*, Prentice-Hall, Englewood Cliffs, NJ (1982).
14. P. Henderson, "me too - a language for software specification and model building - preliminary report," Computing Science FPN-9, University of Stirling (1984).
15. C. Minkowitz, "Specification to Prototype - A comparison of two formal methods of software design," Department of Computer Science, University of Stirling, Scotland (1984).
16. B. Sufrin, "Towards a formal specification of the ICL Data Dictionary," *ICL Technical Journal*, pp. 195-217 (November 1984).